VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
STUDIJŲ PROGRAMA: INFORMATIKA

Skatinamasis mokymasis su žmogaus grįžtamuoju ryšiu

Reinforcement Learning from Human Feedback

Baigiamasis bakalauro darbas

Atliko: Dalius Budrys
VU el.p.: dalius.budrys@mif.stud.vu.lt

Vadovas: asist. dr. Linas Petkevičius

Vilnius - 2023

Summary

This thesis delves into the exploration of reinforcement learning from human feedback as a solution for general reinforcement learning problems. After a thorough review of the relevant literature, the study compares two distinct reinforcement learning methodologies: traditional reinforcement learning and a variant of reinforcement learning that integrates human feedback loops to enhance the agent's learning speed by utilizing human feedback. The findings reveal that while our implementation of the reinforcement learning from human feedback algorithm provides a significant edge in training speed, it also introduces additional challenges, such as complex code requirements and prolonged training duration.

Keywords: reinforcement learning, offline reinforcement learning, reinforcement learning from human feedback.

Santrauka

Šiame darbe gilinamasi į skatinamąjį mokymą su žmogaus grįžtamuoju ryšiu, kaip sprendimą bendrosioms skatinamojo mokymosi problemoms spręsti. Atlikus išsamią literatūros apžvalgą, darbe lyginami du skirtingi skatinamojo mokymosi metodai: tradicinis skatinamasis mokymasis ir skatinamasis mokymasis su žmogaus grįžtamuoju ryšiu, kuris siekia padidinti agento mokymosi greitį nukreidamas agentą teisinga linkme naudodamas žmogaus ekspertizę. Rezultatai parodo, kad nors mūsų skatinamojo mokymosi su žmogaus grįžtamuoju ryšiu algoritmo įgyvendinimas suteikia reikšmingą pranašumą pagal mokymosi greičiui, jis taip pat kelia papildomus iššūkius, tokius kaip sudėtingas kodas, reikalavimai ir ilgesnė agento mokymosi trukmė.

Raktiniai žodžiai: skatinamasis mokymas, „offline“ skatinamasis mokymasis, skatinamasis mokymasis su žmogaus grįžtamuoju ryšiu.

Contents

Introduction

Nowadays, reinforcement learning (RL) is considered as a powerful tool for enabling autonomous agents to learn complex tasks in dynamic environments. The approach of RL has led to significant advancements in various domains such as natural language processing models (such as ChatGPT[1]) [SH23], gaming [ACT19], healthcare [KCB+19] and economy [ZTS+21]. However, in many real-world scenarios, defining an accurate reward function for an RL algorithm is both challenging and time-consuming therefore. In such cases, reinforcement learning from human feedback has emerged as a promising direction, allowing human experts to guide the learning process [CLB+17].

One popular approach to incorporating human feedback in reinforcement learning, as described in the work [CLB+17], involves learning from comparisons using deep neural networks. In this method, the human reviewer is presented with a pair of trajectories generated by the agent and is asked to select the one they prefer. The RL algorithm then uses this preference information to learn and improve its performance. However the paper in the described work did not present any sources of implementation.

Aim

The aim of this to create a reinforcement learning algorithm that leverages human feedback through multiple generated trajectories, utilizing a ranking system to rate the data and adjust it based on the rankings for offline learning, to thereby try and improve the learning speed. This method departs from the traditional focus on learning from comparisons using deep neural networks. Instead, it aims to harness human expertise efficiently and effectively, drawing on recent trends in human-in-the-loop reinforcement learning.

Goals

Goals for this topic:

- Investigate related literature about reinforcement learning (RL), reinforcement learning from human feedback (RLHF).
- Implement a RL algorithm that incorporates human feedback to increase learning speed.
- Train agents with RL and RLHF algorithms using a specific environment.
- Modify the implemented algorithm until results meet set benchmarks.
- Compare results of all tested algorithms and variations.
- Formulate conclusions on reinforcement learning from human feedback based on the results.

[1] `https://openai.com/blog/chatgpt`

1 Reinforcement Learning

Reinforcement learning is a class of solution methods that work well on a class of problems. Reinforcement learning problems involve learning what to do - how to map situations to actions - so as to maximise a numerical reward signal [SB14].

This method of learning can be described as a learning child - where without anyone to teach, the only way to learn is to attempt solving the problem and learning from your mistakes or successes. This type of learning is the opposite of following a set of instructions.

Basic RL workflow is described in Figure 1.

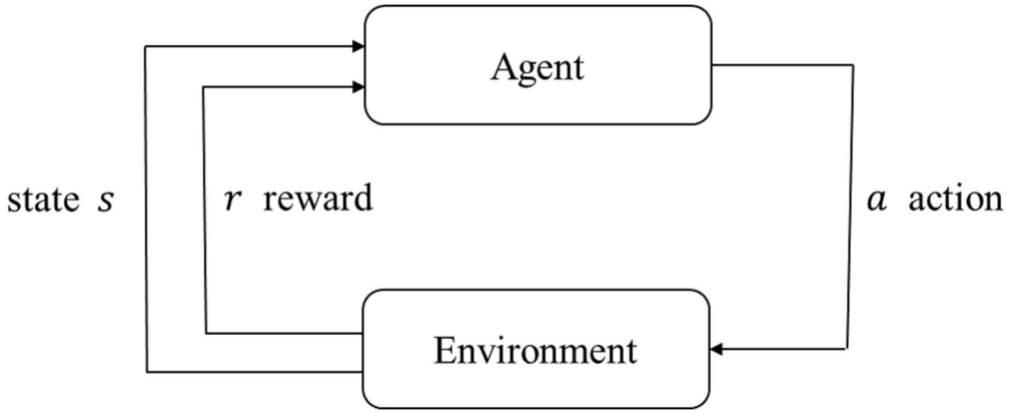

Figure 1: When the Agent performs an action, the state of the environment is changed, and a reward signal is feedback to the Agent. The Agent selects the next action according to the reward signal and the current state of the environment while trying to maximize rewards. The actions selected affect not only the immediate rewards, but also the state of the environment at one point and the final values [HHF+19].

1.1 Basic Reinforcement Learning System

A basic reinforcement learning system consists of these elements ([SB14]):

- **Agent** - the subject that is learning by interacting with the given environment.
- **Environment** - the space in which the agent operates, environment responds to the actions, that the agent makes.
- **Policy** defines the learning agent's way of behaving at a given time. Policy is a mapping from perceived states of the environment to actions to be taken when in those states [SB14].
- **Reward signal** - the goal in a reinforcement learning problem. The training is performed with the sole purpose to maximize the reward.
- **Value function** specifies what is good in the long run. Values indicate the long-term desirability of states after taking into account the states that are likely to follow, and the rewards available in those states [SB14].

- **(Optionally) - Model of the environment** allows impressions about the environment and its behavior to be made.

1.2 Markov Decision Process

Markov Decision Processes (MDPs) are a fundamental aspect of reinforcement learning theory, providing a mathematical framework for modeling decision-making in dynamic environments.

To define Markov Decision Process we first need to define a Markov Process (known as Markov Chain). A sequence of states is Markov if and only if the probability of moving to the next state S_{t+1} depends only on the present state S_t and not on the previous states $S_1, S_2, \cdots, S_{t-1}$. That is, for all t, $\mathbb{P}[S_{t+1}|S_t) = \mathbb{P}[S_{t+1}|S_1, S_2, \cdots, S_t]$ [Han18].

A Markov Process is a tuple (S, P), where:

- S is a (finite) set of states.
- P is a state transition probability matrix. $P_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$.

Having defined A Markov Process we can define Markov Decision Process. A Markov Decision Process is a tuple (S, A, P, γ, R), where:

- S is a finite set of states.
- A is a finite set of actions.
- P is the state transition probability matrix. $P^a_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$
- $\gamma \in [0, 1]$ is called the discount factor.
- $R : S \times A \to \mathbb{R}$

Markov Decision Processes are widely used in reinforcement learning to represent and solve sequential decision-making problems, allowing agents to learn optimal policies for acting within their environments.

1.3 Monte Carlo Prediction and TD Methods

Monte Carlo (MC) methods are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns are available, here we define Monte Carlo methods only for episodic tasks. The term "Monte Carlo" is often used more broadly for any estimation method whose operation involves a significant random component.

Monte Carlo Prediction:
Lets suppose we wish to estimate $v_\pi(s)$, the value of state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in an episode the first visit to s. s may be visited multiple times in the same episode. Let us call the first time it is visited in an episode the *first visit* to s. The *first visit* Monte Carlo method estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the *every visit* Monte Carlo method averages the returns following all visits to s. [SB14]

Initialize:
    $\pi \leftarrow$ policy to be evaluated
    $V \leftarrow$ an arbitrary state-value function
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
    Generate an episode using π
    For each state s appearing in the episode:
        $G \leftarrow$ return following the first occurrence of s
        Append G to $Returns(s)$
        $V(s) \leftarrow$ average($Returns(s)$)

Figure 2: The first-visit MC prediction for estimating v_π. Capital V is used for approximate value function because after initialization it becomes a random variable [SB14].

Temporal Difference Prediction: Temporal Difference (TD) learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics.

Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy π, both methods update their estimate v of v_π for the non-terminal states S_t occurring in that experience. Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$. A simple every-visit Monte Carlo method suitable for non-stationary environment is

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

where G_t is the actual return following time t, and α is a constant step-size parameter. The difference between Monte Carlo and Temporal Difference methods is that MC methods must wait until the end of the episode to determine the increment to $V(S_t)$, whereas TD methods only need to wait until the next step. At time $t+1$ they immediately form a target and make an update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. The simplest TD method, known as TD(0) is

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

In effect, the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This method is known as TD(0). [SB14]

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A; observe reward, R, and next state, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 3: TD(0) prediction. The process iteratively updates the value function $V(s)$ for a given policy π, using the observed reward R and the estimated value of the next state [SB14].

In summary, Monte Carlo prediction methods wait until the end of an episode to update the value estimates, whereas Temporal Difference prediction methods can update the estimates at each time step. This makes TD learning more suitable for online learning and continuous tasks, as it can learn from incomplete sequences of experience [SB14].

1.4 Reinforcement Learning Environments

If we think of reinforcement learning as a concept in real life, it can be applied to any environments as long as they provide the necessary feedback for the agent to learn. By having a goal in mind (reward), the agent can explore different actions and strategies to achieve it.

In reinforcement learning the agent performs the actions based on the reward and observation and since reward is a numerical value, the environments are classified by their action space and observation space.

1.4.1 Action Space

Environment's action space defines the actions that the agent can take at a given point in time. There are two well defined types of action spaces:

- **Discrete** action space allows the agent to select a single action from a list of all possible actions. This type is commonly used in environments where performing one action at a time is sufficient, and the total number of actions is not too large.
- **Continuous** action space enables the agent to perform any number of actions from a list of all possible actions, expressed as a real-value vector. Continuous action spaces are used for more complex environments, where there are many possible actions and real-value precision is necessary.

Many tasks of interest, most notably physical control tasks, have continuous (real valued) and high-dimensional action spaces.

1.4.2 Observation Space

The observation space can also be defined as discrete or continuous:

- **Discrete** observation space is perceived by the agent as a specific state out of a number of possible ones, for example a game of chess has a finite amount of states.
- **Continuous** observation space is perceived by the agent through continuous variables. An example of this would be driving a car, or flying a rocket, where there are practically infinite amount of variations when it comes to sensory input.

Unlike action space, an environment can incorporate both types of observations since the observations are typically expressed as a floating-point number array. This flexibility allows RL to be applied to a wide range of problems, making it a versatile and powerful tool in various domains.

1.5 Reinforcement Learning Exploration vs Exploitation

In reinforcement learning, agents face a fundamental challenge: balancing exploration and exploitation. To maximize the cumulative reward, an agent must:

- **Exploit** what it has already learned by selecting actions known to yield high rewards.
- **Explore** new actions to discover potentially better strategies and improve future rewards.

To obtain a higher reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing the reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it already knows in order to obtain reward, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best. On a stochastic task, each action must be tried many times to gain a reliable estimate its expected reward [SB14].

This balance between exploration and exploitation can be difficult to achieve, especially when the environment is complex or stochastic.

1.6 Reinforcement Learning Algorithms

Having discussed the foundational concepts of reinforcement learning, the focus of this section will be the heart of the RL framework: the algorithms that drive learning and decision-making processes in RL agents. We will try to examine their underlying principles, key differences, and respective strengths and weaknesses, and address other challenges inherent to the RL setting.

1.6.1 On-Policy and Off-Policy Reinforcement Learning Comparison

On-policy RL involves iteratively collecting experience by interacting with the environment, typically with the latest learned policy, and then using that experience to improve the policy. There are scenarios where this is impractical, such as health care, education or robotics [LKTF20]. On-policy algorithms are a viable way to train agents, where sampling the environment has no limit, however, this has its limits, for example, if the environment is very complex.

In **off-policy** RL, the agent is still free to interact with the environment. However, it can update its current policy using experiences collected from any previous policies. This increases the sample efficiency of training since the agent does not have to discard all of its previous interactions and can instead maintain a buffer where old interactions can be sampled multiple times during training [PMC23]. The off-policy paradigm is deeply related to Offline Reinforcement Learning, which is further discussed in section 1.8.1. The algorithm mainly used for experimentation part of this work, called DQN, is also an off-policy algorithm.

1.6.2 Value-Based Reinforcement Learning Algorithms

We will begin by introducing a value based, off-policy algorithm called Q-learning - one of the early breakthroughs in reinforcement learning[2]. Q-learning is defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max Q(S_{t+1}, a) - Q(S_t, A_t)].$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated [SB14].

Another value-based algorithm is Deep Q-Network (DQN). Since this algorithm was mainly used for experimentation part of this work it has its own separate section 1.7 that goes deeper into detail on how the algorithm works.

1.6.3 Policy-Based Reinforcement Learning Algorithms

Methods such as TD-learning typically try to estimate the expected long-term reward of a policy for each state s and time step t, also called the *value function* $V_t^\pi(s)$. The value function is employed to assess the quality of executing action a in state s. This evaluation is then used to directly compute the policy via action selection or to update the policy π.

However, value function methods struggle with the challenges encountered in some fields, such as robotics RL, as these approaches require filling the complete state-action space with data [DNP13] whereas policy-based reinforcement learning

[2]Q-learning was introduced in 1989 by Chris Watkins [Wat89], this was 34 years ago at the time of writing this work

methods focus on the policy itself without the need of a separate value function.

Lets take a look at a policy-based *policy search* (PS) algorithm.

Model-based policy search algorithms typically assume the following set-up: The state s evolves according to the Markov dynamics

$$s_{t+1} = f(s_t, a_t) + w, a_0 \sim p(a_0),$$

where f is a non-linear function, s is a control signal (action), and w is additive noise, often chosen to be i.i.d. Gaussian. The initial state distribution $p(s_0)$ is usually given by a Gaussian distribution $N(s_0|\mu_0^s, \Sigma_0^s)$. Policy-search objective is to find a parameterized policy π_θ^* that maximizes the expected long term reward

$$\pi_\theta^* \, arg\, max \sum_{t=1}^{T} \gamma \mathbb{E}[r(s_t, a_t)|\pi_\theta], \gamma \in [0, 1],$$

where r is an immediate reward signal, γ a discount factor, and the policy π_θ is parameterized by parameters θ. Therefore, finding π_θ in equation is equivalent to finding the corresponding optimal policy parameters θ^* [DNP13].

1.6.4 Model-Based Reinforcement Learning Algorithms

Model-free RL methods learn directly from interactions with the environment, but **model-based** RL methods can simulate transitions using the learned model, resulting in increased sample efficiency. This is particularly important in domains where each interaction with the environment is expensive.

The key idea behind Model-Based RL is to learn a transition model that allows for simulation of the environment without interacting with the environment directly. Model-based RL does not assume specific prior knowledge. However, in practice, prior knowledge can by incorporated to speed up learning. Model learning plays an important role in reducing the amount of required interactions with the (real) environment, which may be limited in practice. For example, it is unrealistic to perform millions of experiments with a robot in a reasonable amount of time and without significant hardware wear and tear [ADBB17].

One prominent example of a model-based reinforcement learning algorithm is the Monte Carlo Tree Search (MCTS). MTCS is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results.

Monte Carlo Tree Search rests on two fundamental concepts: that the true value of an action may be approximated using random simulation; and that these values may be used efficiently to adjust the policy towards a best-first strategy. The algorithm progressively builds a partial game tree, guided by the results of previous exploration of that tree. The tree is used to estimate the values of moves, with these estimates (particularly those for the most promising moves) becoming more accurate as the tree is built [BPW+12].

Four steps are applied per search iteration of the algorithm:

1. **Selection**: Starting at the root node, a child selection policy is recursively applied to descend through the tree until the most urgent expandable (has not visited potential children) node is reached.

2. **Expansion**: One or more child nodes are added to expand the tree, according to the available actions.

3. **Simulation**: A simulation is run from the new node(s) according to the default policy.

4. **Backpropagation**: The simulation result is "backed up" (i.e. backpropagated) through the selected nodes to update their statistics.

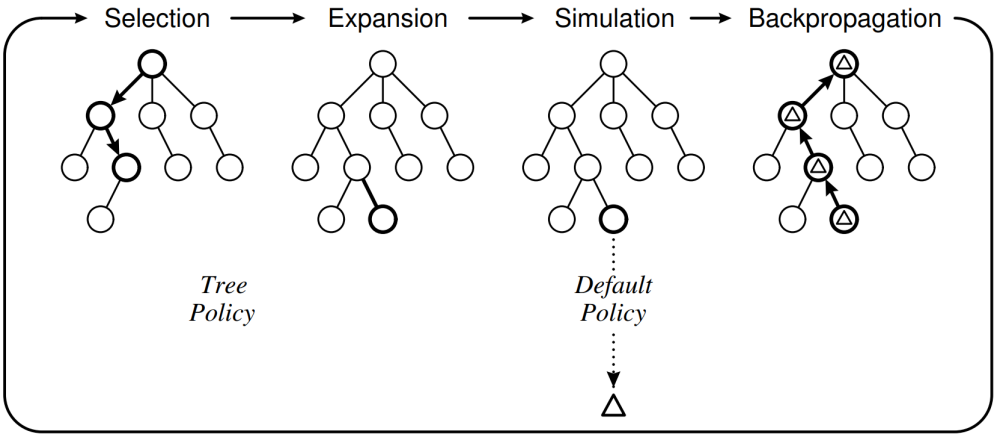

Figure 4: One iteration of the general MCTS approach [BPW+12].

These may be grouped into two distinct policies:

1. **Tree Policy**: Select or create a leaf node from the nodes already contained within the search tree (selection and expansion).

2. **Default Policy**: Play out the domain from a given non-terminal state to produce a value estimate (simulation).

MTCS Search pseudo-code

function $MTCSSearch(s_0)$
    create root node v_0 with state s_0
    while within computational budget **do**
        $v_1 \leftarrow TreePolicy(v_0)$
        $\Delta \leftarrow DefaultPolicy(s(v_1))$
        $Backup(v_1, \delta)$
    end while
    return $a(BestChild(v_0))$

1.7 DQN Deep Reinforcement Learning Algorithm

As reinforcement learning algorithms have evolved, the incorporation of deep learning[3] techniques has given rise to a new class of powerful algorithms, one of the most prominent being the Deep Q-Network (DQN) algorithm. DQN is a value, off-policy and model based algorithm.

DQN combines the principles of Q-learning with deep neural networks to create a groundbreaking method that has demonstrated human-level performance in a variety of Atari[4] games. The theory of reinforcement learning provides a normative account, deeply rooted in psychological and neuroscientific perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations.

Deep Q-network (DQN) is a network which is able to combine reinforcement learning with a class of artificial neural network known as deep neural networks, in which several layers of nodes are used to build up progressively more abstract representations of the data, have made it possible for artificial neural networks to learn concepts such as object categories directly from raw sensory data. DQN makes use of particular architecture known as the deep convolutional network, which uses hierarchical layers of tiled convolutional filters [MKS+15].

We consider tasks in which the agent interacts with an environment through a sequence of observations, actions and rewards. The goal of the agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function:

$$Q^*(s,a) = \max \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, a_t = a, \pi]$$

which is the maximum sum of rewards r_t discounted by γ at each time step t, achievable by a behaviour policy $\pi = P(a|s)$, after making an observation (s) and taking an action a.

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values $r + \gamma \max Q(s', a')$. These instabilities with a variant of Q-learning, which uses two key ideas. First, the algorithm uses a biologically inspired mechanism that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution termed experience replay. Second, an iterative update that adjusts the action-values (Q) towards target values that are only peri-

[3]Deep learning is a type of machine learning that utilizes artificial neural networks with many layers.

[4]Atari, founded in 1972, is an influential video game company whose games are often used as benchmarks for RL.

odically updated is used, thereby reducing correlations with the target.

The approximate value function $Q(s, a; \theta_i)$ is parameterized using the deep convolutional neural network shown in Figure 2. in which θ_i are the parameters (that is, weights) of the Q-network at iteration i. To perform experience replay the agent's experiences are stored: $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time-step t in a data set $D_t = \{e_1, \cdots e_t\}$. During learning, Q-learning updates are applied on samples (or minibatches) of experience $(s, a, r, s^{'}) \sim U(D)$, drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration i uses the following loss function:

$L_i(Q_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r + \gamma \max Q)(s^{'}, a^{'}; \theta_i^{-}) - Q(s, a; \theta_i)^2]$

in which γ is the discount factor determining the agent's horizon, hi are the parameters of the Q-network at iteration i and θ_i^{-} are the network parameters used to compute the target at iteration i. The target network parameters θ_i^{-} are only updated with the Q-network parameters (θ_i every C steps and are held fixed between individual updates.

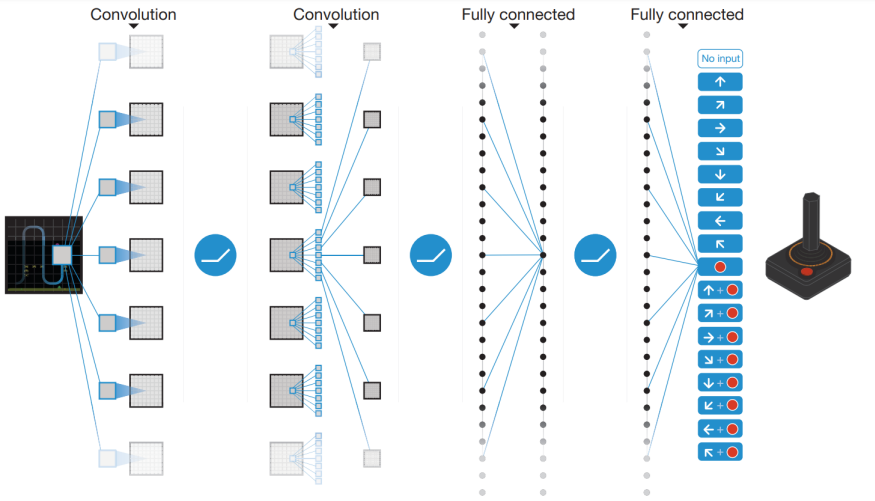

Figure 5: Schematic illustration of the convolutional neural network used to play Atari games [MKS+15].

1.8 Offline Reinforcement Learning

Offline reinforcement learning (also called batch reinforcement learning) is a data-driven RL paradigm that allows agents to learn from static datasets of previously collected experiences. The end goal of Offline RL is still to optimize the objective in the Equation, however the agent no longer has the ability to interact with the environment and collect additional transitions, using behavior policy [LKTF20].

Simply said, the model is being trained while viewing recorded past experiences, similar to how a human would attempt to learn from recorded instructions. This type of data utilization is useful in fields where there is no real way to attempt to train the model using online reinforcement learning, such as health care, because experimenting is not an option if the environment is impossible to simulate. However if the records of previous experiences are stored they could be used to train the model.

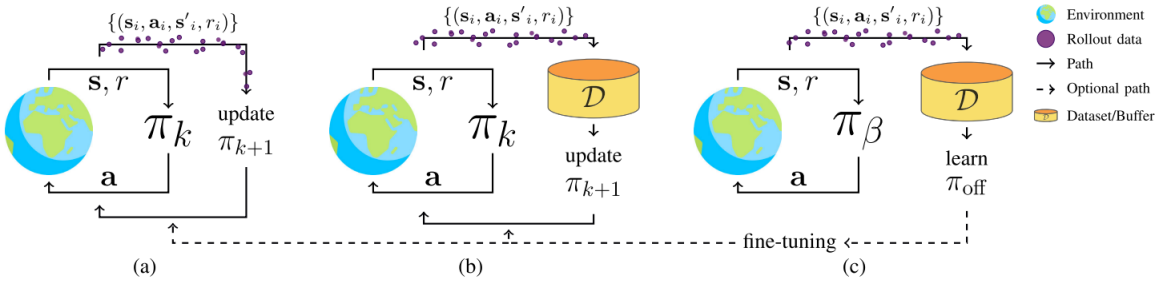

Figure 6: Illustration of the different RL paradigms, including: (a) - Online RL, (b) - Off-Policy online RL, (c) - Offline RL. In online RL only new experiences are used. In off-policy RL, we reuse previous experiences but still rely on a continuous collection of new experiences. In contrast, offline RL only uses previous experiences (datasets) to learn the new policies [PMC23].

In theory, any off-policy method could be used to learn a policy from a dataset of previously collected experiences. However, these methods often fail when exclusively working with offline data since they were devised under the assumption that further online interactions are possible, and algorithms can typically rely on these interactions to correct erroneous behavior. Finding a balance between increased generalization and avoiding unwanted behaviors outside of distribution is one of the core problems of offline RL. Moreover, this problem is further exacerbated by the widespread use of high-capacity function approximators. Most of the novel offline RL algorithms directly address this issue by proposing different losses or training procedures capable of mitigating distributional shift [LKTF20].

In section 4.4.1 we run into a similar issue where the offline trained model starts partially mimicking human behaviour due to model being overfitted to a specific dataset.

2 Reinforcement Learning from Human Feedback

Many tasks that are being attempted to solve using reinforcement learning are too complex to solve.

Reinforcement learning from human feedback is a RL approach that focuses on human feedback, opposed to receiving the reward from a reward function.

This approach can effectively solve complex RL tasks without access to the reward function, while providing feedback on less than 1% of agent interactions with the environment. This reduces the cost of human oversight far enough that it can be practically applied to state-of-the-art RL systems [CLB+17].

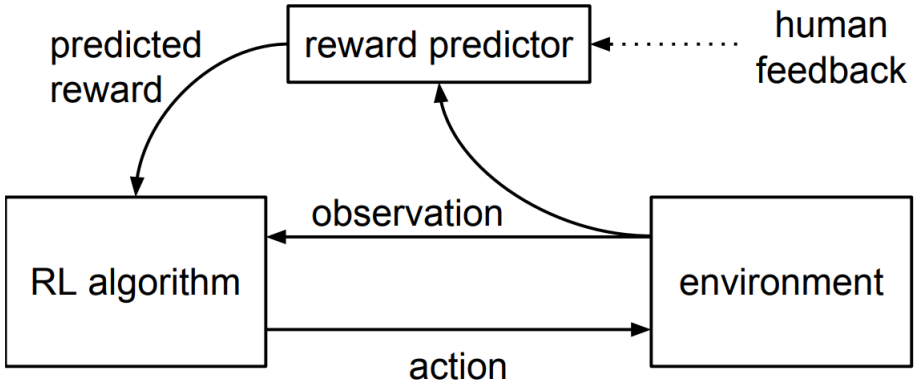

Figure 7: The reward predictor is trained asynchronously from comparisons of trajectory segments, and the agent maximizes predicted reward [CLB+17].

2.1 Preliminaries and Method

This section goes more into depth on how the RLHF algorithm works as described in the original work [CLB+17].

2.1.1 Setting and Goal

We consider an agent interacting with an environment over a sequence of steps; at each time t the agent receives an observation $o_t \in O$ from the environment and then sends an action $a_t \in A$ to the environment.

In traditional reinforcement learning, the environment would also supply a reward $r_t \in \mathbb{R}$ and the agent's goal would be to maximize the discounted sum of rewards. Instead of assuming that the environment produces a reward signal, we assume that there is a human overseer who can express preferences between trajectory segments. A trajectory segment is a sequence of observations and actions, $\sigma = ((o_0, a_0), (0_1, a_1), \cdots, (0_{k-1}, a_{k-1})) \in (O \times A)^k$. These trajectory segments could be evaluated by a human and informally, the goal of the agent is to produce trajectories which are preferred by the human, while making as few queries as possible to the human.

The algorithms' behaviours can be identified in 2 ways:

Quantitative: Preferences $\succ$ are generated by a reward function $r : O \times A \to \mathbb{R}$ if:

$$((o_0^1, a_0^1), \cdots, (0_{k-1}^1, a_{k-1}^1)) \succ ((o_0^2, a_0^2), \cdots, (0_{k-1}^2, a_{k-1}^2))$$

whenever

$$r((o_0^1, a_0^1) + \cdots + (0_{k-1}^1, a_{k-1}^1)) > r((o_0^2, a_0^2) + \cdots + (0_{k-1}^2, a_{k-1}^2))$$

If the human's preferences are generated by a reward function r, then our agent ought to receive a high total reward according to r. So if we know the reward function r, we can evaluate the agent quantitatively. Ideally the agent will achieve reward nearly as high as if it had been using RL to optimize r.

Quantatative: Sometimes there is no reward function by which we can quantitatively evaluate behavior (this is the situation where our approach would be practically useful). In these cases, all we can do is qualitatively evaluate how well the agent satisfies to the human's preferences. The workflow is as follows: start from a goal expressed in natural language, ask a human to evaluate the agent's behavior based on how well it fulfills that goal, and then present videos of agents attempting to fulfill that goal [CLB+17].

Instead of relying on an environment-generated reward signal, the agent seeks to optimize its behavior based on the preferences of a human overseer evaluating trajectory segments. This approach allows the agent to effectively adapt to human preferences, which may be difficult to quantify or may change over time. Evaluation of the agent's performance can be conducted both, using a known reward function (quantitatively), and by assessing how well the agent satisfies the human's preferences based on their goal expressed in natural language (qualitatively).

2.1.2 Collecting Human Feedback

The human overseer is given a visualization of two or more segments and then indicates which segment they prefer, that the segments are equally good, or that they are unable to compare the segments.

Lets say we have two segments, where:
σ^1 is the first segment, σ^2 is the second segment and μ is a distribution over {1, 2} indicating which segment the user preferred. If the human selects one segment as preferable, then μ puts all of its mass on that choice. If the human marks the segments as equally preferable, then μ is uniform. Finally, if the human marks the segments as incomparable, then the comparison is not included [CLB+17].

2.1.3 Fitting the Reward Function

Reward function can be interpreted as estimate $\hat{r}$ as a preference-predictor if we view $\hat{r}$ as a latent factor explaining the human's judgments and assume that the human's probability of preferring a segment i depends exponentially on the value of the latent reward summed over the length of the clip:

$$\hat{P}\left[\sigma^1 \succ \sigma^2\right] = \frac{\exp \Sigma \hat{r}(o_t^1, a_t^1)}{\exp \Sigma \hat{r}(o_t^1, a_t^1) + \exp \Sigma \hat{r}(o_t^2, a_t^2)}$$

$\hat{r}$ is chosen to minimize the cross-entropy loss between these predictions and the actual human labels:

$$loss(\hat{r}) = - \sum_{(\sigma^1, \sigma^2, \mu) \in D} \mu(1) \log \hat{P}\left[\sigma^1 \succ \sigma^2\right] + \mu(2) \log \hat{P}\left[\sigma^2 \succ \sigma^1\right]$$

A human might see patterns early in the training that would potentially allow the agent maximize long term rewards, therefore incorporating the feedback correctly into the reward function is very important.

3 Experiment Prerequisites

Before covering the process and the results of the experiment section, this section will focus on explaining the software and hardware used to perform the experiments.

3.1 OpenAI Gym

OpenAI Gym[5] is a Python[6] toolkit that focuses on the episodic setting of reinforcement learning, where the agent's experience is broken down into a series of episodes. In each episode, the agent's initial state is randomly sampled [BCP+16].

The reason why OpenAI Gym was chosen for this work is due to its popularity and usage example availability. The toolkit comes with ready to use environments which share a common interface, which is most commonly supported amongst public reinforcement learning libraries.

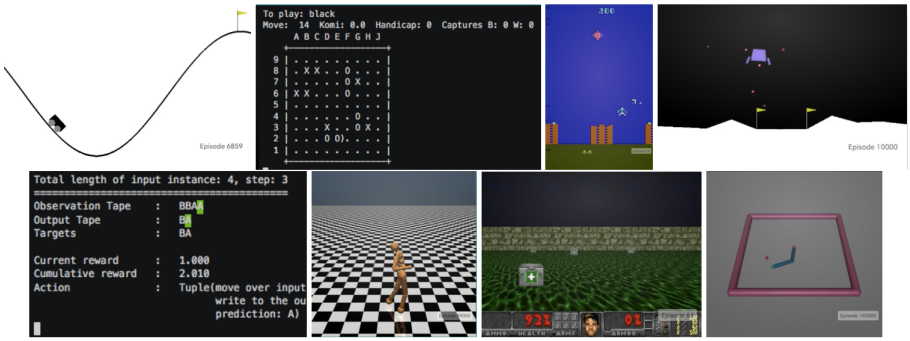

Figure 8: Images of some of the environments from OpenAI Gym [BCP+16].

The core functions provided by the OpenAI Gym that are used for agent training:

- **obs, rewards, done, info = env.step(self, action)** - performs an action (a step) and returns the current observations of the environment (obs), acquired reward (rewards), weather the episode is complete (done) and some additional information (info).
- **obs = env.reset(self)** - resets the environment so the episode can be restarted and returns the initial observation of the environment, required to perform the first step.

This work also utilizes some important helper functions:

- **play(self, env)** - creates a game window for the environment that a human can play. This is particularly used to record human gameplay for offline training.
- **rgb_array = env.render(mode="rgb_array")** - returns an RGB array of pixels that are representative of current state of the environment and can be used to create images or videos (more details in section 4.3.2).

[5] `https://www.gymlibrary.dev/`

[6] Python is a programming language, popular for solving RL problems `https://www.python.org/about/`

3.2 Lunar Lander Game

LunarLander is a reinforcement learning environment developed by OpenAI as part of the OpenAI Gym library. The environment is a simulation for the task of landing a lunar module on the moon's surface. The goal of the agent is to learn a control policy that allows the lander to touch down within a designated landing zone without damaging the hull of the ship while minimizing fuel consumption.

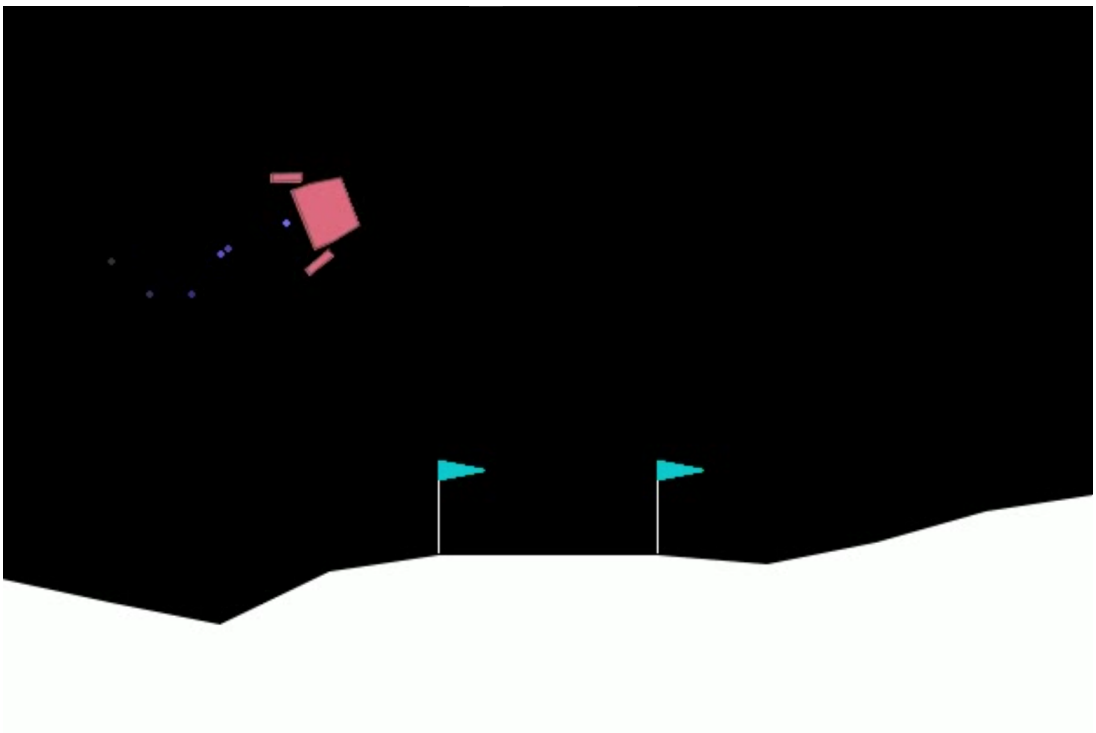

Figure 9: The visual representation of Lunar Lander environment. The ship is pink, the purple particles indicate engine thrust and the landing zone is contained within the 2 blue flags.

3.2.1 Actions

Four discrete actions are available to take when playing the Lunar Lander game:

1. Fire the main engine (thrust upwards)
2. Fire the left orientation engine (rotate clockwise)
3. Fire the right orientation engine (rotate counter-clockwise)
4. Do nothing (free-fall)

3.2.2 Observations

The observation space is a mixed eight-dimensional space (2.3.2 points out scenarios in which the space can be mixed), that contains six continuous and two discrete variables:

1. X-coordinate of the lander
2. Y-coordinate of the lander
3. X-velocity of the lander
4. Y-velocity of the lander
5. Angle of the lander
6. Angular velocity of the lander
7. Left leg contact with the ground (binary)
8. Right leg contact with the ground (binary)

3.2.3 Rewards

The reward structure is as follows:

- +100 for landing safely within the designated landing zone.
- -100 for crashing or coming to rest outside the landing zone.
- +10 for each leg making contact with the ground.
- -0.3 for each step the main engine is active.
- A small reward based on the lander's distance from the landing pad and its velocity

3.2.4 Maximum Potential Rewards

The environment is considered solved, when the agent achieves an average reward of +200 over 100 consecutive attempts, however the maximum possible reward is slightly higher and the training revealed it to be around 300 if the initial velocity is right.

3.2.5 Termination

The environment is terminated when any of these criteria are met:

- Environment run-time has reached 1000 steps.
- The lander has landed successfully, and stopped moving.
- The lander has crashed.

3.3 d3rlpy - an Offline and Offline Reinforcement Learning Library

d3rlpy is an open-sourced[7] offline deep reinforcement learning library for Python. d3rlpy supports a set of offline deep RL algorithms as well as off-policy online algorithms. API of all implemented algorithms is fully documented, plug-and-play and standardized so that users can easily start experiments with d3rlpy [SI22].

An advantage of this library is that it also supports online reinforcement learning therefore making it suitable for our experiments without implementing any RL algorithms from scratch.

3.3.1 Key d3rlpy Classes and Functions

This work utilizes these key classes and functions:

- **algorithm = algos.[Algorithm Name](...)** - creates and returns an object that executes the algorithm.[Algorithm Name] can be replaced with any algorithm name that the library provides.
- **MPMDataset(data)** - allows to create a dataset file from saved observations, actions, rewards, terminals. This was used to convert human gameplay into data used for training.
- **algorithm.build_with_env(env)** or **algorithm.build_with_dataset(dataset)** - Instantiates implementation object with OpenAI Gym object using the environment or with the dataset provided.
- **algorithm.fit(env, n_steps, ...)** - trains the model offline with the given dataset provided via algorithm.build_with_env(dataset).
- **algorithm.fit_online(env, n_steps, explorer, ...)** - trains the model online with the set explorer strategy.

[7] Open-source code is made free for any modification or redistribution.

3.3.2 d3rlpy algorithms

d3rlpy library provides a wide range of algorithms:

Algorithm	Discrete	Continuous	Optimized for offline RL
NFQ	✓	×	✓
DQN	✓	×	?
Double DQN	✓	×	?
DDPG	×	✓	?
TD3	×	✓	?
SAC	✓	✓	?
BCQ	✓	✓	✓
BEAR	×	✓	✓
CQL	✓	✓	✓
AWAC	×	✓	✓
CRR	×	✓	✓
PLAS	×	✓	✓
TD3+BC	×	✓	✓
IQL	×	✓	✓

Some of the algorithms are not explicitly marked as offline RL, however all algorithms support offline learning despite some of them not being optimal. Results in section 4.2.3 indicate that the offline learning for DQN provides good results, despite not being optimal for offline reinforcement learning.

3.4 Experiments Hardware

The device used for training was utilizing:

- **GPU** - NVIDIA GeForce RTX 3070 Ti 8GB VRAM
- **CPU** - Intel(R) Xeon(R) W-2295 @ 3.00GHz, 18 Cores, 36 Virtual Cores
- **RAM** - 64 GB (4 x 16GB) @ 3.200GHz

During training not even half of total available resources were utilized, due to d3rlpy library not utilizing CPU multi-threading to full potential and bottle-necking because 2/36 virtual cores were utilized at 100% with average of 14% utilization across all cores and GPU being utilized only at 7%.

No more than 2 GB of RAM was used at one time which accounts for around 3% of system's RAM.

4 Experiments

This section delineates the experimental setup designed to evaluate and juxtapose the performance of three distinct experiments. Each of these experiments represents a different approach to learning, with varying degrees of reliance on offline learning human feedback. Further, each successive implementation builds upon the modifications of its predecessor. The aim is to collect data and later investigate how these methods perform in comparison to each other using a specific environment.

All experiments will be trained while abiding these **criteria**:

1. A maximum of 10 million online training timesteps will be executed.
2. Results will be smoothed to account for a range of current and past results.

Benchmarks:

1. The number of time-steps required to reach a reward of 0.
2. The number of time-steps required to reach a reward of 200.
3. Identification of any other observed positive or negative trends.

Following experiments code for this thesis can be accessed in GitHub[8].

4.1 Experiment 1: Online Reinforcement Learning with DQN

The first experiment utilizes the Deep Q-Network (DQN) algorithm. This approach focuses solely on online reinforcement learning, wherein the agent learns and updates its model purely on its interactions with the environment. Consequently, this experiment signifies a more traditional reinforcement learning method, devoid of any integration of human feedback or offline learning components.

4.1.1 Algorithm Selection and Parameter Tuning

As discussed in Section 3.4, d3rlpy provides a range of algorithms to select from, with the initial goal being to successfully employ one of them. Despite the comprehensive documentation and usage examples provided by the library, achieving satisfactory results with any of the given algorithms proved difficult. The agents either failed to reach the desired goal quickly enough or did not reach it at all, with some even performing worse than a completely randomized agent (see Figure 10).

During the process of selecting and fine-tuning algorithm, it was initially observed that the performance of DQN was superior to the other tested algorithms when using the constant epsilon explorer. However, the results were still not satisfactory. By exploring various parameters and observing examples found in the author's GitHub repository[9] for training Atari environments, adjustments were made to the algorithm's parameters to achieve better performance. Key changes included:

- Adjusting learning rate from the default value of 0.0000625 to 0.00025.

[8] `https://github.com/picorro/reinforcement-learning-using-human-feedback-bachelors-work`
[9] `https://github.com/takuseno/d3rlpy/blob/master/reproductions/online/dqn.py`

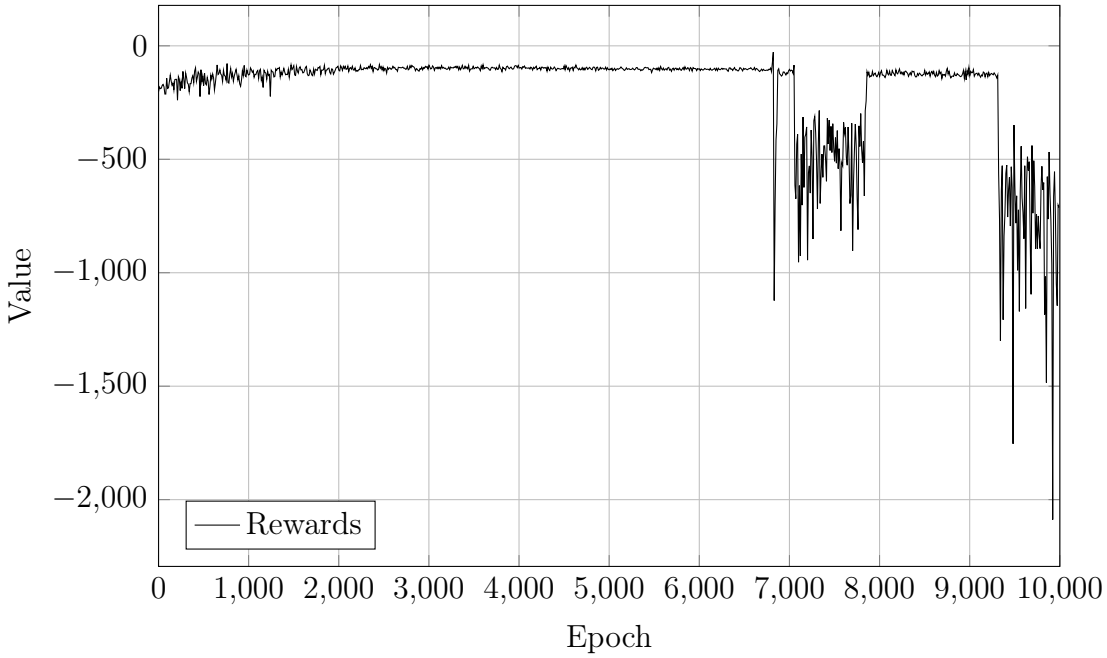

Figure 10: A failed attempt at online RL using the NFQ algorithm with default parameters and a constant exploration rate of 0.3. Each epoch accounts for 1000 steps. The agent's performance remained stagnant from the start with 2 major drops towards the end of the training.

- Introducing optim_factory = models.optimizers.RMSpropFactory()
- Introducing q_func_factory="mean"
- Introducing scaler="pixel"

In addition to these algorithm changes, several parameters of the **fit()** function were modified:

- Exploration policy was changed from a constant 0.3 to a linear descent from 1 to 0.1 over one million steps. This allowed for the agent to more consistently reach the desired result.
- The buffer size was also adjusted from 10 million to 1 million, providing a more suitable memory capacity for storing and sampling experiences since having a buffer that is too large may put too much emphasis on old experiences.

Final algorithm parameters:

- **learning rate = 0.00025**
- **optim_factory = RMSpropFactory()** - RMSpropFactory() is an adaptive learning rate optimization algorithm that has been specifically designed for training deep neural networks [10].
- **target_update_interval = 2500** - target network's parameters are updated every 2,500 steps in the environment.
- **batch_size = 32** - algorithm updates the Q-network by randomly sampling 32 transitions from the experience replay buffer at a time.
- **n_frames = 4** - takes an image of the environment every 4 steps.
- **scaler = "pixel"** - analyses the image by pixel values.
- **use_gpu = True** - incorporates GPU resources for training.

Final training parameters:

- **buffer = 1000000** - amount of step information stored for off-policy replayability
- **explorer = LinearDecayEpsilonGreedy(1, 0.1, 1000000)** - epsilon will start at 1, and during 1 million steps linearly go down to 0.1.
- **n_steps = 10000000** - all trainings were performed for 10 million steps.
- **n_steps_per_epoch = 1000** - each epoch consists of 1000 steps (this helps reduce log sizes).
- **update_start_step = 1000** agent should take at least 1000 steps in the environment before starting to update the Q-network.

[10] https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

4.1.2 Results of Experiment 1

In this section, we review the training results of Experiment 1, including benchmarks and a graph depicting the reward earned over time.

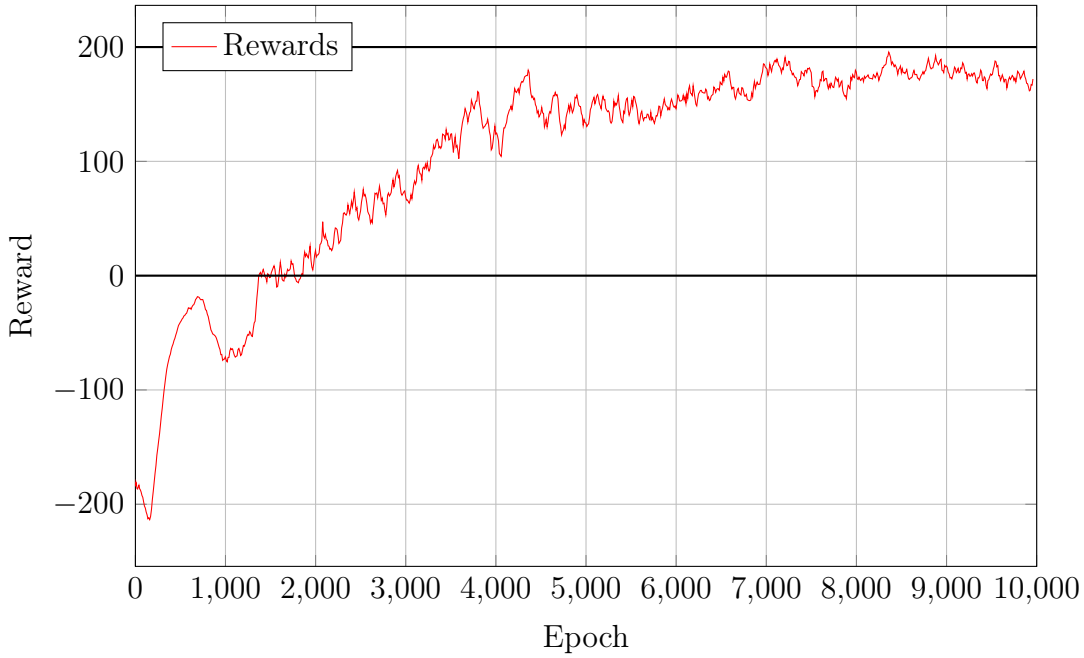

Figure 11: Training results of Experiment 1. The graph depicts the rewards earned during training. Each epoch represents 1000 steps. A smoothing factor of 0.97 is applied to the data.

Benchmark review for 5 consecutive run average:

1. 0 reward - approximately 1.4 million timesteps.
2. 200 - was not achieved (although some specific runs managed to reach it)

Experiment 1 produced a mix of positive and negative outcomes, exhibiting a degree of inconsistency. This inconsistency likely stems from the agent being unable to reach good results during the period of linearly decreasing exploration. After the exploration has reached a minimal value of 0.1 the agent is barely able to make any differences in its strategy.

4.2 Experiment 2 - Combination of Offline and Online Learning

The second experiment aims to improve on the first experiment. Instead of relying on the initial exploration to succeed during the first million steps, we provide the algorithm with recorded gameplay data from a human. This data allows for the algorithm to skip the initial phases of learning where the actions are most random by utilizing offline RL.

4.2.1 Offline Pre-training

The main issue with the first experiment, which exclusively used online RL, was its inconsistency. If the agent failed to find an effective strategy during the initial 1 million steps (the period where the exploration rate decreases to 0.1), it was very unlikely to achieve good results overall. In an attempt to address this issue, we introduce offline pre-training. This enables us to set the exploration rate lower, making it much less likely for the training in its entirety to be unsuccessful.

To perform offline training for a model, we need some data. We already covered the possibility of recording human gameplay and turning it into a dataset by using OpenAI Gym's **play(self, env)** and d3rlpy **MPMDataset(data)** functions (sections 3.1 and 3.4.1 respectively). Following these methods, a 10-minute human gameplay session was recorded and converted into a dataset for offline learning. The recording length of 10 minutes was chosen as it provided a substantial amount of data for training without being too lengthy or tedious for the human participant.

The parameters for offline training were left as default with the exception of learning rate changed to match the online learning rate of 0.00025. The training was performed for 2 million steps. It was determined by trial and error that 2 million was a good amount of steps, enabling the agent to learn and also not over-fit the model to the particular dataset. Overfitting could occur if the model becomes too adapted to the learning patterns from a specific dataset and fails to generalize well to new, unseen situations. By limiting the training steps, we aim to strike a balance between learning useful information from the dataset and maintaining the model's ability to generalize.

4.2.2 Online Training Parameterization

Prior to commencing online training, it is essential to assess the model's current progress in order to appropriately calibrate the exploration rate. Setting the exploration rate excessively high may result in the model losing its progress; even though the model may be capable of solving the problem adequately, the majority of its moves would be random. Conversely, if the exploration rate is set too low and the model has not made sufficient progress, it will continue to employ ineffective strategies, as there will be an insufficient number of random moves for discovering better solutions.

Ideally, performance evaluation could be conducted through a series of test runs. However, in this study, we adopted a manual trial-and-error strategy to achieve the desired outcomes. Upon obtaining satisfactory results, the exploration parameters were maintained consistently across all runs, ensuring the data could be compared with other experiments.

After some trial and error, we chose to use a linear descent explorer (starting at 0.7 and decreasing to 0.1) during the first million steps for the online training. This exploration strategy was found to strike a balance between exploring different possibilities and exploiting the knowledge gained during the pre-training phase.

4.2.3 Experiment 2 Results

The results of the second experiment were highly encouraging. Not only were successful outcomes achieved in individual runs, but the combined reward data from five consecutive runs also demonstrated a consistent and expedient accomplishment of higher rewards by the algorithm.

Let's first look at the averaged data:

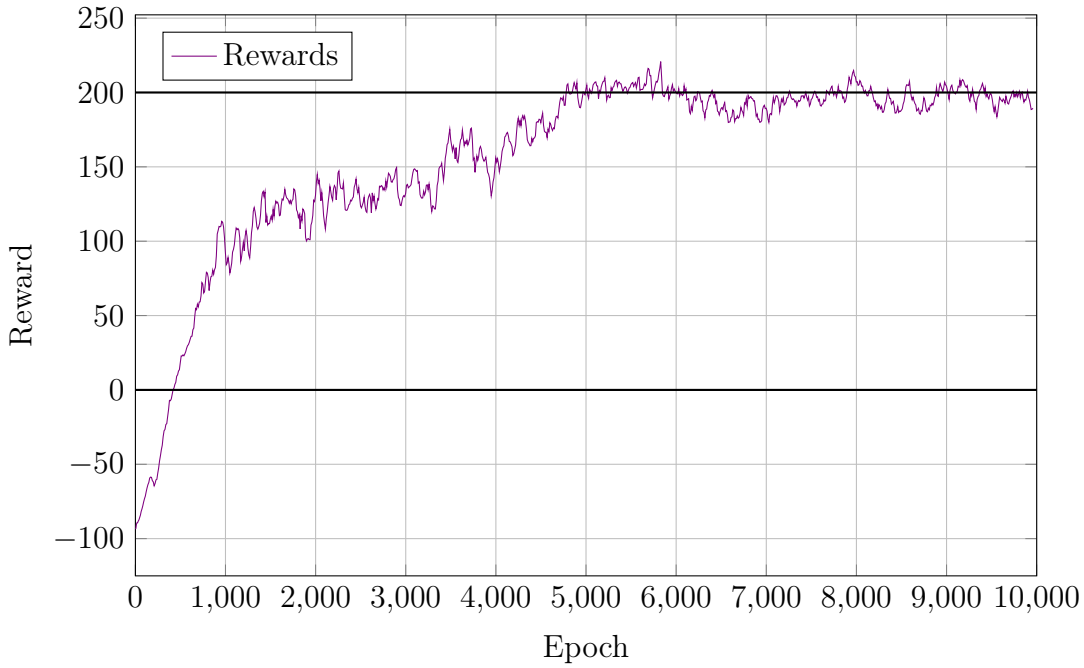

Figure 12: Results of Experiment 2 training. The graph represents the rewards obtained during training in the second experiment. Each epoch corresponds to 1000 steps. A smoothing factor of 0.97 is applied to the data.

Benchmark review for 5 consecutive run average:

1. 0 reward - Achieved in 0.4 million timesteps.
2. 200 reward - Achieved in 4.8 million timesteps.
3. The initial training curve is notably steep due to the model being pre-trained and exploration rate being 30 percent lower in comparison to the first experiment.

4.3 Experiment 3 - Offline and Online Learning with Human Interruption Loop

The objective of Experiment 3 is to further amplify the initial rewards curve by leveraging human expert feedback. This is accomplished by evaluation multiple trajectories and using those evaluations to guide the training process in the most promising direction.

4.3.1 Trajectory generation

In the loop, the current model state serves as the basis for generating the algorithm's trajectories. We use five different epsilon values: 0.05, 0.1, 0.2, 0.3 and 0.5, where epsilon determines whether the action will be random or the current policy. With each step of the environment a random number n is generated and used to deviate the trajectory from the current model slightly.

The pseudo-code for the step loop is as follows:
$if(n < epsilon) =\geq$ make a random move
$else =\geq$ make an optimal move based on current model's prediction

This type of trajectory generation ensures that the data generated will introduce novelty since exploration in RL is defined as randomizing the moves.

4.3.2 Creating videos from Environments

Since the goal of the training is to finish as quick as possible and multiple environments are trained at one time, to extract the videos for gameplay some code is needed. The videos that were used in evaluation prompts, when human expert opinion was needed, **rgb_array = env.render(mode="rgb_array")** method was used.

This method returns the a matrix that contains red, green and blue values for each pixel, that can be converted into a video frame (an image), and then using a stream of these frames a video can be made.

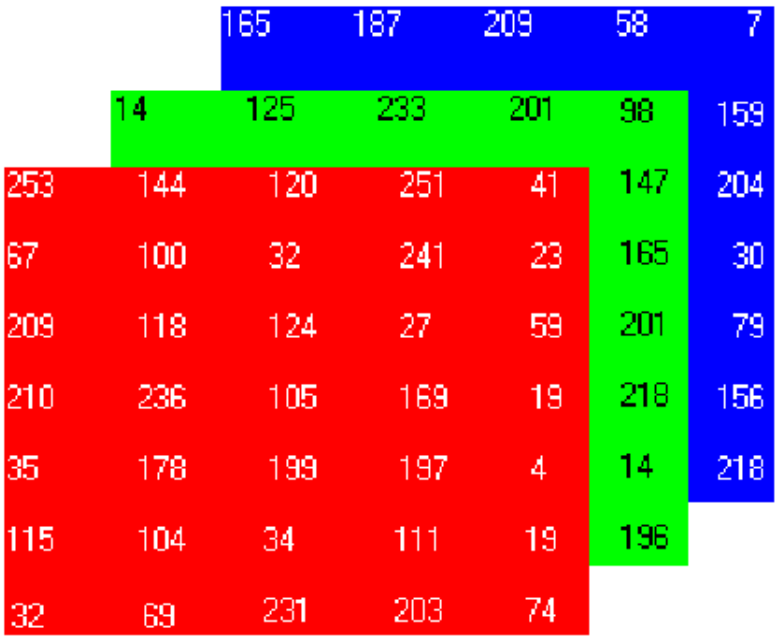

Figure 13: An 3-dimensional RGB matrix visual representation. Each rectangle represents one pixel on 2-dimensional grid represents one channel's one pixel's color. Combining all three matrices produces final colors for the represented pixels [Cou01].

4.3.3 Human Evaluation User Interface

The algorithm prompts the user to evaluate five videos to generate data for offline training based on the recorded trajectories using a graphical user interface window that plays all videos on loop and contains a text field to enter the ranking as a five number sequence.

After collecting the user's input, algorithm modifies generated datasets based on user's ranking ant continues to train the model.

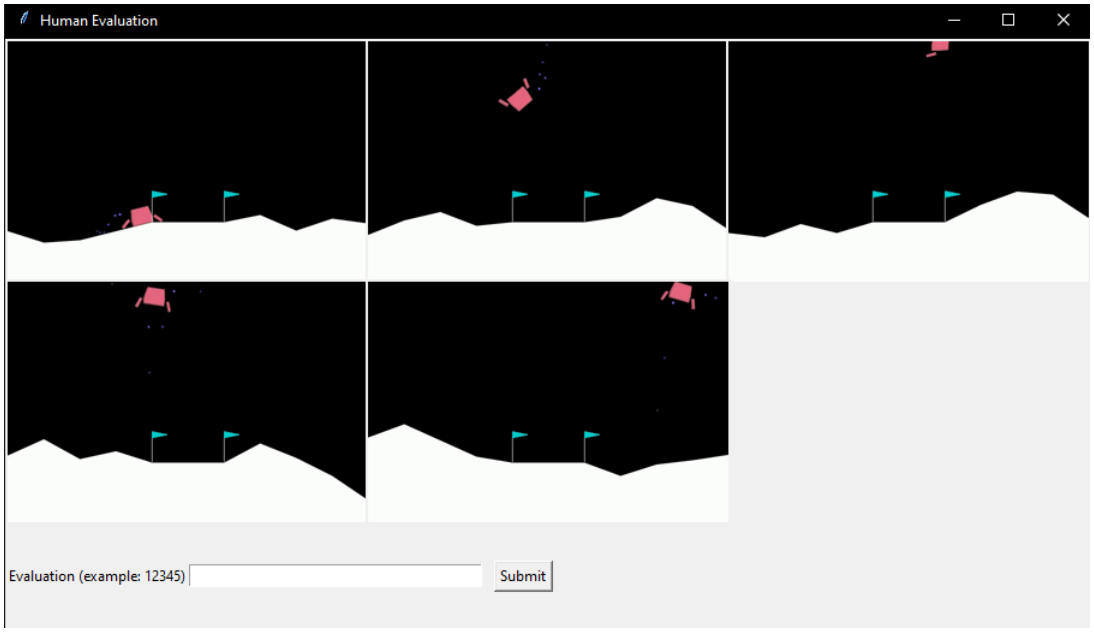

Figure 14: The GUI window used to display the videos for human ranking. At the bottom there is a text-field that allows to rate the videos using a sequence of numbers. The UI was made using a python library called Tkinter.

4.4 Algorithm Workflow

The algorithm begins similarly to Experiment 2, wherein it utilizes the same data recorded by human and conducts offline training for 2 million steps. Following this initial training, the algorithm enters a loop that: generates trajectories, creates datasets based on human expert trajectory evaluations and carries out both offline and online training, and finally, archives the results.

Given the complexity of the complete workflow, the subsequent sections will delve into each key component of the final algorithm in a more understandable and detailed manner.

Following that, more in depth explanations about how this workflow was chosen will be provided, since without the context of the final version, it would be hard to understand the reasoning behind these modifications.

4.4.1 Initialization and Setup

The algorithm starts by setting up the initial values, such as environment and offline, online training steps. It also establishes the parameters for epsilon-greedy exploration.

4.4.2 Initial Offline and Online Training

For the initial offline training, the algorithm is fit to the pre-recorded dataset for 2 million steps. During the initial online training phase, the algorithm generates step and epsilon progression arrays, which regulate exploration progression and the number of training steps per human intervention. The buffer is maintained at a

constant size of 1 million and is not reset at any point during training (meaning all interactions with the environment can be sampled during training).

4.4.3 Human Feedback Loop

Upon completion of the initial offline and online training, the algorithm enters a loop for a specified number of human interventions. In each cycle, it generates five trajectories with different epsilon values[11], records videos of these trajectories, and solicits human feedback on the videos. Based on the human feedback, the algorithm adjusts the last reward in the dataset to reflect the human evaluation.

The datasets with updated rewards are combined along with the initial recorded data, and the algorithm performs offline training on this new dataset.

The online training resumes with the updated model, utilizing the same buffer and epsilon-greedy explorer using values stored in the array set during the initialization phase. The process is repeated for each intervention, updating the model weights, and incorporating human feedback via the graphical user interface in each iteration. First iterations of the loop are fast since only a small amount of online training done, however last iteration needs to fine-tune the model and takes more than 99% of all online training steps performed in this method.

4.4.4 Data Generation

Once the loop finishes executing, all of the online training logs are combined to produce into a single TensorBoard[12] file. A copy of the current neural network model is created at the end of every epoch (1000 steps), including the final version and can be re-used to review or re-training at any future point.

[11] During recording loop if a random number in range 0-1 is less than the epsilon, a random move is made, otherwise the best predicted move by the model is made.

[12] TensorBoard is a tool for providing the measurements and visualizations needed during the machine learning workflow, in this case generating graphs from the training results.

4.4.5 Pseudo-Code

Let us go over the algorithm in more detail on key parts from the perspective of extremely simplified code:

```
1: fit(dataset, steps) # initial offline training
2: Initialize online explorer
3: Initialize online buffer
4: fit_online(env, steps, explorer, buffer) # initial online training
5: for in range(0, interventions)
6:     trajectory_epsilons # array of epsilons
7:     for epsilon in trajectory_epsilons
8:         while(True)
9:             if random(0 - 1) < epsilon then take random action
10:            else take best predicted action
11:            Append dataset and record video frame
12:            if episode finished then break
13:        Save dataset and video
14:    Human evaluates the videos
15:    Edit datasets based on ranking
16:    fit(dataset) # train offline with edited dataset
17:    fit_online()
18: Combine logs
19: exit
```

In summary, the algorithm is a hybrid of offline and online reinforcement learning methods, enhanced by human feedback to improve the model's performance. By incorporating human evaluation into the learning process, the algorithm can better align with human preferences and generate more desirable behavior in the agent, especially early in the training cycle. This iterative process of generating trajectories, acquiring human feedback, and updating the model via offline and online training ensures that the algorithm continually refines its understanding of the task.

4.4.6 Offline Model Overfitting

Early trials incorporated 10 human interventions, gradually decreasing towards the end of training. A significant drop in rewards was noted after each offline training session. A pattern of overfitting to the human-recorded dataset was observed after viewing the recordings. The agent appeared to partially mimic the human behavior: for instance, during data recording, the priority was to minimize the number of failed attempts, often resulting in extended hovering above the landing pad while adjusting the angle. Consequently, the agent developed a tendency to float above the landing pad without actually landing, highlighting the negative impact of overfitting on its performance.

This issue significantly reduced the learning capability for the algorithm, because instead of increasing training speed, offline training would instead reduce the rewards gained by a substantial amount.

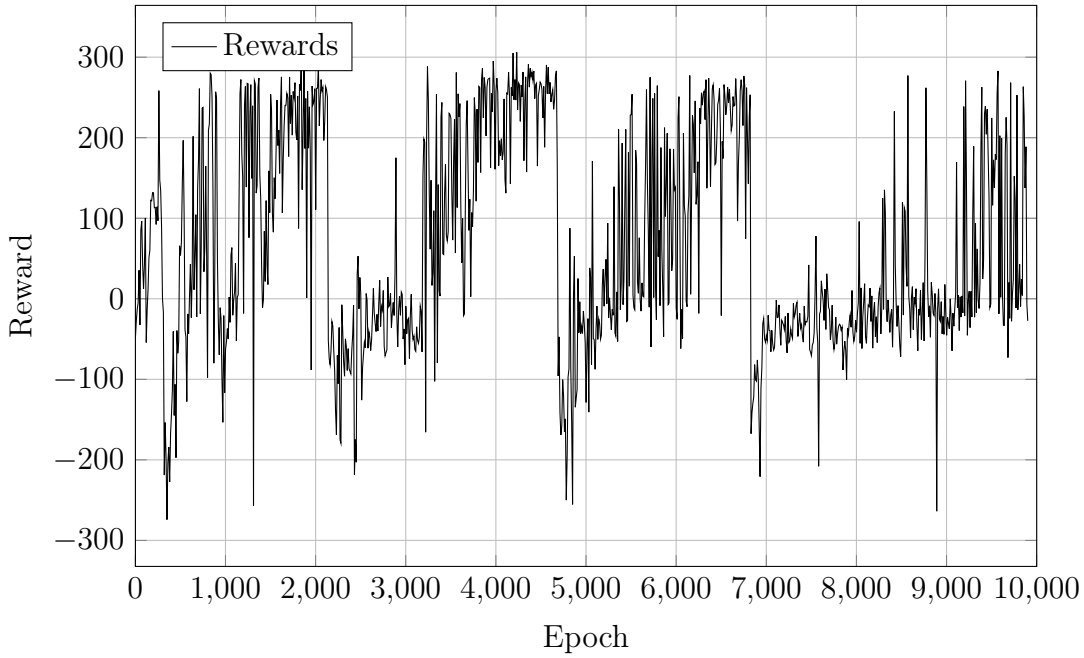

Figure 15: Results for first iteration of investigated RLHF algorithm. Notable drops in results can be observed after the model resumes online training when offline training is finished in the loop. This led to a conclusion that offline learning should only be used during early stages of reinforcement learning for a specific model.

Initially to resolve the issue, multiple solutions were considered such as reducing the learning rate for offline learning, adjusting the epsilon values or training steps, however it became apparent that offline training each time refits the neural network's weights to match the specific dataset used, overall making it a non-viable strategy for later stages of training.

With these observations, the number of human interventions was reduced to three and offline training was performed early in the process, to allow the agent unlearn any undesirable behaviours early in the training process (if there were any) while the exploration is still the primary strategy.

4.4.7 Finalizing Parameters

One of the main challenges for this experiment was to determine the correct ratio of online and offline training, and identifying the appropriate timing for each. Too much offline training can lead to model being overfitted, but not enough training means that the model will be unable to learn faster than the previous iterations (Experiment 2 and Experiment 1).

Through trial and error it was decided to train the model for 25000 steps before each human evaluation, so it can achieve reasonable deviations from the offline fitted model. With a total of 75000 steps the human evaluation occurs very early in the training.

Given the model's extensive offline training experience, it was also necessary to adjust the parameters of epsilon-greedy strategy. This adjustment was made to facilitate a faster increase in rewards. In Experiment 2 the epsilon decreased from 0.7 to 0.1 during 1 million steps, we reduced it to 700 thousand steps (30% difference) to allow the model to finalize the exploration stage more early therefore potentially achieve better results in a shorter time.

4.4.8 Experiment 3 Results

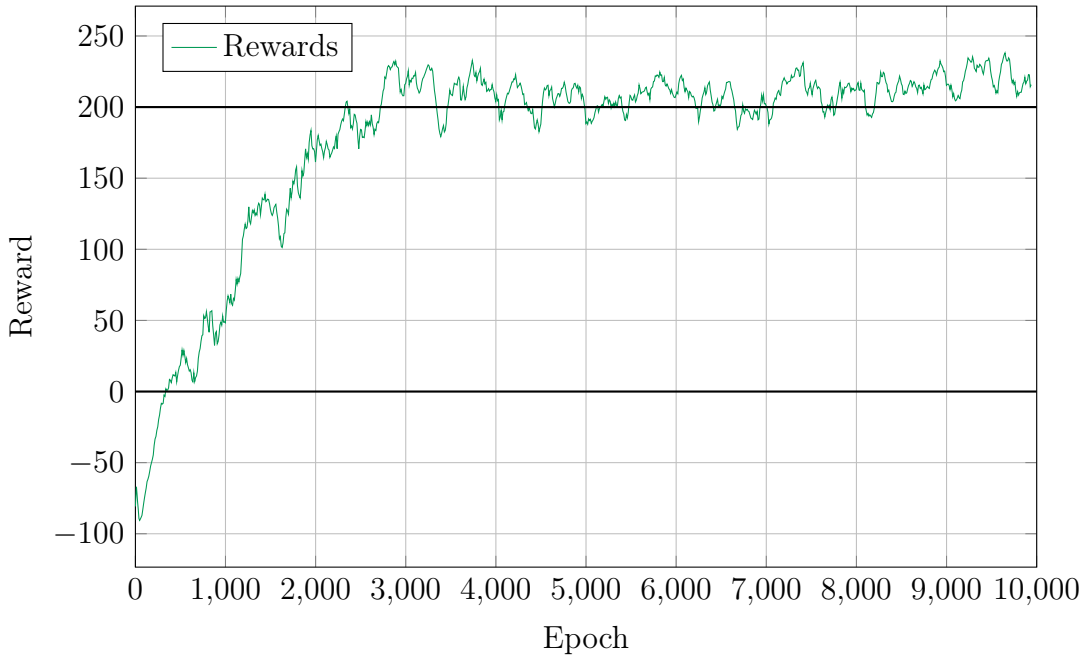

Figure 16: Experiment 3 training results. The graph represents the rewards earned during training using the second experiment. Each epoch corresponds to 1000 steps. A smoothing factor of 0.97 is applied to the data.

Benchmark review for 5 consecutive run average:

1. 0 reward - 0.3 million timesteps.
2. 200 - 2.7 million timesteps.
3. Overall the reward curve is very similar to Experiment 2, but with a steeper initial curve.

These results suggest that the adjustments made in Experiment 3 succeeded in accelerating the initial learning process, as evidenced by the more rapid increase in rewards.

4.5 All Experiment Comparison

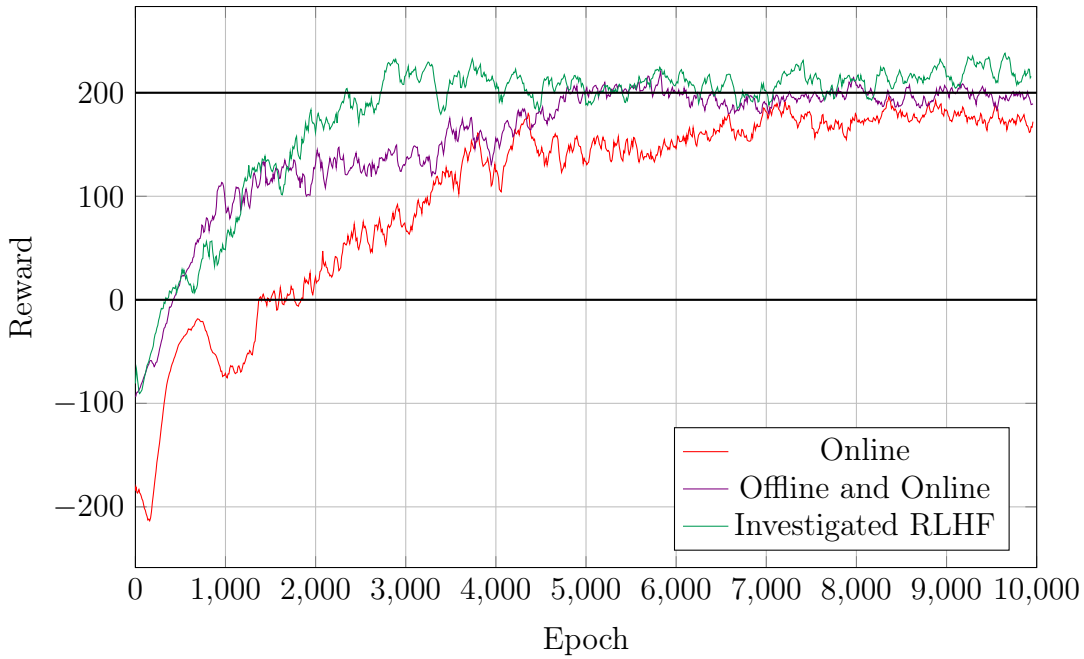

Figure 17: Comparison of training results across all experiments. Each epoch corresponds to 1000 steps. A smoothing factor of 0.97 is applied to the data.

Figure 17 presents a comparison of the training results across all three experiments. Overall, Experiment 3, which incorporated our proposed RLHF strategy, was the most successful in terms of online training speed. Not only did the model reach a reward of 200 faster than in the other two experiments, but the graph for Experiment 3 also mostly remained above the other two throughout the training process.

However, it is important to note that while the online training speed was improved for both Experiments 2 and 3, these experiments also incorporated offline training steps at the outset. Furthermore, the RLHF strategy, which yielded the best performance, also required human interaction during the initial phases of training.

Let's consider the results in relation to training time:

Algorithm	**Online steps to 200 reward**	**Total training time**
Online	Did not reach	20 h
Offline and Online	4.8 million	23 h
Investigates RLHF	2.7 million	27 h

Table 1: Reward data was smoothed using a factor of 0.97 as in previous diagrams.

It should be noted that the training time for the RLHF implementation was calculated assuming that human evaluation occurred immediately when prompted.

Results

The primary objective of this thesis - to implement a reinforcement learning algorithm that incorporates human feedback - was successfully achieved. The specific goals that were accomplished are as follows:

- An extensive review of the literature on Reinforcement learning (RL) and reinforcement learning from human feedback (RLHF) literature was conducted. After identifying the original algorithm a modified version of the RLHF algorithm was developed.
- We proposed and empirically evaluated two different RL methodologies: the traditional RL and our version of RLHF, in a selected environment - the OpenAI Gym Lunar Lander.
- The performance of all three different models was compared: the original online RL model did not reach the reward of 200, offline and online RL model model took 4.8 million timesteps to reach it, and our implementation of RLHF took 2.7 million timesteps (56% faster than offline and online RL model).

Conclusions

- Although the human feedback interruption loop strategy offers an advantage in online training speed, it takes much longer time to train the models overall and the algorithm is more complicated to execute. Implementing and adapting the algorithm is challenging due to multiple types of hyperparameters and the additional code required.
- Our study demonstrates that our implementation of the RLHF algorithm can effectively solve reinforcement learning problems in standard environments, such as the ones provided in OpenAI Gym.
- The modifications proposed to the original online RL training paradigm, including offline training, combined online and offline training, and RLHF, significantly improved model training speed (reduced required episode count). Modifications to the traditional online RL training paradigm improved the result.

References

[ACT19] Kai Arulkumaran, Antoine Cully, and Julian Togelius. AlphaStar. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, jul 2019.

[ADBB17] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, nov 2017.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. 2016.

[BPW+12] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1:1–43, 03 2012.

[CLB+17] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. 2017.

[Cou01] Jane Courtney. Application of digital image processing to marker-free analysis of human gait. 01 2001.

[DNP13] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A survey on policy search for robotics. *Foundations and Trends® in Robotics*, 2(1–2):1–142, 2013.

[Han18] Xintian Han. A mathematical introduction to reinforcement learning. 2018.

[HHF+19] Honglan Huang, Jincai Huang, Yanghe Feng, Jiarui Zhang, Zhong Liu, Qi Wang, and Li Chen. On the improvement of reinforcement active learning with the involvement of cross entropy to address one-shot learning problem. *PLOS ONE*, 14(6):1–17, 06 2019.

[KCB+19] Matthieu Komorowski, Leo Celi, Omar Badawi, Anthony Gordon, and Aldo Faisal. Understanding the artificial intelligence clinician and optimal treatment strategies for sepsis in intensive care. 03 2019.

[LKTF20] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. 2020.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[PMC23] Rafael Figueiredo Prudencio, Marcos R. O. A. Maximo, and Esther Luna Colombini. A survey on offline reinforcement learning: Taxonomy, review, and open problems. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–0, 2023.

[SB14] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2014.

[SH23] Sakib Shahriar and Kadhim Hayawi. Let's have a chat! a conversation with chatgpt: Technology, applications, and limitations, 2023.

[SI22] Takuma Seno and Michita Imai. d3rlpy: An offline deep reinforcement learning library. 2022.

[Wat89] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford, 1989.

[ZTS+21] Stephan Zheng, Alexander Trott, Sunil Srinivasa, David C. Parkes, and Richard Socher. The ai economist: Optimal economic policy design via two-level deep reinforcement learning, 2021.