COMPSCI 687 Fall 2023 Final Project: Deep Q-Learning and Proximal Policy Optimization on Atari Games

Prasann Desai
University of Massachusetts Amherst
pmdesai@umass.edu

Dzung Pham
University of Massachusetts Amherst
dzungpham@cs.umass.edu

1. Introduction

In this project, we implemented two seminal policy gradient methods, namely Deep Q-Learning (DQN) [1, 2] and Proximal Policy Optimization (PPO) [3]. Our implementations make use of deep convolutional neural networks (CNN) to play two Atari games (Pong and Breakout) from the Gymnasium library [4]. We referred extensively to the original papers introducing DQN and PPO to choose the hyperparameters and policy/value network architecture. Hyperparameter optimization was limited to the optimizer's learning rate (and the clipping threshold for PPO's loss) due to time and computing resource constraints involved in training CNNs. Our results demonstrate that the implementations are capable of learning from purely visual domain and even achieving comparable performance to the the original papers' findings.

Contribution: Prasann Desai implemented and evaluated DQN. Dzung Pham implemented and evaluated PPO.

2. Algorithms

In this section, we describe the high-level ideas behind DQN and PPO as well as the implementation details.

2.1. Deep Q-Learning with Experience Replay

2.1.1. Overview. DQN [1] was introduced by DeepMind in the paper titled "Playing Atari with Deep Reinforcement Learning," which was published in December 2013. It was the first deep learning model that could learn directly from high-dimensional sensory input like vision and speech using RL. The DQN algorithm mainly constitutes training a convolutional neural network with a variant of Q-learning along with stochastic gradient descent to update the weights, whose input is raw pixels and whose output is a value function estimating future rewards. To ensure that the learning occurs on uncorrelated data with stationary underlying distributions, DQN uses an *Experience Replay* mechanism which randomly samples previous transitions and thereby smoothens the training distribution over many past behaviors. The main objective of this algorithm is to create a single neural network agent that can successfully learn to play two Atari games (Pong and Breakout) from the Gymnasium library [4].

2.1.2. Experience Replay. This algorithm uses the Experience Replay technique in which it stores the agent's experiences at each time-step in a data-set D pooled over many episodes into a *replay memory*. In this implementation, the replay memory is a *deque* data structure containing tuples of the format = $(s_t, a_t, r_t, s_{t+1}, terminated, truncated)$. In the inner loop of the algorithm, minibatch Q-learning updates are applied by sampling random tuples from the replay memory. This method has several advantages over standard Q-learning: First, each step of experience is potentially used in many weight updates, which allows for greater data efficiency. Secondly, it avoids learning from consecutive samples which we know are highly co-related; randomizing the samples breaks these correlations. Thirdly, when learning on-policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. In our implementation, our algorithm only stores the last N experience tuples in the replay memory, and samples uniformly at random from D when performing updates.

2.1.3. Network Architecture. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the pre-processing step described in Section 3.1.

- The first hidden layer convolves 16 8×8 with stride 4, followed by ReLU activation
- The second hidden layer convolves 32 4×4 with stride 2, followed by ReLU activation
- The final hidden layer is fully connected and consists of 256 rectifier units.
- The output layer is a fully-connected linear layer with a single output for each valid action. *Breakout* has 4 valid actions and *Pong* has 6 valid actions.
- Based on recommendations in this article, we have also modified the original approach from the DQN paper for improved results. We have set bias as $True$ in the last 2 linear layers and initialized the network weights using $nn.init.xavier\_uniform\_$ with a gain appropriate for

ReLU activations. It helps with initializing weights in a way that aims to keep the variance of activations roughly the same across different layers. For more stable training, we have also used *SmoothL1Loss* instead of the MSE loss function to avoid exploding gradients caused by outliers.

2.1.4. Pseudocode. We summarize our implementation of DQN in Algorithm 1.

Algorithm 1 DQN with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function i.e. the Q-network with random weights and create a copy "Target Q-Network" with the same weights.
for episode 1 to M **do**
  Initialize state s_1 = x_1 and preprocessed sequenced $\Phi_1 = \Phi(s_1)$
  Every alternate episode, update the Target Q-network with the current weights of the Q-network.
  for timestep $t = 1$ to T **do**
    Select action a_t using an epsilon-greedy strategy based on $Q(s_t, a)$
    Execute action a_t in emulator, observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample a random minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from D
    If Φ_{i+1} is non-terminal, Set target $y_i = r_i + \gamma \max'_a targetQ(\Phi_{i+1}, a'; \theta)$
    If Φ_{i+1} is terminal, Set target $y_i = r_i$
    Perform a minibatch gradient descent step based on SmoothL1Loss on y_i and $Q(\Phi_i, a_i; \theta)$
  end for
end for

2.2. Proximal Policy Optimization

2.2.1. Objective. PPO [3] was introduced in 2017 by researchers at OpenAI and has been used to successfully train several widely used AI-powered applications, particularly ChatGPT.[1] It aims to prevent "catastrophic" policy gradient updates by utilizing a novel optimization objective that constrains how much a policy can change. This "clipped surrogate objective" is formally defined as follows:

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

where:

- $\hat{A}_t$ is the estimated advantage of taking action a_t at time t (equivalent to $Q(a_t, s_t) - V(a_t)$)
- π_θ is the policy function which yields the probability of taking a given action
- $r_t(\theta)$ is the ratio of the current and previous probabilities of taking action a_t given s_t, i.e.:

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

- ϵ is the clipping hyperparameter $\in (0, 1)$, usually in the range $[0.1, 0.3]$
- $\text{clip}(x, \epsilon_1, \epsilon_2) = \min(\max(x, \epsilon_1), \epsilon_2)$

Thus, minimizing this objective w.r.t. θ will lead to increased probability for actions with $\hat{A}_t > 0$ and reduced probability for actions with $\hat{A}_t < 0$. The clip operator will limit the impact of changing the probability ratio to be within $[1-\epsilon, 1+\epsilon]$. This loss can be easily implemented using automatic differentiation libraries like PyTorch. Note that $\pi_{\theta_{\text{old}}}$ are treated as constants in the optimization.

2.2.2. Advantage Estimation. To estimate the advantages, PPO makes use of the Generalized Advantage Estimation method (GAE) [5]. Given the reward discount factor γ and the GAE parameter λ, $\hat{A}_t$ is estimated as follows:

$$\hat{A}_t = \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t}(r_k + \gamma V(s_{k+1}) - V(s_k)) \quad (2)$$

where T is the fixed length of the considered trajectory (thus $t \leq T$), r_k is the reward received at time k, and V is the state-value function (based on $\pi_{\theta_{\text{old}}}$). Note that the estimated advantages are also treated as constants in the PPO optimization.

2.2.3. Value Function Optimization. As the policy function and the value function share weights in our implementation, we need to incorporate a value loss into the PPO's surrogate objective. We define $L^{VF}(\theta)$ as follows:

$$L^{VF}(\theta) = \frac{1}{2}\hat{E}_t\left[(V^\theta(s_t) - V_t^{target})^2\right]$$

where $V^{target} = V^{\theta_{\text{old}}}(s_t) + \hat{A}_t$. Then, the new objective is:

$$L^{CLIP+VF}(\theta) = L^{CLIP}(\theta) + L^{VF}(\theta)$$

Note that in the original PPO paper, a value coefficient c is used in the loss, i.e. $L^{CLIP+VF}(\theta) = L^{CLIP}(\theta) + cL^{VF}(\theta)$. Our chosen objective is thus equivalent to setting $c = 1$, which was also adopted by the original paper when evaluating on the Atari domain.

2.2.4. Network Architecture. Both of our policy and value networks share a 3-layer CNN with the same architecture as [2, 3]. Specifically, the shared CNN consists of the following layers, in order:

- A CNN layer with 32 filters of size 8×8 with stride 4, followed by ReLU activation
- A CNN layer with 64 filters of size 4×4 with stride 2, followed by ReLU activation
- A CNN layer with 64 filters of size 3×3 with stride 1, followed by ReLU activation
- A fully-connected layer of size 512, followed by ReLU activation

1. https://openai.com/blog/chatgpt

The last layer of the policy network is a linear layer with size equal to the number of actions available, while the last layer of the value network is of size 1. Based on recommendations in [6, 7, 8], we initialize the layers' weights using the orthogonal initialization scheme with bias set to 0.0. The standard deviations are $\sqrt{2}$ for the layers in the shared CNN. The final linear layer of the policy and value networks have standard deviation 0.01 and 1.0, respectively.

2.2.5. Pseudocode. We summarize our implementation of PPO in Algorithm 2.

Algorithm 2 PPO

Input: A game environment G with n actors, policy network π, value network V, time steps T, learning rate λ, number of update epochs E, discount rate γ, GAE discount rate λ_{GAE}, clip threshold ϵ
Output: The updated policy and value networks.

```
for iteration = 1, 2, ... do
    D ← an empty array of experience data (state, action, value, reward, termination)
    π_old ← π
    for t = 1, 2, ..., T do
        Take a step in G for all n actors by randomly sampling actions from π_old
        Append experience (state, action, estimated value using V, reward, termination) to D
    end for
    Estimate advantage Â using γ, λ_GAE for each experience in D
    for epoch = 1, 2, ..., E do
        for each minibatch of states, actions, values, advantages from D and Â do
            L ← clipped PPO loss + Value loss
            Optimize π and V using ∇L
        end for
    end for
end for
return π, V
```

3. Experiments

In this section, we discuss the details of our experiment setup and our hyperparameter optimization process.

3.1. Evaluation Domains

We evaluated DQN and PPO on two famous Atari games, namely Pong[2] and Breakout[3] from the Gymnasium library [4]. In Pong (Figure 1), the agent controls the right paddle and competes against the left paddle to deflect the ball away its goal towards its opponent's goal. The game ends when either player gets to 21 scores first. In Breakout (Figure 2), the agent controls a paddle to bounce a ball towards brick wall at the top of the screen to destroy the wall. Once a wall is completely destroyed, a new wall will automatically replace it. The game ends when the agent runs out of "lives" or achieves a maximum score of 864.

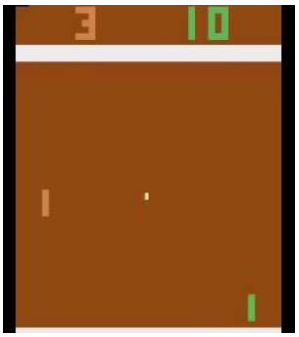

Figure 1. Pong

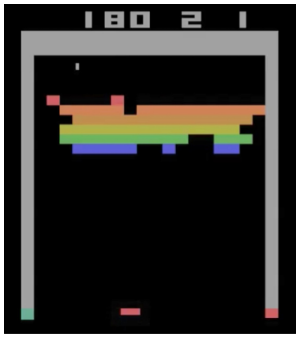

Figure 2. Breakout

The games are preprocessed in more or less the same fashion as [2, 3]. Specifically:

- The initial state is obtained by randomly taking the no-op action (i.e. do nothing) for a maximum of 30 no-ops.
- 4 frames are skipped between each successive step.
- 2 most recent frames from the skipped frames are max-pooled to form the game state.
- Frames are resized from 210×180 to 84×84.
- Frames are grayscaled with the pixel values in $[0, 1)$.
- Rewards are clipped to $\{-1, 0, 1\}$ depending on the sign.
- Games are automatically reset when terminated or truncated due to time limit.

For PPO, we collect the training data from multiple concurrent actors by leveraging the vectorized environment capability provided by the Gymnasium library.

3.2. Optimizers and Hyperparameters

3.2.1. DQN. For the experiments, we have used the RMSProp optimizer with default hyperparameters set by PyTorch. The behavior during training was ϵ-greedy with ϵ annealed linearly from 1 to 0.05 over the first 50,000 frames and fixed at 0.05 thereafter. We trained for a total of 1 million frames and used a replay memory of 100,000 most recent frames. Before training starts, the replay memory is pre-filled with random $100 * batch\_train\_size$ transitions. A set of 100 frames from the replay memory is chosen at random and the learning progress (trend of Average Q-value) of the algorithm is evaluated on those frames over time. For each experiment, we set $\gamma = 0.99$, $batch\_train\_size = 32$, $annealing\_steps = 50000$, $num\_training\_episodes = 50$, $num\_epochs = 100$ ($\approx$10000 frames per epoch). As a result, one epoch corresponds to 10000 minibatch weight updates or roughly 12 minutes of training time which implies $\approx$20 hours of training time per experiment. Please note that due to time and resource constraints, the training is executed on a lower scale in comparison to the original paper,

2. https://gymnasium.farama.org/environments/atari/pong/
3. https://gymnasium.farama.org/environments/atari/breakout/

but the results are demonstrative and in sync with those of the original paper [1]. The hyper-parameter tuning process for choosing the learning rate α and ϵ was done on a much lower scale. The best α value was chosen by checking if it produced an increasing learning curve (increasing Average Q-value) consistently (which is why, we eventually settled with $\alpha = 0.0001$). The best ϵ was chosen by checking if the average rewards per episode weren't fluctuating too much over time after significant learning was done (which is why, we eventually settled with $\epsilon = 0.01$).

3.2.2. PPO. We used the Adam optimizer with default hyperparameters set by PyTorch. We set $\gamma = 0.99, \lambda_{GAE} = 0.95$, number of steps $= 128$, number of actors $= 8$, number of update epochs $= 3$, number of minibatches $= 4$, and total number of steps $= 10000000$. Due to time and resource constraints, we performed a grid search only on the optimizer's learning rate from $\{0.0001, 0.0003, 0.001\}$ and clipping parameter from $\{0.1, 0.2, 0.3\}$.

3.3. Computing Platform

We relied extensively on the Google Colab platform to execute our experiments since it provides an NVIDIA T4 GPU which can speed up training on visual inputs significantly compared to CPU. Due to the time and resources constraints imposed by Google Colab, each run could take up to 20 hours. We also utilized a personal desktop with an NVIDIA RTX 3090 GPU to help with the hyperparameter search.

4. Results

In this section, we present our evaluation results on Pong and Breakout. We note that to maintain consistent comparisons with the original papers of DQN and PPO, we report the two algorithms' performance in slightly different manners.

4.1. DQN

Figures 3, 5, 4, and 6 showcase DQN's performance. All the graphs and respective metrics are identical to the performance metrics used in the DQN paper [1]. *Average Reward per Episode* is an epoch-level metric calculated by taking the average of net rewards observed in each of the episodes in that epoch. At the end of every epoch, we also take an *average of maximum predicted Q values over the possible actions* for a fixed set of states set aside before the overall training starts. Over time, as a function of time/epochs elapsed, we plot these metrics over the 100 epochs in our experiments to demonstrate the learning of q-function by our trained network. Generally, both averaged reward plots are indeed quite noisy, giving one the impression that the learning algorithm is not making steady progress. However, another more stable, metric is the policy's estimated action-value function Q, which provides an estimate of how much

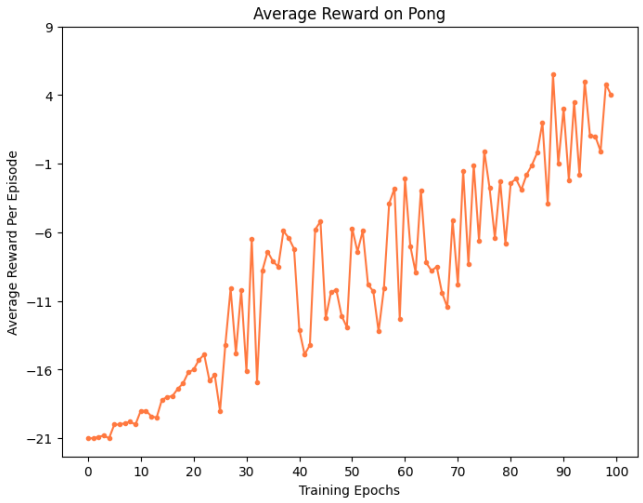

Figure 3. DQN's episodic rewards vs training epochs on Pong

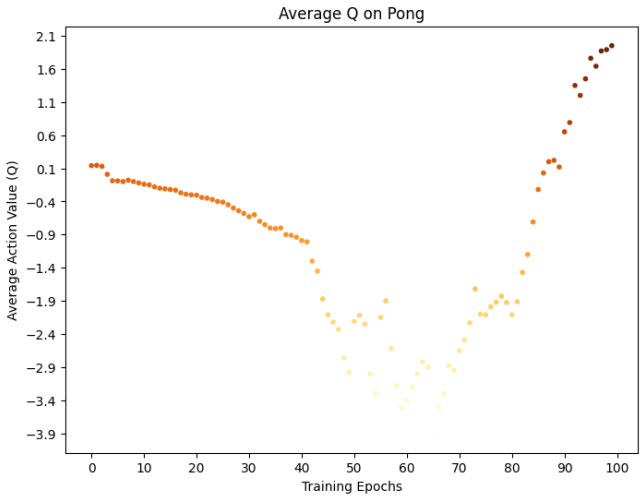

Figure 4. DQN's Average Q value vs training epochs on Pong

discounted reward the agent can obtain by following its policy from any given state. The two plots in 4 and 6 show that the average predicted Q increases much more smoothly than the average total reward obtained by the agent over time indicating progressive learning.

4.1.1. Pong. Despite the comparatively limited training resources, DQN was able to achieve a max score of ≈ 5 on Pong. As we can see from Figure 3, there is an upward trend of average rewards observed over time, showcasing the network's ability to get better at playing Pong as the agent gets trained over more and more frames. For the Average Q metric, we observe that the q-value keeps decreasing and after close to 50 epochs, the behavior starts reversing gradually [Figure 4]. This dip is observed because negative rewards are observed at a fast pace and frequently, causing q-values to decrease and also encouraging exploration. Once the agent explores all actions on enough frames, it slowly

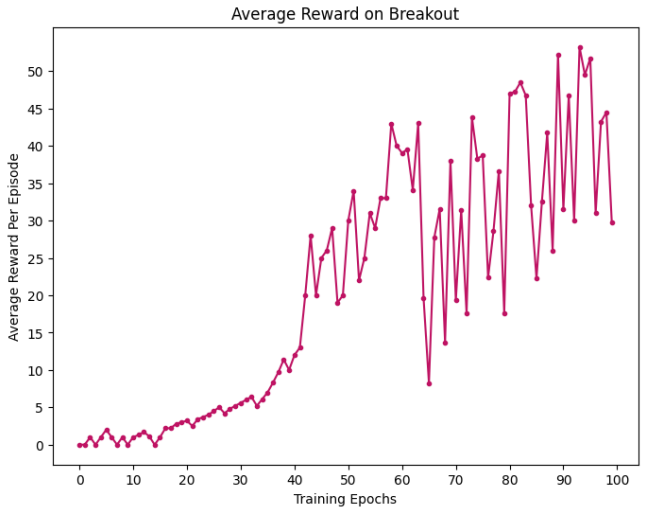

Figure 5. DQN's episodic rewards vs training epochs on Breakout

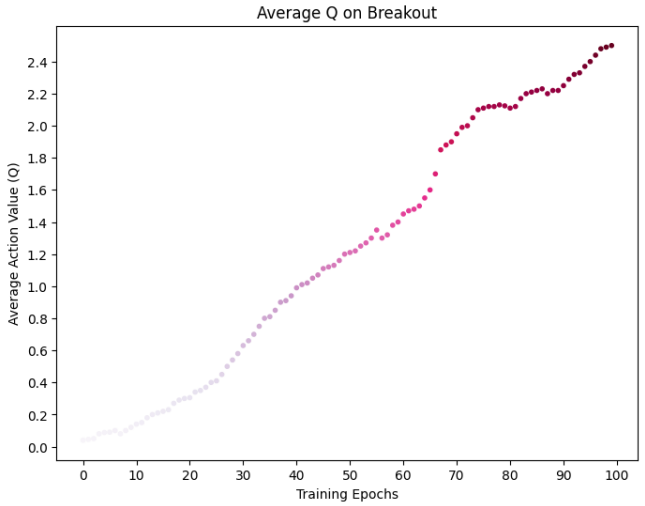

Figure 6. DQN's Average Q value vs training epochs on Breakout

starts to choose the actions with the least bad action value for those frames and the average Q metric starts to increase. Eventually, as training progresses further, the agent starts to achieve positive values for both metrics, which essentially means it gets better at playing Pong.

4.1.2. Breakout. Despite the comparatively limited training resources, DQN was able to achieve a score of $\approx$ 52.4 on Breakout. As we can see from Figure 5, there is an upward trend of average rewards observed over time, showcasing the network's ability to get better at playing Breakout as the agent gets trained over more and more frames. There is also a definite upward trend in *Average Episodic Q* in Figure 6. In the original DQN paper, their model is able to achieve a reward of $\approx$ 150-160 consistently due to its Agent being trained on 10x the number of frames in comparison to our implementation. Nevertheless, based on the learning curves we believe our implementation of DQN can successfully learn to play Breakout.

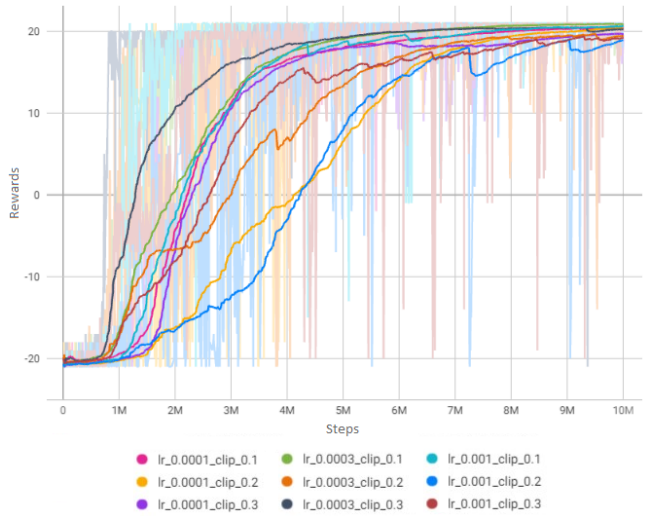

Figure 7. PPO's episodic rewards vs training steps on Pong

TABLE 1. OVERALL AVERAGE REWARDS AND LAST 100 EPISODES AVERAGE REWARDS FOR PPO ON PONG

λ	ϵ	Overall avg.	Last 100 avg.
0.0001	0.1	9.84	20.41
0.0001	0.2	1.89	20.73
0.0001	0.3	8.86	19.95
0.0003	0.1	13.52	**20.92**
0.0003	0.2	8.39	20.13
0.0003	0.3	**13.86**	20.26
0.001	0.1	12.27	20.87
0.001	0.2	3.55	20.72
0.001	0.3	8.61	19.62

4.2. PPO

Figures 7 and 8 showcase PPO's performance, which we collected using the Tensorboard library (built into PyTorch). The figures were in fact also drawn using Tensorboard. We set the (Exponential Moving Average) smoothing factor of Tensorboard to 0.99 to demonstrate the average performance more clearly. We follow the original PPO paper's figures, which plot the episode rewards against the number of environment steps (i.e. each actor's step will count as one step).

4.2.1. Pong. PPO was able to attain the maximum score of 21 in all hyperparameter settings (similar to the original PPO paper). As the agent gets trained more, the performance also stabilizes. From Table 1, we can see that a learning rate of 0.0003 and clip epsilon of 0.3 yields the best average performance overall. Reducing the clip epsilon to 0.1 yields the best average performance in the last 100 episodes of training (and 2nd best overall average performance).

4.2.2. Breakout. PPO managed to achieve the best average rewards of 193.49 for the last 100 episodes when setting

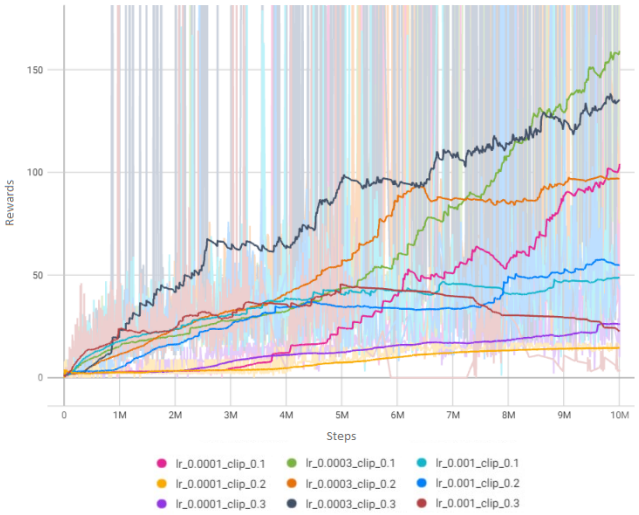

Figure 8. PPO's episodic rewards vs training steps on Breakout

TABLE 2. OVERALL AVERAGE REWARDS AND LAST 100 EPISODES AVERAGE REWARDS FOR PPO ON BREAKOUT

λ	ϵ	Overall avg.	Last 100 avg.
0.0001	0.1	30.24	152.7
0.0001	0.2	7.39	14.82
0.0001	0.3	12.13	29.39
0.0003	0.1	63.48	**193.49**
0.0003	0.2	45.36	121.99
0.0003	0.3	**71.68**	117.15
0.001	0.1	33.13	64.27
0.001	0.2	29.32	79.01
0.001	0.3	28.50	6.56

learning rate to 0.0003 and clip epsilon to 0.1. Increasing the clip epsilon to 0.3 yields the best overall average performance. These two settings happen to coincide with the results we found for Pong. In the original PPO paper, the authors achieved an average reward of 274.8 for Breakout for the last 100 episodes. We believe their higher performance is thanks to various other implementation details that we did not included (e.g. entropy loss, value loss clipping).

5. Conclusion

In this project, we have implemented DQN and PPO – two seminal reinforcement learning algorithms that are capable of learning from complex input domains. Our evaluations on Pong and Breakout have demonstrated the learning capability of our implementations. We look forward to learning more about other policy gradient methods in the future.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. [Online]. Available: https://arxiv.org/abs/1312.5602

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015. [Online]. Available: https://www.nature.com/articles/nature14236

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1707.06347

[4] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," Mar. 2023. [Online]. Available: https://zenodo.org/record/8127025

[5] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1506.02438

[6] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "Implementation matters in deep rl: A case study on ppo and trpo," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=r1etN1rtPB

[7] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem, "What matters for on-policy deep actor-critic methods? a large-scale study," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=nIAxjsniDzg

[8] S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, and W. Wang, "The 37 implementation details of proximal policy optimization," in *ICLR Blog Track*, 2022, https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/. [Online]. Available: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/