Learning robotic manipulation with behavioral cloning

Marc Ducret Lucas Willems

Abstract

Learning the policy that maximizes reward in an environment can be done without learning from reward but by cloning the behavior of an expert. This report presents algorithms (BC, DAgger, DART) that exploit this idea and compares them on the Mujoco FetchPickAndPlace environment.

1. Introduction

The policy that maximizes reward in an environment can be learnt without learning from reward (i.e. without doing reinforcement learning) but by cloning the behavior of an expert (e.g. a human) that already knows the optimal policy.

In this report, we present (section 2) three algorithms (BC, DAgger, DART) that exploit this idea and compares them (section 3) on the Mujoco FetchPickAndPlace environment.

2. Behavioral cloning algorithms

Behavioral cloning is possible if we have access to some behavior. Usually, this access is made through *trajectories*, i.e. sequences of observations and actions $(o_1, a_1), ..., (o_T, a_T)$ where a_t is the action taken by the behavior when it receives the observation o_t and o_{t+1} the observation sent by the environment after receiving action a_t. Trajectories collected from the behavior of an expert are called *demonstrations*.

2.1. Vanilla Behavioral Cloning (aka BC)

Vanilla behavioral cloning consists in collecting demonstrations $d_1, ..., d_N$ and learning to map the observations in the demonstrations to the optimal actions given by the expert. This can be done with deep learning for example.

Issue. In practice, an agent trained with BC tends to perform well (meaning it takes the optimal actions) on expert's trajectories, but worst on its own trajectories (what really matters). This gap in performance, called *covariate shift*, is due to error accumulation (figure 1): each error made by the agent deviates it more from expert's trajectory and leads it to places where it is less familiar and more likely to do bigger errors and deviate even more.

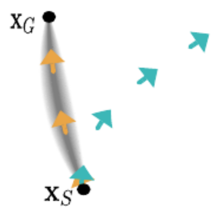

Figure 1. Error accumulation in BC

2.2. DAgger

The DAgger algorithm (introduced in [2]) aims at reducing covariate shift. It doesn't train the agent on expert's trajectories but rather on its own trajectories corrected by the expert (figure 2), i.e. on its own trajectories where its actions had been replaced by expert's actions.

A coefficient β_t (that varies over time) is introduced to find a trade-off between the percentage of agent's corrected trajectories and expert's trajectories in the training dataset. The more the agent learns, the less expert's trajectories there are.

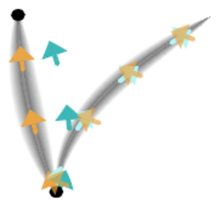

Figure 2. DAgger

Notations. π^* designates the expert policy.

Issue. This algorithm is on-policy meaning that the expert corrects agent's actions. This can be tedious for a human expert.

Algorithm 1: DAgger

```
1 $\mathcal{D} := \emptyset$ ;
2 Initialize classifier $\hat{\pi}_1$ ;
3 for $i \leftarrow 1$ to $N$ do
4     $\pi_i := \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$ ;
5     Sample T-step trajectories $\gamma_1, ..., \gamma_K$ using $\pi_i$ ;
6     $\mathcal{D} := \mathcal{D} \cup \{(s, \pi^*(s)) : s$ state in $\gamma_1, ..., \gamma_K\}$ ;
7     Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$ ;
8 return best $\hat{\pi}_i$ on validation ;
```

2.3. DART

The DART algorithm (introduced in [1]) is off-policy, as BC, but is robust. It adds noise Σ_t (that varies over time) to expert's trajectories to anticipate agent's errors (figure 3). The more the agent learns, the less error is added.

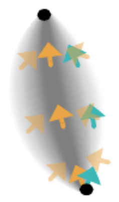
Figure 3. DART

Algorithm 2: DART

```
  Input: $\psi_1$
1 $\mathcal{D} := \emptyset$ ;
2 Initialize classifier $\hat{\pi}$ ;
3 for $i \leftarrow 1$ to $N$ do
4     Sample T-step trajectories $\gamma_1, ..., \gamma_K$ using $\pi^*_{\psi_i}$ ;
5     $\mathcal{D} := \mathcal{D} \cup \{(s, \pi^*(s)) : s$ state in $\gamma_1, ..., \gamma_K\}$ ;
6     Train classifier $\hat{\pi}$ on $\mathcal{D}$ ;
7     Compute $\psi_{i+1}$ ;
8 return $\hat{\pi}$ ;
```

3. Comparison of the algorithms

3.1. The Mujoco FetchPickAndPlace environment

All the algorithms had been compared on the FetchPickAndPlace environment (figure 4) of Mujoco where the robot has to pick a cube and bring it to some 3D point (represented by the red ball).

Observations are images of dimensions $4 \times 224 \times 244$ where each pixel is of dimension 4: 3 dimensions for color (RGB) and 1 for depth.

Actions are of dimensions 4: 3 for describing the velocity of the arm, 1 for describing the velocity of the gripper.

The reward is -1 until the task is achieved.

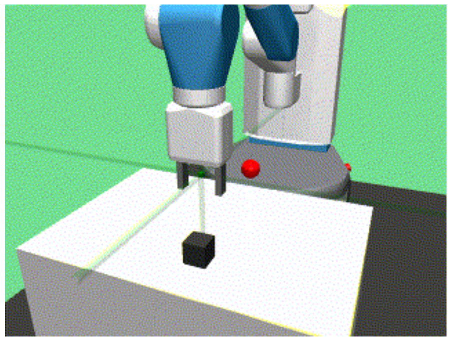
Figure 4. FetchPickAndPlace environment of Mujoco

3.2. Experiments

Difficulties. We encountered several difficulties in order to run experiments:

- We failed at installing MuJoCo on Google Cloud license what led us to use Marc's Windows machine.
- But, the provided code and mujoco-py were not compatible with Windows and required some complicated bug-fixing.
- We had a lot of trouble to adapt the provided code: the provided code was not suited for DAgger and DART that require to add trajectories in the trajectories dataset during training and it was hard for us to understand exactly what we had to modify and where.

Modification of the original code. By just adapting the provided code, we got mediocre results (figure 5).

Hence, we had to consequently modify the provided code.

First, we avoided storing trajectories on the disk and dealing with compression by implementing an online version. In this implementation, trajectories are sampled in batches of 32 while the previous 32 trajectories are being used for training. Training is faster because there is no costly compression and file writing / reading. Moreover it can easily use many trajectories and since every trajectory is only used once, there should be no overfitting. Finally, it was much simpler to implement DAgger and DART in this setting.

Secondly, to make DAgger and DART working correctly, we made a different expert. The provided expert plans all its actions in advance and will therefore not correct errors. Our expert does not use any memory and only chooses its actions based on the current observation, it will therefore react to any error introduced by DAgger or DART.

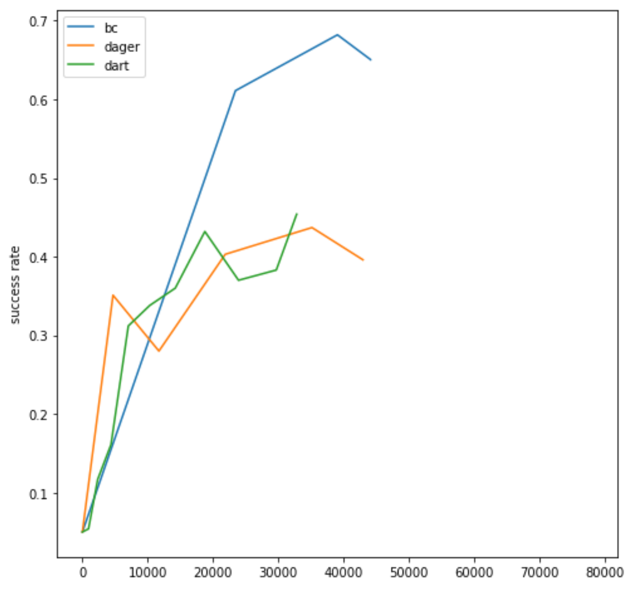

Figure 5. By just adapting the provided code, BC performed better than DART and DAgger.

Experiments setting. All parameters shared between algorithms remained constant during experiments. For each algorithm, we sampled slightly less than $200,000$ episodes which was about 12 hours of training.

- DAgger used $\beta_t = 0.9995^t$ with t being the trajectory batch number (β starts at 1 and ends at 0.05).
- DART used $\alpha = 3$ as suggested by [1] (α is the noise factor).

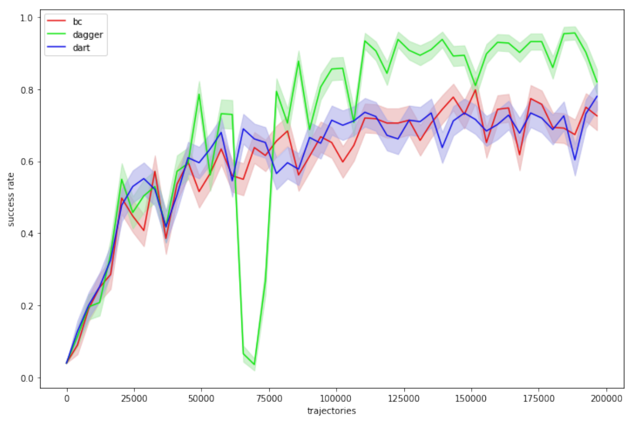

Figure 6. Evaluation success rate

Results. Figure 6 shows success rate at different stages of training for each algorithm. Success rate is computed by averaging on 500 unseen trajectories. The horizontal axis is the number of seen episodes. Figure 7 shows the maximum values in Figure 6. In both figures errors are twice the standard deviation of the average of 500 Bernoulli random variables with p being the observed success rate.

BC	DAgger	DART
$80\% \pm 4\%$	$96\% \pm 2\%$	$78\% \pm 4\%$

Figure 7. Best evaluation success rate

Results analysis. In our experiments, DAgger significantly increased performance, getting a lot closer to expert performance of 99.7%. DART performed like BC but better results might be possible by tuning α.

In this setting, BC is already rather good at learning because most states can be an initial state, therefore the phenomenon of getting to unexplored territory when accumulating errors is less relevant than in other settings.

References

[1] M. Laskey, J. Lee, R. Fox, A. D. Dragan, and K. Goldberg. DART: noise injection for robust imitation learning. In *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings*, pages 143–156, 2017.

[2] S. Ross, G. J. Gordon, and J. A. Bagnell. No-regret reductions for imitation learning and structured prediction. *CoRR*, abs/1011.0686, 2010.