Double A3C on Playing Atari Games

Lingjie Kong[1]
Stanford University
Stanford Center for Professional Development
ljkong@stanford.edu

Ruixuan Ren[1]
Stanford University
Stanford Center for Professional Development
ruixuan@stanford.edu

Abstract

Reinforcement Learning (RL) is an area regarding how agents act to an unknown environment for maximizing its rewards. Unlike Markov Decision Process (MDP) in which agent has full knowledge of its state, rewards, and transitional probability, RL agent utilizes exploration and exploitation to cover model uncertainty. Because the model usually has a large input feature space, a neural network (NN) is often used to summarize the correlation between input feature and output state action value. Our goal is to improve existing algorithms, specifically on state-of-the-art Asynchronous Actor-Critic (A3C) algorithm. We implement DQN and vanilla A3C to play OpenAI Gym Atari 2600 games to obtain benchmark performance. Then we will propose our implementation on double A3C, an improved version of the vanilla A3C algorithm. We will compare its performance, data efficiency and computation efficiency with the other methods.

1. Introduction

Behaviorist psychology regarding taking the best actions to optimize agent's reward at a specific state inspired the development of reinforcement learning. Up to now, reinforcement learning has been studied in many disciplines such as control theory, information theory, statistics, and so on.

Markov Decision Process (MDP) was used to solve classical decision-making problem where agent has full knowledge of the environment including state, reward, and transitional probability. Due to the limitation in knowledge of the environment, Q learning was developed to let agent explore to find potential optimal solution as well as exploit to optimize the current good solution.

Due to large input state space, it is impossible to use a look-up table like in MDP. Neural network (NN) is used instead. Since a single-hidden-layer neural network is a universal function approximator, it can capture the non-linear relationship between input and output. The network will be trained using the gradient of its loss function, carried out by forward and backward propagations in the NN. The fully trained model will be used to infer based on the current state input, what will be the optimal action to take in order to maximize its rewards.

Reinforcement Learning bring new challenges on how to build and train an efficient neural network. Specifically, RL agent must learn from sparse and noisy data collected through its interactions with the environment. These sparse and noisy data might cause instability during training. Moreover, reward can be delayed. Therefore, it requires efficient method to reward early actions that bring good results later in the training. The environment is often assumed to be static in reinforcement learning. However, as the agent interacts with the real environment, it might change the environment. Therefore, good algorithms are desired to capture the changes in environment dynamics as well.

In this article, the performances of deep Q-network (DQN) and asynchronous advantage actor-critic (A3C) will be compared, using three Atari games: Pong, Breakout and Ice Hockey. A variant of the A3C – double A3C will be presented. Double A3C utilizes the strengths from double DQN and A3C, and we hope to see better results from it compared to the benchmarks.

Figure 1(left to right) Pong, Breakout, Ice Hockey

2. Related Work

High-dimensional visual input is challenging because of its large scale of data. Therefore, many successful RL model in the past is based on carefully hand-selected features. However, it is impossible to handpick features for every environment, and a more generic framework is desired.

The breakthrough in computer vision leads to new ideas to extract feature representations from environment more efficiently [1]. Neural network structures such as convolutional neural networks (CNN), multilayer perception and Boltzmann machine graphic model are often used, which can take large size input features with a relatively small amount of trainable variables.

In addition to the challenge of input feature representation, other challenges are also presented in reinforcement learning. Unlike supervised learning which assumes identical and independent distributed (IID) dataset,

[1]All authors contributed equally to this work.

reinforcement learning must learn from noisy, delayed and highly correlated rewards.

Q-Learning [2] is often used to train reinforcement learning model to reach decent level of performance in simple environment. In Q-Learning, one needs update Q value Q(s, a) for state action pair, where each Q(s, a) is the expected utility of taking specific action a in state s, following the optimal policy onwards. However, it is impossible to explicitly store Q value for large input state space. One common solution is to use function approximation, where we extract features θ from (s, a) and define a function $f(\theta)$ to approximate Q(s, a). Then, instead of optimizing the estimation of Q values, the model is trained to optimize the parameters in $f(\theta)$.

Deep Reinforcement Learning [3] uses a neural network, specifically Deep Q-Network (DQN), as approximate function $f(\theta)$. It has been shown that the agents trained by DQN can reach better-than-human performances in playing many Atari 2600 games. Further studies of Double DQN [4] and Dueling DQN [5] improve both the convergence speed and performance compared to vanilla DQN. With accessibility to GPU, DQN can be trained in relatively fast speed.

Recently, asynchronous method has shown the potential to outperform previous algorithms like DQN [6]. In particular, Asynchronous Advantage Actor-Critic (A3C), can be trained two times faster than DQN with only multi-core CPU, and achieves higher performances in most of the Atari 2600 games.

3. Approach

Convolutional neural network includes convolutional layers, activation functions and fully connected layers. One can also add max pooling and normalization layers to help improve speed of convergence as well as performance. The basic structure is so called AlexNet [1] as shown below.

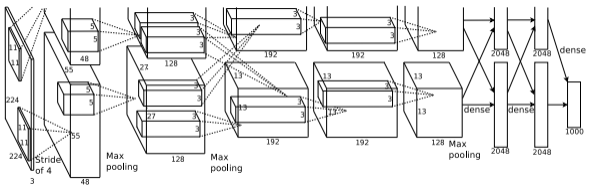


Figure 2 Convolutional Neural Networks

In the context of discrete action space, DQN will pass fully connected layers output through couple more dense layers to generate output with the same dimension of action space. Action with the highest Q value will be picked as the optimal action in that state.

DQN is often trained with experience replay which helps break the correlation between sampled data and improves data efficiency. Specifically, agent will store certain amount of sampled data in buffer and pick them randomly in training. DQN will be trained using gradients, calculated from the minimization of the loss between current Q value and the updated Q value. The latter is estimated by taking the optimal action under the current Q value model. DQN algorithm [3] is as below.

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 3 DQN

However, vanilla DQN has the problem of maximization bias from using the same network for predicting Q value, and then extracting the state value by taking the maximum on Q. This problem can be resolved by utilizing two independent networks, known as double Q-learning algorithm. In Double Q-learning, two Q value functions are trained independently, with one to determine the greedy policy and the other to determine its value [7].

Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s', a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s', a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s', a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s', b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

Figure 4 Double Q Learning

Instead of training a second network, double DQN copies over older weights of the target network to the second network, which is used for action evaluation when calculating gradients for target network's weights. It has been shown that Double DQN finds better policies and obtains better results on the Atari 2600 domain compared to Vanilla DQN [4].

Another breakthrough on DQN is to use Dueling Network Architecture [5]. Instead of a single network for Q value, dueling DQN models state value and state action advantage function separately in the fully-connected layers. Dueling DQN updates estimations on action values and action benefits simultaneously, which gives better action-state values approximation. Assumes weights α and β are the parameters of fully-connected layers for state value and action advantage. Then we can express the action-state value as:

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

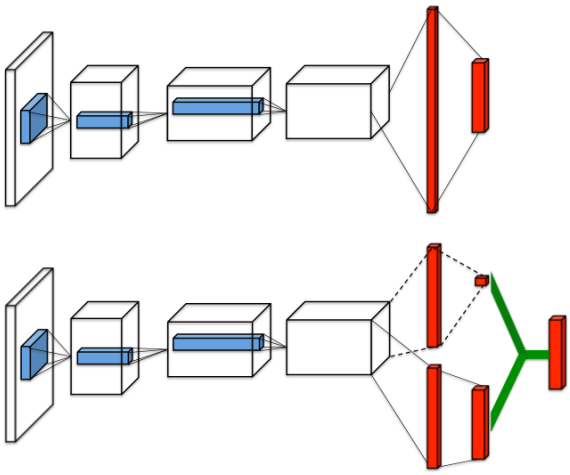


Figure 5 Single stream Q-network (top) and dueling Q-network (bottom)

Asynchronous advantage actor-critic (A3C) algorithm demonstrates better performance than any aforementioned algorithms. A3C utilizes multi-threaded asynchronous variant of advantage actor-critic algorithm, in which the actor is to improve the current policy and the critic evaluates the current policy. The algorithm of A3C is as below [6].

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 6 A3C

Our approach evolves from Double Q learning [7] and state-of-the-art A3C algorithm [6]. The key technique in Double Q learning is to train two Q value functions independently. We believe the such technique can also be applied to A3C algorithm to help improve convergence speed. Specifically, we add a second set of parameters and randomly pick from one set to update both θ for policy and θ_v for value in the classical A3C algorithm (Figure 6). We hope two independent values will break the correlation in sampling and give better speed of convergence. We call this method double A3C as shown in **Error! Reference source not found.**(b). We also vary the number of shared trainable parameters between the two value networks, to see whether that will affect the algorithm's performance. Based on double A3C, we designed another network with less shared convolutional and fully connected layers as shown in **Error! Reference source not found.**(c) called less shared double A3C. In Figure 7(d), one more network with almost no shared parameters is called no shared double A3C.

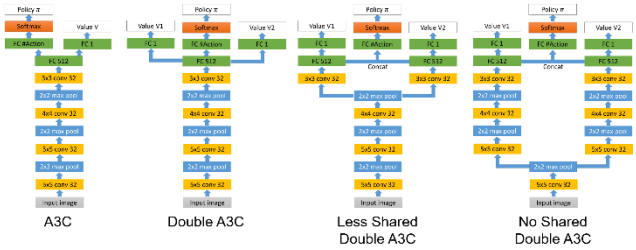


Figure 7 Network architecture of (a) vanilla A3C, (b) double A3C, (c) less shared double A3C (d) no shared double A3C

In the A3C network in Figure 7(a), the input is 4 consecutive 84x84x3 RGB image frames. Therefore, the total input size is 84x84x12. Four convolutional layers and three max pooling layers were used to extract the input image information. The first two convolutional layers each uses 32 filters of 5x5. The third conv layer uses 64 filters of 4x4 and the last layer uses 64 filters of 3x3. All conv layers use stride of 1 and all max-pooling layers use 2x2 with no stride. After the final conv layer, a fully connected layer will reshape and generate output with dimension of 512 (FC512). The output from this fully connected layer (FC512) will be used to generate a value with dimension of 1. This output value will estimate the input state value. Meanwhile, a different fully connected layer will be applied to generate output with the dimension equal to the number of actions. This will be past into a Softmax layer to convert output to policy π which is a probability representation between 0 and 1.

In A3C, value parameter θ_v and policy parameter θ share common parameters from the first convolutional layer up to the first fully connected layer. Only the output layers are different for value and policy networks. Using the same features to generate value and policy can stabilize the model performance during training. It can also increase training speed because less trainable variables are presented.

Unlike A3C which only has one single V to estimate the input state's value, double A3C will have two different value estimations V_1 and V_2. Meanwhile, it will have one single policy π. The training update for double A3C is similar to classical A3C. However, out of the two value estimations, only one will be sampled randomly for calculating gradient. If V_2 is sampled for update at a certain state, the return R will be initialized by V_1 as below:

$$R = \begin{cases} 0 & for\ terminal\ s_t \\ V_1(s_t, \theta'_{v_1}) & for\ non-terminal\ s_t \end{cases}$$

The following updates for θ_{v_2} and θ will be as below:

$$d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi(a_i|s_i;\theta')\left(R - V_2(s_t, \theta'_{v_2})\right)$$

$$d\theta_{v_2} \leftarrow d\theta_{v_2} + \partial\left(R - V_2(s_t, \theta'_{v_2})\right)^2 / \partial\theta'_{v_2}$$

Similarly, if V_1 is selected, we can use V_2 for predicting R and V_1 for calculating the loss.

For double A3C in Figure 7(b), the first set of value parameters θ_{v_1}, the second set of value parameters θ_{v_2}, and the policy parameters θ share some variables as well. Similar to vanilla A3C, double A3C shares parameters from the first convolutional layer up to the first fully connected layer. After that, different fully connected layers are used to generate V_1. V_2, and π individually. By using two different value functions, we believe it can help remove the correlation from consecutive state samples. The uncorrelated estimation from one value function will help another value function in terms of faster convergence and eventually better performance.

Under the condition that double A3C still shares most parameters from the first convolutional layers to the first fully connected layers, we try to build network with even less shared parameters, resulting in more independent estimations for V_1 and V_2. By doing so, we hope to break the correlation between sampled sequence of states even further and achieve better convergence. Therefore, we introduce less shared double A3C in Figure 7(c) and no shared double A3C in Figure 7(d). Less shared double A3C has 3 shared convolutional and max pooling layers while no shared double A3C only has 1 shared convolutional and max pooling layers. To generate the policy π, fully connected layers from V_1 and V_2 will be concatenated and then Softmax will be applied to the combined features.

4. Experiment Results

Three Atari games: Breakout, Ice Hockey and Pong were used to evaluate performances of DQN, A3C, double A3C, less shared double A3C and no shared double A3C. We analyze both the speed of convergence with respect to time as well as training efficiency measured by average score after each epoch.

Our A3C model is built based on existing A3C model using tensorpack [8]. Our code is published under our Github repository [9].

Different from our DQN which only maintains single agent for training, our A3C model keeps 3 agents in parallel to collect samples for update. In addition, the DQN model uses experience replay and target network, which are assumed not necessary in A3C [6].

All models are trained on Microsoft Azure cloud computing, with a 6-core E5-2690v3 Intel CPU and a K80 NVIDIA GPU.

The results for pong, breakout, and ice hockey are shown in Figure 8 and Figure 9. We evaluated average score vs. epochs to show training efficiency. In each epoch, the model is trained on 6000 batches, each containing 128 gradients calculated by the agents. Specifically, by using the same amount of data in each epoch, we want to compare the data efficiency of different algorithms. We also evaluated average score vs. time to ssee how long it will take for each algorithm to converge.

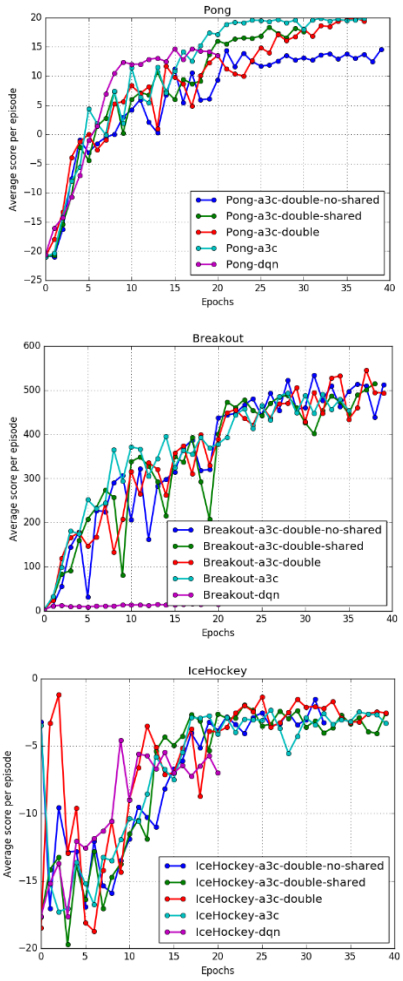
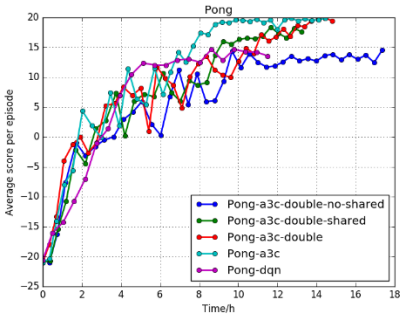


Figure 8 Comparison of data efficiency between all A3C and DQN. X-axis is the total number of training epochs in which each epoch has 6000 update steps. Y-axis is the average score



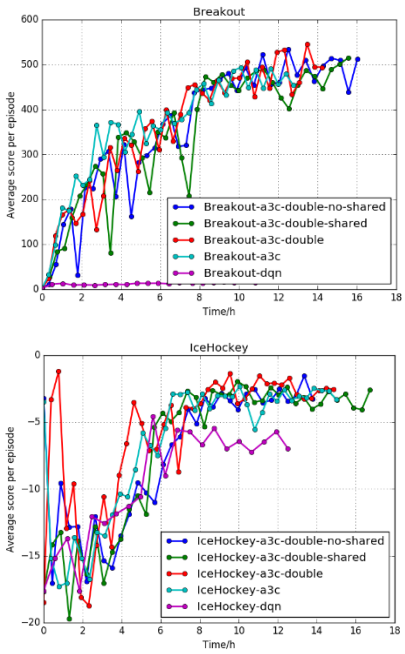


Figure 9 Comparison of training speed between all A3C and DQN. X-axis is the total number of training time. Y-axis is the average score

Figure 8 shows the average score at the end of each epoch. Pong is a relative simple game. Therefore, all A3C-based algorithms as well as the DQN model reach the same level of score at a similar pace. The no shared A3C shows relatively poor performance compared to all other models. But we attribute this mainly to the fact that we didn't have time nor the resource to run each case multiple times, since no shared A3C shows good performance in the other two more complex games. Breakout is an intermediate level game. All A3C-based agents outperform DQN which failed to learn the game. Ice hockey is a much harder game. Even though all models converge to better scores than they begin with, their scores are still far from normal human player's level. In all three games, the performances of A3C-based variants are all similar to that of vanilla A3C.

Figure 9 shows the speed of convergence. For DQN, it takes about 37 minutes for each epoch update to finish. For no shared double A3C and less shared double A3C, they both spend about 25 minutes in each epoch update. Double A3C and Vanilla A3C are the fastest, taking about 22 minutes to learn each epoch.

5. Discussion

A model's performance at a specific game varies by the game's level of complexity. Because of Pong's relative simplicity, DQN is able to achieve the same level of performance as all A3C-based models. All networks converge around the same score after 20 iterations in 8 hours of training. Again, the relative poor performance of no share A3C is ignored, assuming that it is just one 'unlucky' instance. The graphics of Pong is much simpler, with only one ball travelling between two pads. Its reward is almost immediate compared to the other two games. This simplicity plays down the advantages of A3C, which reduces correlations between states by using parallel workers, and minimizes variance by predicting value and policy separately. The fact that we only have one GPU for this training also hampers A3C's ability to speed up training with parallelism.

When trained on breakout, all members of the A3C family including vanilla A3C, double A3C, less shared double A3C, and no shared double A3C have the same level of performance. Noticeably, the A3C family outperforms DQN, which is expected because of the increased complexity of the game. Interestingly, one can observe that with less shared trainable parameters, the average score of the model at early stage fluctuates much more violently. This can be reasoned by the large difference between the two sets of value parameters during early stages of training. The difference only reduces with further training, witnessed by the smaller magnitudes of the score fluctuations.

Ice hockey is the most complicated game among the three. Its much delayed reward, more variations in graphics and complicated game mechanism all contribute to the enormous difficulty to train agents on this game. All network structures from DQN to the A3C family cannot master this game in a human level at the end of training. In addition, it requires the collaboration between the two players in a team to score a goal. Instead of modeling both players' actions in a simple network, one might consider distinguishing these two players from the graphic inputs, and then model them in separate networks.

Overall, A3C, double A3C, less shared double A3C, and no shared double A3C all have the same level of performance in the three games. Adding a second value function to vanilla A3C does not help to improve the training speed or efficiency. We hoped to improve the performance of A3C by introducing a second value function so that the sequence of experiences can be broken up and their correlations can be reduced. But from the results, we can observe that assigning the experience randomly to updating one of two value functions does not help reduce correlations. One can potentially use a lot more independent value functions to really remove the correlations between experiences. But it is more computationally expensive, and experience replay is arguably a better option. Furthermore, the randomly initialized behavior policy in each parallel worker has already minimized the correlations in a sequence of experiences to a certain degree. The room for improvement in this area may not be that much at all.

One more thing to note is that double A3C is inspired by double DQN. However, double Q is used in DQN only to remove the maximization bias introduced by taking

maximum over the Q values. On the other hand, vanilla A3C does not introduce maximization bias in the first place.

6. Future Work

We plan to explore potential improvements on A3C in other aspects in the future.

Due to the asynchronous nature of the A3C algorithm, when the gradient is calculated by an agent, the central network parameters may have already been updated a few times by other works. This causes a policy lag between gradient and policy, and the learner becomes off-policy. In order to achieve more stable training and faster convergence, we can use algorithms such as V-trace to correct the policy lag [10].

In an effort to further reduce correlations between states, prioritized experience replay can be implemented, which will also focus the training on most important gradients. A3C only uses one central learner which cannot be efficiently expanded to large parallel computing systems to increase training speed dramatically. Using multiple parallel learners along with many parallel actors is potentially another interesting direction [11].

7. Conclusion

Reinforcement learning is a powerful tool to solve complicated Atari game without hand selected features. Deep neural network agents such as DQN and A3C can reach human level performance in simple games after hours of training utilizing cloud resource.

We propose three variants of vanilla A3C: double A3C, less shared double A3C, and no shared double A3C, by introducing a second value function. To our disappointment, the variants' performances are similar to vanilla A3C's, mainly because using two value functions is not sufficient to break up the correlations between experiences. To further explore potential improvements on A3C, v-trace [10] and prioritized experience replay [11] will be implemented on top of it in the future.

8. References

[1] A. Krizhevsky, I. Sutskever, and H. Geoffrey E., "ImageNet Classification with Deep Convolutional Neural Networks," *Adv. Neural Inf. Process. Syst. 25*, pp. 1–9, 2012.

[2] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 279–292, 1992.

[3] D. Zoran *et al.*, "Playing Atari with Deep Reinforcement Learning," *arXiv*, vol. 32, no. Ijcai, pp. 1–9, 2016.

[4] D. S. H. Van Hasselt, A. Guez, "Deep Reinforcement Learning with Double Q-learning," p. in proceedings of AAAI 2016.

[5] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling Network Architectures for Deep Reinforcement Learning," *arXiv*, no. 9, pp. 1–16, 2016.

[6] V. Mnih et al., "Asynchronous Methods for Deep Reinforcement Learning," p. in proceedings of ICML 2016.

[7] H. Van Hasselt, A. C. Group, and C. Wiskunde, "Double Q-learning," *Nips*, pp. 1–9, 2010.

[8] Y. Wu, "A3C-Gym," no. GitHub Repository, p. https://github.com/ppwwyyxx/tensorpack

[9] L. Kong and R. Ren, "Deep Reinforcement Learning on Playing OpenAI Gym Games," no. GitHub Repository, p. https://github.com/lingjiekong/CS234Project.

[10] L. Espeholt et al., "IMPALA:Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures," no. arXiv:1802.01561v2, 2018.

[11] D. Horgan et al., "DISTRIBUTED PRIORITIZED EXPERIENCE REPLAY," no. ICLR, 2018.