DQN and PPO - A comparison of popular value and policy gradient RL models

Joshua Kim, Sina Abady, Vedant Shah
University of Toronto
{jhr.kim, sina.abady, vedant.shah}@mail.utoronto.ca

Abstract

In this report we will compare two Reinforcement Learning (RL) Algorithms, Deep Q Networks (DQN) and Proximal Policy Optimization (PPO). Specifically, we will compare the methods on a number of OpenAI's Gym (gym) games. We explore benefits and drawbacks of the methods on the same games, as well as well known modifications to overcome challenges unique to the environments.

1 Introduction and Related Works

1.1 Deep Q-Network

There are two major branches of RL: value and policy gradient methods. The first algorithm we will explore is a DQN, a value-based method. The goal of general q-learning is to learn a parameterized approximator function $Q_\theta(s, a)$ for the optimal function $Q^*(s, a)$. For DQN, this parameterization is the weights and biases of a neural network. Then, given a state s, the policy will find $a^* = argmax_a Q_\theta(a, s)$.

For the implementation of DQN, our structure is largely based on 'Playing Atari with Deep Reinforcement Learning (Mnih, Kavukcuoglu, et al. 2013)'. This was the first paper outlining an RL capable of playing Atari games. For hyperparameter tuning, we refer to 'Hyperparameter Optimization for Deep Reinforcement Learning in Vehicle Energy Management (Liessner et al. 2019)'. Finally, to have our agent be robust to policies involving multiple actions at a given step, we use methods outlined in 'Action Branching Architectures for Deep Reinforcement Learning (Tavakoli et al. 2019)'.

1.2 Proximal Policy Optimization

The second framework we study is a policy gradient method. The goal of these methods is to find an optimal map directly from state to action. Unlike value based models, policy gradient models are robust to stochastic and continuous actions spaces. In more exact words, the goal of this algorithm is to find a policy $\pi_\theta(a|s)$, whose parameters θ are optimized using gradient ascent on a performance objective J (or a local approximation). Since these optimizations are performed on-policy, they only update using data collected while acting under the most recent policy, which can lead to some unique challenges.

To implement PPO, we rely on the structure suggested in 'Proximal Policy Optimization Algorithms (Schulman et al. 2017)', a paper which introduces the algorithm. We use generalized advantage estimation (Schulman et al. 2018), and approximation methods for our reward/advantage function 'Policy Gradient Methods for Reinforcement Learning with Function Approximation (Sutton et al. 1999)'. We also explore a TRPO algorithm based on 'Trust Region Policy Optimization (Schulman et al. 2017)', which provides robust performance for optimizing neural networks and other non-linear functions.

2 Method and Algorithm

2.1 DQN with Experience Replay

```
procedure DQN
    Initialize memory $D$ to size $N$
    Initialize action-value $Q_\theta$ with random parameters $\theta$
    Initialize target action-value $\hat{Q}_{\hat{\theta}}$ with parameters $\hat{\theta} = \theta$
    for episodes $j = 1 : M$ do
        $s_1 \leftarrow \{x_1\}$
        $\phi_1 \leftarrow \phi(s_1)$
        store transition into $D$
        for $t = 1 : T$ do
            select a random action $a_t$ with probability , otherwise,
            $a_t \leftarrow argmax_a Q_\theta(\phi(s_t), a)$
            observe $r_t$ and $x_{t+1}$ given $a_t$
            $s_{t+1} \leftarrow s_t, a_t, x_{t+1}$
            $\phi_{t+1} \leftarrow \phi(s_{t+1})$
            store transition in $D$
            sample minibatch ($j = 1 : J$)
            if episode terminates at next step $y_j \leftarrow r_j$, otherwise,
            $y_j \leftarrow r_j + \gamma \max_{a'} \hat{Q}_{\hat{\theta}}(\phi_{j+1}, a')$
            perform gradient descent on $L = (y_j - Q_\theta(D_j))^2$ w.r.t $\theta$
            if t % C = 0 then▷ Every C steps
                $\hat{Q} \leftarrow Q$
            end if
        end for
    end for
end procedure
```

We use a Convolution network with the structure. The images are scaled down in dimension and set to grayscale to reduce the number of parameters needed in the network. The frames input to the network are then stacked (stochastically or deterministically) to promote velocity (change) detection of the images.

We support multi-choice action spaces. We take actions that were above the mean of all Q values that the network provided for a given state. This is not as accurate as embedding each action into its own layer (Tavakoli et al. 2019), but for discrete multi actions, this is time efficient and far more effective than taking a single action each frame for mutli-action spaces. This provides a simple way to improve the performance of DQNs on games with mutli-variable action spaces without increasing the complexity of the underlying networks.

2.2 PPO

```
 1: procedure PPO($\theta_0, \phi_0$)
 2:     for k=1:K do
 3:         Collect $D_k = \{\tau_i\}$ by sampling $\pi_k = \pi(\theta_k)$
 4:         Compute reward-to-go $\hat{R}_t$
 5:         Compute Advantage Estimates , $\hat{A}_t$ based on current value function $V_{\phi_k}$
 6:         maximize PPO-Clip objective,
 7:         $\theta_{k+1} \leftarrow argmax_\theta$
 8:         Fit $V$ by performing gradient descent on MSE, $(V_\phi(s_t) - \hat{R}_t)^2$
 9:     end for
10: end procedure
```

There are two branches of PPO. The first, PPO-Penalty, uses KL-Divergence like TRPO, but penalizes in the objective function instead as a constraint. The second, which is what we implement, is PPO-Clip. This uses clipping to remove incentives for the new policy to deviate too far away from the old policy. We can represent the surrogate objective as referenced (Schulman et al. 2017):

$$L_{\theta_k} = \min(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}, \quad clip(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, 1-\epsilon, 1+\epsilon) A^{\pi_{\theta_k}})$$

Then the operations on lines 7 and 8 can be expanded as:

$$\theta_{k+1} \leftarrow argmax_\theta \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{0 \leq t \leq T} \min(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)))$$

$$\phi_{k+1} \leftarrow argmin_\phi \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{0 \leq t \leq T} (V_\phi(s_t) - \hat{R})^2$$

For this report, we define the advantage function, named Bellman error, as $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$, where $Q^\pi(s_t, a_t)$ is the Q-function and $V^\pi(s_t)$ is the on-policy value function. The function can be simplified into $A^\pi(s_t, a_t) = r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$ (refer Appendix for derivation). This allows credit assignment to action space, which could not be done via reward function. The drawback of this approach is that 2 sets of network parameter, θ for policy and ϕ for value function, need to be trained.

3 Discussion and Experiments

3.1 DQN

Below is the comparison in survival time from using the mean-action heuristic compared to the standard Q action heuristic. (Figure 1 and 2). Similarly see (Figure 9 and 10) for relevant reward graphs.

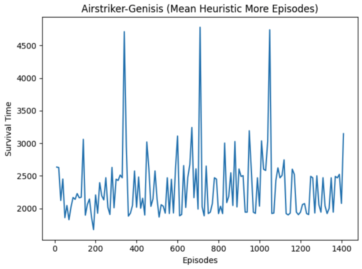

Figure 1: Mean action heuristic beats game thrice

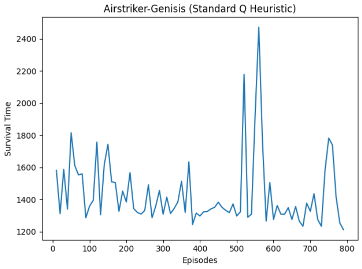

Figure 2: Standard Q Heuristic underperforms

By tuning replay buffer size, batch size and the target network update frequency, we can adjust the short vs long term memory of our agent. We find the small replay memory lead to "catastrophic forgetting" and tend to overfit the memory available. In our experimentation, this resulted in many occurrences of the DQN being able to beat the game at some iterations, but never converging to a consistent result (Figure 1). On the other hand, large replay memory sizes are highly prone to underfitting and require an immense amount of training time and cannot converge to a policy to beat the level.

Our best performing Airstriker agent had a small memory size and large batch size, but a very infrequent target network update. The agent behaves consistently– in our case sticking to the right side of the screen (Figure 5)– and survives longer, but never explores the space of the game effectively. To combat this we increase the replay memory size, and the agent scored far more points. However, the agent never beats the level due to the underfitting caused by this change.

From the lessons learned from training Airstriker, we chose more ideal hyperparameters (Table 2) for Riverrider and Skiing.

We then apply the DQN to standard Discrete Action Spaces on the games Skiing and Riverraid. They trained with large memories and high target network update frequencies with smaller batch sizes. This lead them to converge steadily to a result due to the continuous and frequent batch training of observations (Figure 6). However, this limited its exploration, as epsilon was set to rapidly decrease given limited episodes. Given more computational power and time, ideally these games are played for longer, with a larger epsilon to produce consistent improvements and better convergence through adequate exploration.

3.2 PPO

There are a number of experiments and tweaks we made to the PPO algorithms to improve performance/reduce training time. The first experiment was CLIP vs no CLIP. It is known that PPO without Clipping can save computation time and still maintain a reasonable trust region (Anonymous 2019). We also applied clipping to the value function. Originally, PPO only clips the policy network, but we apply the same logic to the value network as well (Schulman et al. 2017), to get the following objective:

$$L^V = \min[(V_{\theta_t} - V_{targ})^2; (clip(V_{\theta_t}, V_{\theta_{t-1}} - \epsilon, V_{\theta_{t-1}} + \epsilon) - V_{targ})^2]$$

As seen in Figure 3 and 4, PPO with clipping demonstrates a more stable training than PPO without clipping as the fluctuation in the reward plot diminishes as training timestep increases. Thus, PPO with clipping has been selected for this report.

We also experimented with different advantage functions. While Bellman error function is commonly implemented for policy gradient models, a simple reward-to-go system $\hat{R}_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})$, where the function is a discounted sum of future rewards, can instead be used. This can especially be useful when the action space is continuous or the computation resource is limited. Since games with discrete action space are only considered for the purpose of this report however, the originally suggested advantage function was implemented.

Policy gradient methods are on-policy whereas value methods are off-policy, hence it is sample inefficient and only uses sampled trajectories once. Policy gradient can be a more stable method since it directly updates the policy. PPO provides a better convergence and performance rate than DQN, but is sensitive to changes and initial conditions. Hence, hyperparameter tuning can be more challenging. Refer to Table 3 for hyperparameters selected for PPO experiments.

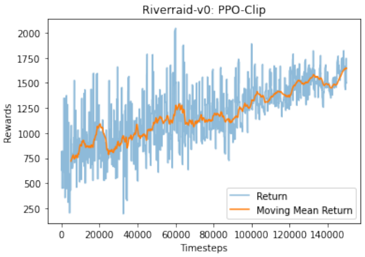

Figure 3: Training plot of rewards using PPO with clipping

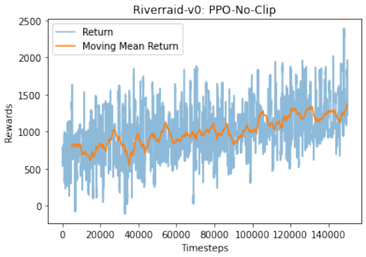

Figure 4: Training plot of rewards using PPO without clipping

3.3 Comparison

Riverraid and skiing are played well by the DQN agent after training, but nowhere near perfect as would be expected from a Deep Reinforcement Agent, and PPO tends to excel in both training time, and produces higher scores for these two games in particular.

Table 1: Score by algorithm

Game	DQN Score	PPO Score
Skiing	-17350	-15396
Riverrider	1390	1622

See Figures 6 and 7 the training curves of the DQN agent on both games and Figures 3 and 8 for the training curve of PPO on both games

4 Conclusion

When using DQN, we would ideally use the hyperparameters researched at Deepmind, and provide a large memory size with extremely frequent target network updates. However this would take an exorbitant amount of time for even a programmer with good hardware availability. The best performing models can take hours to run and only after the ideal hyperparameters are tuned for each game. This means that when experimenting the implementation of the DQN it takes too long to find the optimal hyperparameters and train them without a deep expertise of the problem space. A better solution is PPO, an on-policy algorithm which can remedy the training time problem, with ability for multi-processing.

PPO trains a stochastic on-policy. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. We ran into this issue, causing the policy to get trapped in local optima. Introducing noise to network parameters, analogous to Natural Evolution Strategies (NES), to this PPO may improve performance through better exploration (Li et al. 2019).

5 Git Repo

https://github.com/SinaAb/CSC413Final

6 Participation

Sina: Implementation and Tuning of DQN
Joshua: Implementation and Tuning of PPO
Vedant: Preliminary research and paper discussion

7 Additional Derivations

7.1 Bellman Error Advantage Function Simplification

Action-value function, also known as the Q-function, is defined as:

$$Q^\pi(s,a) = \mathbb{E}[G_t \mid s_t = s, a_t = a]$$

and the relationship between the action-value function and the value function is:

$$V^\pi(s) = \sum_a \pi(a \mid s) Q^\pi(s,a)$$

The Bellman Equation is a recursive formula such that:

$$\begin{aligned} Q^\pi(s,a) &= r(s,a) + \gamma \mathbb{E}_{p(s'|s,a)\pi(a'|s')}[Q^\pi(s' \mid a')] \\ &\approx r(s,a) + \gamma V^\pi(s_{t+1}) \end{aligned}$$

Therefore, the Bellman Error Advantage function can be simplified into:

$$A^\pi(s_t, a_t) = r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$$

7.2 PPO-Clip Objective Simplification

Refer to the following link: https://drive.google.com/file/d/1PDzn9RPvaXjJFZkGeapMHbHGiWWW20Ey/view

8 Citations, figures, tables, references

The structure for the Deep Q Network is seen below, sourced from the Deepmind Atari Paper

```
FullConnected(OBSERVATION SPACE SHAPE)
Convolution(32, 8, strides=4, activation="relu")
Convolution(64, 4, strides=2, activation="relu")
Convolution(64, 3, strides=1, activation="relu")
FullyConnected(512, activation="relu")
FullyConnected(ACTION SPACE SIZE)
```

The policy network structure for the Proximal Policy Optimization (PPO) is seen below:

```
FullConnected(OBSERVATION SPACE SHAPE)
Convolution(32, 8, strides=4, activation="relu")
Convolution(64, 4, strides=2, activation="relu")
Convolution(64, 3, strides=1, activation="relu")
FullyConnected(512, activation="relu")
FullyConnected(128, activation="relu")
FullyConnected(128, activation="relu")
FullyConnected(128, activation="relu")
FullyConnected(ACTION SPACE SIZE)
```

The value network structure for the Proximal Policy Optimization (PPO) is seen below:

```
FullConnected(OBSERVATION SPACE SHAPE)
Convolution(32, 8, strides=4, activation="relu")
Convolution(64, 4, strides=2, activation="relu")
Convolution(64, 3, strides=1, activation="relu")
FullyConnected(512, activation="relu")
FullyConnected(32, activation="relu")
FullyConnected(32, activation="relu")
FullyConnected(ACTION SPACE SIZE)
```

Table 2: DQN Hyperparamters

Game	Hyperparameter	Value	Game	Hyperparameter	Value
Airstriker	α	0.01	Riverraid/Skiing	α	0.000625
Airstriker	γ	0.995	Riverraid/Skiing	γ	0.987
Airstriker	ϵ	0.999	Riverraid/Skiing	ϵ	0.994
Airstriker	Target Update	500	Riverraid/Skiing	Target Update	500
Airstriker	Memory Size	5×10^3	Riverraid/Skiing	Memory Size	100×10^3
Airstriker	Replay Counter	500	Riverraid/Skiing	Replay Counter	128
Airstriker	Batch Size	300	Riverraid/Skiing	Batch Size	32
Airstriker	Loss Function	MSE	Riverraid/Skiing	Loss Function	MSE

Table 3: PPO Hyperparamters

Game	Hyperparameter	Value
Riverraid/Skiing	Horizon	128
Riverraid/Skiing	Adam stepsize	$2.5 \times 10^{-4} \times \alpha$
Riverraid/Skiing	Epoch number	3
Riverraid/Skiing	Minibatch size	128
Riverraid/Skiing	Discount factor (γ)	0.99
Riverraid/Skiing	Clipping parameter (ϵ)	0.2

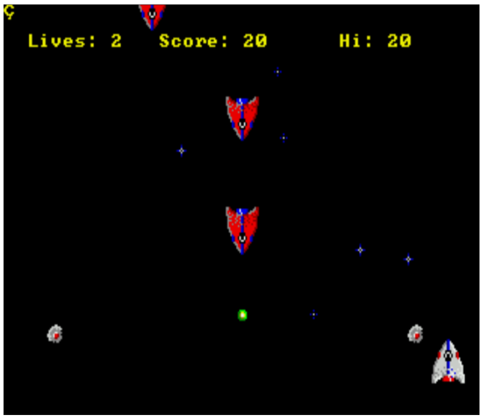

Figure 5: Our DQN agent sticks to the right

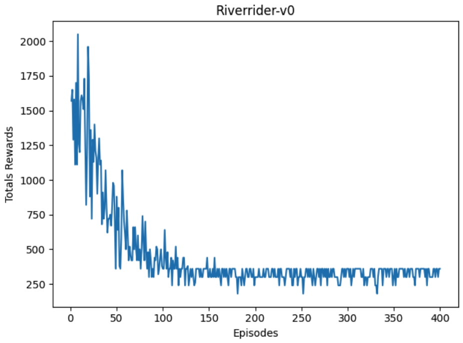

Figure 6: DQN far better convergence for Riverrider

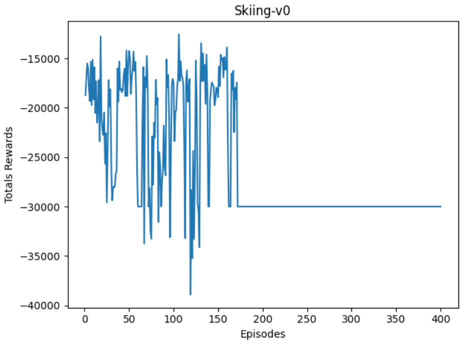

Figure 7: DQN Training curve for Skiing-v0 converges to a constant score

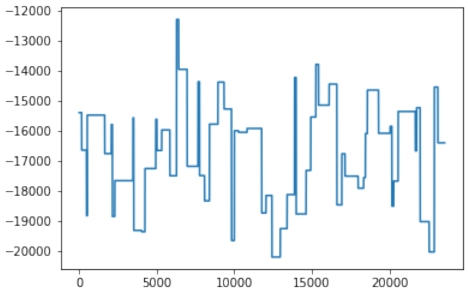

Figure 8: PPO Training curve for Skiing-v0 (with clip)

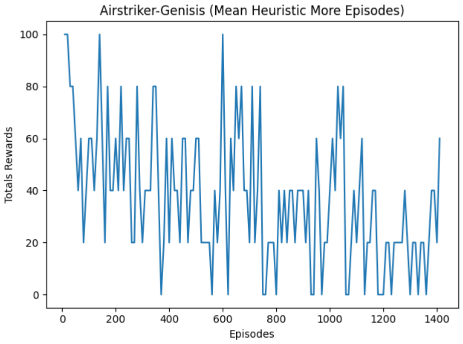

Figure 9: DQN Training curve of rewards given the mean action heuristic. Survival time mentioned in the DQN section above paints a better picture of the success of the algorithm.

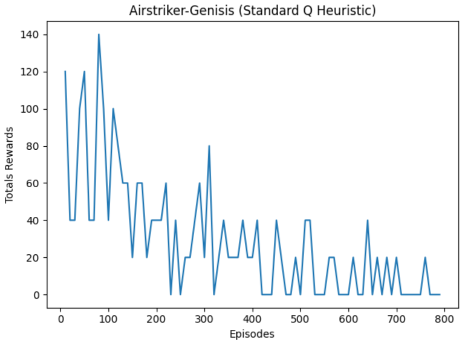

Figure 10: DQN Training curve of rewards given the standard Q action heuristic. Survival time mentioned in the DQN section above paints a better picture of the success of the algorithm.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning, 2013; arXiv:1312.5602.

[2] Roman Liessner, Jakob Schmitt, Ansgar Dietermann and Bernard Baker. Hyperparameter Optimization for Deep Reinforcement Learning in Vehicle Energy Management, 2019;

[3] Arash Tavakoli, Fabio Pardo and Petar Kormushev. Action Branching Architectures for Deep Reinforcement Learning, 2017, AAAI 32: 4131-4138 (2018); arXiv:1711.08946.

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017; arXiv:1707.06347.

[5] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation, 2015; arXiv:1506.02438.

[6] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan and Pieter Abbeel. Trust Region Policy Optimization, 2015; arXiv:1502.05477.

[7] Anonymous authors. Implementation Matters in Deep Policy Gradients: A case study in PPO and TRPO, 2019;

[8] Lianjiang Li, Yunrong Yang and Bingna Li. Combine PPO with NES to Improve Exploration, 2019, arXiv:1905.09492.