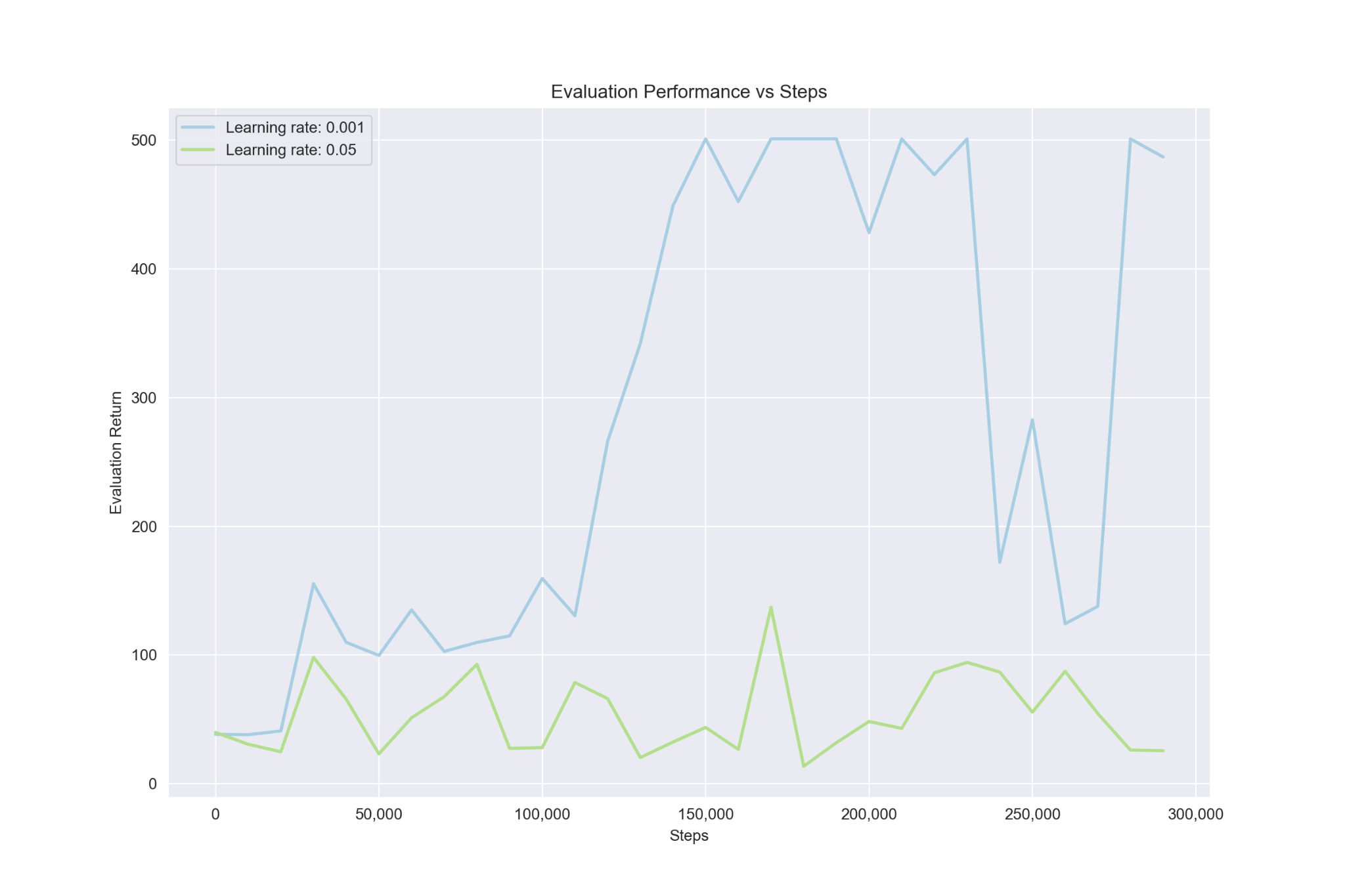
Class project 3

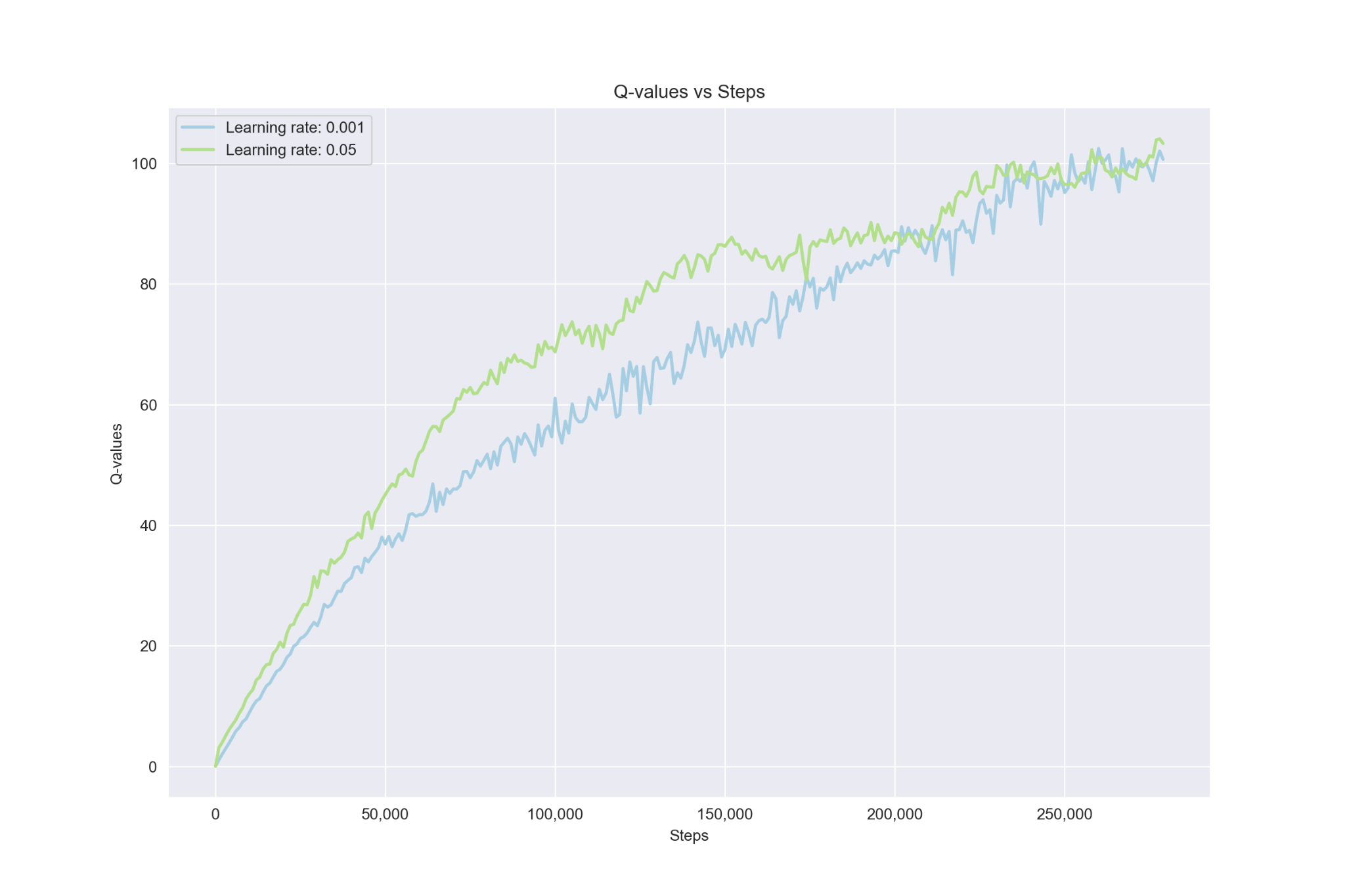
Matouš Bohoněk, 2025/05

# Basic Q-learning

## Cartpole

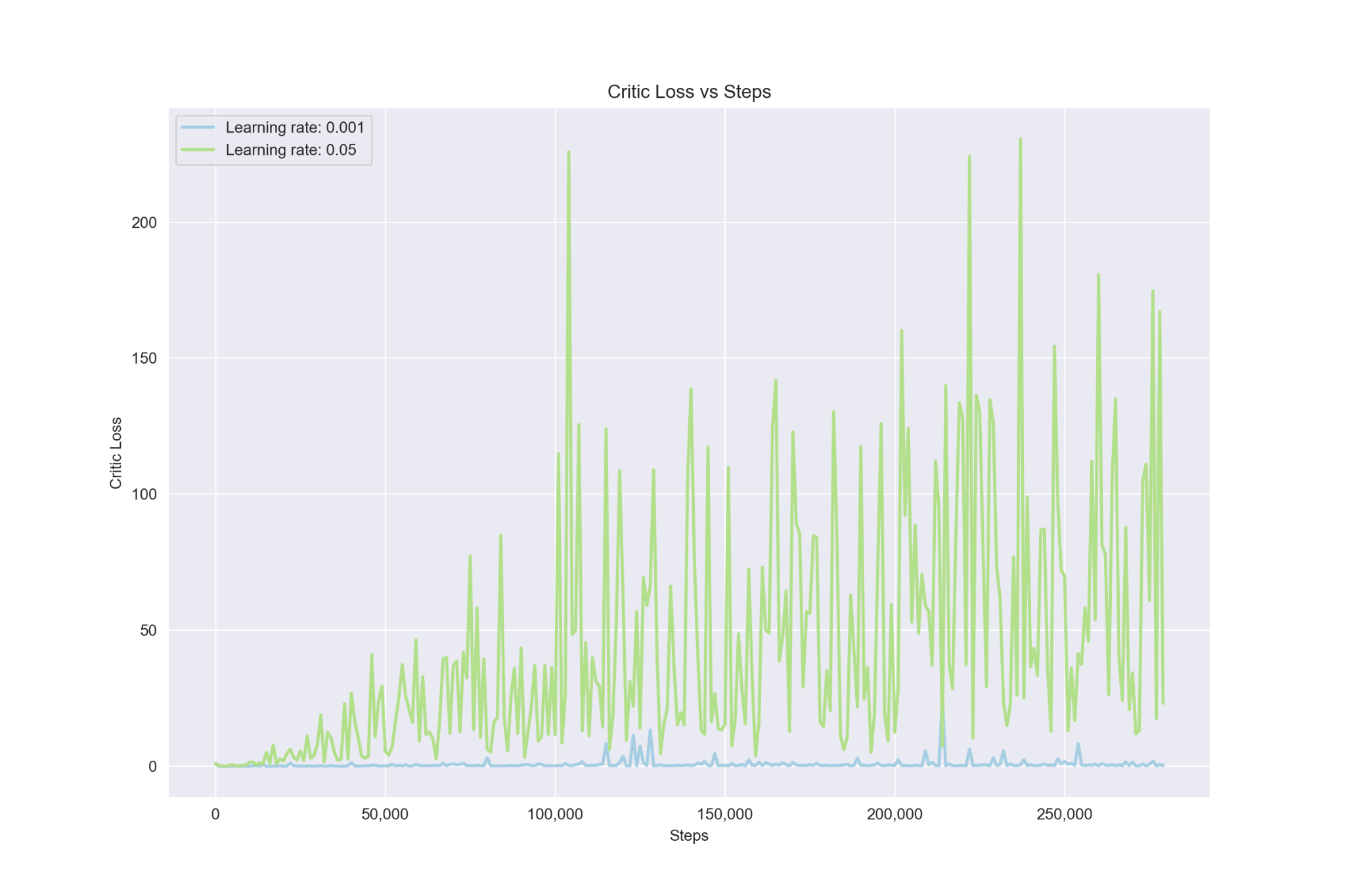


As we can see, the basic Q-learning algorithm was able to reach a return of 500. This indicates our implementation should be working. However when we change the learning rate from 0.001 to 0.05 the evaluation performance drops significantly. What happens to (a) the predicted Q-values and (b) the critic error?

1. The Q-values remain mostly the same, with only a slight difference in the opening stages of training where they go up faster for the higher learning rate 

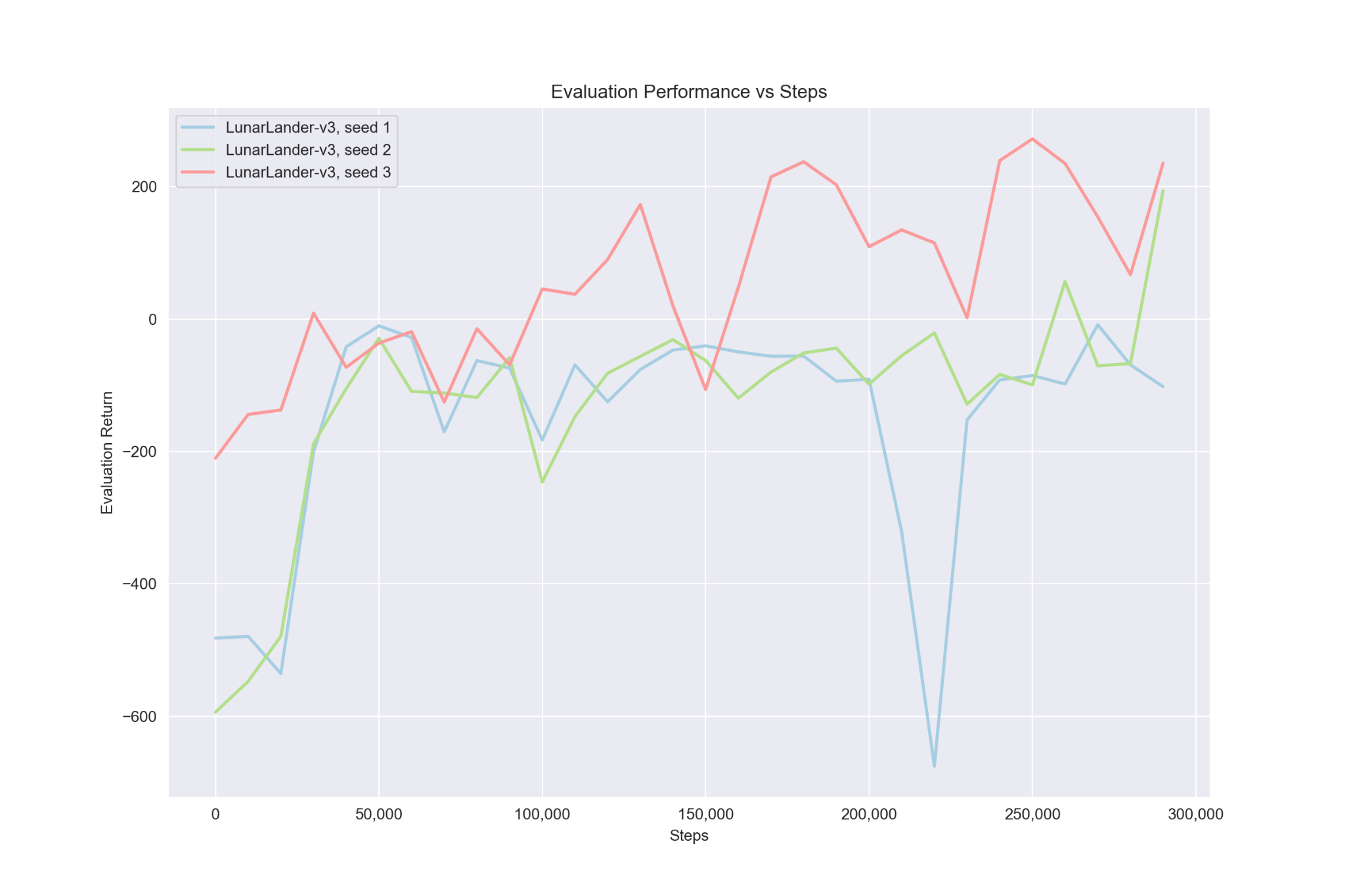
This clearly indicates that higher learning rate is overestimating its expected return as the real return it can achieve is about 5x lower that that of the lower learning rate.

1. The critic error is much more volatile and also much bigger for the higher learning rate.



## LunarLander

As expected, we didn’t achieve a great return in most runs. In one run, however, we got lucky and achieved a score of over 200.



# Double Q-Learning

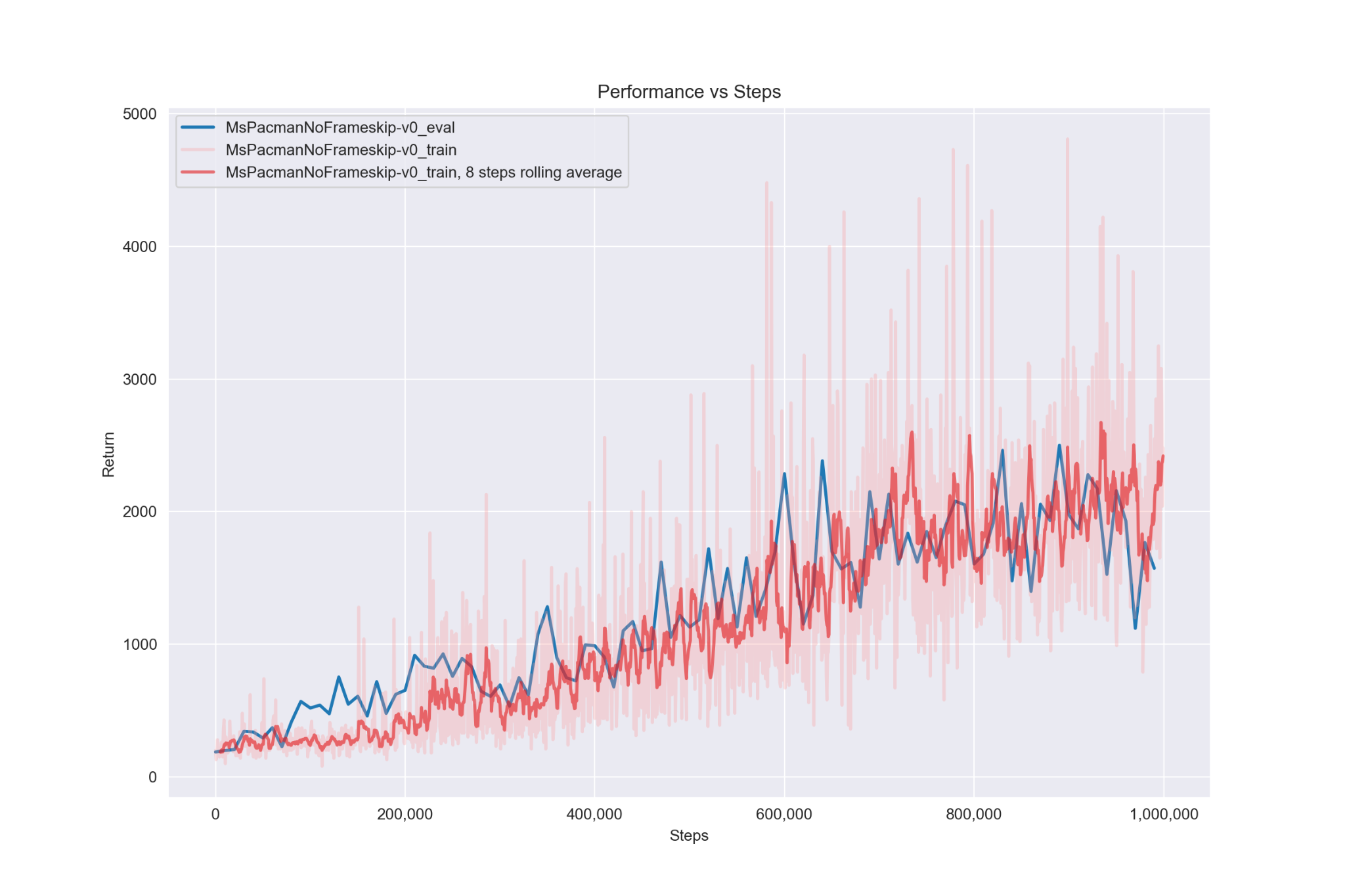
## LunarLander



We can see that double DQN didn’t perform significantly better than the vanilla version. One reason might be that we got lucky with one really good vanilla run that scored over 200, another reason might be that this problem just isn’t complicated enough for double Q-learning to be so beneficial.

## MsPacman

The training of this agent took way longer than expected. The assignment estimated around 6 hours on a CPU but it took more than double of that. Because of compatibility issues I had to change some parts of the original code which might’ve resulted in a poorer performance. The results were however good, achieving a return of over 2000:



We can notice that in the first 200,000 steps the evaluation return is quite higher than the training return. This is likely due to the epsilon in training being much higher at the early stages than in evaluation which means that the training policy is quite close to a random policy while the evaluation policy uses at least some of the learned behavior. In training epsilon starts at 1 and over the first 200,000 steps linearly decreases to about 0.62 while the epsilon in evaluation is set to 0.02.

# Experimenting with Hyperparameters (LunarLander)

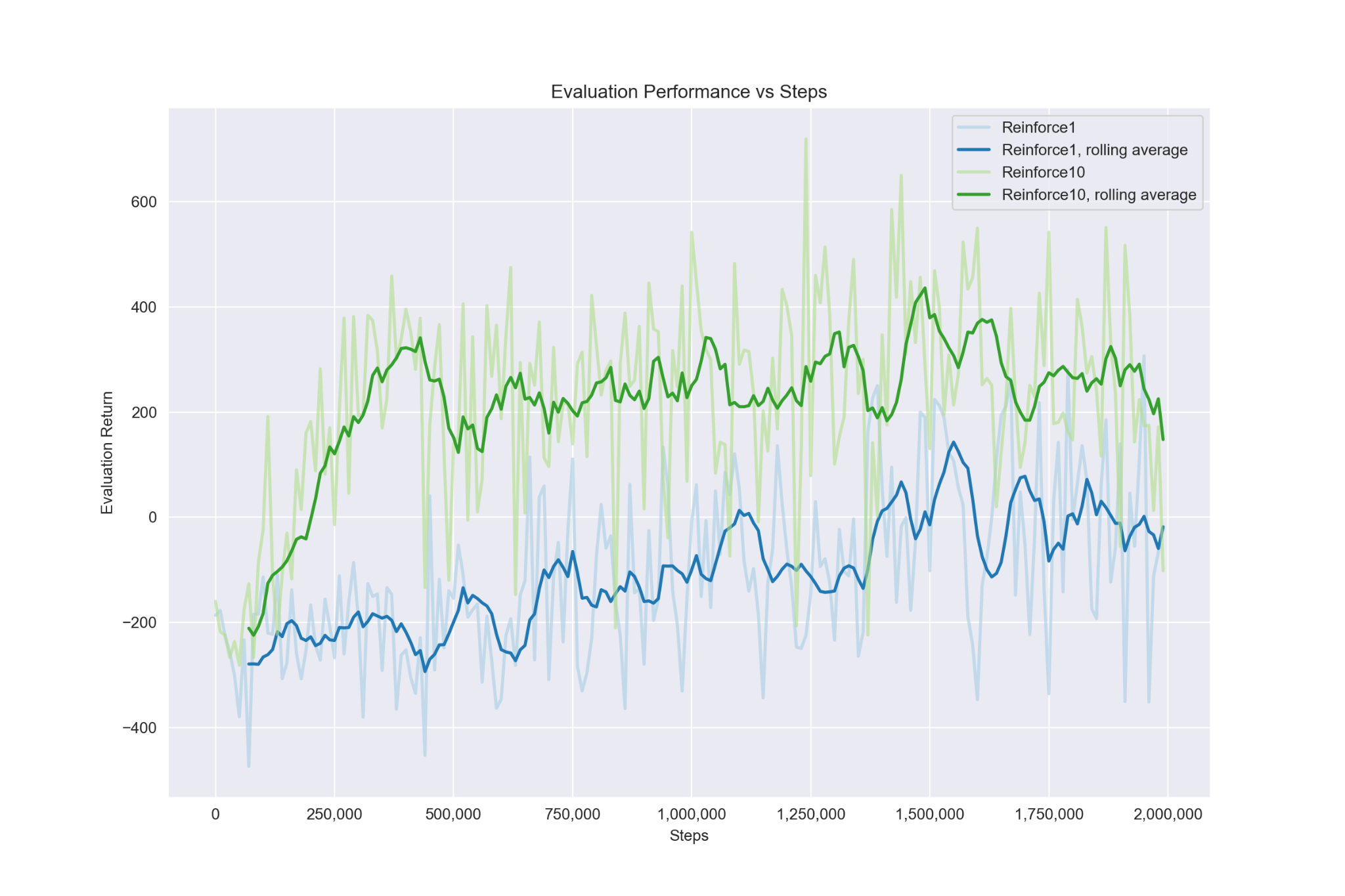
For this experiment I chose to tune the hidden\_size parameter. I wanted to see whether the lunar lander could benefit from a bigger network or if using a smaller network can still give the same results. I chose the sizes 32, 64 (default), 96 and 164 to try both a small and big increase in size.



The results are not that interesting. We can see that increasing the network size did somewhat improve performance, the biggest network might’ve just needed more training steps to reach its full potential (as can be the case with bigger neural networks), but overall all network sizes can reach similar levels of performance in this limited number of steps.

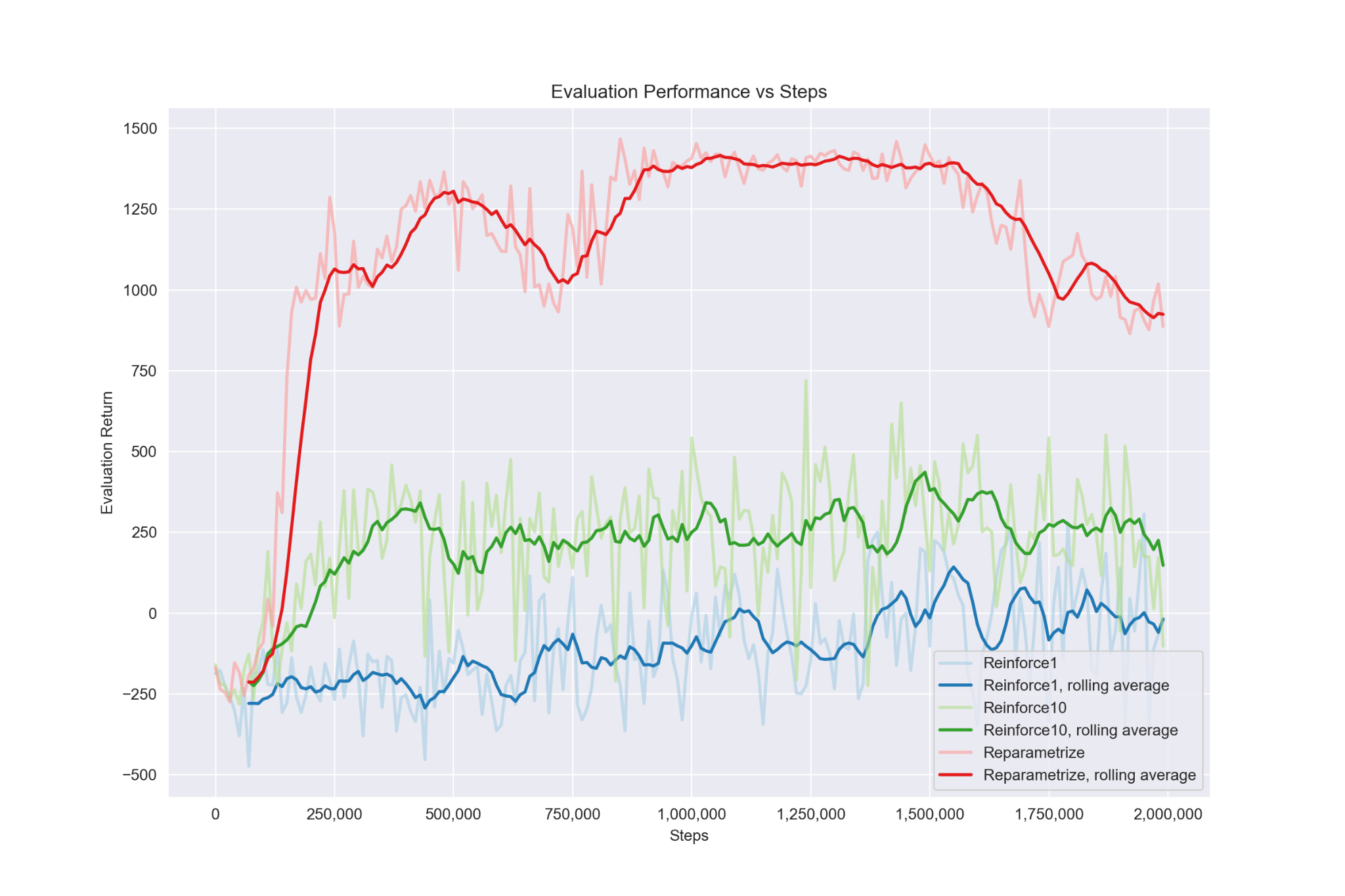
# Continuous Actions with Actor-Critic

## Actor with REINFORCE (HalfCheetah-v4)



We can see that there is a clear advantage to drawing more samples when calculating the actor loss. The reason is that drawing more samples reduces variance.

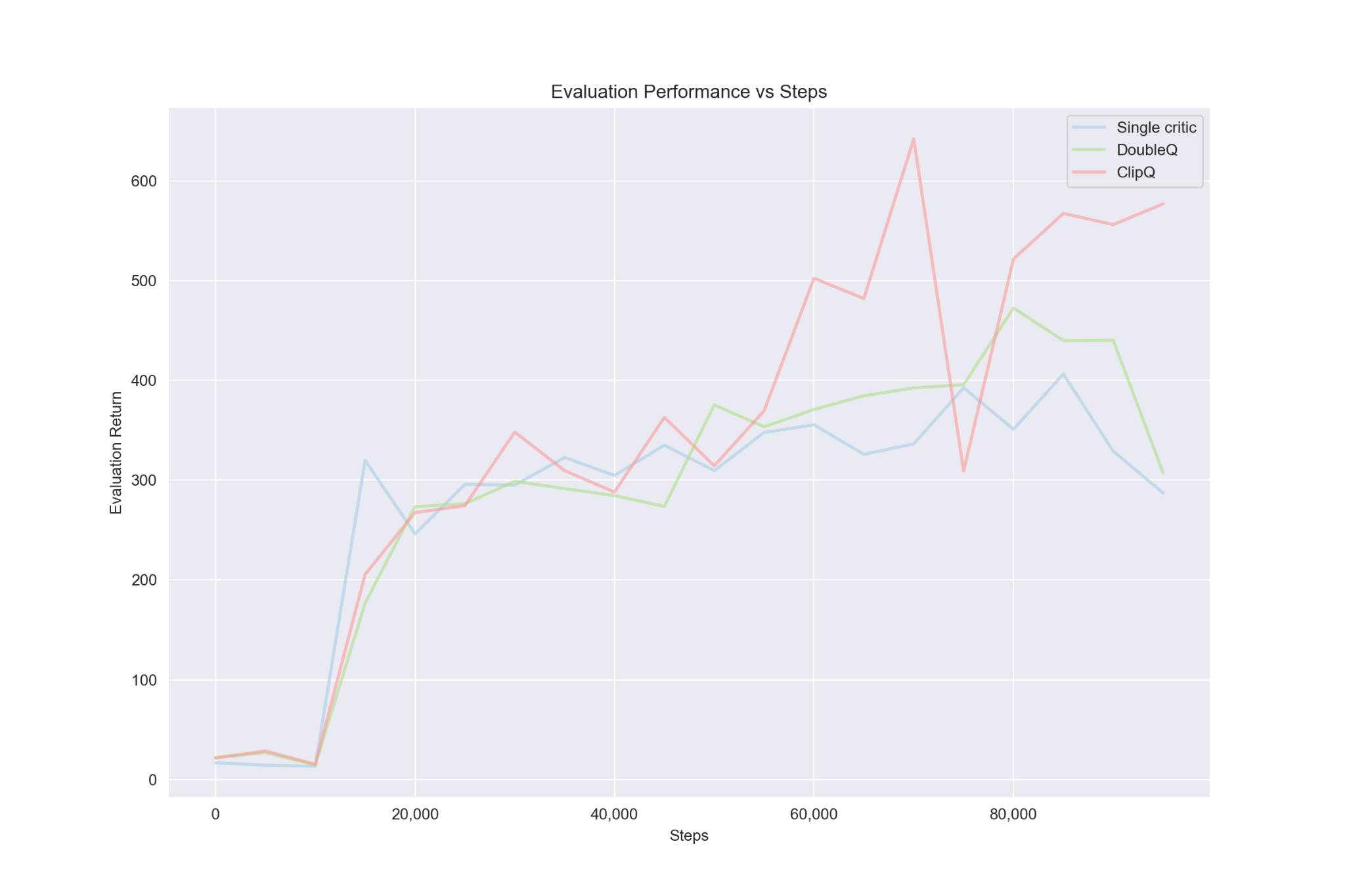
## Actor with REPARAMETRIZE (HalfCheetah-v4)

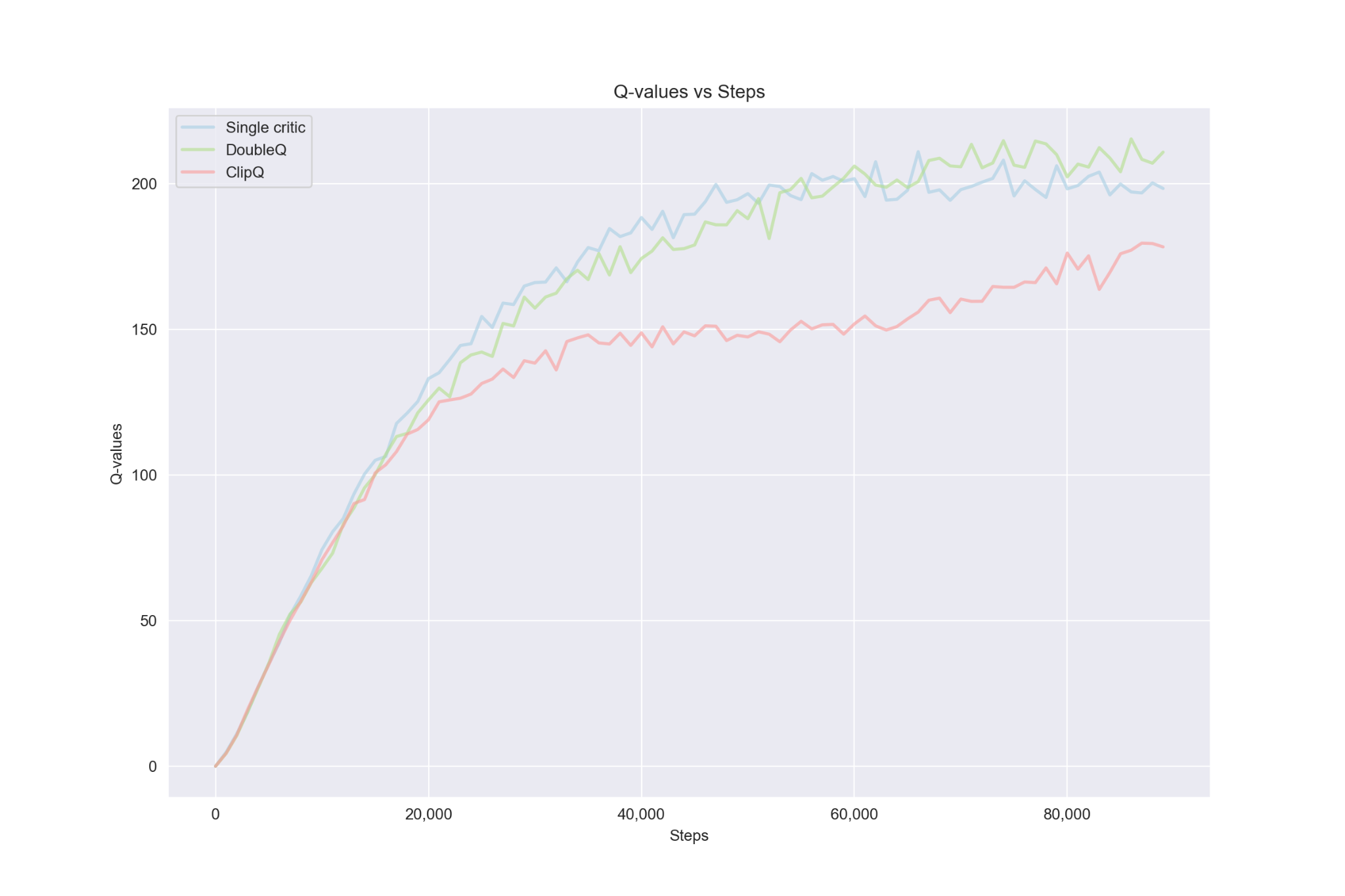


Adding reparametrization helped significantly, improving the return almost 3x compared to the REINFORCE10. This shows how much better reparametrization can be.

**There was no humanoid\_sac.yaml file, training the humanoid was done later at the end of this report.**

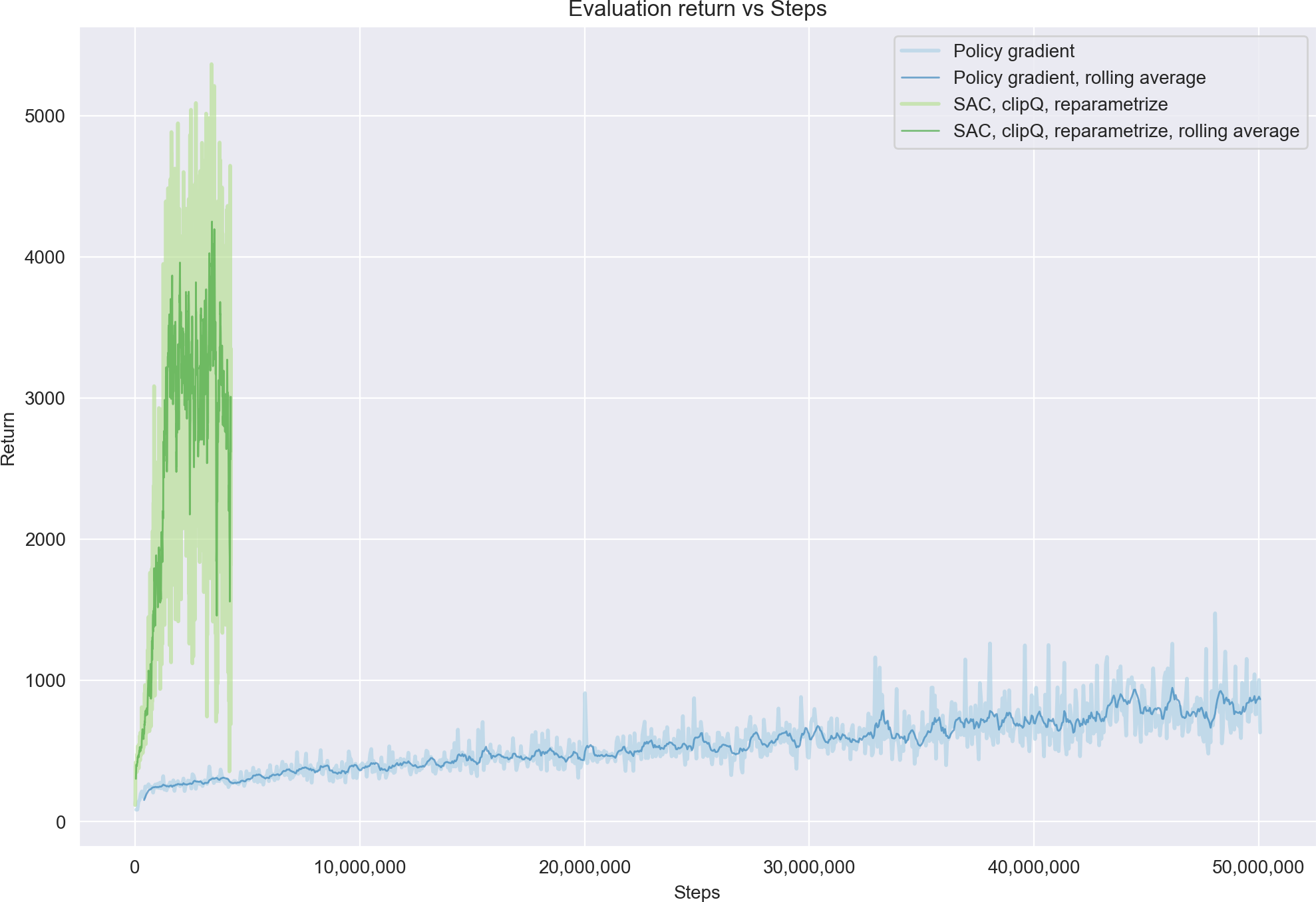
## Stabilizing Target Values



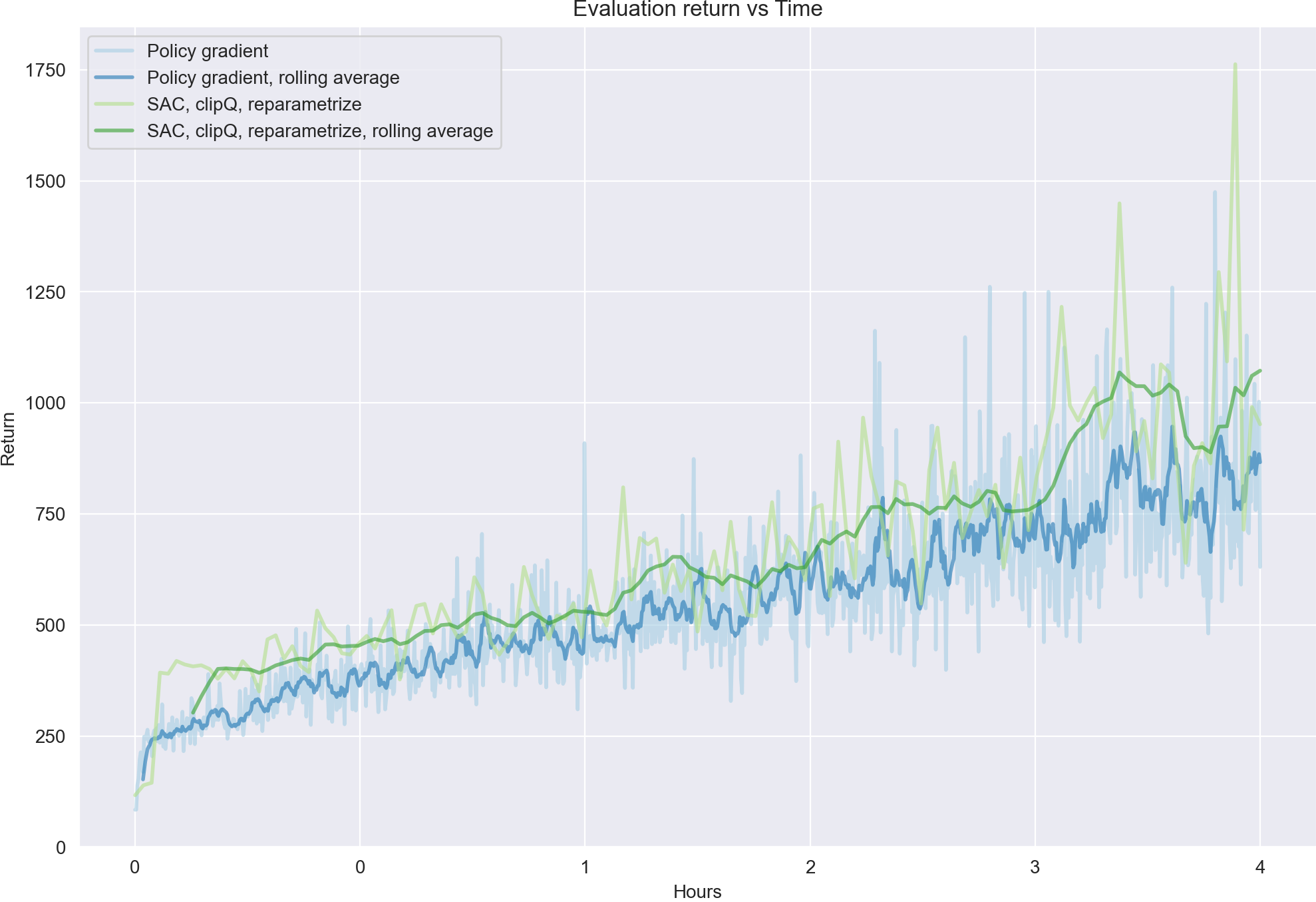


From this experiment we can see that clipping achieved much better results compared to single critic or DoubleQ. When looking at the Q-values we can also see that the clipped version has the lowest, which suggests that it is overestimating much less than the other two which in turn gives a better policy.

## Humanoid



We can see that the SAC is much better in terms of sample efficiency and even overall performance compared to the policy gradient. The performance dropped off after about 70% of the training steps for some reason. The training took much longer at 25 hours which is almost 6x more than the training time of policy gradient (4 hours). But even if we consider the SAC performance after 4 only hours we can see that it is still a bit better than the policy gradient:



# Discussion

This project took me the longest to finish. The main reason is the training time of each experiment. Another reason is package versioning problems. Even though we are provided with exact version numbers for each package I was unable to install some key packages in their required version. Using a newer version resulted in some changes in behavior (for example some functions return different things, some environments are no longer supported and required me to use a newer version, etc…). The main change is the migration from Gym to Gymnasium. For future runs I think it would be a good idea to update the assignments to use the newest versions of packages such as pytorch, Gymnasium,...