RLD

Reinforcement Learning

Alexander Baumann
Yunfei ZHAO

March 1, 2022

In this report we aim at demonstrating some basic elements and algorithms of reinforcement learning, with some implementations and demonstrations of questions from TMEs.

All **source code** of this report can be found on github by the following link: https://github.com/alexanderbaumann99/Reinforcement-Learning, The authors of source code are Alexander Baumann and Yunfei ZHAO. The project is based on given code [1] from M.Sylvain Lamprier.

Contents

List of Figures

Chapter 1

TME 9: GAN

1.1 Algorithm

A gernerative adversarial network (GAN) is a generative model consisting of a generator G and a discriminator D (see fig. 1.1). The generator takes samples from the latent distribution as an input and generates fake data. The goal of the discriminator is to distinguish the real data and the fake data from the generator. Overall, one obtains the following loss to optimize:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log(D(x))] + \mathbb{E}_{z\sim p_Z(z)}[\log(1-D(G(z)))]$$

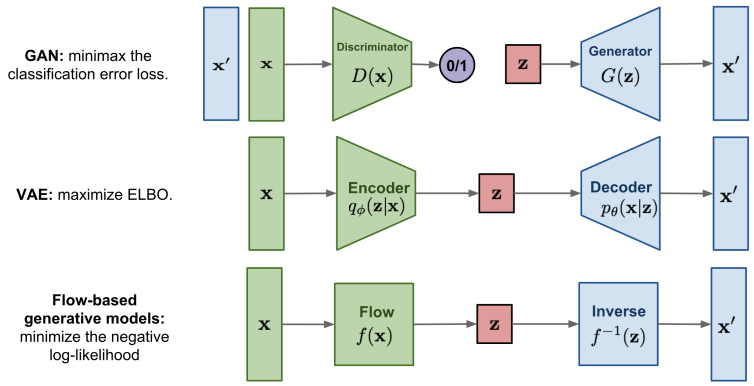


Figure 1.1: Comparison of three categories of generative models [5]

1.2 Experiments

The ground truth data set is the celebA data set which consists of over 200.000 images of celebrities. Sample images can be seen in fig. 1.2.

The architecture of both networks is inspired from the DCGAN [4]. The generator and discriminator are deep convolutional neural networks. The former consists of transposed convolutions in order to upsample the latent space sample, the latter of normal convolutions to downsample. The dimension of the latent space is 100 and the latent distribution is $\mathcal{N}(0,1)$. We optimized the GAN through the Adam optimizer using a learning rate of 0.0002. Since the training of a GAN takes a long time, we did this on Kaggle with the help of a GPU. Overall, we trained 25 epochs. In fig. 1.3, we see the distinction by the discriminator between the real and the

Figure 1.2: Sample images from the celebA data set

generated images. From there, we can deduce that the generator improves over time since the discriminator classifies more generated images as real. This fact can also be verified by having a look at the generated images from a fixed noise in fig. 1.4.

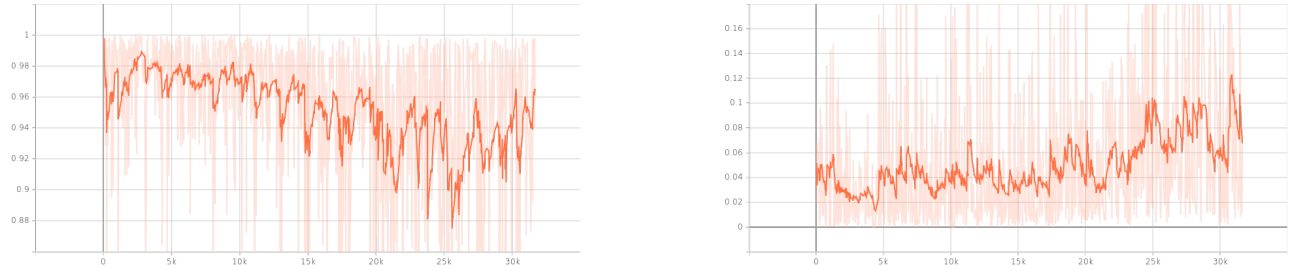


Figure 1.3: Prediction of the real data (i.e. $D(x)$) and generated data (i.e. $D(G(z))$) of the discriminator, trained on celebA

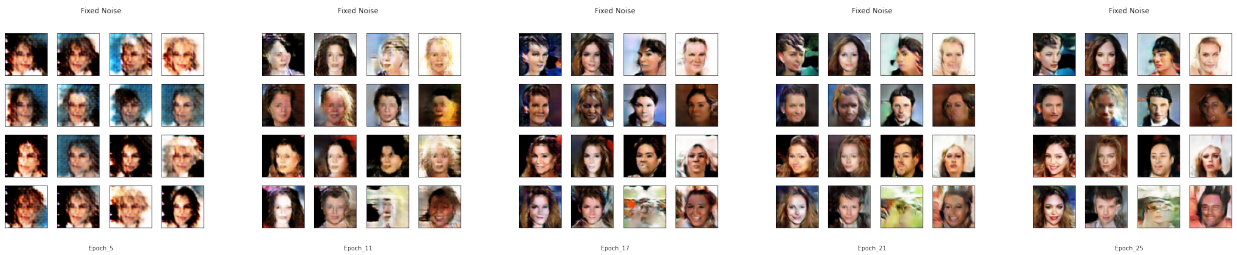


Figure 1.4: Generated images from a fixed noise, trained on celebA

We also trained the GAN network on the MNIST data set which consists of handwritten numbers from 0 to 9. The generator and discriminator were again built of transposed and normal convolutions. Since the ground truth images have size $1 \times 28 \times 28$, we had to modify the kernel size.
The data set includes only 60.000 training images and is therefore significantly smaller than the celebA data set. Additionally, the images are not RGB images. That is why training is much faster with this data set.
In fig. 1.5, one can see that the generator performs much better and even in a shorter amount of time, comparing to the training on the celebA data set. Indeed, the generated images from epoch 6 are already quite good (see fig. 1.6).

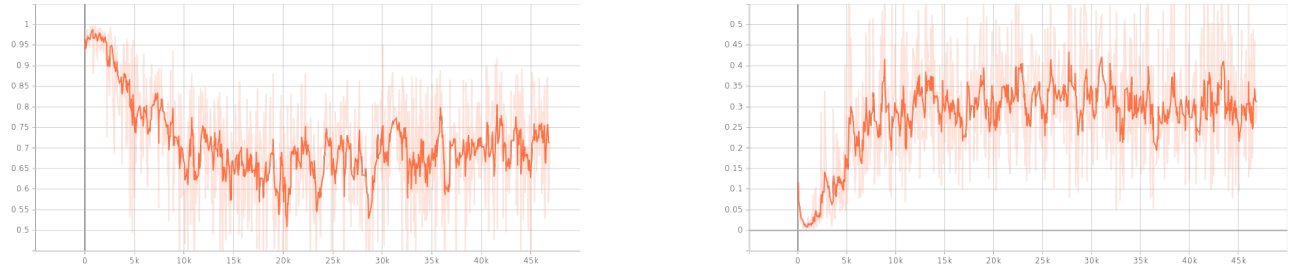


Figure 1.5: Prediction of the real data (i.e. $D(x)$) and generated data (i.e. $D(G(z))$) of the discriminator, trained on MNIST

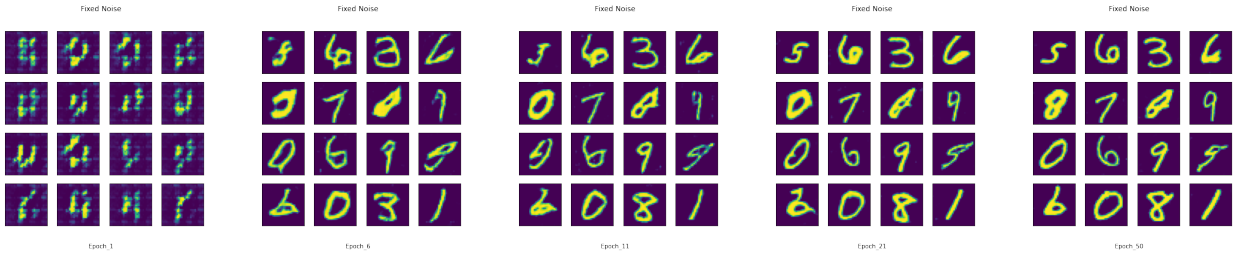


Figure 1.6: Generated images from a fixed noise, trained on MNIST

Chapter 2

TME 10: VAE

2.1 Algorithm

The second type of generative models, which we discuss here, is the variational auto-encoder (VAE). The idea is to encode the observations to a latent space. There, one samples some latent variables which are then decoded to the observation space. In this way, one generates data from the latent space (see fig. 1.1).
Let $p(X)$ be the distribution of the observation space and $q(Z)$ the distribution of latent space. Then one can deduce the following equality:

$$\log p(X) = D_{KL}\left(q(Z)\|p(Z|X)\right) + V_L(q) \tag{2.1}$$

with

$$V_L(q) = \mathbb{E}_{q(Z)}\left[\log\left(\frac{p(X,Z)}{q(Z)}\right)\right]$$
$$D_{KL}\left(q(Z)\|p(Z|X)\right) = -\mathbb{E}_{q(Z)}\left[\log\left(\frac{p(Z|X)}{q(Z)}\right)\right]$$

Since the Kullback-Leibler divergence is always positive, we obtain $V_L(q)$ as a lower bound of $\log p(X)$. So, the objective is to maximize $V_L(q)$ in order to maximize p(X).
We model the distribution $p(x|z)$ with a neural network with weights θ and the latent distribution $q(z)$ with a distribution of the form $q_\phi(z|x)$. From eq. (2.1), we then obtain:

$$\log p_\theta(x) = D_{KL}\left(q_\phi(z|x)\|p_\theta(z|x)\right) + V_L(\phi;\theta;x)$$

Then one can calculate:

$$V_L(\phi;\theta;x) = \mathbb{E}_{q_\phi(z|x)}\left[\log\left(\frac{p_\theta(x,z)}{q_\phi(z|x)}\right)\right] \qquad = -D_{KL}\left(q_\phi(z|x)\|p(x)\right) + \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$$

If we choose the distributions in a simple way, then one can deduce an explicit expression for the Kullback-Leibler divergence. Here, the following assumption are made:

$$p(x) = \mathcal{N}(0, I)$$
$$q_\phi(z|x) = \mathcal{N}\left(\mu_\theta(x), \sigma_\theta(x)\right), \qquad z \in \mathbb{R}^J$$

With this choice, we get:

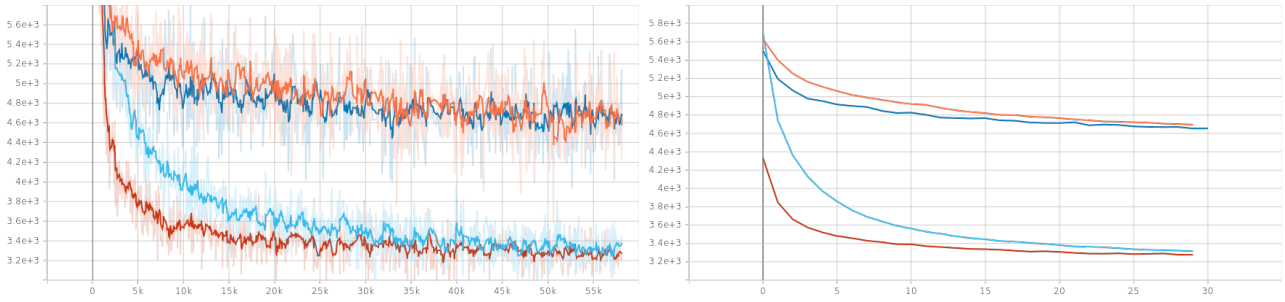
$$-D_{KL}\left(q_\phi(z|x)\|p(x)\right) = \frac{1}{2}\cdot\sum_{j=1}^{J}\left(1 + \log\left(\sigma(x)_j^2\right) - \mu(x)_j^2 - \sigma(x)_j^2\right)$$

Now, we still need to choose $p_\theta(x|z)$. Since the images from the data set will be in black and white, its values are between 0 and 1. Hence, we can model each pixel with a Bernoulli distribution $Bernoulli(\lambda_\theta(x))$ where $\lambda_\theta(x)$ is the output pixel of the decoder. That is why it is necessary that the final activation of the decoder is the sigmoid function. With this distribution, we obtain minus the cross-entropy loss as the second loss term in $V_L(\phi;\theta;x)$. Overall, we have the following loss which is to minimize:

$$\mathcal{L} = -V_L(\phi;\theta;x) = -\frac{1}{2}\cdot\sum_{j=1}^{J}\left(1+\log\left(\sigma(x)_j^2\right)-\mu(x)_j^2-\sigma(x)_j^2\right)+\text{Cross-Entropy}\left(\text{Decoder}(z),x\right)$$

2.2 Experiments

As the ground-truth, the MNIST data is used which consists of 60.000 images of handwritten numbers from 0 to 9. For the modelling it is essential that the pixel values are between 0 and 1.
As encoder-decoder architecture, we compare two different approaches. The first one is to use fully connected networks (FCN). We implemented networks with two hidden layers of dimension 256 and ReLU as activation. The second approach is to use convolutional neural networks (CNN). The encoder uses normal convolutions, the decoder transposed convolutions. The output of the convolutions in the encoder are put into a small fully connected network to give us the mean and variance for the latent distribution. As mentioned before, the decoder in both cases must have a sigmoid activation in the end. We chose the number of parameters in both networks in such a way they are similar. So, we can properly compare the performances.We also experimented with the latent dimension dim(Z) and compared both approaches with a latent dimension of 2 and 20.
The model performances can be seen in fig. 2.1. One can deduce that the VAEs with the higher latent dimension are significantly superior to the ones with dim(Z)=2. Furthermore, it is noticeable that the CNN auto-encoders have especially in the beginning of training and testing a better performance than the fully connected encoders.
We compared reconstructed test images of the best (CNN with dim(Z)=20) and the worst model (FCN with dim(Z)=2) after epoch 30 (see fig. 2.2). Since fig. 2.1 indicates that the loss of the CNN model is significantly lower than the loss of the FCN model in the beginning of the training phase, we visualize the reconstructions of the test after epoch 0 in fig. 2.3 . There, one can indeed verify this fact.
However, these images are just reconstructions and not new generated images. To do that, we sample latent space vectors from $\mathcal{N}(0,1)$ and decode them. The results are shown in fig. 2.4. By looking at the distribution of μ in fig. 2.5, we can conclude that it does make sense to take $\mathcal{N}(0,1)$-samples since then we obtain vectors whose entries are near of $\mu(x)$ for some x in the train set. Overall, one can also deduce by looking at the images that the CNN architecture with dim(Z)=20 is the best model among them. Furthermore, it is noticeable that the reconstructions and the generated images with latent dimension of 20 are clearer and less blurry than those with dim(Z)=2.
Finally, in fig. 2.5 one can see the distribution of $\mu(x)$, where x runs through all images in the test set. The colour of each scatter point indicates its ground-truth value (i.e. the number between 0 and 9). In this way, one can see which numbers the model can well distinguish and which numbers not so well. One can conclude that 0,1,5 and 6 are well recognized by the encoder. However, it is hard to differentiate between 7 and 9 for example.



(a) Train loss (b) Test loss

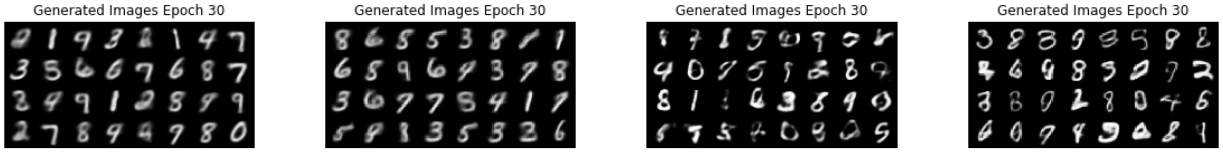
Figure 2.1: Orange: FCN, dim(Z)=2; Dark Blue: CNN, dim(Z)=2;
Light Blue: FCN,dim(Z)=20; Red: CNN, dim(Z)=20

(a) FCN, dim(Z)=2 (b) CNN, dim(Z)=20

Figure 2.2: Comparison of reconstructions of the test set by the worst and best model

(a) FCN, dim(Z)=20 (b) CNN, dim(Z)=20

Figure 2.3: Comparison of reconstructions of the test set after epoch 0, CNN vs. FCN



(a) FCN, dim(Z)=2 (b) CNN, dim(Z)=2 (c) FCN, dim(Z)=20 (d) CNN, dim(Z)=20

Figure 2.4: Generated images by the VAE from $\mathcal{N}(0, 1)$-samples

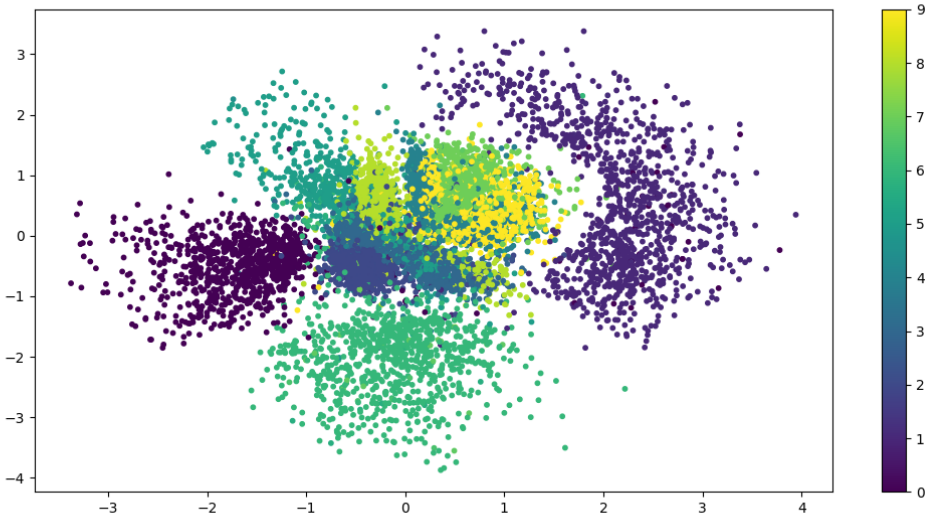


Figure 2.5: Distribution of $\mu(x)$ for all x in the test set for the CNN VAE with dim(Z)=2

Chapter 3

TME 11: Multi-Agents RL

In this chapter, we explorer multi-agents tasks. We will explorer three kind of problems They are respectively:

- Cooperative navigation: Three agents (in blue) try to reach three targets without collision and they will try to reach to three target by collaboration and they have the same global rewards.
- Physical deception: Two agent (in blue) and one adversary (in red). There are two landmarks and only one of it is target and the adversary desires to reach the target landmark but it does not know which of the landmarks is the correct one. The agent will try to spread to two bases in the same time to deceit the adversary. Two agents have the reward with regard to the distance of the closest agent to the target and if the adversary is close to the target, agents will have a penalisation. As for the adversary, it will get reward if it is close to the target.
- Predator-prey: We have 3 predators (in red) and 1 prey (in green). the prey move faster than three predators and the prey try to avoid predators and the predators collaborate to catch the prey.

3.1 Algorithm

We use an algorithm called MADDPG (Multi-agent Deep Deterministic Policy Gradient) proposed in **Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments**[3]. This method is based on DDPG. The main difference is showed below.

3.1.1 Hypothesis of MADDPG

In MADDPG we suppose that if we know the actions taken by all agents, the problem is stationary even thought policies of agents evaluate. We also suppose that each agent is able to observe actions taken by other agents for each step, but it has only its own observation. We follow the schema given by figure 3.1. We have:

$$P(s'|s, a_1, ..., a_N, \pi_1, ..., \pi_N) = P(s'|s, a_1, ..., a_N) = P(s'|s, a_1, ..., a_N, \pi'_1, ..., \pi'_N) \tag{3.1}$$

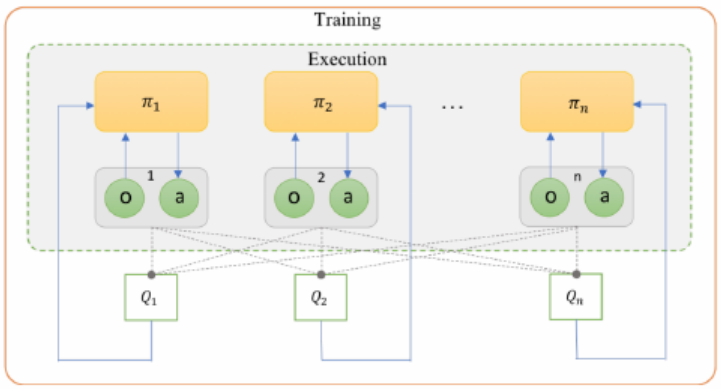


Figure 3.1: Schema of multi-agent DDPG.

3.1.2 Critic Networks

Each agent possesses their own critic network Q_i which is updated based on the reward of a transition and observations and actions of all agents. While, for classic DDPG for Q network of each agent, it takes only the observation and action of this agent into consideration. So we update critic network by minimising the loss function:

$$L(\theta_i) = \frac{1}{S}\sum_j (y^j - Q_i^\mu(x^j, a_1^j, ..., a_N^j))^2 \tag{3.2}$$

$$y^j = r_i^j + \gamma Q_i^{\mu'}(x'^j, a'_1, ..., a'_N)|_{a'_k=\mu'_k(o'^j_k)} \tag{3.3}$$

- **i** is the index of corresponding agent.
- **j** is the index of transition sampled from replay buffer.
- **N** is number of agents.
- **S** is number of transitions
- μ represent policies of agents and μ' are target policies and we use target policy to generate corresponding actions to update critic network . In this way, we can stabilise the learning process of Q network. And we do not take directly the destination action like what we do in SARSA, in this way, we take into account of the evaluation of other agents.

3.1.3 Actor Networks

For actor networks, the update process is exactly the same as what we use for DDPG where we use deterministic policy gradient theory. Further more, the policy of an agent we use here will take only into account the observation from this agent so that this policy can be used independently in an environment. The policy of an agent is updated by sampled policy gradient:

$$\nabla_{\theta_i} J(\mu_i) \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(o_i^j)\nabla_{a_i} Q_i^\mu(x^j, a_1^j, ..., a_i^j, ..., a_N^j)|_{a_i=\mu_i(o_i^j)} \tag{3.4}$$

There are also two versions of actor network, the first version is described as above. To avoids over-learning of policy of other agents concurrent, we attribute each agent K policies and each time we sample an agent uniformly to take an action and update this policy.

3.2 Experiments

- For critic network of each agent, we used a fully connected network with one hidden layers of size 128 with activation function LeakyRelu and final activation function of Tanh.
- For actor network of each agent, we use the same neural network structure as critic network.
- The optimization is made by the Adam optimizer, using a learning rate of $1 \cdot 10^{-3}$ for polities and $1 \cdot 10^{-2}$ for critics.
- The weights of the target networks are updated for each learning step, with soft update of 0.9 for old target parameters.
- The networks is optimized every 10 events and for the first 130 events, agents react randomly.
- We use Ornstein–Uhlenbeck process to add noise with sigma equal to 0.2 for training and 0 for testing (no noise). Furthermore we have a memory of size $1 \cdot 10^3$ for each policies and it is used from which mini-batches of size 128 are sampled in the learning step.
- The maximum length of an episode is 25 events for training and 100 for testing.
- 1 testing episodes are run for every 100 episodes.
- The reward signal is clipped in interval [-100, 100].

3.2.1 Cooperative navigation

For cooperative navigation we used environment defined in **simple_spread.py**. The networks are run for 100k events with MADDPG with 5 policies for each agent. Some results are shown in figure 3.2. Most of time, agents are able to get to the landmarks, but when landmarks are

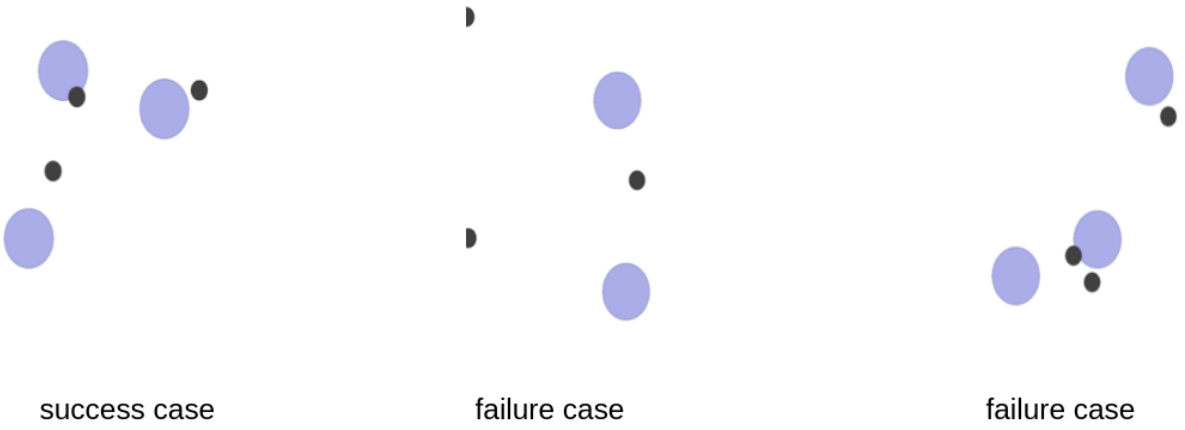


Figure 3.2: Success case and failure cases for simple spread environment.

located to the border of map or when they are too close to each other, agent are fail to match their position, we think that for the border case, there are few examples of this kind of examples

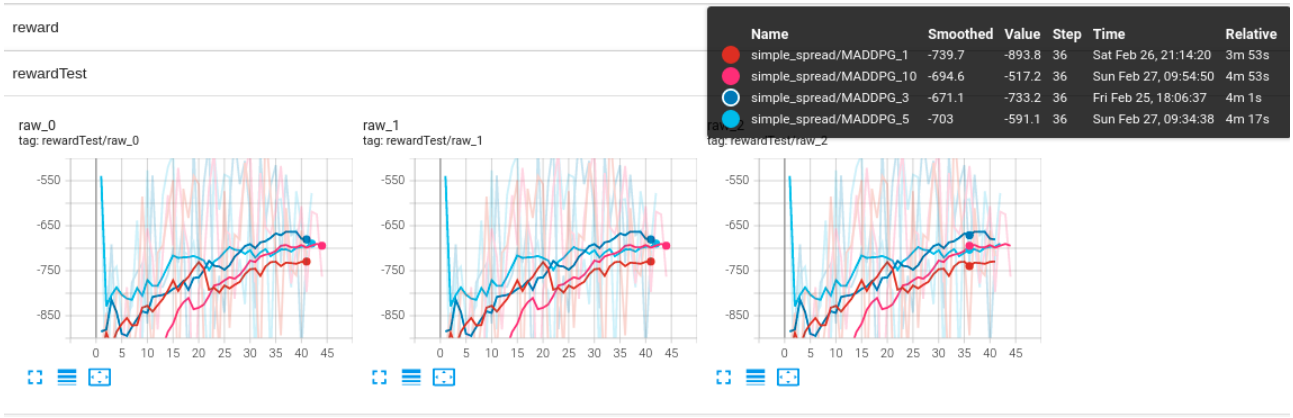


Figure 3.3: Testing curves of MADDPG with different number of agents for simple spread environment.

during training and for landmarks that are close to each other, it is hard for agent to chose a landmark to go.

In figure 3.3, we notice that, MADDPG with multi-agent policies models perform better than MADDPG with only one policy and model with three agent policies have better convergence speed.

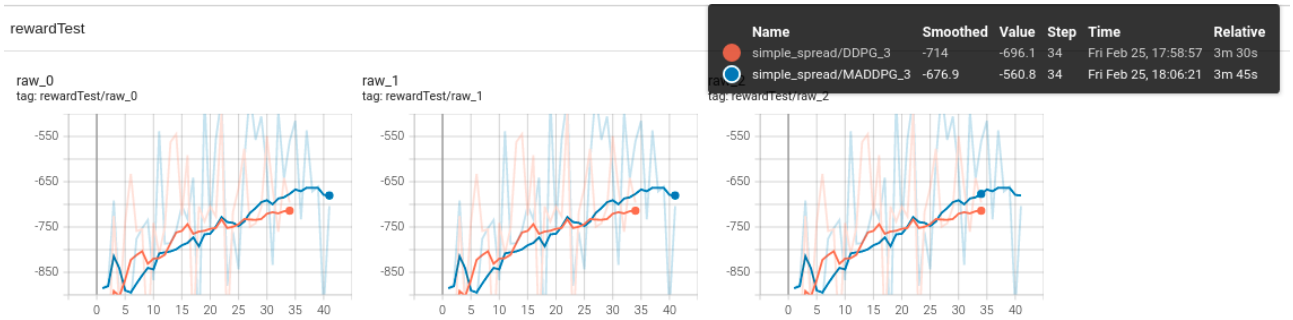


Figure 3.4: Testing curves of MADDPG and classic DDPG

We compare than MADDPG with classical DDPG. In DDPG, each agent take only their own observation and their own action into consideration for critic network. In figure 3.4, We found that MADDPG performs slightly better than DDPG that is quite shocking. Because even thought agents do not consider other agents performance and their states, it still achieve a performance not far below MADDPG. To be noted that our MADDPG does not perform as well as Dr. Lamprier's result as his test reward higher than -500.

3.2.2 Physical deception

Physical deception is a problem with adversary agent that makes the problem more complex. WeFigure 3.5 shows the evaluation of sampled examples over time. At the beginning both agents and adversary do not know what to do. Then agents learn to deceit adversary, and agent start to get better reward, while after a moment, agents try to get as far as possible from landmarks to avoid the penalty when adversary come close to the target. Figure 3.6 shows the learning curve.

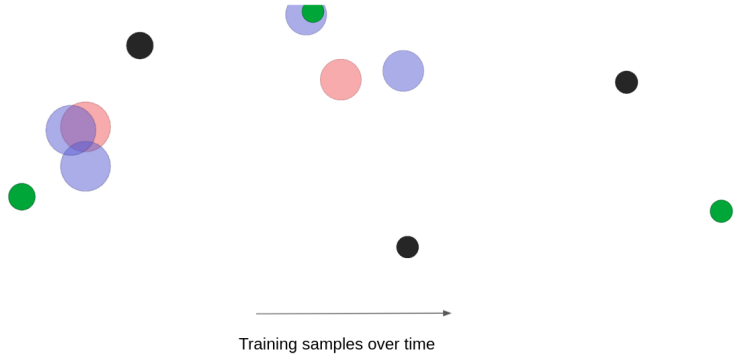


Figure 3.5: Examples of agents behaviors in simple adversary map over time.

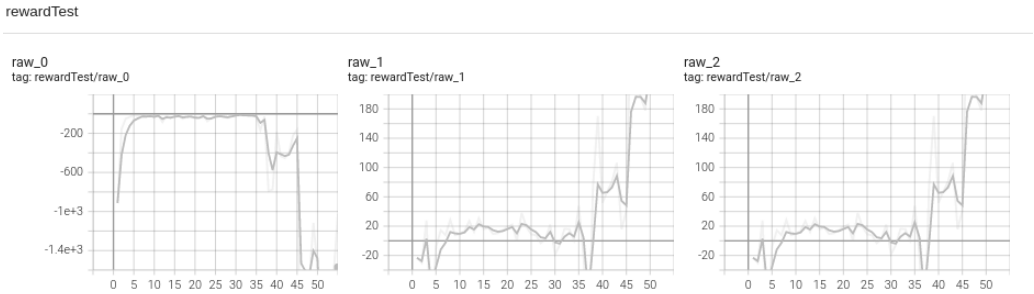


Figure 3.6: testing reward curve for simple adversary map over time.

3.2.3 Predator-prey

We used MADDPG with three policies. In most of time, predator is not able to catch the prey, but we list some success case here.

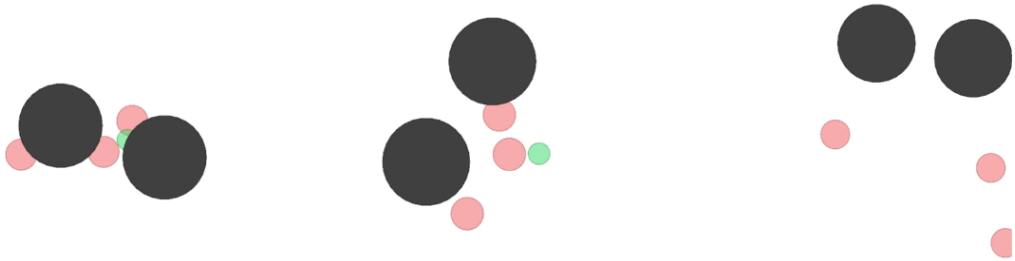


Figure 3.7: Examples of agents behaviors in predator-prey environment.

Figure 3.8: testing reward curve predator-prey environment.

Chapter 4

TME12: Imitation Learning

The learning process for reinforcement learning models is often difficult to converge. Making a compromise between exploration and exploitation is also a complex point to train an agent. Moreover, in the more difficult case, the reward function and the interaction with the environment are not available. On the other hand, in many problems, one can use an expert, and the demonstrations given by this one can accelerate and improve the learning process: this is the main idea of imitation learning.

There are generally three kinds of imitation learning:

- Behavioral Cloning
- Interactive Policy Learning
- Apprenticeship Learning

In this chapter, we are going to explorer **Behavior Cloning** and **GAIL**. GAIL is an algorithm of inverse reinforcement learning that is a kind of apprenticeship learning following the idea of GAN by using discriminant network to provide reward signal.

The environment we use for experiment is LunarLander(discret action space) and the demonstrations are provided by an PPO agent that is trained to achieve accumulated reward of 230.

4.1 Algorithm

4.1.1 Behavior Cloning

For Behavior Cloning, our agent is a sample neural network. We try this neural network by supervised learning on demonstration dataset.

$$\max_{\theta} \sum_{(s_i,a_i)\in\mathcal{E}} log\pi_\theta(a_i|s_i) \tag{4.1}$$

Where $\mathcal{E}$ represent demonstration dataset, and π_θ is the policy network of our agent.

4.1.2 GAIL

GAIL inherit the esprit of GAN, in this way, agent is able to learn from expert's demonstrations even in continue space by a discriminator which we used to distinguish a transition from agent or expert.

Discriminator

The discriminator try to distinguish tuples of (state, action) by attributing 1 to tuples from expert and 0 to tuples from agent. So we update discriminator by minimising the following equation:

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_\omega log(D_\omega(s,a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_\omega log(1 - D_\omega(s,a))] \tag{4.2}$$

Where τ_i is agent transition and τ_E is expert transitions. In a nutshell we use the discriminator to provide a reward to policy, the more agent's behavior is indistinguishable from expert behavior, the higher reward it will get. In this way the agent will not just copy exactly tuples of state and action from expert like in behavior cloning, it will try to imitate expert in a more homogeneously.

Agent

For reinforcement learning agent, we use PPO with **clipped objective** and **advantage estimation**. The policy update follows these equations with K steps:

$$\theta_{i+1} = \underset{\theta}{argmax} L^{CLIP}_{\theta_k}(\theta) - \lambda \nabla_\theta H(\pi_\theta) \tag{4.3}$$

$$L^{CLIP}_{\theta_k}(\theta) = \underset{\tau \sim \pi_k}{E} \left[\sum_{t=0}^{T} \left[min(r_t(\theta)\hat{A}^{\pi_k}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}^{\pi_k}_t \right] \right] \tag{4.4}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \tag{4.5}$$

Where $H(\pi_\theta)$ is entropy on policy we try to maximise during training to have more exploration. $r_t(\theta)$ is used to avoid policy changing too fast and explore in a save zone. And we use

4.2 Experiments

4.2.1 Behavior Cloning

For Behavior Cloning, we use advantage value $\hat{A}^{\pi_k}_t$ to update our policy.

- **Policy network**: two hidden layers respectively 64 and 32 neurons with Tanh activation function and a Softmax function before the output to generate the probabilities corresponding to each action.
- **Policy network update**: We use whole demonstrations to update the policy network for each 100 events and the max events length for each episode is 200 events and we show the accumulated reward of each episode.

We observe that agent's policy network can learn the demonstration as the loss continue to reduce while the reward increases before first 400 episode and then it become high variance. We think that may due to the surprised learning over-fit demonstrations and when agent is given a state that is more or less far from states in demonstration, the policy can not give an appropriate action. This problem may also due to the fact that we do not have enough samples of demonstrations from expert.

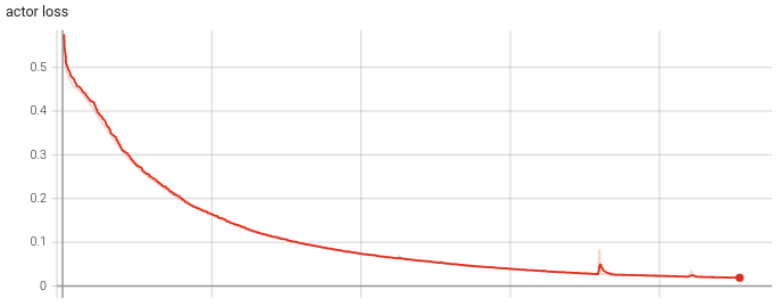


Figure 4.1: Loss of policy network over times of optimisation.

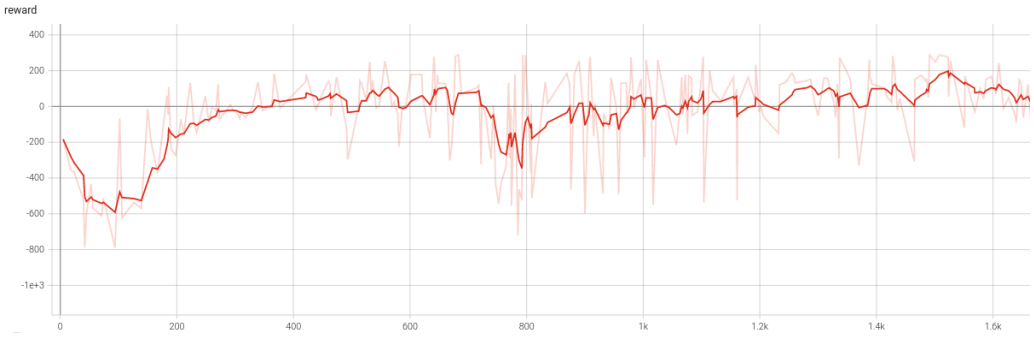


Figure 4.2: accumulated reward of agent over number of episodes.

4.2.2 GAIL

- For the critic network, we used a neural network consisting of two hidden layers of dimension 64 with an tanh activation which was optimised by the Adam optimizer with a learning rate of $1 \cdot 10^{-4}$. The Huber Loss is used as loss for the critic network in order to prevent exploding gradients.

- Actor network is consisted of three hidden layers of dimension 64 with an tanh activation which was optimised by the Adam optimizer with a learning rate of $1 \cdot 10^{-4}$. The Huber Loss is used as loss for the critic network in order to prevent exploding gradients. Because after some experiments, we think that to generate a policy is more complicated and it needs more parameters.

- Discriminator is composed of only one hidden layer, with a final Sigmoid function to generate a value of [0, 1] and its learning rate is $2 \cdot 10^{-4}$.

- We used the following values as the discount rate and as the parameter for the advantage estimation :
  $\gamma = 0.999$, $\lambda = 0.99$

- As suggested in the paper which introduces the PPO, we used the following PPO-related hyper-parameters:
  $\epsilon_{clip} = 0.2$

- The entropy coefficient is set to 0.001.

- The agent was trained for 15 times every 500 steps and the discriminator is trained for 5 times . The maximal length of an episode is for training is 500 steps and 200 steps for testing.

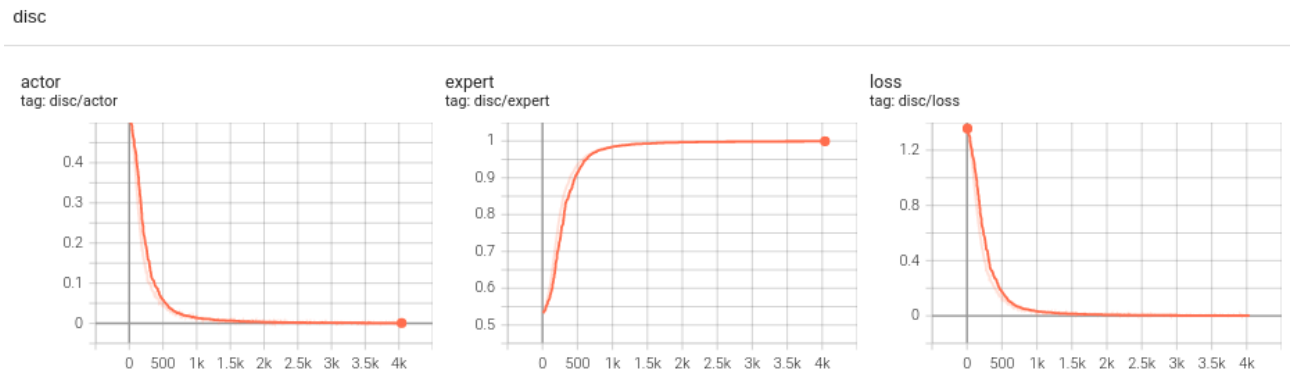


Figure 4.3: Discriminator output for agent and expert and its Loss.

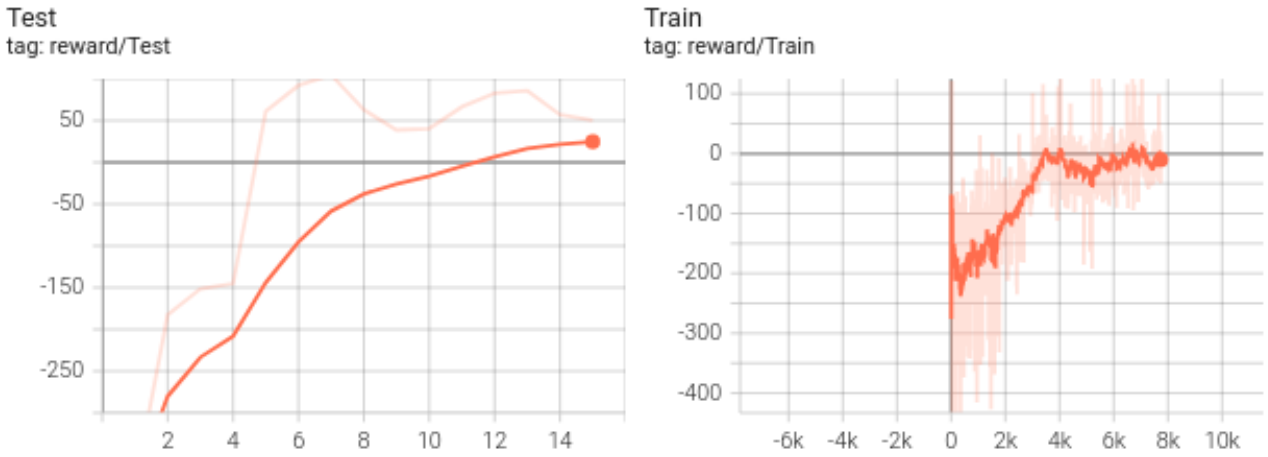


Figure 4.4: Accumulated reward of GAIL agent over number of episodes.

We have done quite a lot experiments with different hyper-parameters and figure 4.3 and figure 4.4 shows of one of our best results. We are quite surprised that we can not achieve the same level result given by Dr.Lamprier and it is somehow even worse than what we have is BC.

We tried to use directly what we implemented with a external reward from the environment, our PPO agent can achieve good result but with reward provided by GAIL discriminator, it is not good enough.

We observed that the discriminator can distinguish tuples of action and state from expert and agent easily, the value tend to 1 and o respectively, but our agent fail to do the right thing. We need to do some further analysis to deal with this problem.

Chapter 5

TME 13: Automatic Curriculum RL

5.1 Algorithm

Sometimes the agent faces a problem which is too complex for him at first glance. In these cases, one helps the agent by giving him a goal in the beginning of each episode. Then his actions depend additionally on the given goal and not just on the current state.
In particular, we have a policy and a universal value function of the form:

$$\begin{aligned} \pi &: S \times A \times G \to [0,1] \\ Q &: S \times A \times G \to \mathbb{R} \end{aligned} \tag{5.1}$$

where G is the set of goals.
A goal will be just a state which has to be reached by the agent. Furthermore, the reward in the learning process is modified to a sparse reward function. That means that we use the following reward in an episode with goal g:

$$r(s,a,s') = \begin{cases} 1, & s' = g \\ -0.1, & s' \neq g \end{cases} \tag{5.2}$$

5.1.1 DQN with goals

First, we consider the deep Q-learning algorithm with goals. For this algorithm, it is necessary that an expert has already defined a set of goals G. In the beginning of each episode, a goal is sampled from this set G. Afterwards, the usual DQN algorithm is run. However, one uses a policy, a universal value function and a reward as in eq. (5.1) and eq. (5.2). In the learning process, the old and new states are concatenated with the goal of the corresponding episode before applying the usual training algorithm.
Obviously, a downside of this algorithm is that one needs a rich set of goals G given by an expert. Otherwise, the algorithm is not very effective. This problem is tackled in the next section.

5.1.2 HER: Hindsight Experience Replay

Here, the set G can consist of only one goal which is far away from the initial state. So the necessary expert knowledge is not much. If one then applies the DQN algorithm with goals, the agent will likely not reach the goal.
Now, the idea of HER is to generate some artificial goals in the end of each episode. In our

case, we define the last state of the episode as another goal $g' = s_T$ and restore all transitions and the resulting rewards of this episode with goal g' in the memory. Thus, each episode get stored twice in the memory, once with the initial goal $g \in G$ and once with the last state s_T as goal of the episode. The rest of the algorithm is the same as the DQN with goals.
In this way, one approaches more and more the desired state g even though the set of goals consists of only one single state.

5.1.3 Iterative Goal Sampling

The reinforcement learning algorithm is again based on the deep Q-learning with goals, as in section 5.1.1. Furthermore, we assume the same setting as in the HER algorithm 5.1.2, i.e. only one goal or few goals are given by the environment. However, in complex environments it is not enough to be conditioned on a goal which is very far away. So the idea is to iteratively generate fictional goals which are nearer to the agent. These are sampled from a goal memory $\mathcal{G}$ which consists of final states of previous episodes. To reduce the risk of forgetting the final goal of the environment, the agent is conditioned on a real goal by a probability of $1 - \beta$ and on a fictional goal by a probability of β.
Another modification is to iteratively sample the goals in one episode, as the name of the algorithm suggests. That means that one does not finish the episode if a goal is reached. Instead, one samples a new goal and continues the trajectory until the maximum number of steps is reached.
The crucial aspect of this algorithm is of course how to create and sample these generated goals. The goal memory $\mathcal{G}$ is a list of limited size where the goals are added via the FIFO update rule. A final state of a trajectory will be added every n episodes if it does not exist in the goal buffer so far. Given such a memory, one samples goals from a certain distribution.
Let $g_i \in \mathcal{G}$ be a fictional goal, n_i be the number of times this goal was used as a goal for the agent and v_i be the number this goal was reached. Then one can define the random variable X_i which indicates if the agent achieves the goal g_i when conditioned on g_i. This random variable has the following entropy H_i :

$$H_i = -\mathbb{E}_{\pi(\cdot|p_i)}\left[\log(X_i)\right] = -\frac{v_i}{n_i} \cdot \log\left(\frac{v_i}{n_i}\right) - (1 - \frac{v_i}{n_i}) \cdot \log\left(1 - \frac{v_i}{n_i}\right)$$

A fictional goal is sampled from the goal memory with the following distribution p:

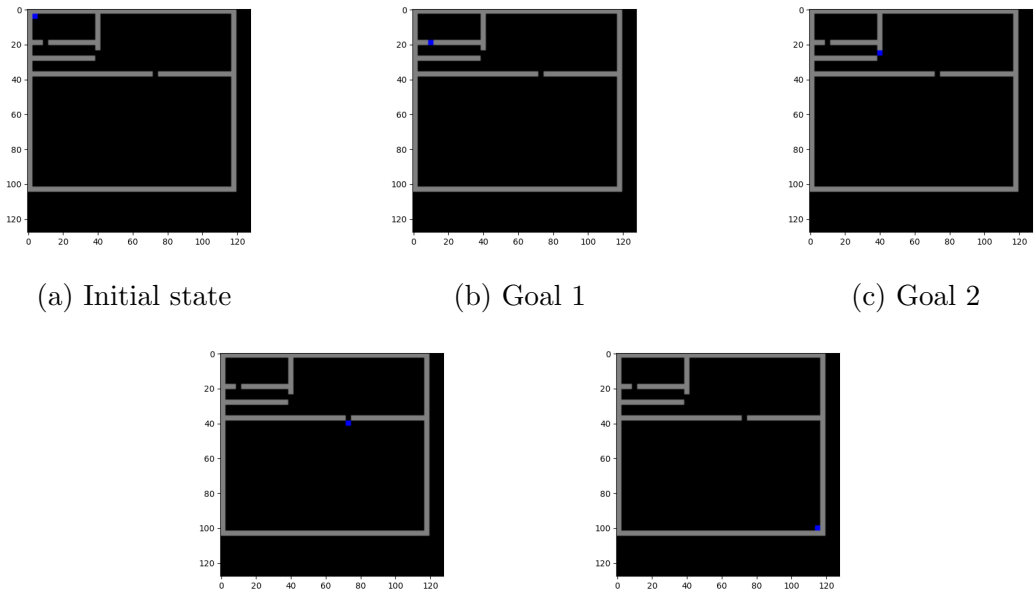
$$p(g_i) = \frac{\exp(\alpha \cdot H_i)}{Z}, \qquad Z = \sum_{g_j \in \mathcal{G}} \exp(\alpha \cdot H_j)$$

Here, α denotes a hyper-parameter which controls the influence of the entropy. So, for $\alpha \to \infty$ the distribution tends to the Dirac distribution centered at $\arg\min H_j$ and for $\alpha \to 0$ the distribution tends to the uniform distribution over the goal memory $\mathcal{G}$. This way of sampling ensures that a fictional goal with a high level of information serves more likely as goal for the agent.
When a goal g is reached, the transitions of the whole trajectory up to this state is stored in the replay buffer with this goal g. Inspired from the HER algorithm section 5.1.2, one can also store the trajectory multiple times with different goals. Afterwards, one samples a new goal in the described way and continues the trajectory. After every k episodes, one runs multiple learning steps of the DQN in order to update the Q-network.

5.2 Experiments

5.2.1 Gridworld Plan 2

We consider the gridworld environment with the corresponding goal which can be seen in fig. 5.1. So the goals 1-3 help the agent finding a way through the labyrinth and reaching the final goal. When running the algorithm from section 5.1.1, all goals are given. However, for the HER algorithm, only the final goal is given.
For both experiments, we used the following hyper-parameters:



(a) Initial state (b) Goal 1 (c) Goal 2

(d) Goal 3 (e) Final Goal

Figure 5.1: Gridworld plan 2 and the given goals

- As the universal value network, we used a fully connected network with two hidden layers of size 200 and activation function tanh.
- The optimization is made by the Adam optimizer, using a learning rate of $1 \cdot 10^{-3}$.
- The weights of the target network are updated every 1000 iterations.
- The Q-network is optimized every 10 iterations.
- The exploration rate of the ϵ-greedy strategy is equal to 0.2 during the whole experiment, hence there is no decay. Furthermore, the discount rate is 0.99. A memory of size $1 \cdot 10^6$ is used from which mini-batches of size 1000 are sampled in the learning step.
- The maximum length of an episode is 100 steps.
- Every 1000 episodes, 100 testing episodes are run.

In fig. 5.2, the test performance of both algorithms is visualized. Note however that the given set of goals are different. So, the HER algorithm works quite good even though there is almost no additional information (goals) given. This can also be verified in fig. 5.3, where the x-y-coordinates of the final state in each episode is plotted. The target state has coordinates (33,38). Thus, the algorithm learns to approach to the final goal.

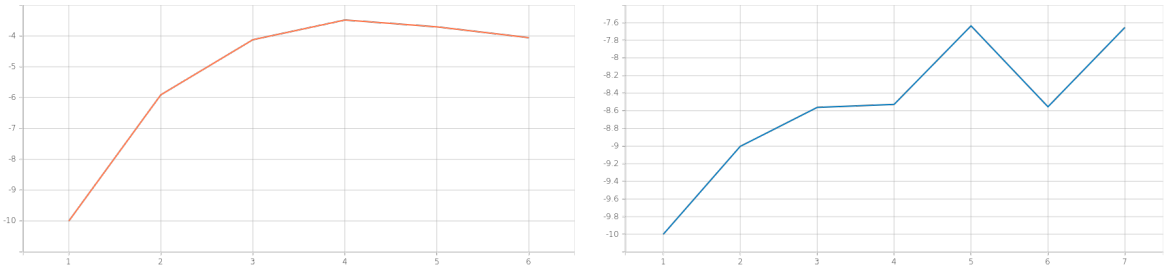


Figure 5.2: Mean test rewards of 100 episodes with DQN with goals (left) and HER (right) on gridworld plan2Multi and plan2

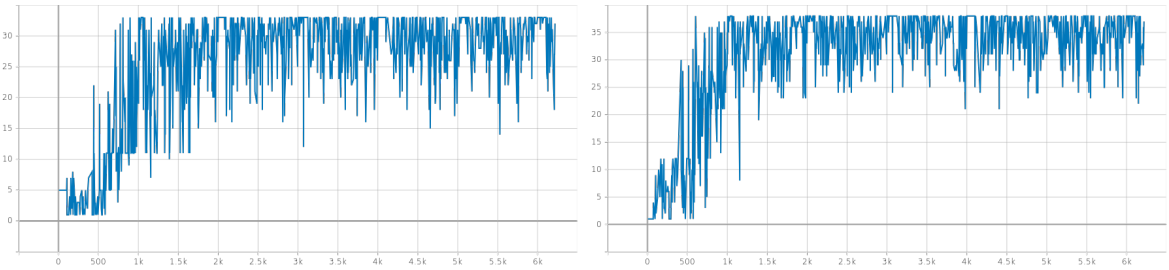
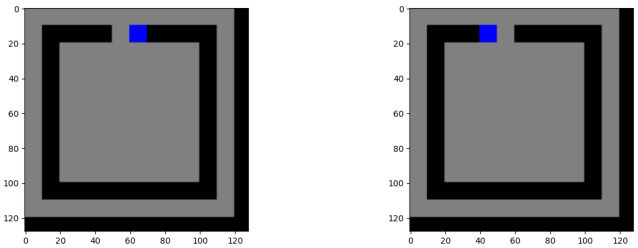


Figure 5.3: x-y-coordinates of the final state in each episode (left x, right y) when running the HER

5.2.2 Gridworld Plan 3

To see the superior performance of iterative goal sampling, we consider the following gridworld environment: When we run the HER algorithm on this plan, the agent stays at the initial



(a) Initial state of plan 3 (b) Final goal of plan 3

state during testing, since he thinks that the shortest way to reach the goal is to go left, hence through the wall. The reason is that almost all other states have a larger euclidean distance to the goal.
So, we will apply the algorithm of section 5.1.3 to this problem and hope for a better performance. We will use the following hyper-parameters:

- As the universal value network, we used a fully connected network with two hidden layers of size 200 and activation function tanh.
- The optimization is made by the Adam optimizer, using a learning rate of $1 \cdot 10^{-3}$.
- The weights of the target network are updated every 500 iterations.
- The Q-network is optimized in the end of every 10 episodes and then 100 optimization steps are performed.

- The exploration rate of the ϵ-greedy strategy is equal to 0.2 during the whole experiment, hence there is no decay. Furthermore, the discount rate is 0.99. A memory of size $1 \cdot 10^6$ is used from which mini-batches of size 1000 are sampled in the learning step.
- As specific parameters of this algorithm, we use $\alpha = 3$ and $\beta = 0.9$ (see section 5.1.3 for the meaning). Furthermore, the goal memory has a limited size of 10 goals. The last state of every 10 episodes is added if it does not already exist in the memory.
- The maximum length of an episode is 100 steps.
- Every 1000 episodes, 100 testing episodes are run.

We see the results of the testing period in fig. 5.5. Note that the agent samples only real goals from the environment in that time. One can deduce that the agent learns perfectly the way to the final goal. In fact, he reaches the final goal in each of the 100 test episodes.

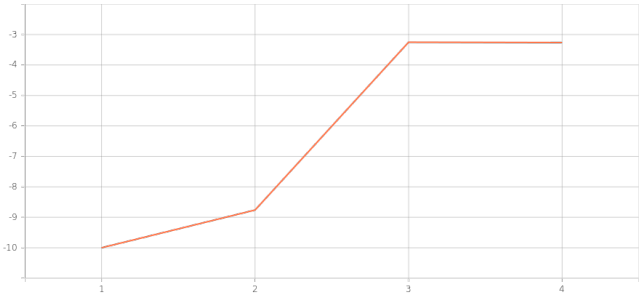


Figure 5.5: Mean test rewards of 100 episodes with iterative goal sampling on gridworld plan 3

Chapter 6

TME 14: Flow models

6.1 Algorithm

Normalizing flows are the third type of generative models which were discussed in the course. The model consists of an encoder-decoder architecture where the decoder is the inverse function of the encoder. This property will limit the choice of encoders. In fig. 1.1 we see the structure of a flow-based model.
After applying the encoder on the original data, we sample under a prior distribution in the latent space and reconstruct the original data through the inverse of the encoder, the decoder. To optimize the model, we use the following theorem:

Theorem 1 *Let $f : X \to Z$ be a diffeomorphism, let p_X be a distribution in X and p_Z a distribution in Z. Then:*

$$p_X(x) = p_Z(f(x)) \cdot |\det(J_f)|$$

Obviously, we want to maximize $p_X(x)$ in generative models, so we train the flow model with the loss

$$-\log(p_X(x)) = -\log(p_Z(f(x)) - \log(|\det(J_f)|)$$

The remaining question is now how to obtain a function f whose inverse and determinant of the Jacobian is known and who has enough parameters for the learning process. First, we have the approach to define f as the composition of some functions f_i. Then we can use the following properties:

$$f = f_1 \circ \ldots \circ f_n \qquad \Rightarrow f^{-1} = f_n^{-1} \circ \ldots \circ f_1^{-1} \qquad \text{and} \qquad \det(f) = \det(f_1) \cdot \ldots \cdot \det(f_n)$$

In this exercise, we used the Glow model to define the functions f_i. The Glow model consists of blocks of the following form (see fig. 6.1):

1. ActNorm is just an affine transformation where the scale and shift parameter is initialized such that the post-actnorm activation has zero mean and unit variance (similar to Batch normalization).

2. Invertible 1x1 Convolution is just a matrix transformation by a invertible matrix.

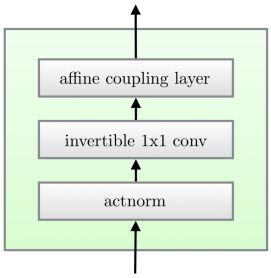


Figure 6.1: A block in the glow model [2]

3. Let $x \in \mathbb{R}^{2l}$, then the affine coupling layer has the following form:

$$\begin{aligned} s &= MLP_{scale}(x_{1:l}) \\ t &= MLP_{shift}(x_{1:l}) \\ y_{1:l} &= x_{1:l} \\ y_{l+1:2l} &= x_{l+1:2l} \odot \exp(s) + t \end{aligned}$$

Note that one can easily compute the inverse functions and the Jacobian of these functions.

6.2 Experiments

We trained the Glow model on the "make moons" and "make circles" data set from sci-kit learn. We chose the following hyperparameters for the flow model:

- 5 blocks as in fig. 6.1
- Hidden size of the MLP in the coupling layer of 64
- We use the Adam optimizer with a learning rate of $1 \cdot 10^{-4}$ for the moons data set and of $5 \cdot 10^{-5}$ for the circles data set. Additionally, a weight decay of 10^{-7} is used.
- We trained the model for 30.000 Iterations.

In this way, we could generate data which is similarly distributed in comparison with the original data (see fig. 6.2). One can also visualize how the single modules of the flow transform the input. This is done in fig. 6.3. From this, one can deduce that the affine coupling layers make the significant transformations and the convolution modules rotate the input.

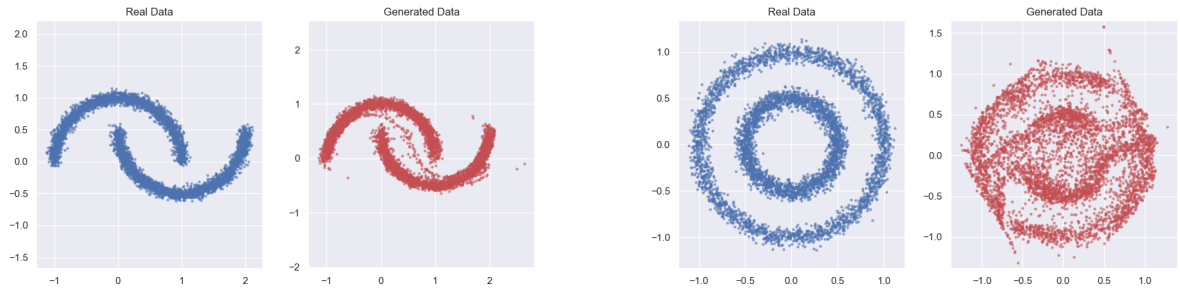


Figure 6.2: Results of the Glow model on the moons (left) and the circles (right) data set

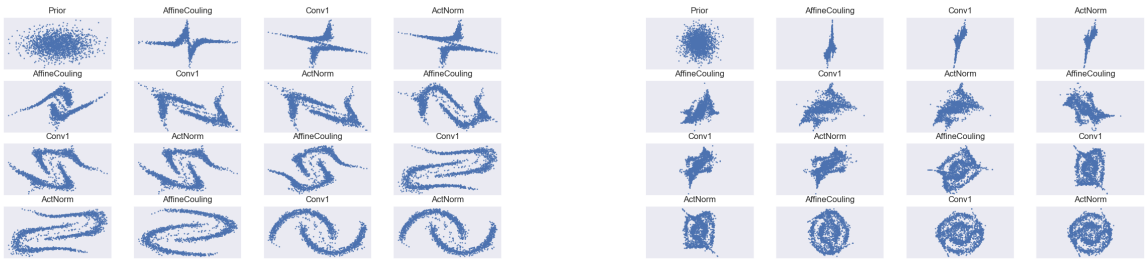


Figure 6.3: Left: Flow visualization of the generated moons (left) and the circles data set (right)

Bibliography

[1] *Cours source, RLD*. https://dac.lip6.fr/master/rld-2021-2022/.

[2] Diederik P. Kingma and Prafulla Dhariwal. *Glow: Generative Flow with Invertible 1x1 Convolutions*. 2018. arXiv: 1807.03039 [stat.ML].

[3] Ryan Lowe et al. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments". In: *CoRR* abs/1706.02275 (2017). arXiv: 1706.02275. URL: http://arxiv.org/abs/1706.02275.

[4] Alec Radford, Luke Metz, and Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2016. arXiv: 1511.06434 [cs.LG].

[5] Lilian Weng. "Flow-based Deep Generative Models". In: *lilianweng.github.io/lil-log* (2018). URL: http://lilianweng.github.io/lil-log/2018/10/13/flow-based-deep-generative-models.html.