Active Learning from Demonstrations

Théo VINCENT
ENS Paris-Saclay & ENPC
theo.vincent@eleves.enpc.fr

1. Introduction

The main purpose of Reinforcement Learning (RL) is to learn a policy that would lead an agent to solve a problem. To learn this policy, most algorithms rely on the Bellman equation that appears when the problem is modelled as a Markov Decision Process. To compute this equation, we need to approximate expectations with estimators that can be built from interactions with the environment. The policy used to interact with the environment is called the behavioral policy.

In this project, we investigate how can we improve RL algorithms that would be provided with demonstrations from expert policies. We will first describe the environment we will work with in section 2. Then, we will explain how we get the expert samples in section 3. In section 4, we will show two examples of supervised losses. In section 5, we will lean on three different settings where demonstrations can be added. Finally, in section 6 and 7 we will experiment the previously introduced algorithms and conclude.

The code for this project is available here [6].

2. Environment

We will focus on only one type of environment: a GridWorld. Figure 1 shows how the game looks like. Note that in our setting the reward only depends on the state, we will still assume that the reward depends on the state-action pair to get more general algorithms. The environment is coming from the package RL-Berry [1].

In some algorithms, we will need to use features to describe the state-action pair. We will simply use one hot encoded vectors. Since there are 31 states and 4 actions for each of them, those features will be of 124 elements.

I chose to set the maximum reward to 0.1 because of the fact that some algorithms use the exponential and this choice reduces the chance of overflows. We will develop this choice in section 5.3.

3. Demonstrations

In the different algorithms that will be presented in section 5, we will use demonstrations composed of single step in some cases and full trajectories in some others. We will use dynamic programming to compute an optimal policy. Figure 2 shows the chosen expert policy. It is important to note that there are plenty of optimal policies. This is only one example.

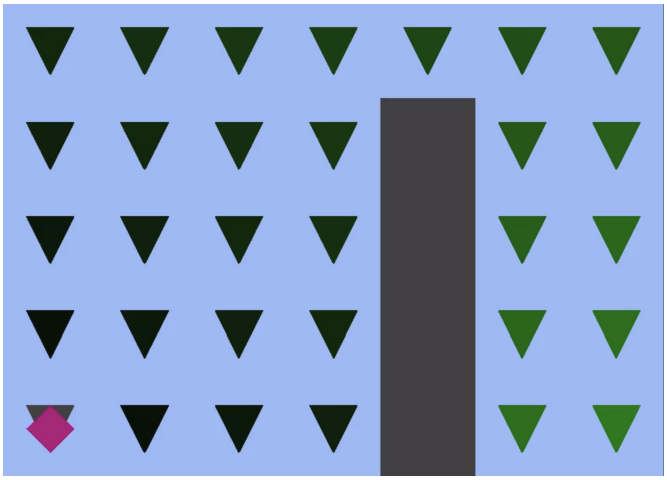

Figure 1. GridWorld: the agent is the pink diamond, the triangles show that the reward keeps increasing until the terminal state located at the bottom left. The black boxes represent a wall.

4. Supervised losses

In the papers that I studied, I have encountered two supervised losses that help to take demonstrations into account. We will call the one from the paper of Kim et al. [4] "large margin loss". For a state s and an expert action a, it can be written as follow:

$$\mathcal{L}_E(s,a) = \max\{1-(Q(s,a)-\max_{a' \in A \setminus \{a\}} Q(s,a')), 0\} \quad (1)$$

We will call the loss introduced in the work of Hester et al. [3] "penalized loss". It can be written as follow:

$$\mathcal{L}_E(s,a) = \max_{a' \in A}\{Q(s,a) + \ell(a,a')\} - Q(s,a) \quad (2)$$

where $\ell(a,a')$ is a penalization loss that is null when $a = a'$ and positive otherwise.

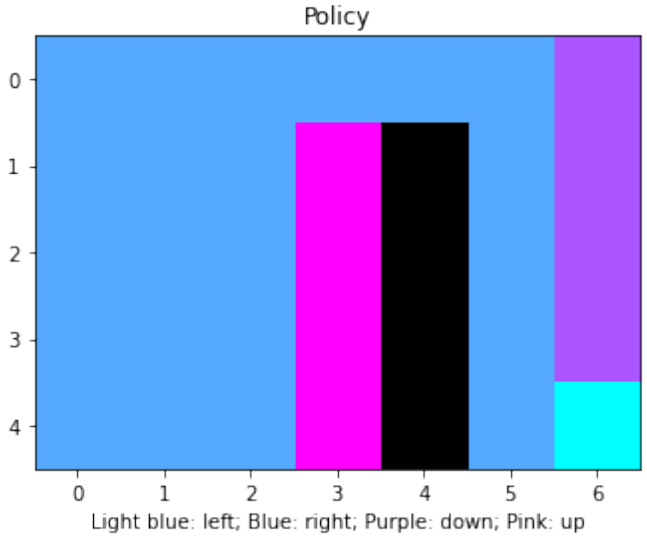

Figure 2. The chosen expert policy on the environment.

It is worth noticing that in our case, those losses are equal if we choose $\delta = 1$ with:

$$\ell(a, a') = \begin{cases} 0 & \text{if } a = a', \\ \delta & \text{if } a \neq a' \end{cases}$$

In our environment, it is hard to say if an action is more relevant than another one. This is why the previous setting for ℓ will be used. The value of delta matters a lot. Setting it to 1 is too high since in our case the maximum reward is 0.1. This is the reason why the large margin loss won't be used and we will only focus on the penalized loss where this "large margin" can be used.

5. Algorithms

5.1. Linear setting

We are first going to investigate the work of Kim et al. [4]. They propose a general Q-learning framework shaped as a Least Square Policy Iteration (LSPI) to solve an RL problem with demonstrations. They define the problem as follow:

$$\begin{aligned} \underset{Q \in F}{\arg\min} \quad & \frac{1}{\#D_B} \sum_{(s,a) \in D_B} (Q(s,a) - U^*(s,a))^2 \\ & + \frac{\alpha}{\#D_E} \sum_{(s,a) \in D_E} \mathcal{L}_E(s,a) + \lambda J(Q) \\ \text{subject to} \quad U^* \in \underset{U \in F}{\arg\min} \quad & \frac{1}{\#D_B} \sum_{(s,a) \in D_B} (U(s,a) \\ & - \hat{\Gamma} Q(s,a))^2 + \lambda' J(U) \end{aligned}$$

where D_B is a set of samples on which we are trying to reduce the Bellman error. D_E is the set of expert samples. $\hat{\Gamma}$ is the approximate Bellman operator, they propose to define it as: $\hat{\Gamma} Q(s,a) = r(s,a) + \gamma Q(s', a')$. J is a regularization function over the space F. α is the trade-off between the Bellman and the supervised loss. λ and λ' are here to help taking into account the regularization terms. The function $\mathcal{L}_E$ is a supervised loss. It can be the function introduced in section 4.

As in the original paper, we are going to investigate this framework in the case of linear approximation which gives a Least Square Temporal Difference (LSTD) version of the framework. They call this algorithm Approximate Policy Iteration with Demonstration (APID). Given a feature function that maps a state-action pair to a vector in $\mathbb{R}^f$, we will call it ϕ, we will approximate $Q(s,a)$ by $\phi(s,a)^T \cdot w$ and $U(s,a)$ by $\phi(s,a)^T \cdot u$ where w and u belong to $\mathbb{R}^f$. For example, this feature function can be the one described in section 2.

I decided to solve the problem iteratively. This means, that from a given fixed w we will find the u minimizing the function:

$$f(u) = \frac{1}{n} \sum_{i=1}^{n} (\phi_i^T u - (r_i + \gamma \phi_i'^T w))^2 + \lambda' u^T u \qquad (3)$$

where $D_B = \{(s_i, a_i, r_i, s_i', a_i')_{i=1}^n\}$ and $\forall i, \phi_i = \phi(s_i, a_i)$ and $\phi_i' = \phi(s_i', a_i')$. As mentioned in the paper, there exist a closed form formula for the minimizer of this function. Indeed, this function is convex and coercive, hence the minimizer is at a critical point. By computing the gradient, we can see that there is only one critical point. Its expression can be derived easily:

$$u^* = (\Phi^T \Phi + \lambda' n I)^{-1} \Phi^T (r + \gamma \Phi' w) \qquad (4)$$

where $\Phi = \begin{bmatrix} \phi_1^T \\ ... \\ \phi_n^T \end{bmatrix}$ and $r = \begin{bmatrix} r_1 \\ ... \\ r_n \end{bmatrix}$.

For the general minimization problem, we can try to solve this problem given a fixed u. The function to optimize is the following:

$$g(w) = \frac{1}{n} \sum_{i=1}^{n} (\phi_i^T w - \phi_i^T u)^2 + \frac{\alpha}{m} \sum_{j=1}^{m} \mathcal{L}_E(s_i, a_i) + \lambda w^T w \qquad (5)$$

where $D_E = \{(s_i, a_i, r_i, s_i', a_i')_{i=1}^m\}$. Depending on the nature of $\mathcal{L}_E$, the complexity of the problem can change. Here, we will be able to apply gradient descent since the gradient of the supervised loss is easy to compute. We cannot compute the minimizer with a closed form formula directly since the formula depends on the region of the space. The gradient is expressed as follow:

$$\frac{\partial g(w)}{\partial w} = \frac{2}{n}\Phi^T\Phi(w-u)+\frac{\alpha}{m}\sum_{j=1}^{m}\frac{\partial \mathcal{L}_E(s_i,a_i)}{\partial w}+2\lambda w \quad (6)$$

where

$$\frac{\partial \mathcal{L}_E(s_i,a_i)}{\partial w} = \phi(s_i,a_i^*) - \phi_i \quad (7)$$

with $a_i^* = argmax_{a\in A}\{\phi(s_i,a)^T w + \ell(a_i,a)\}$

It is interesting to notice that the gradient is not costly to compute since $\Phi^T\Phi$ can be computed offline (from the gradient descent) and in general the number of demonstration m is small.

All in all, the final algorithm is summarized in algorithm 1. The set of demonstrations D_E are single transitions drawn randomly over the states. Similarly, the state of RL samples D_E are single transitions drawn randomly over the states, a greedy policy is used with respect to the last computed w. Here, as the algorithm is following the policy iteration schema, the evaluation step uses the Bellman operator for the greedy policy of the current step. Since the next action is stored in the sample, the set of RL sample has to be emptied at each iteration to keep a strict on policy learning schema.

Algorithm 1 APID

Require:

lr ▷ Learning rate for the gradient descent
tol ▷ Tolerance for the gradient descent
n ▷ Number of RL samples per iteration
D_E ▷ Filled with demonstration samples

for i = 0:n_iterations **do**
  $D_B \leftarrow$ collect n RL samples from ϵ-greedy policy
  $u \leftarrow$ optimize u ▷ From equation 4
  while $||\frac{\partial g(w)}{\partial w}|| \geq tol$ **do**
    $w = w - lr * \frac{\partial g(w)}{\partial w}$ ▷ From equation 6
  end while
end for
return w

5.2. Tabular setting

Let's now focus on the work of Hester et al. [3]. They introduce a Q-learning algorithm that integrates demonstrations during training. The algorithm is called Deep Q-learning from Demonstrations (DQFD). They propose to use a loss that combines Temporal Difference (TD(0)) loss, a N-step TD loss, a supervised loss and a regularization loss. As we are going to investigate the tabular setting, we will ignore the regularization loss. The Q update writes as follow:

$$\begin{aligned} Q(s_t,a_t) = Q(s_t,a_t) + \mu[&\lambda_1(r_t + \gamma \max_{a'\in A} Q(s_t,a)) \\ &+ \lambda_2(r_t + \gamma r_{t+1} + ... + \gamma^n \max_{a'\in A} Q(s_{t+n},a')) \\ &+ \lambda_3 \mathcal{L}_E(s_t,a_t) - Q(s_t,a_t)] \end{aligned} \quad (8)$$

with $\lambda_1 + \lambda_2 + \lambda_3 = 1$ to avoid overshooting.

One of the most powerful idea that the paper has is to use a prioritized replay buffer. In stead of randomly sampling samples over the replay buffer, they compute a probability of sampling each sample. This probability relies on the last computed TD(0) error. It has the following formula:

$$P(i) = \frac{p_i^\alpha}{\sum_{k=1}^n p_k^\alpha} \quad (9)$$

where p_i is defined as $p_i = |\delta_i| + \epsilon^*$, with δ_i the last TD(0) error of sample i and ϵ^* a value that depends on the nature of the sample. If it is a demonstration sample, the value will be high otherwise it will be smaller. The step size μ of equation 8 is defined from the probability p_i with: $\mu = (\frac{1}{np_i})^\beta$, where β is a parameter in (0, 1). When the last TD(0) error is not computed for a sample, I decided to define it has the maximum over all previously computed TD(0) error for all state-action pairs so that it has the highest chance to be sampled. If a new collected sample represents a state-action pair that has already been seen, its last TD(0) error will be set has the error value of the already seen sample. This avoids sampling again a sample that has already been seen.

All in all, the DQfD algorithm for the tabular case is represented in algorithm 2. Contrarily to algorithm 1, the set of RL samples is constantly increasing since we are approximating the optimal Bellman operator here and not the Bellman operator.

5.3. Neural network setting

Let's now dive into the work of Gao et al. [2]. They propose a policy gradient algorithm while taking into account the entropy of the policy. They call their algorithm NAC for Normalized Actor Critic. Usually, policy gradient methods use a neural network with two heads that predicts the V function and the policy. Their idea is to use a neural network to approximate the Q function. Once Q is computed, they obtain the V function and the policy with the following equations:

$$V_Q(s) = \alpha \log \sum_{a\in A} \exp \frac{Q(s,a)}{\alpha} \quad (10)$$

$$\pi_Q(a|s) = \exp \frac{Q(s,a) - V_Q(s)}{\alpha} \quad (11)$$

Equation 10 ensures that $\sum_{a\in A} \pi_Q(a|s) = 1$. We can already see that using the exponential will lead to overflows

Algorithm 2 DQfD

Require:

n_steps ▷ For n_step td loss
n_expert, n_rl ▷ Number of iterations
$\lambda_1, \lambda_2, \lambda_3$ ▷ Weight of the losses
update_freq ▷ Update target frequency
D ▷ Filled with demonstration trajectories

initialize Q and Q_target
$s =$ initial state
for $i = 1$:n_expert + n_rl **do**
  if $i >$ n_expert **then**
    $a \leftarrow$ epsilon greedy from Q
    observe r, s'
    $D \leftarrow (s, a, r, s')$
  end if
  transitions, step_size $\leftarrow$ sample n_step transitions from D
  update Q as equation 8 with transitions and step_size
  store TD(0) error for the first transition
  if $i = 0$ [update_freq] **then**
    $Q\_target \leftarrow Q$
  end if
  if $i >$ n_expert **then**
    $s \leftarrow s'$
  end if
end for
return Q

if it is not taken seriously into account. Indeed, in equation 10, if $\alpha = 0.1$, the term in the exponential worth at most $\frac{Q_{max}}{\alpha}$. To keep reasonable values, I chose to shape the reward so that the Q function is bounded by 1.

We can compute the actor's and the critic's losses:

$$\nabla_\theta J_{\text{actor}} = \nabla_\theta \mathbb{E}_{s,a\sim\pi_Q(.|s)} \left[r(s,a) + \alpha H(\pi_Q(.|s)) \right] \quad (12)$$

$$\nabla_\theta J_{\text{critic}} = \nabla_\theta \mathbb{E}_s \left[\frac{1}{2} (V_Q(s) - \hat{V}(s))^2 \right] \quad (13)$$

where α is the importance given to the entropy term. By deriving the equations, with a sample (s, a, r, s'), they come up with the following equations:

$$\nabla_\theta J_{\text{actor}} = (\nabla_\theta Q(s,a) - \nabla_\theta V_Q(s))(Q(s,a) - (r(s,a) + \gamma \overline{V}_Q(s')) \quad (14)$$

$$\nabla_\theta J_{\text{critic}} = \nabla_\theta V_Q(s)[V_Q(s) - (r(s,a) + \gamma \overline{V}_Q(s') + \alpha H(\pi_Q(.|s))] \quad (15)$$

where $\overline{V}$ is the value of V obtained with a target network. The actor's loss has an interesting property. Indeed, when $Q(s,a) > (r(s,a) + \gamma \overline{V}_Q(s')$, the update is going to decrease the value of Q since it goes to the opposite direction of the gradient of Q but in the same time it increases V, from the equation 10, the Q value of the other actions will be increased. The opposite behavior will also happen when $Q(s,a) < (r(s,a) + \gamma \overline{V}_Q(s')$. This is interesting since the update of one state-action pair also changes the value of the Q function for the other actions. The authors chose to call this algorithm NAC from its "normalizing" behavior. They argue that this allows the agent to learn the demonstrations in a better way since it also changes the Q value of the actions that are not shown as demonstrations.

All in all, the algorithm 3 describes the steps to follow. The authors state that learning from demonstration is way different than off policy learning since in off policy learning the state-action pairs are assumed to be visited infinitely often which is far from being the case when we have demonstrations. This is why they think that having a supervised loss is not a good way to approach the problem.

Algorithm 3 NAC

Require:

n_expert, n_rl ▷ Number of iterations
α ▷ Importance of the entropy
batch_size
update_freq ▷ Update target frequency
D ▷ Filled with demonstration samples

initialize Q and Q_target with θ and θ_{target}
$s =$ initial state
for $i = 0$:n_expert + n_rl **do**
  if $i >$ n_expert **then**
    $a \sim \pi_Q$
    observe r, s'
    $D \leftarrow (s, a, r, s')$
  end if
  batch $\leftarrow$ sample batch_size samples from D
  $\theta = \theta - lr \cdot (\nabla_\theta J_{\text{actor}} + \nabla_\theta J_{\text{critic}})$
  if $i = 0$ [update_freq] **then**
    $\theta_{\text{target}} \leftarrow \theta$
  end if
  if $i >$ n_expert **then**
    $s \leftarrow s'$
  end if
end for
return w

6. Experiments

To evaluate the policy functions that the algorithms will produce, I need to define a metric that mesures how well the agent have understood the problem globally. This is why I chose to count the number of non optimal actions that a

policy would take. I will call it NNOA for Number of Non Optimal Action. This metric is interesting since it let the freedom to the algorithm not to follow the demonstration policy.

I wanted to make a point on the initialisation of the Q function. The three algorithms were really difficult to make them working if the Q function was not initialized as the minimal possible value. In our case the Q function is lower bounded by 0 so I made sure that the Q function were initialized with 0. This would make the Q function learn the reward first. I can try to explain the intuition that I have on it. Let's assume that we have an expert demonstration for a state-action pair (s, a). In most algorithm we use the TD update that has the following shape: $\theta = \theta - \mu \nabla Q(s, a)(Q(s, a) - \Gamma Q(s, a))$, where Γ is an operator on Q that uses the value Q over the other actions. If the Q function is badly initialized on our demonstration pair, we will have $Q(s, a) \leq \Gamma Q(s, a)$, then we will move our weights in the direction to increase $Q(s, a)$. This is something that we want since the pair (s, a) is a demonstration so it should have a greater Q value that the other actions. This pushes us to have a big μ so that the Q function quickly learn the good shape. The problem is that if the Q function is nicely initialized for our demonstration sample, the update will push the Q function to be decreased on the sample so that the Q value for the sample might be smaller than the Q values for the other action and hence a wrong policy. For this reason I chose to initialize the Q function with 0 so that it focuses on the reward.

In the following results, I have to say that the performances highly rely on tuning the hyperparameters.

6.1. Linear setting

I first wanted to check if the algorithm 1 is working with 200 expert samples and with 0 rl samples. In that scenario, $D_B = D_E$ and $\alpha = 0$ so that we recover the LSPI framework. The algorithm managed to replicated the demonstration policy in only 2 iterations.

Table 1 shows the performances of the algorithm on the environment. It compares the LSPI scenario, where the demonstrations are considered as RL samples, with the APID scenario where the demonstrations are only taken into account in the supervised loss. In both scenarios, 20 demonstrations and 100 RL samples were given at each iteration, only the RL samples were changed at each iteration. I let the algorithm run for 10 iterations to learn the best weights.In this table we can see that the APID algorithm significantly performs better than the LSPI algorithm. Nonetheless, the better performance has a cost in time.

6.2. Tabular setting

Before evaluating the performances of the algortihm 2, I wanted to make sure that the algorithm works with 200 demonstration trajectories and 2000 expert iterations. This is the case, the algorithm manages to fully learn the demonstration policy.

	LSPI	APID
NNOA	12.8 ± 2.4	$\mathbf{9.1 \pm 1.18}$
Time	0.4 ± 0.1	24.2 ± 2.0

Table 1. Average and standard deviation of the NNOA for LSPI and APID over 10 experiments. Time in second.

Table 2 shows the performances of the algorithm on the environment. Each experiment has been done with the same setting. Indeed, 20 demonstration trajectories were used. 0 expert iterations and 100 RL iterations were done. This choice seems surprising but in practice, it puts the algorithm with a random replay buffer into a difficult setting since the algorithm has less chance to peak a demonstration sample. This is verified in the table, the prioritized replay buffer is significantly better than the random one on each row. Once again, the time required to compute the best algorithm is bigger. This is due to the computations of the probabilities. If we now compare the rows, the results are not so different from each other. This is where we would need to experiment those losses in a more complex environment.

	prioritized	random
TD loss	$\mathbf{6.8 \pm 1.7}$	12.0 ± 1.9
3-step TD loss	$\mathbf{6.6 \pm 1.8}$	11.2 ± 1.9
Supervised loss	$\mathbf{6.8 \pm 1.9}$	13.0 ± 1.7
Convex combination	$\mathbf{6.3 \pm 1.3}$	11.1 ± 2.4
Time	0.21 ± 0.07	0.04 ± 0.01

Table 2. Average and standard deviation of the NNOA for different settings over 10 experiments. Each column represents a way of sampling samples in the replay buffer. Each line represents a different loss. Time in second.

Interestingly, the experimentation for this algorithm did not work at first. Indeed, the best results were always given by the random replay buffer. When we look that the update formula of Q (equation 8) we see that, in the tabular case, the new value of Q is a convex combination of the old version of Q and the value of Q applied to a custom operator. For this to be true, we need to have μ to be in [0, 1]. The problem is that $\mu = (\frac{1}{np_i})^\beta$ and if the sampled probability is lower than $\frac{1}{n}$ then μ is greater than 1. I decided to cap μ at 1, this change made the algorithm work way better.

6.2.1 Neural network setting

Similarly to what I have done before, I have checked that the algorithm 3 was able to learn when 150 demonstration transitions were given and 50 iterations of batches of size 10 were performed. This was the case, the algorithm managed

to imitated the demonstration policy perfectly.

I chose to compare the NAC algorithm with the exact same algorithm but with a different update, a classical Q-learning update:

$$\nabla_\theta J(s,a) = \nabla_\theta Q(s,a)(Q(s,a) - (r(s,a) + \gamma \max_{a' \in A} Q(s',a')))$$

The architecture used here is simply a linear layer followed by a sigmoid. The sigmoid function is not a problem since the maximal possible Q value is: $\frac{r_{max}}{1-\gamma} = \frac{0.1}{1-0.9} = 1$. On the contrary, it helps to model to stay in reasonable Q values.

Table 3 shows the results of the performances of the algorithm. 20 expert samples were given, 10 iterations on them were performed before collecting RL samples and iterating 500 times. The batch size was 10. We can see that the NAC update performs better that the Q-learning update. It would be interesting to try in other environments since the difference is not significant here. The Q-learning algorithm trains faster since the computations for the gradients of the NAC update are more complex.

	Q-learning	NAC
NNOA	11.6 ± 1.8	$\mathbf{9.5 \pm 2.54}$
Time	4.2 ± 1.3	20.9 ± 4.1

Table 3. Average and standard deviation of the NNOA for Q-learning and NAC over 10 experiments. Time in second.

7. Conclusion

In this work, we have gone through 3 different algorithms that use demonstrations to boost their performances. Each algorithms correspond to a different setting, however their setting are not fixed and can be changed. Through this report, we have seen two major ways of dealing with demonstrations, either we used a supervised loss (algorithms 1 and 2) or we changed the update rule (algorithm 3). It is important to note that some techniques developed for one algorithm can be used for another algorithm. For instance, algorithm 3 could use a prioritized replay buffer introduced in 2.

In this report, we have always fixed the demonstration data set before the training, some algorithms have also been craft in a way where it is possible for the agent to query demonstrations through out the training. For example, there is the DAgger [5] algorithm.

Finally, an extension of this work could consist of using sub-optimal experts. For instance, I could have introduced random sub-optimal actions in the chosen expert. Another idea would be to give the same expert policy described in section 3 in the same environment with the possibility to reach the terminal state just by going always right. This means that the wall at the bottom would be removed. Some works have also formalized the expert policy as an agent, with the same architecture than the principal agent, that is provided with more information. This is why this expert policy is be better. For example Zhang et al. have tried this idea applied to end-to-end driving [7]. The interesting benefit is that as the architecture of the expert is similar to the principal agent, we are sure that the agent can learn the shown policy. It is not always the case if we use human demonstrations. Another possible extension would be to investigate how it is possible to make the agent learn a good policy only from demonstrations. It can be seen as a pre-training phase where the agent has to learn a good enough policy to be able to interact with the environment is a safe way.

References

[1] Omar Darwiche Domingues, Yannis Flet-Berliac, Edouard Leurent, Pierre Ménard, Xuedong Shang, and Michal Valko. RLBerry - A Reinforcement Learning Library for Research and Education, 2021.

[2] Yang Gao, Huazhe(Harry) Xu, Ji Lin, Fisher Yu, Sergey Levine, and Trevor Darrell. Reinforcement learning from imperfect demonstrations, 2018.

[3] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, Gabriel Dulac-Arnold, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep q-learning from demonstrations. In *AAAI*, 2018.

[4] Beomjoon Kim, Amir-massoud Farahmand, Joelle Pineau, and Doina Precup. Learning from limited demonstrations. In *Advances in Neural Information Processing Systems*, 2013.

[5] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. No-regret reductions for imitation learning and structured prediction. *CoRR*, 2010.

[6] Théo Vincent. Reinforcement Learining with Demonstrations. `https://github.com/theovincent/ReinforcementLearningWithDemonstration`, 2021.

[7] Zhejun Zhang, Alexander Liniger, Dengxin Dai, Fisher Yu, and Luc Van Gool. End-to-end urban driving by imitating a reinforcement learning coach. *ICCV*, 2021.