Knowledge-Based Systems 218 (2021) 106844

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Forgetful experience replay in hierarchical reinforcement learning from expert demonstrations

Alexey Skrynnik [a], Aleksey Staroverov [a,b], Ermek Aitygulov [b], Kirill Aksenov [b], Vasilii Davydov [c], Aleksandr I. Panov [a,b,*]

[a] Artificial Intelligence Research Institute FRC CSC RAS, Moscow, Russia
[b] Moscow Institute of Physics and Technology, Moscow, Russia
[c] Moscow Aviation Institute, Moscow, Russia

ARTICLE INFO

Article history:
Received 11 October 2020
Received in revised form 25 January 2021
Accepted 30 January 2021
Available online 12 February 2021

Keywords:
Expert demonstrations
ForgER
Hierarchical reinforcement learning
Learning from demonstrations
Task-oriented augmentation
Goal-oriented reinforcement learning

ABSTRACT

Deep reinforcement learning (RL) shows impressive results in complex gaming and robotic environments. These results are commonly achieved at the expense of huge computational costs and require an incredible number of episodes of interactions between the agent and the environment. Hierarchical methods and expert demonstrations are among the most promising approaches to improve the sample efficiency of reinforcement learning methods. In this paper, we propose a combination of methods that allow the agent to use low-quality demonstrations in complex vision-based environments with multiple related goals. Our Forgetful Experience Replay (ForgER) algorithm effectively handles expert data errors and reduces quality losses when adapting the action space and states representation to the agent's capabilities. The proposed goal-oriented replay buffer structure allows the agent to automatically highlight sub-goals for solving complex hierarchical tasks in demonstrations. Our method has a high degree of versatility and can be integrated into various off-policy methods. The ForgER surpasses the existing state-of-the-art RL methods using expert demonstrations in complex environments. The solution based on our algorithm beats other solutions for the famous MineRL competition and allows the agent to demonstrate the behavior at the expert level.

1. Introduction

Modern reinforcement learning (RL) methods require huge computational resources and a large number of episodes of interaction with the environment, especially for learning effective policies in complex hierarchical and robotic environments [1,2]. One of the most promising approaches to developing sample-efficient RL methods is imitation learning [3,4] and expert demonstrations [5,6]. Expert trajectories for the imitation are obtained by recording human actions or running a pre-trained algorithm. Getting high-quality suboptimal demonstrations is a separate, time-consuming task that is comparable in complexity with data markup for supervised learning [7]. Another challenge is adapting expert trajectories to the capabilities and limitations of agents. It is often impossible to accurately correlate the actions of an expert and an agent, especially in hybrid discrete–continuous cases.

The standard approach is when we simplify the task for collecting expert data by reducing the quality requirements for trajectories and the optimal strategy used by the expert [8,9]. This paper, considers using demonstrations in off-policy RL methods with a replay buffer as the most suitable method for learning and planning on the trajectories collected by a secondary strategy [10,11]. Using noisy trajectories applied for the initial buffer filling allows the agent to learn faster in the environment. However, ineffective actions in demonstrations are another source of agent errors and unobserved states that are not covered by expert trajectories. This is even more likely to lead to a catastrophic drop in total reward and it is the main problem in RL from imperfect demonstrations [8].

We propose a new hierarchical experience replay technique that allows overcoming the main disadvantages for the existing methods of off-policy learning from demonstrations. We propose managed forgetting expert data by reducing their ratio in the experience replay buffer and learning bathes. In contrast to the previously used approaches with a constant priority and a fraction of expert data [5,12], our approach rather effectively deals with the influence of low-quality expert actions and adapts the trajectories to a discrete space of agent actions. We applied a hierarchical approach to use demonstrations and to fill a replay

The code (and data) in this article has been certified as Reproducible by Code Ocean:https://help.codeocean.com/en/articles/1120151-code-ocean-s-verification-process-for-computational-reproducibility. More information on the Reproducibility Badge Initiative is available at www.elsevier.com/locate/knosys.

* Corresponding author at: Artificial Intelligence Research Institute FRC CSC RAS, Moscow, Russia.
E-mail addresses: skrynnik@isa.ru (A. Skrynnik), pan@isa.ru (A.I. Panov).

https://doi.org/10.1016/j.knosys.2021.106844
0950-7051/

buffer. We proposed an algorithm for extracting sub-goals from expert data that correspond to certain parts of trajectories and subtasks. Following the selected sub-goals, the agent constructs a replay buffer in which parts of each sub-task's trajectories are explicitly separated. This replay buffer structure allows generating meta-actions and tuning them during interaction with the environment. Finally, we developed a hierarchical task-oriented augmentation of expert data to partially reduce the requirement for the amount of data necessary for high-quality imitations.

We used these techniques to develop a new Forgetful Experience Replay (ForgER) algorithm that forges expert data while collecting its own experience automatically correlated with subtasks performed in the environment. To investigate the contribution and features of the main ideas, we considered two environment classes, i.e., the indicative simple low-dimensional environments with simple action space (Lunar Lander, Torcs, and others) [13] and the complex hierarchical vision-based Minecraft [14] with hybrid discrete–continuous action space. Using simple environments, we compared ForgER with the well-known state-of-the-art (SOTA) approaches. We showed the effect of the forgetting technique in three main cases: poor quality of expert data, incorrect choice of discretization, and variability of the environment, which leads to a discrepancy between the expert's value function and the environment in which the agent operates. The existing off-policy methods using demonstrations are designed to work in environments where there is no sub-goal structure, and a simple set of actions is used. Our approach extends to work in complex hierarchical environments with a mixed (discrete and continuous) set of actions. We apply ForgER in the well-known complex Minecraft environment, which has recently served as a good benchmark for testing RL algorithms in rich hierarchical settings [15,16]. This allows us to demonstrate the performance of the ForgER and surpass the results of recent SOTA methods in the MineRL competition [17].

The main contributions of this work are as follows. We investigated the impact of poor-quality expert data on the effectiveness of off-policy methods. We proposed a new forgetting mechanism to deal with a catastrophic drop in productivity due to poor-quality expert trajectories and extended the approach of using demonstrations to partially observed and hierarchical environments. We developed a data augmentation method in RL that weakens the requirements for the amount and quality of expert data for efficient imitation. We conducted a detailed experimental study on simple environments and showed the advantage of our method as compared to the known SOTA approaches. Using the proposed algorithm, we surpass the existing algorithms for solving hierarchical tasks in the Minecraft environment.

2. Background

We consider a Markov decision process (MDP) [18] defined by the tuple (S, A, T, R, γ) where S is the state space, A is the action space, the unknown transition probability $T : S \times A \times S \to [0, \infty)$ represents the probability density of reaching $s_{t+1} \in S$ from $s_t \in S$ by taking the action $a \in A$, $\gamma \in [0, 1]$ is the discount factor, and the bounded real-valued function $R : S \times A \to [r_{\min}, r_{\max}]$ represents the reward of each transition. Using the policy $\pi(a_t|s_t)$ to sequentially generate actions we obtain a trajectory through the environment $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$. For any given policy, we define the action–value function (Q-function) as

$$Q^{\pi}(s, a) = \mathbb{E}_{\tau:s_0=s,a_0=a}\left[\sum_t \gamma^t R(s_t, a_t)\right]. \quad (1)$$

This function represents the expected discounted future total return. The goal is to learn the optimal policy π^*, that maximizes the action–value function for each state in S [19].

Q-learning [20] is a well-known RL algorithm that uses samples of experience of the form (s_t, a_t, r_t, s_{t+1}) to estimate the optimal action–value function $Q^*(s_t, a_t)$. Hereby, $Q^*(s_t, a_t)$ is the expected return of selecting action a_t in state s_t and following an optimal policy π^*. Deep RL methods like DQN [21], Double DQN [22] and Dueling DQN [23] parameterize the Q-function and represent it as $Q_\theta(s_t, a_t)$, where neural network weights θ are updated using stochastic gradient descent. There are two main points in DQN that allowed us to apply deep neural networks to the RL problem. First, it uses a separate target network that is copied after a certain number of steps from the regular network. Second, it uses a replay buffer $\mathcal{D}$ where the agent adds all of its experiences. The use of these techniques leads to better stability of the target Q-function.

Most challenging tasks in reinforcement learning provide only partial observations. In our work, we conduct the main experiments in a partially observed environment MineRL, the model of which is represented as a Partially Observable Markov Decision Process (POMDP) [24]. A POMDP is a tuple $(S, O, A, T, R, \omega, \gamma)$, where S, A, R, T, and γ are defined as in an MDP, O is the finite set of the observations, and $\omega(s, o)$ is the observation probability distribution. At every time step t, the agent executes an action $a_t \in A$ and receives a reward $R(s_t, a_t)$ and an observation $o_{t+1} \in O$. The agent does not observe the true state s_{t+1}, and only the observation provides the agent a clue about what the state $s_{t+1} \in S$ is. Usually, to account for the agent's interaction history with the environment the concept of a belief state is introduced. The belief state is a probability distribution $b_t : S \to [0, 1]$ over S, such that $b_t(s)$ is the probability that the agent is in state $s \in S$ given the history up to time t. The belief state b_{t+1} is determined as following

$$b_{t+1}(s') \propto \omega(s', o_{t+1}) \sum_{s \in S} p(s, a_t, s') b_t(s) \quad (2)$$

for all $s' \in S$.

The idea of using demonstrations is to train the agent as much as possible from the demonstration data before running in the real environment. Most algorithms such as DQfD [5] work in a fully observable case. The use of demonstrations occurs once, i.e., during the imitating phase in which the agent learns to imitate the demonstrator. During this imitating phase, the agent samples mini-batches from the demonstration data and updates the network by applying a loss function with a margin classification part [25]. This loss grounds the values of the unseen actions to reasonable values and makes the greedy policy imitate the demonstrator. Using regularization on the network weights and biases prevents the network from overfitting on the relatively small demonstration dataset.

When the imitating phase is completed, the agent begins to fill the replay buffer with self-generating data representing its own experience. In such works as PDD DQN [12] for forming a sample, prioritization is used. It consists of adding different small positive constants to the priorities of the agent and demonstration transitions to control the relative sampling of demonstration versus agent data. In our work, we also use this two-phase approach.

3. Forgetful experience replay

In addition to simple single-goal environments, our approach extends to partially observable hierarchical tasks in which a hierarchical structure of subtasks given or can be extracted. This assumption allows us to simplify the difficult POMDP task with sparse rewards to a set of simpler ones. We define each subtask as a meta-action or an option [26]. An option is triple $(\mathcal{I}, \pi, \beta)$ in which $\mathcal{I} \in O$ is the initiation set, π is the inner option policy, and β is the termination function. Additionally, each option may

have its own function f_p that defines pseudo-rewards. For some tasks, the hierarchical structure can be extracted automatically or semi-automatically using human demonstrations.

We will consider such problems in which sub-goals can be determined by the features extracted from expert trajectories. Examples of such features are the appearance of an item in the character's inventory, reaching a new level, receiving a certain reward. To represent the dependency of one subtask on another, we construct a directed weighted graph $G = (V, E)$, where the vertices $V = \{g_1, g_2, \ldots, g_n\}$ are the extracted subgoals (feature vectors). The set of edges E is defined by transitions from one subgoal to another in expert trajectories, the weights of the edges are proportional to the number of such transitions. This graph needs to be turned into a tree to apply one of the hierarchical reinforcement learning methods. In our method, we use the options framework. Topological sorting and various heuristics can be used for this conversion. The resulting tree does not guarantee the convergence of hierarchical approaches, and adjusting its structure is a further direction of our work.

The agent learning process consists of two phases: the *imitating phase* using demonstrations and the *forging phase* when the agent refines the policy during interaction with the environment. The architecture of the ForgER agent is shown in Fig. 1. In both phases, the agent uses a replay buffer $\mathcal{D}$ structured regarding the current sub-goal tree G. The pseudocode of the ForgER approach is sketched in Algorithm 1. All demonstrations are divided into expert $\mathcal{D}_g^{demo}$ and extra $\mathcal{D}_g^{extra}$ for each subtask g, which are used during the imitating phase. The agent samples mini-batches from the demonstration and updates the network by applying the POMDP goal-specific loss function:

$$L(Q^g) = L_{PDQ}(Q^g) + \lambda_1 L_n(Q^g) + \lambda_2 L_{PE}(Q^g) + \lambda_3 L_{L2}(Q^g) \tag{3}$$

where $g \in V(G)$ is a subtask during which experience is collected. L_{PDQ} is the 1-step double Q-learning loss:

$$L_{PDQ}(Q^g) = \left(R(o_t, a_t) + \gamma Q^g_{\theta'}(o_{t+1}, \arg\max_a Q^g_\theta(o_{t+1}, a)) - Q^g_\theta(o_t, a_t)\right)^2, \tag{4}$$

and L_n is the n-step double Q-learning loss. This part of the overall loss function ensures that the network satisfies the Bellman equation and helps propagate the values of the expert's trajectory to all the earlier states [27]. L_{PE} is a supervised large margin classification loss:

$$L_{PE}(Q^g) = \max_{a_t \in A}[Q^g(o_t, a_t) + l(a_t^E, a_t)] - Q^g(o_t, a_t^E), \tag{5}$$

where a_t^E is the action that the expert demonstrator took in state s_t and $l(a_t^E, a_t)$ is the margin function that is 0 when $a_t = a_t^E$ and positive otherwise. Finally, L_{L2} is an L2 regularization loss on the network weights and biases.

The structured replay buffer is defined by $\mathcal{D} = \{d_i = (o, a, r, o', \lambda_2, g)\}$ in which λ_2 is the margin weight. Each option policy π^g is formed using a new hierarchical augmentation approach. During the forging phase, forgetting is used when the dynamic adjustment of the ratio of expert and agent experience occurs. In addition to the introduced replay buffer, we add noisy layers, that are used along with ϵ-greedy exploration.

3.1. Forgetting in learning from demonstrations

The forgetting approach is part of our architecture designed for hierarchical tasks, but it can be used separately for learning from demonstrations, where it showed better results than the standard approach. In this paper, we address these three problems of expert demonstrations, which could be solved using the ForgER algorithm. The first one is the suboptimality of the expert's policy, which can be caused by the expert's errors. The second one is the discrepancy between the action space of the agent and the expert. For example, the data taken from robot sensors may contain noise and errors, imperfect conditions for recording demonstrations, the limitations of the space in which the agent can act. The third problem is the imbalance of the expert trajectories, which may be caused by incorrect data processing (e.g., the selection of only the best trajectories with the best initial conditions).

Algorithm 1 Forgetful experience replay

Inputs: G : subtask graph, θ : weights for the initial subtask network, θ' : weights for the target subtask network, τ : frequency at which to update the target network, k : the number of the imitating steps, $\mathcal{D}$: structured replay buffer, f_p : goal-oriented pseudo reward function, $f_{rg}(t)$: forgetting ratio function
for $g \in V(G)$ **do**
  $t_g \leftarrow 0$, $\theta_g \leftarrow \theta$, $\theta'_g \leftarrow \theta'$,
  $\mathcal{D}_g^{demo} \leftarrow (o, a, f_p(o, r), o', a', 1, g) : d_i \in \mathcal{D}$ where $g_i = g$
  $\mathcal{D}_g^{extra} \leftarrow (o, a, 0, o', a', 0, g_i) : d_i \in \mathcal{D}$ where $g_i \neq g$
  imitating$(g, \mathcal{D}_g^{demo}, \mathcal{D}_g^{extra})$
end for
for episode $k \in \{1, 2, \ldots\}$ **do**
  forging$(g, \mathcal{D}_g^{demo}, \mathcal{D}_g^{agent}, t_g, f_p, k, f_{rg}(k))$
  Select next subtask g from G
end for

Algorithm 2 ForgER forging phase

function forging
  Inputs: g : current subtask, $\mathcal{D}_g^{demo}$: initialized with demonstration data for current subtask g, $\mathcal{D}_g^{agent}$: initialized with data collected by agent for current subtask g, t: current step, f_p: pseudo reward function, k: current episode, $f_{rg}(k)$: forgetting until episode
  while subtask g not solved **do**
    Sample action from policy $a \sim \pi^{\epsilon Q_{\theta g}}$
    Play action a and observe (o', r).
    Replace (o, a, r, o') by $(o, a, f_p(o, r), o')$
    Store $(o, a, r, o', 0, g)$ in $\mathcal{D}$
    Sample a mini-batch of n transitions from $\mathcal{D}_g^{demo}$ and $\mathcal{D}_g^{agent}$ with forgetting rate $f_{rg}(k)$
    Calculate loss $L(Q^g)$ using target network and perform a learning update to θ_g
    if t **mod** $\tau = 0$ **then** $\theta'_{cs} \leftarrow \theta_{cs}$ **end if**
    $t \leftarrow t + 1$
  end while
end function

Algorithm 3 ForgER imitating phase

function imitating
  Inputs: g : current subtask, $\mathcal{D}_g^{demo}$: initialized with demonstration data for current subtask g, $\mathcal{D}_g^{extra}$: initialized with data from other subtasks
  for steps $t \in \{1, 2, \ldots k\}$ **do**
    Sample a mini-batch of n transitions from $\mathcal{D}_g^{demo}$ and $\mathcal{D}_g^{extra}$
    Calculate loss $L(Q^g)$ using target network θ'_g and perform a learning update to θ_g
    if t **mod** $\tau = 0$ **then** $\theta'_{cs} \leftarrow \theta_g$ **end if**
  end for
end function

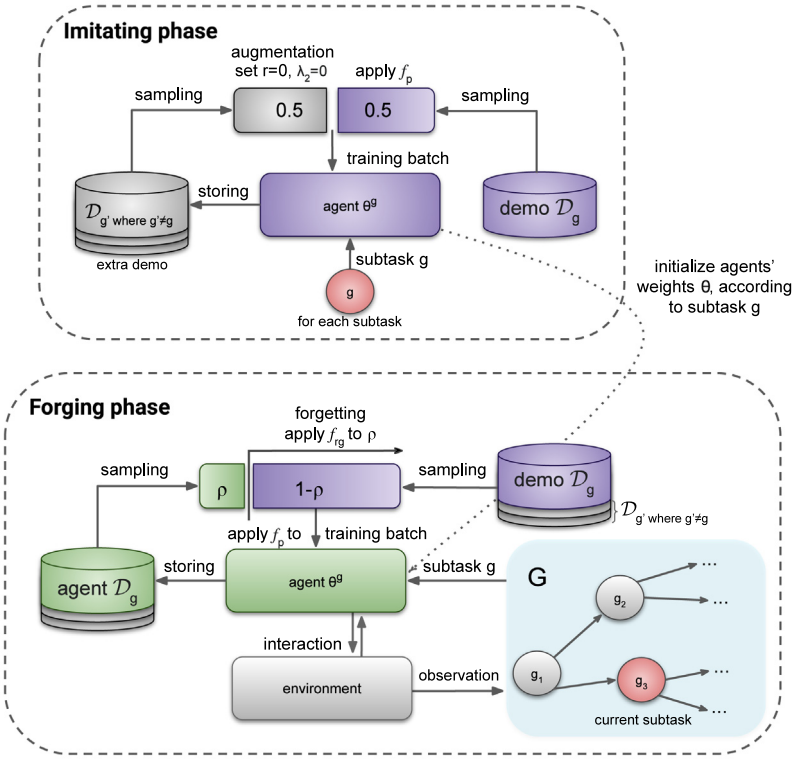

Fig. 1. ForgER architecture diagram for hierarchical setting. The training starts with the **Imitating phase**, that uses only expert data. The idea of augmentation is to use the entire dataset to train each agent in a special way. During the training of each of the agents, the batch is sampled from the buffer $\mathcal{D}_g^{demo}$ with the data belonging to the current subtask g, and from the extra replay buffer $\mathcal{D}_g^{extra}$ with zeroed both rewards and margin loss weights λ_2. The **Forging phase** includes interaction with the environment. This phase is universal and can be applied to non-hierarchical settings in learning from demonstrations task. The sampling from the expert and agent buffers occurs with a certain ratio function $\rho = f_{rg}(t)$ (forgetting rate), which gradually reduces the amount of used expert data.

There are several ways to sample a batch from the replay buffer. The first way is to store all data in the same buffer. Expert demonstrations are always present in the replay buffer and they are sampled with high priority. The second way is to store data in two separated buffers and sample a batch in a specific ratio (e.g., half of the batch data from the expert, half from the agent) [28]. And the final way is a ForgER approach: data is sampled in a dynamic ratio according to $f_{rg}(t)$. The amount of expert data in the batch is gradually decreasing.

Forgetting is the process of dynamically changing the sampling rate of experts and agent data. For example, we can define the sampling rate changing process as $f_{rg}(k) = \min(1, k/d)$ (linear forgetting) in which $f_{rg}(k)$ is the sampling rate (forgetting rate), k is the current episode and d is the last episode of forgetting (i.e., the last episode in which the expert data is used for learning) (see Algorithm 2).

3.2. Task-specific augmentation

The idea of hierarchical augmentation is to use data from other subtasks as extra data on each policy's imitating phase (see Algorithm 3). Both supervised loss function L_{PE} and pseudo rewards f_p are turned off for the extra data.

Using this type of augmentation of ForgER, we can solve two problems. The margin loss function causes the agent to learn how to act as an expert at the cost of generalization. Additional data prevents overfitting. The division into subtasks leads to the fact that only part of the data is used to learn each option policy π. The use of additional data and TD losses allows us to train the agent on additional information from other subtasks. For example, in Minecraft, such behavior as avoiding obstacles or floating out of water can be reused in different subtasks.

4. Experiments

We evaluated ForgER in two classes of environments: *simple* and *hierarchical* sets. The first *simple set* is vector-based and small vision-based environments with a single subgoal: classic Control benchmark (CartPole, MountainCar, Acrobot), environments from Box2D benchmark (LunarLander, CarRacing) [13], and racing simulator *Torcs* with discrete action space. We chose these environments because only in these simple conditions we can compare our method with the existing state-of-the-art methods dealing with expert demonstrations that are not adapted to work in complex hierarchical environments: POfD [29], NAC [8], and DQfD algorithms.

The second *hierarchical set* is vision-based *Minecraft* environments including the setting of the *MineRL* competition [15] with hybrid action space. Using this set, we demonstrated the behavior of the ForgER agent in a hierarchical setting where different subgoals can appear when the main goal is reached: *Navigation* (the agent must reach the target), *Treechop* (the agent needs to chop wood starting with an iron axe for cutting trees), *ObtainIronPickaxe* (obtaining an iron pickaxe extracting many items sequentially), *ObtainDiamond* (obtaining a diamond, which is the rarest element in Minecraft). The *hierarchical set* is utterly complex since the environment for each of the subtasks is procedurally generated. Also, environments from this set have sparse rewards (even dense versions) and belong to a POMDP class because only the first-person view is available for the agent.

In addition to the comparative analysis, we conducted an ablation study to analyze the impact of the quality of expert data and various hyperparameters of our algorithm.

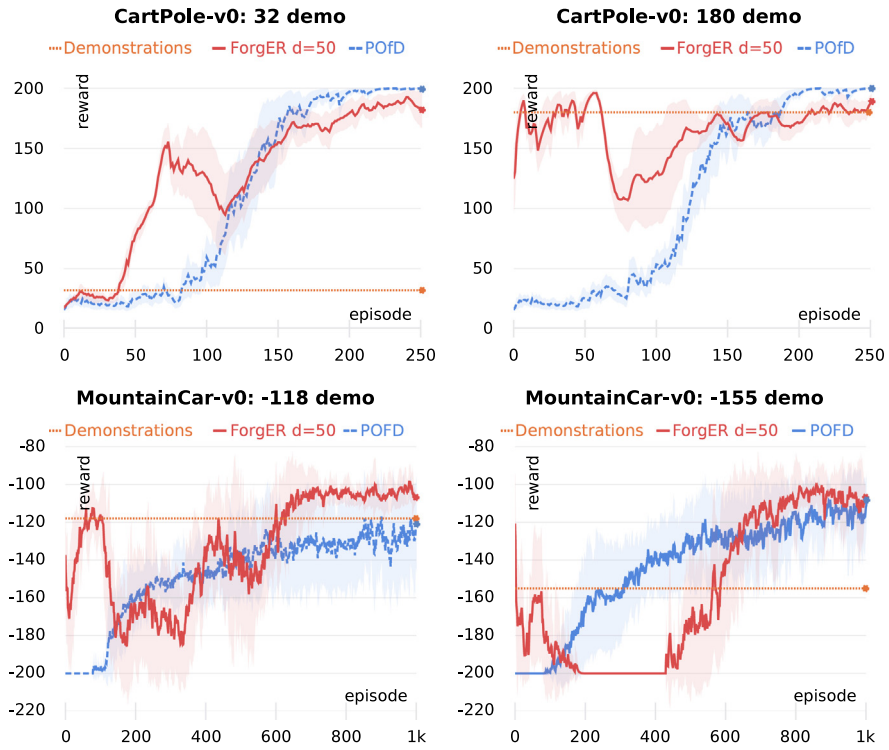

Fig. 2. Two top graphs demonstrate a mean episode reward for the ForgER and the POfD agents in the *CartPole* environment using demonstrations with a different average score. Two bottom graphs demonstrate a mean episode reward for the ForgER and the POfD agents in the *MountainCar* environment.

4.1. Comparative analysis

4.1.1. ForgER vs. POfD in a simple set

This set of experiments shows if the ForgER performance with the imitation phase on demonstrations collected with suboptimal policies can be compared with the POfD. The POfD was chosen as a baseline because it aimed at getting benefits from an imperfect demonstration and outperformed strong algorithms in environments with vector-based observation space. We were able to show that the ForgER shows comparable results with the POfD for simple vector tasks (see Fig. 2).

We evaluated the ForgER in three environments that have discrete action space and for each environment, we collected two sets of demonstrations with a different average score. The ForgER showed comparable performance on demonstrations with a high average score but was outperformed by the POfD on demonstrations with a low average score. However, the low quality of demonstrations is usually not caused by bad performance of an expert. Normally, demonstrations are collected with a high average score but in a setup different from the one in which the agent is acting. For example, demonstrations can be collected by humans with continuous actions that were discretized for the agent. The ForgER significantly outperforms the POfD on the visual-based CarRacing environment (see top right picture on Fig. 3). We relate these results with the fact that the POfD approach rely more on physical control benchmarks, while the ForgER works better on the visual-based tasks.

We collected demonstrations with a high average score in environments with continuous action space (*LunarLanderContinuous*, *CarRacing*), and then the actions were discretized. We used *LunarLander* action space (four actions) for *LunarLanderContinuous* discretization and custom discretization for *CarRacing* (four actions). The action from the continuous action space was mapped to the nearest action from the discretized space. In this case, the ForgER fully outperformed the POfD (see Fig. 3). For each algorithm and for each (demonstration, environment) pair, only one hyperparameter was tuned: forgetting speed for the ForgER (d parameter in the linear forgetting function $f_r g(k)$) and the reward coefficient for the POfD. The results were averaged across four seeds.

4.1.2. ForgER vs. NAC vs. DQfD in a simple set

In Torcs, the agent's goal is to drive as fast as possible. The action space is a Cartesian product between left, no-op, right and up, no-op, down, while the observation space is a vector with a size of 29, in which all the information about the car and the track is contained. The reward is computed at each step, and it depends on the velocity of the car projected along the track direction. The expert dataset was obtained using a PPO agent, and it had an average reward of 9230. In the ForgER, a linear function with $d = 50$ was used as the forgetting function $f_r g(k)$. The results were averaged over five random seeds. As can be seen in Fig. 4 (middle), the results of the NAC and DQfD are similar to those obtained in the article [8]. However, the ForgER results are much higher (about 20% as compared to the NAC).

4.2. Ablation study

In this series of experiments, our goal was to demonstrate the impact of the expert data quality, the action discretization options, the task-oriented augmentation on the behavior of the ForgER and DQfD, and the characteristics of the overfitting process. Here, we used a number of environments from the MineRL competition.

4.3. MineRL environments

MineRL is a shell on the game Minecraft, which presents several environments (subtasks):

- *Navigate*: In this environment, the agent must reach the target. In addition to standard observations, the agent has

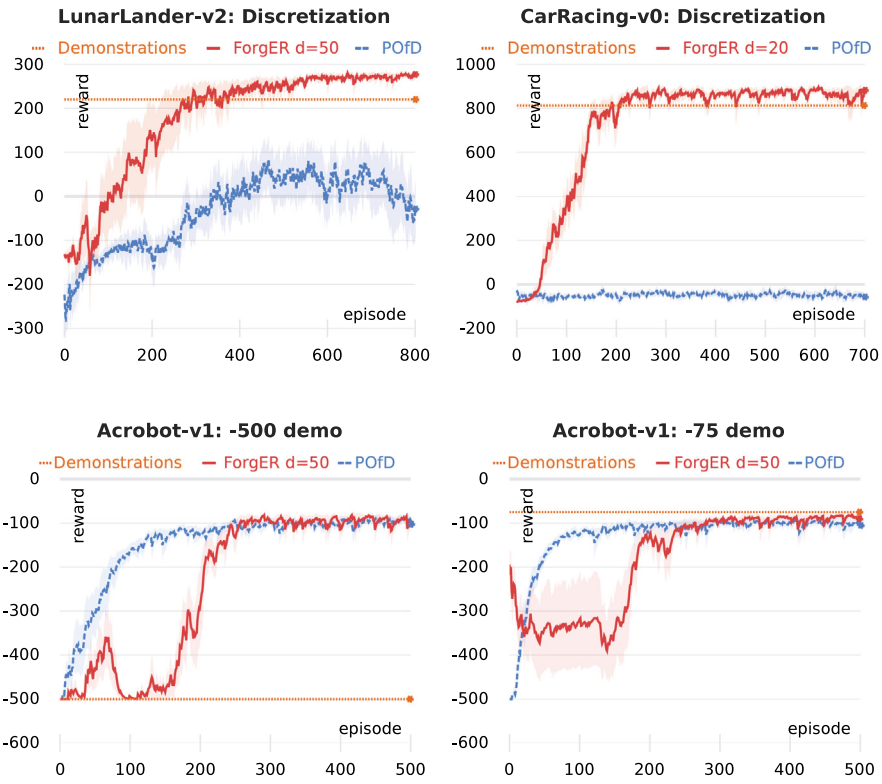

Fig. 3. ForgER vs POfD in *LunarLander* (**top left**) and *CarRacing* (**top right**) with expert data after discretization. Two bottom graphs demonstrate a mean episode reward for the ForgER and the POfD agents in the *Acrobot* environment using demonstrations with a different average score.

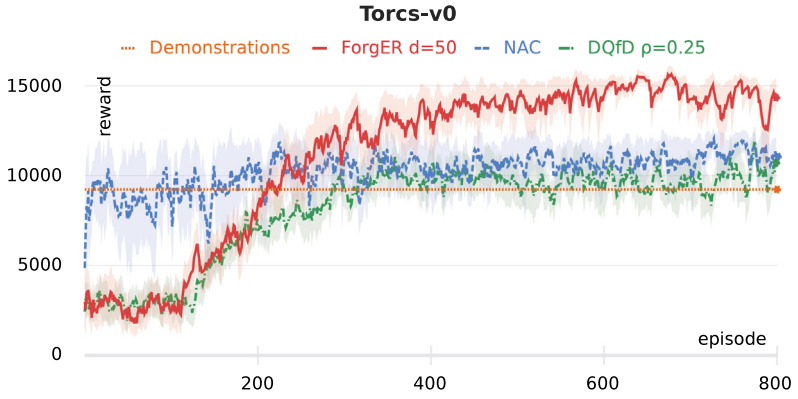

Fig. 4. Mean episode reward for the ForgER, NAC, DQfD agents in *Torcs*. The other approaches do not perform as well as the ForgER.

access to a compass that points to a target located 64 meters from the starting location. The agent receives a reward +100 for achieving the goal, after which the episode ends. There is also a Dense version in which the agent receives a reward for each step, depending on the approach to the goal.

- *Treechop*: In this environment, the agent needs to chop wood. The agent starts in a forest with an iron ax for cutting trees. The agent receives +1 for each unit of wood received, and the episode ends as soon as the agent receives 64 units, or a time limit is reached.
- *ObtainIronPickaxe*: The main goal of this environment is an iron pickaxe. To solve this environment, it is necessary to extract many items, and this must be done sequentially. That is, for the extraction of an iron pickaxe, it is necessary to adhere to a hierarchy. There are also two versions of this environment: in the first version, the reward is received for the item obtained for the first time, and in the Dense version, for each item received.
- *ObtainDiamond*: The main goal of this environment is a diamond, which is the rarest element in Minecraft. This environment is similar to the previous one, however, after receiving the iron pickaxe, the game does not stop, it is also necessary to obtain a diamond. As for *ObtainIronPickaxe*, there are two versions: regular and Dense.

As mentioned above, for the environments *ObtainIronPickaxe* (except diamond) and *ObtainDiamond*, the rewards are given to the agent only for receiving an item. (See Table 1)

It is worth mentioning that, as observations, the agent receives a colored image with resolution (64, 64) for all environments, also in *ObtainDiamond* and *ObtainIronPickaxe*, the agent receives an inventory dictionary, in *Navigate* environment, the value of the compass indicating the target is also obtained. But the space of actions is much more complicated and hybrid, i.e., in addition to discrete values, there are also continuous ones.

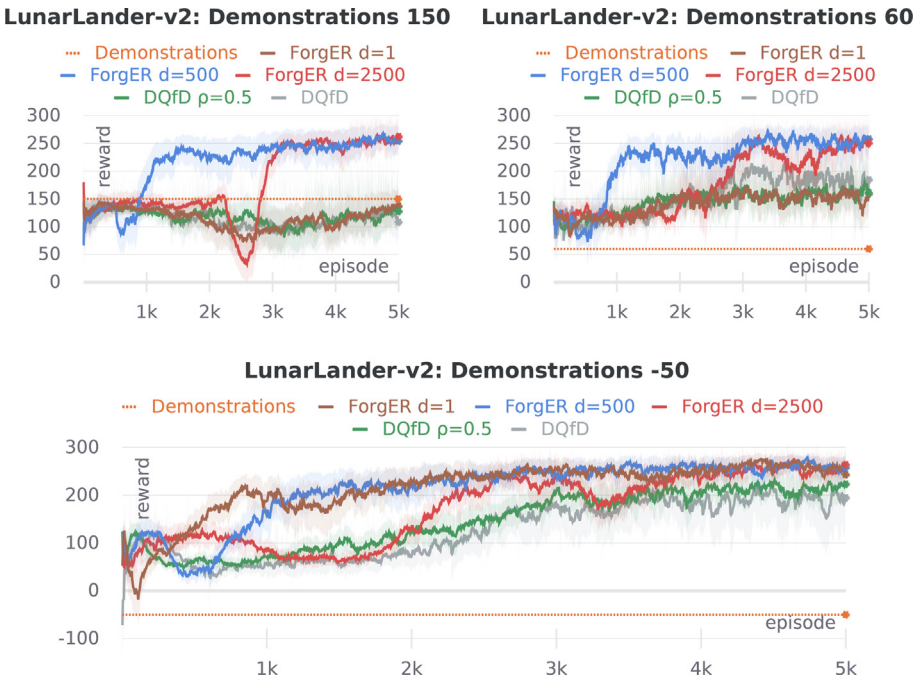

Fig. 5. Top graph on the left shows a mean episode reward with high-level expert data for the ForgER and the DQfD in *LunarLander*. The top graph on the right shows a mean episode reward with medium-level expert data. Due to the wider coverage of states during expert trajectories, the vanilla DQfD approach draws more out of it and gets more rewards, but the linear forgetting still outperforms it. The bottom graph shows a mean episode reward with low-level expert data. This data contains no useful policy at all, so the faster we forget it, the higher our reward is.

Table 1
Rewards for *ObtainDiamond* (left) and the action space for *ObtainDiamond* and *ObtainIronPickaxe* environments (right).

Item	Reward	Action	Type
Log	1	Attack	Discrete(2)
Planks	2	Back	Discrete(2)
Stick	4	Camera	Box(shape=2, [−180, 180])
Crafting table	4	Craft	Enumerated(5)
Wooden pickaxe	8	Equip	Enumerated(8)
Cobblestone	16	Forward	Discrete(2)
Furnace	32	jump	Discrete(2)
Stone pickaxe	32	Left	Discrete(2)
Iron ore	64	NearbyCraft	Enumerated(8)
Iron ingot	128	NearbySmelt	Enumerated(3)
Iron pickaxe	256	Place	Enumerated(7)
Diamond	1024	right	Discrete(2)
		Ssneak	Discrete(2)
		Sprint	Discrete(2)

Moreover *nearbyCraft*, *nearbySmelt*, *craft*, *place*, and *equip* actions can be used only in *ObtainDiamond* and *ObtainIronPickaxe* environments.

4.3.1. Impact of expert data quality

To explore the impact of the expert data quality on the efficiency of off-policy methods, we use a well-known environment *Lunar Lander*. Our experiment consists of three parts with different levels of expert data quality. In the first one, high-level expert trajectories with total rewards from 100 to 200 were taken. The second one contains medium-level trajectories with episode rewards from 0 to 100. The third one contains low-level trajectories with the lunar lander being broken with episode rewards from −100 to 0. These trajectories were taken from one pre-trained agent, picking different trajectories for each part of the experiment. We tested four cases in each part: with a constant demo-ratio 0.5, with complete forgetting after the imitation stage, with forgetting until episode 500, with forgetting until episode 2500 and comparison of these results with the DQfD algorithm and the original pre-trained policy. In the second and third cases, the forgetting rate changed linearly during the training phase. After the imitating phase, we used a sample with 100% of a batch from the expert data buffer. And this rate decreases to 0% to the episode, after which we want to fully forget the expert trajectories. The purpose of these experiments is to prove that forgetting expert data regardless of their quality improves the learning process, and the forgetting rate is crucial in this process. All the experiment curves are a mean of ten separate experiments with the same parameters.

With high quality expert data (see Fig. 5), we can achieve a higher reward by changing the forgetting rate and allocating the agent and the expert data in a replay buffer. In linearly forgetting cases, we can see that a significant improvement appears only after we stop using expert data. When the fraction of the expert data becomes low, for a moment, the performance of an agent drops, but then immediately grows up and overcomes the expert data by far. The longer we train on the expert data, the longer this period is. That happens because when we maintain the expert data, we maintain the expert policy, which is not optimal, and getting out of this local minimum requires more time. But this suboptimal policy guides the agent in the right direction in the early stage, and we can suggest that forgetting the expert data too early is not giving a performance at all. The forgetting rate is a hyperparameter that needs to be configured for each task.

It would seem that with worse expert data, the performance also should be worse, but that is not always true. The worse replay buffer includes a more diverse state and action pairs, so the expert policy is not precise, and the margin loss $L_{PE}(Q^g)$ cannot overfit it. It is like a rookie trying to act like a pro and using advanced skills; the chances that one fails while trying to perform this task is higher than just using mid-level skills with more awareness. That is a case in the vanilla DQfD approach since our approach deals with the right forgetting rate and outperforms others demo-rate approaches.

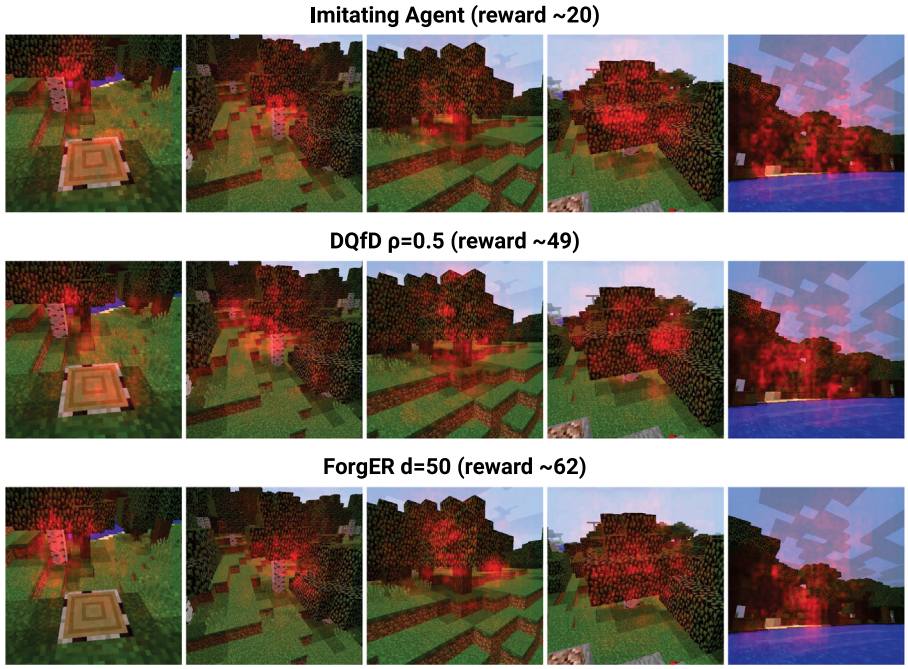

Fig. 6. Salience analysis of the imitating, DQfD, and ForgER agents on *Treechop* with the best discretization. The ForgER agent focusing on important details: the trees' crowns when the agent is far from them and the trunks when the agent is close. Noteworthy is the first frame on which the ForgER, unlike other agents, ignores the stump. If the agent tries to cut it down, then it can dig itself down. It is very difficult for the agent that fell underground to get to the surface, and there are no examples of such behavior in the demonstrations.

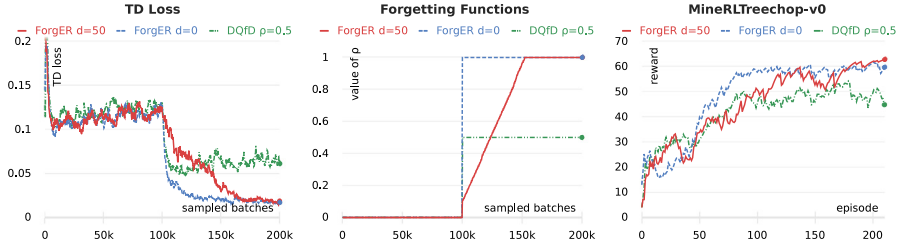

Fig. 7. Several variants of forgetting function $f_{rg}(k)$ realization. The TD loss curve for the ForgER and DQfD agents is shown on the left. The first 100k sampled batches match the imitating phase. The ForgER shows less TD loss than the DQfD approach. The ratio ρ of the sampling expert and the agent data in the replay buffer is indicated in the middle. THe total reward in the *Treechop* environment is shown on the right.

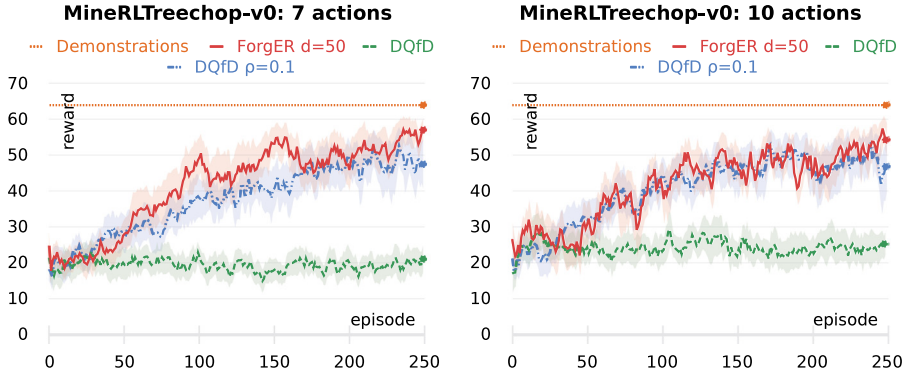

Fig. 8. Left graph shows a mean episode reward for the agents with discretization with seven actions. The right graph shows a mean episode reward for the agents with discretization with ten actions. This is a better discretization, which is why the difference between the ForgER and fixed ratio agent is so insignificant.

4.3.2. Overfitting

In this experiment, we consider the problem of overfitting on the demonstrations in the *Treechop* environment, which leads to the use of the margin loss $L_{PE}(Q^g)$. We created salience maps after the imitating phase. The ForgER agent focuses on important details: crowns and trunks of trees. Paying attention to the crowns of trees when the agent is far from them is a more general strategy, as texture crowns always match, while trunk textures may vary. In most cases, the agent focuses on the nearest tree. The imitation and the DQfD agents pay attention to unimportant details: the blocks that are not related to the tree-chopping task indicate overfitting (see Fig. 6).

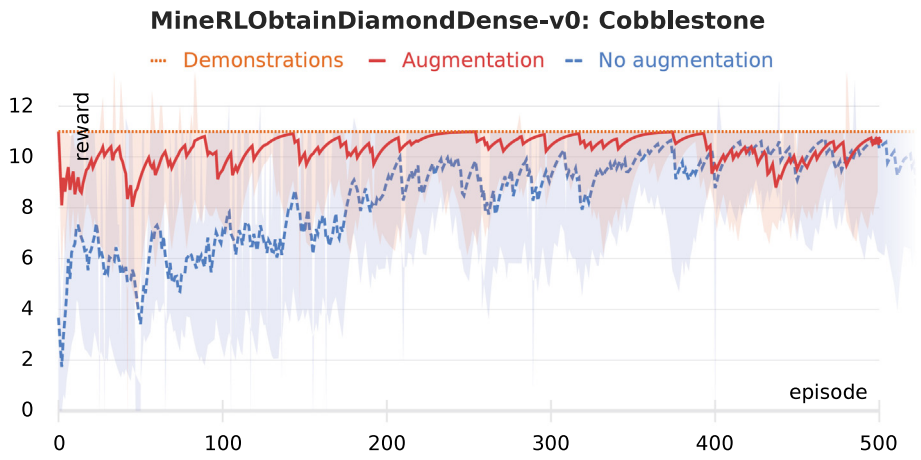

Fig. 9. Mean episode reward for the *Cobblestone* subtask, in which the agent must get 11 cobblestones. The reward for an episode can be higher than 11 if the agent at the last moment picks up several cobblestones at once. Without augmentation, it takes much time for the agent to understand where a cobblestone is.

This fact of overfitting is confirmed by the TD loss diagram in Fig. 7. The TD loss is strongly correlated with the f_{rg} grow parameter and behaves differently after the imitating phase in Fig. 7 in comparison with the DQfD. The TD loss of the ForgER agent after the forging phase is almost twice less than the DQfD one. The ForgER agent shows better performance: 62 vs. 49 for the *Treechop* task.

4.3.3. Discretization

Discretization is a mapping between the continuous action space of the environment and the discrete action space of the agent. However, despite the fact that it lowers the number of actions performed by the agent, it also limits the agent's action space. The discretized actions from expert demonstrations may be inaccurate. To explore how forgetting affects the learning process with different discretization mappings, we trained the agent for *Treechop* with seven and ten actions and with different replay buffer structures. Each discretization was used with frameskip 4. The rotation angle is determined using the sum of four frames. For other actions, the most frequent action was selected.

For each mapping, we trained three configurations of the agent. The first configuration used forgetting the expert data after 50 steps. The second configuration used the DQfD buffer. The third configuration had a fixed ratio $\rho = 0.1$ of the expert data in the batch. Each version of the agent had 150000 pre-train steps and 250 episodes of training in the environment. Human demonstrations are considered as expert data.

As can be seen from the graphs in Fig. 8, when the discretization is not accurate enough to map expert actions, forgetting performs better, and when the discretization is good enough, it does not perform worse even without forgetting.

4.3.4. Augmentations

Task-specific augmentation was evaluated on the *Cobblestone* subtask in the *ObtainDiamond* (dense) environment and human demonstrations as expert data. It was compared with the version of the algorithm without augmentation. Both were averaged across three trials (see Fig. 9).

The agents for the other subtasks were not updated during evaluation. The *Cobblestone* agents had 50000 imitation steps and 50 episodes to forget expert demonstrations. Discretization with ten actions for better behavior cloning was used. Significant dispersion can be explained by a great variety of *Minecraft* worlds, where the agent can appear.

4.3.5. Loss ablation study

In this part of the experimental study, we evaluated the significance of all components of the loss function in two different environments: an environment for which copying behavior is difficult (*Treechop*) and in an environment with high-quality demonstrations (*LunarLander*). This experiment shows comparisons of the ForgER with turned off: TD loss $L_{PDQ}(Q^g)$, n-step TD loss $L_n(Q^g)$, margin loss $L_{PE}(Q^g)$ and L2 regularization loss $L_{L2}(Q^g)$. In a *Treechop* environment (see Fig. 10), disabling these loss functions has a substantial impact on the performance: n-step TD loss (degrade), margin loss (improve). We explain the degrading of the performance with disabled n-step TD loss by the dense reward function of this environment and the imperfectness of the demonstrations after discretization. The agent learns a final policy based more on the rewards that can be proved by the results with disabled margin loss. Disabling margin loss leads to better performance even comparing with the ForgER for forgetting function $d = 50$. Noteworthy, this result is almost identical to ForgER d = 0 in Fig. 7. Moreover, the results for the disabled margin loss during the imitating phase and shortly after it are noticeably worse.

The *LunarLander* environment has a sparse reward function for the goal achievement, so the impact is very different from the previous case (see Fig. 11). Now the agent with disabled margin loss starts with the worst results, and the best results show the agent with the disabled TD loss. In both cases, the ForgER shows reliable results for different types of the reward function, which proves the importance of each part of the loss.

4.4. Hierarchical setting

For the task of obtaining a diamond in MineRL (*hierarchical set*), we demonstrate the ForgER ability to use the extracted subtask graph G and use the goal-oriented organization of the experience replay and data augmentation. In a crucial experiment, we compare our ForgER approach, the ForgER++ modification (a heuristically modified hierarchy of subtasks G) with the best MineRL competition solution [17]. Because it was impossible to recreate the competition's limitations, we reproduced the best solution without restrictions on the training time and the number of steps. The algorithms were tested on 1000 episodes using a common pool of 1000 seeds responsible for the procedural generation of the world and the agent's initial position. This approach allowed the agent to mine a diamond for the first time in the MineRL competition released. As a result, our approach surpassed the best solution of the MineRL competition, which, in its turn,

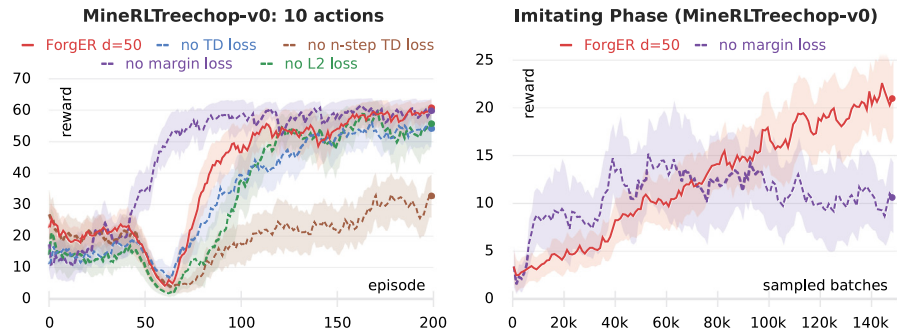

Fig. 10. Rewards of the ForgER $d = 50$ with some losses disabled in the *Treechop* environment with ten action discretization and 150000 imitating steps (left). The evaluation results on the imitating phase for the ForgER and the disabled margin loss agents (right). The ForgER outperforms all other agents during the forging phase except the agent with disabled margin loss, due to specifics of the environment and the quality of the demonstrations. However, disabling this loss leads to worse results during the imitating.

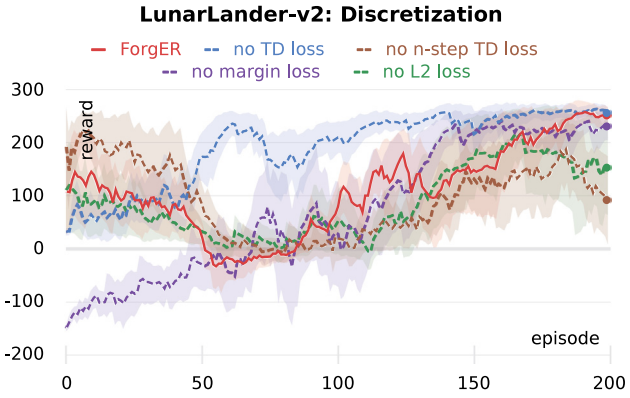

Fig. 11. Rewards of the ForgER (forgetting function: d=50, 10^5 imitating steps) with some losses disabled in the *LunarLander* environment with action discretization. The best performance shows the agent without the TD loss. The agent with the disabled margin loss shows the worst results after the imitating phase.

was the best among all other solutions based on the DQfD, PPO, GAIL, and other algorithms.

For the task of obtaining a diamond in MineRL, we propose the following automatic approach for subtasks extraction. We consider the time of the item appearance in the inventory in chronological order. The sequential items of the same type are combined into an item with quantity. The final subgoal tree can be a sequence obtained from a single trajectory. An example of a marked sequence of subtasks presented here: *log*(6), *planks*(24), *crafting table*(1), *sticks*(4), *wooden pickaxe*(1), …, *iron pickaxe*(1).

To complete the diamond task, the tree diminishes to a chain of subtasks. Each subtask is to obtain the required amount of the indicated resource. The agent receives a pseudo-reward when it obtains an item related to the current subtask (for example, +1, for receiving one *log*). If the agent receives several items at a single step, then the reward of +1 will be added for each one:

$$f_p := \begin{cases} +1, & \text{if acquired item relates to current subtask,} \\ 0, & \text{otherwise.} \end{cases}$$

In addition to a pseudo reward in the environment, pseudo rewards are added to the expert data. We also used the fact that the actions to craft the items are strongly related to the subtasks. The craft actions aimed at solving the subtask were included in each subtask.

Table 2 shows the results of testing three algorithms: the best algorithm presented at the competition (trained from scratch), the ForgER with an automatically extracted chain of subtasks, and the ForgER with a modified chain of subtasks (ForgER++).

Table 2
The results of testing the algorithms on 1000 evaluation episodes. For each item in the row, it shows the number of episodes in which the agent obtained a reward for receiving it. The first column shows the results of the best solution in the MineRL competition.

Item	MineRL	ForgER	ForgER++
Log	859	**882**	867
Planks	805	**806**	792
Stick	718	747	**790**
Crafting table	716	744	**790**
Wooden pickaxe	713	744	**789**
Cobblestone	687	730	**779**
Stone pickaxe	642	698	**751**
Furnace	19	48	**98**
Iron ore	96	109	**231**
Iron ingot	19	48	**98**
Iron pickaxe	12	43	**83**
Diamond	0	0	**1**
Mean reward	57.701	74.09	**104.315**

We used several task-specific settings in the ForgER and the ForgER++. We added a small white noise (mean = 0, std = 0.6) to the camera rotation actions to improve both exploration and behavior of policies after the imitating phase. Also, we used the *log* subtask policy trained in the auxiliary *TreeChop* environment.

4.5. Hyperparameters

Table 3 shows the parameters for the ForgER approach used in all our experiments. Table 4 represents the parameters, that differ for visual and vector environments. The value for the number of the pre-training steps varies for each of the environments.

In contrast to DQfD, we used $l = 0.4$, instead of $l = 0.8$ for the value of L_{PE}, since this showed slightly better results for hierarchical vision-based tasks. Also, we conducted experiments where we replaced margin loss with cross-entropy and KL loss, but it did not make any significant changes. Furthermore, in a part of the experiments, we used the DQfD with a fixed ρ, which is an attempt to modify the approach for imperfect demonstrations.

As a strategy for choosing the forgetting function (or d in particular), we recommend adhering to the following principles:

- If you know that the demonstrations are problematic in any way, you should use fast forgetting (low values of d according to episode length). It is beneficial to control the forgetting process by paying attention to learning and loss curves.
- When you choose the forgetting function, you do not have to re-run imitating phase (use the saved weights).

Generalization or online adjacent of the forgetting function is a future work of our research.

Table 3
Shared parameters used for both environment sets.

Parameter	Value
N-step return weight λ_1	1.0
Margin loss weight λ_2	1.0
L2 regularization weight λ_3	10^{-5}
Expert margin l	0.4
ϵ-greedy initial	0.1
ϵ-greedy final	0.01
ϵ-greedy decay	0.99
Prioritized replay exponent α	0.4
Prioritized replay constant ϵ_a	0.0001
Prioritized replay constant ϵ_d	1.0
Prioritized replay importance sampling exponent β_0	0.6
n-step return	10
batch size	32

Table 4
Parameters used for *simple* and *visual* environments.

Parameter	Value for *simple set*	Value for *visual set*
Target network update period τ	2000	3000
Use noisy layers	False	True
Agent replay buffer capacity	100,000	450,000

5. Related work

Some components in the ForgER are not new. The proper replay buffer sampling policy can solve a wide range of learning process problems. In the prioritized experience replay method [12], the authors suggest picking the samples from the buffer regarding how often these samples are used for backpropagation in the network and based on the values of their temporal difference (TD) loss function.

The idea to store the human experience in the additional expert buffer to increase performance in difficult Atari games was proposed in the Human Experience Replay [30]. More than five hours of the gameplay was stored and this data was sampled as 16 out of 32 examples in the training batch without any pre-training and supervised loss. Due to the modified experience replay, the authors for the first time achieved better than random agent performance in Montezuma's Revenge.

A combination of TD and classification losses in a batch algorithm in a model-free setting is the key components in RL with Expert Demonstrations the (RLED) [31]. The DQfD differs from RLED in that the agent is pre-trained on the demonstration data initially, and the batch of self-generated data grows over time. It is issued as an experience replay to train deep Q-networks. In addition, a prioritized replay mechanism is used to balance the amount of demonstration data in each mini-batch [32]. The DQfD uses a single buffer to store both for the expert and the agent trajectories. These techniques provide the next level of performance in tasks like Montezuma's Revenge and stay as a state-of-the-art algorithm in learning from demonstration. We use the same learning approach but with a different experience replay buffer structuring strategy that helped achieve a significant increase in efficiency.

There are also implementations of algorithms that learn from demonstration based not on the DQN but on policy optimization methods. One of them is the Normalized Actor Critic (NAC) [8], which is almost entirely based on the Soft Actor Critic [33] (SAC). However it has a special normalizing gradient supplement, which prevents overfitting due to the fact that the expert data is often imperfect. As it turns out, the NAC does not work well in vision-based environments. Policy Optimization with Demonstrations (POfD) [29] is an on-policy algorithm; the main difference from the Generative Adversarial Imitation Learning (GAIL) [34] is that the GAIL optimizes the policy to confuse the discriminator, whereas the POfD has a demonstration-guided exploration term in learning objection. The POfD showed massive improvement over the GAIL, using a low number of expert data. The POfD also proved that it is not biased by imperfect data. Despite all these, the GAIL based algorithms are very computationally heavy, and they are very sensitive to changes in the action space.

The most recent research in the field of learning from demonstration is a Recurrent Replay Distributed DQN from Demonstrations (R2D3) [28], which outperformed multiple states of the art baselines. The authors extended the R2D2 algorithm and added an expert data buffer. In conclusion, it was mentioned, that the ratio of the expert data and the agent data is one of the key parameters of their algorithm, and its fine-tuning could significantly increase the results. The R2D3 used only a fixed ratio and established that a small demo ratio is the most desirable case. We extend this investigation and suggest more appropriate ways to treat the demo-ratio as a decreasing variable. We also adapt the approach of using demonstrations to the hierarchical case.

The idea of reusing a faulty experience is proposed in Hindsight Experience Replay [11] (HER) for a multi-goal RL setting. The rollout is stored in the replay buffer with a transformed goal, assuming what the new goal is the state the agent saw in that rollout. The HER can be applied when for each state we can find a goal corresponding to it. The goal transformation is somewhat similar to task-specific augmentation of the ForgER imitating phase. In both cases, we obtain additional data for training.

Increasing the amount of training data is one of the best ways to increase the generalization property of a trained model. Augmentation is a method of increasing the training set based on already available data, which is widely used in machine learning, especially in computer vision. Augmentation of visual observation in reinforcement learning is a fairly new field. Reinforcement Learning with Augmented Data [35] proposes a simple plug-and-play module to enhance algorithms with data augmentations. The authors showed that augmentations can significantly improve the data-efficiency and generalization of RL methods for a wide range of environments. Data-regularized Q [36] (DRQ) introduces a simple regularization technique based on image augmentation, which improves the performance of the SAC approach trained directly from image pixels.

6. Conclusion

We presented the ForgER, a novel algorithm for reinforcement learning from demonstrations in complex partially observable environments including hierarchical settings. We proposed a task-oriented structure of the experience replay buffer with an embedded procedure of forgetting imperfect expert trajectories. By exploiting the hierarchical structure of the demonstrations in case of its availability, we can obtain hierarchical policies that generalize substantially better than the SOTA methods. Additionally, we structured the replay buffer by using augmented data on the imitating phase for each task specified policy. With the ForgER, we achieved two main goals: we attained high sample efficiency by combining a hierarchical approach and using demonstration data, and at the same time, we reduced the quality requirements for these expert trajectories.

We experimentally investigated the ForgER techniques and showed the features of their implementation in the case of different sources of imperfectness in expert data. In our experiments, we demonstrated that the ForgER can solve very complex hierarchical vision-based environments such as *Minecraft*, where we solve the main problem of obtaining diamond in the MineRL setting. In the MineRL competition, various tricks were used to adapt the known approaches to the hierarchical POMDP environment, and only our complex original method helped to achieve the main goal of this competition and surpass all other solutions.

Broadly, we showed that the ForgER outperforms the existing RL approaches based on expert demonstrations, especially those with high-dimensional inputs and hierarchical complex goals. Though we used a simple sequential graph of subgoals, in future work, we aim to explore how more complex subgoal structures with loops can be automatically detected and how they can improve both the imitation and forging phase. In addition, we believe that the main idea of using forgetting experience replay will open doors to incorporating more sophisticated memory-based techniques into RL.

CRediT authorship contribution statement

Alexey Skrynnik: Conceptualization, Methodology, Investigation, Writing - original draft, Writing - review & editing. **Aleksey Staroverov:** Software, Validation, Writing - original draft. **Ermek Aitygulov:** Software, Validation, Visualization. **Kirill Aksenov:** Software, Investigation. **Vasilii Davydov:** Software. **Aleksandr I. Panov:** Conceptualization, Methodology, Formal analysis, Writing - original draft, Writing - review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Russian Science Foundation, project no. 20-71-10116. We gratefully thank the AIM Tech Company for its organizational and computing support, the organizers of the MineRL competition for the proposed challenging task, as well as the anonymous reviewers, whose comments seriously helped to improve this work.

References

[1] D. Yarats, A. Zhang, I. Kostrikov, B.o. Amos, J. Pineau, R. Fergus, Improving sample efficiency in model-free reinforcement learning from images, 2019, arXiv e-prints, arXiv:1910.01741, arXiv:1910.01741.

[2] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, S. Levine, QT-Opt: Scalable deep reinforcement learning for vision-based robotic manipulation, 2018, arXiv e-prints, arXiv:1806.10293, arXiv:1806.10293.

[3] Y. Duan, M. Andrychowicz, B. Stadie, O.J. Ho, J. Schneider, I. Sutskever, P. Abbeel, W. Zaremba, One-shot imitation learning, in: Advances in Neural Information Processing Systems, 2017, pp. 1087–1098.

[4] Y. Zhu, Z. Wang, J. Merel, A. Rusu, T. Erez, S. Cabi, S. Tunyasuvunakool, J. Kramár, R. Hadsell, N. de Freitas, N. Heess, Reinforcement and imitation learning for diverse visuomotor skills, 2018, arXiv e-prints, arXiv:1802.09564, arXiv:1802.09564.

[5] T. Hester, M. Vecerík, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband, G. Dulac-Arnold, J. Agapiou, J.Z. Leibo, A. Gruslys, Deep Q-learning from demonstrations, in: AAAI, 2017.

[6] M. Vecerik, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, M. Riedmiller, Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards, 2017, arXiv e-prints, arXiv:1707.08817, arXiv:1707.08817.

[7] D. Martínez, G. Alenya, C. Torras, Relational reinforcement learning with guided demonstrations, Artificial Intelligence 247 (2017) 295–312.

[8] Y. Gao, J. Lin, F. Yu, S. Levine, T. Darrell, et al., Reinforcement learning from imperfect demonstrations, 2018, arXiv preprint arXiv:1802.05313.

[9] A. Rajeswaran, V. Kumar, A. Gupta, G. Vezzani, J. Schulman, E. Todorov, S. Levine, Learning complex dexterous manipulation with deep reinforcement learning and demonstrations, 2017, arXiv preprint arXiv:1709.10087.

[10] B. Eysenbach, R.R. Salakhutdinov, S. Levine, Search on the replay buffer: Bridging planning and reinforcement learning, in: Advances in Neural Information Processing Systems, 2019, pp. 15220–15231.

[11] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O.P. Abbeel, W. Zaremba, Hindsight experience replay, in: Advances in Neural Information Processing Systems, 2017, pp. 5048–5058.

[12] T. Schaul, J. Quan, I. Antonoglou, D. Silver, Prioritized experience replay, in: Proceedings Ofthe International Conference on Learning Represen- Tations, 2016.

[13] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, 2016, arXiv preprint arXiv:1606.01540.

[14] M. Johnson, K. Hofmann, T. Hutton, D. Bignell, The malmo platform for artificial intelligence experimentation, in: IJCAI, 2016, pp. 4246–4247.

[15] W.H. Guss, C. Codel, K. Hofmann, B. Houghton, N. Kuno, S. Milani, S. Mohanty, D.P. Liebana, R. Salakhutdinov, N. Topin, et al., The MineRL competition on sample efficient reinforcement learning using human priors, NeurIPS Competit. Track (2019).

[16] T. Shu, C. Xiong, R. Socher, Hierarchical and interpretable skill acquisition in multi-task reinforcement learning, 2017, arXiv preprint arXiv:1712.07294.

[17] A. Skrynnik, A. Staroverov, E. Aitygulov, K. Aksenov, V. Davydov, A.I. Panov, Hierarchical deep Q-network from imperfect demonstrations in minecraft, 2019, arXiv e-prints, arXiv:1912.08664, arXiv:1912.08664.

[18] M.L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, John Wiley & Sons, 2014.

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

[20] C.J.C.H. Watkins, P. Dayan, Q-learning, Mach. Learn. 8 (3–4) (1992) 279–292.

[21] D.S.V. Mnih, K. Kavukcuoglu, et al., Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529–533.

[22] H. Van Hasselt, A. Guez, D. Silver, Deep reinforcement learning with double Q-learning, in: Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI), 2016, pp. 2094–2100.

[23] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, N. Freitas, Dueling network architectures for deep reinforcement learning, in: International Conference on Machine Learning, PMLR, 2016, pp. 1995–2003.

[24] A.R. Cassandra, L.P. Kaelbling, M.L. Littman, Acting optimally in partially observable stochastic domains, in: AAAI, vol. 94, 1994, pp. 1023–1028.

[25] M. Piot, B. Geist, O. Pietquin, Boosted bellman residual minimization handling expert demonstrations, in: EuropeanConference on Machine Learning (ECML), 2014.

[26] R.S. Sutton, D. Precup, S. Singh, Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning, Artif. Intell. 112 (1–2) (1999) 181–211.

[27] V. Mnih, A.P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, in: International Conference on Machine Learning, 2016, pp. 1928–1937.

[28] T.L. Paine, C. Gulcehre, B. Shahriari, M. Denil, M. Hoffman, H. Soyer, R. Tanburn, S. Kapturowski, N. Rabinowitz, D. Williams, et al., Making efficient use of demonstrations to solve hard exploration problems, 2019, arXiv preprint arXiv:1909.01387.

[29] B. Kang, Z. Jie, J. Feng, Policy optimization with demonstrations, in: International Conference on Machine Learning, 2018, pp. 2469–2478.

[30] I. Hosu, T. Rebedea, Playing atari games with deep reinforcement learning and human checkpoint replay, CoRR abs/1607.05077 (2016) http://arxiv.org/abs/1607.05077.

[31] B. Piot, M. Geist, O. Pietquin, Boosted bellman residual minimization handling expert demonstrations, in: Machine Learning and Knowledge Discovery in Databases, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 549–564.

[32] T. Hester, M. Vecerík, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, A. Sendonaris, G. Dulac-Arnold, I. Osband, J. Agapiou, J.Z. Leibo, A. Gruslys, Learning from demonstrations for real world reinforcement learning, CoRR abs/1704.03732 (2017) http://arxiv.org/abs/1704.03732.

[33] T. Haarnoja, A. Zhou, P. Abbeel, S. Levine, Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, CoRR abs/1801.01290 (2018) http://arxiv.org/abs/1801.01290.

[34] J. Ho, S. Ermon, Generative adversarial imitation learning, CoRR abs/1606.03476 (2016) http://arxiv.org/abs/1606.03476.

[35] M. Laskin, K. Lee, A. Stooke, L. Pinto, P. Abbeel, A. Srinivas, Reinforcement learning with augmented data, 2020, arXiv:2004.14990.

[36] I. Kostrikov, D. Yarats, R. Fergus, Image augmentation is all you need: Regularizing deep reinforcement learning from pixels, 2020, arXiv:2004.13649.