Comparing Reinforcement Learning Methods under OpenAI Gym Environments

IP, Ho Pan
Department of Mathematics
hpipab@connect.ust.hk

Luk, Wing San
Department of Computer Science and Engineering
wslukaa@connect.ust.hk

Abstract

The purpose of this project is to implement different reinforcement learning methods, compare their performance under simple and complicated OpenAI Gym Environments , and draw insights into the characteristics of different methods. The methods that we used in this project included Q Learning, Deep Q Learning (DQN), Duelling Deep Q Learning (DDQN), Cross Entropy Method (CE), State–action–reward–state–action (SARSA), Asynchronous Advantage Actor-Critic (A3C). We focus on the following environments: Cartpole-v0, Breakout-v0, Taxi-v2, Centipede-ram-v0, Pong-v0, and SpaceInvaders-v0.

1. Introduction

Research applying reinforcement learning or different machine learning methodologies on playing video games has become very popular nowadays. In this paper, we consider the performance of several reinforcement learning methods on different games under OpenAI Gym environments, compare their relative performance, and analyse it with a view to identifying possible factors that might influence the performance of these methods under different types of environments. Our motivation is to draw further insights into the characteristics of different reinforcement learning methods, so that we can identify which methods are better suited for certain environments.

In this project, we would first start from testing different reinforcement learning methods in the simplest environment, Cartpole. After evaluating different reinforcement learning methods' performance (rewards vs episodes), we would apply these reinforcement learning methods in more complicated environments (Atari Games like SpaceInvaders, Breakout, Pong, Taxi). We will also conduct experiments by using different hyperparameters under different game environments to see how these methods' performances vary. We expect reinforcement learning methods would perform better with more rewards in fewer number of episodes in simpler environments(e.g. Cartpole). We also expect that Ddqn would perform the best while CEM perform the worst due to the model's simplicity.

2. Related Works

There were many previously published research on using reinforcement learning in different game environments. We took reference from the paper published by Deepmind in 2013 about how to apply reinforcement learning playing Atari [1]. We also took reference from the paper "Asynchronous Methods for Deep Reinforcement Learning" [2] when implementing the method of A3C. Moreover, we studied the paper about CEM [3] before we implemented CEM on different game environments. When we tried to implement ddqn, we also took reference from "Deep reinforcement learning with double Q-Learning" on how to implement ddqn correctly. Finally, we read the paper comparing Sarsa and Q-Learning to get some insights for our projects in how to compare different reinforcement learning methods under different situations. [5]

3. Data

The data used in this project was generated from the frames, rewards (scores) and actions collected when our agents playing the cartpole and different Atari games, including Breakout, Taxi, Pong and Space Invaders. The game environments used were simulated by Open AI Gym so that our agents can play games there. States in our models were preprocessed images of the screen. For example, in breakout, images were 210x160x3 RGB colors. They were then converted to grey color and scaled to a 84 x 84 box. 4 consecutive frames were used to feed our models. In order to treat these data for training our agents, we started by assigning an action space for our agents. For example, "left" and "right" were assigned in action space for cartpole. After that, we started off our agents by randomly assigning an action to them. Rewards (score) and next state were observed. Data from the transition was then collected as a tuple (state, action, reward, next state, terminal). A memory was set so

that certain number of most recent tuples were used to fit our models.

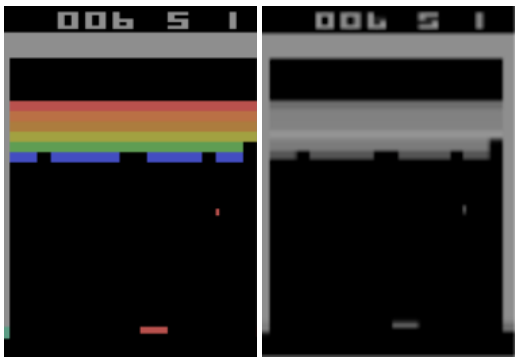

Figure 1: Frames before and after preprocessing

4. Methods

We implemented different reinforcement learning methods on different game environments and obtain the results of rewards each methods gained after different number of episodes. We would apply ddqn, dqn, cem, sarsa, q-learning and a3c on cartpole and ssveral Atari games. We would also compare the performances of dqn under different hyperparameters setting when playing cartpole.

For Q-learning, we first have the Bellman function

$$v(s) = \mathbb{E}[R_{t+1} + \lambda v(S_{t+1})|S_t = s] \tag{1}$$

Then we define the Action-Value function as

$$Q^\pi(s,a) = \mathbb{E}_{s'}[r + \lambda Q^\pi(s',a')|s,a] \tag{2}$$

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)] \tag{3}$$

and the advantage function as

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s,a) \tag{4}$$

We have

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \tag{5}$$

$$\implies Q^*(s,a) = \mathbb{E}_{s'}[r + \lambda \max_{a'} Q^*(s',a')|s,a] \tag{6}$$

For forward pass,

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s,a;\theta_i))^2] \tag{7}$$

where

$$y_i = \mathbb{E}_{s'\sim\epsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a] \tag{8}$$

For backward pass,

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\varepsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)] \tag{9}$$

For Deep Q Learning, we use experience replay. We continually update a replay memory table of transitions $(st, at, rt, st+1)$ as game episodes are played, and train Q-network on random mini-batches of transitions from the replay memory, with algorithm shown in figure 2.

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 ϵ 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 $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2: Deep Q-learning Algorithm

The DDQN algorithm is another version of DQN, with the target y_i replaced by

$$y_i^{DDQN} = r + \gamma Q(s', \underset{a'}{\text{argmax}} Q(s', a'; \theta_i); \theta^-) \tag{10}$$

We might be tempted to construct the aggregating module,

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) \tag{11}$$

A. Double DQN Algorithm

Algorithm 1: Double DQN Algorithm.

input : $\mathcal{D}$ – empty replay buffer; θ – initial network parameters, θ^- – copy of θ
input : N_r – replay buffer maximum size; N_b – training batch size; N^- – target network replacement freq.
for *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  for $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_\mathcal{B}$
    Sample next frame x^t from environment $\mathcal{E}$ given (s, a) and receive reward r, and append x^t to $\mathbf{x}$
    if $|\mathbf{x}| > N_f$ **then** delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ **end**
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s, a, r, s') to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
    Sample a minibatch of N_b tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
    Construct target values, one for each of the N_b tuples:
    Define $a^{\max}(s';\theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s';\theta); \theta^-), & \text{otherwise.} \end{cases}$
    Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
  end
end

Figure 3: Double Deep Q-learning Algorithm

Another algorithm we adopt is policy gradient algorithm, shown in Figure 4. As we have parametrized policy

$$\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\} \tag{12}$$

For Policy Gradient, we define

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\right] \tag{13}$$

$$\implies J(\theta) = \int_\tau r(\tau) p(\tau;\theta) d\tau \tag{14}$$

$$\nabla J(\theta) = \mathbb{E}_{\tau \backsim p(\tau;\theta)}[r(\tau)\nabla_\theta \log p(\tau, \theta)] \quad (15)$$

We have

$$\nabla J(\theta) \approx \sum_{t \geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t) \quad (16)$$

Here we used an actor-critic approach based on the DPG algorithm,

$$\nabla J(\theta) \approx \mathbb{E}_{s_t \backsim \rho^\beta}[\nabla_\theta Q(s, a|\theta)|_{s=s_t, a=\mu(s_t|\theta)}]$$

$$= \mathbb{E}_{s_t \backsim \rho^\beta}[\nabla_\theta Q(s, a|\theta)|_{s=s_t, a=\mu(s_t)} \nabla_\theta \mu(s|\theta)|_{s=s_t}] \quad (17)$$

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 4: Policy Gradient Algorithm

For the Asynchronous Advantage Actor-Critic method (A3C):

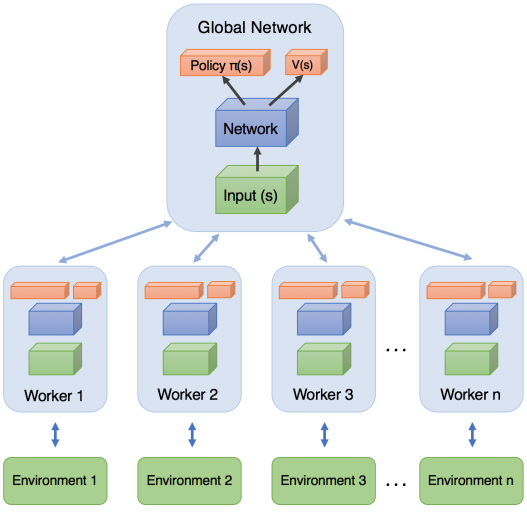

Figure 5: Asynchronous Advantage Actor-Critic

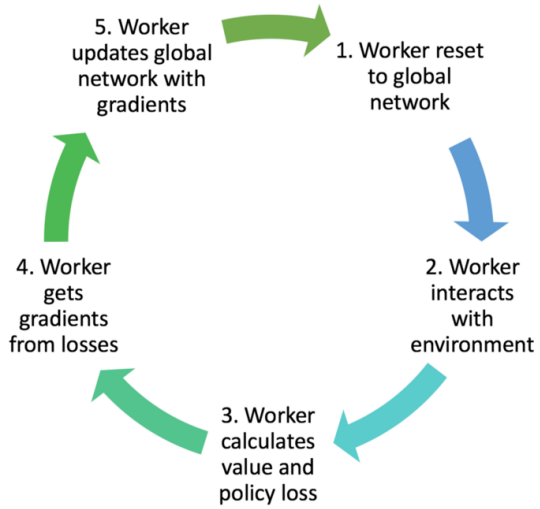

Figure 6: Asynchronous Advantage Actor-Critic

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t; \theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i; \theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 7: Asynchronous Advantage Actor-Critic

In A3C there is a global network, and multiple worker agents which each have their own set of network parameters (shown in figure 5). Each of these agents interacts with its own copy of the environment at the same time as the other agents are interacting with their environments. This network will consist of convolutional layers to process spatial dependencies, followed by an LSTM layer to process temporal dependencies, and finally, value and policy output layers.

In the case of A3C, our network will estimate both a value function $V(s)$ and a policy $\pi(s)$. These will each be separate fully-connected layers sitting at the top of the network. Critically, the agent uses the value estimate to update the policy more intelligently than traditional policy gradient methods. Figure 7 is example code for establishing the network graph itself.

Using advantage estimates rather than just discounted re-

turns allows the agent to determine not just how good its actions were, but how much better they turned out to be than expected. The Advantage Estimate: $A = R - V(s)$ where $R = \gamma(r)$.

For the cross-entropy (CE) method, it is a Monte Carlo method for importance sampling and optimization, as in figure 8.

Suppose we want to estimate

$$\mathbb{E}_u[H(x)] = \int H(x) f(x; u) dx \tag{18}$$

If $H(x)$ is a low probability event, then we have to adopt the importance sampling, and the expectation becomes

$$\mathbb{E}_u[H(x)] = \frac{1}{N} \sum_{i=1}^{N} x_i' \frac{f(x; u)}{f(x; v)} \tag{19}$$

So we have to take educated random guesses on actions. Select top performers of guesses and use them as seeds for next round of guessing, as shown in figure 8.

1. sample $\mathbf{A}_1, \ldots, \mathbf{A}_N$ from $p(\mathbf{A})$
2. evaluate $J(\mathbf{A}_1), \ldots, J(\mathbf{A}_N)$
3. pick the *elites* $\mathbf{A}_{i_1}, \ldots, \mathbf{A}_{i_M}$ with the highest value, where $M < N$
4. refit $p(\mathbf{A})$ to the elites $\mathbf{A}_{i_1}, \ldots, \mathbf{A}_{i_M}$

Figure 8: Cross Entropy Method

State–action–reward–state–action (SARSA) is an algorithm for learning a Markov decision process policy. SARSA is similar to Q-Learning , while the key difference between SARSA and Q-learning is that SARSA is an on-policy algorithm. It implies that SARSA learns the Q-value based on the action performed by the current policy instead of the greedy policy, shown in figure 9.

The update rule for SARSA is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

We will evaluate our results by comparing rewards against no. of episodes in different environments by different methods. We would show these comparison by plotting charts, graphs and tables or with some data visualization.

5. Experiments

For the Cartpole environment, we set the number of steps taken by all these methods to be 50000 number of steps, as in figure 10. Under the same condition with fixed factors, these reinforcement learning methods are comparable.

By our observation, there are some oscillations on graph of DQN and the DQN reinforcement learning algorithm have relatively unstable improvement over the training process, as in figure 11 and 12. DDQN has a relatively stable

SARSA(λ): Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
- Sates $\mathcal{X} = \{1, \ldots, n_x\}$
- Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
- Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
- Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
- Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
- Discounting factor $\gamma \in [0, 1]$
- $\lambda \in [0, 1]$: Trade-off between TD and MC
- **procedure** QLEARNING($\mathcal{X}$, A, R, T, α, γ, λ)
  - Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
  - Initialize $e : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ with 0. ▷ eligibility trace
  - **while** Q is not converged **do**
    - Select $(s, a) \in \mathcal{X} \times \mathcal{A}$ arbitrarily
    - **while** s is not terminal **do**
      - $r \leftarrow R(s, a)$
      - $s' \leftarrow T(s, a)$ ▷ Receive the new state
      - Calculate π based on Q (e.g. epsilon-greedy)
      - $a' \leftarrow \pi(s')$
      - $e(s, a) \leftarrow e(s, a) + 1$
      - $\delta \leftarrow r + \gamma \cdot Q(s', a') - Q(s, a)$
      - **for** $(\tilde{s}, \tilde{a}) \in \mathcal{X} \times \mathcal{A}$ **do**
        - $Q(\tilde{s}, \tilde{a}) \leftarrow Q(\tilde{s}, \tilde{a}) + \alpha \cdot \delta \cdot e(\tilde{s}, \tilde{a})$
        - $e(\tilde{s}, \tilde{a}) \leftarrow \gamma \cdot \lambda \cdot e(\tilde{s}, \tilde{a})$
      - $s \leftarrow s'$
      - $a \leftarrow a'$
  - **return** Q

Figure 9: State–action–reward–state–action

Paramters setting on different reinforcement learning methods in Cartpole	setting
target model update rate	1.00E-02
memory limits	50000
optimizer	Adam with learning rate 1e-3
step size	50000

Figure 10: Hyper parameters setting

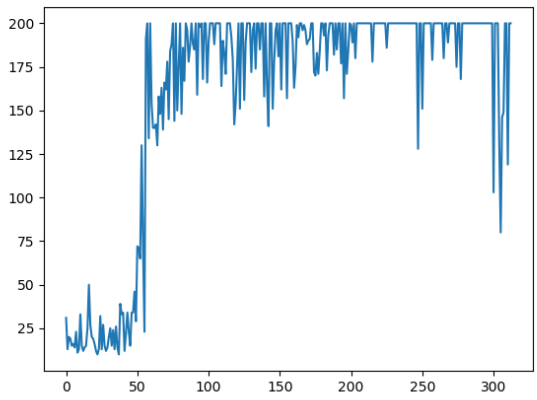

Figure 11: DQN graph of reward against episode on Cartpole

progress and converge better than DQN, even if the time for both algorithm first reach the bounded reward is similar. After the rewards of the DDQN agent reach the maximum point of the graph, they keep at the highest position without lowering the scores, meaning that DDQN actually provides a stably increasing path for the neural network. In contrast, after the DQN rewards reached the highest point,

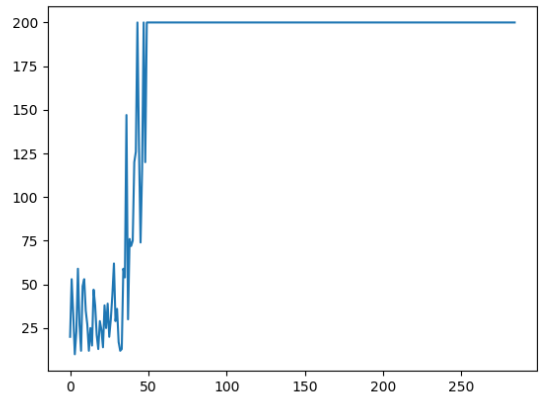

Figure 12: DDQN reward against episode on Cartpole

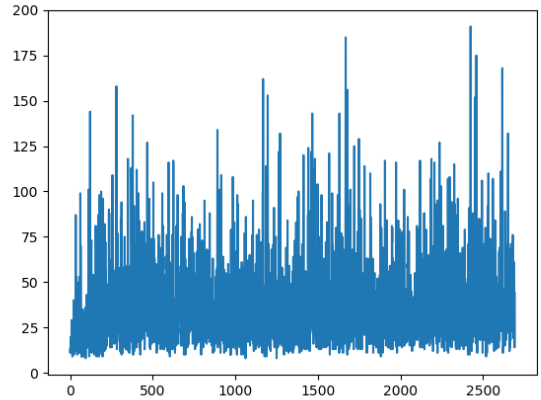

Figure 13: CEM reward against episode on Cartpole

it will lower at some points on the graph, and even be reduced to around half of the highest one, which means that the DQN is improving the rewards very slowly, and not very stable. So in the same period of time, DDQN is able to achieve higher scores than DQN.

In particular, DQN is just Q-learning, which uses neural networks as a policy and use method like experience replay, target networks and reward clipping, and there are a couple of statistical problems. First, DQN approximate a set of values that are very interrelated while DDQN solves this problem. Another reason is that DQN tend to be overoptimistic. It will over-appreciate being in this state although this only happened due to the statistical error, which DDQN solve as well.

Also, by decoupling the estimation $Q(s,a) = V(s) + A(s,a)$, our DQN calculated values of each action at a given state and DDQN learn which states are valuable without

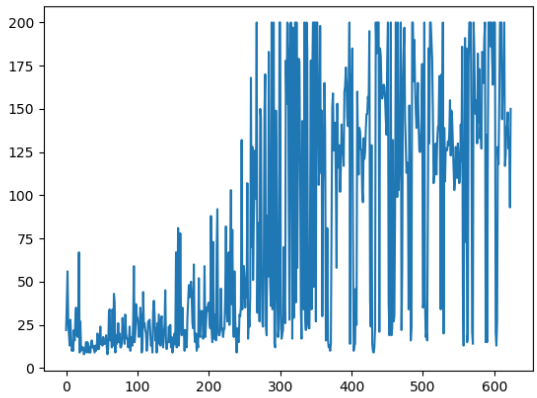

Figure 14: SARSA reward against episode on Cartpole

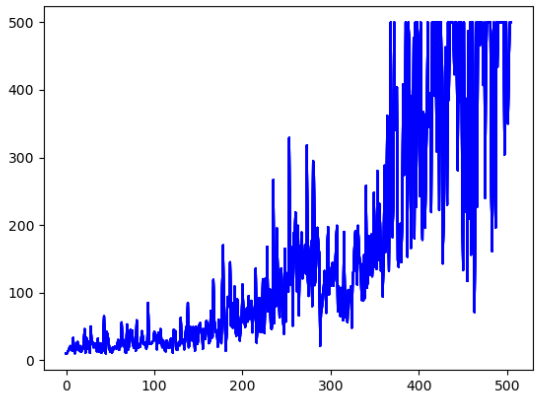

Figure 15: A3C reward against episode on Cartpole

Sarsa	Dqn	Ddqn	CEM	A3C
2214.71813	430.90229	244.5472055	359.0261631	614.3071

Figure 16: Total variance of rewards against consecutive episode

Comparison on different limits in memory	Variance of difference in absolute rewards against consecutive espisode
Dqn (limit set to 1000):	1828.486825
Dqn(limit set to 20000):	578.7411021
Dqn(limit set to 50000, original limit)	430.90229

Figure 17: Total variance under different limit

having to learn the effect of each action at each state, since it is also calculating V(s).

For the cross entropy method, which is a model free, policy based algorithm, we observe that the rewards does not increase, not converge, and keep oscillating. One reason behind is that cross entropy is a relatively simple method compared with other methods we mentioned, and this cartpole environment is too complicated for it to work, since

this method only works in simple environment, and does not work well in complicated environment that require complex policies.

For the SARSA, we can see that although the rewards over episodes are increasing over time, the model does not increase as fast as the DQN and DDQN, which means that this is actually not a very efficient algorithm even if cartpole is a simple environment. Also, after 300 episode, the oscillation of rewards is very large and it does not converge with the largest variance compared with other methods, as described in figure 14, meaning that this is a very unstable method under the cartpole environment.

For A3C, the training process steadily increased, with a very smooth trajectory. But it seems that the rewards does not increase increase as fast as DQN and DDQN. The variance also increases when the rewards increase, which means it becomes more unstable over time.

Testing with different hyper-parameters under Cartpole environment: Rewards vs number of episodes by DQN setting memory storing different limits of recent transitions before fitting the model.

We would also like to see the effects by changing hyper parameters on different reinforcement learning methods. In this project, we have picked DQN for doing this experiment in cartpole. This time, we changed the number of previous transitions stored in memory before fitting the model. We tried it from limit of 1000, 20000 and 50000 (original limit) respectively, as in figure 17, 16, and 10 respectively. Surprising, we found out that if we increase the number of limits of transitions in memory before fitting the model, the variance of difference in rewards against consecutive episodes will increase. i.e. Var(abs(previous episode reward - current episode reward)), as in figure 17. Therefore, we would make a conclusion that keeping the original limit (50000) on Dqn would make the models to converge faster and train more efficiently.

For breakout environment, we can easily observe that Q Learning does not converge and not improve over time, while DQN and A3C both have improvement, shown in figure 20, 21, 22. One reason is that Q Learning does not have the experience replay. The second reason is that the environment is much more complicated than cartpole as it is a non markovian environment. We can also observe that DQN and A3C have similar performance. So DQN and A3C can deal with more complicated, complex and non markovian environments.

For taxi and centipede environments, we use SARSA and Q Learning methods. The difference between these two is the max function because SARSA is a on-policy algorithm. SARSA will update the Q table used for action selection. However, As a off-policy algorithm, Q Learning does not necessarily updated the action selection, but Q table will be updated with each action taken using the update equation.

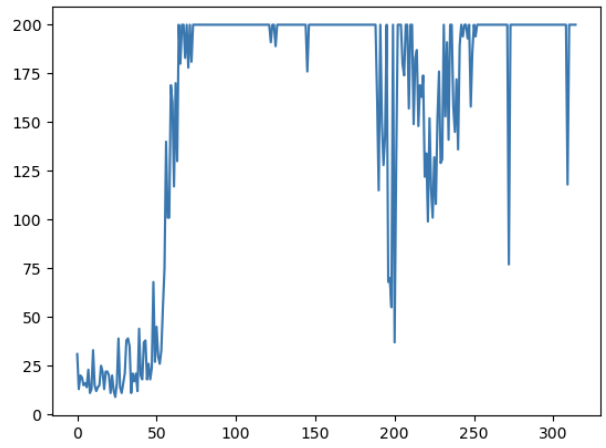

Figure 18: DQN reward against episode on Breakout with limit = 20000

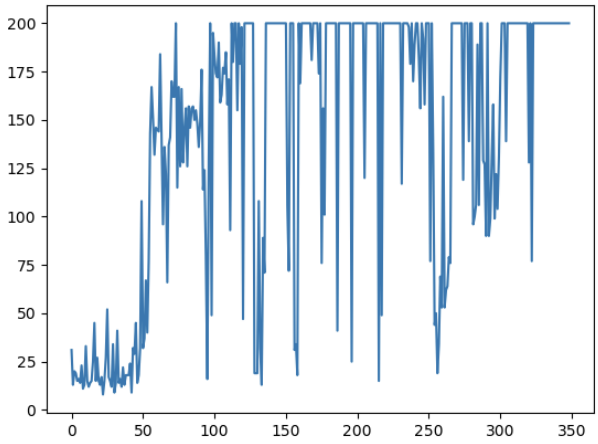

Figure 19: DQN reward against episode on Breakout with limit = 1000

Since these two methods are based on Markov assumption, we have similar results produced by them under the Taxi and Centipede environments.

For taxi environment, the rate of convergence is different between SARSA and Q Learning occurs because of epsilon-greedy implementation. SARSA converges faster than Q Learning under the taxi environment as it takes the epsilon probability of random action into consideration when its policy is updated.

Both methods successfully increased their rewards and converge to the maximum point over time under the taxi environment, while they do not improve the performance of agents and keep oscillating under the centipede environment. The reason behind is that centipede environment is a stochastic, non-markovian environment since there are some monsters randomly appearing near the player, which

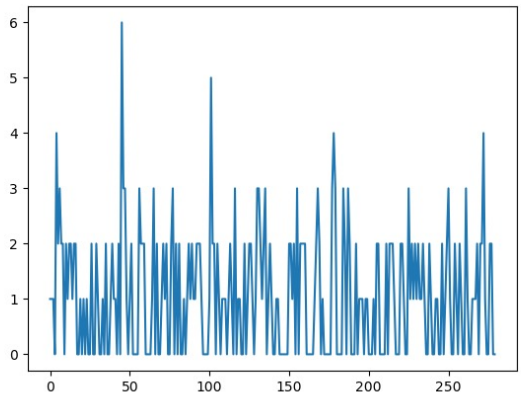

Figure 20: Q-Learning reward against episode on Breakout

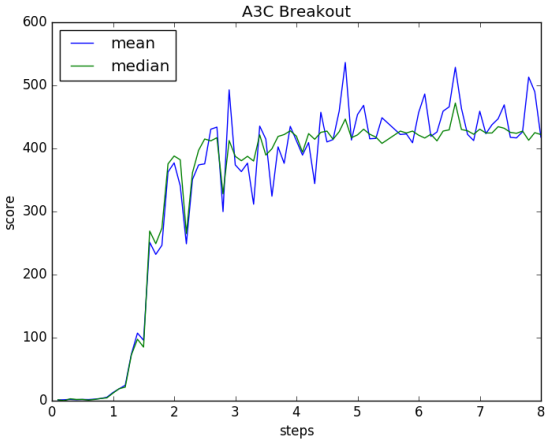

Figure 22: A3C reward against episode on Breakout

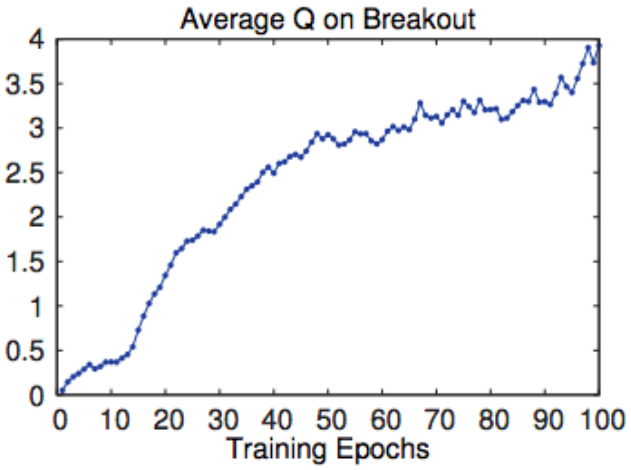

Figure 21: DQN reward against episode on Breakout

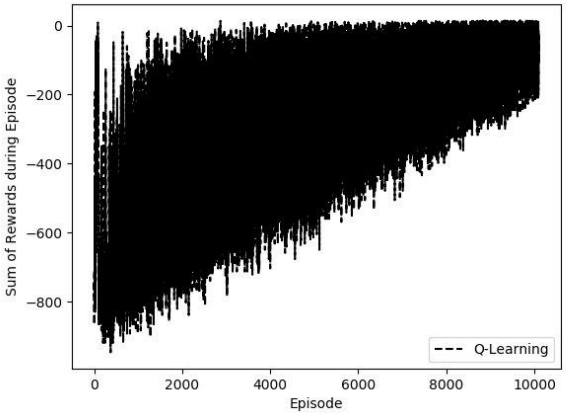

Figure 23: Q Leaning reward against episode on Taxi

creates some unexpected factors and thus a stochastic environment. It implies that both Q Learning and SARSA cannot deal with some complicated and non markovian environments.

So, Naive Q and SARSA converges on markovian environments, but not on non-markovian environments.

6. Conclusion

From our experiments, we found out that DDQN performed the best by quickly converge to obtain high rewards in few number of episodes in Cartpole, while CEM performed the worst in Cartpole. SARSA also did not perform well in Cartpole with high variance in rewards against number of episodes when training.

In Breakout, DQN and A3C performed the best, while Q-learning performed the worst which did not converge well when training.

In Taxi, SARSA converged faster than q-learning, and both methods performed well.

In centipede, which is a game with randomness, SARSA and Q Learning both did not converge while training and had bad performance.

When we increased the number of limits in transitions for memory before fitting models, we found that it would perform better in DQN

The rate of convergence difference between SARSA and Q Learning occurs because of epsilon-greedy implementation.

Deep Q-Learning using Experience Replay can solve the problem with non-markovian property.

SARSA and Q Learning can only improve rewards and converge under markovian environment, such as Cartpole and Taxi, but cannot under non markovian environment,

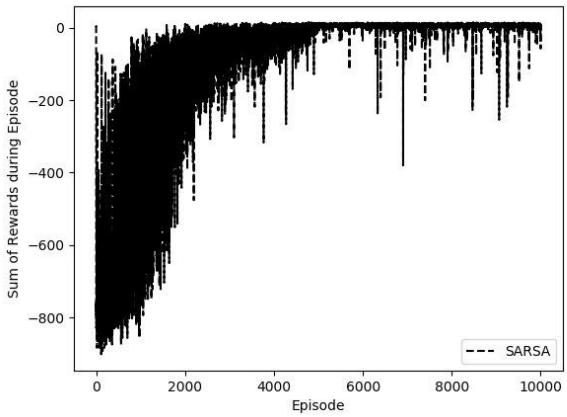

Figure 24: SARSA reward against episode on Taxi

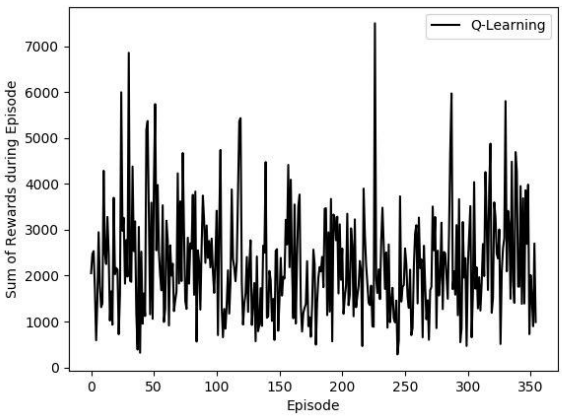

Figure 25: Q Learning reward against episode on centipede

such as breakout and centipede.

7. Supplementary Notes

7.1. Odds encountered

We have encountered many odds during the progress of the project. One of the difficulties we face is the installation and implementation of libraries on both Dev- cloud and Azure, the cloud computing platform by Intel and Microsoft respectively. The syntax on terminal is very different from the usual one as it requires additional authorization and permission, and there are some conflicts due to environmental errors and variety of versions.

Due to the lack of GPU of our computers, we try to finish the training in virtual machines on Cloud. However, running reinforcement models for playing Atari Games on cloud is very costly.

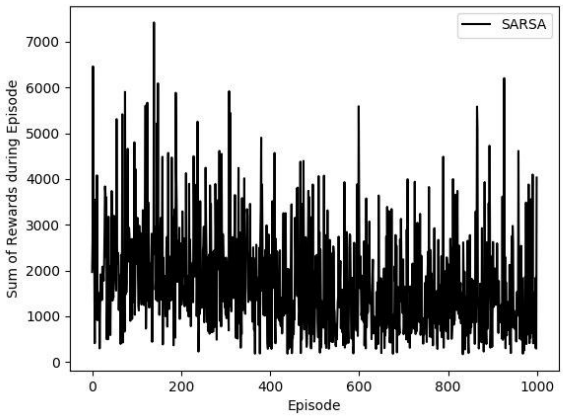

Figure 26: SARSA reward against episode on centipede

Moreover, even using GPU it takes so long to train agents in Atari games. Normally it takes 4-5 days if we want the agent to be trained and performed as well as human-beings. As a result, it is impossible to compare wide range of reinforcement methods in Atari Games given time limitations.

7.2. Limitations

Some methods like SimPLe has only been described in published paper and its corresponding source code has not been open-sourced. Therefore, it is difficult to imitate their results from what were shown from the papers published. This put limitations for our implementations on new reinforcement learning methods on game environments.

Moreover, running the algorithm on different machine may create differences. Many factors like training hardwares are difficult to be the same as what the paper authors were using. Different results might be caused due to these factors which are supposed to be control variables in the research.

7.3. Further Investigations

We would like to apply different reinforcement learning methods on 3D-environments in the future, like training ants in MuJoCo. We would also plan to apply visualization dashboards on more different Atari Games or gaming environments, instead of Breakout only.

7.4. Supplementary Materials

Videos:

`https://www.youtube.com/watch?v=wCuF8tyJtoE`

`https://www.youtube.com/watch?v=ggyNBrlnrv0`

`https://www.youtube.com/watch?v=Dip9uNazXXM&feature=youtu.be`

https://youtu.be/9SmLs2v8kD4

Code:

https://github.com/wslukaa/Comp4901J_FinalReport/tree/master

Visualization:

https://github.com/wslukaa/Comp4901J_FinalReport_Visualization/tree/master

References

[1] Balduzzi, D., Garnelo, M., Bachrach, Y., Czarnecki, W. M., Perolat, J., Jaderberg, M. & Graepel, T. (2019), arXiv e-prints, arXiv:1901.08106.

[2] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16), Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. JMLR.org 1928-1937.

[3] P Kroese, Dirk & Y Rubinstein, Reuven & Porotsky, Sergey & L D Ltd, A & Taimre, Thomas. (2012). Cross-Entropy Method.

[4] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double Q-Learning. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16). AAAI Press $2094 - 2100$.

[5] Corazza, Marco and Sangalli, Andrea, Q-Learning and SARSA: A Comparison between Two Intelligent Stochastic Control Approaches for Financial Trading (June 10, 2015). University Ca' Foscari of Venice, Dept. of Economics Research Paper Series No. No. 15/WP/2015.Available at SSRN: https://ssrn.com/abstract=2617630 or http://dx.doi.org/10.2139/ssrn.2617630