arXiv:1912.02877v1 [cs.LG] 5 Dec 2019

Training Agents using Upside-Down Reinforcement Learning

Rupesh Kumar Srivastava[1*] Pranav Shyam[1†] Filipe Mutz[2‡]
Wojciech Jaśkowski[1] Jürgen Schmidhuber[12]

[1]NNAISENSE
[2]The Swiss AI Lab IDSIA

NNAISENSE Technical Report

Abstract

Traditional Reinforcement Learning (RL) algorithms either predict rewards with value functions or maximize them using policy search. We study an alternative: Upside-Down Reinforcement Learning (Upside-Down RL or ⅂Ꙅ), that solves RL problems primarily using supervised learning techniques. Many of its main principles are outlined in a companion report [34]. Here we present the first concrete implementation of ⅂Ꙅ and demonstrate its feasibility on certain episodic learning problems. Experimental results show that its performance can be surprisingly competitive with, and even exceed that of traditional baseline algorithms developed over decades of research.

1 Introduction

While there is a rich history of techniques that incorporate supervised learning (SL) into reinforcement learning (RL) algorithms, it is believed that fully solving RL problems using SL is not possible, because feedback from the environment provides *error signals* in SL but *evaluation signals* in RL [2, 30]. Put simply, an agent gets feedback about how useful its actions are, but not about which actions are the best to take in any situation. On the possibility of turning an RL problem into an SL problem, Barto and Dietterich [2] surmised: "In general, there is no way to do this."

In a companion technical report, Schmidhuber [34] proposes to bridge this gap between SL and RL through *Upside-Down Reinforcement Learning* (⅂Ꙅ), where environmental feedback – such as the *reward* – is an input rather than the learning target as in traditional RL algorithms based on reward prediction [37]. Here we develop a practical ⅂Ꙅ algorithm for episodic tasks and show that it is indeed possible to train agents in general model-free settings[1] without using value-based algorithms such as Q-learning [41], or policy-based ones such as policy gradients and evolutionary algorithms [22, 42]. Instead, ⅂Ꙅ uses pure SL to train an agent on all past experiences, and sidesteps the issues arising from the combination of function approximation, bootstrapping and off-policy training [37]. We first describe its basic principles, then experimentally demonstrate its practical feasibility on three RL problems with both sparse and dense reward structure.

*Correspondence to: rupesh@nnaisense.com
†Now at OpenAI.
‡Now at IFES, Brazil.
[1]Stochastic environments with high-dimensional inputs, scalar and possibly sparse rewards, no expert demonstrations.

9102 ceD 5] GL . sc [1v7782 0.2191 : vi Xra

使用颠倒强化学习训练智能体

Rupesh Kumar Srivastava[1*] Pranav Shyam[1†] Filipe Mutz[2‡] Wojciech Ja kowski[1] Jürgen Schmidhuber[12]

1NNAISENSE [2]瑞士AI实验室IDSIA

NNAISENSE Technical Report

摘要

传统的强化学习（RL）算法要么使用价值函数预测奖励，要么使用策略搜索最大化奖励。我们研究了另一种方法：倒置强化学习（Upside-Down RL 或 RL），它主要使用监督学习技术来解决RL问题。其许多主要原则在一份配套报告[34]中有所概述。在这里，我们首次提出了RL的具体实现，并在某些 episodic 学习问题上展示了其可行性。实验结果表明，其性能出人意料地具有竞争力，甚至可以超过几十年研究中开发的传统基线算法。

1 引言

尽管存在将监督学习（SL）纳入强化学习（RL）算法中的丰富技术历史，但人们认为完全使用SL解决RL问题是不可能的，因为环境反馈在SL中提供了错误信号，在RL中提供了评估信号[2, 30]。简单来说，智能体收到关于其行为有用性的反馈，但不知道在任何情况下哪种行为是最好的。关于将RL问题转换为SL问题的可能性，Barto和Dietterich[2]推测：“一般来说，这是不可能的。”

在一篇配套的技术报告中，Schmidhuber [34] 提出通过上下反转强化学习（RL）来弥合监督学习（SL）和强化学习（RL）之间的差距，其中环境反馈——例如奖励——是输入而不是传统基于奖励预测的RL算法中的学习目标[37]。在这里，我们开发了一种适用于 episodic 任务的实用 RL 算法，并展示了确实可以在一般无模型设置[1]中训练代理，而无需使用基于值的方法如 Q 学习[41]，或基于策略的方法如策略梯度和进化算法[22, 42]。相反，RL 使用纯粹的监督学习来训练代理所有过往的经验，并绕过了函数逼近、自举和离策训练带来的问题[37]。我们首先描述其基本原理，然后通过三个具有稀疏和密集奖励结构的RL问题的实验演示其实际可行性。

*Correspondence to: rupesh@nnaisense.com
†Now at OpenAI.
‡Now at IFES, Brazil.
[1]Stochastic environments with high-dimensional inputs, scalar and possibly sparse rewards, no expert demonstrations.

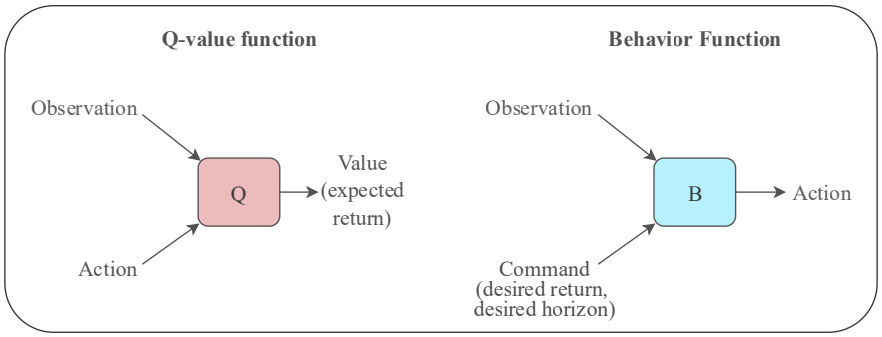


Figure 1: A key distinction between the action-value function (Q) in traditional RL (e.g. Q-learning) and the behavior function (B) in ƬƦ is that the roles of actions and returns are switched. In addition, B may have other command inputs such as desired states or the desired time horizon for achieving a desired return.

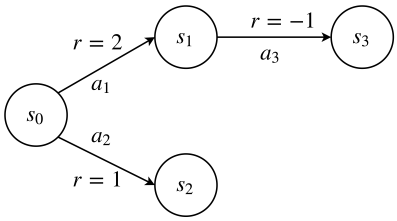


Figure 2: A toy environment with four discrete states.

Table 1: A behavior function for the toy environment.

State	Desired Return	Desired Horizon	Action
s_0	2	1	a_1
s_0	1	1	a_2
s_0	1	2	a_1
s_1	-1	1	a_3

2 Upside-Down Reinforcement Learning

2.1 Terminology & Notation

In what follows, s, a and r denote *state*, *action*, and *reward* respectively. The sets of values of s and a ($\mathcal{S}$ and $\mathcal{A}$) depend on the environment. Right subscripts denote time indices (e.g. $s_t, t \in \mathbb{N}^0$). We consider the Markovian environments with scalar rewards ($r \in \mathbb{R}$) as is typical, but the general principles of ƬƦ are not limited to these settings. A *policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$ is a function that selects an action in a given state. A policy can be stochastic, in which case it maps a state to a probability distribution over actions. Each *episode* consists of an agent's interaction with the environment starting in an initial state and ending in a terminal state while following any policy. A *trajectory* τ is the sequence $\langle (s_t, a_t, r_t, s_{t+1}) \rangle, t = 0, \ldots, T-1$ containing data describing an episode of length T. We refer to any subsequence of a trajectory as a *segment* or a *behavior*, and the cumulative reward over a segment as the *return*.

2.2 Knowledge Representation

Traditional model-free RL algorithms can be broadly classified as being *value-based* or *policy-based*. The core principle of value-based algorithms is reward prediction: agents are trained to predict the expected discounted future return for taking any action in any state, commonly using TD learning. Policy-based algorithms are instead based on directly searching for policies that maximize returns. The basic principle of ƬƦ are different from both of these categories: given a particular definition of *commands*, it defines a *behavior function* that encapsulates knowledge about the behaviors observed so far compatible with known commands. The nature of the behavior function is explained using two examples below.

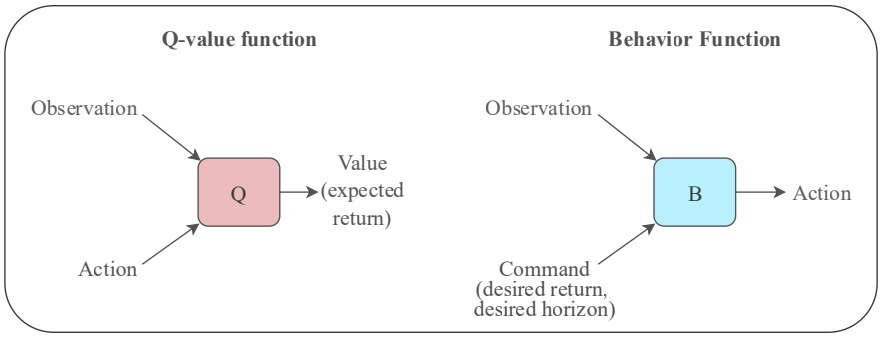


Figure 1: A key distinction between the action-value function (Q) in traditional RL (e.g. Q-learning) and the behavior function (B) in ƬƦ is that the roles of actions and returns are switched. In addition, B may have other command inputs such as desired states or the desired time horizon for achieving a desired return.

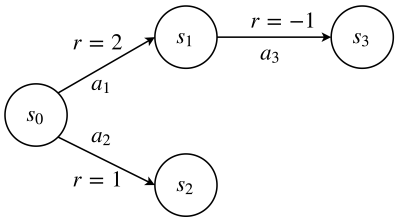


Figure 2: A toy environment with four discrete states.

Table 1: A behavior function for the toy environment.

State	Desired Return	Desired Horizon	Action
s_0	2	1	a_1
s_0	1	1	a_2
s_0	1	2	a_1
s_1	-1	1	a_3

2 Upside-Down Reinforcement Learning

2.1 Terminology & Notation

In what follows, s, a and r denote *state*, *action*, and *reward* respectively. The sets of values of s and a ($\mathcal{S}$ and $\mathcal{A}$) depend on the environment. Right subscripts denote time indices (e.g. $s_t, t \in \mathbb{N}^0$). We consider the Markovian environments with scalar rewards ($r \in \mathbb{R}$) as is typical, but the general principles of ƬƦ are not limited to these settings. A *policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$ is a function that selects an action in a given state. A policy can be stochastic, in which case it maps a state to a probability distribution over actions. Each *episode* consists of an agent's interaction with the environment starting in an initial state and ending in a terminal state while following any policy. A *trajectory* τ is the sequence $\langle (s_t, a_t, r_t, s_{t+1}) \rangle, t = 0, \ldots, T-1$ containing data describing an episode of length T. We refer to any subsequence of a trajectory as a *segment* or a *behavior*, and the cumulative reward over a segment as the *return*.

2.2 Knowledge Representation

Traditional model-free RL algorithms can be broadly classified as being *value-based* or *policy-based*. The core principle of value-based algorithms is reward prediction: agents are trained to predict the expected discounted future return for taking any action in any state, commonly using TD learning. Policy-based algorithms are instead based on directly searching for policies that maximize returns. The basic principle of ƬƦ are different from both of these categories: given a particular definition of *commands*, it defines a *behavior function* that encapsulates knowledge about the behaviors observed so far compatible with known commands. The nature of the behavior function is explained using two examples below.

Illustrative Example 1. Consider a simple dart-throwing environment where each episode lasts a single step. An agent learning to throw darts receives a return inversely proportional to the hit distance from the center of the board. In each episode, the agent observes the initial state of the dart, and takes an action that determines the force and direction of the throw. Using value-based RL for this task would amount to training the agent to predict the expected return for various actions and initial states. This knowledge would then be used for action selection e.g. taking the action with the highest expected return.

In ꓤR, the agent's knowledge is represented not in terms of expected returns for various states and actions, but in terms of actions that are compatible with various states and desired returns i.e. the inputs and targets of the agent's learning procedure are switched. The dart throwing agent would be trained to directly produce the actions for hitting desired locations on the board, using a behavior function B[2] learned using its past experience. Figure 1 schematically illustrates this difference between B and the Q-value function commonly used in value-based RL. Since this environment consists of episodes with a single time step, both Q and B can be learned using SL. The next example illustrates a slightly more typical RL setting with longer time horizons.

Illustrative Example 2. Consider the simple deterministic Markovian environment in Figure 2 in which all trajectories start in s_0 or s_1 and end in s_2 or s_3. Additionally consider *commands* of the type: achieve a given desired return in a given desired horizon from the current state. A behavior function based on the set of all unique behaviors possible in this environment can then be expressed in a tabular form in Table 1. It maps states and commands to the action to be taken in that state compatible with executing the command. In other words, it answers the question: "if an agent is in a given state and desires a given return over a given horizon, which action should it take next?" By design, this function can now be used to execute any valid command in the environment without further knowledge.

Two properties of the behavior function are notable. First, the output of B can be stochastic even in a deterministic environment since there may be multiple valid behaviors compatible with the same command and state. For example, this would be the case if the transition $s_0 \rightarrow s_2$ had a reward of 2. So in general, B produces a probability distribution over actions. Second, B fundamentally depends on the set of trajectories used to construct it. Using a loss function L, we define the optimal behavior function $B^*_{\mathcal{T}}$ for a set of trajectories $\mathcal{T}$ as

$$
\begin{aligned}
B^*_{\mathcal{T}} = \arg\min_{B} \sum_{(t_1,t_2)} & L(B(s_{t_1}, d^r, d^h), a_{t_1}), \\
& \text{where } 0 < t_1 < t_2 < \text{len}(\tau)\ \forall\, \tau \in \mathcal{T}, \\
& d^r = \sum_{t=t_1}^{t_2} r_t \text{ and } d^h = t_2 - t_1.
\end{aligned} \tag{1}
$$

Here $\text{len}(\tau)$ is the length of any trajectory τ. For a suitably parameterized B, we use the cross-entropy between the observed and predicted distributions of actions as the loss function. Equivalently, we search for parameters that maximize the likelihood that the behavior function generates the available data, using the traditional tools of supervised learning. Similarly, we can define a behavior function *over a policy*. Instead of a set of trajectories, B^*_{π} minimizes the same loss over the distribution of trajectories generated when acting according to π.

2.3 An ꓤR Algorithm for Maximizing Episodic Returns

In principle, a behavior function can be learned for any policy that generates all possible trajectories in an environment given sufficient time (e.g. a random policy) and then used to select actions that lead to any desired return in a desired horizon achievable in the environment. But such a learning procedure is not practical since it relies on undirected exploration using a fixed policy. Moreover, in environments with scalar rewards, the goal is to learn to achieve high

[2]Denoted C by Schmidhuber [34]. We use B here for compatibility with the "behavior function" nomenclature.

Illustrative Example 1. Consider a simple dart-throwing environment where each episode lasts a single step. An agent learning to throw darts receives a return inversely proportional to the hit distance from the center of the board. In each episode, the agent observes the initial state of the dart, and takes an action that determines the force and direction of the throw. Using value-based RL for this task would amount to training the agent to predict the expected return for various actions and initial states. This knowledge would then be used for action selection e.g. taking the action with the highest expected return.

In ꓤR, the agent's knowledge is represented not in terms of expected returns for various states and actions, but in terms of actions that are compatible with various states and desired returns i.e. the inputs and targets of the agent's learning procedure are switched. The dart throwing agent would be trained to directly produce the actions for hitting desired locations on the board, using a behavior function B[2] learned using its past experience. Figure 1 schematically illustrates this difference between B and the Q-value function commonly used in value-based RL. Since this environment consists of episodes with a single time step, both Q and B can be learned using SL. The next example illustrates a slightly more typical RL setting with longer time horizons.

Illustrative Example 2. Consider the simple deterministic Markovian environment in Figure 2 in which all trajectories start in s_0 or s_1 and end in s_2 or s_3. Additionally consider *commands* of the type: achieve a given desired return in a given desired horizon from the current state. A behavior function based on the set of all unique behaviors possible in this environment can then be expressed in a tabular form in Table 1. It maps states and commands to the action to be taken in that state compatible with executing the command. In other words, it answers the question: "if an agent is in a given state and desires a given return over a given horizon, which action should it take next?" By design, this function can now be used to execute any valid command in the environment without further knowledge.

Two properties of the behavior function are notable. First, the output of B can be stochastic even in a deterministic environment since there may be multiple valid behaviors compatible with the same command and state. For example, this would be the case if the transition $s_0 \rightarrow s_2$ had a reward of 2. So in general, B produces a probability distribution over actions. Second, B fundamentally depends on the set of trajectories used to construct it. Using a loss function L, we define the optimal behavior function $B^*_{\mathcal{T}}$ for a set of trajectories $\mathcal{T}$ as

$$
\begin{aligned}
B^*_{\mathcal{T}} = \arg\min_{B} \sum_{(t_1,t_2)} & L(B(s_{t_1}, d^r, d^h), a_{t_1}), \\
& \text{where } 0 < t_1 < t_2 < \text{len}(\tau)\ \forall\, \tau \in \mathcal{T}, \\
& d^r = \sum_{t=t_1}^{t_2} r_t \text{ and } d^h = t_2 - t_1.
\end{aligned} \tag{1}
$$

Here $\text{len}(\tau)$ is the length of any trajectory τ. For a suitably parameterized B, we use the cross-entropy between the observed and predicted distributions of actions as the loss function. Equivalently, we search for parameters that maximize the likelihood that the behavior function generates the available data, using the traditional tools of supervised learning. Similarly, we can define a behavior function *over a policy*. Instead of a set of trajectories, B^*_{π} minimizes the same loss over the distribution of trajectories generated when acting according to π.

2.3 An ꓤR Algorithm for Maximizing Episodic Returns

In principle, a behavior function can be learned for any policy that generates all possible trajectories in an environment given sufficient time (e.g. a random policy) and then used to select actions that lead to any desired return in a desired horizon achievable in the environment. But such a learning procedure is not practical since it relies on undirected exploration using a fixed policy. Moreover, in environments with scalar rewards, the goal is to learn to achieve high

[2]Denoted C by Schmidhuber [34]. We use B here for compatibility with the "behavior function" nomenclature.

Algorithm 1 Upside-Down Reinforcement Learning: High-level Description.

```
Initialize replay buffer with warm-up episodes using random actions        // Section 2.3.1
Initialize a behavior function                                              // Section 2.3.2
while stopping criteria is not reached do
    Improve the behavior function by training on replay buffer              // Exploit; Section 2.3.3
    Sample exploratory commands based on replay buffer                      // Section 2.3.4
    Generate episodes using Algorithm 2 and add to replay buffer            // Explore; Section 2.3.5
    if evaluation required then
        Evaluate current agent using Algorithm 2                            // Section 2.3.6
    end if
end while
```

returns and not to achieve any possible return over any horizon. Therefore, the concrete algorithm used in this paper trains a behavior function on the set of trajectories (or the agent's experience) so far and incorporates minimal additions that enable the continual collection of trajectories with higher returns.

High-level pseudo-code for the proposed algorithm is described in Algorithm 1. It starts by initializing an empty replay buffer to collect the agent's experiences during training, and filling it with a few episodes of random interactions. The behavior function of the agent is continually improved by supervised training on previous experiences recorded in the replay buffer. After each training phase, the behavior function is used to act in the environment to obtain new experiences that are added to the replay buffer. This procedure continues until a stopping criterion is met, such as reaching the allowed maximum number of interactions with the environment. The remainder of this section describes each step of the algorithm and introduces the hyperparameters. A concise list of hyperparameters is also provided in Appendix A.

2.3.1 Replay Buffer

ꓤR does not explicitly maximize returns, but instead relies on exploration to continually discover higher return trajectories so that the behavior function can be trained on them. To drive learning progress, we found it helpful to use a replay buffer containing a fixed maximum number of trajectories with the highest returns seen so far, sorted in increasing order by return. The maximum buffer size is a hyperparameter. Since the agent starts learning with zero experience, an initial set of trajectories is generated by executing random actions in the environment. The trajectories are added to the replay buffer and used to start training the agent's behavior function.

2.3.2 Behavior Function

As described earlier, at any time t during an episode, the current behavior function B produces an action distribution in response to the current state s_t and command $c_t := (d_t^r, d_t^h)$, where $d_t^r \in \mathbb{R}$ is the *desired return* and $d_t^h \in \mathbb{N}$ is the *desired time horizon* at time t. The predicted action distribution $P(a_t|s_t, c_t) = B(s_t, c_t; \theta)$, where θ denotes a vector of trainable parameters, is expected to lead to successful execution of the command c_t interpreted as: "achieve a return d_t^r during the next d_t^h steps". For a given initial command input c_0, B can be used to generate a trajectory using Algorithm 2 by sampling actions predicted for the current command and updating the command according to the obtained rewards and elapsed time.

An important implementation detail is that d_t^h is always set to $\max(d_t^h, 1)$ such that it is a valid time horizon. Furthermore, d_t^r is clipped such that it is upper-bounded by the maximum return achievable in the environment. This only affects agent evaluations (not training) and avoids situations where negative rewards (r_t) can lead to desired returns that are not achievable from any state (see Algorithm 2; line 8).

Algorithm 1 Upside-Down Reinforcement Learning: High-level Description.

```
Initialize replay buffer with warm-up episodes using random actions        // Section 2.3.1
Initialize a behavior function                                              // Section 2.3.2
while stopping criteria is not reached do
    Improve the behavior function by training on replay buffer              // Exploit; Section 2.3.3
    Sample exploratory commands based on replay buffer                      // Section 2.3.4
    Generate episodes using Algorithm 2 and add to replay buffer            // Explore; Section 2.3.5
    if evaluation required then
        Evaluate current agent using Algorithm 2                            // Section 2.3.6
    end if
end while
```

returns and not to achieve any possible return over any horizon. Therefore, the concrete algorithm used in this paper trains a behavior function on the set of trajectories (or the agent's experience) so far and incorporates minimal additions that enable the continual collection of trajectories with higher returns.

High-level pseudo-code for the proposed algorithm is described in Algorithm 1. It starts by initializing an empty replay buffer to collect the agent's experiences during training, and filling it with a few episodes of random interactions. The behavior function of the agent is continually improved by supervised training on previous experiences recorded in the replay buffer. After each training phase, the behavior function is used to act in the environment to obtain new experiences that are added to the replay buffer. This procedure continues until a stopping criterion is met, such as reaching the allowed maximum number of interactions with the environment. The remainder of this section describes each step of the algorithm and introduces the hyperparameters. A concise list of hyperparameters is also provided in Appendix A.

2.3.1 Replay Buffer

ꓤR does not explicitly maximize returns, but instead relies on exploration to continually discover higher return trajectories so that the behavior function can be trained on them. To drive learning progress, we found it helpful to use a replay buffer containing a fixed maximum number of trajectories with the highest returns seen so far, sorted in increasing order by return. The maximum buffer size is a hyperparameter. Since the agent starts learning with zero experience, an initial set of trajectories is generated by executing random actions in the environment. The trajectories are added to the replay buffer and used to start training the agent's behavior function.

2.3.2 Behavior Function

As described earlier, at any time t during an episode, the current behavior function B produces an action distribution in response to the current state s_t and command $c_t := (d_t^r, d_t^h)$, where $d_t^r \in \mathbb{R}$ is the *desired return* and $d_t^h \in \mathbb{N}$ is the *desired time horizon* at time t. The predicted action distribution $P(a_t|s_t, c_t) = B(s_t, c_t; \theta)$, where θ denotes a vector of trainable parameters, is expected to lead to successful execution of the command c_t interpreted as: "achieve a return d_t^r during the next d_t^h steps". For a given initial command input c_0, B can be used to generate a trajectory using Algorithm 2 by sampling actions predicted for the current command and updating the command according to the obtained rewards and elapsed time.

An important implementation detail is that d_t^h is always set to $\max(d_t^h, 1)$ such that it is a valid time horizon. Furthermore, d_t^r is clipped such that it is upper-bounded by the maximum return achievable in the environment. This only affects agent evaluations (not training) and avoids situations where negative rewards (r_t) can lead to desired returns that are not achievable from any state (see Algorithm 2; line 8).

Algorithm 2 Generates an Episode using the Behavior Function.

Input: Initial command $c_0 = (d_0^r, d_0^h)$, Initial state s_0, Behavior function $B(;\theta)$
Output: Episode data E

$E \leftarrow \varnothing$
$t \leftarrow 0$
while episode is not over **do**
  Compute $P(a_t|s_t, c_t) = B(s_t, c_t; \theta)$
  Execute $a_t \sim P(a_t|s_t, c_t)$ to obtain reward r_t and next state s_{t+1} from the environment
  Append (s_t, a_t, r_t) to E
  $s_t \leftarrow s_{t+1}$ // Update state
  $d_t^r \leftarrow d_t^r - r_t$ // Update desired reward
  $d_t^h \leftarrow d_t^h - 1$ // Update desired horizon
  $c_t \leftarrow (d_t^r, d_t^h)$
  $t \leftarrow t + 1$
end while

2.3.3 Training the Behavior Function

As discussed in Section 2.2, B admits supervised training on a large amount of input-target examples from any past episode. The goal of training is to make the behavior function produce outputs consistent with all previously recorded trajectories in the replay buffer according to Equation 1.

To draw a training example from a random episode in the replay buffer, time step indices t_1 and t_2 are selected randomly such that $0 \leq t_1 < t_2 \leq T$, where T is the length of the selected episode. Then the input for training B is $(s_{t_1}, (d^r, d^h))$, where $d^r = \sum_{t=t_1}^{t_2} r_t$ and $d^h = t_2 - t_1$, and the target is a_{t_1}, the action taken at t_1. To summarize, the training examples are generated by selecting the time horizons, actions, observations and rewards in the past, and generating input-target pairs consistent with them.

Several heuristics may be used to select and combine training examples into mini-batches for gradient-based SL. For all experiments in this paper, only "trailing segments" were sampled from each episode, i.e., we set $t_2 = T - 1$ where T is the length of any episode. This discards a large amount of potential training examples but is a good fit for episodic tasks where the goal is to optimize the total reward until the end of each episode. It also makes training easier, since the behavior function only needs to learn to execute a subset of possible commands. To keep the setup simple, a fixed number of training iterations using Adam [13] were performed in each training step for all experiments.

2.3.4 Sampling Exploratory Commands

After each training phase, the agent can attempt to generate new, previously infeasible behavior, potentially achieving higher returns. To profit from such exploration through generalization, one must first create a set of new initial commands c_0 to be used in Algorithm 2. We use the following procedure to sample commands:

1. A number of episodes from the end of the replay buffer (i.e., with the highest returns) are selected. This number is a hyperparameter and remains fixed during training.
2. The exploratory desired horizon d_0^h is set to the mean of the lengths of the selected episodes.
3. The exploratory desired returns d_0^r are sampled from the uniform distribution $\mathcal{U}[M, M + S]$ where M is the mean and S is the standard deviation of the selected episodic returns.

This procedure was chosen due to its simplicity and ability to adjust the strategy using a single hyperparameter. Intuitively, it tries to generate new behavior (aided by environmental stochasticity) that achieves returns at the edge of the best known behaviors in the replay. While Schmidhuber [34] notes that a variety of heuristics may be used here,

Algorithm 2 Generates an Episode using the Behavior Function.

Input: Initial command $c_0 = (d_0^r, d_0^h)$, Initial state s_0, Behavior function $B(;\theta)$
Output: Episode data E

$E \leftarrow \varnothing$
$t \leftarrow 0$
while episode is not over **do**
  Compute $P(a_t|s_t, c_t) = B(s_t, c_t; \theta)$
  Execute $a_t \sim P(a_t|s_t, c_t)$ to obtain reward r_t and next state s_{t+1} from the environment
  Append (s_t, a_t, r_t) to E
  $s_t \leftarrow s_{t+1}$ // Update state
  $d_t^r \leftarrow d_t^r - r_t$ // Update desired reward
  $d_t^h \leftarrow d_t^h - 1$ // Update desired horizon
  $c_t \leftarrow (d_t^r, d_t^h)$
  $t \leftarrow t + 1$
end while

2.3.3 Training the Behavior Function

As discussed in Section 2.2, B admits supervised training on a large amount of input-target examples from any past episode. The goal of training is to make the behavior function produce outputs consistent with all previously recorded trajectories in the replay buffer according to Equation 1.

To draw a training example from a random episode in the replay buffer, time step indices t_1 and t_2 are selected randomly such that $0 \leq t_1 < t_2 \leq T$, where T is the length of the selected episode. Then the input for training B is $(s_{t_1}, (d^r, d^h))$, where $d^r = \sum_{t=t_1}^{t_2} r_t$ and $d^h = t_2 - t_1$, and the target is a_{t_1}, the action taken at t_1. To summarize, the training examples are generated by selecting the time horizons, actions, observations and rewards in the past, and generating input-target pairs consistent with them.

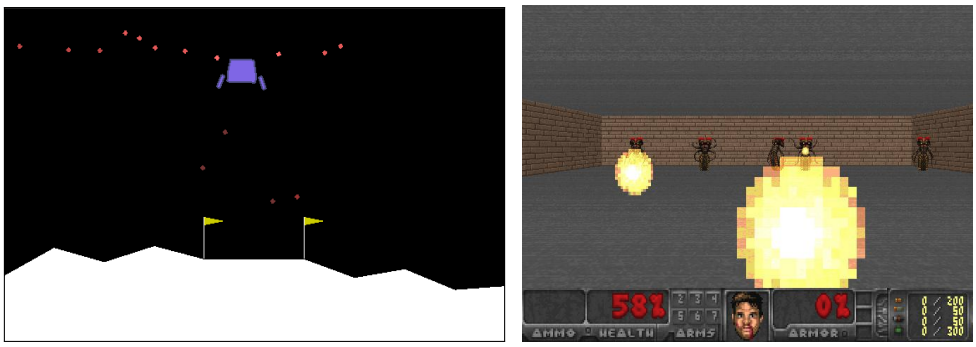
Several heuristics may be used to select and combine training examples into mini-batches for gradient-based SL. For all experiments in this paper, only "trailing segments" were sampled from each episode, i.e., we set $t_2 = T - 1$ where T is the length of any episode. This discards a large amount of potential training examples but is a good fit for episodic tasks where the goal is to optimize the total reward until the end of each episode. It also makes training easier, since the behavior function only needs to learn to execute a subset of possible commands. To keep the setup simple, a fixed number of training iterations using Adam [13] were performed in each training step for all experiments.

2.3.4 Sampling Exploratory Commands

After each training phase, the agent can attempt to generate new, previously infeasible behavior, potentially achieving higher returns. To profit from such exploration through generalization, one must first create a set of new initial commands c_0 to be used in Algorithm 2. We use the following procedure to sample commands:

1. A number of episodes from the end of the replay buffer (i.e., with the highest returns) are selected. This number is a hyperparameter and remains fixed during training.
2. The exploratory desired horizon d_0^h is set to the mean of the lengths of the selected episodes.
3. The exploratory desired returns d_0^r are sampled from the uniform distribution $\mathcal{U}[M, M + S]$ where M is the mean and S is the standard deviation of the selected episodic returns.

This procedure was chosen due to its simplicity and ability to adjust the strategy using a single hyperparameter. Intuitively, it tries to generate new behavior (aided by environmental stochasticity) that achieves returns at the edge of the best known behaviors in the replay. While Schmidhuber [34] notes that a variety of heuristics may be used here,



(a) LunarLander-v2 (b) TakeCover-v0

Figure 3: Test environments. In LunarLander-v2, the agent does not observe the visual representation, but an 8-dimensional state vector instead. In TakeCover-v0, the agent observes a down-sampled gray-scale visual inputs.

in practice it is very important to select exploratory commands that lead to behavior that is meaningfully different from existing experience so that it drives learning progress. An inappropriate exploration strategy can lead to very slow or stalled learning.

2.3.5 Generating Experience

Once the exploratory commands are sampled, it is straightforward to generate new exploratory episodes of interaction by using Algorithm 2, which works by repeatedly sampling from the action distribution predicted by the behavior function and updating its inputs for the next step. A fixed number of episodes are generated in each iteration of learning, and added to the replay buffer.

2.3.6 Evaluation

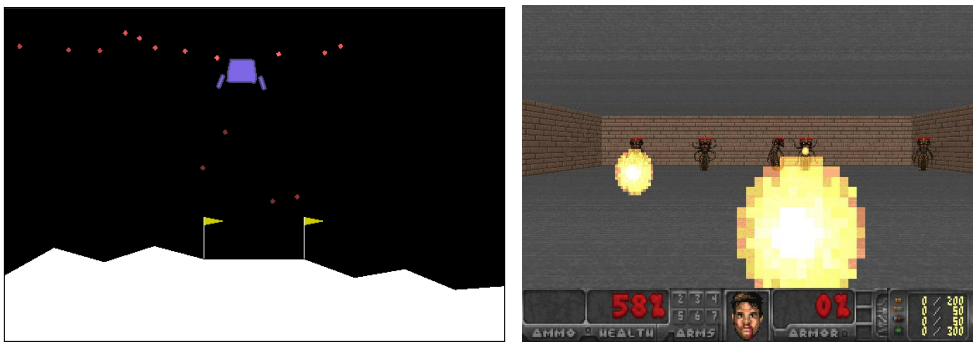
Algorithm 2 is also used to evaluate the agent at any time using evaluation commands derived from the most recent exploratory commands. The initial desired return d_0^r is set to the lower bound of the desired returns from the most recent exploratory command, and the initial desired horizon d_0^h from the most recent exploratory command is reused. In certain conditions, *greedy* actions – using the mode of the action distribution – can also be used, but we omit this option here for simplicity.

3 Experiments

The goal of our experiments was to determine the practical feasibility of TƎ and put its performance in context of two well-known traditional RL algorithms: Deep Q-Networks (DQN; 20) and Advantage Actor-Critic (A2C; synchronous version of the algorithm proposed by Mnih et al. [21]).

3.1 Environments

LunarLander-v2 (Figure 3a) is a simple Markovian environment available in the Gym RL library [4] where the objective is to land a spacecraft on a landing pad by controlling its main and side engines. During the episode the agent receives negative reward at each time step that decreases in magnitude the closer it gets to the optimal landing



(a) LunarLander-v2 (b) TakeCover-v0

Figure 3: Test environments. In LunarLander-v2, the agent does not observe the visual representation, but an 8-dimensional state vector instead. In TakeCover-v0, the agent observes a down-sampled gray-scale visual inputs.

in practice it is very important to select exploratory commands that lead to behavior that is meaningfully different from existing experience so that it drives learning progress. An inappropriate exploration strategy can lead to very slow or stalled learning.

2.3.5 Generating Experience

Once the exploratory commands are sampled, it is straightforward to generate new exploratory episodes of interaction by using Algorithm 2, which works by repeatedly sampling from the action distribution predicted by the behavior function and updating its inputs for the next step. A fixed number of episodes are generated in each iteration of learning, and added to the replay buffer.

2.3.6 Evaluation

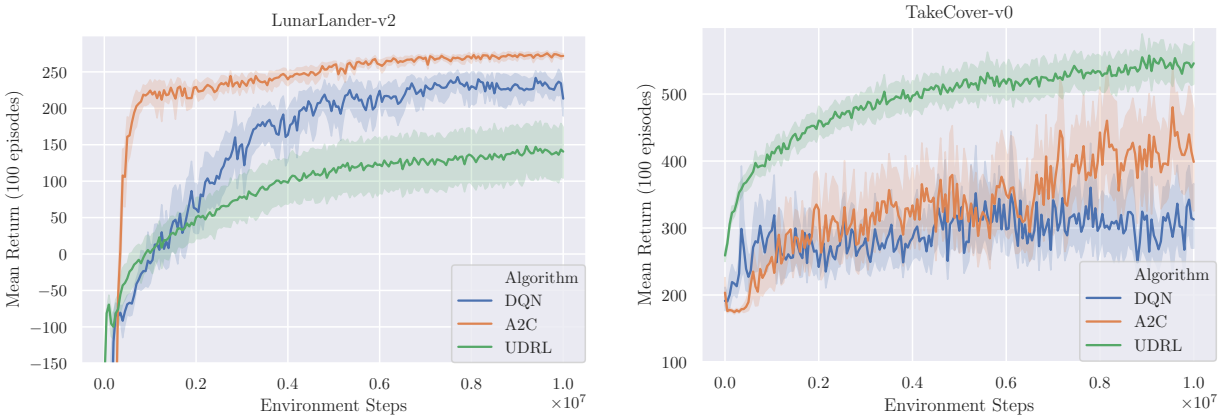
Algorithm 2 is also used to evaluate the agent at any time using evaluation commands derived from the most recent exploratory commands. The initial desired return d_0^r is set to the lower bound of the desired returns from the most recent exploratory command, and the initial desired horizon d_0^h from the most recent exploratory command is reused. In certain conditions, *greedy* actions – using the mode of the action distribution – can also be used, but we omit this option here for simplicity.

3 Experiments

The goal of our experiments was to determine the practical feasibility of TƎ and put its performance in context of two well-known traditional RL algorithms: Deep Q-Networks (DQN; 20) and Advantage Actor-Critic (A2C; synchronous version of the algorithm proposed by Mnih et al. [21]).

3.1 Environments

LunarLander-v2 (Figure 3a) is a simple Markovian environment available in the Gym RL library [4] where the objective is to land a spacecraft on a landing pad by controlling its main and side engines. During the episode the agent receives negative reward at each time step that decreases in magnitude the closer it gets to the optimal landing



(a) On LunarLander-v2, ƬƦ is able to train agents that land the spacecraft, but is beaten by traditional RL algorithms.

(b) On TakeCover-v0, ƬƦ is able to consistently yield high-performing agents, while outperforming DQN and A2C.

Figure 4: Evaluation results for LunarLander-v2 and TakeCover-v0. Solid lines represent the mean of evaluation scores over 20 runs using tuned hyperparameters and experiment seeds 1–20. Shaded regions represent 95% confidence intervals using 1000 bootstrap samples. Each evaluation score is a mean of 100 episode returns.

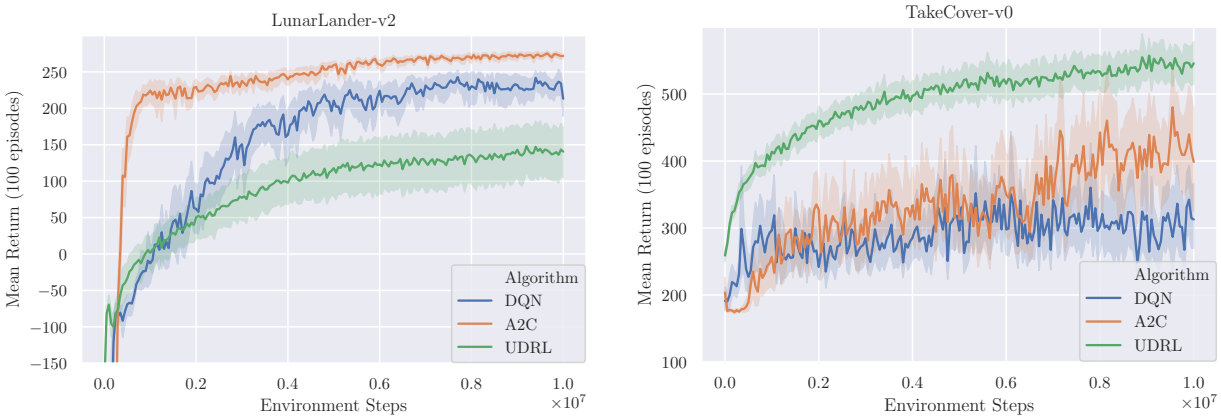
position in terms of both location and orientation. The reward at the end of the episode is -100 for crashing and +100 for successful landing. The agent receives eight-dimensional observations and can take one out of four actions.

TakeCover-v0 (Figure 3b) environment is part of the VizDoom library for visual RL research [12]. The agent is spawned next to the center of a wall in a rectangular room, facing the opposite wall where monsters randomly appear and shoot fireballs at the agent. It must learn to avoid fireballs by moving left or right to survive as long as possible. The reward is +1 for every time step that the agent survives, so for ƬƦ agents we always set the desired horizon to be the same as the desired reward, and convert any fractional values to integers. Technically, the agent has a non-Markovian interface to the environment, since it cannot see the entire opposite wall at all times. To reduce the degree of partial observability, the eight most recent visual frames are stacked together to produce the agent observations. The frames are also converted to gray scale, and downsampled from an original resolution of 160×120 to 32×32.

3.2 Setup

All agents were implemented using artificial neural networks. The behavior function for UDRL agents was implemented using fully-connected feed-forward networks for LunarLander-v2, and convolutional neural networks (CNNs; 16) for TakeCover-v0. The command inputs were scaled by a fixed scaling factor, transformed by a fully-connected sigmoidal layer, and then multiplied element-wise with an embedding of the observed inputs (after the first layer for fully-connected networks; after all convolutional layers for CNNs). Apart from this small modification regarding UDRL command inputs, the network architectures were identical for all algorithms.

All experiments were run for 10M environmental steps, with the agent being evaluated for 100 episodes at 50K step intervals. For each environment, random sampling was first used to find good hyperparameters for each algorithm and model based on final performance. With this configuration, final experiments were executed with 20 seeds (from 1 to 20) for each environment and algorithm. Random seeds for resetting the environments were sampled from [1M, 10M) for training, [0.5M, 1M) for evaluation during hyperparameter tuning, and [1, 0.5M) for final evaluation with the best hyperparameters. Details of the hyperparameter tuning procedure are provided in Appendix B.



(a) On LunarLander-v2, ƬƦ is able to train agents that land the spacecraft, but is beaten by traditional RL algorithms.

(b) On TakeCover-v0, ƬƦ is able to consistently yield high-performing agents, while outperforming DQN and A2C.

Figure 4: Evaluation results for LunarLander-v2 and TakeCover-v0. Solid lines represent the mean of evaluation scores over 20 runs using tuned hyperparameters and experiment seeds 1–20. Shaded regions represent 95% confidence intervals using 1000 bootstrap samples. Each evaluation score is a mean of 100 episode returns.

position in terms of both location and orientation. The reward at the end of the episode is -100 for crashing and +100 for successful landing. The agent receives eight-dimensional observations and can take one out of four actions.

TakeCover-v0 (Figure 3b) environment is part of the VizDoom library for visual RL research [12]. The agent is spawned next to the center of a wall in a rectangular room, facing the opposite wall where monsters randomly appear and shoot fireballs at the agent. It must learn to avoid fireballs by moving left or right to survive as long as possible. The reward is +1 for every time step that the agent survives, so for ƬƦ agents we always set the desired horizon to be the same as the desired reward, and convert any fractional values to integers. Technically, the agent has a non-Markovian interface to the environment, since it cannot see the entire opposite wall at all times. To reduce the degree of partial observability, the eight most recent visual frames are stacked together to produce the agent observations. The frames are also converted to gray scale, and downsampled from an original resolution of 160×120 to 32×32.

3.2 Setup

All agents were implemented using artificial neural networks. The behavior function for UDRL agents was implemented using fully-connected feed-forward networks for LunarLander-v2, and convolutional neural networks (CNNs; 16) for TakeCover-v0. The command inputs were scaled by a fixed scaling factor, transformed by a fully-connected sigmoidal layer, and then multiplied element-wise with an embedding of the observed inputs (after the first layer for fully-connected networks; after all convolutional layers for CNNs). Apart from this small modification regarding UDRL command inputs, the network architectures were identical for all algorithms.

All experiments were run for 10M environmental steps, with the agent being evaluated for 100 episodes at 50K step intervals. For each environment, random sampling was first used to find good hyperparameters for each algorithm and model based on final performance. With this configuration, final experiments were executed with 20 seeds (from 1 to 20) for each environment and algorithm. Random seeds for resetting the environments were sampled from [1M, 10M) for training, [0.5M, 1M) for evaluation during hyperparameter tuning, and [1, 0.5M) for final evaluation with the best hyperparameters. Details of the hyperparameter tuning procedure are provided in Appendix B.

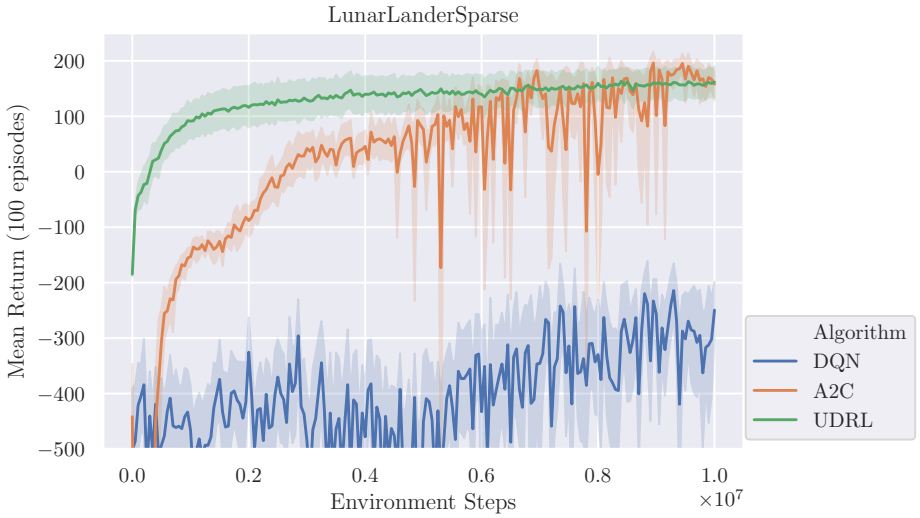


Figure 5: Results for LunarLanderSparse, a sparse reward version of LunarLander-v2 where the cumulative reward is delayed until the end of each episode. TƎ learns both much faster and more consistently than both DQN and A2C. Plot semantics are the same as Figure 4.

3.3 Results

The results of the final 20 runs are plotted in Figure 4, with dark lines showing the mean evaluation return and shaded regions indicating 95% confidence intervals with 1000 bootstrap samples.

For LunarLander-v2, a return of 100–140 indicates successful landing and returns above 200 are reached by close-to-optimal policies. TƎ lags behind DQN and A2C on this task. Inspection of the individual agents (for different seeds) showed that it is able to consistently train agents that land successfully, but while some agents learn quickly and achieve returns similar to A2C/DQN, some others plateau at lower returns.[3] We conjecture that this environment is rather suitable for TD learning by design due to its dense reward structure and large reward signals at the end.

For TakeCover-v0, the maximum possible return is 2100 due to episodic time limits. However, due to the difficulty of the environment (the number of monsters increases with time) and partial observability, the task is considered solved if the average reward over 100 episodes is greater than 750. On this task, TƎ comfortably outperforms both DQN and A2C, demonstrating its applicability to high-dimensional control problems.

3.4 Importance of Reward Structure: Sparse Lunar Lander

Is the better performance of DQN and A2C on LunarLander-v2 primarily because its reward function is more suitable for traditional algorithms? To answer this question, the setup was modified to accumulate all rewards until the end of each episode and provide them to the agent only at the last time step. The reward at all other time steps was zero, so that the total episode return remained the same. The evaluation study was repeated, including a hyperparameter search for all three algorithms, for the new *LunarLanderSparse* environment. The results for the final 20 runs are plotted in Figure 5.

TƎ agents (green) learned faster and more reliably with this reward function and outperformed DQN and A2C, both of which suffered from severe difficulties in dealing with delayed and sparse rewards. A2C could achieve high rewards on this task due to a large hyperparameter search, but its performance was both very sensitive to hyperparameter settings

[3]This highlights the importance of evaluating with a sufficiently large number of random seeds, without which TƎ can appear on par with DQN.

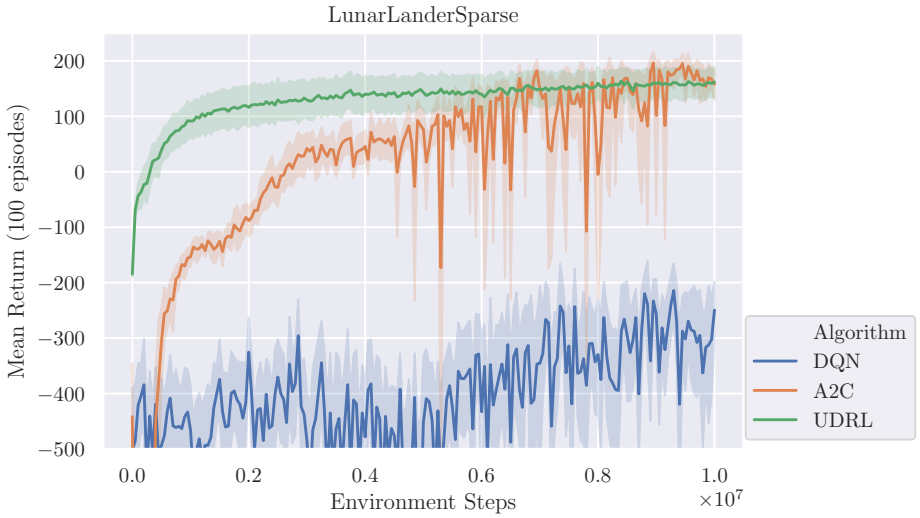


Figure 5: Results for LunarLanderSparse, a sparse reward version of LunarLander-v2 where the cumulative reward is delayed until the end of each episode. TƎ learns both much faster and more consistently than both DQN and A2C. Plot semantics are the same as Figure 4.

3.3 Results

The results of the final 20 runs are plotted in Figure 4, with dark lines showing the mean evaluation return and shaded regions indicating 95% confidence intervals with 1000 bootstrap samples.

For LunarLander-v2, a return of 100–140 indicates successful landing and returns above 200 are reached by close-to-optimal policies. TƎ lags behind DQN and A2C on this task. Inspection of the individual agents (for different seeds) showed that it is able to consistently train agents that land successfully, but while some agents learn quickly and achieve returns similar to A2C/DQN, some others plateau at lower returns.[3] We conjecture that this environment is rather suitable for TD learning by design due to its dense reward structure and large reward signals at the end.

For TakeCover-v0, the maximum possible return is 2100 due to episodic time limits. However, due to the difficulty of the environment (the number of monsters increases with time) and partial observability, the task is considered solved if the average reward over 100 episodes is greater than 750. On this task, TƎ comfortably outperforms both DQN and A2C, demonstrating its applicability to high-dimensional control problems.

3.4 Importance of Reward Structure: Sparse Lunar Lander

Is the better performance of DQN and A2C on LunarLander-v2 primarily because its reward function is more suitable for traditional algorithms? To answer this question, the setup was modified to accumulate all rewards until the end of each episode and provide them to the agent only at the last time step. The reward at all other time steps was zero, so that the total episode return remained the same. The evaluation study was repeated, including a hyperparameter search for all three algorithms, for the new *LunarLanderSparse* environment. The results for the final 20 runs are plotted in Figure 5.

TƎ agents (green) learned faster and more reliably with this reward function and outperformed DQN and A2C, both of which suffered from severe difficulties in dealing with delayed and sparse rewards. A2C could achieve high rewards on this task due to a large hyperparameter search, but its performance was both very sensitive to hyperparameter settings

[3]This highlights the importance of evaluating with a sufficiently large number of random seeds, without which TƎ can appear on par with DQN.

(compared to ⅂Ꙅ) and rather unstable during training. Overall, we find that ⅂Ꙅ is capable of training agents with both sparse and dense rewards (LunarLanderSparse and TakeCover-v0), but in some environments sparse rewards may work better than dense rewards. This counterintuitive property is an important avenue for future research.

The results above also lead us to a broader observation about RL benchmarks. The "actual" task in the LunarLander-v2 and LunarLanderSparse environments is exactly the same. Even the total episode return is the same for the same sequence of actions in the two environments by construction. Yet we obtain very different results from algorithms based on different principles simply based on the choice of the reward function. However, reward functions for benchmark problems are often developed side-by-side with existing algorithms and this can inadvertently favor certain learning paradigms over others. A potential way out of this bias is to evaluate each algorithm using a variety of reward functions for the same underlying RL task in order to understand its applicability to new RL problems.

3.5 Sensitivity of Trained Agents to Desired Returns

The ⅂Ꙅ objective trains the agent to achieve all known returns over known horizons, but the complete learning algorithm used in our experiments is designed to achieve higher returns as training progresses. This is done by keeping the highest return episodes in the replay buffer, and also biasing exploration towards higher returns. Nevertheless, at the end of training, the agents were found to able to exhibit large changes in behavior based on the level of initial desired episode return (d_0^r).

We evaluated agents at the end of training on all three environments by setting various values of d_0^r and plotting the obtained mean episode return over 100 episodes. The results are shown in Figure 6. Figures 6a and 6b show a strong correlation between obtained and desired returns for randomly selected agents on LunarLander-v2 and LunarLanderSparse. Note that in the later stages of training, the agents are only trained on episodes with returns close to the maximum.

Not all agents achieve such strong correspondence between desired and obtained returns. Figure 6c shows another agent that solves the task for most values of desired returns, and only achieves lower returns for very low values of desired returns. This indicates that stochasticity during training can affect how trained agents generalize to different commands, and suggests another direction for future investigation.

In the TakeCover-v0 environment, it is rather difficult to achieve precise values of desired returns. Stochasticity in the environment (the monsters appear randomly and shoot in random directions) and increasing difficulty over the episode imply that it is not possible to achieve lower returns than 200 and it gets considerably hard to achieve higher mean returns. The results in Figure 6d reflect these constraints, but still show that the agent is sensitive to the command inputs to some extent.

4 Related Work

Improving RL through SL has a long and rich history beyond the scope of this paper [33, 37]. For example, RL systems based on experience replay [18] attempt to leverage SL on past (off-policy) experience to improve value function approximation. Off-policy training can be augmented with goal-conditional value functions [11] (see also Pong et al. [27], Schaul et al. [31]) such that value functions for goals not being pursued by the current policy can also be updated based on the same interaction history. This idea was recently combined with experience replay by Andrychowicz et al. [1] and extended to policy gradients by Rauber et al. [28]. Oh et al. [23] proposed learning to imitate past good actions to augment actor-critic algorithms. Despite some differences, all of the above algorithms still rely on reward prediction using learned value functions, whereas ⅂Ꙅ uses neither.

There is also substantial prior work on learning reward or goal-conditional policies that directly produce actions [5, 6, 15, 35], but these rely either on a pre-trained world model, a dataset of goal and optimal policy parameters, or policy gradients for learning. While the behavior function in ⅂Ꙅ does bear a high-level similarity to goal-conditioned

(compared to ⅂Ꙅ) and rather unstable during training. Overall, we find that ⅂Ꙅ is capable of training agents with both sparse and dense rewards (LunarLanderSparse and TakeCover-v0), but in some environments sparse rewards may work better than dense rewards. This counterintuitive property is an important avenue for future research.

The results above also lead us to a broader observation about RL benchmarks. The "actual" task in the LunarLander-v2 and LunarLanderSparse environments is exactly the same. Even the total episode return is the same for the same sequence of actions in the two environments by construction. Yet we obtain very different results from algorithms based on different principles simply based on the choice of the reward function. However, reward functions for benchmark problems are often developed side-by-side with existing algorithms and this can inadvertently favor certain learning paradigms over others. A potential way out of this bias is to evaluate each algorithm using a variety of reward functions for the same underlying RL task in order to understand its applicability to new RL problems.

3.5 Sensitivity of Trained Agents to Desired Returns

The ⅂Ꙅ objective trains the agent to achieve all known returns over known horizons, but the complete learning algorithm used in our experiments is designed to achieve higher returns as training progresses. This is done by keeping the highest return episodes in the replay buffer, and also biasing exploration towards higher returns. Nevertheless, at the end of training, the agents were found to able to exhibit large changes in behavior based on the level of initial desired episode return (d_0^r).

We evaluated agents at the end of training on all three environments by setting various values of d_0^r and plotting the obtained mean episode return over 100 episodes. The results are shown in Figure 6. Figures 6a and 6b show a strong correlation between obtained and desired returns for randomly selected agents on LunarLander-v2 and LunarLanderSparse. Note that in the later stages of training, the agents are only trained on episodes with returns close to the maximum.

Not all agents achieve such strong correspondence between desired and obtained returns. Figure 6c shows another agent that solves the task for most values of desired returns, and only achieves lower returns for very low values of desired returns. This indicates that stochasticity during training can affect how trained agents generalize to different commands, and suggests another direction for future investigation.

In the TakeCover-v0 environment, it is rather difficult to achieve precise values of desired returns. Stochasticity in the environment (the monsters appear randomly and shoot in random directions) and increasing difficulty over the episode imply that it is not possible to achieve lower returns than 200 and it gets considerably hard to achieve higher mean returns. The results in Figure 6d reflect these constraints, but still show that the agent is sensitive to the command inputs to some extent.

4 Related Work

Improving RL through SL has a long and rich history beyond the scope of this paper [33, 37]. For example, RL systems based on experience replay [18] attempt to leverage SL on past (off-policy) experience to improve value function approximation. Off-policy training can be augmented with goal-conditional value functions [11] (see also Pong et al. [27], Schaul et al. [31]) such that value functions for goals not being pursued by the current policy can also be updated based on the same interaction history. This idea was recently combined with experience replay by Andrychowicz et al. [1] and extended to policy gradients by Rauber et al. [28]. Oh et al. [23] proposed learning to imitate past good actions to augment actor-critic algorithms. Despite some differences, all of the above algorithms still rely on reward prediction using learned value functions, whereas ⅂Ꙅ uses neither.

There is also substantial prior work on learning reward or goal-conditional policies that directly produce actions [5, 6, 15, 35], but these rely either on a pre-trained world model, a dataset of goal and optimal policy parameters, or policy gradients for learning. While the behavior function in ⅂Ꙅ does bear a high-level similarity to goal-conditioned

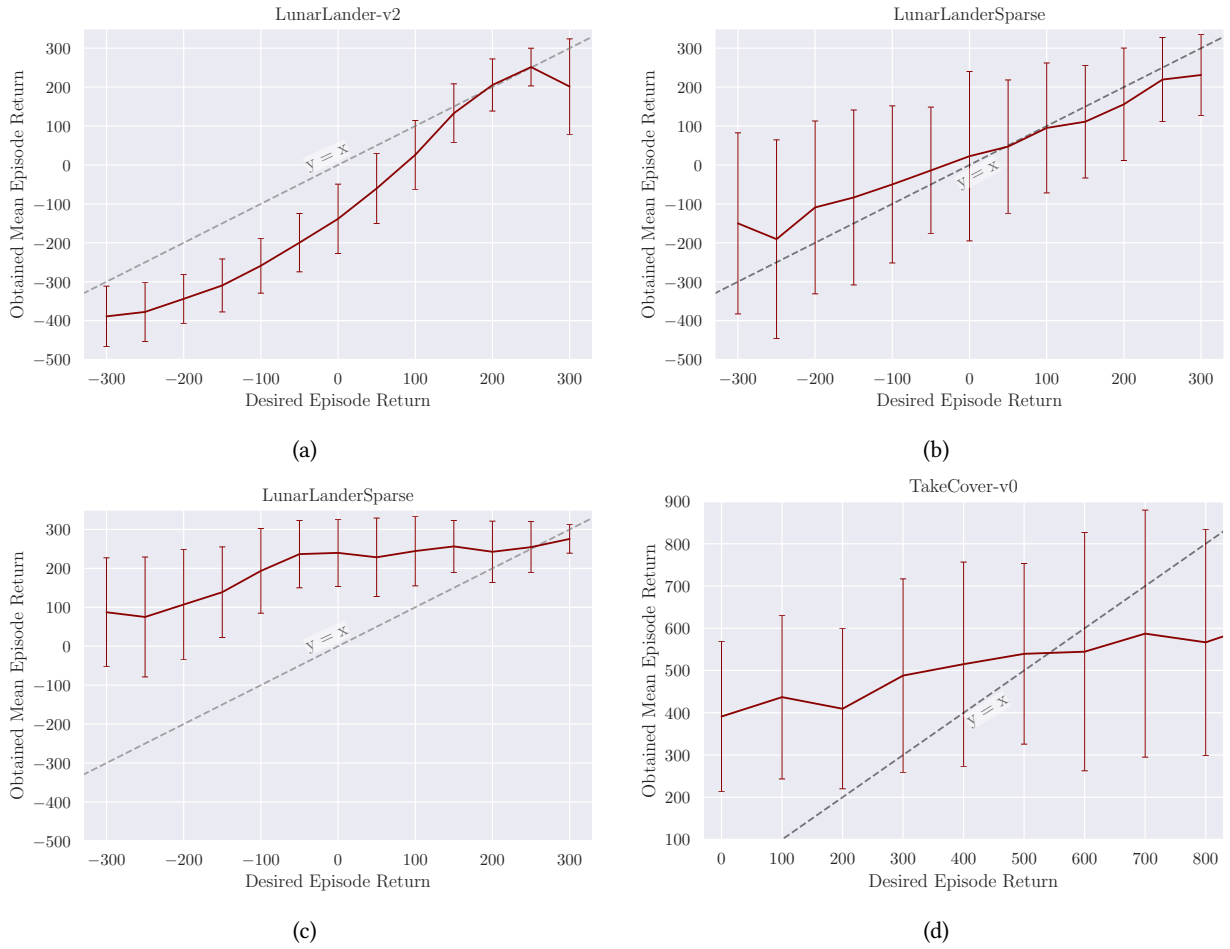


Figure 6: Obtained vs. desired episode returns for UDRL agents at the end of training. Each evaluation consists of 100 episodes. Error bars indicate standard deviation from the mean.

policies, the differences motivating its name are: (a) it takes time-varying desired returns and time horizons as inputs (Algorithm 2), as opposed to fixed goal states, (b) it does not predict rewards at all, (c) it is trained using SL (not policy gradients) on all past behavior, eliminating distinctions between on-policy and off-policy training.

PowerPlay [32, 36] uses extra task inputs to directly produce actions and is also continually trained to make sure it does not forget previously learned skills. However, its task inputs are not selected systematically based on previously achieved tasks/rewards.

A control approach proposed by Dosovitskiy and Koltun [7] uses SL to predict future values of measurements (possibly rewards) given actions, which also sidesteps traditional RL algorithms. A characteristic property of ⅂Ꙅ, however, is its very simple shortcut: it learns the mapping from rewards to actions directly from (possibly accidental) experience, and does not predict rewards at all.

5 Discussion

We introduced basic ideas of ⅂Ꙅ and showed that it can solve certain challenging RL problems. Since RL benchmarks often tend to get developed alongside RL algorithms, a departure from traditional paradigms may motivate new

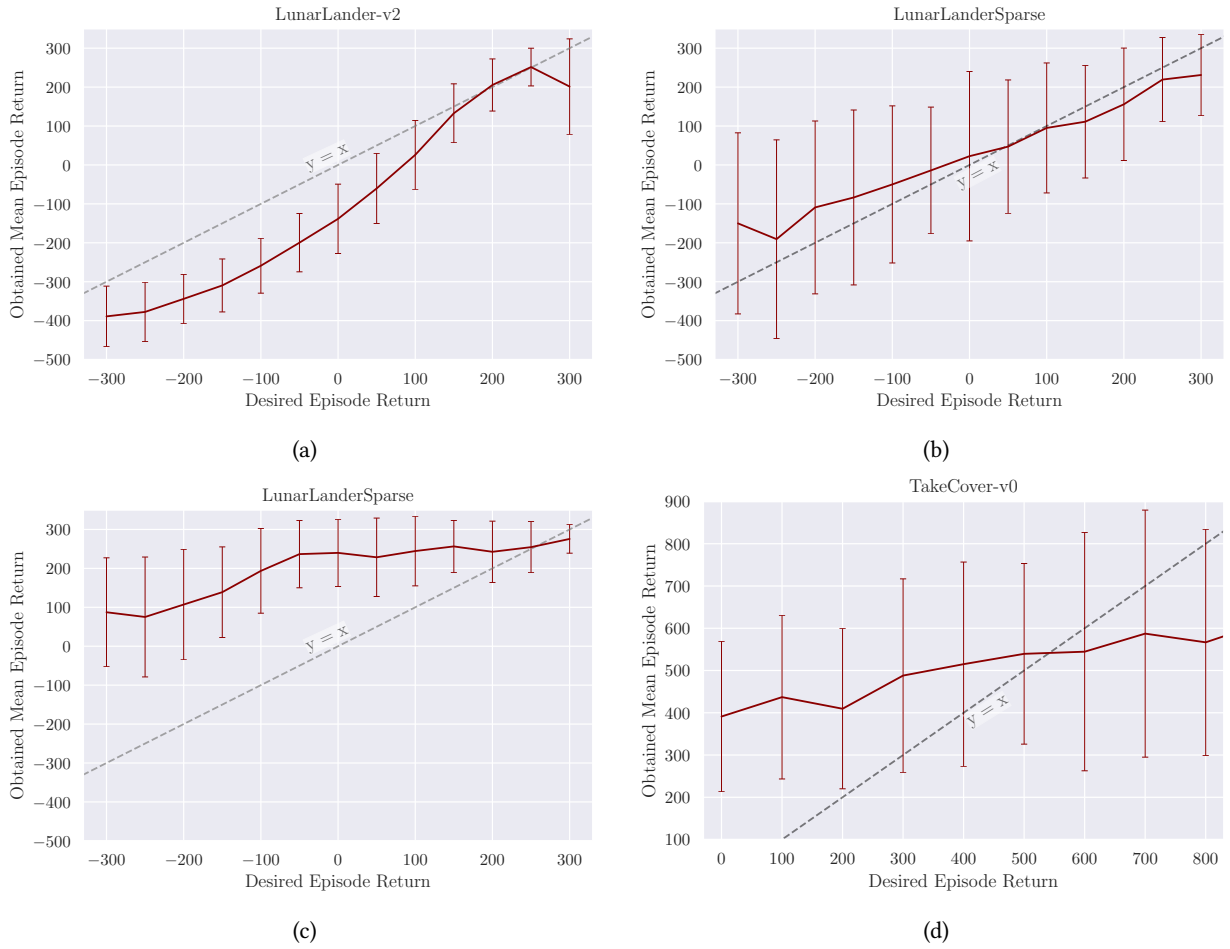


Figure 6: Obtained vs. desired episode returns for UDRL agents at the end of training. Each evaluation consists of 100 episodes. Error bars indicate standard deviation from the mean.

policies, the differences motivating its name are: (a) it takes time-varying desired returns and time horizons as inputs (Algorithm 2), as opposed to fixed goal states, (b) it does not predict rewards at all, (c) it is trained using SL (not policy gradients) on all past behavior, eliminating distinctions between on-policy and off-policy training.

PowerPlay [32, 36] uses extra task inputs to directly produce actions and is also continually trained to make sure it does not forget previously learned skills. However, its task inputs are not selected systematically based on previously achieved tasks/rewards.

A control approach proposed by Dosovitskiy and Koltun [7] uses SL to predict future values of measurements (possibly rewards) given actions, which also sidesteps traditional RL algorithms. A characteristic property of ⅂Ꙅ, however, is its very simple shortcut: it learns the mapping from rewards to actions directly from (possibly accidental) experience, and does not predict rewards at all.

5 Discussion

We introduced basic ideas of ⅂Ꙅ and showed that it can solve certain challenging RL problems. Since RL benchmarks often tend to get developed alongside RL algorithms, a departure from traditional paradigms may motivate new

problem domains that fit ꓤꓤ better than traditional RL.

Many TD-based RL algorithms use discount factors that distort true environmental returns. TD learning is also very sensitive to the frequency of taking actions, which can limit its applicability to robot control [26]. In contrast, ꓤꓤ explicitly takes into account observed rewards and time horizons in a precise and natural way, does not assume infinite horizons, and does not suffer from distortions of the basic RL problem. Note that other algorithms such as evolutionary RL [22] can also avoid these issues in other ways.

Well-known issues arise when off-policy TD learning is combined with high-dimensional function approximation. These issues — referred to by Sutton and Barto [37] as the *deadly triad* — can severely destabilize learning and are usually addressed by adding a variety of ingredients, though complete remedies remain elusive [38].

ꓤꓤ, on the other hand, works fundamentally in concert with high-capacity function approximators (since tabular behavior functions can not generalize), does not require learning from non-stationary targets and does not distinguish between on-policy and off-policy training. Instead, it brings fundamental questions related to catastrophic forgetting [19], continual learning [29], and generalization from past experience to the forefront.

6 Future Research Directions

There are several directions along which the ideas presented in this paper may be extended. On the agent architecture side, using recurrent instead of feedforward neural networks as behavior functions will be necessary for general partially observable environments, and useful for fully observable but noisy environments. New formats of command inputs and/or architectural modifications tailored to them are likely to substantially improve the inductive bias of ꓤꓤ agents. In general, a wide variety of well-known SL techniques for model design, regularization and training can be employed to improve ꓤꓤ's learning stability and efficiency.

Many aspects of the training algorithm were kept deliberately simple for this initial study. Future work should utilize other semantics for command inputs such as "reach a given goal state in at most T time steps", and strategies for sampling history segments other than just trailing segments. Similarly, it is probably unnecessary to generate a constant number of exploratory episodes per iteration, which decreases sample efficiency. We also expect that hyperparameters such as the number of optimization updates per iteration can be automatically adjusted during training.

Our current version of ꓤꓤ utilizes a very simple form of exploration enabled by its design: it simply attempts to achieve high returns by generalizing from known states and returns. This was sufficient to drive learning in the tested environments with small number of available actions and high stochasticity that helps discover new behaviors to learn from. In other environments, additional forms of undirected (random) and directed exploration are likely to be fruitful or even necessary.

Finally, there is a vast open space of possible combinations of ꓤꓤ and algorithms based on learning environmental models, Temporal Differences, optimal control and policy search. Such combinations may lead to more general learning agents that are useful in a variety of environments.

Contributions

All authors contributed to discussions. JS proposed the basic principles of ꓤꓤ [34]. WJ developed the code setup for the baselines. PS and FM developed early implementations and conducted initial experiments on fully deterministic environments. RKS developed the final algorithm, supervised PS and FM, conducted experiments and wrote the paper.

problem domains that fit ꓤꓤ better than traditional RL.

Many TD-based RL algorithms use discount factors that distort true environmental returns. TD learning is also very sensitive to the frequency of taking actions, which can limit its applicability to robot control [26]. In contrast, ꓤꓤ explicitly takes into account observed rewards and time horizons in a precise and natural way, does not assume infinite horizons, and does not suffer from distortions of the basic RL problem. Note that other algorithms such as evolutionary RL [22] can also avoid these issues in other ways.

Well-known issues arise when off-policy TD learning is combined with high-dimensional function approximation. These issues — referred to by Sutton and Barto [37] as the *deadly triad* — can severely destabilize learning and are usually addressed by adding a variety of ingredients, though complete remedies remain elusive [38].

ꓤꓤ, on the other hand, works fundamentally in concert with high-capacity function approximators (since tabular behavior functions can not generalize), does not require learning from non-stationary targets and does not distinguish between on-policy and off-policy training. Instead, it brings fundamental questions related to catastrophic forgetting [19], continual learning [29], and generalization from past experience to the forefront.

6 Future Research Directions

There are several directions along which the ideas presented in this paper may be extended. On the agent architecture side, using recurrent instead of feedforward neural networks as behavior functions will be necessary for general partially observable environments, and useful for fully observable but noisy environments. New formats of command inputs and/or architectural modifications tailored to them are likely to substantially improve the inductive bias of ꓤꓤ agents. In general, a wide variety of well-known SL techniques for model design, regularization and training can be employed to improve ꓤꓤ's learning stability and efficiency.

Many aspects of the training algorithm were kept deliberately simple for this initial study. Future work should utilize other semantics for command inputs such as "reach a given goal state in at most T time steps", and strategies for sampling history segments other than just trailing segments. Similarly, it is probably unnecessary to generate a constant number of exploratory episodes per iteration, which decreases sample efficiency. We also expect that hyperparameters such as the number of optimization updates per iteration can be automatically adjusted during training.

Our current version of ꓤꓤ utilizes a very simple form of exploration enabled by its design: it simply attempts to achieve high returns by generalizing from known states and returns. This was sufficient to drive learning in the tested environments with small number of available actions and high stochasticity that helps discover new behaviors to learn from. In other environments, additional forms of undirected (random) and directed exploration are likely to be fruitful or even necessary.

Finally, there is a vast open space of possible combinations of ꓤꓤ and algorithms based on learning environmental models, Temporal Differences, optimal control and policy search. Such combinations may lead to more general learning agents that are useful in a variety of environments.

Contributions

All authors contributed to discussions. JS proposed the basic principles of ꓤꓤ [34]. WJ developed the code setup for the baselines. PS and FM developed early implementations and conducted initial experiments on fully deterministic environments. RKS developed the final algorithm, supervised PS and FM, conducted experiments and wrote the paper.

Acknowledgments

We thank Paulo E. Rauber for discussions during the early phase of this project. We are also grateful to Faustino Gomez for several suggestions that improved an earlier draft of this report, and Nihat Engin Toklu for discussions that led to the LunarLanderSparse environment.

References

[1] Andrychowicz, M., F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba
2017. Hindsight Experience Replay. *arXiv:1707.01495 [cs]*.

[2] Barto, A. G. and T. G. Dietterich
2004. Reinforcement learning and its relationship to supervised learning. In *Handbook of Learning and Approximate Dynamic Programming*, P. 672.

[3] Bradski, G.
2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

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

[5] Da Silva, B., G. Konidaris, and A. Barto
2012. Learning parameterized skills. *arXiv preprint arXiv:1206.6398*.

[6] Deisenroth, M. P., P. Englert, J. Peters, and D. Fox
2014. Multi-task policy search for robotics. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, Pp. 3876–3881. IEEE.

[7] Dosovitskiy, A. and V. Koltun
2016. Learning to Act by Predicting the Future. *arXiv:1611.01779 [cs]*.

[8] Hakenes, S.
2018. Vizdoomgym. `https://github.com/shakenes/vizdoomgym`.

[9] Hill, A., A. Raffin, M. Ernestus, A. Gleave, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu
2018. Stable Baselines. *GitHub repository*.

[10] Hunter, J. D.
2007. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

[11] Kaelbling, L. P.
1993. Learning to Achieve Goals. In *IJCAI*, Pp. 1094–1099. Citeseer.

[12] Kempka, M., M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski
2016. Vizdoom: A doom-based AI research platform for visual reinforcement learning. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, Pp. 1–8. IEEE.

[13] Kingma, D. and J. Ba
2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[14] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber
2017. The Sacred Infrastructure for Computational Research. In *Proceedings of the 16th Python in Science Conference*, Katy Huff, David Lippa, Dillon Niederhut, and M. Pacer, eds., Pp. 49 – 56.

Acknowledgments

We thank Paulo E. Rauber for discussions during the early phase of this project. We are also grateful to Faustino Gomez for several suggestions that improved an earlier draft of this report, and Nihat Engin Toklu for discussions that led to the LunarLanderSparse environment.

References

[1] Andrychowicz, M., F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba
2017. Hindsight Experience Replay. *arXiv:1707.01495 [cs]*.

[2] Barto, A. G. and T. G. Dietterich
2004. Reinforcement learning and its relationship to supervised learning. In *Handbook of Learning and Approximate Dynamic Programming*, P. 672.

[3] Bradski, G.
2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

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

[5] Da Silva, B., G. Konidaris, and A. Barto
2012. Learning parameterized skills. *arXiv preprint arXiv:1206.6398*.

[6] Deisenroth, M. P., P. Englert, J. Peters, and D. Fox
2014. Multi-task policy search for robotics. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, Pp. 3876–3881. IEEE.

[7] Dosovitskiy, A. and V. Koltun
2016. Learning to Act by Predicting the Future. *arXiv:1611.01779 [cs]*.

[8] Hakenes, S.
2018. Vizdoomgym. `https://github.com/shakenes/vizdoomgym`.

[9] Hill, A., A. Raffin, M. Ernestus, A. Gleave, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu
2018. Stable Baselines. *GitHub repository*.

[10] Hunter, J. D.
2007. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

[11] Kaelbling, L. P.
1993. Learning to Achieve Goals. In *IJCAI*, Pp. 1094–1099. Citeseer.

[12] Kempka, M., M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski
2016. Vizdoom: A doom-based AI research platform for visual reinforcement learning. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, Pp. 1–8. IEEE.

[13] Kingma, D. and J. Ba
2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[14] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber
2017. The Sacred Infrastructure for Computational Research. In *Proceedings of the 16th Python in Science Conference*, Katy Huff, David Lippa, Dillon Niederhut, and M. Pacer, eds., Pp. 49 – 56.

[15] Kupcsik, A. G., M. P. Deisenroth, J. Peters, and G. Neumann
2013. Data-efficient generalization of robot skills with contextual policy search. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*.

[16] LeCun, Y., B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel
1990. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems*, Pp. 396–404.

[17] Liaw, R., E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica
2018. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*.

[18] Lin, L.-J.
1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321.

[19] McCloskey, M. and N. J. Cohen
1989. Catastrophic interference in connectionist networks: The sequential learning problem. *The Psychology of Learning and Motivation*, 24:109–164.

[20] Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu
2016. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*.

[21] Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al.
2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[22] Moriarty, D. E., A. C. Schultz, and J. J. Grefenstette
1999. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:241–276.

[23] Oh, J., Y. Guo, S. Singh, and H. Lee
2018. Self-Imitation Learning. *arXiv:1806.05635 [cs, stat]*.

[24] Oliphant, T. E.
2015. *Guide to NumPy*, 2nd edition. USA: CreateSpace Independent Publishing Platform.

[25] Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer
2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[26] Plappert, M., M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, V. Kumar, and W. Zaremba
2018. Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research.

[27] Pong, V., S. Gu, M. Dalal, and S. Levine
2018. Temporal Difference Models: Model-Free Deep RL for Model-Based Control. *arXiv:1802.09081 [cs]*.

[28] Rauber, P., A. Ummadisingu, F. Mutz, and J. Schmidhuber
2017. Hindsight policy gradients. *arXiv:1711.06006 [cs]*.

[29] Ring, M. B.
1994. *Continual Learning in Reinforcement Environments*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712.

[30] Rosenstein, M. T. and A. G. Barto
2004. Supervised Actor-Critic Reinforcement Learning. In *Handbook of Learning and Approximate Dynamic Programming*, P. 672.

[31] Schaul, T., D. Horgan, K. Gregor, and D. Silver
2015. Universal Value Function Approximators. In *International Conference on Machine Learning*, Pp. 1312–1320.

[15] Kupcsik, A. G., M. P. Deisenroth, J. Peters, and G. Neumann
2013. Data-efficient generalization of robot skills with contextual policy search. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*.

[16] LeCun, Y., B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel
1990. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems*, Pp. 396–404.

[17] Liaw, R., E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica
2018. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*.

[18] Lin, L.-J.
1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321.

[19] McCloskey, M. and N. J. Cohen
1989. Catastrophic interference in connectionist networks: The sequential learning problem. *The Psychology of Learning and Motivation*, 24:109–164.

[20] Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu
2016. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*.

[21] Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al.
2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[22] Moriarty, D. E., A. C. Schultz, and J. J. Grefenstette
1999. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:241–276.

[23] Oh, J., Y. Guo, S. Singh, and H. Lee
2018. Self-Imitation Learning. *arXiv:1806.05635 [cs, stat]*.

[24] Oliphant, T. E.
2015. *Guide to NumPy*, 2nd edition. USA: CreateSpace Independent Publishing Platform.

[25] Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer
2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[26] Plappert, M., M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, V. Kumar, and W. Zaremba
2018. Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research.

[27] Pong, V., S. Gu, M. Dalal, and S. Levine
2018. Temporal Difference Models: Model-Free Deep RL for Model-Based Control. *arXiv:1802.09081 [cs]*.

[28] Rauber, P., A. Ummadisingu, F. Mutz, and J. Schmidhuber
2017. Hindsight policy gradients. *arXiv:1711.06006 [cs]*.

[29] Ring, M. B.
1994. *Continual Learning in Reinforcement Environments*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712.

[30] Rosenstein, M. T. and A. G. Barto
2004. Supervised Actor-Critic Reinforcement Learning. In *Handbook of Learning and Approximate Dynamic Programming*, P. 672.

[31] Schaul, T., D. Horgan, K. Gregor, and D. Silver
2015. Universal Value Function Approximators. In *International Conference on Machine Learning*, Pp. 1312–1320.

[32] Schmidhuber, J.
2013. PowerPlay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in psychology*, 4:313.

[33] Schmidhuber, J.
2015. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

[34] Schmidhuber, J.
2019. Reinforcement learning upside down: Don't predict rewards – just map them to actions. *NNAISENSE/IDSIA Technical Report.*

[35] Schmidhuber, J. and R. Huber
1991. Learning to generate artificial fovea trajectories for target detection. *International Journal of Neural Systems*, 2(1 & 2):125–134.

[36] Srivastava, R. K., B. R. Steunebrink, and J. Schmidhuber
2013. First Experiments with PowerPlay. *Neural Networks*, 41:130–136.

[37] Sutton, R. S. and A. G. Barto
2018. *Reinforcement Learning: An Introduction.* MIT press.

[38] van Hasselt, H., Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil
2018. Deep Reinforcement Learning and the Deadly Triad. *arXiv:1812.02648 [cs].*

[39] Walt, S. v. d., S. C. Colbert, and G. Varoquaux
2011. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30.

[40] Waskom, M., O. Botvinnik, D. O'Kane, P. Hobson, J. Ostblom, S. Lukauskas, D. C. Gemperline, T. Augspurger, Y. Halchenko, J. B. Cole, J. Warmenhoven, J. de Ruiter, C. Pye, S. Hoyer, J. Vanderplas, S. Villalba, G. Kunter, E. Quintero, P. Bachant, M. Martin, K. Meyer, A. Miles, Y. Ram, T. Brunner, T. Yarkoni, M. L. Williams, C. Evans, C. Fitzgerald, Brian, and A. Qalieh
2018. mwaskom/seaborn: v0.9.0 (july 2018).

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

[42] Williams, R. J.
1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

[32] Schmidhuber, J.
2013. PowerPlay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in psychology*, 4:313.

[33] Schmidhuber, J.
2015. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

[34] Schmidhuber, J.
2019. Reinforcement learning upside down: Don't predict rewards – just map them to actions. *NNAISENSE/IDSIA Technical Report.*

[35] Schmidhuber, J. and R. Huber
1991. Learning to generate artificial fovea trajectories for target detection. *International Journal of Neural Systems*, 2(1 & 2):125–134.

[36] Srivastava, R. K., B. R. Steunebrink, and J. Schmidhuber
2013. First Experiments with PowerPlay. *Neural Networks*, 41:130–136.

[37] Sutton, R. S. and A. G. Barto
2018. *Reinforcement Learning: An Introduction.* MIT press.

[38] van Hasselt, H., Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil
2018. Deep Reinforcement Learning and the Deadly Triad. *arXiv:1812.02648 [cs].*

[39] Walt, S. v. d., S. C. Colbert, and G. Varoquaux
2011. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30.

[40] Waskom, M., O. Botvinnik, D. O'Kane, P. Hobson, J. Ostblom, S. Lukauskas, D. C. Gemperline, T. Augspurger, Y. Halchenko, J. B. Cole, J. Warmenhoven, J. de Ruiter, C. Pye, S. Hoyer, J. Vanderplas, S. Villalba, G. Kunter, E. Quintero, P. Bachant, M. Martin, K. Meyer, A. Miles, Y. Ram, T. Brunner, T. Yarkoni, M. L. Williams, C. Evans, C. Fitzgerald, Brian, and A. Qalieh
2018. mwaskom/seaborn: v0.9.0 (july 2018).

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

[42] Williams, R. J.
1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

A Upside-Down RL Hyperparameters

Table 2 summarizes the name and role of all the hyperparameters for ƬƗ.

Table 2: A summary of UDRL hyperparameters

Name	Description
`batch_size`	Number of (input, target) pairs per batch used for training the behavior function
`horizon_scale`	Scaling factor for desired horizon input
`last_few`	Number of episodes from the end of the replay buffer used for sampling exploratory commands
`learning_rate`	Learning rate for the ADAM optimizer
`n_episodes_per_iter`	Number of exploratory episodes generated per step of UDRL training
`n_updates_per_iter`	Number of gradient-based updates of the behavior function per step of UDRL training
`n_warm_up_episodes`	Number of warm up episodes at the beginning of training
`replay_size`	Maximum size of the replay buffer (in episodes)
`return_scale`	Scaling factor for desired horizon input

B Hyperparameter Tuning

Hyperparameters were tuned by randomly sampling 256 configurations for LunarLander-v2 and LunarLanderSparse, and 72 configurations for TakeCover-v0. For LunarLander-v2, each random configuration of hyperparameters was evaluated with 3 random seeds and the results were averaged. For other tasks, each configuration was evaluated with a single seed.

The best hyperparameter configuration was selected based on the mean of evaluation scores for last 20 evaluations, yielding the configurations with the best average performance towards the end of training.

In the following subsections, we define the lists of values for each of the hyperparameters that were tuned for each environment and algorithm. For DQN and A2C, any other hyperparameters were left at their default values for Stable-Baselines, but we did enable additional tricks not found in the original papers such as multi-step bootstrapping or double Q-learning.

B.1 LunarLander-v2 & LunarLanderSparse

Architecture Hyperparameters

- Network architecture (indicating number of units per layer): [[32], [32, 32], [32, 64], [32, 64, 64], [32, 64, 64, 64], [64], [64, 64], [64, 128], [64, 128, 128], [64, 128, 128, 128]]

DQN Hyperparameters

- Activation function: [tanh, relu]
- Batch Size: [16, 32, 64, 128]
- Buffer Size: [10,000, 50,000, 100,000, 500,000, 1,000,000]
- Discount factor: [0.98, 0.99, 0.995, 0.999]
- Exploration Fraction: [0.1, 0.2, 0.4]

A Upside-Down RL Hyperparameters

Table 2 summarizes the name and role of all the hyperparameters for ƬƗ.

Table 2: A summary of UDRL hyperparameters

Name	Description
`batch_size`	Number of (input, target) pairs per batch used for training the behavior function
`horizon_scale`	Scaling factor for desired horizon input
`last_few`	Number of episodes from the end of the replay buffer used for sampling exploratory commands
`learning_rate`	Learning rate for the ADAM optimizer
`n_episodes_per_iter`	Number of exploratory episodes generated per step of UDRL training
`n_updates_per_iter`	Number of gradient-based updates of the behavior function per step of UDRL training
`n_warm_up_episodes`	Number of warm up episodes at the beginning of training
`replay_size`	Maximum size of the replay buffer (in episodes)
`return_scale`	Scaling factor for desired horizon input

B Hyperparameter Tuning

Hyperparameters were tuned by randomly sampling 256 configurations for LunarLander-v2 and LunarLanderSparse, and 72 configurations for TakeCover-v0. For LunarLander-v2, each random configuration of hyperparameters was evaluated with 3 random seeds and the results were averaged. For other tasks, each configuration was evaluated with a single seed.

The best hyperparameter configuration was selected based on the mean of evaluation scores for last 20 evaluations, yielding the configurations with the best average performance towards the end of training.

In the following subsections, we define the lists of values for each of the hyperparameters that were tuned for each environment and algorithm. For DQN and A2C, any other hyperparameters were left at their default values for Stable-Baselines, but we did enable additional tricks not found in the original papers such as multi-step bootstrapping or double Q-learning.

B.1 LunarLander-v2 & LunarLanderSparse

Architecture Hyperparameters

- Network architecture (indicating number of units per layer): [[32], [32, 32], [32, 64], [32, 64, 64], [32, 64, 64, 64], [64], [64, 64], [64, 128], [64, 128, 128], [64, 128, 128, 128]]

DQN Hyperparameters

- Activation function: [tanh, relu]
- Batch Size: [16, 32, 64, 128]
- Buffer Size: [10,000, 50,000, 100,000, 500,000, 1,000,000]
- Discount factor: [0.98, 0.99, 0.995, 0.999]
- Exploration Fraction: [0.1, 0.2, 0.4]

- Exploration Final Eps: [0.0, 0.01, 0.05, 0.1]
- Learning rate: $\text{numpy.logspace}(-4, -2, \text{num} = 101)$
- Training Frequency: [1, 2, 4]
- Target network update frequency: [100, 500, 1000]

A2C Hyperparameters

- Activation function: $[\tanh, \text{relu}]$
- Discount factor: [0.98, 0.99, 0.995, 0.999]
- Entropy coefficient: [0, 0.01, 0.02, 0.05, 0.1]
- Learning rate: $\text{numpy.logspace}(-4, -2, \text{num} = 101)$
- Value function loss coefficient: [0.1, 0.2, 0.5, 1.0]
- Decay parameter for RMSProp: [0.98, 0.99, 0.995]
- Number of steps per update: [1, 2, 5, 10, 20]

Upside-Down RL Hyperparameters

- `batch_size`: [512, 768, 1024, 1536, 2048]
- `horizon_scale`: [0.01, 0.015, 0.02, 0.025, 0.03]
- `last_few`: [25, 50, 75, 100]
- `learning_rate`: $\text{numpy.logspace}(-4, -2, \text{num} = 101)$
- `n_episodes_per_iter`: [10, 20, 30, 40]
- `n_updates_per_iter`: [100, 150, 200, 250, 300]
- `n_warm_up_episodes`: [10, 30, 50]
- `replay_size`: [300, 400, 500, 600, 700]
- `return_scale`: [0.01, 0.015, 0.02, 0.025, 0.03]

B.2 TakeCover-v0

Architecture Hyperparameters All networks had four convolutional layers, each with 3×3 filters, 1 pixel input padding in all directions and stride of 2 pixels.

The architecture of convolutional layers (indicating number of convolutional channels per layer) was sampled from [[32, 48, 96, 128], [32, 64, 128, 256], [48, 96, 192, 384]].

The architecture of fully connected layers following the convolutional layers was sampled from [[64, 128], [64, 128, 128], [128, 256], [128, 256, 256], [128, 128], [256, 256]].

Hyperpameter choices for DQN and A2C were the same as those for LunarLander-v2. For TЯ the following choices were different:

- `n_updates_per_iter`: [200, 300, 400, 500]
- `replay_size`: [200, 300, 400, 500]

- Exploration Final Eps: [0.0, 0.01, 0.05, 0.1]
- Learning rate: $\text{numpy.logspace}(-4, -2, \text{num} = 101)$
- Training Frequency: [1, 2, 4]
- Target network update frequency: [100, 500, 1000]

A2C Hyperparameters

- Activation function: $[\tanh, \text{relu}]$
- Discount factor: [0.98, 0.99, 0.995, 0.999]
- Entropy coefficient: [0, 0.01, 0.02, 0.05, 0.1]
- Learning rate: $\text{numpy.logspace}(-4, -2, \text{num} = 101)$
- Value function loss coefficient: [0.1, 0.2, 0.5, 1.0]
- Decay parameter for RMSProp: [0.98, 0.99, 0.995]
- Number of steps per update: [1, 2, 5, 10, 20]

Upside-Down RL Hyperparameters

- `batch_size`: [512, 768, 1024, 1536, 2048]
- `horizon_scale`: [0.01, 0.015, 0.02, 0.025, 0.03]
- `last_few`: [25, 50, 75, 100]
- `learning_rate`: $\text{numpy.logspace}(-4, -2, \text{num} = 101)$
- `n_episodes_per_iter`: [10, 20, 30, 40]
- `n_updates_per_iter`: [100, 150, 200, 250, 300]
- `n_warm_up_episodes`: [10, 30, 50]
- `replay_size`: [300, 400, 500, 600, 700]
- `return_scale`: [0.01, 0.015, 0.02, 0.025, 0.03]

B.2 TakeCover-v0

Architecture Hyperparameters All networks had four convolutional layers, each with 3×3 filters, 1 pixel input padding in all directions and stride of 2 pixels.

The architecture of convolutional layers (indicating number of convolutional channels per layer) was sampled from [[32, 48, 96, 128], [32, 64, 128, 256], [48, 96, 192, 384]].

The architecture of fully connected layers following the convolutional layers was sampled from [[64, 128], [64, 128, 128], [128, 256], [128, 256, 256], [128, 128], [256, 256]].

Hyperpameter choices for DQN and A2C were the same as those for LunarLander-v2. For TЯ the following choices were different:

- `n_updates_per_iter`: [200, 300, 400, 500]
- `replay_size`: [200, 300, 400, 500]

- `return_scale`: [0.1, 0.15, 0.2, 0.25, 0.3]

C Software Implementation

Our setup directly relied upon the following open source software:

- Gym 0.11.0 [4]
- Matplotlib [10]
- Numpy 1.15.1 [24, 39]
- OpenCV [3]
- Pytorch 1.1.0 [25]
- Ray Tune 0.6.6 [17]
- Sacred [14]
- Seaborn [40]
- Stable-Baselines 2.5.0 [9]
- Vizdoom 1.1.6 [12]
- Vizdoomgym [8]

- `return_scale`: [0.1, 0.15, 0.2, 0.25, 0.3]

C Software Implementation

Our setup directly relied upon the following open source software:

- Gym 0.11.0 [4]
- Matplotlib [10]
- Numpy 1.15.1 [24, 39]
- OpenCV [3]
- Pytorch 1.1.0 [25]
- Ray Tune 0.6.6 [17]
- Sacred [14]
- Seaborn [40]
- Stable-Baselines 2.5.0 [9]
- Vizdoom 1.1.6 [12]
- Vizdoomgym [8]