Linear Operator Distillation Agent

Maximilian Alfano-Smith

May 2022

For my CSE 185 Final Project, I will building on the Random Network Distillation paper by proposing a Reinforcement Learning agent with multiple random network based intrinsic reward signals; these signals along with a policy network are than used to form the agent's behavioral policy. I shall detail my design and implementation of this agent, as well as run a hyper parameter search and final test experiments on the Cart pole environment. My intended audience is Reinforcement Learning and Machine Learning researchers interested in different approaches to how we model the agent's policy.

CSE 185 Final Project
Maximilian Alfano-Smith
mealfano

Abstract

In the field of machine learning the most general problem statement is that of Reinforcement Learning; however given its generality it is often quite difficult to solve. The most popular methods are derivatives of the Actor Critic Framework, and our contribution follows in this tradition by building on the Random Network Distillation technique in order to build a behavioral policy that corresponds to multiple intrinsic rewards. We investigate this model's performance on the simple Cartpole environment.

1 Introduction

While many things have changed in the world and in my life over the last two years, Reinforcement Learning Research has been a constant through the good the bad and the ugly. Because of this, it seemed fitting to write about a Reinforcement Learning agent for my CSE 185 final project. Reinforcement Learning is a subfield of Machine Learning that focuses on problems in which the computer must learn to perform some decision task with the objective of maximizing a reward it receives based on the decisions it makes completing that task. In order to reason about tasks of this form, we formalize them as Markov Decision Processes, which consist of:

- An agent that is interacting with the process and its associated policy π.
- A set of states S that the agent can be in.
- A set of possible actions A the agent can take.
- An environment that tracks the agent's current state S_t and, using just this information, maps an input action A_t from the agent to the agent's next state S_{t+1} and its reward $R_t = R(S_t, A_t)$ for taking that action in its current state. Where $P(S_{t+1} = s'|S_t = s, A_t = a)$ (abbreviated $P(s'|s, a)$) is the probability that the next state the environment will return is s' given the agent's current state is s and it decided to use action a. The key assumption of processes of this type (called the Markov

assumption) is that only the current state and action are taken into account by the environment when determining the following state and reward.

Since the following state S_{t+1} and reward R_t depend only on the current state S_t and action A_t at time t, we can model our agent's policy π as purely as a distribution over possible actions A, condition on the agent's current state S_t:

$$P(A_t = a|S_t = s) = \pi(a|s)$$

We then define a decayed expected state value function:

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \left[R(s, a) + \gamma \sum_{s' \in S} P(s'|a, s) V_\pi(s') \right]$$

Where $s = S_t, s' = S_{t+1}$ and $0 < \gamma \leq 1$ is a decay term that controls how much future rewards are taken into account when determining a states value. Now to form an expected value function for π, $J(\pi)$, we need to employ the Markov assumption as it implies there exists a stationary distribution over states S given our policy π, $d_\pi(S_t = s)$, which is essentially the probability of drawing the state s from an collection infinite of samples collected using the policy π.

$$J(\pi) = \sum_{s \in S} d_\pi(s) V_\pi(s)$$

Since in contemporary Reinforcement Learning we are often using neural networks and other gradient optimized non-linear function approximators, we would like to find $\nabla J(\pi)$ without differentiating d_π. Luckily an approximation called Policy Gradient was found and later improved in the form of the Actor Critic Algorithm [1]. In traditional action critic, we use separate neural networks to model both the policy π and an estimate of its associated state expected value, $\overline{V}_\pi$, which allows us to form a measure of the current policy's performance called the advantage:

$$A^{\pi, \overline{V}_\pi}(s, a) = R(s, a) + \gamma \overline{V}_\pi(s') - \overline{V}_\pi(s)$$

This then gives us our Actor Critic estimate of the policy's gradient:

$$\nabla J(\pi) \approx \mathbb{E}(A^{\pi,\overline{V}_\pi}(s,a)\nabla_\pi \ln(\pi(a|s))$$

As one might expect, taking the expectation of a shifting gradient repeatedly leads to an algorithm with a lot of noise and variance. Luckily the Proximal Policy Algorithm (abbreviated as PPO) [3] helps alleviate this somewhat by keeping the gradient from getting too skewed, and therefore we have integrated it into our agent. The theory of Actor Critic and Policy Gradient alogorithms is very deep and rich; for a much deeper dive refer to Lilian Weng's blog post on it [2]. The main paper we are expanding upon is Random Network Distillation [4] which attempts to improve the exploration of the PPO algorithm by effectively giving the agent an arbitrary approximation task in addition to their normal Reinforcement Learning objective through Random Network Distillation. The agent's approximation task is to essentially train one function approximator (by collecting samples of states from S) to emulate a randomly initialized neural network that is of higher model capacity. Creating a scenario where the approximator will never actually achieve its goal leads to a constant non-zero signal of loss between approximator and approximated that can be used as an intrinsic reward $I_t = I(S_{t+1})$ for the agent; this signal is then added to the extrinsic reward R_t in order to smooth out the reward signal and consequently the policy gradient in sparse extrinsic reward environments. In most implementations, a separate critic is used to estimate a state's policy conditioned expected value relative to this intrinsic reward signal.

1.1 Linear Operator Distillation Agent

We build on the Random Network Distillation technique by proposing an agent called the Linear Operator Distillation Agent (LODA) with the following attributes:

· k random network, approximator pairs $(N_1, \tau_1), (N_2, \tau_2), ..., (N_k, \tau_k)$, where their corresponding intrinsic reward signals are $\hat{I} = \{I_1, I_2, ..., I_k\}$. Where $I_i(s) = (N_i(s) - \tau_i(s))^2$.

A nuance of our implementation is that all τ are linear operators represented as vectors rather than neural networks, allowing for a decayed gradient estimate over all samples rather then a mini batch.

- A neural network with parameters θ that maps the current state to k logit vectors $\hat{l} = \{l_1(s), l_2(s), ..., l_k(s)\}$ (with for all i $|l_i(s)| = |A|$), which are then averaged in a weighted sum (using weights $W(S_t)$) to form the logits of the behavioral policy π. So the logit for the jth action given current state s would be $\sum_{i=1}^{k} W_i(s) l_{i,j}(s)$, which is equal to its log likelihood under policy π.
- $k+1$ critic networks, with $\overline{V}_{\pi,1}, \overline{V}_{\pi,2}, ..., \overline{V}_{\pi,k}$ corresponding to random network, linear operator, pairs from earlier, and $\overline{V}_{\pi,k+1}$ corresponding to just the extrinsic reward signal R. So for $i \in \{1, ..., k\}$, $\overline{V}_{\pi,i}(s) \approx I_i(s) + \gamma \overline{V}_{\pi,i}(s')$, and in the one other case we have $\overline{V}_{\pi,k+1}(s) \approx R(s,a) + \gamma \overline{V}_{\pi,k+1}(s')$

The weights for determining the behavioral policy are defined as:

$$W_i(s) = \frac{e^{-I_i(s)}}{\sum_{j=1}^{k} e^{-I_j(s)}}$$

We use the correspondence between the set of logit vectors $\hat{l}$ and intrinsic reward signals $\hat{I}$ to define a unique advantage for each logit vector where σ is the weight of the intrinsic reward:

$$A_i^{\pi,\overline{V}_\pi}(s,a) = (R(s,a) + \gamma \overline{V}_{\pi,k+1}(s') - \overline{V}_{\pi,k+1}(s)) + \sigma(I_i(s') + \gamma \overline{V}_{\pi,i}(s') - \overline{V}_{\pi,i}(s))$$

This then gives us our approximation of the policy gradient, and from this follows the objective function we plug into our automatic differentiation software PyTorch.

$$L(\theta) = \sum_{i=1}^{k} (\mathbb{E}(A_i^{\pi,\overline{V}_\pi}(s,a) \nabla_\theta \ln(\pi_i(a|s))) \approx \nabla J(\pi)$$

Note : π_i is the policy formed purely using only the ith set of logits. The underlying intuition behind this agent is that each logit vector is associated with an intrinsic reward and our weighting favors logits vectors that correspond to intrinsic rewards which are currently lower than the others; this should hopefully skew the policy towards increasing those intrinsic rewards. We only train each logit vector with its corresponding intrinsic reward and the extrinsic reward to ensure it is biased towards its corresponding intrinsic reward.

2 Methods

One difference in our algorithm from the original is our use of linear operators for the approximators, and a benefit of this is that we can streamline the gradient calculation by keeping a decayed estimate of the current state's outer product, H, and the matrix of its product with each target value $\overline{T}$. A full derivation of this gradient estimation technique can be found in the Appendix. A note is that while this does limit the model capacity of our approximators, our ultimate objective is for them to be imperfect models anyways.

2.1 Sudo Code

We shall touch up on the implementation here, but all of it can be found in this project's GitHub Repo:

https://github.com/max9613/CSE185FinalProject

Below is the basic sudo code for the LODA optimization algorithm.

```
LODA(steps, $\theta$):
    Receive initial state $S_t := S_0$
    for step in range(steps):
        Use $\theta$ and $S_t$ to get $\hat{l}$ and using this form $\pi(S_t)$
        Get $A_t \sim \pi(S_t)$
    Give $A_t$ to the environment and get $R_t, S_{t+1}$
    Add the transition $(S_t, A_t, R_t, S_{t+1})$ to buffer $B$
    Decay $H$ and add to it $S_t S_t^\top$
    Decay $\overline{T}$ and add to it the matrix $M$, where $M_i = I_i(S_t)S_t$
    Use $H$ and $\overline{T}$ to update all $\tau_i$
    Compute $L(\theta)$ with respect to $B$ and add it times $\frac{\alpha}{e}$ to $\theta$
    Perform gradient descent using the contents of $B$ on all $k+1$ critics
```

2.2 Initial Hyperparameter Search

An unfortunate reality of Reinforcement Learning is that a lot of time is spent finding good hyper parameters, which are values set by the programmer that have a large effect on the model's performance but cannot be directly learned. For the Linear Operator Distillation agent, the main hyper parameters are:

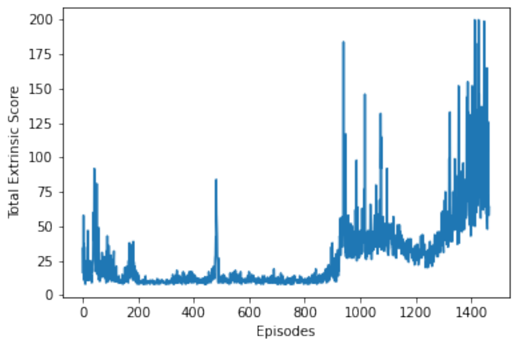
- k : The number of intrinsic rewards, network pairs, and logit vectors; this is a parameter we search over.
- α : The learning rate of the critic and approximators. We use the same value for both; this is a parameter we search over.
- The learning rate of the policy we fix as $\frac{\alpha}{e}$, since the original Actor Critic paper specifies it must be smaller then the learning rate of the critic [1].
- $\hat{d}$: The decay term on H and T; this is a parameter we search over.
- σ : The weight of the intrinsic reward.
- PPO : Whether we use PPO with an epsilon of 0.2 (value specified as best in the paper [3]) or not at all; this is a parameter we search over.

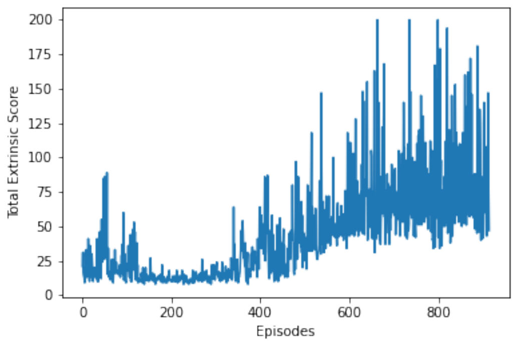
- ϕ : The non-linearity for our policy, critic, and random networks.
- random seed : The value that determines which deterministic "random" sequence our experiment will follow.

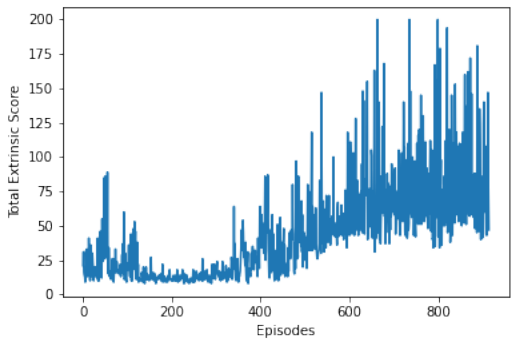
When testing a new Reinforcement Learning model, it is very helpful to run the same environment with a lot of different hyper parameters each for a relatively small number of steps (relative to one's computational resources; mine are limited) in order to see which hyper parameters are conducive to learning. This way long form experiments are hopefully not wasted on bad hyper parameter combinations. For this initial search, we chose the simple cart pole environment in the Open Ai gym python library [5]. Our initial search was 40000 steps run on all combinations of the following hyper parameter values:

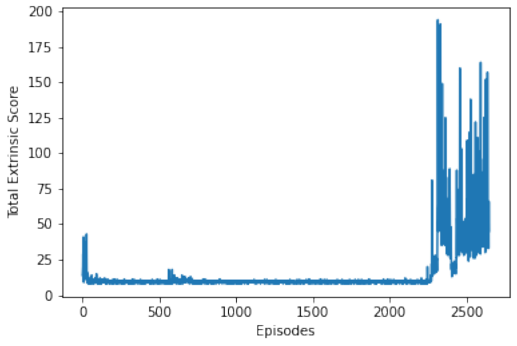
- k : 1, 4, 16
- α : $0.003 = \frac{3}{e^3}, 0.001 = \frac{1}{e^3}$
- $\hat{d}$: 0.5, 0.9
- σ : 0.001
- PPO : $\epsilon = 0.2$, off
- ϕ : Tanh
- random seed : 42

The results are underwhelming but promising, as can be seen in the top four performers there are definite gains in total extrinsic score (this is one's easiest metric for "learning") – the horizontal axis of each graph is the episode (on efull run through environment) number and the vertical access is the total extrinsic score for that episode – but it is slow:

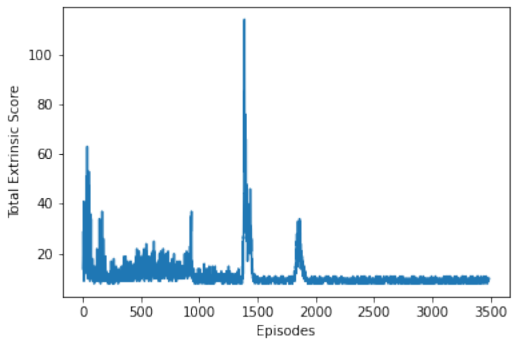


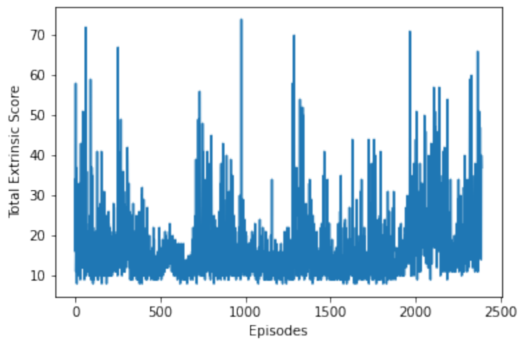




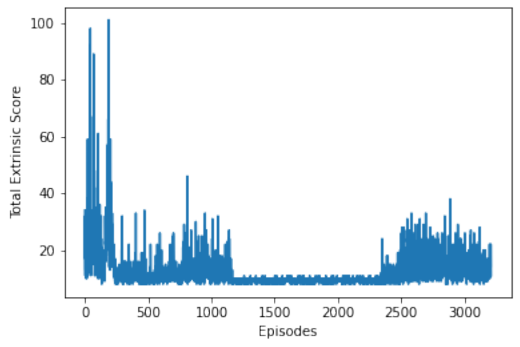


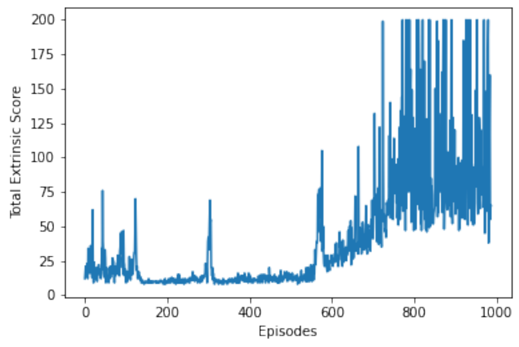
However, most look like the following:

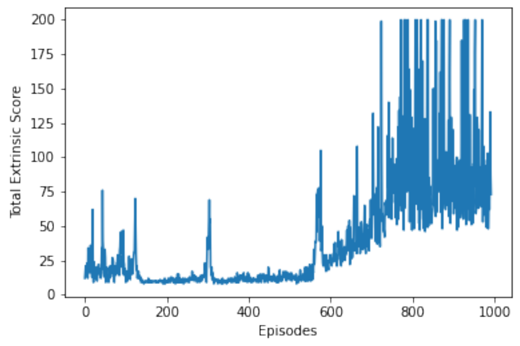


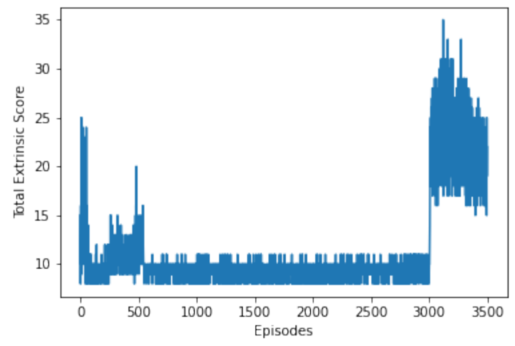


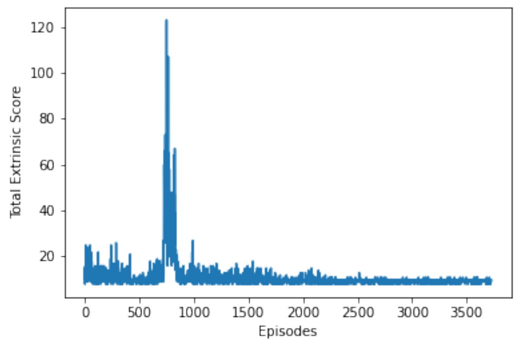
The display no increase or short lived increase. Given how few promising results there were we needed to check for whether or not the few good combinations were just random chance, so we multiplied our seed by ten and re-ran the same hyper parameter search with the new seed 420. Sadly only two of the previous top four combinations remained in the top four. The following six graphs correspond to previous six graphs, just run with the new seed:

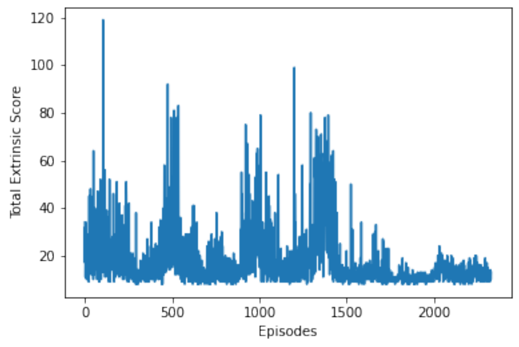




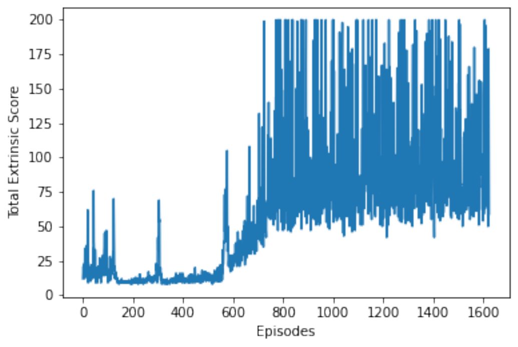


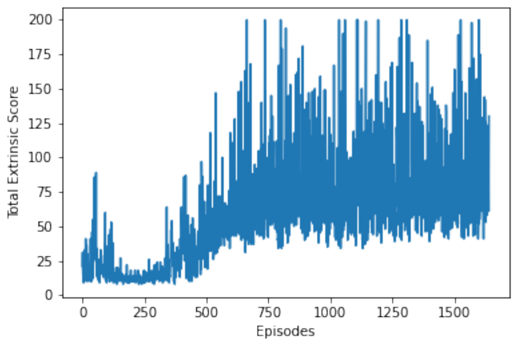


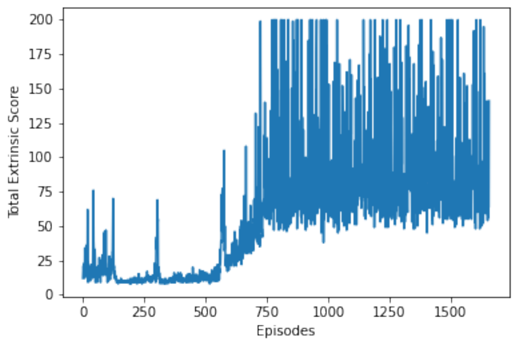


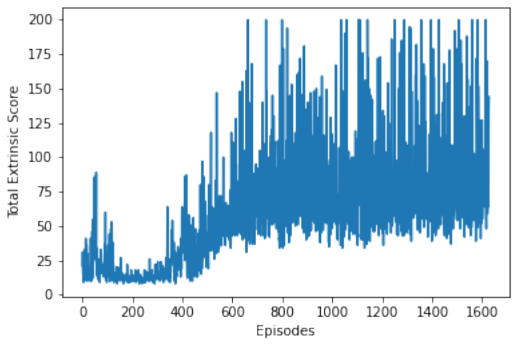


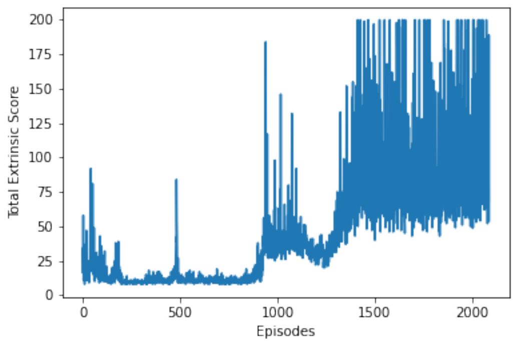
As one might infer from these few graphs, there was even less instances of ”learning” with this seed. Given the sparsity of good combinations combined with the uncertainty relating to the seed, we decided to re-run the experiment again on both seeds but for 100000 steps instead of 40000, as a final brute force effort to find some good hyper parameters to run for 1000000 steps. This experiment did indeed yield better results with the top six performing combinations showing signs of ”learning” and, even better some, only differ by seed:

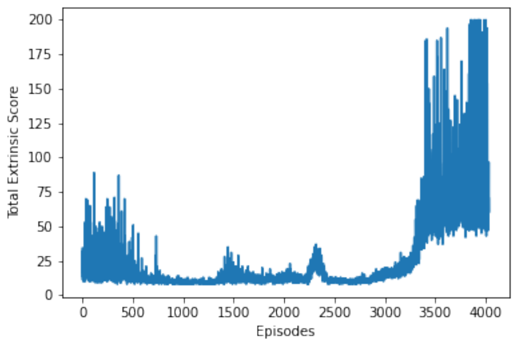












Due to computational resource constraints, we can only run a few combinations for 1000000 steps, so we limited our search to just the hyper parameters of the top six. The following hyper parameter values are present in at least one of the top six:

- k : 4, 16
- α : $0.003 = \frac{3}{e^3}, 0.001 = \frac{1}{e^3}$
- $\hat{d}$: 0.5, 0.9
- σ : 0.001

· PPO : off

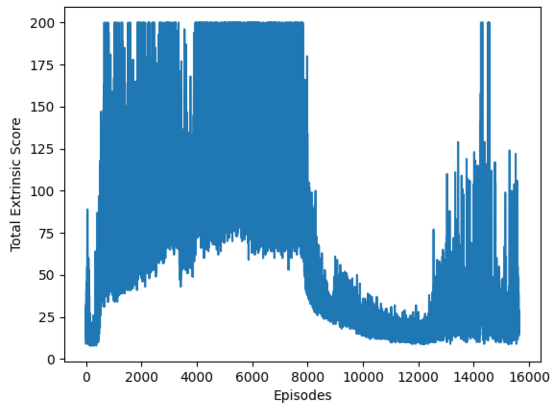
· ϕ : Tanh

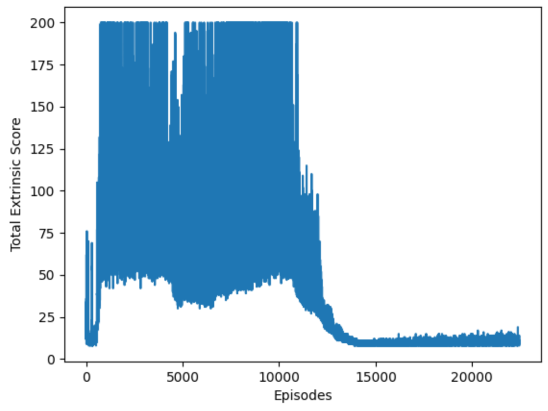
· random seed : 42, 420

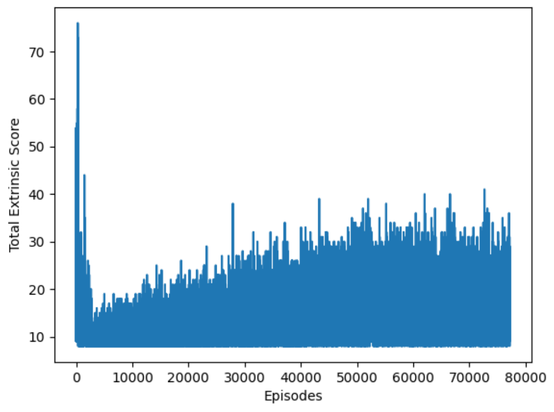
Using these 16 combinations, we ran our final experiment, of 1000000 steps for each hyper parameter combination. See Results sections for more.

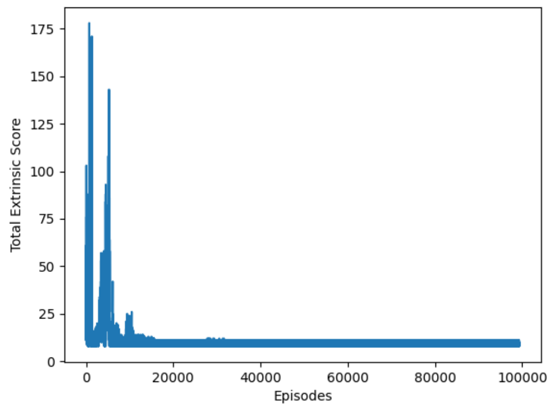
3 Results

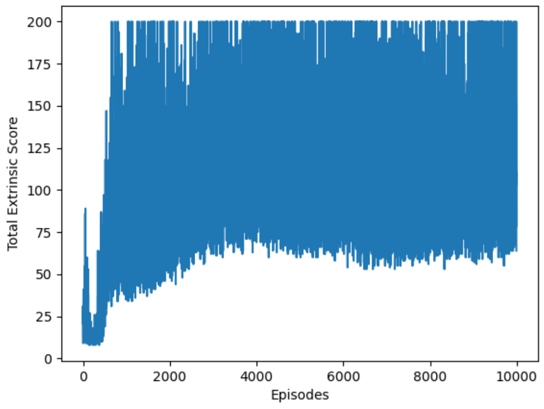
Below are the extrinsic reward graphs for the final set of experiments described at the end of Section 2.2. The left and right hand graphs on each row correspond to hyper parameter sets that only differ by the seed.

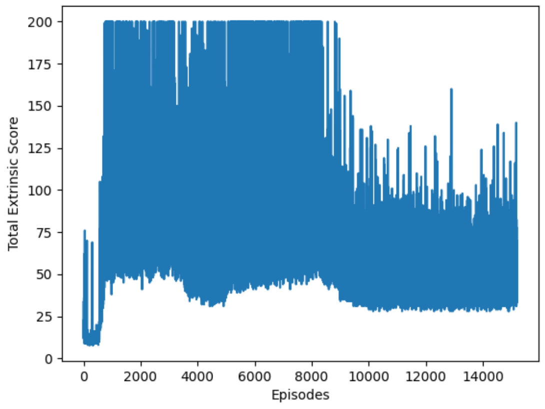


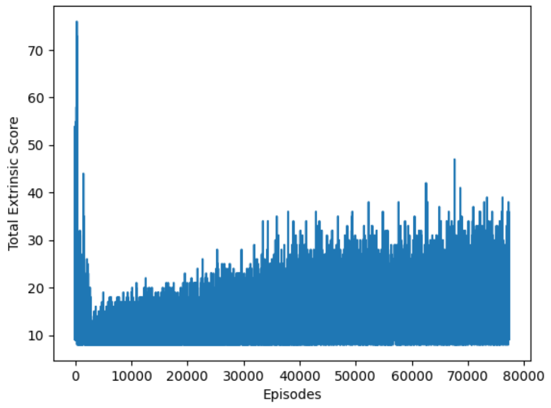


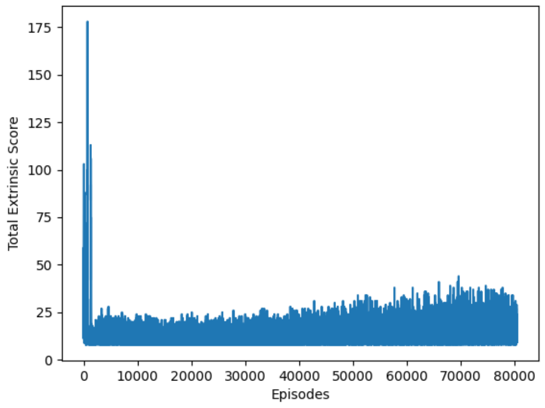


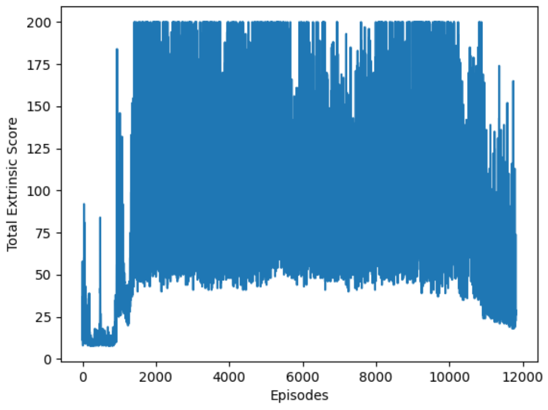


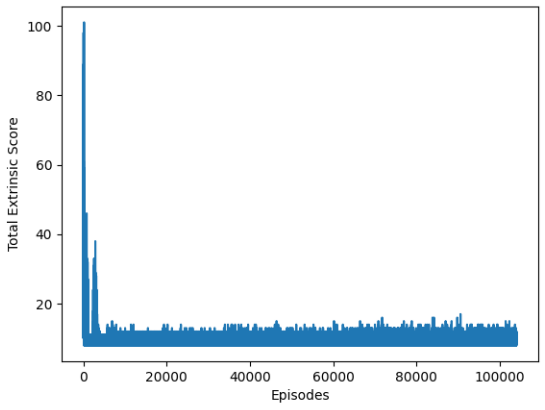

Total Extrinsic Score
Episodes

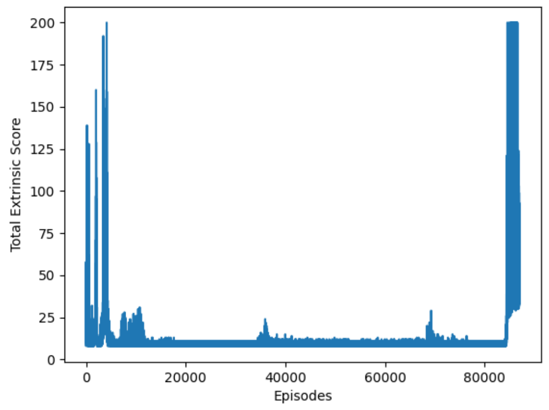

Total Extrinsic Score
Episodes

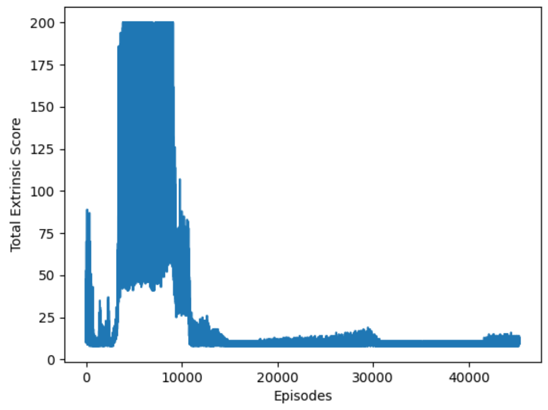

Total Extrinsic Score
Episodes

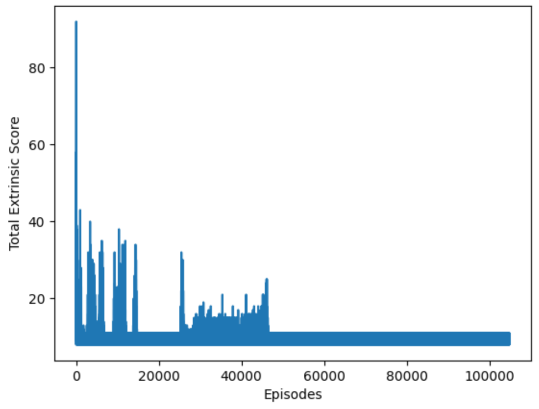

Total Extrinsic Score
Episodes

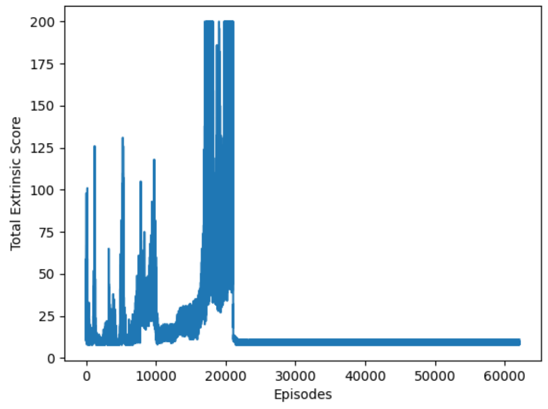

Total Extrinsic Score
Episodes

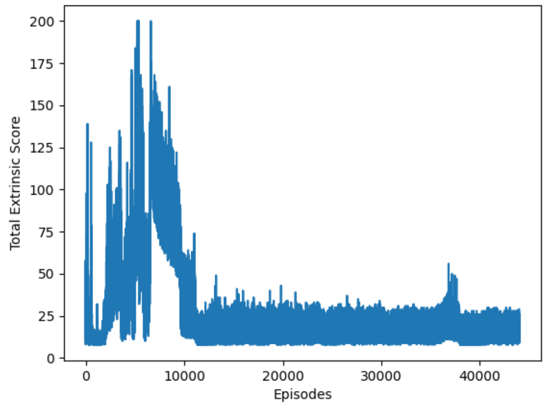

Total Extrinsic Score
Episodes

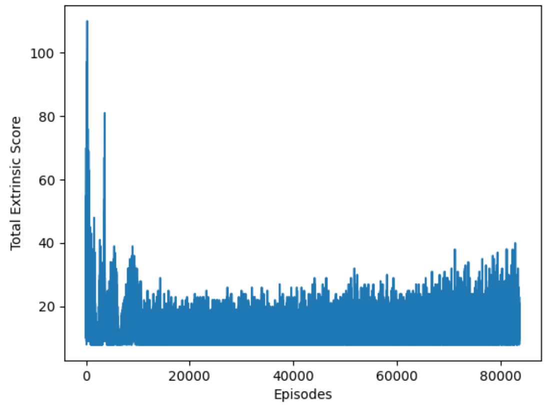












4 Discussion

To put these results into perspective an implementation of PPO we wrote earlier this year could reliably reach between 200 and 500 total extrinsic score in the same number of steps, so these results are worse then our baseline sadly. One glaring problem revealed by these results is many of the hyper parameter combinations increase in total extrinsic reward and then plummet back down; this is emblematic of a very common issue in deep learning known as "catastrophic forgetting" where continued optimization erases performance gains rather then continuing them. The PPO algorithm is supposed to help combat this issue; however our initial hyper parameter search ruled it out, and, given more time for this project, it would be a good idea to re run these experiments with PPO on. That being said, in our previous experience with PPO on less complex agents, we have observed that it can still be very hyper parameter sensitive and occasionally experience catastrophic forgetting, so it is not a perfect fix. Additionally these results illuminate the high variance of the LODA model as it experiences large swings in total score, an issue that could possibly be addressed through more rigorous probabilistic analysis of the model. Finally, the only hyper parameter combination that retained its gains is the third row, and, as can be seen in those graphs there isn't much continued learning, it just gets somewhere a bit better then it started and avoids catastrophic forgetting, so a follow up experiment would likely need to do more then just increase computational resources.

5 Conclusion

While our experiments may have ultimately not beaten our baseline it was still interesting to see the Actor Critic method working somewhat on our novel method for constructing the agent's behavioral policy; this combined with the fact most of the issues we encountered are quite common place in Reinforcement Learning, means we can definitely take heart in these less then stellar results. However, ultimately in the wider sphere of Reinforcement Learning, I think that recently emerging decision transformer methods – while requiring a lot more computation – are much more promising then derivatives of Actor Critic; in fact given more computational resources this paper would have been on a decision transformer derivative.

References

[1] Konda V, Tsitsiklis J (1999); Actor-Critic Algorithms; Neurips

[2] Weng, L. (April 8, 2018). Policy Gradient Algorithms. Lil'Log

[3] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. (2017). Proximal Policy Optimization Algorithms.. CoRR, abs/1707.06347.

[4] A Burda, Yuri; Edwards, Harrison; Storkey, Amos; Klimov, Oleg (2018); T Exploration by Random Network Distillation ; J arXiv e-prints; P arXiv:1810.12894

[5] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. ArXiv Preprint ArXiv:1606.01540.

Appendix

Scalar Map Approximation Using a Linear Operator

Suppose we have observed n inputs, $X^* = \{X_1, X_2, ..., X_n\} \subseteq \mathbb{R}^k$, to an unknown function $\phi : \mathbb{R}^k \to \mathbb{R}$, and their corresponding outputs $Y^* = \{Y_1, Y_2, ..., Y_n\}$, such that $Y_i = \phi(X_i)$. We would like to approximate ϕ using a linear operator represented by the vector $W \in \mathbb{R}^k$; this vector will be selected to minimize the following function:

$$MSE(X^*, Y^*, W) = \sum_{i=1}^{n} \frac{(Y_i - X_i^\top W)^2}{2n}$$

However since ϕ is likely non-linear and we may not be able to store all examples we have seen, we will bias toward recent examples (Where X_n is most recent and X_1 is oldest) to make the approximation more tractable; therefore we will add a decay term $0 < \gamma \leq 1$ and replace the $\frac{1}{2n}$ with $\frac{1}{2\sum_{j=1}^{n} \gamma^{n-j}}$. Consequently we have the decayed mean squared error :

$$DMSE(X^*, Y^*, W, \gamma) = \sum_{i=1}^{n} \frac{\gamma^{n-i}(Y_i - X_i^\top W)^2}{2\sum_{j=1}^{n} \gamma^{n-j}}$$

Observe that when $\gamma := 1$ the following equality holds:

$$DMSE(X^*, Y^*, W, 1) = \sum_{i=1}^{n} \frac{1^{n-i}(Y_i - X_i^\top W)^2}{2\sum_{j=1}^{n} 1^{n-j}} = \sum_{i=1}^{n} \frac{(Y_i - X_i^\top W)^2}{2n} = MSE(X^*, Y^*, W)$$

We would like to perform gradient descent on W, so we must first find the gradient of our loss:

$$\nabla_W DMSE(X^*, Y^*, W, \gamma) = \sum_{i=1}^{n} \frac{\gamma^{n-i}}{2\sum_{j=1}^{n} \gamma^{n-j}} \nabla_W (Y_i - X_i^\top W)^2$$

$$= \sum_{i=1}^{n} \frac{\gamma^{n-i}}{2\sum_{j=1}^{n} \gamma^{n-j}} 2(Y_i - X_i^\top W)(-X_i^\top) = \sum_{i=1}^{n} \frac{\gamma^{n-i}}{\sum_{j=1}^{n} \gamma^{n-j}} (Y_i - X_i^\top W)(-X_i^\top)$$

$$= \frac{1}{\sum_{j=1}^{n} \gamma^{n-j}} \sum_{i=1}^{n} \gamma^{n-i}(Y_i - X_i^\top W)(-X_i^\top) = \frac{1}{\sum_{j=1}^{n} \gamma^{n-j}} \sum_{i=1}^{n} \gamma^{n-i}((X_i^\top W)X_i^\top - Y_i X_i^\top)$$

$$= \frac{1}{\sum_{j=1}^{n} \gamma^{n-j}} \sum_{i=1}^{n} \gamma^{n-i}((W^\top X_i)X_i^\top - Y_i X_i^\top) = \frac{1}{\sum_{j=1}^{n} \gamma^{n-j}} \sum_{i=1}^{n} \gamma^{n-i}(W^\top (X_i X_i^\top) - Y_i X_i^\top)$$

Suppose we define $D_{(n)} = \frac{1}{\sum_{j=1}^{n} \gamma^{n-j}}$ giving us:

$$= D_{(n)} \sum_{i=1}^{n} \gamma^{n-i}(W^\top (X_i X_i^\top) - Y_i X_i^\top) = D_{(n)} \left[\sum_{i=1}^{n} \gamma^{n-i}(W^\top (X_i X_i^\top)) - \sum_{i=1}^{n} \gamma^{n-i}(Y_i X_i^\top) \right]$$

$$= D_{(n)} \left[W^\top \sum_{i=1}^{n} \gamma^{n-i}(X_i X_i^\top) - \sum_{i=1}^{n} \gamma^{n-i}(Y_i X_i^\top) \right]$$

We shall write these two decayed sums as parameterized variables (note: $H_{(n)}$ is the hessian matrix of $DMSE(X^*, Y^*, W, \gamma)$ with respect to W).

$$H_{(n)} = \sum_{i=1}^{n} \gamma^{n-i}(X_i X_i^\top)$$

$$T_{(n)} = \sum_{i=1}^{n} \gamma^{n-i}(X_i Y_i)$$

So we can write our gradient in terms of these values:

$$\nabla_W DMSE(X^*, Y^*, W, \gamma) = D_{(n)} \left[W^\top H_{(n)} - T_{(n)} \right]$$

When implementing this on a computer, we track $D_{(n)}^{-1}$ instead as it is also a simple decayed sum; these three decayed sum variables have the following update rules:

$$H_{(n+1)} := X_{n+1} X_{n+1}^\top + \gamma H_n$$

$$T_{(n+1)} := X_{n+1} Y_{n+1} + \gamma T_n$$

$$D_{(n+1)}^{-1} := 1 + \gamma D_{(n)}^{-1}$$