Imitation Learning: Dagger implementation with robotic applications

Ntagkas Alexandros

July 12, 2024

Contents

Introduction

Imitation learning is a technique in machine learning where an agent learns to perform tasks by mimicking expert behavior. Unlike traditional reinforcement learning, which relies on rewards to guide learning, imitation learning directly leverages demonstrations from experts to train models more efficiently and effectively. One of the prominent algorithms in this domain is DAGGER (Dataset Aggregation), introduced to address the shortcomings of earlier imitation learning approaches.
We are going to implement the algorithm in 2 different scenarios:

1. 2D Path Planning
2. Pick and Place with 7-DOF Franka

In each of the 2 cases I am going to explain the logic behind the expert and the privileged learning pipeline.

The expert uses the actual position of the robotic system and obstacle/goal positions, and the network we are going to train uses only visual feedback. To execute the examples you have to

```
pip install pygame, torch, gymnasium, panda_gym,pickle
```

1 2D Path Planning

For the purpose of this implementation I created a 2D world using pygame:

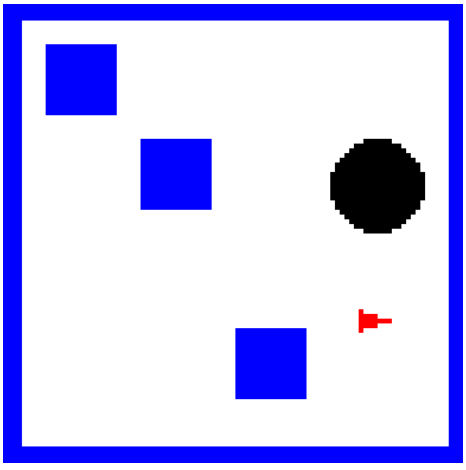


Figure 1: World Example

1.1 Enviroment

I have created a Class called *Enviroment* that contains everything that has to do with rendering, the robot and the training setup in general. It contains the method *step* which takes the action as input and returns the observation (image) and if the episode is done , the method *reset* that resets the episode and the action expert method.

1.2 Expert

The expert in the case of 2d Path planning will be implemented with Artificial Potential Fields. The goal (black circle) acts as an **attractor** and the obstacles and the boarder act with a **repulsive** force.

The robot (red triangle) is an **omnidirectional** drive robot so can achieve any direction we want without constrains.

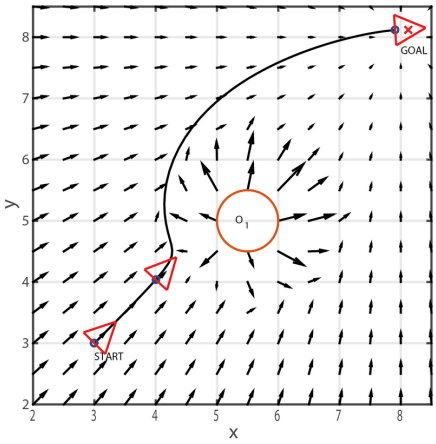


Figure 2: Artificial Potential Fields Example

The goal induces a velocity to the robot of the form:

$$v_{goal} = k_{p1}(goal - robot)$$

so it is attracting the robot and the **n obstacles** induce a force:

$$v_{obst} = \sum_{i=1}^{n} k_{p2}(\text{robot} - \text{obstacle}_i)$$

The final velocity that we use as a command for the robot is $v = v_{goal} + v_{obst}$.

Note: It is usually better if we do not include the obstacles that are not close to the robot. If we use all the obstacles there are many cases where the robot will get stack. So the previous sum will be modified as:

$$S = \{i \mid 1 \leq i \leq n \text{ and } \|\text{robot} - \text{obstacle}_i\| < \text{d}\}$$

$$v_{obst} = \sum_{i \in S} k_{p2}(\text{robot} - \text{obstacle}_i)$$

,also I cliped the velocity from the obstacles to avoid very sudden movement.

1.3 Agent Training

Now that we have build the expert we can move on to train our agent with the DAGGER algorithm.

Algorithm 1 DAGGER Algorithm

Initialize $\mathcal{D} \leftarrow \emptyset$.
Initialize $\hat{\pi}_1$ to any policy in Π.
for $i = 1$ to N **do**
  Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$.
  Sample T-step trajectories using π_i.
  Get dataset $\mathcal{D}_i = \{(s, \pi^*(s))\}$ of visited states by π_i and actions given by expert.
  Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$.
  Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$.
end for
Return best $\hat{\pi}_i$ on validation.

The algorithm suggests for a dataset D to be created at the start which we will be augmenting with new data every episode. Since we want to train out agent exclusively with images the dataset will be :

$D = (s, a)$ where s is the image and a is the action the expert would have taken at this state/image. We will be augmenting the dataset with the *pickle*.

Now that we have obtained the data we need to train a classifier that can map the current image to the desired output. This can be done with a Convolutional Neural Network (CNN).

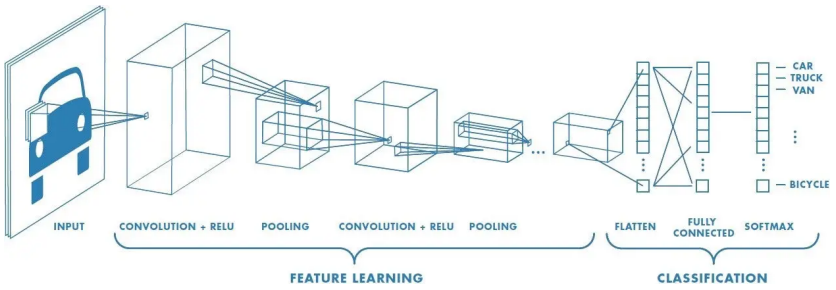


Figure 3: CNN

We will be using the mean squared error loss:

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{1}$$

You can run the DAGGER pipeline with:

```
path/to/your/python main.py
```

1.4 Results

Throughout training I saw that the obstacles and the goal have to stay in the same position to have consistent results and we only change the start position of the robot. For the training I used the following constants:

Parameter	Value
b	0.9^{i-1}
T	100
learning rate	1e-4
k_{p1}	1
k_{p2}	2
obstacle clip	150
d	20

Using the command below and replacing model with the model name (model_17.pth) we can see the result of the trained agent.

```
python test_agent.py -p "./dagger_models/model"
```

Below we see the error from the target in numerous attempts using the trained policy.

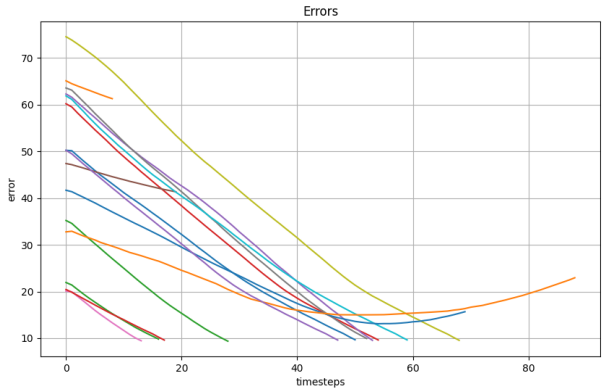


Figure 4: Errors

We can see that in most cases the error goes to 10 which means that the task was successful, in some cases it stops very early which probably indicates a collision with the obstacles, and in some cases where it misses the target. However, the great majority of trajectories go to the goal position so the task is completed.

2 Pick and Place FRANKA

To implement the pick and place scenario I used the Panda Gym library, using some custom wrappers for the task itself.

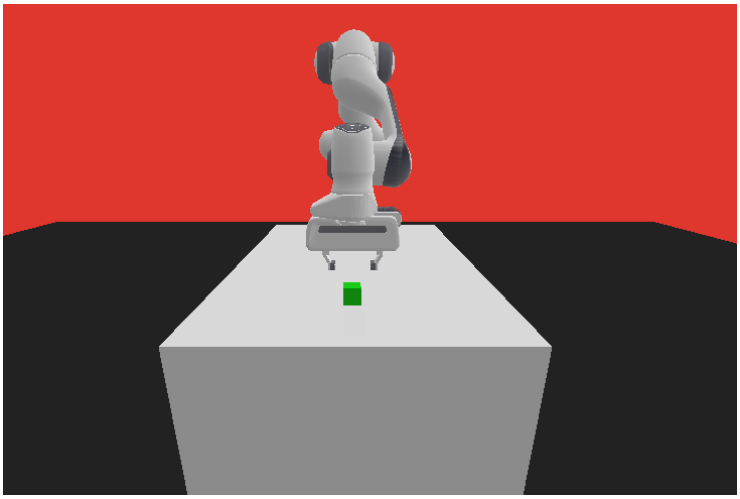
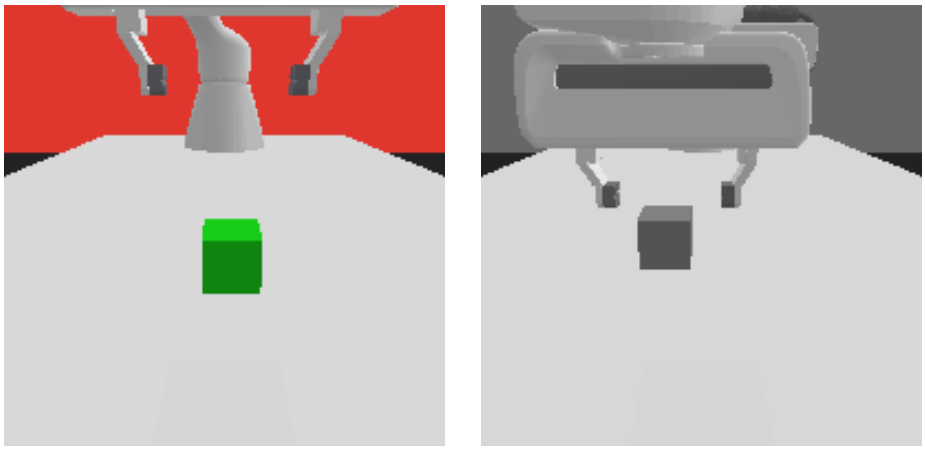


Figure 5: World Example

2.1 Enviroment

I have created a Class called *PandaEnv* that contains everything that has to do with rendering, the robot and the task. I created a Task where the goal position is constant and the object starts at uniformly random positions in the xy plane : $0 \leq x \leq range, -range \leq y \leq range$.

Another difficulty of the task is to place the camera close enough to get a good view of the cube but not too close. Also to process the image we convert it to grayscale, and this is done as so: $gray = 0.2989 * red + 0.5870 * blue + 0.1140 * green$.



(a) Original View (b) Grayscale Convertion

2.2 Expert

The expert in the case of pick and place will be a Finite State Machine (FSM) and the build in inverse kinematics of Panda Gym.

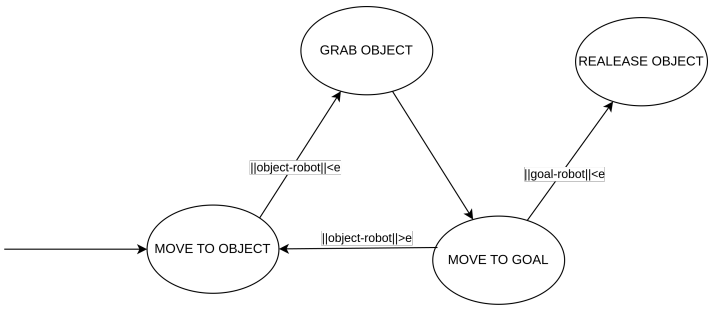


Figure 7: FSM

At start we have to move to the object.

If we reach this position grab the object and move to the goal.

If we lose it in the way return to MOVE TO OBJECT.

If we reach the goal release the object.

At every step where we need to move the robotic arm we provide the desired end effector position and the inverse kinematics takes the robotic arm to that position.

Run the expert with :

```
path/to/your/python test_expert.py
```

2.3 Agent Training

Now that we have build the expert we can move on to train our agent with the DAGGER algorithm.
The dataset will be of the same form.
You can run the DAGGER pipeline with:

```
path/to/your/python dagger.py
```

2.4 Results

For the training I used the following constants:

Parameter	Value
b	0.3^{i-1}
T	200
learning rate	1e-4
e	0.05
goal	(0.2,0,0)

Using the command below and replacing model with the model name (model_5.pth) we can see the result of the trained agent.

```
python test_agent.py -p "./dagger_models/model"
```

Below we see the error from the gaol in numerous attempts using the trained policy.

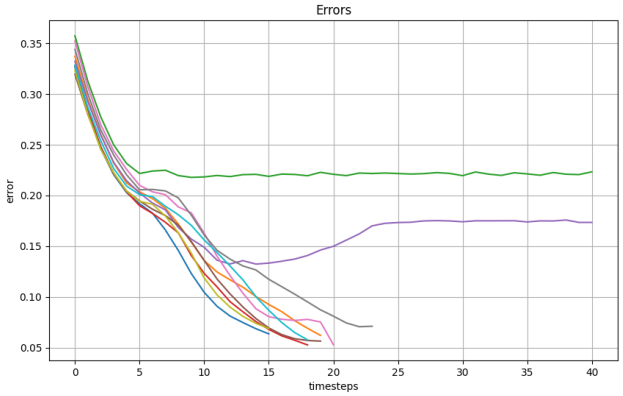


Figure 8: Errors

We can see that in most cases out of the 15 trajectories the error goes to 0.05 which means that the task was successful and in some cases 2/15 it gets stuck at some point where the policy chooses to release the object before it should. However, the great majority of trajectories go to the goal position so the task is completed.

Note: Videos of the trained policies will be provided.

3 Conclusion

Through the use of expert demonstrations and iterative refinement of the policy, we have demonstrated how DAGGER can effectively train agents to perform complex tasks using visual feedback alone.
Throughout these experiments I came across some disadvantages of DAGGER:

1. **Increasing Dataset Size**: As DAGGER progresses, the dataset grows larger with each iteration, which can lead to increased storage requirements and longer training times. This can slow down the overall training process.

2. **Overfitting**: There is a risk of overfitting to the aggregated dataset, especially if the dataset becomes highly imbalanced or if the expert's demonstrations are not diverse enough. This can limit the generalization ability of the learned policy.

3. **Exploration-Exploitation Trade-off**: The balance between exploration and exploitation can be challenging. If the initial policy is poor, the learner might not explore enough diverse states, leading to a biased dataset that does not represent the full state space well.

4 Bibliography

References

[1] Ross, Stephane, Geoffrey Gordon, and Drew Bagnell, *A reduction of imitation learning and structured prediction to no-regret online learning*, Proceedings of the fourteenth international conference on artificial intelligence and statistics, pp. 627–635, 2011.