



Inverted Double Pendulum

Optimal decision making for complex problems

Minne Adrien s154340

Master Civil Engineer

Contents

1	Domain	1
2	Continuous Action Space	2
2.1	Training Parameters	2
2.2	Training Methodology	3
2.3	Final Results	4
3	Discrete Action Space	4
3.1	Training Parameters	4
3.2	Training Methodology	5
3.3	Final Results	5
4	FQI	5
5	Possible improvements	6

1 Domain

- State space (as returned by the environnement) :
 - x : position of the slider, $[-1; 1]$
 - \dot{x} : velocity of the slider, $[-\infty; \infty]$
 - x_2 : position of the top rod of the pendulum, $[-1; 1]$
 - $\cos(\phi_1)$: the cosine of angle ϕ_1 (fig 1), $[-1; 1]$
 - $\sin(\phi_1)$: the sine of angle ϕ_1 (fig 1), $[-1; 1]$
 - $\dot{\phi}_1$: derivative of ϕ_1 , $[-\infty; \infty]$
 - $\cos(\phi_2)$: the cosine of angle ϕ_2 (fig. 1), $[-1; 1]$
 - $\sin(\phi_2)$: the sine of angle ϕ_2 (fig. 1), $[-1; 1]$
 - $\dot{\phi}_2$: derivative of ϕ_2 , $[-\infty; \infty]$
- Action Space : $U = [-1, 1]$
- Dynamics : see code on github
- Integration Step : 0.0165
- Reward Signal : $r_a + r_{dp} + r_{vp}$, where :
 - r_a = alive bonus = 10
 - r_{dp} = distance penalty = $0.01 * x_{robot}^2 + (y_{robot} + 0.3 - 2)^2$
 - r_{vp} = velocity penalty = 0
- Time horizon : $T \rightarrow \infty$
- Deterministic (but chaotic) environment

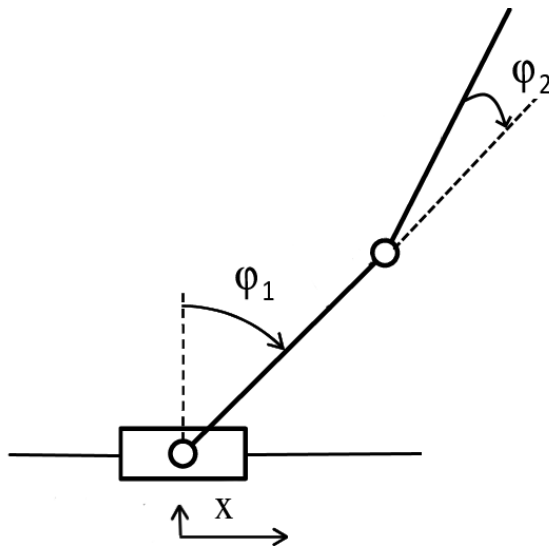


Figure 1: Double Pendulum (diagram edited with my poor knowledge of image modification (in other words : Microsoft Paint), so excuse me for the poor quality)

REINFORCE algorithm:


- 
1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
 2. $\nabla_\theta J(\theta) \approx \sum_i (\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)) (\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i))$
 3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Figure 2: REINFORCE Algorithm

2 Continuous Action Space

For the continuous action space case, I used a policy gradient method with a Gaussian law and REINFORCE algorithm to train my agent. The idea behind policy gradient methods is to have a policy with parameters θ , $\pi_\theta(a|s)$ that will take an action a based on the current state s . The algorithm will generate a trajectory with this policy, observe the results and then update the parameters of the policy to improve it.

The update of the policy is done through the REINFORCE algorithm, presented on figure 2. The policy will be modelised by a neural network. We can transform the update function from the REINFORCE algorithm into the loss function L that will be used for the neural network. For a batch of trajectories $\{t_i\}$ of size M , trajectories with t steps, we have

$$L = \frac{1}{M} \sum_{i=0}^M \sum_t \log(\pi_\theta(a_t^i|s_t^i)) * \sum_t R(s_t^i, a_t^i)$$

with $R(s_i, a_t)$ being the cumulative reward of the trajectory until step t .

As stated before, I used a Gaussian law. This means that the Neural Network will give the mean of a Normal Distribution as output. We can then build a normal distribution with the mean outputted by the NN, and a fixed variance. The policy will generate an action by sampling an action from this distribution.

To summarise, at each episode I will :

- Generate a batch of trajectories from the current policy. The policy will sample an action from the Normal Distribution outputted by the NN.
- Update the weights of the NN using the previously defined loss function.
- Repeat for a certain number of episodes.

One problem of the policy gradient methods is that they suffer from high variance during training. This problem can be lessened by using a baseline b and changing the loss function into :

$$L = \frac{1}{M} \sum_{i=0}^M \sum_t \log(\pi_\theta(a_t^i|s_t^i)) * \sum_t (R(s_t^i, a_t^i) - b)$$

and taking b as the mean reward for the batch.

To help the training of the NN, I will also normalize the data. To do that, I generated lots of random trajectories to get the mean and std of the states, then during the training I subtract the mean and divide the std on the state before feeding them to the NN.

The code is in the file `continuous.py`.

2.1 Training Parameters

There are several parameters that will influence the quality and speed of the training:

Decay factor The decay factor γ wasn't specified at the time I did these experiments, I thus used $\gamma = 0.99$.

Neural Network architecture the architecture of the neural network that I chose is a simple neural network with a single hidden layer with 128 neurons and a Tanh activation function.

Optimizer and learning rate I will use Adam optimizer to train the network. Choosing the initial learning rate is important. A learning rate too high will cause the algorithm to not converge, but a learning rate too low will cause it to converge very slowly. I will test learning rates of 0.01, 0.005, and 0.001

Batch size The batch size is also important. A batch size too small will cause the backpropagation to take steps in the wrong direction due to the high variance of the trajectories. I will test a batch size of 1, 4, 16 and 32.

Variance The NN will output the mean of the Normal distribution we will use to sample the action, but the variance of this distribution have to be defined first. I will try a value of 0.5 and 1 for the variance.

2.2 Training Methodology

To compare the different sets of parameters, the algorithm will be trained on 10000 trajectories. I use a number of trajectories instead of a number of episode to be able to compare the convergence speed for different batch size. The number of episodes is thus $\frac{\text{number of trajectories}}{\text{batch size}}$. An trajectory will be stopped when the pendulum reach 200 time steps before dying (to not generate infinite trajectories), or when the environment return that the pendulum is "dead".

To assess the performance of a set of parameters, the main indicators I will look at is the average return for all the trajectories, as well as the oscillations present in the average return for each episode of the training. By oscillations I don't mean the small oscillations in average return from episode to episode that will always happens, but rather when the average returns decreases for a long period of time. To distinguish the two, I will draw a moving average of the return instead of the return itself : this will dampen the small oscillations while keeping the more important one. For example on figure 3 we see that there are clear problems in the convergence). The oscillations are hard to define programmatically, I will thus mainly look at graphs of the evolution of the return per episode to see whether they are present.

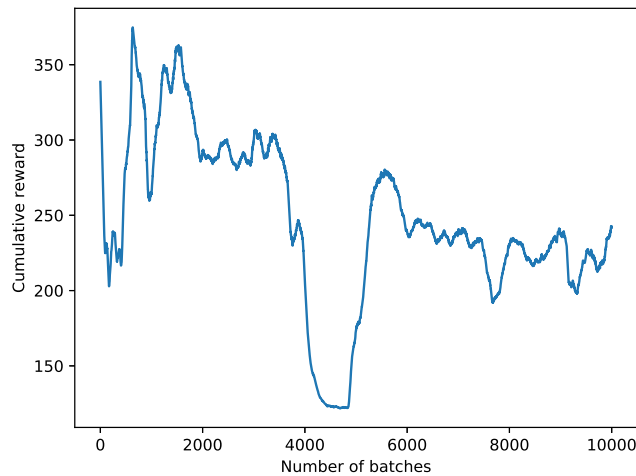


Figure 3: Cumulative reward for $\sigma = 1$, $\text{batch_size} = 1$, $\alpha = 0.01$; We clearly see that these hyper parameters aren't working.

I encountered a problem to find the best parameters. Policy Gradients suffer the drawback of having an high variance, thus to find the best parameters it would be needed to train several times the algorithm over the same parameters and averaging of the results to compare to other set of parameters. However, due to my limited computing power, it would be very hard to achieve this. Training all the set of parameters once already took me close to a full day. To have somewhat meaningful results, I thus averaged how a certain parameter performed with all the combinations of the other parameters. For example, the performance of $\sigma = 1$ is the average performance of all the models that have $\sigma = 1$. For the final model, I chose all the parameters that performed best. This is not optimal because it doesn't capture the dependencies between the parameters (for example a higher batch size could allow for a higher learning rate because of the lower variance). Nevertheless, it gives me a decent approximation of the

best parameters.

The best parameters found with this methodology are :

- $\sigma = 1$
- learning rate = 0.001. Higher learning rate led to way more oscillations than low learning rates.
- batch size = 32. The results were always better for higher batch size. It may have been worth it to test even greater batch sizes.

2.3 Final Results

I then trained the model with the selected parameters. I stopped the training and considered the environment as solved when the pendulum survive more than 150 steps for more than 128 consecutive trajectories. I chose an arbitrary value on the number of steps alive rather than the cumulative reward because it's easier to select a decent threshold beforehand for them. It's important to ensure that a lot of consecutive trajectories are "solved", to be sure that the algorithm didn't only learn a subset of the environment.

The results are on figure 4. We see that the cumulative reward increase almost monotonously. We reach on average trajectories of length 150 around the 1300 batches mark, but it takes still a lot of time to have the environment "solved" for all initial states and trajectories, which happens around the 2200 batch mark.

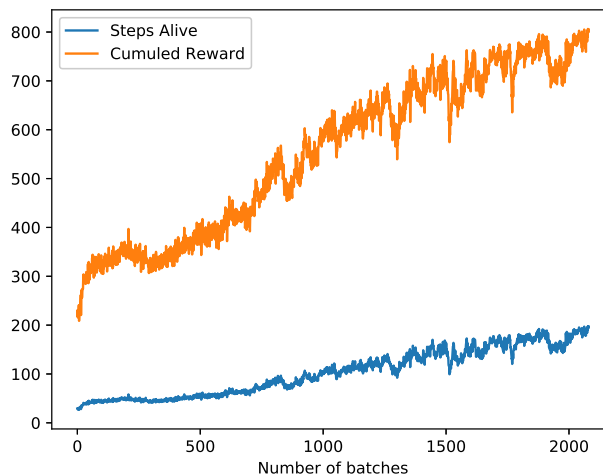


Figure 4: Training for continuous action space

3 Discrete Action Space

For discrete action space, I will also use a policy gradient method with the REINFORCE algorithm. But this time, instead of having the NN output the mean of a Normal distribution, it will end with a softmax layer that will output the probabilities for each of the possible actions. The policy that decide which action to take will then sample an action, with the probability of each action being the result of the softmax layer. The rest of the algorithm is exactly the same as the continuous case.

The code is in the file `discrete.py`.

3.1 Training Parameters

There are several parameters that will influence the quality and speed of the training:

Decay factor The decay factor γ wasn't specified at the time I did these experiments, I thus used $\gamma = 0.99$.

Neural Network architecture I used the same as for the continuous case, only with as much output as possible actions, and a softmax layer.

Optimizer and learning rate I also used Adam optimizer and tested the same learning rate values.

Batch size I tested the same values than for the continuous case (1, 4, 16, 32).

Action space discretization I tested 2 different discretization : only the extrema (-1, 0, 1), or more possible actions (-1, -0.5, 0, 0.5, 1).

3.2 Training Methodology

I used the same methodology as the continuous case with the same drawbacks. The best parameters found were :

- Action space -1, 0, 1. It's however worth noting that more possible actions make it harder for the network to learn all the probabilities of each action. It thus seems normal that less actions performs better on the short term. But for the long term, it is possible that more precise actions to control the pendulum would be better, assuming that the representation capacity of the network is enough to learn correctly each action probabilities.
- learning rate = 0.001.
- batch size = 16. Strangely here, a batch size of 16 performed better than 32.

3.3 Final Results

The results are on figure 5. Since we use almost the same algorithm than for the continuous action space case, the comparison is easy to make by looking only at the results. The results are very similar to the continuous action space, but it took us a bit longer to reach the end of the training (around 2500 batch instead of 2200). It may be that the continuous action space performs slightly better, or a simple "luck" problem, due to the variance of the policy gradients methods.

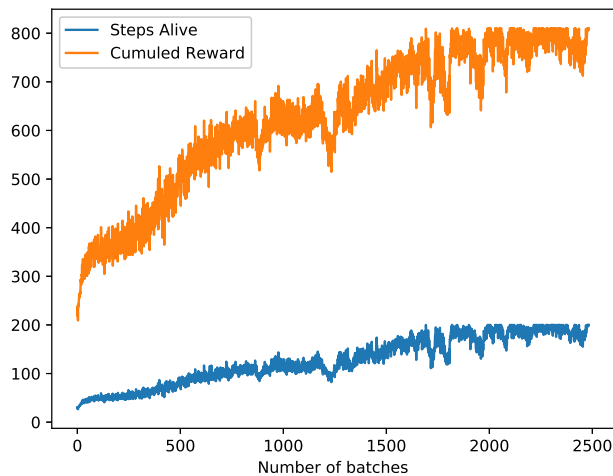


Figure 5: Training for discrete action space

4 FQI

The last step of the project was to compare the previously discussed algorithm with the FQI algorithm with an ensemble of trees. Unfortunately, I wasn't able to make the algorithm work on the double inverted pendulum problem. The expected cumulative reward of the optimal policy obtained with FQI stay stagnant across iterations of the algorithm. I believe there could be two causes for the problem :

- **The complexity and dimensionality of the problem.** The double inverted pendulum is a chaotic system. Small perturbations could lead to a very different result. It has also a somewhat high dimensionality (9 dimensions), it is thus difficult to learn and should require a lot of iterations of FQI to have good results. Also, learning to solve this environment with an ensemble of trees requires a lot of trees due to its dimensionality and complexity. But the more trees, the longer it takes to fit the data. The number of iterations needed, along with the number of trees needed, and the fact that for unknown reasons, the "ExtraTreesRegressor" from sklearn only works if single threaded on my machine (no multi-thread is possible), makes it very hard to train the algorithm, it simply takes too long.
- **The dataset.** In addition to the previous problem, generating the dataset to train the model also proves difficult. The problem is that (to my knowledge), there isn't a way to initialize the pybulletgym environment in a chosen initial state. It always starts the environment with the cart in the middle and the pendulum slightly moved from the vertical. That means that if we generate the dataset with a random walk policy like I did for the assignment 2, a lot of states will not be visited in the dataset. And if the states aren't in the training set, the algorithm won't be able to accurately learn the correct actions for these states, even more so due to the complexity of the double pendulum environment.

The methodology I wanted to use to compare it to the previously discussed algorithms is the following. After each iteration of FQI I generate a set of trajectories from the current policy, and average their cumulative reward. I can then compare the cumulative reward at this iteration of the algorithm along the number of trajectories of training needed to reach this point for both algorithms (the number of episodes times the batch size for the policy gradients methods, and the number of trajectories in the dataset times the number of FQI iteration for the FQI algorithm).

There is however a point that I could find from these incomplete experiments to compare FQI with the policy gradients methods. That is the performance of the algorithm. Fitting a model with a lot of trees multiple times on a big dataset takes a lot of time. The FQI algorithm with an ensemble of trees will thus always be less computationally efficient than policy gradients methods with a shallow network for complex problems.

5 Possible improvements

There are several improvements that I see possible from the current state of my work for the policy gradient methods :

- Using actor critic algorithms for the training of the policy
- From the continuous case, having the network generate the variance of the Normal Law along with the mean
- Trying other activation functions and architectures for the NN to see if they improve the performance