Playing Atari Game with Deep Reinforcement Learning Methods

Biao SHI, Haiyang JIANG, Jingnan CAO, Ranyu FAN, Yuyan ZHAO

***Abstract*—We present different reinforcement learning models and strategies to learn a control policy from 2-Dimension RGB images. The models include Convolutional Neural Networks along with fully connected neural networks, based on Deep Q-learning, Double Deep Q-learning, Dueling Network, Actor-Critic, and Advantage Actor-Critic methods. We applied and evaluated these methods to an Atari game named "Riverraid" from the GYM environment, and we optimized them with different strategies including Multi-step TD Target, Experience Replay, and REINFORCE. The best result we achieved is by an Advantage Actor-Critic model with REINFORCE after 20,000 episodes and 6 hours training. The agent is able to earn an average accumulative reward of 1250.3 and play until the third scenario of the game.**

I. Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision is one of the challenges of reinforcement learning. With the emergence of Convolutional Neural Networks (CNN)[3], deep learning networks are able to extract high-level features from raw sensory data, which made it possible to directly process the states of video game.

With CNN and fully connected neural network, we apply multiple methods to construct the model: Deep Q-learning which trains the Q-function by a single network; Double Deep Q-learning which trains two separate Q-function by two networks to overcome the overestimation of single DQN; Dueling Network which has two streams to separately estimate V-value, and takes advantages of each action to increase the stability of the optimization using a common CNN; Actor-Critic which trains two independent networks, one as actor to generate the best action for given state, and the other as critic to judge the score of this action; Advantage Actor-Critic which adds a baseline in the policy gradient process to reduce the variance of prediction and accelerate the convergence.

Moreover, we optimize the methods using novel strategies include: REINFORCE to update the policy function after each completed episode of the game using the idea of Monte Carlo; Multi-step TD Target to compute rewards of multiple steps as TD target to update the Q-value with lower variance; Experience Replay to train the network on random data in the past transitions instead of the current sequence, avoiding training on consequent and highly correlated states.

Our code can be found here: [1]. It is developed in Google Colab with GPU support. GIF of the trained agent to play the game can be seen in the folder "/gif".

[1]The link to our code: https://drive.google.com/drive/folders/1i5BigdhJ3q-haYgeFg5mNM17qkTcQpU5?usp=sharing or https://github.com/iLori-Jiang/Playing-Atari-with-RL

II. Background

A. Atari Environments

The Atari environments provided by Gym[1] are a collection of classic Atari video games, which is an interesting tool to develop reinforcement learning algorithms. In our project, we choose the game "Riverraid" as the environment. In "Riverraid", the player need to avoid enemy objects and shoot them as many as possible. The player can move left and right, and fire the weapon. The map is procedurally generated, meaning that each play round offers a different layout of obstacles and challenges.

B. Deep Q-learning Network

Deep Q-learning Network (DQN) is a widely used reinforcement learning algorithm, which is developed based on Q-learning. Q-learning aims to learn the optimal policy by following the update rule, where R_t is the reward at this time-step, α is the learning rate, and γ is the discount factor.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Each pair of states and actions are stored in the Q-table during the update. However, Q-table does not perform well when there are too many pairs, or the state-action space is continuous as in the case of our project. In order to handle the environment with a large number of states and actions, we choose Deep Q-learning algorithm which uses deep Neutral Network to approximate the Q-values for each action based on the state, instead of using Q-table.

$$Q(S, A) = q_w(S)_A$$

$$w \leftarrow w + \alpha[R_t + \gamma \max_{a \in A} q_w(S_{t+1})_a - q_w(S_t)_{A_t}] \nabla_w [q_w(S_t)]_{A_t}$$

where q_w is a deep Neutral Network with parameters w.

C. Double Deep Q-learning Network

As our environment is non-stationary, the shortcoming of DQN is that it may overestimate the Q-value in a given state and stuck with one specific action. Double Deep Q-learning Network (DDQN) is an improvement of DQN, which use two networks of the same structure. One network is used to learn the replay just like DQN. And the other is a copy of the last episode of the first one, which has a lower difference between values than the main model. Therefore, we can update the Q-value as following:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma Q'(S_{t+1}, a) - Q(S_t, A_t)]$$

$$a = \max_{a \in A} Q(S_{t+1}, a)$$

where $Q'(S_{t+1}), a$ is given by the secondary model.

D. Dueling Network

Dueling DQN is an improvement of DQN for accurate estimation of Q-values. It represents two separate estimators: one for the state value function and one for the state-dependent action advantage function. As presented in the paper[5], for an agent behaving according to a stochastic policy π, the values of the state-action pair (s, a) and the state s are defined as follows:

$$Q^\pi(s, a) = \mathbf{E}[R_t | s_t = s, a_t = a, \pi]$$

$$V^\pi(s) = \mathbf{E}_a[Q^\pi(s, a)]$$

We define the advantage function, relating the value and Q functions:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \tag{1}$$

From those definitions, we can find out that

$$\mathbf{E}_a[A^\pi(s, a)] = 0 \tag{2}$$

In the Dueling Network architecture, the equation 1 can be rewritten as

$$Q^\pi(s, a; \theta, \alpha, \beta) = V^\pi(s; \theta, \alpha) + A^\pi(s, a; \theta, \beta) \tag{3}$$

where θ represents the parameters common on the neural network, α represents the parameters on the value stream and β represents the parameters on the advantage stream.

The equation 3 is unidentifiable in the sense that given Q we cannot recover V and A uniquely. To address this issue of identifiability, we can add a constraint according to the equation 2.

$$\begin{aligned} Q^\pi(s, a; \theta, \alpha, \beta) =& V^\pi(s; \theta, \alpha) \\ &+ \big(A^\pi(s, a; \theta, \beta) - \frac{1}{|\mathcal{A}|} \sum_{a}^{'} A^\pi(s, a'; \theta, \beta)\big) \end{aligned}$$

This can increase the stability of the optimization: the advantages need to change as fast as the mean.

E. Actor-Critic

Different from the previous methods which are focusing on Q-value, policy gradient methods focuse on updating the control policy of the agent, and are ubiquitous in model free reinforcement learning algorithms. They play the "actor" part of Actor-Critic methods.

In essence, policy gradient methods update the probability distribution of actions, so that actions with higher expected reward have a higher probability value for an observed state. Its goal can be defined as:

$$J(\theta) = \mathbf{E}[\sum_{t}^{T} r_t] \tag{4}$$

which is to learn a policy that maximizes the cumulative future reward to be received starting from any given time t until the terminal time T. Since this is a maximization problem, we take the gradient ascent to update the parameters:

$$\theta_{t+1} \leftarrow \theta_t + \nabla_\theta J(\theta) \tag{5}$$

$$\nabla_\theta J(\theta) = \mathbf{E}_\tau[\sum_{t=1}^{T} \nabla_\theta log \pi_\theta(a_t | s_t)(\sum_{t'=t+1}^{T} \gamma^{t'-t-1} r_{t'})] \tag{6}$$

where $\gamma \in [\mathbf{0}, \mathbf{1}]$ is the discount factor in order to weight immediate rewards more than future rewards. Thus, we have declared the method to update the control policy of the agent, which can plays the "actor" role to generate action given certain state.

Moreover, we can recall that actually the final part of the equation can be estimated by Q-value function, where ω is the parameter of the function:

$$\mathbf{E}_\tau[\sum_{t'=t+1}^{T} \gamma^{t'-t-1} r_{t'}] = Q_\omega(s_t, a_t) \tag{7}$$

Therefore, we can introduce Q-value estimation into the update process:

$$\nabla_\theta J(\theta) = \mathbf{E}_\tau[\sum_{t=1}^{T} \nabla_\theta log \pi_\theta(a_t | s_t) Q_\omega(s_t, a_t)] \tag{8}$$

where Q function can play as "critic" role to judge the score of the action given by the "actor".

In a nut shell, to implement Actor-Critic method we need to train two independent neural network, one as "actor" and the other as "critic". The goal of the "critic" is the same as the DQN to predict the Q value as precise as possible. While the goal of the "actor" is to gain as much as score from the "critic", in another world the "actor" is trained by the "critic".

By using this method, we can avoid the poor convergence issues faced by pure Q learning, and have much smoother learning curves and performance improvement guarantees with every update thanks to policy gradient.

F. Advantage Actor-Critic (A2C)

Given the advantages of Actor-Critic method, it still have some glaring issues includes noisy gradients and high variance if we update the policy parameter through Monte Carlo method (i.e. taking random samples). Since each episode the agent plays can deviate from each other at great degrees, it introduces inherent high variability in log probabilities and cumulative reward values in equation. 6.

Besides the issues with gradients, another problem is that the episodes might have a cumulative reward of zero. The essence of policy gradient is increasing the probabilities for "good" actions and decreasing those of "bad" actions in the policy distribution. If the cumulative reward is zero, the agent cannot know whether a series of actions is "good" or "bad" and thus cannot learn from it.

Therefore, a possible solution to reduce variance and increase stability is by subtracting the cumulative reward by a baseline. Intuitively, this means how much improvement it is

to take a specific action compared to the average action at the current state. By this, most zero cumulative rewards will become negative so that the agent can recognize them as "bad" actions. We can use V value as the baseline, to subtract it from the Q value in equation. 8, which can be called as Advantage:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{9}$$

Moreover, consider the relationship between Q value and V value:

$$Q(s_t, a_t) = \mathbf{E}[r_t + \gamma V(s_{t+1})] \tag{10}$$

we can rewrite the Advantage equation as:

$$A_v(s_t, a_t) = r_t + \gamma V_v(s_{t+1}) - V_v(s_t) \tag{11}$$

where v is the parameter for the V value function. Thus, we use V value instead of Q value to update the policy. Besides, we can see this equation is in the same form of TD target.

We can further rewrite the policy gradient function in the equation. 8 to be:

$$\nabla_\theta J(\theta) = \mathbf{E}_\tau[\sum_{t=1}^{T} \nabla_\theta log \pi_\theta(a_t|s_t) A_v(s_t, a_t)] \tag{12}$$

G. Optimizing Strategies

Besides the methods above, we can optimize the training process with following strategies.

1) REINFORCE: For policy gradient methods like Actor-Critic, our goal is as the equation. 4, where the expectation is hard to compute since we have no clues about the probability distribution. Therefore, we can use the Monte Carlo idea to replace the expectation by playing big amount of episodes N and compute their average cumulative reward:

$$\mathbf{E}[\sum_{t}^{T} r_t] = \frac{1}{N} \sum_{n=1}^{N} (\sum_{t}^{T} r_t) \tag{13}$$

In this way, we can use the cumulative reward of a single episode to update the policy gradient, and play big amount of episodes to simulate the probability distribution. However, one drawback of this strategy is that we cannot update the policy before an episode is done. This can lead to slow update when training in a long episode scenario.

2) Multi-step TD Target: Temporal Difference (TD) learning is used as the goal in Q-learning by focusing on the differences between the agent experiences and its prediction in time. The methods aim to provide and update some estimate V/Q-value and update as the agent experiences them.

The most basic method for TD learning is the TD(1) method. However, it share the same idea with stochastic gradient descent, which will introduce high variance in gradient and lead to divergence[2]. Therefore, we can use the TD target of multiple steps (λ) to accumulate the multiple real rewards, which can be closer to ground true, lower the variance of gradient and facilitate convergence.

$$\mathbf{TD_{Error}}(\lambda) = Q(s_t, a_t) - (\sum_{i=0}^{\lambda-1} (\gamma^i r_{t+i}) + \gamma^\lambda \max_a Q(s_{t+\lambda}, a))$$

3) Experience Replay: In the scenario of consequent episode such as video games, the states are highly correlated in an single episode, and the difference between two consecutive frames is usually trivial. Since most deep learning algorithms assume the data samples to be independent, training on high correlation data can decrease the robustness and stability of the agent.

Therefore, to solve the problems above, experience replay method is introduced. We store the agent's experiences at each time-step, $\mathbf{e_t} = (\mathbf{s_t}, \mathbf{a_t}, \mathbf{r_t}, \mathbf{s_{t+1}})$ in a data-set $\mathbf{D} = \mathbf{e_1}, ..., \mathbf{e_N}$, pooled over many episodes into a replay memory. During the inner loop of the algorithm, we apply Q-learning updates, or mini-batch updates, to samples of experience from the memory.

This approach has several advantages over standard online Q-learning: (1) Each step of experience is potentially used in many weight updates, which allows for greater data efficiency. (2) It breaks the correlations and therefore reduces the variance of the updates. (3) The behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters.

III. Methodology

A. Gym Environment

As described, we use the Atari game "Riverraid" as the basic environment. The main features of our environment are the following:

- State Space: The agent can observe an RGB image of size 210×160 pixels, including a score bar on the bottom. Therefore, the value of the observation is continuous.

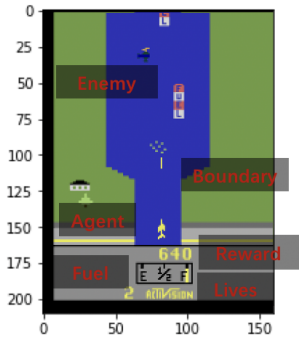

Fig. 1: Image of the environment "Riverraid"

- Action Space: Generally, there are 18 actions we can take in the environment. But we mainly consider the following 4 actions.

Code	Action
0	Stay Still
1	Start Game/Shoot
2	Move Right
3	Move Left

TABLE I: Action space of the environment "Riverraid"

- Reward: The reward is obtained by destroying enemy objects with the action taken. If the action taken does

not break any objects, the reward will be zero. Besides, different types of objects bring different rewards.

Object	Reward
Tanker	30 points
Helicopter	60 points
Fuel Depot	80 points
Jet	100 points
Bridge	500 points

TABLE II: Reward function of the environment "Riverraid"

The agent have 5 lives to play a game, and when the agent touches the wall/enemy or runs out of fuel, it will lose one life. A final accumulative reward will be displayed on the screen when the agent lose all its lives. In the training phase, we limit the agent to only one life and take the final accumulative reward of this life as the metric.

B. Common Network Structure

First we require a CNN structure to extract the feature of the images from the game to enable the agent understand the states of the game. The input to the extract architecture consists of an $84 \times 84 \times 4$ image produced by the preprocessing map. The first hidden layer convolves 32 filters of 8×8 with stride 4 and applies a ReLU activation function for nonlinearity. The second hidden layer convolves 64 filters of 4×4 with stride 2, again followed by a ReLU. The third convolutional layer that convolves 64 filters of 3×3 with stride 1 followed by a ReLU. Thus, the final output feature is of size $7 \times 7 \times 64$, and we can flatten the feature into 1×3136. This flattened feature will be learned by the agent to understand the states of the game.

In the following steps, this feature will be feed into the final hidden layer, which is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a multiple output for the value of the state *V(state)* or the value for each valid action *Q(state, action)* or the probability of each valid action as control policy Π*(state)*.

C. Detailed Implementations of Algorithms

The implementations of different algorithms are all using the same network structure as above, only with the difference in the final output dimension and the method of defining the loss and updating the parameters. The details are illustrated in the V-A section.

Common	
Training Episode	3,000
Max Step	10,000
Learning rate	0.001
Gamma	0.99
Optimizer	Adam
DQN	
Initial epsilon	0.99
Decay rate of epsilon	0.995
Epsilon threshold	0.10
Synchronize frequency	5
Experience Replay	
Memory size	10,000
Batch size	64
Training Episode	1,000

TABLE III: Fixed hyper parameters for all the algorithms

To compare the performance of the algorithms, we fix the hyper parameters. It should be noticed that since experience replay method would train for more time (on each step of the episode, the agent will update the parameters on replays of batch size, instead of updating them after one episode is finished), we set the training episode of experience replay method to 1,000 instead of 3,000 to balance the training time.

IV. Results and Discussion

As illustrated, we use accumulative reward, which is the total reward received in an episode of game, as the major metric to compare the algorithms. The more enemy objects the agent destroys in the game, the bigger the reward it gets. The second minor metric is the runtime of an episode, which means the duration the agent survived in the game. The longer the agent survives in the game, the more chances it has to earn rewards.

Following is our training result for all the algorithms shown in Fig. 2 (only the average reward at is showed, the graph of reward and runtime will be shown in V-B section). Note that as illustrated, the episode for experience replay is reduced to 1,000 instead of 3,000 to balance the training time. However, to better compare the performance in the diagram, we set the x-axis of them to be the same with the others. This should give us a more intuitive perspective, though it is incorrect. Based on the curves, we can have some preliminary conclusions:

- In the DQN family: (1) The original DQN has poor performance and slow learning efficiency; (2) Using both Dueling Network and Double architecture can make the results better. (3)The use of experience replay works well and the training is more efficient. However, it should also be noted that this method uses more data per episode than other methods, which also makes the training time longer.
- In the AC family: (1) A2C performs much better than AC thanks to the lower variance given the benchmark. (2) REINFORCE method shows its efficiency in the beginning, while experience replay method shows its potential in the long run. Note that AC with REINFORCE keeps on decreasing at the end of training, and the experience replay keeps stable positive gradient of the curve. (3) Experience replay is much time-consuming, 1,000 episodes for training is still cost five times as the others. Even given such amount of training time, the experience replay method still could not beat REINFORCE methods.
- Compare between the two families: (1) All the methods are better than the DQN baseline. (2) A2C performs close to and slightly better than Double Dueling DQN. (3) AC performs close to Dueling DQN. (4) Double Dueling DQB with experience replay beats all the others with huge advantage. However, this method cost approximately 13 times as the second best method A2C with REINFORCE. Also, in the end of the training the gradient of the curve tend to be zero or even negative, indicating it has been stuck in a local minimum.

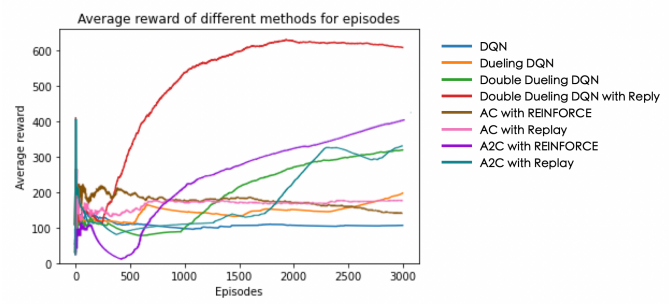

Fig. 2: Average reward comparison of the algorithms and methods.

Considering the time consumed and the increasing tendency, we believe A2C with REINFORCE is very promising. What's more, DQN methods have been applied to play the Atari games in many papers, while AC methods are rarely used, so we are curious to see how well can the AC methods play the game. Therefore, we extend its training episode to 20,000 while keep the other hyper parameters fixed. Total training time is nearly 6 hours running on Colab with GPU accelerated. The result is shown in Fig. 3, we can say the improvement is steadily and promising as we expected. The final average reward reached 1250.3, and the agent was able to play the game until third scenario of the game. Even at the end of the training, the average curve still keep a tendency of increasing, showing that it is far from convergence. Besides, it should be noticed that the maximum step of the agent is limited by an upper bound, which is the third scenario of the game where exists a giant obstacle in the middle of the road and the agent is required to avoid it. From the curve we can see that the agent haven't figured out how to avoid it in the given training time.

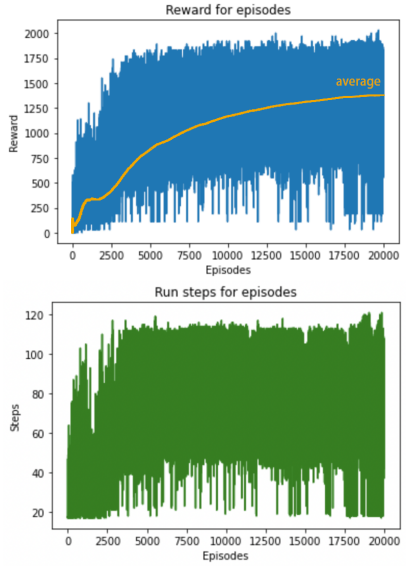

Fig. 3: Reward and runtime of A2C model with the REINFORCE policy trained for 20,000 episodes.

V. Conclusions and Future Works

In conclusion, we have developed and compared different reinforcement learning algorithms in the Atari environment "Riverraid", including DQN, Double DQN, Dueling Network, Actor-Critic, Advantage Actor-Critic, and update their parameters using TD learning, REINFORCE, and experience replay methods.

We fix the network structure and all the hyper parameters to be the invariants, and try to make minimum change between the algorithms to compare their performance ideally. With the preliminary training result of all the algorithms, we draw main conclusions: (1) AC family and DQN family have almost the same overall performance considering the training time. (2) A2C with REINFORCE consumes the least time to train, while still yields good performance.

Therefore, we trained the agent using A2C with REINFORCE for 20,000 episodes and got a final average reward of 1250.3, of which the agent able to play until the third scenario of the game. The average reward and runtime beats the results of all the other algorithms as expected, and the training curve still keep a tendency of increasing without convergence.

We find that there are some challenges for the agent to play the game: (1) There are five enemies that the agent should eliminate or avoid on its way. The agent need to learn the representations of these enemies in the extracted feature, especially the enemies are different from one another. (2) The background of the game is constantly changing as the agent keep moving forward. Since the agent will lose its live if it touches the boundary of the road, the agent should also learn the representations of the road about where it can walk and where it should avoid. However, since the roads are constantly changing, the knowledge the agent learn from previous roads might not apply to next ones, also the knowledge from the next ones would confuse its determination about the previous ones. It is not only difficult and confusing for machine, but also for human. It might be possible that the agent would learn better on other games with more invariant state.

There are also some drawbacks of our experiment: (1) We failed to balance the training time. As discussed, the training time vary largely between the methods and the difference is up to 13 times. Therefore, the comparison is not very strict and convincing. (2) It should be noticed that we introduce multi-step TD target method to update Q learning in II section, but we don't have time to implement it and compare its effect. (3) We didn't process the data very scientifically. We only show the average of the results without the variance, thus make them less intuitive and informative. It reminds us that we should always save the data during training in case for further processing.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Kristopher De Asis, J Hernandez-Garcia, G Holland, and Richard Sutton. Multi-step reinforcement learning: A unifying algorithm. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[4] Jesse Read. Lecture vi - reinforcement learning iii, 2023.

[5] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

APPENDIX

A. Implementations of the Algorithms

1) Deep Q-learning Agent: With reference to the lecture of our course[4], we define the Deep Q-learning agent based on the algorithm of Vanilla version.

Algorithm 1 Deep Q-learning

Initialize parameters w of q_w
for $episode = 1$ to M **do**
  Observe an initial state s_0
  for $t = 1$ to T **do**
    Take an action a_t with ϵ-greedy policy
    Move to a new state s_{t+1} and receive a reward r_t
    Set the target $y = r_t + \gamma \max_a q_w(s_{t+1}, a)$
    Set the estimate $\hat{y} = q_w(s_t, a_t)$
    Get gradient $g \leftarrow \nabla_w (y - \hat{y})^2$
    Update parameters $w \leftarrow w - \alpha g$
  end for
end for

2) Double Q-learning Agent:

Algorithm 2 Double Deep Q-learning

Initialize parameters w of q_w
Copy w to target parameter w' of $q_{w'}$
for $episode = 1$ to M **do**
  Observe an initial state s_0
  for $t = 1$ to T **do**
    Take an action a_t with ϵ-greedy policy
    Move to a new state s_{t+1} and receive a reward r_t
    Set the target $y = r_t + \gamma \max_a q_{w'}(s_{t+1}, a)$
    Set the estimate $\hat{y} = q_w(s_t, a_t)$
    Get gradient $g \leftarrow \nabla_w (y - \hat{y})^2$
    Update parameters $w \leftarrow w - \alpha g$
    Replace target parameter w' by w every N steps
  end for
end for

3) Dueling Network Agent: Dueling Network agent could use the same algorithm as DQN or Double DQN. The difference is that he introduces a state value function when predicting the value of Q-function. In our neural network structure, it is reflected in the addition of an one dimensional output.

4) Actor-Critic Agent: Actor-Critic uses two networks, one as actor to generate the action, and one as critic to judge the action. The goal of the critic is to predict the reward of the environment given state and action, and the actor need to earn as much as soccer from the critic. In another word, the critic is trained by the environment and the actor is trained by the critic.

Algorithm 3 Actor-Critic

Initialize parameters θ of Π_θ, w of V_w
for $episode = 1$ to M **do**
  Observe an initial state s_0
  for $t = 1$ to T **do**
    Calculate the probability distributions of the actions $P(a_t|s_t) = \Pi_\theta(s_t)$
    Sample an action a_t from the distributions $P(a_t|s_t)$
    Calculate the value of current state $V_w(s_t)$
    Move to a new state s_{t+1} and receive a reward r_t
    Calculate the value of new state $V_w(s_{t+1})$
    Set the target $y = r_t + \gamma V_w(s_{t+1})$
    Set the estimate $\hat{y} = V_w(s_t)$
    Set error in estimate $error = y - \hat{y}$
    Get gradient for actor $g_{actor} \leftarrow \nabla_\theta log P(a_t|s_t) V_w(s_t)$
    Update actor parameters $\theta \leftarrow \theta - \alpha g_{actor}$
    Get gradient for critic $g_{critic} \leftarrow \nabla_w (error)^2$
    Update critic parameters $w \leftarrow w - \alpha g_{critic}$
  end for
end for

Note that normally Actor-Critic would use REINFORCE or experience replay methods to update the parameters.

5) Advantage Actor-Critic Agent: A2C add a benchmark inside the computing of policy gradient. The algorithm is the same with AC, except for using the value of the state / action $g_{actor} \leftarrow \nabla_\theta log P(a_t|s_t) V_w(s_t)$, A2C use the difference between the future state and current state $g_{actor} \leftarrow \nabla_\theta log P(a_t|s_t) error$.

6) REINFORCE for Policy Learning: REINFORCE uses the accumulative loss of each episode to train the the control policy of the agent.

Algorithm 4 REINFORCE

Initialize parameters θ of Π_θ
for $episode = 1$ to M **do**
  Observe an initial state s_0
  Initialize a $reward\_list$ and a $action\_list$
  for $t = 1$ to T **do**
    Calculate the probability distributions of the actions $P(a_t|s_t) = \Pi_\theta(s_t)$
    Sample an action a_t from the distributions $P(a_t|s_t)$
    Move to a new state s_{t+1} and receive a reward r_t
    Add r_t into $reward\_list$, add a_t into $action\_list$
  end for
  Construct a list of $accumulative\_reward\_list$
  with $u_t = \sum_{i=t}^{T} r_i$
  Calculate the policy gradient
  $g \leftarrow \sum_{t=1}^{T} \nabla_\theta log P(a_t|s_t) u_t$
  Update parameters $\theta \leftarrow \theta - \alpha g$
end for

7) Experience Replay: Experience Replay collects the transition that the agent played, and sample them randomly to train the agent. For each step of the game, the agent would train for 'BATCH SIZE' replays.

Algorithm 5 Experience Replay

Initialize replay buffer using $Deque$ structure of size $MEMORY\_SIZE$
for $episode = 1$ to M **do**
  Observe an initial state s_0
  for $t = 1$ to T **do**
    Take an action a_t by the model
    Move to a new state s_{t+1} and receive a reward r_t, and a Bool d_t whether the game has done
    record this transition $(s_t, a_t, r_t, s_(t+1), d_t)$ to the buffer
    if Replays in buffer $>= BATCH\_SIZE$ **then**
      Random sample a batch with $BATCH\_SIZE$ of replays
      Computing the loss on the batch by the model prediction
      Updating the parameters $w \leftarrow w - \alpha \nabla_w$ according to the gradient of the loss
    end if
  end for
end for

B. Graphs of the Training Process

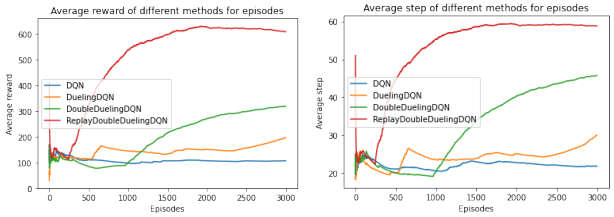

Fig. 4: Average reward and runtime of DQN, DQN with Dueling Network, Double DQN with Dueling Network and Double DQN with Dueling Network and Experience Replay

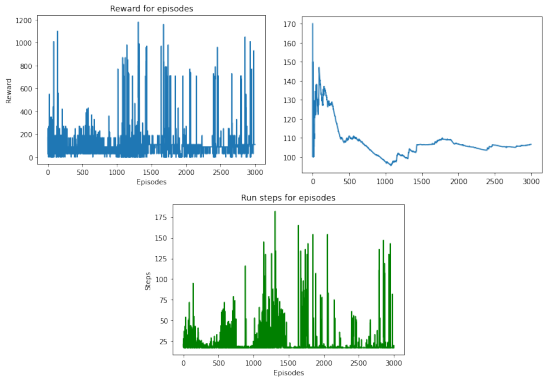

Fig. 5: Reward and runtime of DQN

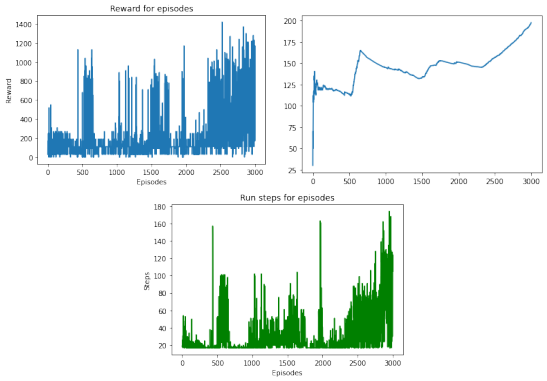

Fig. 6: Reward and runtime of DQN with Dueling Network

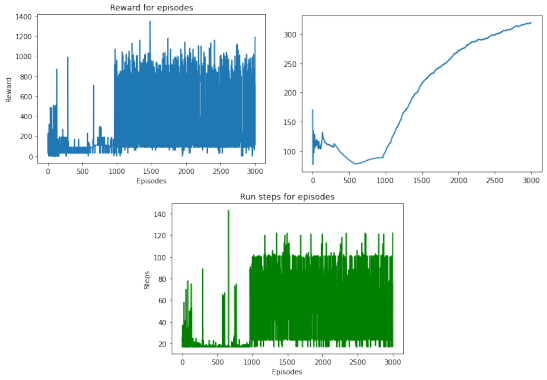

Fig. 7: Reward and runtime of Double DQN with Dueling Network

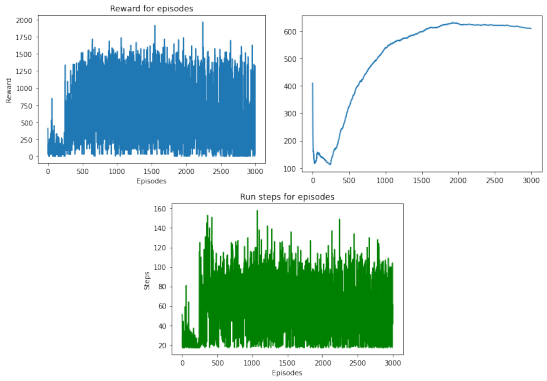

Fig. 8: Reward and runtime of Double DQN with Dueling Network and Experience Replay

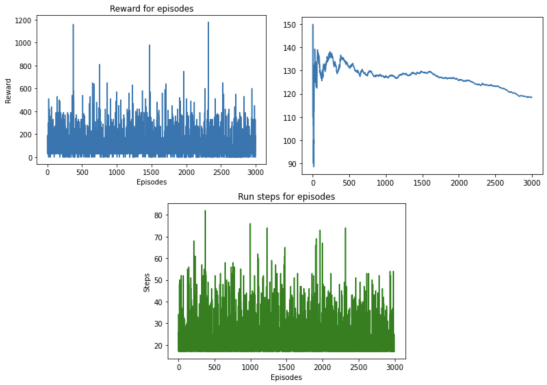

Fig. 9: Reward and runtime of Actor-Critic with REINFORCE

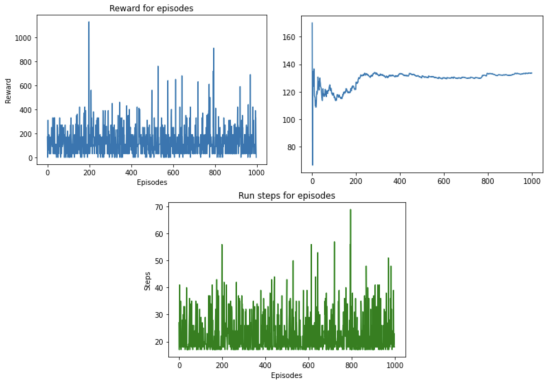

Fig. 10: Reward and runtime of Actor-Critic with Experience Replay

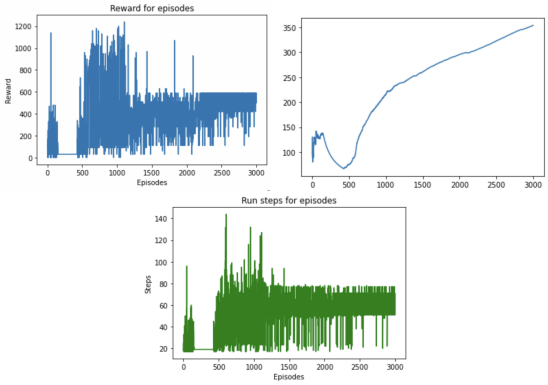

Fig. 11: Reward and runtime of Advantage Actor-Critic with REINFORCE

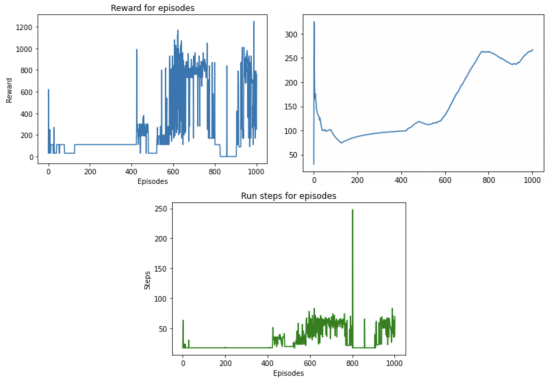

Fig. 12: Reward and runtime of Advantage Actor-Critic with Experience Replay