WHAT MATTERS IN RECURRENT PPO FOR LONG EPISODIC AND CONTINUING PARTIALLY OBSERVABLE TASKS

Kshitij Gupta
Mila Québec AI Institute
kshitij.gupta@mila.quebec

ABSTRACT

Recurrent PPO is increasingly becoming a popular algorithm for solving partially observable tasks. However, most existing publicly available implementations use handcrafted techniques, are brittle and are difficult to reproduce, slowing down overall research progress. Not only do they have variations, they do not account for staleness in hidden states and advantage estimates leading to poor performance in more complex environments. In this work, we study and demonstrate the importance of various design decisions for Recurrent PPO in partially observable domains with long episodes and in continuing tasks. We also show that simple strategies like updating hidden states (before collecting new experience) and recomputing hidden states (before each epoch in minibatch gradient descent) can prevent staleness in updates and significantly improve performance. Finally, we provide practical insights and recommendations for implementing Recurrent PPO.

1 INTRODUCTION AND MOTIVATION

Recently, Reinforcement Learning (RL) algorithms have shown increasing success in fully observable settings. While most RL algorithms are tested in the fully observable task, there are numerous real world applications that are partially observable. Partially observable environments pose additional challenges that make adaptation of existing algorithms non-trivial. Recent works (Ni et al., 2021) argue that recurrent model-free RL is a competitive baseline and performs as strongly as some recent state of the art algorithms across a range of different POMDP settings.

However, they are not completely reliable and are also hard to reproduce and brittle. It has also been shown that these algorithms require careful implementation and attention to detail due to numerous moving parts. Various high-level and low-level design decisions are made that drastically affect the performance of each algorithm. These choices have not been studied extensively in the case of Recurrent PPO and is the aim of this study.

Most popular implementations of Recurrent PPO that available publicly are complex codebases with slight variations. They are hand crafted specifically for fixed environments and settings, making reproducibility and comparisons across algorithms difficult. This often leads to delays and incomplete/partial and unfair comparisons, thus slowing down the research community (Andrychowicz et al., 2020).

Not only do they have slight variations, they do not incorporate a lot of the findings found in recent off-policy RL settings (Kapturowski et al., 2019) and fully observable settings (Andrychowicz et al., 2020), like recalculating advantages and hidden states before each epoch. The objective of this work is to study and compare various important design choices that are commonly used and additionally test and compare new variants inspired by recent findings in other settings. We also show that the impact of these variations becomes more significant with increasing episode lengths, complexity and memory requirements in the problem setting.

Our key goal in this work is to investigate the multitude of differences in the implementation of Recurrent PPO in complex long episodic and continuing partially observable environments. Hence, as our key contributions we:

1. Analysed and researched popular codebases and prior work including both off-policy RL and on-policy Non recurrent RL to identify numerous potential factors that could affect the performance in Recurrent PPO for long episodic and continuing tasks.
2. Implemented a unified framework to test and perform ablation study of all the factors and combinations.
3. Conducted large scale study with more than 75-100 experiments over three environments with varying complexity, memory requirements and episode lengths to study the impact of each factor and various permutations of the factors.
4. Find that small, intuitive changes that are usually not present in most popular implementations can drastically improve the performance.
5. Analysed experimental results to provide practical insights and recommendations for implementing Recurrent PPO.

2 BACKGROUND

Standard Reinforcement Learning (RL) Setting: In the standard RL setting, an agent interacts with the environment and learns a policy π that maps states to a distribution over actions. At every timestep the agent uses the state and its internal representation to produce an action. After executing this action, the agent transitions to a new state and receives a reward. The agent maximizes the expected return which is the sum of discounted rewards throughout the episode.

Partially observable Markov Decision Process: There are numerous real world applications that are partially observable, and POMDPs provide a framework to study these problems. Common POMDP tasks include problems where the states are partially occluded, random frames are dropped, noise is added to the state, or only part of the complete state space is observed. POMDPs are hard to solve because of non-stationarity of the hidden states, and the curse of dimensionality: the size of the history grows linearly with the horizon length. (Ni et al., 2021)

Recurrent Model Free RL: Recurrent policies using LSTMs (Hochreiter & Schmidhuber, 1997) and GRUs (Cho et al., 2014) have often been used to compress the entire history of past observations into the hidden state which is also used along with the current observation to make decisions. This strategy is very simple and can be applied to numerous tasks. Ni et al. (2021) argues that recurrent model-free RL is competitive with the recent state of the art algorithms across a range of different POMDP settings.

PPO with Clipped Surrogate objective (Schulman et al., 2017): The Proximal Policy Optimization (PPO) algorithm combines ideas from A2C (having multiple workers) and TRPO (it uses a trust region to improve the actor). The main idea is that after an update, the new policy should be not too far from the old policy. For that, PPO uses clipping to avoid updates that are too large. PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small.

Algorithm 1 PPO with Clipped Objective

Input: Initial policy parameters θ_0, clipping threshold ϵ
for k = 0, 1, 2, . . . **do**
  Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Compute policy update

$$\theta_{k+1} = \text{argmax}_\theta \, \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathbb{E}_{\pi \sim \pi_k} \left[\sum_{t=0}^{T} \left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t^{\pi_k}) \right] \right]$$

end for

Generalized Advantage Estimation (GAE) (Schulman et al., 2018): Proposes the generalized advantage estimator, and has argued that the key to variance reduction in policy gradient approaches is using good estimates of the advantage function. There are two parameters γ and λ to adjust the bias variance trade off. It is a method that combined multi step returns in the following way: where lambda is a hyperparameter choice controlling the bias variance trade off.

$$
\begin{aligned}
A_0^{GAE} &= \delta_0 + (\lambda\gamma)\delta_1 + (\lambda\gamma)^2\delta_2 + \ldots + (\lambda\gamma)^{n-1}\delta_{n-1} \\
A_1^{GAE} &= \delta_1 + (\lambda\gamma)\delta_2 + (\lambda\gamma)^2\delta_3 + \ldots + (\lambda\gamma)^{n-2}\delta_{n-1} \\
\Rightarrow A_{t-1}^{GAE} &= \delta_{t-1} + (\lambda\gamma)A_t^{GAE}
\end{aligned}
$$

N step return: Where N depends on the sequence length for each sequence in the batch of data collected

$$
\hat{V}_t^{(N)} = \sum_{i=t}^{t+N-1} \gamma^{i-t} r_i + \gamma^N V(s_{t+N}) \approx V^\pi(s_t)
$$

3 Related work

Implementations: Numerous implementations of Recurrent PPO (Willems & contributors, 2017; Pleines & contributors, 2021; `seungeunrho` & contributors, 2019; Kostrikov & contributors, 2018) are publicly available. However, each one of them implements a slightly different version and hand-crafts techniques to perform well on specific environments. None of the above implementations directly take into account stale hidden states, stale advantage targets, and different targets for Value function updates.

Engstrom et al. (2020) also investigates the consequences of "code-level optimizations" and show that such optimizations have a major impact on agent behavior and are responsible for most of PPO's gain in cumulative reward over TRPO. Similarly Andrychowicz et al. (2020) also studies impact of numerous design choices for on-policy RL algorithms. However, these studies mainly focus on non-recurrent PPO and fully observable domains. This work develops a unified framework to study and test various differences in popular Recurrent PPO algorithms and also highlight the benefit of refreshing hidden states, recalculating advantages and using different Targets for Value function updates.

Recurrent model free RL: Ni et al. (2021) shows that the careful design and implementation of recurrent model-free RL is critical to the performance of the algorithm. The main design decisions considered in that paper include the actor critic architecture, conditioning on previous actions and or rewards, the underlying model free RL algorithms and context length in the RNN. However, they do not extensively focus on various algorithmic differences like stale hidden states and advantages, and varying targets for value function updates for the PPO algorithm.

It was shown in off-policy RL (Kapturowski et al., 2019), that storing and reusing hidden states in the replay buffer can suffer from the effect of 'representational drift' leading to 'recurrent state staleness', as the stored recurrent state generated by a sufficiently old network could differ significantly from a typical state produced by a more recent version. This could potentially result in poor performance. Ndousse (2020) also shows this effect in the case of on-policy RL. It was also shown in Andrychowicz et al. (2020) that even in the on-policy RL scenario, the advantage values used to estimate the state values in algorithms like PPO can become state over the course of a single update, resulting in poor performance. Andrychowicz et al. (2020) suggests recalculating advantages before the start of each epoch resulting in more accurate estimates of advantages improving performance. While these studies show the effects of stale hidden states and advantage estimates and its potential impact on the performance, they do not consider the impact of these factors in partially observable domains in the on-policy RL setting, which is the focus of this study.

4 Detailed discussion of the Research/Analysis Questions studied

4.1 Environment and Experimental Setting

Various environments with varying levels of complexity, episode length and memory requirements are considered to evaluate the impact of different factors with respect to the complexity of the environment. The following environments are considered:

Cartpole: A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. The velocity in the observation space has been masked to make the environment partially observable.

Minigrid (Chevalier-Boisvert et al., 2018): Minigrid is built to support tasks involving natural language and sparse rewards. The observations are dictionaries, with an 'image' field, partially observable view of the environment, a 'mission' field which is a textual string describing the objective the agent should reach to get a reward, and a 'direction' field which can be used as an optional compass. By default, sparse rewards are given for reaching a green goal tile. A reward of 1 is given for success, and zero for failure. There is also an environment-specific time step limit for completing the task

Minigrid Memory: This environment is a memory test. The agent starts in a small room where it sees an object. It then has to go through a narrow hallway which ends in a split. At each end of the split there is an object, one of which is the same as the object in the starting room. The agent has to remember the initial object, and go to the matching object at split. For these experiments a more complex environment is used compared to Pleines & contributors (2021) by increasing the number of possible start states to 1000 from 100, and using the `MiniGrid-MemoryS13-v0` (instead of `MiniGrid-MemoryS9-v0`) environment with 845 Maximum timesteps instead of 405 maximum timsteps. The agent's view is decreased from the original memory environment proposed in Chevalier-Boisvert et al. (2018) to increase the memory requirement and make it more challenging.

The following configuration was used for these experiments:

- Reward function: A reward of 1 is given for success, and 0 for failure
- Action space: Rotate left/right, move forward
- Observation space: Partially observable view of the environment using a compact and efficient encoding, with 3 input values per visible grid cell, 7x7x3 values total
- Maximum number of timesteps: 845
- 1000 possible start state configurations for each seed
- 16 workers
- Implementation is extended from Pleines & contributors (2021)

Jellybean (Platanios et al., 2020): Jelly Bean World (JBW) is a 2D infinite grid world designed to enable and facilitate research towards never-ending learning in a multimodal setting. JBW allows experimentation over 2D grid worlds which are filled with items and in which agents can navigate. Learning tasks can be defined in terms of reward functions and reward schedules.

The following configuration was used for the purpose of these experiments:

- Reward function - Avoid Onion:
  - +10: Pick any item up
  - -1: Pick onion up
  - -0.005: Living Reward
- The action space consists of three actions:

  - '0': Move forward
  - '1': Turn left
  - '2': Turn right
- Observation Space:
  - Scent: Vector with shape [S], where S is the scent dimensionality
  - Vision: Matrix with shape [2R+1, 2R+1, V], where R is the vision range and V is the vision/color dimensionality. In these experiements R = 8, V = 3
- Max timesteps: Trained for 2-4 Million timesteps
- Number of workers: 1
- Batch gradient descent

4.2 Model Architecture

For Jellybean: Observation is processed by convolutional layers. The output representation is passed through an LSTM or a two-stream Transformer (combination of GRU with Transformer). The output is then passed through two separate heads consisting of linear layers for the policy and value function.

For Cartpole and Minigrid: A GRU is used instead of LSTM. And the output of the GRU is also passed through a shared linear layer before passing though different policy and value function heads comprising of linear layers.

4.3 Factors being considered

- Stale Hidden states
- - Update_hidden: Updating hidden state before performing more rollout
  - Recalculate_hidden: Recalculating Hidden states before each epoch
- Stale Advantage targets
  - Recompute_advantange: Recomputing Advantages before each epoch
- Using different targets for the value function update:
  - GAE+Value (Normal)
  - TD Target: 1 Step TD Targets
  - N-step TD Target
- Mini batch vs Batch Gradient Descent
- Resetting hidden states between episodes
- For Continuing Task: Using Average Reward scenario
- Note: For jellybean: Refresh is a combination of Recalculate_hidden and Recompute_advantange

4.4 Evaluation Metrics and Seeds

For Cartpole and Minigrid: The mean of the Returns is plotted across different episodes from different rollout workers and the median is plotted across 3 Random seeds to ignore outliers.

For Jellybean: The average reward across 100,000 timesteps is being considered across 2-3 Random seeds.

4.5 Algorithm Design

Algorithm 2 Algorithm Design

Input: Initial policy parameters θ_0, and value parameters ϕ_0
for k = 0, 1, 2, . . . **do**
    Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$ and current hidden states h. *Reset hidden states when Done=True if reset_hidden = True
    If not recompute_advantages: Estimate GAE Advantages
    for epoch j = 0, 1, 2, . . . **do**
        If recompute_advantages: Estimate GAE Advantages
        Compute policy update
        Compute Value update using different targets: N-step TD Target, TD Target or GAE+Value,
        Take M steps of minibatch SGD (via Adam) or do 1 step of Batch GD
        if recompute_hidden_states or recompute_advantages **then**
            Pass entire batch through Model to output New Value estimates and Hidden states
            If recompute_hidden_states: Update Hidden states to the new calculated Hidden states
            If recompute_advantages: Update Values to newly calculated Value estimates
        end if
    end for
    If update_hidden: Update current hidden states h for new rollouts
end for

4.6 Discussion

4.6.1 Cartpole

As we can see from Figure 1 in the Cartpole environment these factors do not change the performance significantly. However, as we move to more complex environments with longer episodes and more memory requirements, the impact of these factors becomes significant as shown below.

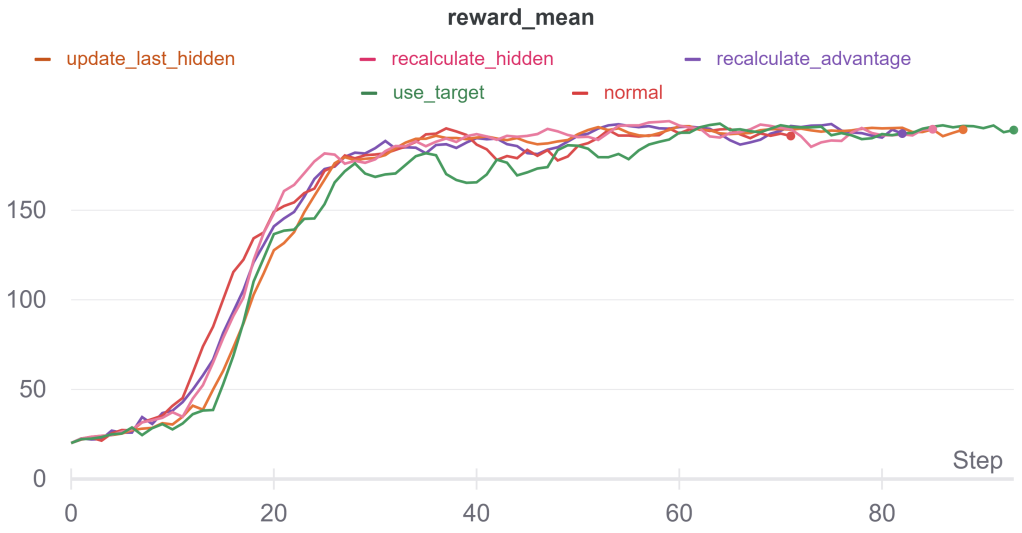

Figure 1: Cartpole

4.6.2 MINIGRID MEMORY ENVIRONMENT

Resetting Hidden State: As we can see in Figure 2, not resetting hidden states between episodes significantly improves the performance even in episodic environments. This could potentially be because the hidden memory is able to learn and perform credit assignment across episodes.

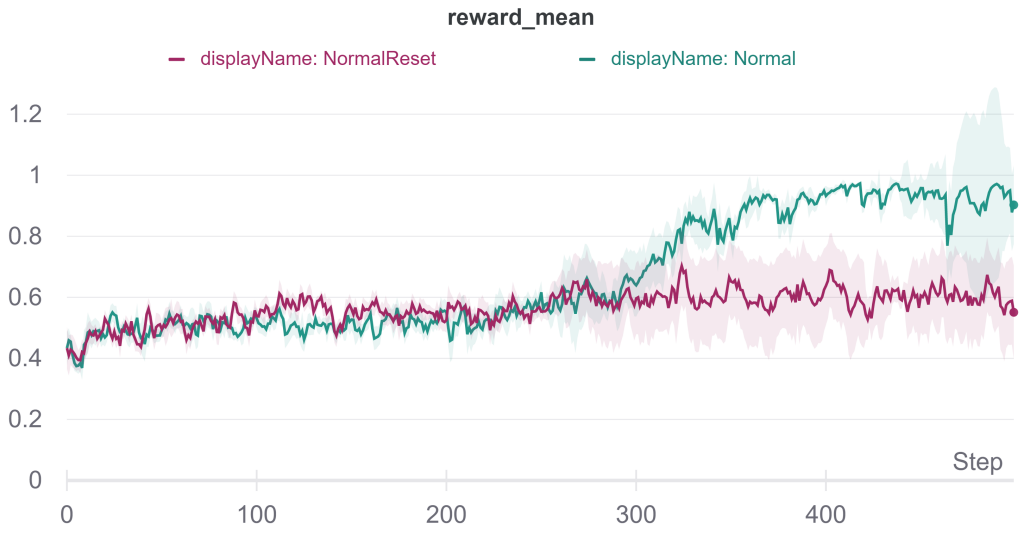

Figure 2: Reset vs No Reset

Minibatch vs Batch Gradient Descent: As we can see in the Figure 3, minibatch gradient descent outperforms batch gradient descent.

For the rest of the experiments in the Minigrid environment we run all experiments with minibatch gradient descent without resetting the hidden states across episodes.

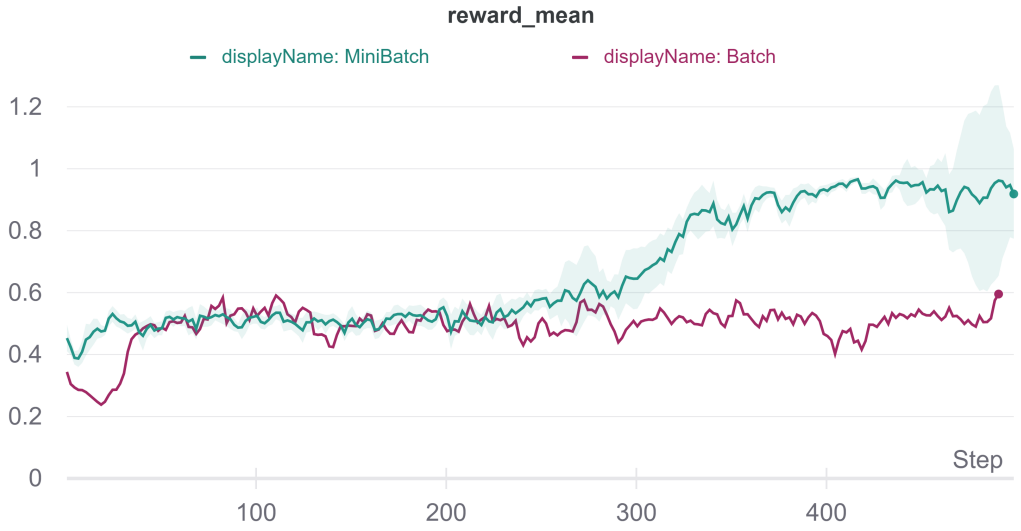

Figure 3: Batch vs Minibatch Gradient Descent

Recalculating Advantages, Hidden states and Updating Hidden states: We study the importance of recalculating advantages, hidden states and updating the hidden states before performing rollouts to avoid stale states and targets. We can see see as shown below that these factors significantly improve the performance by not using stale hidden states and advantage estimates.

We can see in Figure 4 that updating the hidden states alone converges to the optimal policy the fastest. We can also see that recalculating hidden states improves the performance and sample efficiency compared to the Normal PPO variant. Surprisingly we can see that recalculating advantages hurts the performance of PPO.

We can also see from Figure 5 that combining both the factors (updating hidden states and recalculating hidden states) significantly improves the performance and increases the sample efficiency by 2 times compared to the Normal PPO variant. However, we can see from Figure 8 that also recalculating advantages with the above combination degrades the performance, indicating that recalculating advantages in general degrades performance in the Minigrid Environment.

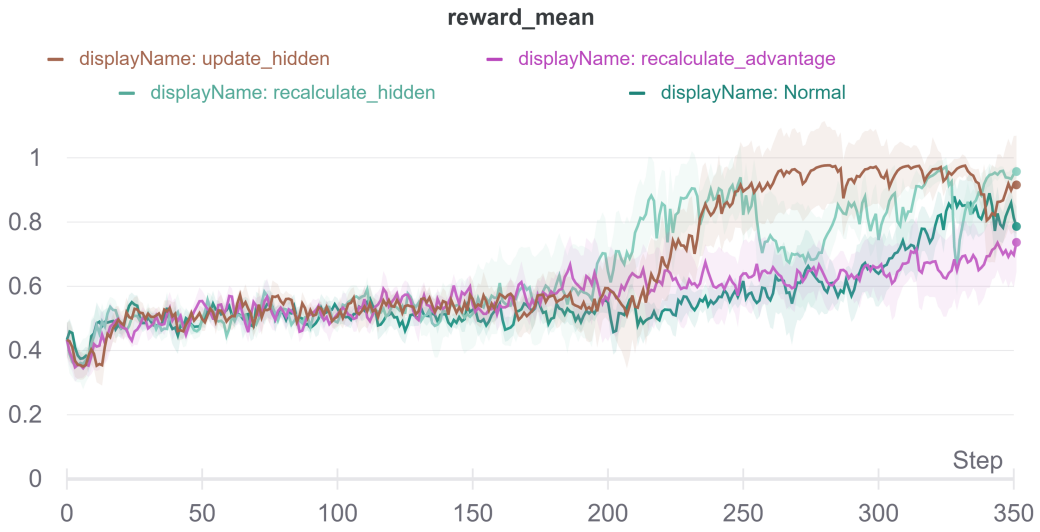

Figure 4

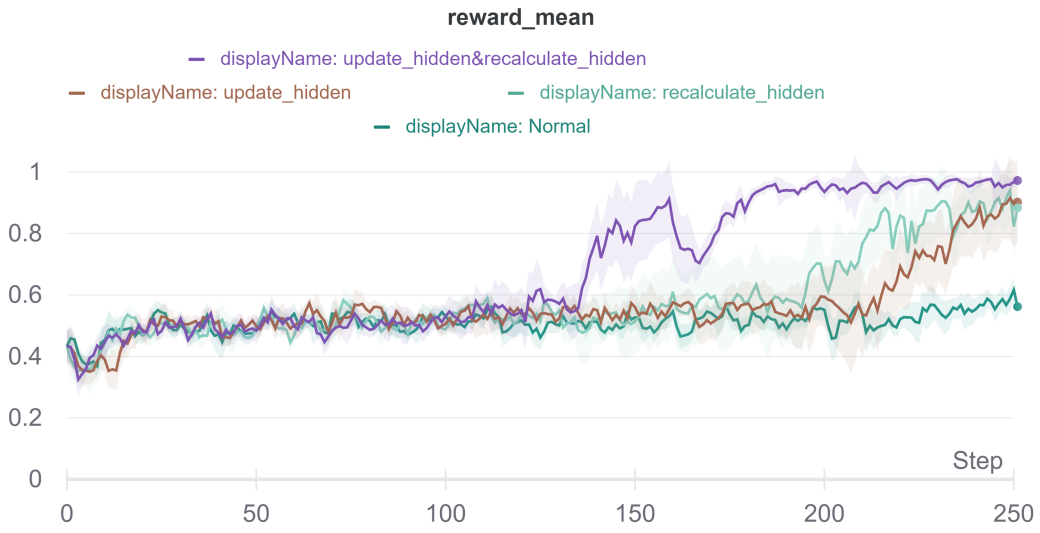

Figure 5

Effect of using different Targets for Value function updates: From Figure 6 we can see that without using any other factor, Normal PPO which uses GAE+Value estimate performs the best. We can also see from Figure 7 that when TD targets are combined with update_hidden and recalculate_hidden variant, the performance significantly degrades compared to using N-step TD Targets or the popular GAE+Value target.

Interesting Observation: We can also see from Figure 8 that in the update_hidden+recalculate_hidden+recalculate_advantage, using TD targets performs the best compared to N-step TD Targets and GAE+Value targets.

Interesting Observation: The update_hidden+recalculate_hidden variant was run long enough after convergence for different targets. It can be seen from Figure 9 that using N Step TD Targets helps stabilize the performance of this variant after convergence as compared to using the typical GAE+Value Target.

This indicates that the effect of using different targets depends on the variant of the algorithms and can lead to both performance improvement or degradation.

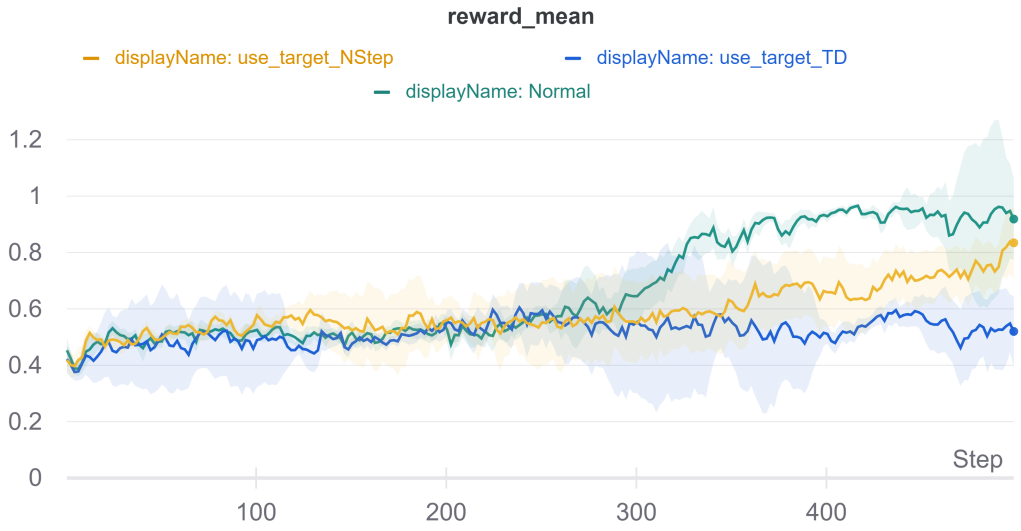

Figure 6

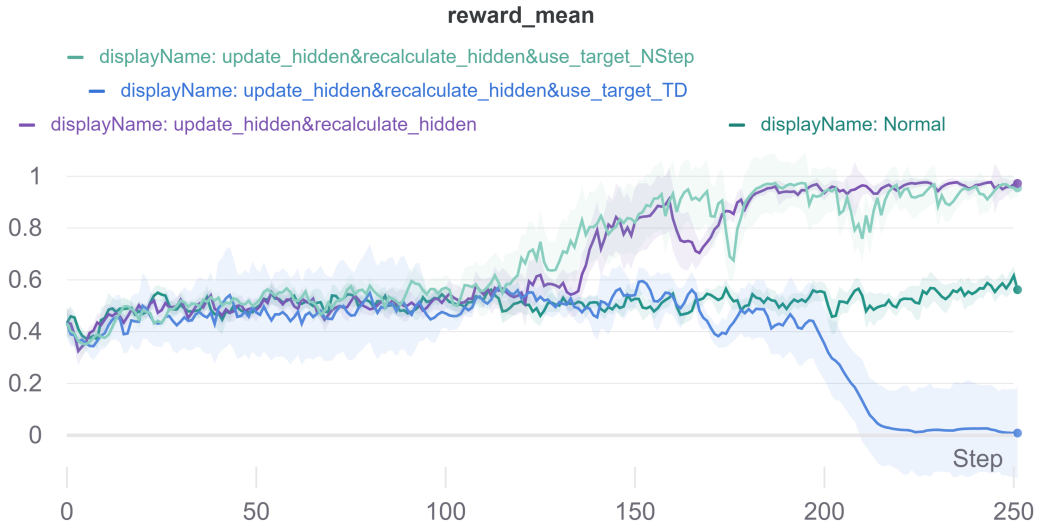

Figure 7

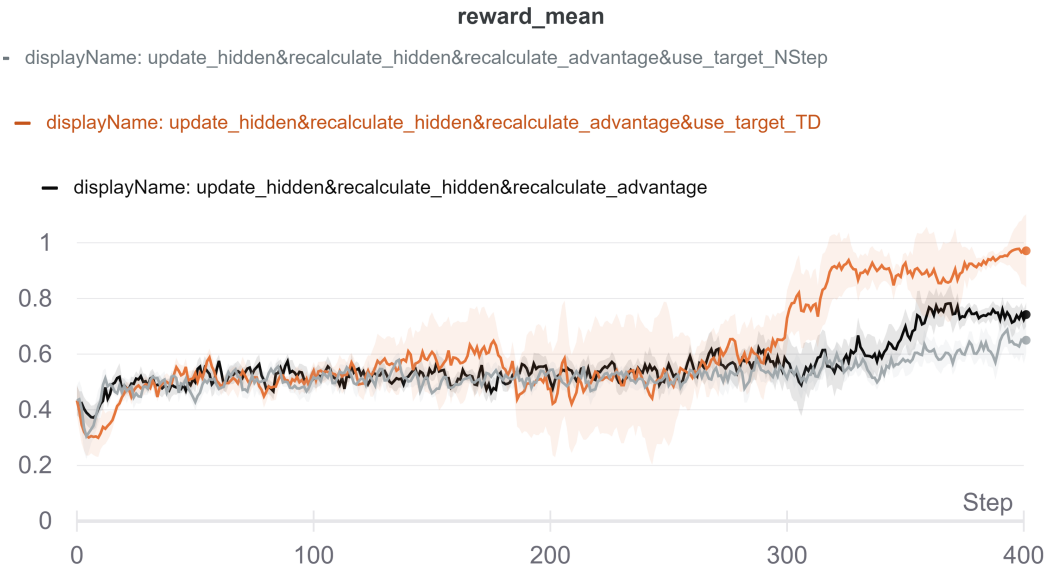

Figure 8

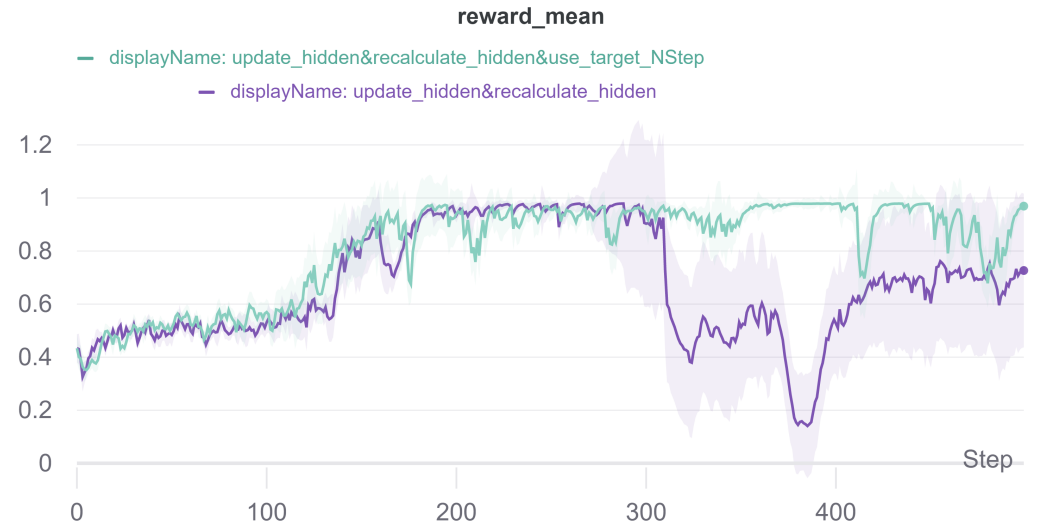

Figure 9

4.6.3 JELLYBEAN ENVIRONEMENT

Recalculating Advantages, Hidden states and Updating Hidden states: For the Jellybean Environment we can see in Figure 10 that LSTMPPORefresh (Recalculate Hidden states and Recalculating Advantages) performs significantly better than the normal LSTMPPO version. We can also see from Figure 11 that just updating the hidden states before performing further rollouts to collect new experience can significantly boost performance.

Using Different Targets: As we an see from Figure 11 Using TD targets significantly boosts performance compared to Normal PPO. We can also see that updating hidden states while using TD Targets further improves performance. However, we can see from Figure 10, also recomputing advantages and hidden states (refresh variant), degrades the performance while using TD Targets. This indicates that recalculating advantages and hidden states is not always beneficial and using different targets has different effects for different variants.

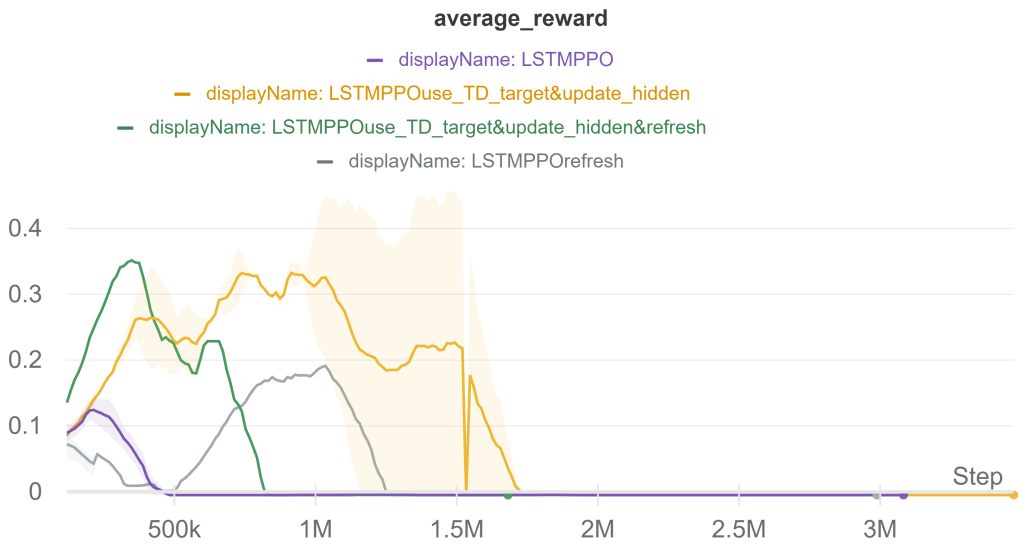

Figure 10

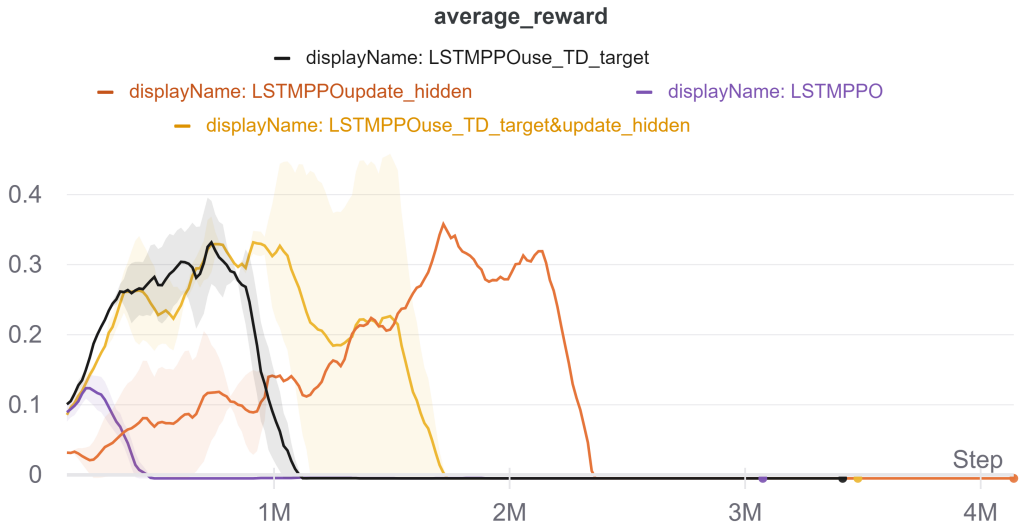

Figure 11

Average Reward Formulation: As we can see from Figure 12 that using average rewards results in more consistent performance compared to using normal rewards, but the average performance is much lower than the highest average reward the agent using normal reward formulation achieves.

Two-stream Transformer Architecture: Two-stream transformer architecture was also implemented and tested. The performance is very poor initially compared to LSTM/GRU, however, the performance interestingly shoots up after 2M timesteps as seen in 12. Note that due to limited time and compute this architecture was not fine-tuned and requires more fine-tuning, since one run took 17-20 hours.

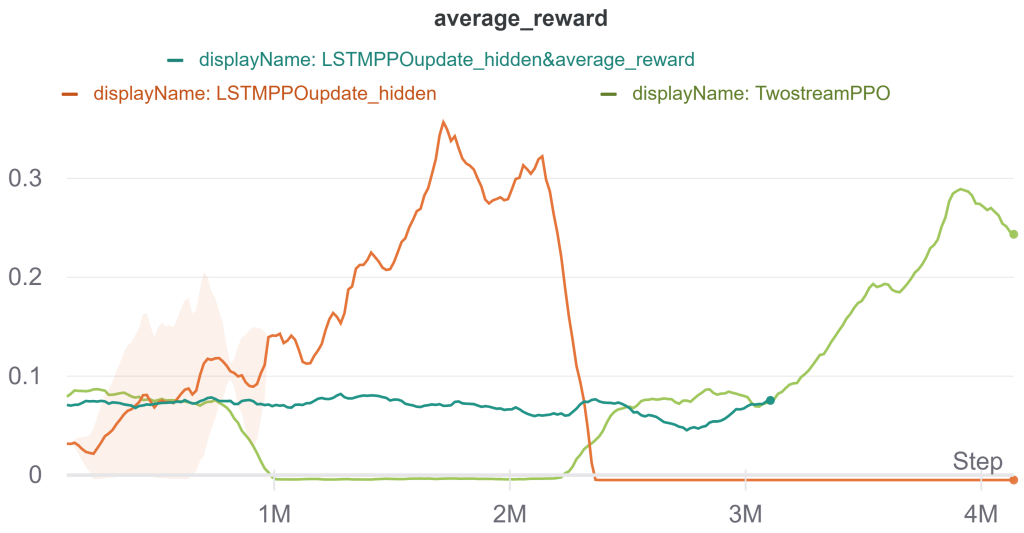

Figure 12

4.7 Summary

We now summarize the main findings of our experiments below:

- As episodes become longer and environments become more partially observable (requires more memory), the effect of these factors increases
- Updating the last hidden state and recalculating all hidden states before each epoch improves the sample efficiency 2 to 3 times in Minigrid World.
- It is recommended to try different targets for the value function update and not just GAE+Value which is popularly used:
  - Different targets have different effects across different variants
  - N-step TD Targets can help stabilize training after convergence
- Recalculating advantages can improve the performance but could potentially degrade the performance when done with certain variants
- Minibatch gradient descent is very important and can significantly boost the performance compared to Batch Gradient Descent
- Not resetting hidden states across episodes can significantly improve the performance in certain episodic tasks.

We hope these findings can provide a useful initialization for Recurrent PPO.

5 Conclusion

In this work, we study and demonstrate the importance of various design decisions for Recurrent PPO in Partially Observable domains with long episodes and in continuing tasks. We also show how updating hidden states and recomputing hidden states can significantly improve performance. Surprisingly this is not done in most of the popular repositories implementing Recurrent PPO. We encourage future work to further study the impact of these decisions across more complex tasks and multi-task settings as well as other on-policy RL algorithms.

6 Link to Code

https://github.com/kshitijkg/RLTransformer

7 CONTRIBUTIONS BY EACH TEAM MEMBER

The entire work in this report was done by the author.

REFERENCES

Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters in on-policy reinforcement learning? a large-scale empirical study, 2020.

Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. https://github.com/maximecb/gym-minigrid, 2018.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep policy gradients: A case study on ppo and trpo, 2020.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997.

Steven Kapturowski, Georg Ostrovski, Will Dabney, John Quan, and Remi Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=r1lyTjAqYX.

Ilya Kostrikov and contributors. Pytorch implementation of advantage actor critic (a2c), proximal policy optimization (ppo), scalable trust-region method for deep reinforcement learning using kronecker-factored approximation (acktr) and generative adversarial imitation learning (gail). https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail, 2018.

Kamal Ndousse. Stale hidden states in ppo-lstm. https://kam.al/blog/ppo_stale_states, 2020. Accessed December 2021.

Tianwei Ni, Benjamin Eysenbach, and Ruslan Salakhutdinov. Recurrent model-free rl is a strong baseline for many pomdps, 2021.

Emmanouil Antonios Platanios, Abulhair Saparov, and Tom Mitchell. Jelly bean world: A testbed for never-ending learning, 2020.

Marco Pleines and contributors. Baseline implementation of recurrent ppo using truncated bptt. https://github.com/MarcoMeter/recurrent-ppo-truncated-bptt, 2021.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

seungeunrho and contributors. Implementations of basic rl algorithms with minimal lines of codes. https://github.com/seungeunrho/minimalRL, 2019.

Lucas Willems and contributors. Rl starter files. https://github.com/lcswillems/rl-starter-files, 2017.