Building AI Game Agent using Reinforcement Learning

Apoorva Manjunath, Faiz Punjani

Northeastern University, Boston, MA
manjunath.ap@northeastern.edu, punjani.f@northeastern.edu

Abstract

The aim of this project was to implement AI to play games using Reinforcement Learning. OpenAI gym toolkit is used to develop and train the agents with CartPole, FrozenLake and Atari 2600 Pong environments. Agents are trained using Deep Q-learning with experience replay using Dueling network architecture and Proximal Policy Optimization algorithm in Actor Critic style on each of these environments. After training for a number of episodes, the agents are evaluated by running them for 100 episodes. Our results show that both algorithms performed comparably well on each of the game environments. In environments like CartPole and Frozen Lake the agents learned to get the max rewards in almost every episode. Various experiments were done to tune the hyper parameters for each algorithm and for each environment. We found that DQN performs the best when combined with Experience replay. Also PPO is capable of learning optimal policies but hyperparameter tuning may take time for complex environments like Pong.

Introduction

In recent years, Reinforcement Learning(RL) techniques have demonstrated remarkable progress on a variety of challenging problems ranging from Atari to the game of Go. Just like humans do, these AI agents learn by trial-and-error using rewards, gaining knowledge from experience to use it to be able to choose appropriate action in a state.

Common practice often used in analysing and comparing the learning ability of different RL algorithms involves comparing these algorithms based on their performance in learning to play video games from raw pixel data of the game screen, a termination signal and a reward signal. While video games have a measure of scores that make comparison simpler, they are complex and challenging to be mastered even for human players. Using this approach, we implement and compare performances of off-policy Deep Q-learning Network (DQN) and on-policy Proximal Policy Optimization (PPO) algorithms on three environments of OpenAI gym. Detailed description of these environments can be found towards the end of this section.

Motivated by the widely successful Deep-RL algorithm by DeepMind(Mnih et al., 2015), we explored the idea of using deep neural networks to represent the Q-network and to train this Q-network to predict total reward. We use the DQN algorithm to learn to play games like Atari and other problems without any prior knowledge of game rules.

PPO belongs to the family of Policy Gradient algorithms which are on-policy algorithms. Policy Gradient methods have a convergence problem which is addressed by the natural policy gradient which involves a second-order method. The computational complexity of this is too high and much research is done to reduce this complexity by approximation. But PPO handles this slightly differently by clipping the objective function which allows it to use a first order method to optimize the objective function. According to a quote from OpenAI on PPO:

> Proximal Policy Optimization (PPO), which performs comparably or better than state-of-the-art approaches while being much simpler to implement and tune.

PPO became one of the most popular RL techniques after it was introduced in (Schulman et al., 2017) and has various versions of it. In this project we implemented the Actor-Critic style PPO and applied it on the different environments from OpenAI.

We applied our RL algorithms to different environments of varying complexity and compared how each of them performs in these environments. The environments used are CartPole, Frozen Lake and Atari Pong from OpenAI gym. More details about these environments are in the next section.

Background

Environments

CartPole

CartPole is a classic control environment where a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart to move it right or left. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical or the cart moves more than 2.4 units from the center. The max possible reward in each episode is 500 after which the episode automatically ends. The input to the game agent is a representation of the input state including cart position, cart velocity, pole angle and velocity of pole tip.

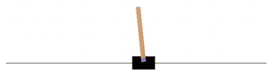


Figure 1. Cartpole environment

Frozen Lake

In the Frozen Lake environment the agent controls the movement of a character in a grid world which represents a frozen lake. Some tiles of the grid are walkable, and others lead to the agent falling into the hole where the ice has melted. Additionally, the movement direction of the agent is uncertain as the ice is slippery and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile. The agent can take four actions to move the character left, right, up and down.

The letters of the grid in Figure 2 represent the following:

S: starting point, safe

F: frozen surface, safe

H: hole, fall to your doom

G: goal, safe

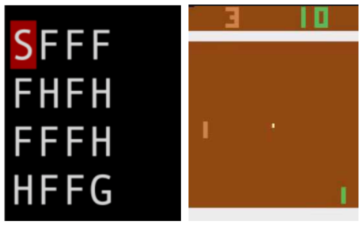


Figure 2. **Left:** FrozenLake **Right:** Atari 2600 Pong environment

Atari Pong

Atari Pong is a table tennis themed game where the player controls an in-game paddle by moving it. The player competes against a computer opponent controlling a second paddle on the opposing side. The Player uses the paddle to hit a ball back and forth. The goal is to reach eleven points before the opponent; points are earned when one fails to return the ball to the other. The range of reward for each episode is between -21 and 21 both inclusive. The input to the agent is the raw screen frames, which are 210 x 160 pixel images with a 128 color palette. The agent can take actions up and down to move the paddle in game.

Q-learning Methods

We use a sequential decision making setup where an agent interacts with an environment over discrete time steps. In the case of an Atari game environment, the agent represents the state consisting of M consequtive image frames $s_t = (x_{t-M+1},...,x_t) \in S$ at time step t. At each time-step the agent selects an action a_t from the set of legal game actions, $A = \{1, ..., K\}$ and receives a reward r_t from the game emulator.

The goal of the agent is to learn by interacting with the emulator and maximizing the rewards. As we discount future rewards by a factor γ at each time-step, we define the future discounted return at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time-step at which the game terminates. The optimal action-value function $Q^*(s,a)$ is the maximum expected return that can be obtained by using any strategy after seeing some sequence s and taking action a, $Q^*(s,a) = max_\pi E\,[R_t | s_t = s, a_t = a, \pi]$ where π is the policy mapping sequences to action.(Mnih et al., 2013)

If the optimal value $Q^*(s',a')$ of the sequence s' at the next time step was known for all legal actions a', then the optimal strategy is to select the action a' maximizing the expected value of $r + \gamma Q^*(s',a')$,

$$Q^*(s,a) = E_{s'}[r + \gamma max_{a'} Q^*(s',a') \mid s,a]$$

This algorithm is off-policy: it learns about the greedy strategy and ensures there is enough exploration and exploitation of the state space. We do this using ε-greedy strategy that uses a greedy strategy with probability $1 - \varepsilon$ and chooses a random action with probability ε.

Deep Q-Learning Networks

To estimate the action-value function, we use a function approximator $Q(s,a;\theta) \approx Q^*(s,a)$ like neural network with weights θ as a Q-network. A Q-network can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i) = E_{s,a,r,s'}[(y_i^{DQN} - Q(s,a;\theta_i))^2], \text{ where}$$

$$y_i^{DQN} = r + \gamma\ max_{a'} Q(s',a';\theta_{i-1}) \text{ and}$$

$$\nabla\theta_i L_i(\theta_i) = E_{s,a,r,s'}[(y_i^{DQN} - Q(s,a;\theta_i))\ \nabla\theta_i Q(s,\ a;\ \theta_i)]$$

is the specific gradient update.

For this implementation, we have combined DQN with *experience replay.* During learning, the agent accumulates a dataset which is a memory of past experiences $D_t = \{e_1, e_2, ..., e_t\}$ from many episodes. When training the Q-network, instead of using only current experience, the network is trained by sampling batches of experiences uniformly at random from D.

Experience replay increases data efficiency through re-use of experience samples and it reduces variance since uniform sampling from the replay buffer reduces the correlation among the samples used in the update. (Wang, Z., 2015)

Dueling Network Architecture

The important aspect of Dueling Network Architecture (Wang, Z., 2015) is that it is unnecessary to estimate the value of each action choice. In some states it is important to know which action to take and in many other states the choice of action has no repercussion on what happens. This architecture separates the representation of state values and (state-dependent) action advantages. It consists of two streams that represent the value and advantage functions, while sharing a common feature learning module. The Dueling network outperforms a single stream network, with the performance gap increasing with the number of actions. This is used for the implementation of DQN with Pong environment in below experiments, where number of actions are not too high, thus the performance improvement due this is not significant.

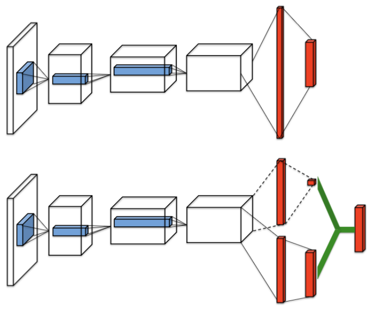


Figure 3. Single stream Q-network(top) and the dueling Q-network(bottom). Figure from (Wang et al., 2015)

Policy Gradient Methods

Policy gradient methods are a set on-policy algorithms which are targeted at modeling and optimizing the policy directly. The agents receive rewards when interacting with the environment using the current policy which is used to adjust the current policy. The policy is usually modeled with a parameterized function respect to θ, $\pi_\theta(a|s)$. The objective is to maximize the discounted returns (accumulated rewards) R_t following a parameterized policy. This can be done by computing an estimator of the policy gradient and applying a stochastic descent algorithm on it. The most commonly used gradient estimator is of the form

$$g = E_t\,[\nabla_\theta \log \pi_\theta(a_t \mid s_t)A_t]$$

where A_t is an estimator of the advantage function at timestep t and E_t is the empirical average over a finite number of samples.

Generalized Advantage Estimate (GAE)

Advantage can be defined as a way to measure how much better off we can be by taking a particular action when we are in a particular state. We want to use the rewards that we collected at each time step and calculate how much of an advantage we were able to obtain by taking the action that we took. One of the better ways to calculate advantage estimates is called generalized advantage estimation proposed in (Schulman et al., 2015b). The generalized estimator is defined as

$$A_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+1}^V$$

$$\delta_{t+1}^V = r_{t+1} + \gamma V(s_{t+l+1}) - V(s_{t+l})$$

γ is the discount factor to control future rewards and parameter λ controls the trade-off between bias and variance. Both γ and λ lie in the range (0, 1) inclusive.

Proximal Policy Optimization (PPO)

PPO belongs to the class of policy gradient algorithms and is based on Trust Region Policy Optimization (TRPO) algorithm.

In TRPO (Schulman et al., 2015a), an objective function (surrogate objective) is maximized subject to a constraint on the size of the policy update. The surrogate objective is maximized based on the KLD constraint between the old and new policy and can be written as,

$$maximize_\theta\ E_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t\right]$$

$$subject\ to\ E_t\left[KL\left[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right]\right] \le \delta$$

where θ_{old} represents the policy parameters before the update and δ is the KLD upper limitation. The conjugate gradient algorithm can be applied to the above equation to solve it efficiently but it's still a complex problem.

PPO tries to overcome that by replacing the original constrained problem with a clipped surrogate objective such that the KL constraint is implicitly imposed. Clipping the surrogate objective helps remove incentives for the new policy to get far from the old policy. The clipped surrogate objective can be written as

$$L^{CLIP}(\theta) = E_t\left[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A_t)\right]$$

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

where ε is the hyper parameter to control the clipping ratio. There is also another alternative to the clipped surrogate objective which uses a penalty on KL divergence. But the various experiments performed suggest that the clipped PPO version performs better than the PPO KL penalty version.

Related Work

There has been a lot of related work done in RL especially for the environments in OpenAI gym as a toolkit for developing and comparing RL algorithms.

In the paper presenting the first deep learning model by DeepMind(Mnih et al., 2013), they applied the DQN method to seven Atari 2600 games from Arcade Learning Environment. They concluded that the model outperformed all previous approaches in six games and surpassed human experts on three of them, including Pong. In this paper, the average total reward with DQN algorithm by running ε -greedy policy with value 0.05 is 20 and the best performing episode obtained reward of 21.

When Dueling Network architecture(Wang et al., 2015) was used on Pong, compared to the baseline single network, dueling architecture performed 0.24% better.

PPO was introduced in the paper (Schulman et al., 2017) in which they ran PPO on the games in the Atari Domain and compared it to other algorithms like A2C and ACER. They used 2 scoring metrics: (1) average reward per episode over the entire training period (which favors fast learning) and (2) average reward per episode over the last 100 episodes of training (which favors final performance). PPO won on 30 Atari games according to the first metric and 19 Atari games according to the second metric out of the total 49 Atari games they ran the algorithms on. Although the PPO algorithm given in the paper is slightly different from our implementation as it uses multiple parallel actors to collect samples from the environment whereas we only use one actor in our implementation.

(Chen et al., 2018) is another paper that built on PPO by using an adaptive clipping approach instead of fixed clipping. They compared the new adaptive clipping approach with the original PPO implementation mentioned in (Schulman et al., 2017). There are more papers building on the PPO algorithm but our project is about implementing PPO in Actor-Critic style and comparing it with other RL algorithms on different environments.

Project Description

Deep Q-learning with Experience replay

The generalized Deep Q-learning process described in the previous sections was used to train all three environments.

```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
      end for
  end for
```

Figure 4. Algorithm for Deep Q-learning with experience replay

Hyperparameters

1. Gamma: The discounting factor that is multiplied by future rewards to slightly decrease the effect of these rewards on the agent.
2. Epsilon: Represents the proportion of random actions to the actions that are taken based on existing knowledge of the agent that is accumulated in episodes. Since the initial knowledge of the agent is none, it is set to high value and is gradually reduced until a minimum limit is reached.
3. Epsilon Decay: As the agent learns, the speed at which the epsilon value should decrease is controlled by this.
4. Replay size: The size of replay buffer that stores the past experiences and is used during training.
5. Hidden layer nodes: This represents the number of nodes in the hidden layer of the neural network
6. Learning rate: A tuning parameter that decides the step size at each iteration, maintaining a trade-off between the rate of convergence and overshooting

Cartpole and FrozenLake

We use a three layer neural network for DQN. The input layer is linear with a number of nodes based on state space. Second layer and third layer have 50 and 100 nodes respectively. Each node in the output layer represents an action in the environment. This network uses Adam's method for optimization technique and Mean Squared Error function for loss computation.

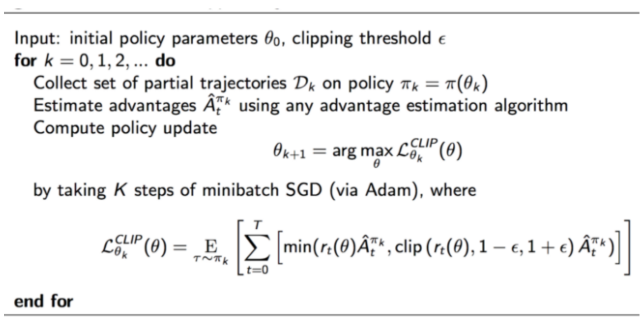
Atari 2600 Pong

In case of Atari games like Pong, the input to the agent is the raw Atari frames, which are 210 x 160 pixel images with a 128 color palette. Since this is computationally heavy, we do preprocessing of these images to reduce the input dimensionality. These images are converted from RGB representation to grayscale and the frames are cropped to process only the playing area of the screen removing the top score board of 20 pixels. The frames are resized and normalized to obtain 64 x 80 images that are fed to the learning model.

In this implementation, we use a three layer convolutional layer to which the processed image of 64 x 80 is fed. The first layer gives 32 square filters of size 8 and stride 4. Second layer has 64 filters of size 4 and stride 2. Third layer has 64 filters with kernel size 3 and stride 1. After each layer a batch normalization is applied and this produces an output image array of size 1536. Finally as per the Duel Network architecture, there is an action stream and a state value stream. Action layer is a full connected linear layer with an output for each valid action which are 6 in case of Pong. The state value layer is also a fully connected linear layer and has only one node for the final value in the output layer.

Proximal Policy Optimization (PPO)

We implemented the PPO algorithm with clipped surrogate objective L^{CLIP} given in the background section. We ran a stochastic gradient descent algorithm Adam on the clipped surrogate objective and optimized it. The algorithm is as follows:



```
Input: initial policy parameters $\theta_0$, clipping threshold $\epsilon$
for $k = 0, 1, 2, \ldots$ do
    Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
    Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
    Compute policy update
        $\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}^{CLIP}(\theta)$
    by taking $K$ steps of minibatch SGD (via Adam), where
        $\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k})\right]\right]$
end for
```

Figure 5. PPO with Clipped Objective Algorithm. Figure from (Achiam, 2017)

We use Generalized Advantage Estimation (GAE) algorithm to calculate advantages. The function $\mathrm{clip}(r_t(\theta),1-\epsilon,1+\epsilon)$ clips the ratio to be no more than $1+\epsilon$ and no less than $1-\epsilon$.

Actor-Critic Style Implementation

We implemented the PPO algorithm in an Actor-Critic style framework. An actor model (policy model) π_θ has its own model parameters θ. With the actor model, PPO Agent performs the task of learning what action to take under a particular observed state. The PPO Agent takes actions based on the predictions of the actor model. The reward of these actions can be positive or negative. This reward is taken in by the critic model $V\mu$ with its model parameters μ. The role of the critic model in the PPO Agent is to learn to evaluate if the action taken by the actor model led the PPO agent to be in a better state or not, and the critic model's feedback is used for actor model optimizations. Figure 6 shows the overall process of Actor-Critic PPO algorithm.

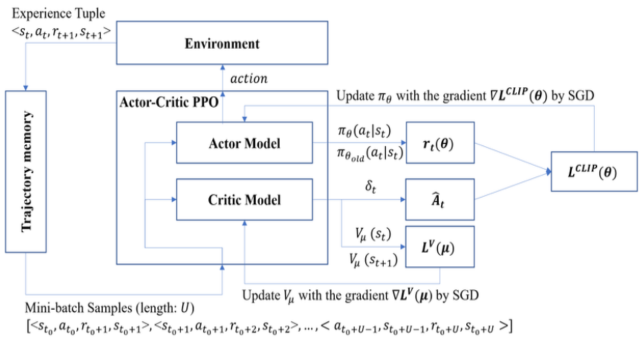

Experience Tuple
$<s_t, a_t, r_{t+1}, s_{t+1}>$
Environment
action
Actor-Critic PPO
Update π_θ with the gradient $\nabla L^{CLIP}(\theta)$ by SGD
Trajectory memory
Actor Model
Critic Model
$\pi_\theta(a_t|s_t)$
$\pi_{\theta_{old}}(a_t|s_t)$
$r_t(\theta)$
$L^{CLIP}(\theta)$
δ_t
$\hat{A}_t$
$V_\mu(s_t)$
$V_\mu(s_{t+1})$
$L^V(\mu)$
Update V_μ with the gradient $\nabla L^V(\mu)$ by SGD
Mini-batch Samples (length: U)
$[<s_{t_0}, a_{t_0}, r_{t_0+1}, s_{t_0+1}>, <s_{t_0+1}, a_{t_0+1}, r_{t_0+2}, s_{t_0+2}>, \ldots, < a_{t_0+U-1}, s_{t_0+U-1}, r_{t_0+U}, s_{t_0+U} >]$

Figure 6. Actor-Critic PPO Algorithm process. Figure from (Lim et al., 2020)

In our implementation, the Actor and Critic models are made up of Neural Networks. The Actor outputs the probabilities of each action to take in a particular state. The PPO agent takes the action with the highest probability (best action) in each state. The Critic outputs the value function used to evaluate the action taken by the PPO agent based on the best action given by the Actor. The Actor updates the policy distribution in the direction suggested by the Critic to learn the optimal policy.

Hyperparameters

PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning. The following hyperparameters are used in our implementation:

1. Gamma: The discounting factor that is multiplied by future rewards to slightly decrease the effect of these rewards on the agent.
2. Update Interval: The max number of samples after which the policy is updated in an episode.
3. Mini Batch Size: The number of samples to be included in one batch of SGD.
4. Clip Ratio: The clipping parameter used to clip the policy ratio to ensure that the deviation from the previous policy is relatively small.
5. Lambda: This is a smoothing parameter in GAE used for reducing the variance in training.
6. Epochs: Number of epochs for which SGD should run.
7. Actor Learning Rate: Learning rate of the actor model.
8. Critic Learning Rate: Learning rate of the critic model.
9. Hidden layer nodes: This represents the number of nodes in the hidden layers of the Actor and Critic Neural Networks.

Cartpole

The state representation of input in the Cartpole environment as described in the environments section is given as input to the Actor and Critic Models.

Frozen Lake

In the Frozen Lake environment, the discrete state size is 16 and the input to the agent is a number representing the state. The input state number is one-hot encoded and passed as the input to the Actor and Critic Models.

Atari Pong

In the case of Atari Pong, input is 210x160 raw pixel images with a 128 color palette. Since this is computationally heavy, we do preprocessing of these images to reduce the input dimensionality by removing the score board and non playable area of the screen and downsampling to 80x80 pixel images. We also make the paddles and the balls white and all the background black. Figure 7 shows what a processed image looks like.

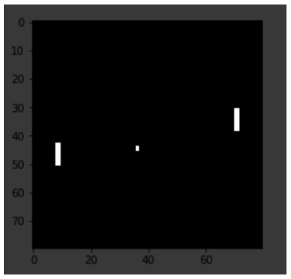


Figure 7. Processed Atari Pong input image

The input to the Actor and Critic models is made up of both current and previous input images so that the model can detect the motion of the ball. Each image is converted to an array of size 6400 concatenated to get the final array of size 2*6400 which is the input to the models.

Experiments

We performed experiments on three environments - Cartpole, Frozen Lake and Atari 2600 Pong.

Deep Q-learning with Experience replay

We use the same network for Cartpole and Frozenlake that can generalize the parameters based on state dimensions and action space. However, we use different implementations of the same algorithms for Pong to incorporate the input image format with convolutional layers.

Here are the values of various hyperparameters we used for the experiments below.

Hyperparameters for Cartpole and FrozenLake

1. Gamma: 0.9	2. Epsilon minimum: 0.01
3. Epsilon: 0.3	4. Replay size: 20
5. Epsilon Decay: 0.99	6. Learning rate: 0.001

Hyperparameters for Atari Pong

1. Gamma: 0.97	2. Epsilon minimum: 0.05
3. Epsilon: 1	4. Replay size: 50000
5. Epsilon Decay: 0.99	6. Learning rate: 0.00025

PPO Actor-Critic Style

The PPO Actor-Critic algorithm used is the same in all the environments. The actor and critic have separate neural network models with almost the same structure except for the last output layer. The actor output probabilities for each action whereas the critic outputs one value used for actor evaluation. The neural network models are made up of multiple fully connected hidden layers which differs based on environments.

Here are the values of various hyperparameters we used for the experiments below.

Common Hyperparameters Cartpole and FrozenLake

1. Gamma: 0.99	2. Epochs: 3
3. Clip Ratio: 0.1	4. Actor LR: 0.0005
5. Lambda: 0.95	6. Critic LR: 0.001

Remaining Hyperparameters for Cartpole

1. Update Interval: 5	2. Mini Batch Size: 5
3. Hidden layer nodes: 1 - 32, 2 - 16	

Remaining Hyperparameters for FrozenLake

1. Update Interval: 10	2. Mini Batch Size: 10
3. Hidden layer nodes: 1 - 512, 2 - 256	

Hyperparameters for Atari Pong

1. Gamma: 0.99	2. Lambda: 0.95
3. Update Interval: 2000	4. Epochs: 3
5. Mini Batch Size: 100	6. Actor LR: 0.0001
7. Clip Ratio: 0.1	8. Critic LR: 0.0002
9. Hidden layer nodes: 1 - 512, 2 - 256, 3 - 128	

Results for Cartpole

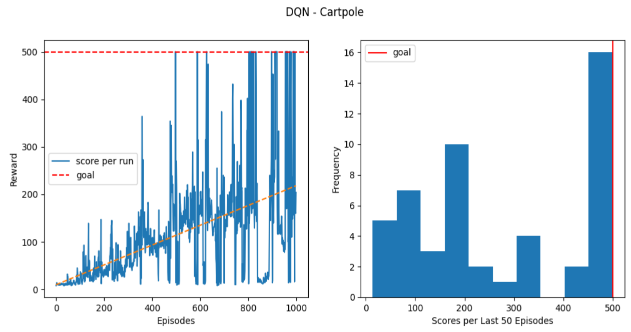


Figure 8. Cartpole using DQN in training **Left:** Learning curve **Right:** Histogram of scores

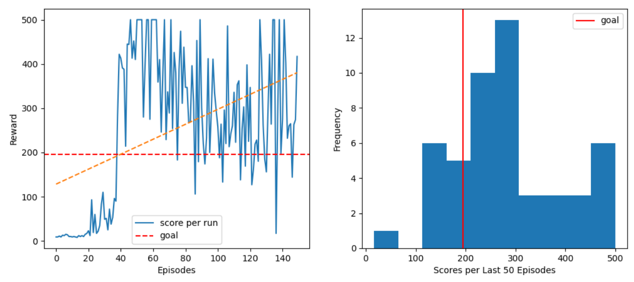


Figure 9. Cartpole using DQN with replay in training **Left:** Learning curve **Right:** Histogram of scores

The learning curves show the total reward received in each episode. The plots indicate that the agent starts reaching higher rewards much earlier during training in case of DQN with experience replay than standard DQN model. In both cases they reach the maximum reward of 500 in the Cartpole environment.

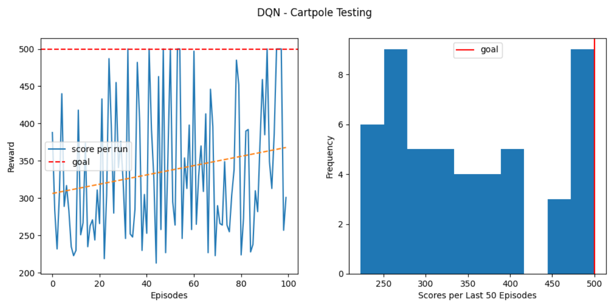


Figure 10. Cartpole using DQN in testing **Left:** Graph of rewards per episode **Right:** Histogram of scores

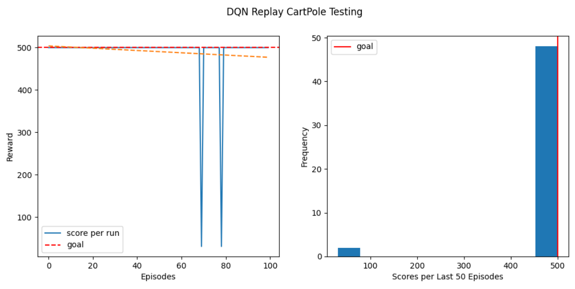


Figure 11. CartPole using DQN with replay in testing **Left:** Graph of rewards in each episode **Right:** Histogram of scores

During the testing phase of Cartpole with DQN, the agent using experience replay outperforms the agent using simple DQN. Even for a replay size of 20 and 50 hidden layers in the network, experience replay shows huge differences in performance. Combining with experience replay makes the agent more robust and smart since it remembers more than just the last action.

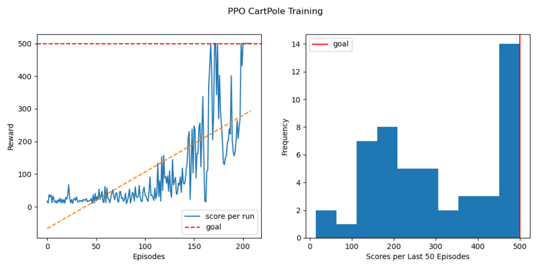


Figure 12. Cartpole using PPO in training **Left:** Learning curve **Right:** Histogram of scores

During the training phase of CartPole using PPO we stop training when the agent gets 500 which is the max reward for 10 episodes consecutively. The learning curve shows that the agent starts learning after around 100 episodes and starts getting consecutive max rewards around 200 episodes.

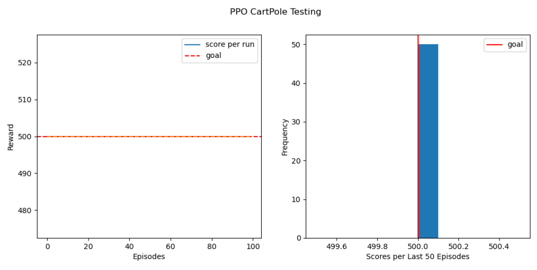


Figure 13. Cartpole using PPO in testing **Left:** Graph of rewards per episode **Right:** Histogram of scores

During the testing phase of CartPole using PPO for 100 episodes, the trained agent consecutively got a max reward of 500. This shows that the agent was able to learn the optimal policy in the CartPole environment.

Results for FrozenLake

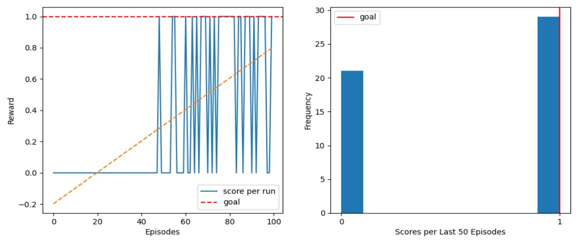


Figure 14. FrozenLake using DQN in training **Left:** Learning curve **Right:** Histogram of scores

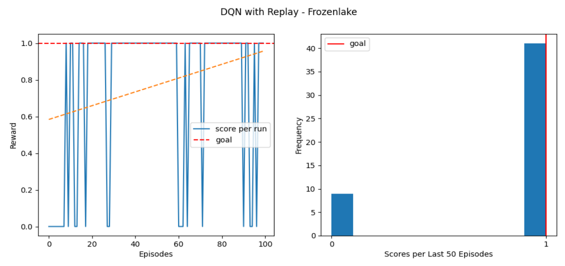


Figure 15. FrozenLake using DQN with replay in training **Left:** Learning curve **Right:** Histogram of scores

In the FrozenLake environment with slippery=False an agent trained for 100 episodes itself learns quickly reaching the goal. Compared to DQN agent, the pattern where the DQN agent with replay sees the first maximum score early in the episodes is common like in Cartpole.

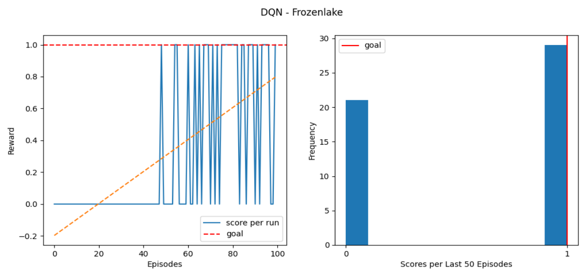


Figure 16. FrozenLake using DQN in testing **Left:** Graph of rewards per episode **Right:** Histogram of scores

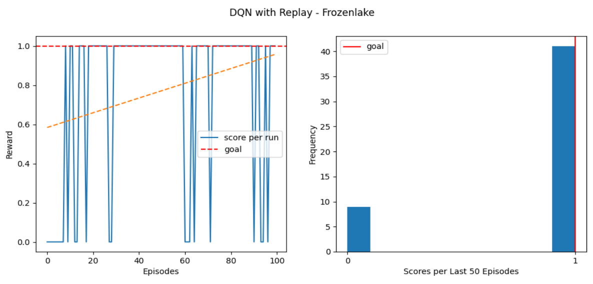


Figure 17. FrozenLake using DQN with replay in testing **Left:** Graph of rewards per episode **Right:** Histogram of scores

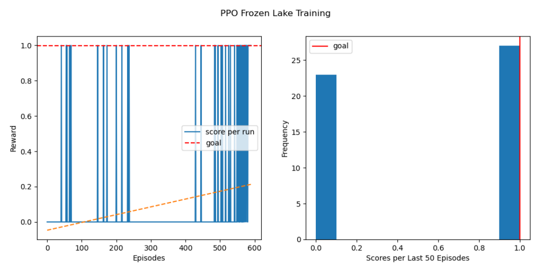


Figure 18. FrozenLake using PPO in training **Left:** Learning curve **Right:** Histogram of scores

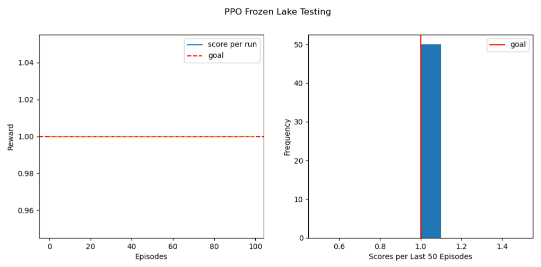


Figure 19. FrozenLake using PPO in testing **Left:** Graph of rewards per episode **Right:** Histogram of scores

In the FrozenLake environment with slippery=False using PPO, we can see that during the training the agent learns to reach the goal consecutively at around 600 episodes and we stop training. The trained agent also consecutively reached the goal during the testing phase from which it is evident that it learned the optimal policy.

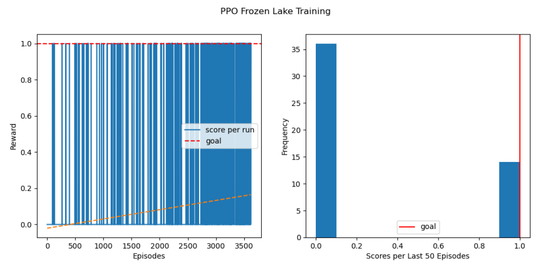


Figure 20. Slippery FrozenLake using PPO in training **Left:** Learning curve **Right:** Histogram of scores

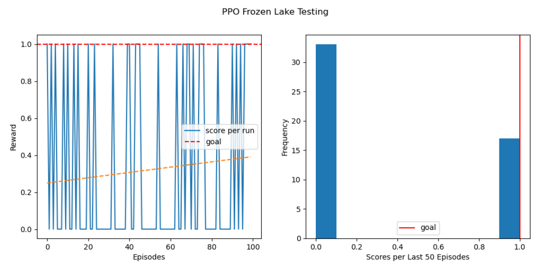


Figure 21. Slippery FrozenLake using PPO in testing **Left:** Graph of rewards per episode **Right:** Histogram of scores

In the FrozenLake environment with slippery=True using PPO there is only a 33.33% chance of moving in the direction where we want to. We stop training when the agent reaches the goal 5 times out of the last 10 episodes. We can see that during the training the agent reaches the goal many times but is having trouble learning due to the stochastic nature of the environment. The training stops at around 3500 episodes and the trained agent reaches the goal around 33% of time during the testing phase which seems good considering there's only a 33.33% chance of moving in the direction we want. Thus it looks like the agent has learned a near optimal policy.

Results for Atari Pong

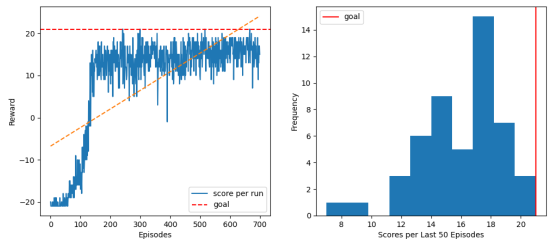


Figure 22. Atari Pong using DQN with replay during training **Left:** Learning curve **Right:** Histogram of scores

The Pong agent using DQN with experience replay and Duel DQN architecture is trained for 700 episodes. It crossed the range of negative rewards to positive around 120 episodes itself and also achieved the highest possible winning score of +21 early in training.

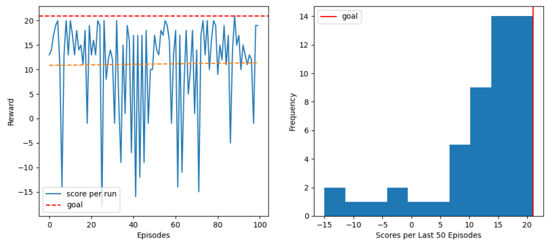


Figure 23. Atari Pong using DQN with replay in testing **Left:** Graph of rewards per episode **Right:** Histogram of scores

In the testing phase of Atari Pong agent with DQN for 100 episodes, the average reward obtained is 11.12 and the reward in the best episode is +21. As per paper(Mnih et al., 2013), the human expert agent's average total reward in Pong with ε-greedy policy is -3. From the plots we see that scores for the majority of the episodes lie in the range of 11 to 21. It took a running time of 753 seconds for testing 100 episodes and performed a total of 201858 cumulative steps in all episodes.

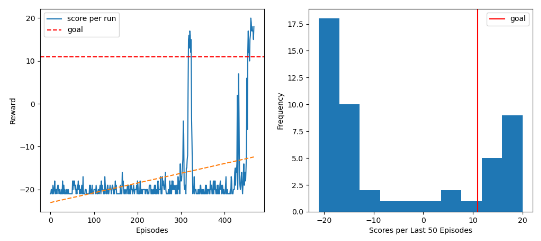


Figure 24. Pong using PPO in training **Left:** Learning curve **Right:** Histogram of scores

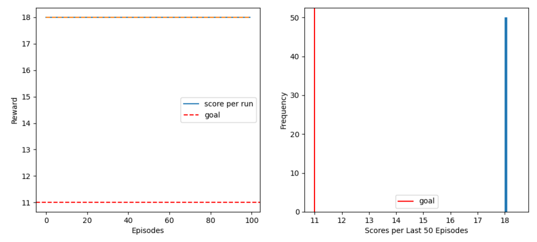


Figure 25. Pong using PPO in testing **Left:** Graph of rewards per episode **Right:** Histogram of scores

During the training phase of Pong using PPO we stop training when the agent gets a reward more than 11 (which means it's winning the game) consecutively for 10 episodes. We see that the agent consecutively wins for 10 episodes at around 450 episodes. The trained agent also consecutively won with 18 points during the testing phase. The hyperparameter tuning for Pong using PPO took more time than expected as the neural networks were bigger and the episodes lasted longer which meant it took hours to train. The tuning is still not perfect as the agent hasn't learned the optimal policy where it wins with 21 points.

	CartPole	FrozenLake		Pong
		~Slippery	Slippery	
DQN Avg	490	0.82	0.66	11.12
PPO Avg	500	1	0.4	+18
DQN Best	500	1	1	+21
PPO Best	500	1	1	+18

Table 1. Total average reward and reward in single best performing episode for each algorithm and environment

Table 1 shows the comparison of each algorithm for all environments by average and maximum reward. DQN works well in stochastic environments like FrozenLake. DQN does not always give maximum reward during testing even though it reaches the highest possible reward possibly because it did not explore all the possible states in training. PPO is more consistent and is able to give the best reward in majority cases consistently but does not learn as well as DQN in stochastic environments like Frozen Lake.

Conclusion

Reinforcement Learning techniques like off-policy Deep Q-Learning and on-policy Proximal Policy Optimization worked very well to train agents on the Atari domain for Pong and other problems like Cartpole and FrozenLake in the OpenAI gym environment. In Both these algorithms performed comparably well in all three environments. In CartPole and Frozen Lake environments the agents learned to get the max rewards in almost every episode. In Pong the agents learned to win in every episode even though they were not able to get the max rewards. There was a clear indication that DQN when combined with Experience replay outperforms the plain DQN model, even for smaller replay memory size. We also found that PPO is capable of learning optimal policies but hyperparameter tuning may take time for complex environments like Pong. Further work on this should include experimenting the models on different environments and different hyperparameters to better understand the limitations of these algorithms.

References

(Mnih et al., 2013) V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. CoRR, abs/1312.5602, 2013.

(Wang, Z., 2015) Wang, Z., de Freitas, N., and Lanctot, M. Dueling Network Architectures for Deep Reinforcement Learning. ArXiv e-prints, November 2015

(Mnih et al., 2015) V.Mnih, K.Kavukcuoglu, D.Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. 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, February 2015.

(Schulman et al., 2015a) J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust Region Policy Optimization. ArXiv e-prints, February 2015.

(Schulman et al., 2015b) J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. ArXiv e-prints, June 2015.

(Achiam, 2017) Joshua Achiam, Advanced Policy Gradient Methods, UC Berkeley, OpenAI, October 2017

(Lim et al., 2020) H.-K. Lim, J.-B. Kim, J.-S. Heo, and Y.-H. Han, Federated reinforcement learning for training control policies on multiple IoT devices, Sensors, vol. 20, no. 5, p. 1359, Mar. 2020

(Schulman et al., 2017) J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. ArXiv e-prints, July 2017.

(Chen et al., 2018) Gang Chen, Yiming Peng, and Mengjie Zhang. An adaptive clipping approach for proximal policy optimization. CoRR, abs/1804.06461, 2018.