Soft Actor-Critic with Hindsight Experience Replay

Arne Klages
University Osnabrück
Osnabrück, Germany
aklages@uni-osnabrueck.de

Jan-Luca Schröder
University Osnabrück
Osnabrück, Germany
janlschroede@uni-osnabrueck.de

Erik Nickel
University Osnabrück
Osnabrück, Germany
ernickel@uni-osnabrueck.de

September 17, 2022

Abstract—**Sparse reward settings are one of the biggest challenges in Reinforcement Learning (RL). Hindsight Experience Replay (HER) is one approach to tackle such scenarios. In this project report, we show how HER can be implemented in combination with an Soft Actor-Critic (SAC) agent to solve sparse reward settings. Furthermore, the two most important strategies for goal sampling (*future* and *final* [1]) are compared against a new strategy called *k final*. This is done to provide information about how the *final* strategy compares to *future* if it has a *k* parameter as well, that controls the number of additional relabeled samples added to the *replay buffer*. In our experiments, we were able to show that SAC with HER can solve sparse reward problems that can not be solved with SAC alone. Moreover, SAC+HER performed better than SAC with reward shaping. The results of comparing the different goal sampling strategies show that the *k final* significantly outperforms the simple *final* strategy, but still lags behind the *future* goal sampling strategy.**

Index Terms—**reinforcement learning, deep learning, soft actor-critic (SAC), hindsight experience replay (HER)**

Acronyms

DDPG	Deep Deterministic Policy Gradient
DRL	Deep Reinforcement Learning
ERB	Experience Replay Buffer
HER	Hindsight Experience Replay
MDP	Markov Decision Process
MLP	Multilayer Perceptron
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
SAC	Soft Actor-Critic

I. Introduction

The topic of Reinforcement Learning (RL) has risen in interest a lot in the past years and decades. RL is a specific area of machine learning. It is often used in combination with neural networks. The primary goal here is to achieve the maximal reward in a specific problem, situation, or environment. RL gets applied on different problems, including resource management, recommendations, and other sequential decision-making problems. Deep Reinforcement Learning (DRL) extended the use case of RL using deep artificial neural networks to find policies that solve certain decision-making problems. When solving a problem or task with DRL there are numerous design decisions, some of which are given by the formulation of the problem, some can be decided on by the researcher.

These problems are often formulated as a Markov Decision Process (MDP) (Earliest books: [2], [3]). Here, the neural networks work as approximations for a certain value or alternatively policy functions.

The agent might be given a model of the environment which details the behaviors and dynamics, this type is then called a model-based approach. As there is often no absolute truth model of the environment available, an other approach is called model-free RL. Here, the agent purely learns from its experiences inside the environment. However, both approaches have their own set of advantages and disadvantages. Furthermore, there are on-policy and off-policy algorithms. On-policy algorithms try to maximize the policy, based on the action the actor selected in the current policy. Therefore, the policy used for choosing the action and for updating the values is the same. Examples for such algorithms are proximal policy optimization, SARSA and trust region policy optimization. Off-policy algorithms learn a policy from other actions than the one from our current policy. Therefore, it is independent of the agent's actions and has two policies: one behavior policy for choosing the action (exploration) and one target policy used for the estimation and updating. Examples for such algorithms are deep Q-networks, deep deterministic policy gradient and SAC.

As we advance in the topics of RL and DRL, also the problems become more complex, especially as we look at real-world scenarios, like robotic arms. These learning scenarios are often sparse and have a binary reward, giving only a reward dependent on if the task was successfully completed or not.

Such sparse reward scenarios with binary reward are an important problem in Reinforcement Learning. Classical DRL algorithms are not sufficiently equipped to tackle such problems. Sometimes reward shaping helps to counter sparse reward settings, but this approach can have certain pitfalls as well, as leading to undesired behavior if used improperly, for example (see Problems with Reward Shaping).

Most RL agents can not learn anything from such a scenario. This is where HER comes into play (see Hindsight Experience Replay). Therefore, in this project, we test SAC in combination with HER.

In the following sections we will start with explaining the Related Work, the Background necessary for our approach, we will then explain the Experiments we used, the Implementation and do an Evaluation of results. Finally, we will wrap up everything in the Conclusion.

II. Related Work

Our main focus, for related work, was the paper by Ho et al. (2020) [4]. They proposed an algorithm called SHER (Soft Hindsight Experience Replay) for sparse environments, based on the original HER algorithm. They showed that their algorithm performed better than HER [1] and the related CHER (Curriculum-guided Hindsight Experience Replay) [5].

Prianto et al., (2020) [6] combined SAC with HER for path planning in multi-arm manipulators. With this combination, they tried to combat the issues concerning high dimensionality in DRL problems for multi-arm manipulators. They show that this combination outperforms PRM (probabilistic road map) and TD3 (Twin Delayed Deep Deterministic Policy Gradient) in simulation, but also in two real manipulators. Lee and Moon (2015) [7] also successfully combined SAC and HER to tackle the problem of unmanned aerial vehicles, regarding their navigation and control, achieving the desired outcome quicker and with higher accuracy compared to vanilla SAC

In the paper by Xi et al., (2021) [8], the researchers combined the algorithm HAC (heterogeneous actor-critic) algorithm with HER and tested the algorithm on various benchmark tasks, as well as robotic manipulation tasks. In their testing, they also compared their results to SAC, as well as to some other algorithms and achieved better performance than the vanilla SAC.

In the paper by Yang et al. (2021) [9], a variation of HER was introduced. They developed the algorithm MHER (Multi-step Hindsight Experience Replay), in order to be able to solve multi-goal RL-scenarios more efficiently than the original implementation of HER.

III. Background

In the following, we will introduce the work and methods that we used in this paper. Will we explain and shortly discuss SAC and HER, the two parts of our learning algorithm. Furthermore, we will introduce the two environments that we used in our experiments.

A. Soft Actor-Critic

In [10] SAC was firstly introduced as an off-policy DRL algorithm that optimizes a stochastic policy. For our later following implementation, however, we will orient ourselves and use a modern version of [11]. Our version works in continuous environments, but there are also variants that work in discrete environments. It is related to Deep Deterministic Policy Gradient (DDPG) and stochastic policy optimization approaches. An actor and a critic network are trained, where the actor network learns to act on the policy and the critic network rates the performance. SAC employs entropy regularization as one of its central measures. There is the objective of finding the optimal policy that maximizes expected long term reward and long term entropy [12]. The overarching concept here is called the exploration-exploitation trade-off: It is often beneficial for an agent to explore the environment early on and the use the collected knowledge to exploit the best strategy. The entropy can be seen as a measure of randomness in the policy. It is added to the Q-loss $Q^*(s,a)$ and the value function $V^*(s)$ as the expected future entropy. Specifically for those, SAC uses a soft state-value function ($V(s_t)$) and a soft Q-value function ($Q(s_t,a_t)$). They are defined as:

$$V(s_t) = \mathbb{E}_{a_t \sim \pi}[Q(s_t,a_t) - log\pi(a_t|s_t)]$$
$$Q(s_t,a_t) = r(s_t,a_t) + \gamma \mathbb{E}_{s_{t+1}\sim p}[V(s_{t+1}]$$

(defined in [11])

Here, $V(s_t)$ is the expected sum of the reward for a current state s_t, while $Q(s_t,a_t)$ is the expected sum of the reward for a state-action pair of state s_t and action a_t.

Overall, in SAC one policy π_θ and two Q-functions Q_{ϕ_1}, Q_{ϕ_2} are being learned.

For the Q-functions, we can define the respective loss-functions as:

$$L(\phi_1, D) = E_{(s,a,r,s',d)\ \ D}[(Q_{\phi_i}(s,a) - y(r,s',d))^2]$$

(defined in [11])

The target $y(r,s',d)$ for the loss functions is hereby defined as:

$$\begin{aligned} y(r,s',d) =& r + \gamma(1-d)(\min_{j=1,2} Q_{\phi_{targ,j}}(s',\tilde{a}') \\ & - alog\pi_\theta(\tilde{a}'|s')), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s') \end{aligned}$$

(defined in [11])

The samples for SAC, in order to learn the policy for the value function, are selected according to a squashed Gaussian policy using:

$$\tilde{a}_\theta(s,\xi) = tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \xi), \qquad \xi \sim N(0,I)$$

(defined in [11])

Using the reparametrization trick, we can reshape the formula to get to the final policy optimization we want to achieve, you can find more details to the following in [11]:

$$\begin{aligned} V^\pi(s) &= \underset{a\sim\pi}{E} Q^\pi(s,a) + \alpha H\left(\pi(\cdot|s)\right) \\ &= \underset{a\sim\pi}{E} Q^\pi(s,a) - \alpha \log \pi(a|s) \\ = \max_\theta & \underset{\substack{s\sim\mathcal{D}\\ \xi\sim\mathcal{N}}}{E}[\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)] \end{aligned}$$

(defined in [11])

This final form is an expectation over noise instead of an expectation over actions, with no dependence on other policy parameters, using the minimum of the approximates of the Q-functions to be able to approximate the final policy loss. This is then the term-version, according to which the policy gets optimized to. We will go further into the detail of the exact version of SAC we used in the Implementation-section.

B. Hindsight Experience Replay

HER was introduced in [1]. HER uses experienced episodes to learn from the previous actions for the next episode. It is a sample-efficient replay method, applied to off-policy DRL algorithms. An off-policy algorithm allows learning on interactions with the environment, that are not performed by the current policy of the RL agent. It tries to solve the challenge concerned with sparse and binary rewards in RL. Additionally, it is also very useful for multi-goal scenarios. The idea behind the HER algorithm is that it saves additional transitions into the replay buffer. These are generated with the rewards on goals, which are additionally chosen from the episode. Using this approach, the agent can learn from failed attempts of reaching the goal in order to improve the next attempts. The HER algorithm sets a reached state in a previous transition as an alternative goal to learn from that. This does not help the agent to learn how to reach the initial goal directly, but the agent learns about the environment. Because it can also reach neighboring states of that state, eventually it will reach the goal state by replaying the already obtained outcomes. Therefore, the algorithm will learn, by training on this replay buffer with the positive reward, acquired by the set alternative goals. It will propagate value trough generalization in the state-action-goal space. Henceforth, with HER, the algorithm can learn from both failures and from successes.

C. Goal sampling strategies

The researchers in [1] applied different strategies of how to choose the goal and evaluated the strategies performances against each other. Three of the four strategies used have parameter k, which controls how many relabeled episodes are added to replay.

- The *final* strategy only uses the final state of the environment as a new goal state, there is no parameter k.
- The *future* strategy replays k random states, which were observed after the transition and come from the same episode as the transition
- The *episode* strategy replays k random states, which come from the same episode as the transition being replayed
- The *random* strategy samples k random states from the whole past training for the replay

To compare the performance, [1] looked at three tasks. For each of the tasks, the parameter k was changed. Most of the time, raising k over 8 decreased the performance. Consistently, the *random* strategy yielded the worst performance. While for the performance between the other strategies is not always clearly different, the *future* strategy is the only strategy being able to solve the sliding task almost perfectly. However, in the pushing and pick-and-place task, the highest performance of all strategies is similar and are strategies are able to solve this task almost consistently, except for the *random* strategy.

D. Limitations of Hindsight Experience Replay

The HER implementation described in [1] might have the pitfalls of instability and local optimality as described by [4]. The researchers solved these shortcomings by combining HER with MERL (Maximum Entropy Reinforcement Learning), which led to better performance. They called their approach Soft Hindsight Experience Replay.

In addition, HER can not be used with every environment. Often, variants of HER need to be defined for unsupported environments. As an example, HER cannot be directly applied to linguistic goals [13]. Concluding, HER has some limitations, but there is prior work suggesting improvement in performance when using SAC on an appropriate environment.

E. Problems with Reward Shaping

Reward shaping in an environment agnostic-form is not able to solve the problems reliably. To reach the desired performance, often a high level of domain specific knowledge is needed and the effort of creating the highly specialized rewards might make the training of policy unnecessary compared to hard coding the policy in the first place. This was already discussed in [1].

F. Reacher Environments

In this project, we used two different environments for the experiments.

1) MuJoCo Reacher: In the MuJoCo Reacher [14] environment, a two-jointed robotic arm must position the end point of the arm at a target position in a 2D space (see Figure 1). We used the version ”Reacher-v4” for our tests [14].

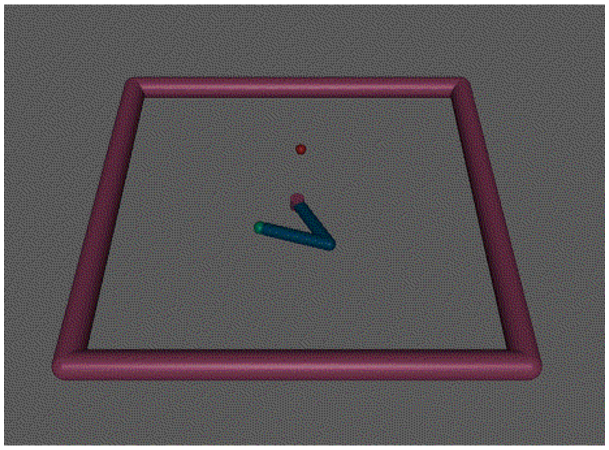

Fig. 1: MuJoCo Reacher environment [14].

2) Panda-gym reach: In the panda-gym reach [15] environment, a robotic arm must position its end point at a target position in a 3D space (see Figure 2). For the sparse reward we used ”PandaReach-v2” and for the dense reward we used ”PandaReachDense-v2”. By default, for all panda-gym environments, the reward is a sparse reward. Therefore, if the agent has reached the desired location the reward is 0, otherwise the reward is -1. The agent has a default tolerance of 5 cm for the goal state ([16]). For the dense reward, the reward is the negative distance between the agent and the goal state ([16]). These sparse and dense rewards were also used in our experiment runs.

IV. Implementation

In this section we discuss the implementation of our approach. The implementation consists of multiple components.

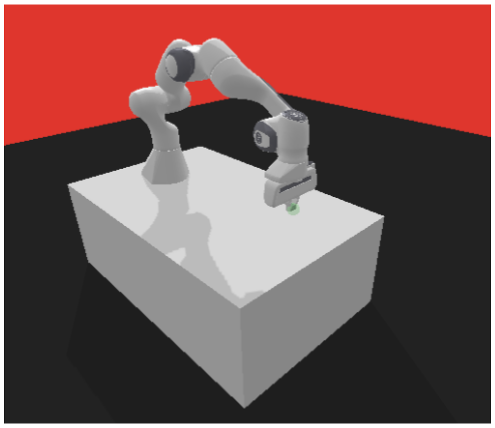

Fig. 2: Panda-gym reach environment [15].

The most important are the SAC agent and its neural networks for the actor and the critic, the HER training module, a standard Experience Replay Buffer (ERB) and environment wrapper classes. Our complete code is available under [17].

A. *Experience Replay Buffer*

The ERB stores transitions from the environment for training. A transition consists of the current environment state, the next action, the subsequent state after taking the action, the resulting reward and a boolean flag that is true if the resulting state is a final state. Transitions can be added with the method `add_transition`. For training of an off-policy RL algorithm, a batch of transitions is randomly sampled from the buffer. This is done with the method `sample_batch`. The size of a batch is specified by the parameter `batch_size`. Since memory space is limited, a buffer can only hold a certain amount of elements. The maximum number of elements that the ERB can hold is determined by the parameter `max_size`. When the maximum size is reached, the oldest items in the replay buffer are replaced by the newly added ones.

B. *SAC+HER*

SAC was introduced in [10] and uses one additional value network besides the actor network and two Q-networks. Our implementation of SAC is a more modern version conforming to the specification of [11], that does not use an additional value network.

HER was first presented by [1], where it was implemented using DDPG to find the optimal policy. There are already some publications on the combination of SAC and HER ([7], [6], [4]), of which we focused on [4].

To solve sparse reward settings, HER replays failed episodes with relabeled goal states. First, our SAC+HER algorithm plays a certain number of episodes in the environment by following the policy of the SAC agent. For each episode, all observed transitions are added to an ERB. Next, a set of additional goals are determined for each transition in the episode. For each sampled goal, the desired goal of the transition is replaced by the new goal and then added to the replay buffer. Finally, a fixed number of SAC update steps are performed. The whole procedure is carried out for a given amount of cycles and epochs. Our SAC+HER version is displayed in algorithm 1.

First, in an update step of SAC, a batch of transitions is sampled from the ERB. For each network, a gradient update is done based on the batch. Our implementation of SAC uses two Q-networks. To determine a Q-value for a state action combination, the minimum of the resulting values of the two networks is used. This is done to limit the overestimation bias of Q-networks. The update formulas of the Q-networks are displayed in lines 19 and 20 in algorithm 1. The other type of network that needs to be trained is the actor (policy) network. This network is updated using the formula in line 21. To balance the weight of reward and entropy, a parameter α is used in the updates. The original HER algorithm from [1] uses a reward scaling factor, the reward is multiplied by. The α parameter should be the inverse of the original reward scaling.

For sampling actions, SAC uses a normal distribution, that is also used to calculate the log probabilities of the corresponding actions. Action values are squashed by a *tanh* function to stay in an interval between -1 and 1. Due to the squashing, the log probabilities of the actions need to be adjusted to the new interval. The Python code for implementing the sampling of actions and calculating their log probabilities in TensorFlow 2 is shown in listing 1. Before action values are passed to the environment, they need to be scaled appropriately if the environment requires values other than those from the -1 to 1 interval.

There are different ways to sample a set of additional goals, these are described in subsection III-C. We implemented the *future* and the *final* strategies from [1]. In addition, we implemented a new goal sampling strategy called *k final*. This is basically the combination of the *final* goal sampling with the additional parameter *k* that controls the number of relabeled transitions in the ERB.

C. *Environment Wrappers*

For our evaluation, we need sparse binary reward environments. This is given in the panda-gym reach environment. To achieve this in the MuJoCo Reacher environment, a wrapper class is created that converts the dense shaped rewards into sparse binary rewards. The rewards are calculated according to the formula:

$$r_t = r_g\left(s_t, a_t, g\right) = \begin{cases} 0, \text{ if } |s_t - g| < \delta \\ -1, \text{ otherwise} \end{cases} \quad [1]$$

Thereby, the goal is considered fulfilled if the distance between the desired goal (point to be reached) and the achieved goal (position of the Reacher's fingertip) is smaller than a δ. For the reacher environment, we use $\delta = 0.015$ as default. If the goal is satisfied, a reward of 0 is returned, otherwise a reward of -1. The Python implementation can be found in the method `reward` in listing 2.

Additionally, HER needs the desired goal and the achieved goal in the current time step from the environment. This information is encoded in the Reacher's observation data. To extract

Algorithm 1: SAC+HER

```
Given:
- initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$
- a strategy $\mathbb{S}$ for sampling goals for replay
- a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \rightarrow \mathbb{R}$
1  Set target parameters equal to main parameters: $\phi_{targ,1} \leftarrow \phi_1, \phi_{targ,2} \leftarrow \phi_2$
2  Initialize replay buffer $R$
3  for epoch=1, E do
4      for cycle=1, C do
5          for episode=1, M do
6              Sample a goal $g$ and an initial state $s_0$
7              for t=0, T-1 do
8                  Sample an action $a_t$ using the policy $\pi_\theta$: $a_t = \pi_\theta(s_t)$
9                  Execute action $a_t$ and observe new state $s_t$
10             for t=0, T-1 do
11                 $r_t := r(s_t, a_t, g)$
12                 Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g, d_t)$ in $R$
13                 Sample a set $G$ of additional goals for replay
14                 for $g' \in G$ do
15                     $r' := r(s_t, a_t, g')$
16                     Store the transition $(s_t||g', a_t, r_t, s_{t+1}||g', d_t)$ in $R$
17         for t=1, N do
18             Randomly sample a batch $B$ of transitions from $R$
19             Compute the targets for the Q-functions:
                 $y(r, s', d) = r + \gamma(1-d)(\min_{j=1,2} Q_{\phi_{targ,j}}(s', \tilde{a}') - \alpha log\pi_\theta(\tilde{a}'|s')),\ \tilde{a}' \sim \pi_\theta(\cdot|s')$
20             Update Q-functions by one step of gradient decent using
                 $\nabla_{\phi_i} \sum \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\phi_i}(s,a) - y(r,s',d))^2$, for $i = 1, 2$
21             Update policy by one step of gradient assent using
                 $\nabla_\theta \sum \frac{1}{|B|} \sum_{s \in B} (\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha log\pi_\theta(\tilde{a}_\theta(s)|s)))$, Where $\tilde{a}_\theta(s)$ is a sample form $\pi_\theta(\cdot|s)$
                 which is differentiable wrt $\theta$ via the reparametrization trick.
22             Update target networks with $\phi_{targ,i} \leftarrow (1-\rho)\phi_{targ,i} + \rho\phi$, for $i = 1, 2$
```

```
@tf.function
def sample_actions_form_policy(
        self, state):
    mu, sigma = self._actor(state)
    distribution = tfd.Normal(mu, sigma)
    actions = distribution.sample()
    log_probs = distribution.log_prob(
      actions)
    actions = tfm.tanh(actions)
    log_probs -= tfm.log(1 - tfm.pow(
      actions, 2) + 1e-6)
    log_probs = tfm.reduce_sum(
      log_probs, axis=-1, keepdims=True)
    return actions, log_probs
```

Listing 1: The SAC agent's method of sampling and squashing actions and their log probabilities form the policy.

the desired goal and the achieved goal from the observation, the wrapper class includes the functions `achieved_goal` and `desired_goal`. The desired goal (the coordinates of the desired point to be reached) can be extracted directly from the observation. In contrast, the achieved goal has to be calculated because the observation does not contain the coordinates of the Reacher's fingertip. Instead, it includes the vector from the fingertip to the target goal. In HER goal states are altered, to provide this functionality, the method `set_goal` is created. It takes an observation and a goal to be set. Because the Reacher observation contains the fingertip to target goal vector, this information must also be adjusted to the new goal. Listing 2 contains the Python code of the Reacher environments wrapper methods.

D. Networks

Our implementation of an SAC agent requires two neuronal networks for training, an actor network and a critic network. For the reach tasks we used a classical Multilayer Perceptron (MLP), consisting of three hidden layers with 256 neurons.

```
def desired_goal(self, state):
  return state[4], state[5]

def achieved_goal(self, state):
  x_t, y_t = self.desired_goal(state)
  x_v, y_v = state[8], state[9]
  x_g, y_g = x_v + x_t, y_v + y_t
  return x_g, y_g

def set_goal(self, state, goal):
  x_g, y_g = goal
  x, y = self.achieved_goal(state)
  new_state = np.array(state)
  new_state[4], new_state[5] = x_g, y_g
  new_state[8], new_state[9] = (x - x_g,
                                y - y_g)
  return new_state

def reward(self, state):
  return -1 if np.linalg.norm(
    [state[8], state[9]]) \
              > self._delta else 0
```

Listing 2: Methods of the sparse Reacher 2d wrapper class.

Each hidden layer uses an Rectified Linear Unit (ReLU) activation function. The critic network is a Q-network that predicts the Q-value of the given state and action pair. Before it is fed into the MLP, state and action are concatenated. The output layer consists of a single neuron with a linear activation function. SAC uses a stochastic policy based on a normal distribution. This requires the mean μ and standard deviation σ as parameters. To determine these parameters, the actor network is given a state for which the best next action should be predicted. Two layers with a single neuron are used as output layers for μ and σ. The layer for μ has a linear output function and the layer for σ has a *softplus* output function. *Softplus* is chosen for σ because the standard deviation needs to be a value greater than zero. The Python code for creating the actor and critic network is shown in listing 3.

```
def create_policy_network(learning_rate,
                          state_dim,
                          action_dim):
  inputs = keras.Input(shape=state_dim)
  x = Dense(256,
          activation=tf.nn.relu)(inputs)
  x = Dense(256, activation=tf.nn.relu)(x)
  x = Dense(256, activation=tf.nn.relu)(x)
  mu = Dense(action_dim,
           activation=None)(x)
  sigma = Dense(
    action_dim,
    activation=tf.nn.softplus)(x)
  model = keras.Model(inputs=inputs,
                      outputs=(mu, sigma))
  model.compile(optimizer=Adam(
    learning_rate=learning_rate))
  return model

def create_q_network(learning_rate,
                     state_dim,
                     action_dim):
  inputs_s = keras.Input(shape=state_dim)
  inputs_a = keras.Input(shape=action_dim)
  x = Concatenate()([inputs_s, inputs_a])
  x = Dense(256, activation=tf.nn.relu)(x)
  x = Dense(256, activation=tf.nn.relu)(x)
  x = Dense(256, activation=tf.nn.relu)(x)
  out = Dense(1, activation=None)(x)
  model = keras.Model(inputs=(
    inputs_s, inputs_a), outputs=out)
  model.compile(optimizer=Adam(
    learning_rate=learning_rate))
  return model
```

Listing 3: Functions to create the actor (policy network) and the critics (Q-networks) for the SAC agent.

V. Experiments

To evaluate the performance of SAC+HER, we compare it to pure SAC in sparse binary reward environments and with SAC in dense reward-shaped settings. This test is performed in the panda-gym reach environment [15].

In a second experiment, we compare how different goal sampling strategies impact the performance of SAC+HER. In [1], the authors introduced different strategies for goal sampling. Their findings suggest that the *future* goal sampling strategy yields the best results. With the exception of *final*, all their goal sampling strategies have a k parameter that controls how often a created episode is relabelled and added to the *replay buffer*. This leaves the question of how good the *final* strategy would be compared to *future* if it also had a k parameter. This experiment aims to answer this question by comparing our newly introduced k *final* goal sampling strategy with the previously presented *final* and *future* strategies. These experiments are conducted in the MuJoCo Reacher environment (Figure 1) in combination with our sparse reward wrapper class.

In the experiments, each test instance runs for 40 epochs. One epoch comprises 50 cycles. In each cycle, 16 episodes are created, and 40 update steps are performed at the end. This structure was also used in [1]. Additionally, one episode has a maximum of 50 steps in the environment.

Per experiment setup, 5 test runs were carried out. At the end of each epoch of a test instance, 500 evaluation episodes are performed to calculate the average return and success rate of that epoch.

For the evaluation, actions are chosen deterministically by simply taking the μ from the actor network as the action (and ignoring σ) as recommended in [10].

The results for SAC without HER are obtained by using the same HER algorithm, but without sampling additional goals, which is basically pure SAC.

For most hyperparameters, we used widespread default choices (see table I). The entropy regularisation coefficient (α) was set to 0.05, as recommended in [4] for the combination of SAC+HER. For the k parameter of the goal sampling strategies, [1] suggests a k of 4 or 8 depending on the environment. In our experiments, we set k to 4.

Parameter	Value
optimizer	Adam
learning rate	0.0003
discount factor (γ)	0.99
polyak averaging factor (τ)	0.005
entropy regularisation coefficient (α)	0.05
replay buffer size	10^6
number of hidden layers	3
number of units per hidden layer	256
nonlinearity of hidden layers	ReLU
samples per minibatch	256

TABLE I: Hyperparameters

VI. Evaluation

In the following, we will go in detail over the results of our experiments and evaluate our findings. The detailed implementation and setup of the experiments in the two environments is explained in section IV. For this, we compared the performance of SAC and SAC+HER in the panda-gym reach environment ([15], Figure 2), which is presented in Figure 3. Additionally, we compared the performance of different goal sampling strategies with SAC+HER in the MuJoCo Reacher environment [14] with our sparse reward wrapper (described in the subsection Environment Wrappers). The results are visualized in Figure 5 and Figure 4. Each experiment and setup ran five times over 40 epochs.

A. *Comparison of SAC+HER sparse vs. SAC sparse vs. SAC dense*

To have a sufficient baseline comparison for our SAC+HER approach, we choose vanilla SAC with sparse rewards and reward shaping. The reward shaping was realized by using a dense environment, compared to the sparse environment used in the other two approaches. Specifically, we used the panda-gym reacher environment [15], which is by design a sparse problem. We expected vanilla SAC not to perform well in this environment and both SAC with reward shaping and SAC+HER to outperform this basic approach. SAC+HER uses the best sampling strategy from the original HER paper [1], which is the *future* goal sampling strategy with a k of 4.

We display our results in Figure 3. It provides the success rate of the approaches over the 40 epochs. SAC+HER is able to solve the task after the first epoch consistently and is stable (with a success rate of 100%) in each following epoch. Pure SAC with reward shaping similarly also manages to achieve very high success rates after the first epoch, however the success rates are less stable (alternating in the range between 89% and 100%) than SAC+HER. Pure SAC with sparse rewards just rarely (with a maximum success rate of

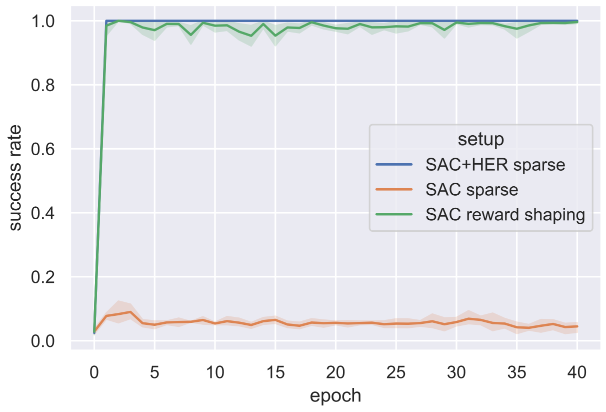

Fig. 3: Comparison of success rates of SAC+HER with sparse binary rewards, pure SAC with sparse binary rewards and SAC with dense shaped rewards over 40 epochs in the panda-gym reach environment. For each test setup, 5 test runs were carried out and the results are displayed with their mean and error interval (maximum and minimum value).

16.2%) manages to complete the environment within the 40 epochs.

To summarize, while pure SAC with reward shaping manages to solve the environment in most cases, SAC+HER outperforms it by reaching consistent 100% success rates after the first epoch. This is in line with our expectations beforehand, as SAC+HER was shown to outperform pure SAC [7], [6], [4].

These findings are in line to what we would expect given that HER profit of exploration that is performed with SAC, given the sparse environment, vanilla SAC would take longer to succeed and does not make meaning full progress in our experiment never reaching a success rate of over 17% and only reaching over 10% success rate in 10% of the episodes, which can not be counted as achieving a stable good success rate.

B. *Comparison of final vs. k final vs. future goal sampling*

Regarding the different goal sampling strategies of SAC+HER in the MuJoCo Reacher [14] environment, we expected k *final* to outperform *final*. In order to be able to further evaluate the performance of k *final*, we also compared in to the performance of *future*, the goal sampling strategy that was previously shown to be the most performant in [1]. This is especially important, as we wanted to test, if *final* also using a parameter k would perform similar to the *future* strategy.

Figure 4 provides the success rates of SAC+HER using either *final*, k *final*, or *future* as the respective goal sampling strategies. For our testing of the goal sampling strategies with SAC+HER, we used a sparse reward wrapper for the MuJoCo's reacher [14] environment. As mentioned before, we selected a k of 4 for both the *future* and the k *final* strategies.

Figure 4 overall shows, that SAC+HER, using *future* as the goal sampling strategy, outperforms both k *final* and

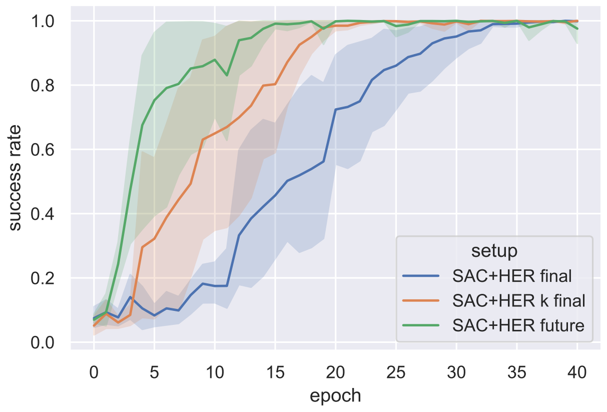

Fig. 4: Comparison of success rates of SAC+HER with different goal sampling strategies (*final*, *k final* and *future*) over 40 epochs in the MuJoCo Reacher environment with our sparse reward wrapper. The *k final* and *future* both use 4 as *k* parameter. For each test setup, 5 test runs were carried out and the results are displayed with their mean and error interval (maximum and minimum value).

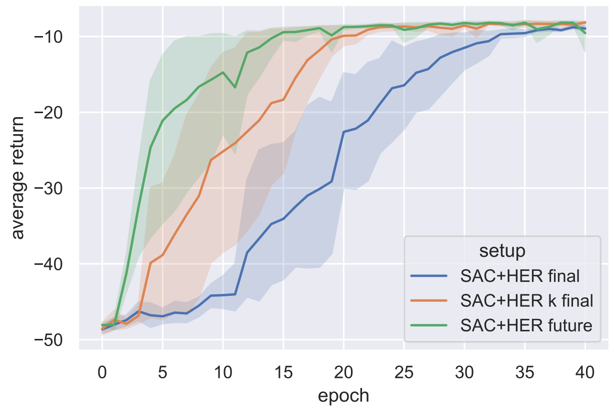

Fig. 5: Comparison of the average returns of SAC+HER with different goal sampling strategies (*final*, *k final* and *future*) over 40 epochs in the MuJoCo Reacher environment with our sparse reward wrapper. The *k final* and *future* both use 4 as *k* parameter. For each test setup, 5 test runs were carried out and the results are displayed with their mean and error interval (maximum and minimum value).

final regarding the success rates. It reaches higher success rates in way earlier epochs than the other two goal sampling strategy. After the 15th epoch, SAC+HER using *future* almost consistently reaches a success rate of 100%. SAC+HER using the *k final* goal-sampling strategy outperforms the *final* strategy after the 4th epoch, reaching a success rate of 100% earlier than *final*. In the first four epochs, *final* and *k final* perform very similar. However, it reaches comparable success rates only in later epochs than *future*.

Figure 5 shows the average return for SAC+HER using one of the three goal sampling strategies *future*, *k final* or *final*. It also shows that, in respect to the average returns, *k final* outperforms *final* regarding both speed and maximal achieved average return after the first 4 epochs, where they perform equally. *k final* achieves the maximal value after the epochs 13 to 23 in all five runs almost consistently, while *final* only reaches such values at the end of the 40 epochs. However, *k final* still gets outperformed by *future*, reaching a higher average return significantly faster (see Figure 5)

While *k final* outperforms *final* both in terms of success rates and average returns significantly, it still gets outperformed by *future* in both of these categories. This is in line with our initial expectations regarding the performance of *k final* over *final*. While *k final* gets outperformed *future*, it still performs significantly better than *final*. Therefore, with the introduction of the parameter *k*, we managed to yield better performance than *final*, but it still falls short behind *future*, which also uses the parameter *k*.

VII. Conclusion

In this project report, we explain how HER can be implemented in combination with an SAC agent to solve sparse reward settings.

An experiment was conducted that shows the advantage of SAC+HER over pure SAC in both, sparse binary and dense reward shaped environments. This experiment shows that HER with SAC can solve sparse reward problems that can not be solved with SAC alone. HER+SAC also performed better than SAC with reward shaping, nevertheless SAC is capable of solving the tested environment in most cases with the help of reward shaping.

In a second experiment, the *future* and *final* strategies for goal sampling were compared with our newly introduced *k final* strategies to provide information on how the *final* strategy compares to *future* if it also has a *k* parameter. The results suggest that the *k final* outperforms the simple *final* strategy significantly, but still falls short against the *future* goal sampling strategy.

Due to time constraints, both tests were conducted in just one environment. In further studies, the influence of the *k* parameter on the *final* goal sampling strategy could be investigated in more complex environments with varying ranges of the *k* parameter.

References

[1] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, “Hindsight experience replay,” 2017. [Online]. Available: https://arxiv.org/abs/1707.01495

[2] R. Bellman, “Dynamic programming,” *Science*, vol. 153, pp. 34 – 37, 1966.

[3] R. A. Howard, “Dynamic programming and markov processes.” John Wiley, 1960.

[4] Q. He, L. Zhuang, and H. Li, “Soft hindsight experience replay,” 2020. [Online]. Available: https://arxiv.org/abs/2002.02089

[5] M. Fang, T. Zhou, Y. Du, L. Han, and Z. Zhang, “Curriculum-guided hindsight experience replay,” in *NeurIPS*, 2019.

[6] E. Prianto, M. Kim, J.-H. Park, J.-H. Bae, and J.-S. Kim, “Path planning for multi-arm manipulators using deep reinforcement learning: Soft actor–critic with hindsight experience replay,” *Sensors*, vol. 20, no. 20, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/20/5911

[7] M. H. Lee and J. Moon, “Deep reinforcement learning-based uav navigation and control: A soft actor-critic with hindsight experience replay approach,” 2021. [Online]. Available: https://arxiv.org/abs/2106.01016

[8] B. Xi, R. Wang, Y. Cai, T. Lu, and S. Wang, “A novel heterogeneous actor-critic algorithm with recent emphasizing replay memory,” *Int. J. Autom. Comput.*, vol. 18, pp. 619–631, 2021.

[9] R. Yang, J. Lyu, Y. Yang, J. Ya, F. Luo, D. Luo, L. Li, and X. Li, “Bias-reduced multi-step hindsight experience replay for efficient multi-goal reinforcement learning,” 2021.

[10] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” 2018. [Online]. Available: https://arxiv.org/abs/1801.01290

[11] “Soft actor-critic openai.” [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/sac.html

[12] D. Seita, “Soft Actor Critic—Deep Reinforcement Learning with Real-World Robots,” 2018. [Online]. Available: http://bair.berkeley.edu/blog/2018/12/14/sac/

[13] G. Cideron, M. Seurin, F. Strub, and O. Pietquin, “Higher: Improving instruction following with hindsight generation for experience replay,” *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 225–232, 2020.

[14] “Openai gym mujoco reacher.” [Online]. Available: https://www.gymlibrary.dev/environments/mujoco/reacher/

[15] “panda-gym.” [Online]. Available: https://panda-gym.readthedocs.io/en/latest/usage/environments.html

[16] Q. Gallouedec, N. Cazin, E. Dellandréa, and L. Chen, “panda-gym: Open-source goal-conditioned environments for robotic learning,” 2021.

[17] “Git project repository.” [Online]. Available: https://gitlab.gwdg.de/uos_ikw_drl_2022_group37/homework/-/tree/main/final_project