Re-implementing PAIRED for Frozen Lake

Seminar: 8.3487 Deep reinforcement learning (SoSe 2022)
Prof. Dr. Elia Bruni, Prof. Dr. Gordon Pipa, Leon Schmid

Nicole Rogalla
Cognitive Science
Universität Osnabrück
Osnabrück, Germany
nrogalla@uni-osnabrueck.de

Jens Huth
Cognitive Science
Universität Osnabrück
Osnabrück, Germany
jehuth@uni-osnabrueck.de

Nils Niehaus
Cognitive Science
Universität Osnabrück
Osnabrück, Germany
nniehaus@uni-osnabrueck.de

30.09.2022

Abstract **- The idea of curriculum learning (CL) has gained popularity in reinforcement learning (RL) research recently. The aim is to train an RL agent sequentially based on a curriculum with tasks of increasing difficulty. One paper exploring the possibilities of CL is "PAIRED: A New Multi-agent Approach for Adversarial Environment Generation"[1] where three RL agents interact: the adversary building the Multi-grid environment, the antagonist ensuring it is solvable and the protagonist - the agent one is trying to train. The adversary aims to maximize the antagonist reward while minimizing the protagonists reward. This paper tried to re-implement their approach for the Open-AI Gym Frozen Lake.**

Index Terms **- Reinforcement Learning, Curriculum Learning, Frozen Lake Gym, PAIRED, Multi-agent Environment**

1 Introduction

Reinforcement learning (RL) is already known for its resemblance to human-kind learning. Every child learns through rewards which influence the behaviour until the desired result is achieved. RL copies that approach with an agent learning through feedback from the environment to excel at a certain task. The society has structured human learning through the education system, where every grade has to be passed before moving onto the next stage. This establishes a curriculum intended to fit to a child's developmental state which leads to a gradual build-up of knowledge and skills. A similar approach was lately also undertaken in RL and simply named curriculum learning. The agent is stepwise challenged with more complex tasks after having accomplished earlier "easier" steps. Jaques and Dennis (2021) are building upon this approach in their blog post "PAIRED: A New Multi-agent Approach for Adversarial Environment Generation"[1] which describes the essence of the paper "Emergent Complexity and Zero-shot Transfer via Unsupervised Environment Design "[2] by Dennis et al. (2021). Dennis et al. (2021) choose "PAIRED" to stand for Protagonist Antagonist Induced Regret Environment Design and already hint towards three interacting RL-agents: the adversary is building the environment, the antagonist and protagonist both try to reach the goal. The protagonist is the focus, as it should learn to solve the environment, while the antagonist makes sure that the environment is solvable, i.e. that there is a path to the goal. This approach implicitly creates a curriculum as the adversary is incentivized to create the most simple solvable environment that the protagonist can not yet solve. The antagonist, however, is allied with the adversary which tries to maximize and minimize

rewards of the antagonist and protagonist respectively. The general process of *PAIRED* [2] can be seen in Figure 1. While their paper is build upon the Multi-grid environment, we aim to re-implement their approach for an adapted version of the Open-AI Gym Frozen Lake. Since this is a rather recent paper, we have not found any other replication of their approach.

Algorithm 1: PAIRED.

Randomly initialize Protagonist π^P, Antagonist π^A, and Adversary $\vec{\Lambda}$;
while *not converged* **do**
  Use adversary to generate environment parameters: $\vec{\theta} \sim \vec{\Lambda}$. Use to create POMDP $\mathcal{M}_{\vec{\theta}}$.
  Collect Protagonist trajectory τ^P in $\mathcal{M}_{\vec{\theta}}$. Compute: $U^{\vec{\theta}}(\pi^P) = \sum_{i=0}^{T} r_t \gamma^t$
  Collect Antagonist trajectory τ^A in $\mathcal{M}_{\vec{\theta}}$. Compute: $U^{\vec{\theta}}(\pi^A) = \sum_{i=0}^{T} r_t \gamma^t$
  Compute: $\textsc{Regret}^{\vec{\theta}}(\pi^P, \pi^A) = U^{\vec{\theta}}(\pi^A) - U^{\vec{\theta}}(\pi^P)$
  Train Protagonist policy π^P with RL update and reward $R(\tau^P) = -\textsc{Regret}$
  Train Antagonist policy π^A with RL update and reward $R(\tau^A) = \textsc{Regret}$
  Train Adversary policy $\vec{\Lambda}$ with RL update and reward $R(\tau^{\vec{\Lambda}}) = \textsc{Regret}$
end

Figure 1: Pseudo-Code for PAIRED algorithm, Adapted from "Emergent Complexity and Zero-shot Transfer via Unsupervised Environment Design," by M. Dennis, N. Jaques, E. Vinitsky, A. Bayen, S. Russel, A. Critch, S. Levine, 2021, 34th Conference on Neural Information Processing Systems

2 Method

An open-source implementation of Dennis et al. (2021) is freely available[1], therefore we could build upon their implementation structure. After gaining an overview of their structure we divided the project into four major components: extending the Frozen Lake environment, implementing suitable RL-agents, implementing the *PAIRED* [2] algorithm and training and visualizing our results. The full project code can be found in the GitHub repository[2].

2.1 Extending the environment

The Frozen lake environment provided by Open-AI Gym[3] features an agent able to move from tile to tile to reach a goal while trying to avoid falling into holes. The first step in our project was to adapt the Frozen Lake environment to be suitable for being build by an adversary agent. This especially involved adding a step_adversary and reset_agent function. Our first approach included making the environment a multi-agent environment in line with their Multi-Grid environment. While we later realised that this functionality was not strictly necessary for our implementation of *PAIRED* [2] , we still decided to keep its possibilities. The multi-agent environment for the Frozen Lake environment is implemented in multifrozenlake.py. The map generation is implemented in adversarial.py. The adversary can build the environment either by placing each object (goal, agent, frozen plate and hole) on free spaces in the map (AdversarialEnv) or by going through each empty tile in the map and deciding which object to place (ReparametarizedAdversarialEnv). Figure 2 displays an example environment build by the adversary.

To reset the environment without resetting the map, the adversarial environment has a reset_agent() function. Additionally, it would be possible to restrict the agents view to just a few tiles in front of it by setting the agent_view_size.

Figure 2: Example Environment

[1] https://github.com/google-research/google-research/tree/master/social_rl/adversarial_env

[2] https://github.com/nrogalla/Group12_DRL_Project

[3] https://github.com/openai/gym/blob/master/gym/envs/toy_text/frozen_lake.py

2.2 RL-agent development

Our agent reinforcement learning architecture of choice is Proximal Policy Optimization (PPO) implemented with the Advantage Actor-Critic(A2C) algorithm using a Generalized Advantage estimation (GAE). For each episode, we would get the actions by sampling from the probabilities of the four actions, initialised with 0.25, using the action to make a step and save the old agent position, the action, the reward, the done flag, the value of the next state, the probabilities from which the action was sampled and the map as seen by the agent into a memory. If the agent reaches a done state, we would also append the value for the last state to calculate the Generalized Advantage Estimation. For the memory we created a class called Buffer, which also includes methods to calculate the matching discounted rewards and advantages for that episode. We calculated the discounted return with the usual formula:

$$\sum_t = \gamma^t * R_t$$

For the advantage calculation we initialised gae with 0 and used:

$$\delta_t = rewards_t + \gamma * value_{t+1} * done_t - values_t$$

$$gae_t = \delta_t + \gamma * \lambda * maskvalue * gae$$

The mask value depends on whether the game is over or not, meaning that if we finish an episode, the GAE is reset by it. Then we normalized the advantages by subtracting the mean and dividing through the standard deviation. With these variables now collected, we trained the agent, calculating the actor loss, critic loss and total loss:

$$a = action, adv = advantage$$
$$p = probability, o = oldprobability$$

$$criticloss = MSE(discountedreturns, values)$$

For t in time-steps:

$$ratio = \frac{p_t(a)}{o_t(a)}$$
$$clipped = clipby(ratio, 1-\epsilon, 1+\epsilon)$$
$$s1 = ratio * adv_t$$
$$s2 = clipped * adv_t$$
$$actorloss = minimum(s1, s2)$$

$$loss = mean(actorlosses) + criticloss - 0.001 * entropy$$

We then proceeded to use the critic loss for the critic network and the total loss for the actor network. As optimizers to apply the gradients we used *ADAM* [3] and a learning rate of 0.0001. We encountered multiple problems when implementing the agent. The major one was instead of learning the best action for every individual state, the agents learned a general policy for all states. This of course led to only a small learning effect and the agent barely ended up finding the goal.

2.3 Networks

Two neural networks - one for the critic and one for the agent - build the basis of our learning algorithm. The input into both networks are both the map and the agents position in the map (see Figure 3). The map can be either the whole map or a map restricted to the agents view. To input the state into our neural networks, they need to be converted into on-hot lists. This conversion can be seen in the following code snippet: For all networks we used one dense layer for the position and a convolutional two dimensional layer for the map. Then we flattened the output of the convolutional layer to concatenate it with the output of the dense layer and give this into

```
obs = {
            'map': maps,
            'position': self.agent_pos
        }
```

Figure 3: observation returned by step()

```
def one_hot(self, map):
        ohm = [x[:] for x in map]
        for i in range(len(map)):
            for j in range(len(map)):
                if map[i][j] == 'G':
                    ohm[i][j] = [0, 0, 1]
                if map[i][j] == 'H':
                    ohm[i][j] = [0, 1, 0]
                if map[i][j] == 'F':
                    ohm[i][j] = [1, 0, 0]
        return ohm
```

Figure 4: Function to turn map into one-hot map

a second dense layer, before it is turned into the output of the network. Since Frozen Lake is a sparse reward environment, we train with rather small values around zero. Therefore, we decided to use a sigmoid activation for all hidden layers as opposed to the usual ReLU activation, as ReLU ignores negative rewards and we wanted to avoid dead neurons. The only difference between actor and critic is the output layer, as we used a dense layer with neurons the size of the action space and a softmax activation for the actor, to get the probabilities for the actions, and a dense layer with one neuron and no activation, as we want to get a single value for the given state. For the adversary we used exactly the same architecture. For the actor network we initialized each layer with zeros, to get an equal probability for every action of exactly 0.25.

2.4 The Idea behind *PAIRED* [2]

Before we go on with the implementation of the Adversary, we will go into more detail about why we are even using the adversary and the calculation of the regret. We use *PAIRED* [2] to achieve an automated unsupervised environment design (UED). The algorithms that we have had at our disposal so far are Domain Randomization or Minimax Adversary. The idea behind Domain Randomization is very straightforward: we create an environment randomized, regardless of the agent or its difficulty. The drawbacks of this are obvious, we create uninteresting environments and if we don't check for validity, we might even create impossible environments. The Minimax Adversary approach is a bit more sophisticated in that regard: The Adversary tries to create environments that minimse the reward of the policy and does not disregard the agent completely. But this has a downside as well. The Adversary creates environments that can be too difficult for the agent to learn, up to unsolvable environments. To show how these approaches compare to PAIRED[2] we included Fig.5 with examples to these as well as an example for a transfer task. Now PAIRED takes this

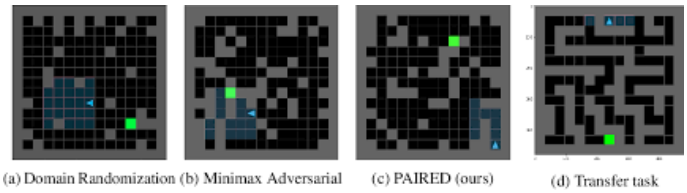

Figure 5: (a) Domain Randomization (b) Minimax Adversarial (c) PAIRED (d) Transfer task example [2]

idea further by introducing an antagonist, which is allied to the adversary. So the actual training agent, which is called the Protagonist, learns together with the Antagonist on the environments created by the Adversary, which tries to create environments in which the Antagonist performs well, but the Protagonist doesn't. The advantage over the other approaches is that the Adversary tries to avoid creating impossible environments, so that the Antagonist can still perform well, but as tough as possible, to minimise the Protagonists reward. So the Adversary creates more difficult but solvable environments while Protagonist and Antagonist continue to learn. The

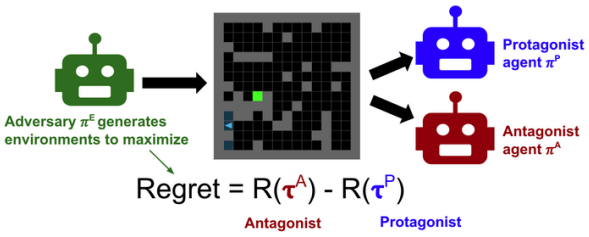

Figure 6: Adversary Example

Regret plays the key role in this type of environment design and learning. The regret is collected

by running both agents, Protagonist as well as Antagonist, through the current environment and collecting their trajectories respectively. To get a difference in the regret, we now take the average rewards of the protagonist and subtract them from the maximum rewards of the antagonist. The regret should gets larger, the better the Antagonist performs and the worse the Protagonist performs, meaning that the regret gets larger if the Protagonist needs more steps to get to the goal. So it is the Adversary's task to maximize the regret between Antagonist and Protagonist and create the most difficult environment for the Protagonist, while the Antagonist can still solve it.

2.5 Implementing *PAIRED* [2]

The next step was bringing both our adversarial environment and the actor together in the PAIRED-algorithm. This is implemented in the driver class. One adversarial epoch is in its core relatively simple. The adversary creates a map using the actor, again also evaluating the map using the critic at every step. After completing the environment, we check if goal and agent position are set and in case they are not, which is possible in the beginning for random actions, we set them in a random place. While creating the environment we collect the according variables. After the environment has been built we start training our agents, the protagonist and the antagonist. First we run the protagonist through the environment and train it for n times, then we evaluate its performance with our get performance function. In this function we run the agent through the environment and sample his actions with his actor network. We are not using the action with the maximum probability here, as we want the agent to avoid getting stuck, if a bad actions should temporarily get a higher probability. We collect the trajectories for a given number of times, and calculate the reward for every trajectory as follows:

$$r = \gamma^t * reward$$

So with an increasing number of steps, the performance worsens. For the protagonist, we now take all performances and calculate the mean. This is the first part to calculate the regret For the Antagonist we do the same. We train it n number of times and collect the performances, only this time we don't take the mean, but the maximum of the collected performances. This is the second part we need for the regret. The regret is then easily calculated by subtracting the mean from the max:

$$REGRET^{\vec{\theta}}(\pi^P, \pi^A) = U^{\vec{\theta}}(\pi^A) U^{\vec{\theta}}(\pi^A)$$

This regret then forms the loss for the adversary, which we can then use to train it accordingly.

3 Outlook

Because of the problems we had already when training, we weren't able to get any viable results, neither for the agents performance in a transfer task, nor the Adversary's performance. If the agents had worked we would have fitted the Adversary and the regret to train with the agents, but right now it's still more in a test state. We wanted to adapt the agents to also include the regret when training them, using the learning update and a negative regret for the protagonist and the respective learning update with a positive regret for the Antagonist and Adversary. Using the max and mean to calculate is supposed reduce noise in the rewards and train the adversary better, which we would have liked to have tested after a successful implementation, as well as how adding the map size as a parameter for the Adversary would have influenced its performance. To show how a functioning PAIRED implementation would have compared to the before-mentioned approaches Domain Randomization and Minimax Adversary, we included the results of the original paper in figure 7, which shows a significantly higher solvable path length in the PAIRED trained agent. We also implemented an additional version of the environment, with sparse and easy reward settings, which we would have liked to test and compare as well, but we will go into more detail about that in the appendix. We hope to be able to finish the algorithm and add these features and comparisons at a later point to properly conclude this implementation.

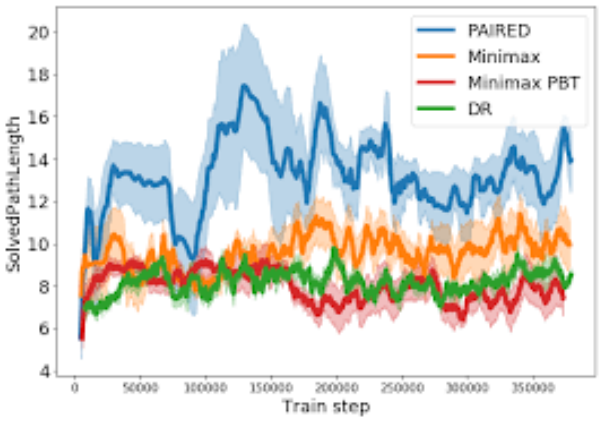

Figure 7: Solved Path Length

References

[1] N. Jaques and M. Dennis, "Paired: A new multi-agent approach for adversarial environment generation," 2021. [Online]. Available: https://ai.googleblog.com/2021/03/paired-new-multi-agent-approach-for.html

[2] M. Dennis, N. Jaques, E. Vinitsky, A. Bayen, S. Russell, A. Critch, and S. Levine, "Emergent complexity and zero-shot transfer via unsupervised environment design," 2020. [Online]. Available: https://arxiv.org/abs/2012.02096

[3] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

4 Appendix

We also developed an additional environment to test the algorithm, but it didn't work as well, as the gradients for the regret always come up empty. We will still give an overview over this part as well.

The environment we created is purely text based and has no proper visualisation yet. We also added another possible map state, a blocked cell, to the environment, which brings the possible values a cell of the map can have to four: a goal, an empty cell, an obstacle and a hole. When creating the environment the user can also choose between two reward settings, sparse and not sparse. Sparse means, the only reward the agent can get is one. In the not sparse setting, the agent gets a positive reward from the goal, but now it is one hundred. The other rewards for the different kinds of cells are negative, with -10 for a hole, as it resets the environment, -1 for an empty cell to further discourage taking more steps and -1 for a blocked cell. If this implementation had worked, we could also have compared if the rewards settings change the performance of the agents and if so how much.

```
[['x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x']
 ['x' 'P' '-' '-' '-' '-' '-' '-' '-' '-' '-' 'x']
 ['x' '-' 'x' '-' '-' '-' '-' '-' '-' '-' '-' 'x']
 ['x' '-' '-' '-' '-' 'x' '-' 'h' '-' '-' '-' 'x']
 ['x' '-' '-' 'x' '-' '-' '-' '-' '-' '-' '-' 'x']
 ['x' '-' '-' '-' '-' 'x' '-' '-' '-' 'h' '-' 'x']
 ['x' 'h' '-' '-' 'x' '-' '-' '-' 'h' '-' '-' 'x']
 ['x' '-' 'h' '-' '-' '-' '-' '-' '-' '-' '-' 'x']
 ['x' '-' 'h' '-' '-' '-' '-' '-' '-' '-' '-' 'x']
 ['x' '-' '-' '-' '-' '-' '-' '-' '-' '-' '-' 'x']
 ['x' '-' '-' '-' '-' '-' '-' '-' '-' '-' 'G' 'x']
 ['x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x']]
```

Figure 8: Character Environment Example

The network architecture looks a bit different in this implementation as well. There are two possible networks one can choose from: One including an LSTM with fixed parameters (would have to be changed in class) and one with two convolutional layers that takes the parameters as arguments for the initialization. The first one is designed after the one used in the original paper, using a convolutional layer with 16 filter and a kernel size of one first, then, after giving it through a batch normalization and activation layer, it is flattened and put through a dense layer with 5 neurons, so it fits the LSTM layer with 16 units. After that it is again given through a dense layer, this time with 32 units, and then to the output layer. The output layers stayed for all actors and critics respectively the same. The solely convolutional architecture consists of two convolutional parts, each of these parts containing one convolutional layer,

one batch normalization layer and one activation layer. The output is then given through a dense layer before it is flattened and given to the output layer. All networks use leaky relu as their activation.

The training class for the PPO Agent and the PAIRED Algorithm stayed pretty much the same as in the above mentioned implementation, except for some minor changes due to a different structure and environment. It would have been really interesting to compare these environments, as well as the different architectures and reward settings and how each of these differences would impact performance. The link to the additional environment and training is here: `https://github.com/H-27/env_FrozenLake-PPO.git`