Reinforcement Learning Methods in an Atari Environment : CS 4803/7643 Spring 2020

Ramesh Arvind Ravi
Georgia Institute of Technology
North Ave NW, Atlanta, GA 30332
rameshar vind@gatech.edu

Angelo Petrolino
Georgia Institute of Technology
North Ave NW, Atlanta, GA 30332
apetrolino3@gatech.edu

Abstract

We analyze the performance of policy gradient oriented reinforcement learning algorithms in a sparse reward environment, specifically proximal policy optimization (PPO) and a modified approach that uses random network distillation (RND). Our experiments show the effectiveness of both algorithms in a discrete-action environment where the episodic rewards require a great deal of sparsely-rewarded exploration. We show both of these approaches are effective, easy to implement, and require little overhead.

1. Introduction/Background/Motivation

Our goal is to benchmark the performance of these two algorithms (PPO and RND) in a sparse reward environment in order to deepen a personal understanding of their structure and measure a final, trained agent against a human player in order to judge their effectiveness. So we will first reproduce models that have already shown to adapt to perform well [7, 5], train these models in our chosen environment, then compare their performance against each other.

In recent years, using reinforcement learning methods to train autonomous agents to play video games has seen an explosive growth in popularity. Breakthroughs were made using Deep-Q networks that showed promising results in the Atari 2600 game Pong [9]. More effective algorithms that relied on policy optimization rather than value optimization showed even greater promise to train autonomous agents with their robustness in ability to handle parallel implementations and their ability to scale to larger models. PPO was found to have more desirable benchmarks than deep Q-learning [7] and was the showcased approach for the OpenAI 5 [4] agent which was able to beat a professional team in the video game Dota 2 in 2019. However, this most recent approach required the utilization of thousands of GPUs over the course of several months which speaks to the limitations of reinforcement learning in complex environments.

For solving the issues of reward sparsity, heavy amount of reward shaping was used in OpenAI 5 with complex reward functions requiring human expertise. In general for sparse reward problems, human expertise is required to resolve the issue. To alleviate this problem of having to specify reward functions for such environments, better exploration strategies were developed such as adding a entropy loss coefficient which incentives policy gradient methods [7]. Better methods such as state pseudo-counts were considered state of the art at the time of RND being published. Pseudo counts maintained an approximate count on the number of times a state was visited with the aid of a neural network to act as a hashing function. These approaches achieved a score of 6600 on Montezuma's Revenge. RND was able surpass this with a score of 8100. More recently Go-Explore (another solution to the sparse reward problem) solved with a score of 18,003,200 [6].

Our approach is constrained in the domain of games with considerably smaller state spaces than Dota 2 due to financial, hardware, and time limitations. We wish to benchmark these two prevalent approaches due to their simplicity and we chose the Atari 2600 game Montezuma's Revenge for its sparse reward nature and ease of iteration through model tuning.

Success in our experiments will not only deepen a personal understanding of the field, but provide a cornerstone on which we can build the foundation to contribute to contemporary literature in a meaningful way. Sparse rewards are a problem that is found throughout in most complex problems. Considering games such as chess, go, or Dota 2. These are representative of the level of complexity agents are expected to solve in real life problems such as in robotics or fleet management. Better exploration strategies greatly simplify this problem for us allowing the agent to confidently explore new state spaces instead of getting stuck in some local sub-optimal policy.

Due to the nature of the actor-action-observation cycle inherent in reinforcement learning methods, the data we will use and collect is entirely coupled to the algorithms and environments we will employ. Thus, no external datasets will be used. The benchmark in which we will determine the success of the agent will be their rewards per episode during training. Those rewards are standardized across OpenAI's Gym library, which is our training environment.

2. Approach

2.1. Proximal Policy Optimization

Both of the following approaches build from the fundamental idea of the policy gradient. The way policy gradient methods work is by first computing an estimator of the policy gradient, then tuning the parameters with stochastic gradient ascent. The most commonly used stochastic gradient estimator is of the form

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta log \pi_\theta(a_t|s_t)\hat{A}_t]$$

where π_θ is the parameterized policy which will be represented by a neural network commonly called an 'actor', and $\hat{A}_t$ is an estimator of the advantage function at timestep t that inherits part of its final value from another neural network, commonly called a 'critic' (V_θ). The expectation $\mathbb{E}_t$ is the average over a batch of samples that switches between sampling and optimization. Using this requires making an loss function with a gradient that is equivalent to the policy gradient estimator $\hat{g}$ which can be found by differentiating the function

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[log \pi_\theta(a_t|s_t)\hat{A}_t]$$

It is shown in a previous study[7] that performing this optimization on L^{PG} in batches can yield sub par results. A further improvement on this loss function was found in the study if one casts the reward as a ratio from adjacent policy updates and rewrite the loss function as

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t]$$

However, it was found that this loss function can sometimes lead to massive gradient steps leading to unstable training due to the potential of $r_t(\theta)$ to explode after certain parameter updates. Much like Integral-clamping for PID controllers in classical control theory, a boundary can be fitted to the reward to give a loss function in the form of

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon))\hat{A}_t]$$

where ε is some clipping constant. This form of PPO is known as PPO-clip and will be the approach we will take in this paper as we believe a stable learning approach will benefit us in tackling the sparsity of reward in Montezuma's Revenge.

Algorithm 1 PPO-Clip pseudocode

```
$T \leftarrow$ number of steps per episode
$M \leftarrow$ size of mini-batch
$K \leftarrow$ number of epoch
$N_{opt} \leftarrow$ number of optimization steps
for episode $= 1, 2, ...$ do
    for $t = 1$ to $T$ do
        Sample $a_t \sim \pi_\theta(a_t|s_t)$
        Sample $s_{t+1}, r_t \sim p(s_{t+1}, r_t|s_t, a_t)$
        Add $s_t, p(a_t|s_t), a_t, r_t$ to optimization batch $B_i$
    end for
    for $j = 1$ to $N_{opt}$ do
        Compute advantage $\hat{A}_j$ from $V_\theta$, and reward-to-go
        Optimize surrogate $L^{CLIP}$ wrt $\theta$, $K$ times and batch
        size $M \leq TN_{opt}$ using Adam
        $\theta \leftarrow \theta_{old}$
    end for
    Clear $B$
end for
```

2.2. Random Network Distillation

Current policy gradient algorithms suffer from poor exploration strategies. Even with entropy regularization it cannot perform well in large state spaces with sparse rewards. The agent would just be stuck in some state space never being able to consistently explore new space. Other off-policy based algorithms such as deep Q-learning utilize simple methods such as ϵ-greedy methods which are not sufficient for extremely large state spaces.

Random network distillation proposes a mechanism to allow reinforcement models with an additional reward signal for exploring new state spaces previously unvisited. It performs this with two additional neural networks: A predictor and target network in addition to the ones required to output a policy. These two networks require the next state being visited by the agent and assess its novelty. Each of these networks output a fixed dimensional output for the state, the mean square error between the outputs of the predictor network and the target network represent the intrinsic bonus awarded to the agent for visiting that state. While back-propagating the network, only the predictor network is updated and the target network is fixed/frozen. Essentially as time progresses, the predictor networks tries to get closer to the target network, thus the name random network distillation (random from the target network being initialized randomly). Since the output from target network is deterministic, the predictor network will eventually converge to the same value. When exploring and visiting a new state, the difference between the target and predictor networks will be high thus granting a larger reward for it encouraging more exploration to this state.

Algorithm 2 RND pseudocode

```
$N \leftarrow$ number of rollouts
$N_{opt} \leftarrow$ number of optimization steps
$K \leftarrow$ length of rollout
$M \leftarrow$ number of initial steps for initializing observation normalization
Sample state $s_0 \sim p_0(s_0)$
**for** $m = 1$ **to** $M$ **do**
    Sample $a_t \sim$ Uniform$(a_t)$
    Sample $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
    Update observation normalization parameters using $s_{t+1}$
**end for**
**for** $i = 1$ **to** $N$ **do**
    **for** $j = 1$ **to** $K$ **do**
        Sample $a_t \sim \pi_\theta(a_t|s_t)$
        $s_{t+1}, e_t \sim p(s_{t+1}, e_t|s_t, a_t)$
        Find intrinsic reward $i_t = \left\| \hat{f}(s_{t+1}) - f(s_{t+1}) \right\|^2$
        Add $s_t, s_{t+1}, a_t, e_t, i_t$ to optimization batch $B_i$
        Update reward normalization parameters using $i_t$
    **end for**
    Normalize the intrinsic rewards contained in $B_i$
    Calculate $R_{I,i}$ and $A_{I,i}$ for intrinsic rewards
    Calculate $R_{E,i}$ and $A_{E,i}$ for extrinsic rewards
    Calculate combined advantages $A_i = A_{I,i} + A_{E,i}$
    Update observation normalization parameters using $B_i$

    **for** $j = 1$ **to** $N_{opt}$ **do**
        Optimize $\theta_\pi$ wrt PPO loss on batch $B_i, R_i, A_i$ using Adam
        $\theta \leftarrow \theta_{old}$
        Optimize $\theta_{\hat{f}}$ wrt distillation loss on $B_i$ using Adam
        $\theta_{\hat{f}} \leftarrow \theta_{\hat{f}_{old}}$
    **end for**
**end for**
```

Additionally in the PPO implementation with RND, instead of simply combining regular rewards and the intrinsic rewards, having a separate value head for intrinsic and regular rewards helps better decouple the reward functions ensuring better gradients to the network. A high level explanation of RND can be seen on Algorithm 2

2.3. Anticipated and Initial Difficulties

Current literature suggests large training times for certain environments which was our first anticipated problem. Further, Montezuma's Revenge is known for such sparse rewards such that an agent could cycle through hundreds of episodes with no reward leading to ineffective training. As a baseline, both the RND and PPO agent was trained in the Atari 2600 game Freeway where the rewards are still

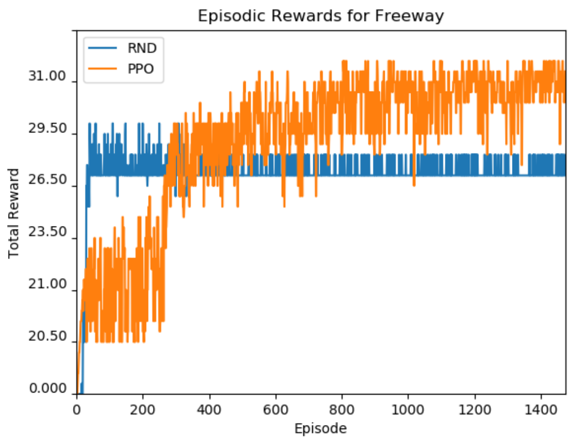


Figure 1: The total rewards gained per episode for PPO and RND agents in Freeway

considered sparse [3] but the action space is a fraction of the size of Montezuma's Revenge to justify our model's architectures before continuing. Another advantage of using Freeway was that due to the ease of iteration through test runs, we were able to more quickly swap around certain layers and tune certain hyperparameters in an expedient fashion. We began modeling in PyTorch and our first attempts at Freeway were met with the input layers of our networks being fully connected layers which resulted in unsatisfactory training coupled with low rewards. A short attempt at an RNN as the input layer proved more effective, but convolution nets provided the best results. Figure 1 shows our breakthrough test results before we moved on to Montezuma's Revenge.

3. Experiments and Results

Our metric for these agents is their total collected rewards versus number of parameter updates for the agent's respective network. We tried recreating the results from the original RND paper by following a similar architecture. Our models comprise of three networks. A predictor and target network for the RND component and an actor network for providing the PPO policy. All the experiments were conducted with the pytorch library.

We conducted multiple experiments along with iterations on fixing bugs:

3.1. PPO

The PPO agent has 3 layers of convolutional blocks with strides of $4, 2, 1$, kernel size of $8, 4, 3$ and $32, 64, 32$ filters respectively followed by a fully connected block. The head of both the actor and critic networks were each connected to an additional fully connected block with outputs respective to the dimensions of the environment.

This network successfully solves basic OpenAI Gym environments such as Cartpole, Lunar Lander, and Mountain Car. In the final tests on Freeway, this structure also showed better performance in the sparse reward environment Freeway. Loss function clipping was adjusted from and range of 0.1 to 0.8 with the most consistent results at 0.2.

3.2. PPO with RND block

This network consists of a similar structure to the PPO. However, the convolutional blocks now have strides of $4, 3, 2$, kernel sizes of $8, 4, 4$, and filters of $32, 64, 128$ respectively. Additionally, there is an average pooling layer following the last convolutional layer. Lastly, an additional RND predictor and target components with the same convolutional blocks (but the predictor network has one additional fully connected layer). The predictor/target network outputs a 256 dimensional vector for the state input, but the target networks weights are fixed.

This version of the code was still able to converge for the simple environments and additionally converged for Freeway albeit worse (very slightly) than regular PPO. It did converge much faster than regular PPO to its highest value. We suspect this could be due to poor hyperparameter selections due to lack of time we chose not to explore tuning this further for freeway. But this version of the code did not converge for Montezuma's revenge even with 20 hours of training time.

We qualitatively evaluate if the model has scope of convergence by analysing the intrinsic reward graph. It should have periodic spikes in the graph indicating exploration of new state spaces and otherwise being on a downward trend. Our initial implementation was always on a downward trend and never had any spikes, indicating an issue in implementation.

3.3. Model Adjustments

There were a lot of tricks required (based on OpenAI's original work) to improve the convergence of our models. The first was to train a number of environments in parallel. Essentially we collect multiple trajectories to increase the available samples to train the PPO/RND model at every iteration. We use gym.vector function to implement this feature and also modify our model architecture to handle batch of trajectories instead. Additionally, we had access to a machine with two NVIDIA GTX 1070 GPU cards to utilize this parallelization. Secondly, there were multiple bugs in the implementation that had to be fixed with respect to calculating the PPO loss.

Even with parallel environments, the batchsize that was used to update the network was too large. It is known that using very large, unshuffled batchsize causes multiple gradients to point in the same direction. Thus, we implemented shuffling of the trajectory samples and used batch size of 4096 per update which led to stable updates. We then implemented reward and state normalization. It has been shown that these normalization steps are key for policy gradient algorithms to converge [8]. For RND, with a large update batch size, we didn't want the predictor/target convergence to be too fast rendering it useless for exploration. We dropped 75% of the updates to it randomly while calculating the loss to maintain good exploration rate.

The environment that was used was the no-frame-skip flavor of Montezuma's Revenge (version 4), but it was shown[5] that the model requires frame skipping with sticky actions (repeating an action from previous frame with some probability). Additionally the state space was extended to incorporate a history of previous 3 frames to allow the agent to understand the effects of motion. We also rescaled the state image to (84, 84) from the original (210, 160) as well as grayscaled it to reduce the channels from 3 to 1. After

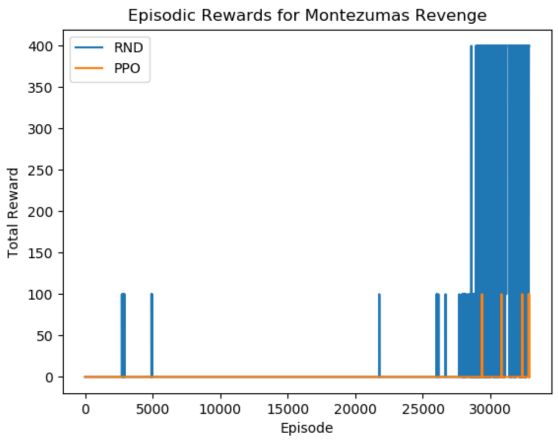
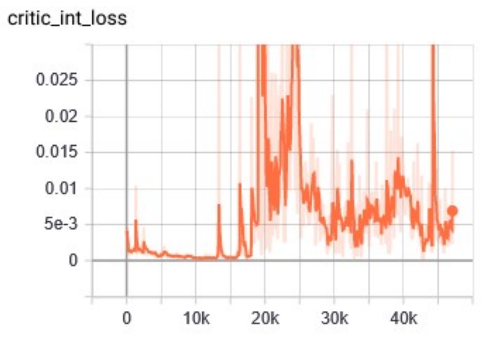


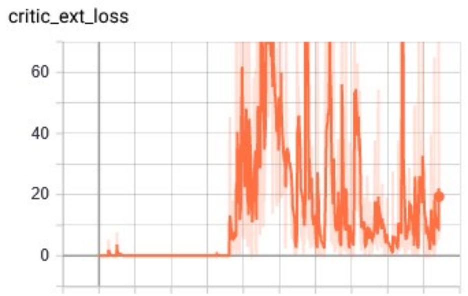
Figure 2: The total rewards gained per episode for PPO and RND agents in Montezuma's Revenge

performing the above updates, the network's training graphs looked stable and presented results similar to what was intended. But again the network failed to produce any results with respect to solving the environment.

The last set of changes made were to the architectures themselves. The most noticeable difference was the average pooling layers instead of flatten the last convolutional output as well as larger fully connected layers (from size 256 to 512). This essentially implies that the representation power of our network was inferior compared to what was used by OpenAI. It has also been shown that average pooling learns smooth features instead of sharper features which are better for memorization purposes. We believe these two changes were critical to our models convergence [2].



(a) Intrinsic loss



(b) Extrinsic loss

Figure 3: The x-axis represents number of parameter updates and the y-axis represents the loss values. Spikes in the intrinsic loss coincide with major spikes in the extrinsic loss as well indicating the visitation of newly explored regions in the state space

.

After performing these updates to the model, we allowed the model to train for around 12 hours. We observed the trends we expected (such as periodic spikes in the intrinsic loss of the RND model) indicating exploration of new state spaces as shown in Figure 3. The model was able to achieve positive score of 400 consistently. Which is a strong indicator that our implementation was correct considering how most algorithms fail to score a positive reward altogether. With these final adjustments we were able to obtain the results in Figure 2.

4. Conclusion

We see that though PPO was able to learn effectively in smaller environments such as Freeway, RND is able to outclass it in more challenging environments such as Montezuma's Revenge. This provided us with a better understanding of policy gradient algorithms and their downsides in sparse reward environments. We also understood how RND can help alleviate this problem and understood how to interpret the RND training process. Translating results from toy environments to larger environments such as Montezuma's Revenge also provided us with an experience with scaling experiments to a large scale such as training multiple environments parallel on multiple GPUs. This provides us with the experience to move on to other papers which have larger computational requirements for replication.

This project also provided us an introduction to research in exploration techniques in reinforcement learning allowing us to better understand state of the art papers in this area such as Go-Explore. We would also like to thank the authors of the original PPO/RND papers (and their code) as well as the the author of the repository[1] on GitHub whose hyperparameters were used as reference along with understanding the tricks used to produce better results. A distribution of work can be seen on Table 1.

References

[1] jcwleo's rnd implementation @ https://github.com/jcwleo/random-network-distillation-pytorch. 5

[2] Maxpooling vs minpooling vs average pooling https://medium.com/@bdhuma/which-pooling-method-is-better-maxpooling-vs-minpooling-vs-average-pooling-95fb03f45a9. 4

[3] Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. Unifying count-based exploration and intrinsic motivation. *CoRR*, 1606:01868. 3

[4] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *arXiv*, 1912:06680, 2019. 1

[5] Yuri Burda, Harrison Edwards, Amos J. Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv*, 1810:12894. 1, 4

[6] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *CoRR*, 1901:10995. 1

[7] Prafulla Dhariwal Alec Radford John Schulman, Filip Wolski and Oleg Klimov. Proximal policy optimization algorithms. *arXiv*, 1707:0645. 1, 2

[8] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. 4

Student Name	Contributed Aspects	Details
Ramesh Arvind Ravi	RND Architecture	Built, trained, tested RND agent. Project analysis.
Angelo Petrolino	PPO Architecture	Built, trained, tested PPO agent. Project analysis.

Table 1: Contributions of team members.

[9] Kavukcuoglu K. Silver D. et al. Mnih, V. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015. 1