Journal of Machine Learning Research (2022) Submitted 12/12; Published N/A

Reinforcement Learning for Stable Planning

Eli-Henry Dykhne EHDYKHNE@UWATERLOO.CA
Department of Computer Science
University of Waterloo
Waterloo, ON, Canada

Editor: Eli-Henry Dykhne

Abstract

This work explores the yet untouched problem of stable planning in which an agent is expected to form plans that can be stuck to in spite of unexpected developments for situations where changes to the plan carry a penalty. Here, I test the ability of several existing reinforcement learning algorithms to handle this new problem in modified versions of traditional reinforcement learning environments. I explore the challenges this problem poses, and demonstrate that existing algorithms can handle this new type of problem.

Keywords: Stable Planning, Reinforcement Learning

1. Introduction

Environments where future actions require costly setup are common in the real world. Rapid changing of future plans can require the reconfiguration of systems, redrafting of reports, and wasting of earlier preparations. For example, a cook may have actions that involve preparing various ingredients before adding them to their dish. If partway through the cooking process, the cook were to decide that an already prepared ingredient was to be replaced with another, the cook would incur the cost of having to quickly prepare the new ingredient, as well as possibly having to discard other prepared ingredients that would no longer work with the change in the recipe. This heretofore unexplored area of problems has the potential to be relevant to any organization where actions require some form of preparation, and where quick changes to plans can only be performed at great expense, like asking someone to work overtime to enact a last-minute change. In deterministic environments without an opponent, this should be possible to achieve by being able to look directly into the future. In stochastic environments, for a stable plan to exist, the environment must be at least partially predictable for the agent to be able to select a plan that is robust to those changes. This work tests the performance of several existing model-free reinforcement learning algorithms on this new class of problem.

2. Related Work

This area of research has not directly been explored, though the expected output bears resemblance to Model Predictive Control, and various trajectory planning algorithms which are expected to produce an entire trajectory rather than just the next action. One work (Schuitema et al., 2010) explores the idea of training agents with time-delayed actions in

which they explore changing the state space by including the previously chosen actions which are yet to be executed. This requires the agent to predict how the environment will change over time to predict the next action but differs from our problem in that these actions have already been committed to. Perhaps the closest work to this is the paper titled "Strategic Attentive Writer for Learning Macro-Actions" (Vezhnevets et al., 2016), which aims to have an agent learn strings of actions that make up macro-actions which can more efficiently be used for learning. As a result, their network is expected to build an internal plan which is continuously updated with observations from the environment. Unlike this work, these actions are not added to the environment state, and our agent is not expected to commit to its plan and use macro-actions, though it is strongly encouraged to. Apart from this, I am unable to find any other works that have explored this

3. Method

To create the environments where the agent would have to plan ahead, we modify common gym environments. These include Cartpole, LunarLander, several MuJoCo (Todorov et al., 2012) environments, and a handful of continuous control environments which we do not run tests on as a result of time constraints. To test existing algorithms, we make use of the StableBaselines3 repository, which provides a set of reinforcement learning algorithms that can handle "MultiDiscrete" action spaces, such that multiple actions can be taken at once. This simplifies the implementation for discrete action environments. Adjustments to the reward given by each environment are scaled by a $costMultiplier$ in order to compensate for each environment's unique reward scaling. Improper scaling can lead to penalties that are too small and simply end up being ignored as noise, or too large, which dominates the reward received at each time step and encourage the agent to try and terminate itself. In environments where this is not possible such as Reacher, this concern is alleviated. We do not claim to have found the perfect reward scaling for each environment. The reward scaling for each environment used can be found in the included GitHub repository[1]. All tests are run 5 times to get some semblance of the variance of each algorithm in each environment.

3.1 Discrete Action Environments

To modify discrete action environments to be able to accept multiple actions, their action spaces are converted from "Discrete" to "MultiDiscrete". Only the first action is taken from the whole set of actions that make up the plan while the rest are saved as $oldActionList$ to be used for reward calculation. The adjustment to the reward is calculated as shown in Algorithm 1. The old plan is compared to the current plan, with the cost being determined by the earliest deviation. To avoid a cost or "punishment", an agent would have to stick to the old plan and predict what action will have to be taken a certain amount of steps in the future. For the majority of the tests, the environments were changed such that agents would have to make a plan for three steps rather than just one. Since the previous actions are now part of the environment, they are added to the observation space, which turns it into a multi-input space. Only a handful of algorithms are capable of dealing with

1. https://github.com/HenryDykhne/StablePlanner

"MultiDiscrete" action spaces in the StableBaselines3 repository, which limited us to testing with PPO, A2C, and TRPO for discrete environments.

Algorithm 1 Calculate Adjustment to Reward for Discrete Action Spaces

```
stepsPlannedFor ← length(oldActionList)
oldActionListWithoutFirst ← oldActionList[1 :]
for i in range(stepsPlannedFor - 1) do
  if actionList[i] ≠ oldActionListWithoutFirst[i] then
    adjustment ← −costMultiplier × (1 − (i ÷ stepsPlannedFor))
    return adjustment
  end if
end for
```

3.2 Continuous Action Environments

Modification of continuous action environments is significantly different. The prior plan is still added to the observations, however, since the action space is often multidimensional to begin with, (to allow control of multiple limbs at once in the MuJoCo (Todorov et al., 2012) environments) the dimension of the action space is multiplied by the number of steps the agent is expected to plan for. During reward adjustment calculation, the dimensions must be taken in blocks of the size of the original action. Since continuous actions are unlikely to be exactly equal, for the reward adjustment, the euclidean distance between actions is measured. This distance is scaled inversely to how far the action is in the future in the plan, once again penalizing agents more heavily for modifying actions that are closer to being executed. It is understood that as the number of dimensions increases, the distances in a multidimensional space also increase. The *costMultiplier* variable also helps to adjust for this. The full procedure is described in Algorithm 2. Due to time constraints, tests are only conducted with PPO, A2C, and TRPO, although such continuous action environments could reasonably be tested with any of the algorithms set up for continuous action spaces, unlike the discrete environments.

Algorithm 2 Calculate Adjustment to Reward for Continuous Action Spaces

```
stepsPlannedFor ← length(oldActionList)
oldActionListWithoutFirst ← oldActionList[1 :]
adjustment ← 0
for i in range(stepsPlannedFor - 1) do
  distance ← euclideanDistance(actionList[i], oldActionListWithoutFirst[i])
  adjustment ← adjustment − costMultiplier × distance × (1 − (i ÷ stepsPlannedFor))
end for
return adjustment
```

4. Results

A fair review of the results can only occur in the context of our success criteria. A technique will be considered successful in its environment only if it can minimize the cost of changing plans (by not changing plans often), and demonstrate proficiency in the environment in

spite of the stable planning requirement by achieving comparable scores to the agents that do not receive penalties for changing plans. In environments where success correlates with longevity such as the Cartpole or Double Inverted Pendulum environment, it is understood that the total cost across an episode may increase over time simply because the agent lives long enough to rack up that penalty. In such environments, the cost per time-step will be considered the more indicative measure for determining if the agent has truly learned to create stable trajectories to solve the environment. All referenced figures are contained within the appendix.

4.1 Discrete Action Environment Results

In simple, discrete action environments such as Cartpole and LunarLander, the agent is able to overcome the challenge of predicting the future and making stable plans. In both cases, PPO prove most effective, with TRPO close behind as shown in Figures A1 and A3. A2C is least capable and shows that it cannot maintain a stable future plan. When asked to plan 5 steps ahead, all tested algorithms have more difficulty with the Cartpole environment as shown in Figures A2 and A4, though PPO and TRPO still manage to master it. In contrast, no tested algorithm is able to master the LunarLander environment under this planning burden. PPO is still able to mitigate the penalties from re-planning to some extent. With a larger policy network and more training time, it is likely that PPO and TRPO would have both been able to achieve optimal performance on LunarLander under this planning load.

To test on a nondeterministic environment, the LunarLander environment was once again used, this time with the wind feature enabled. This feature applies a random wind force to the landing craft at every step. The hope was that the agent would be able to find a robust plan that would be able to handle future perturbations and thus avoid the re-planning penalties. As seen in Figure A5, the agents were able to cope with this added challenge, and perform as well as the normal PPO agent that is not expected to plan ahead.

4.2 Continuous Action Environment Results

The continuous action environments that were tested include Hopper, InvertedDoublePendulum, Reacher and HalfCheetah. They are a more difficult challenge for the agents. TRPO proved most robust in the modified environments and is able to keep up with algorithms in the environments that were not penalized. Conversely, A2C shows a complete inability to cope with the additional task as can be seen in Figures A6 to A9. The way the penalty is calculated allows agents that cannot figure out how to look ahead to minimize their losses to some extent by picking the same action for all future steps, however, upon visual inspection of the actions being selected, it was determined that this was not a strategy any of the agents had settled on.

4.3 Exploration of Resultant Architecture

During testing, it became apparent that in the LunarLander environment, the agent being penalized was outperforming the baseline agent which was not asked to plan ahead. To

explore this further, planning agents run in the same environment without penalization were tested and found to do similarly well. Agents were tasked with performing in environments where they had to predict 3, 4 and 5 steps ahead. It was found that while the penalized agents asked to predict 4 and 5 steps ahead were unable to do better than the baseline even when the penalties were ignored, all other agents were able to consistently outperform the baseline non-penalized PPO agent as can be seen in Figure A10. Since in the non-penalized versions, the agent is effectively free to send whatever information it wants, through the choosing of its actions, it is effectively acting like a small RNN, though testing with RNN variations of PPO fails to produce the same effect. To be sure that the advantage does not come from being able to use its most recent move as a heuristic, a test is run where the agent is only asked to plan one step ahead. This means that the only difference between this agent and the baseline agent is that this agent is able to see its last action as part of the observation space. This agent does no better than the baseline. Despite testing in other environments we are unable to replicate this advantage. It is possible that the advantage comes from some ability to do longer computations by, choosing an action to do in the next time-step, and then being able to do final corrective computations in the next time step to that action. More study is needed to understand why this does not carry over to other environments.

5. Discussion

In the majority of the tested environments, PPO and TRPO are robust enough to be able to handle the challenge of making stable plans for three steps in all tested environments. Longer planning windows were more difficult to solve, but larger networks and more time to train should allow for any stable planning deterministic environment with discrete actions to be solved. Continuous actions prove slightly more difficult for the algorithms to deal with. Though all agents were able to minimize the cost to some degree, none were capable of completely eliminating it and observations during training suggest that more training time would not have helped. Some small amount of penalty is always preserved, as the agent makes last-minute adjustments to the planned action. In some environments such as HalfCheetah, penalized agents were able to outperform their non-penalized counterparts, if the penalties were removed in the evaluation as can be seen in the "Cost Corrected" figures in the appendix. In stochastic environments which are most important with respect to the outlined problem, the algorithm seems capable of solving the problem as well.

6. Limitations

This work is limited by the time constraints of the author, as well as the computational power of the machine to which they had access. As a result, even though the environment was set up for it, tests were not run for visually complex environments. The Car Racing environment was found to be very slow, such that a representative number of tests would not be able to be run in time. While all models were trained five times to get a representative average, a more accurate analysis would have been possible if more time was allotted to perform a larger number of runs.

7. Future Work

Due to the slow training of visually complex environments, they were largely avoided for this work. Future work could return to them to confirm whether or not these techniques remain effective in these environments. Other paths for future work involve the exploration of constrained reinforcement learning methods as a technique for separately minimizing the frequency of plan switching, and experiments with network architecture to see which are most suitable. Model-based reinforcement learning should also be explored for this problem to see if keeping a separate internal world model also allows the agent to create stable plans more effectively. The reward adjustment structures and cost multipliers used should be iterated on and explored to find optimal values and better formulations that are more conducive to learning or more representative of real-life situations. For example, changing continuous actions to be scored like the discrete actions but rather than checking for perfect equality, checking for a certain closeness as a threshold. Finally, a deeper exploration of how to recreate the advantages observed in the LunarLander environment should be more thoroughly explored to see if it can yield a better network architecture for some problems.

8. Conclusion

In summary, the problem of stable planning is one that is able to be handled by algorithms such as PPO and TRPO. Neither algorithm was able to perform perfectly in continuous action environments but were both able to produce future actions that were more stable and required fewer changes as the agent trained more.

References

E. Schuitema, Lucian Busoniu, Robert Babuska, and Pieter Jonker. Control delay in reinforcement learning for real-time dynamic systems: A memoryless approach. pages 3226 – 3231, 11 2010. doi: 10.1109/IROS.2010.5650345.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, Oct 2012. doi: 10.1109/IROS.2012.6386109.

Alexander Vezhnevets, Volodymyr Mnih, John P. Agapiou, Simon Osindero, Alex Graves, Oriol Vinyals, and Koray Kavukcuoglu. Strategic attentive writer for learning macro-actions. *CoRR*, abs/1606.04695, 2016. URL `http://arxiv.org/abs/1606.04695`.

Appendix A. All Relevant Graphs

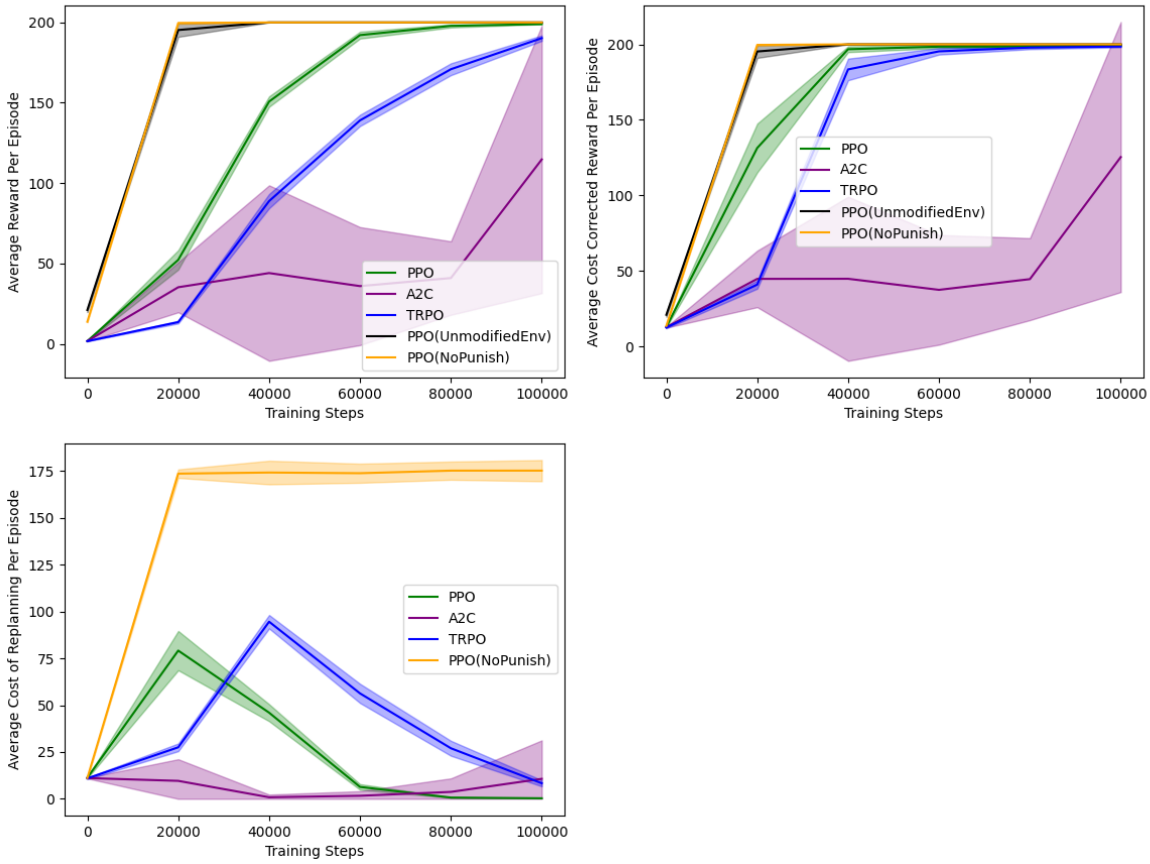

Figure A1: Performance of the PPO, TRPO and A2C algorithm on the modified Cartpole environment where agents are expected to plan 3 steps ahead compared to a PPO agent that is not punished for changing its plans, and an agent which is not expected to plan ahead at all.

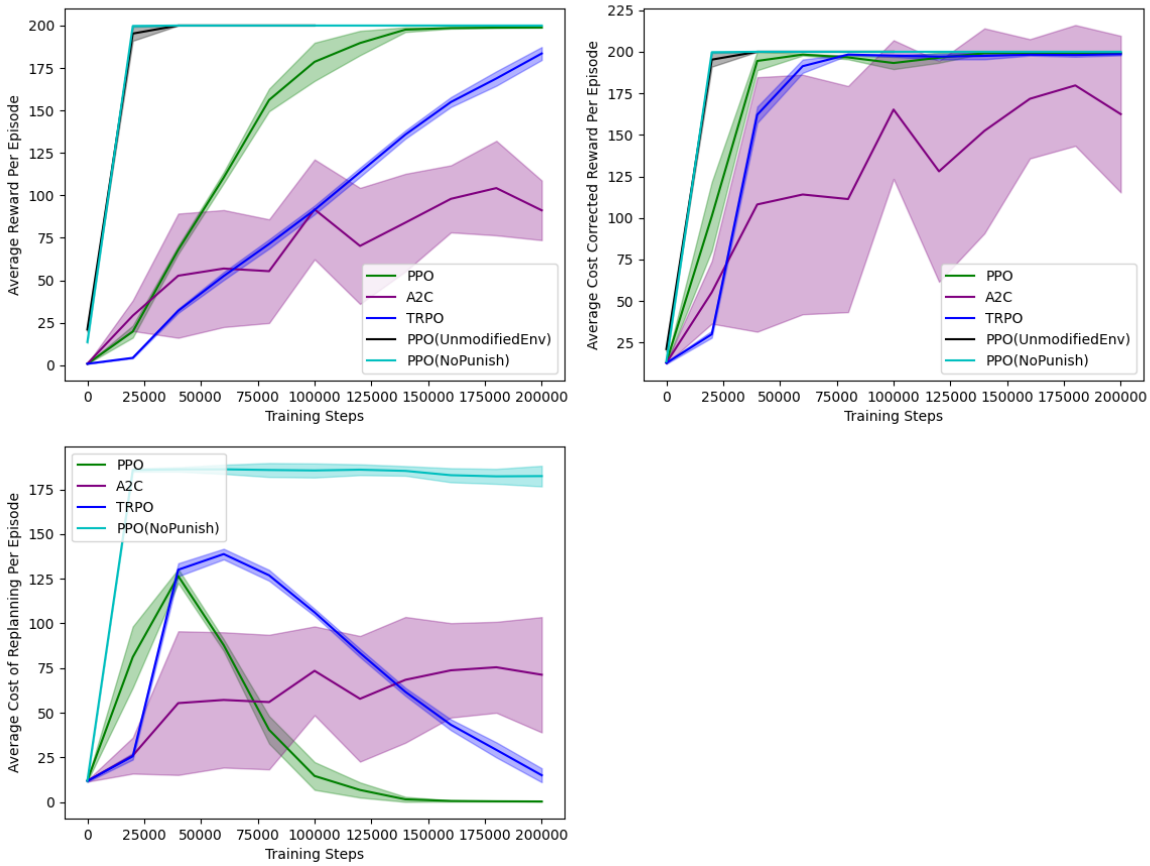

Figure A2: Performance of the PPO, TRPO and A2C algorithm on the modified Cartpole environment where agents are expected to plan 5 steps ahead compared to a PPO agent that is not punished for changing its plans, and an agent which is not expected to plan ahead at all.

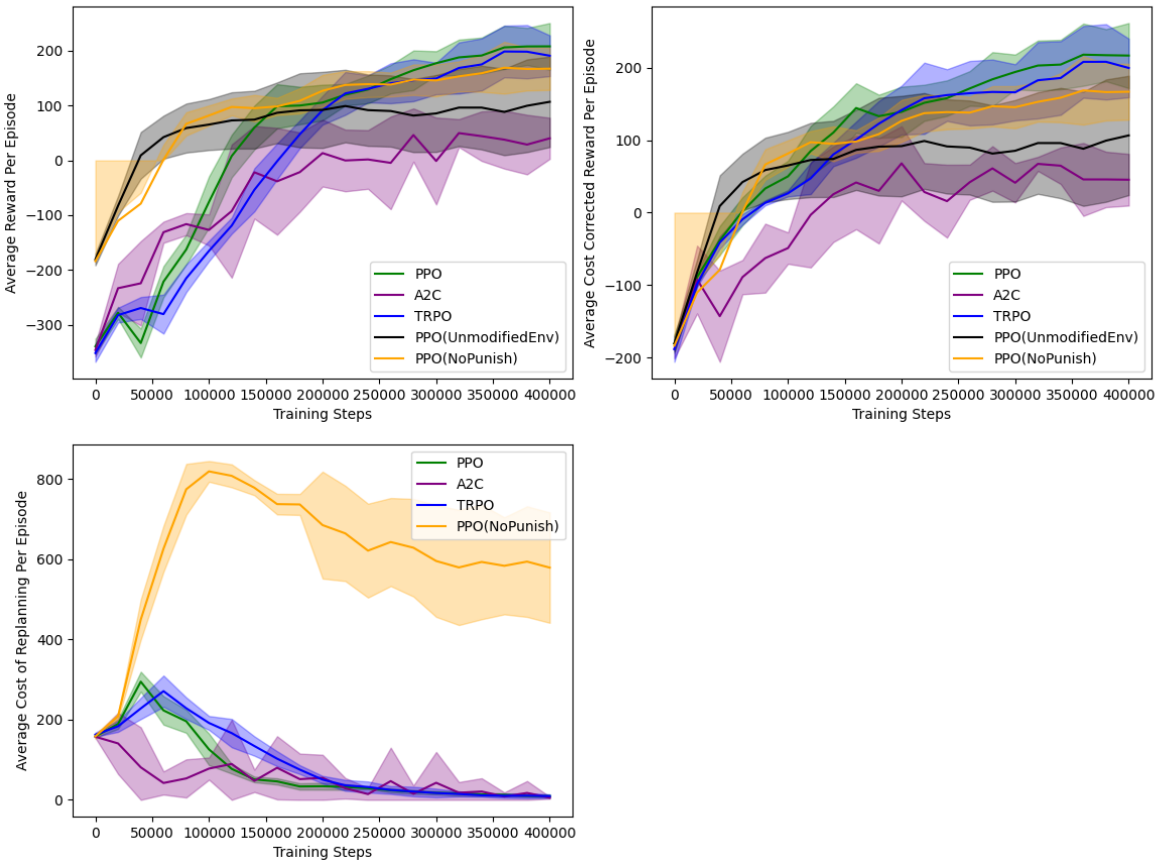

Figure A3: Performance of the PPO, TRPO and A2C algorithm on the modified LunarLander environment where agents are expected to plan 3 steps ahead compared to a PPO agent that is not punished for changing its plans, and an agent which is not expected to plan ahead at all.

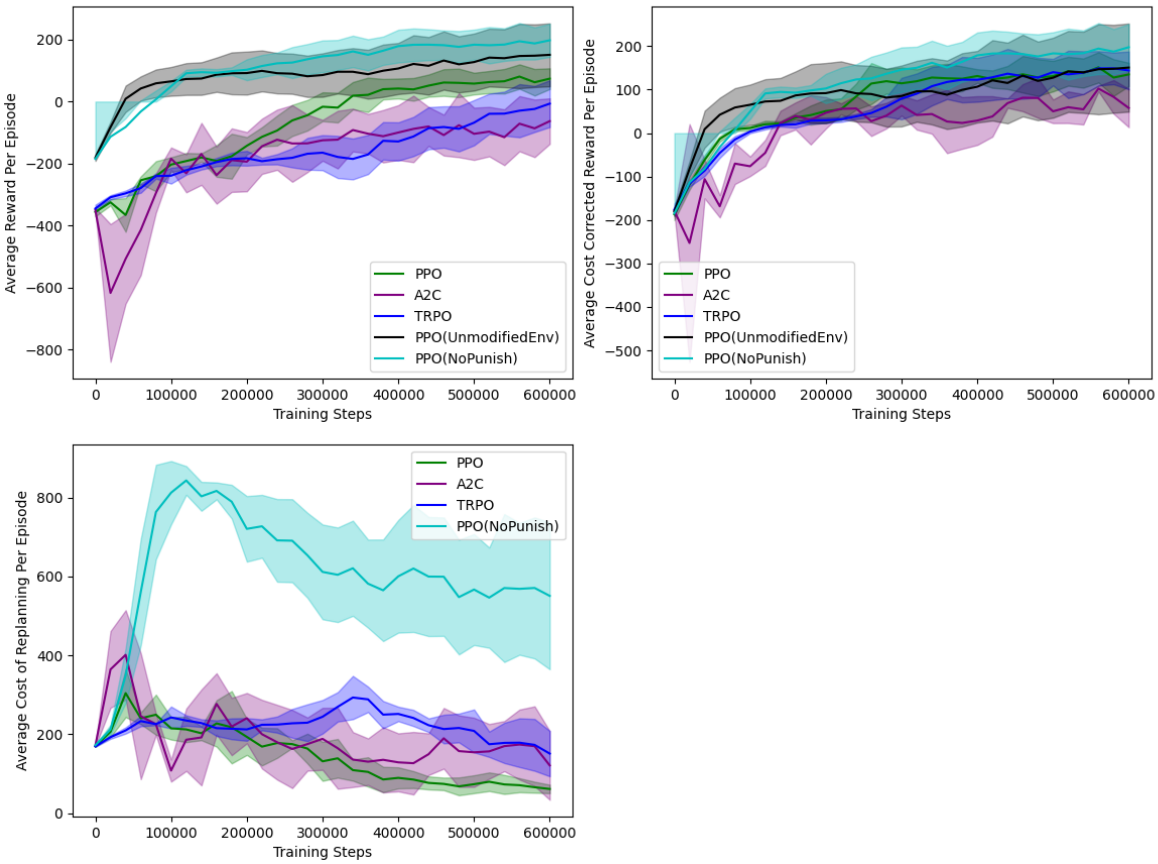

Figure A4: Performance of the PPO, TRPO and A2C algorithm on the modified LunarLander environment where agents are expected to plan 5 steps ahead compared to a PPO agent that is not punished for changing its plans, and an agent which is not expected to plan ahead at all.

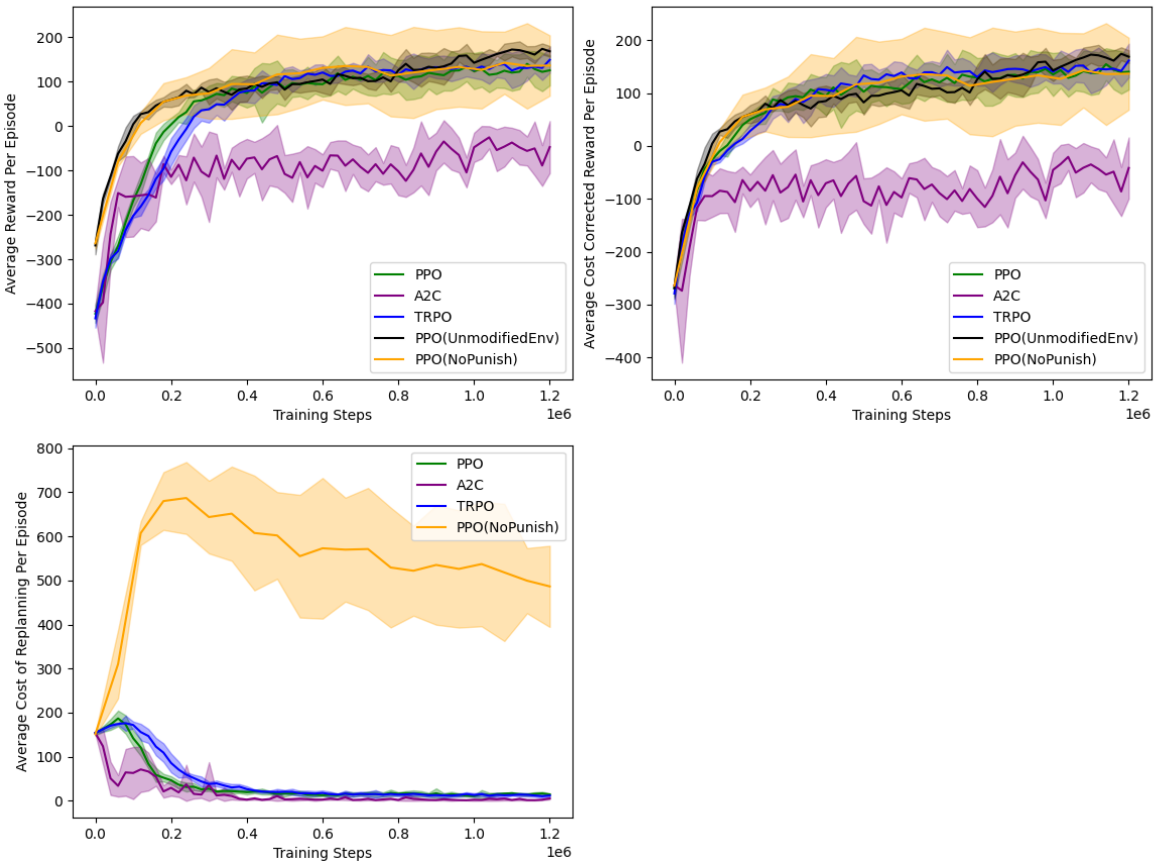

Figure A5: Performance of the PPO, TRPO and A2C algorithm on the modified LunarLander environment where agents are expected to plan 3 steps ahead compared to a PPO agent that is not punished for changing its plans, and an agent which is not expected to plan ahead at all. Critically, this environment is made stochastic with the wind feature, which applies a random wind force at every time-step.

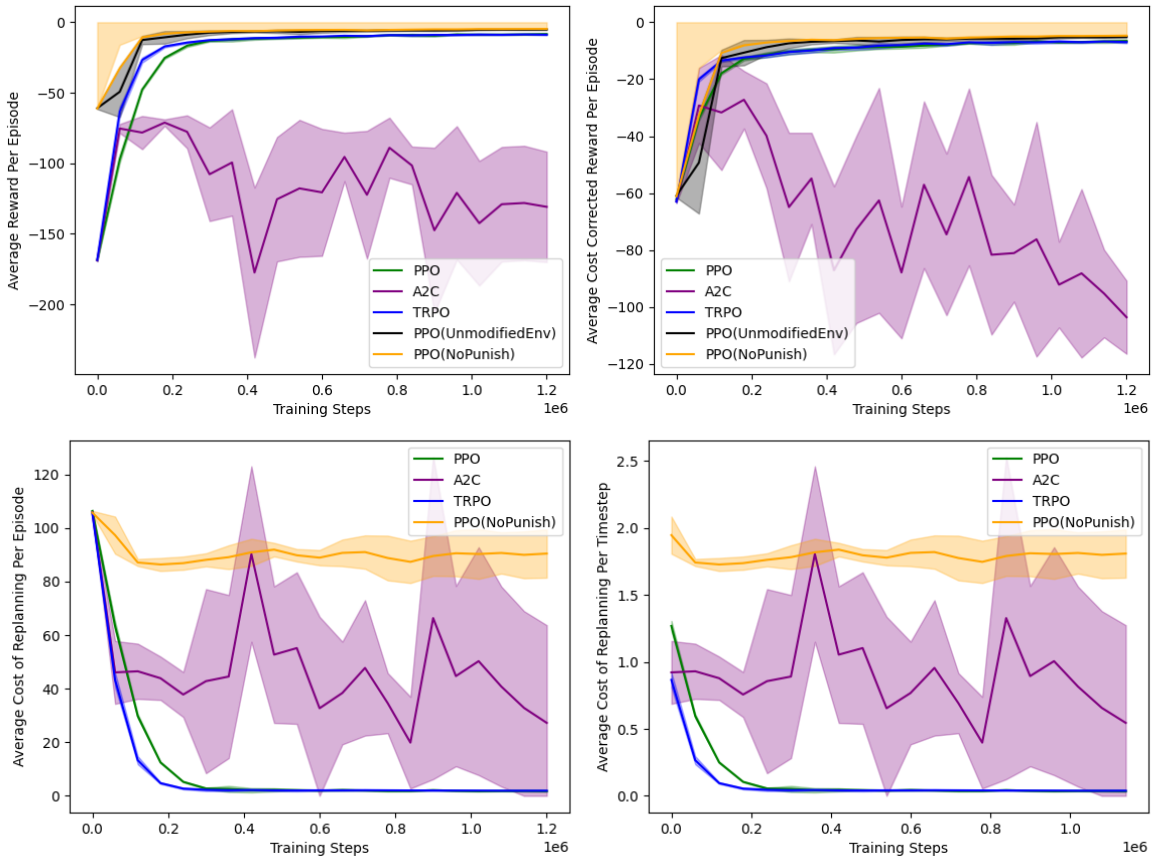

Figure A6: Performance of the PPO, TRPO and A2C algorithm on the modified Reacher environment where agents are expected to plan 3 steps ahead compared to a PPO agent that is not punished for changing its plans, and an agent which is not expected to plan ahead at all.

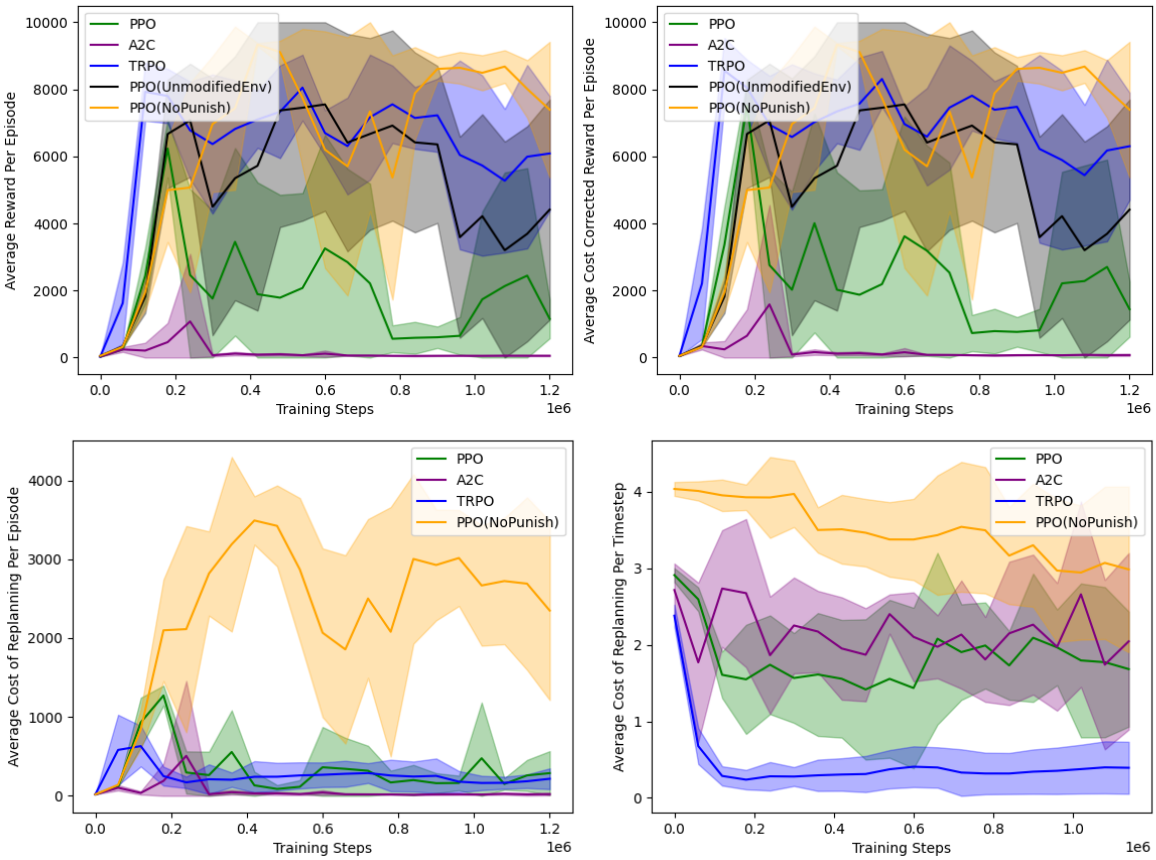

Figure A7: Performance of the PPO, TRPO and A2C algorithm on the modified Inverted-DoublePendulum environment where agents are expected to plan 3 steps ahead compared to a PPO agent that is not punished for changing its plans, and an agent which is not expected to plan ahead at all.

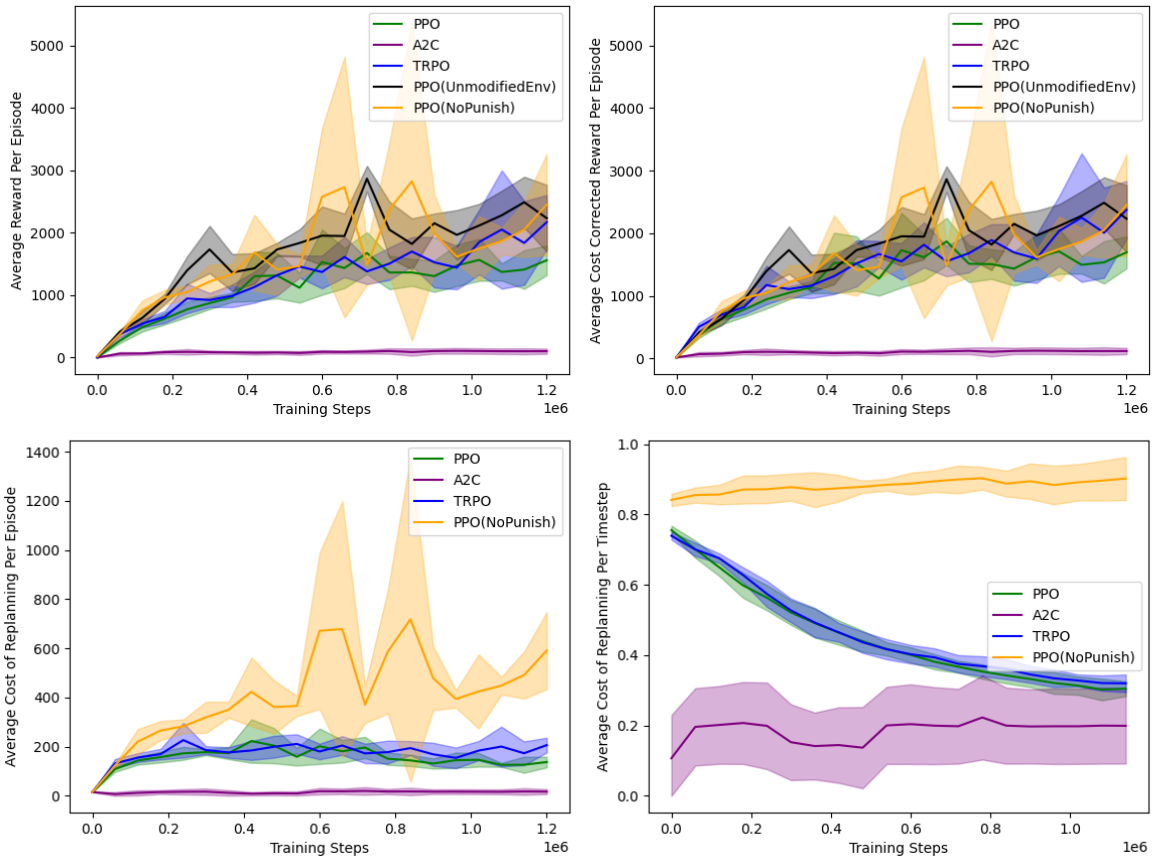

Figure A8: Performance of the PPO, TRPO and A2C algorithm on the modified Hopper environment where agents are expected to plan 3 steps ahead compared to a PPO agent that is not punished for changing its plans, and an agent which is not expected to plan ahead at all.

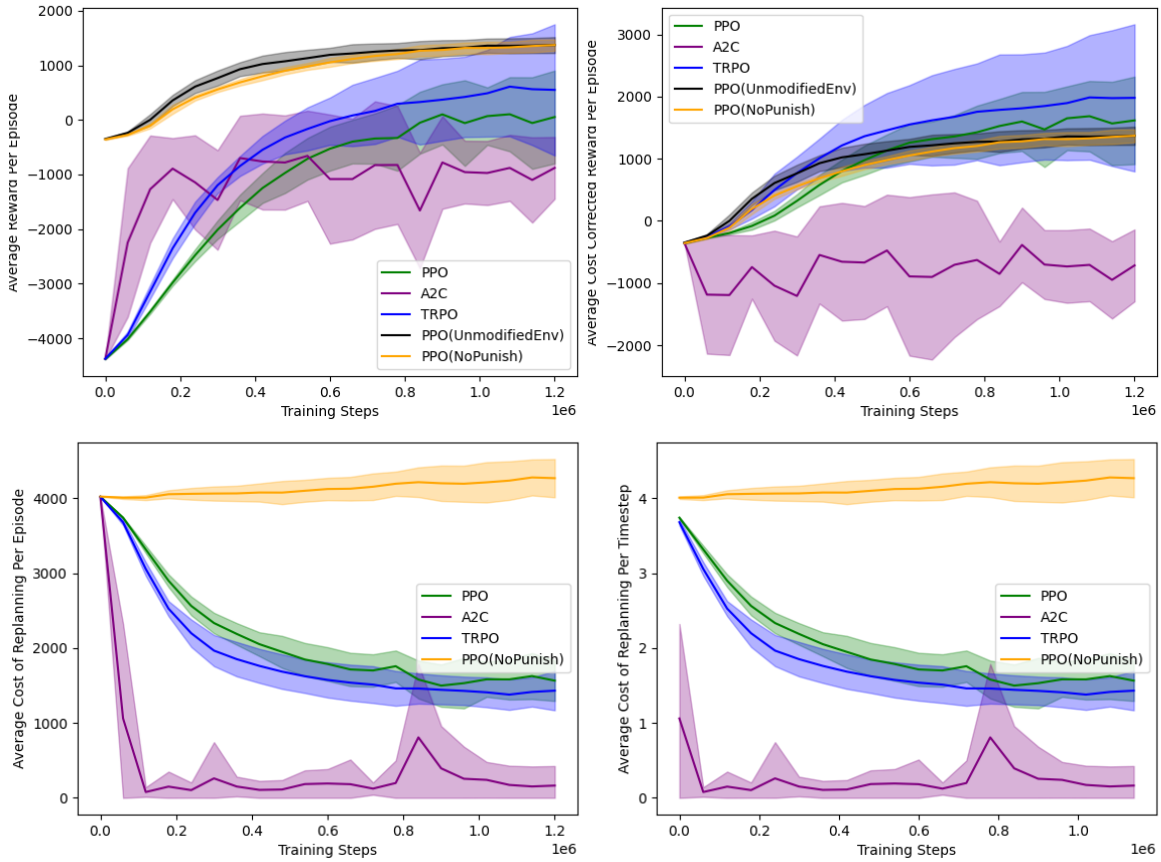

Figure A9: Performance of the PPO, TRPO and A2C algorithm on the modified HalfCheetah environment where agents are expected to plan 3 steps ahead compared to a PPO agent that is not punished for changing its plans, and an agent which is not expected to plan ahead at all.

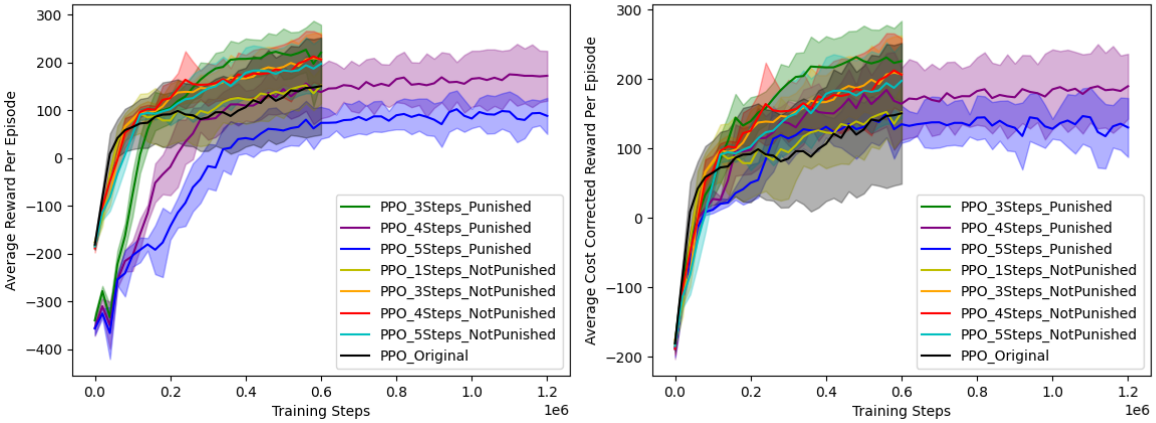

Figure A10: Comparison of PPO agents on the modified LunarLander environment where agents are expected to plan 3, 4 and 5 steps ahead compared to PPO agents that are not punished for changing their plans which are also expected to plan 3, 4 and 5 steps ahead. These are also compared an agent which is not expected to plan ahead at all, and the same agent which gets its previous action but is also not expected to plan ahead. This last agent demonstrates that no advantage is gained by having access to your last move in this environment.