Understanding Multiagent Cooperation and Competition with Deep Reinforcement Learning

Simon Zhuang
simonzhuang@berkeley.edu
3032176109

Stefanus Hinardi
hinardi2@berkeley.edu
24850692

Abstract—In this project, we investigate multiagent cooperation and competition strategies using deep reinforcement learning. We did a code reimplementation of the paper "Multiagent Cooperation and Competition with Deep Reinforcement Learning" that gives us a baseline model. The original implementation was done in Lua, using the Torch framework. We are doing the reimplementation in Python, using the Tensorflow framework. We also detail the environment and training hyperparameters used in the original paper and discuss that results that we obtained. We were able to replicated the results that are described in paper. Further analysis of the result shows that a learning agent may learn from another "more expert" agent in a competitive environment. Moreover, we observed that training in multiple agents simultaneously in competitive environment can be seen as a form of curriculum learning[1]. Lastly, we also extend the project by implementing Double Q-Learning [2] and Dueling Networks [3] to see how it would improve training or testing performance.

1 Problem Definition and Motivation

This problem is interesting because multiagent systems appear in most social, economical, and political situations. We believe that investigating the relationship between reward functions and agent behaviour between agents in multiagent systems may bring about important insights that would be helpful in other important areas of society. For example, understanding how reward functions are related to the behaviours of agents might help us come up with methods to better train agents to result in specific outcomes.

In this paper that we are reimplementing, the authors showed how they observed collaborative and competitive behaviours arise by manipulating the reward scheme (the way the agents are incentivised). More specifically, they used the video game *Pong* to demonstrate how the agents' behaviour changes with the changes in their reward schemes.

We are aiming to reimplement the paper in Python, using the Tensorflow machine learning framework. We also intend to extend the project by implementing Double Q-Learning and Dueling Networks to investigate how they can improve the training speed and quality.

2 Related Work

The sub-field of reinforcement learning has been getting a lot of attention for the past 5 years. Recent work has demonstrated that using neural networks for approximating Q-value functions can lead to good results in complex domains by using techniques like experience replay and target networks during updates [4], giving birth to deep reinforcement learning. Deep reinforcement learning has been successfully applied to continuous action control [5], strategic dialogue management [6] and even complex domains such as the game of Go [7]. However, there has not been a lot of work in extending it to multi-agent systems. The work that we are re-implementing, "Multiagent Cooperation and Competition with Deep Reinforcement Learning", is one of the first works that explore this realm.

The original implementation was done in Lua, using the Torch machine learning framework. The original implementation can be found at this link: https://github.com/NeuroCSUT/DeepMind-Atari-Deep-Q-Learner-2Player. To our knowledge, this is the only work that implements the deep Q network in the multiagent setting.

Since this paper uses a Deep Q Network architecture implemented in an earlier paper [4], there are existing Tensorflow implementations of the Deep Q Network available. However, to our knowledge, there are none implementing the experiments in the multiagent setting using the Tensorflow framework. Implementing that will be our contribution.

We have referenced a source that used Tensorflow to train the Deep Q Network in a single agent setting such as (https://github.com/tambetm/simple_dqn). We based the implementation of parts which were similar (the Deep Q Learning model architecture) but wrote code for the training in the multiagent setting and the collection of game statistics referenced in the paper that we were reimplementing.

3 Implementation Details

3.1 Environment

The paper uses the Xitari environment, a fork of the Arcade Learning Environment v0.4 [8]. In order to support Atari game roms that allow for both players of *Pong* to be controlled, the authors used a modified version of Xitari, Xitari2Player (https://github.com/NeuroCSUT/Xitari2Player), and wrote a Lua wrapper (https://github.com/NeuroCSUT/Alewrap2Player/tree/master/alewrap) on top of it.

In order to conduct the same experiments in Python using Tensorflow, we first ported the Xitari2Player environment over into a Python package (https://github.com/choo8/Xitari2Player). This is done by loading the compiled C++ library in the original Xitari2Player implementation using the *ctypes* Python library. We then created a Python API that allowed us to access the custom 2-player functions in the compiled C++ library.

3.2 Rewarding Scheme

In this paper, the main objective of the authors was to investigate how manipulating the reward functions of the agents could give rise to certain interesting behaviours. In the paper, the reward scheme is as defined below.

	Left player scores	Right player scores
Left player reward	ρ	-1
Right player reward	-1	ρ

TABLE 1: Rewarding Scheme as detailed in paper

The variable ρ can take values between -1 and 1. To study the transition of agents' behaviours from cooperative to competitive, the authors used values of ρ = -1 (cooperative), -0.75, -0.50, -0.25, 0, 0.25, 0.50, 0.75, 1 (competitive). By looking at the behaviour of the agents at the different ρ values, we can observe how the agents' behaviours transit from one of cooperation to that of competition.

These rewarding schemes are encoded in the Atari game roms of the game. Hence, for each value of ρ, there will be a rom specially created for it. We obtained the roms from the original authors at (https://github.com/NeuroCSUT/DeepMind-Atari-Deep-Q-Learner-2Player/tree/master/roms).

In our experiments, due to the long training time, we only managed to experiment with ρ values of -1, -0.5, 0, 0.5, 1.

4 Deep Reinforcement Learning

4.1 Background

The aim of reinforcement learning is to come up with a policy that maximizes the long term reward by coming up with actions at different states in dynamic environments [9]. This problem is a challenging one since agents do not have any information about the environment. One of the most popular algorithm that tackles this problem is the Q-Learning Algorithm [10].

$$Q^*(s, a) = max_\pi E[R_t | s_t = s, a_t = a, \pi]$$

In the past, Google Deep Mind team has adapted this algorithm leveraging Convolution Neural Network as Q function approximator [4]. This result shows that Deep Q Network achieves superhuman performance in a range

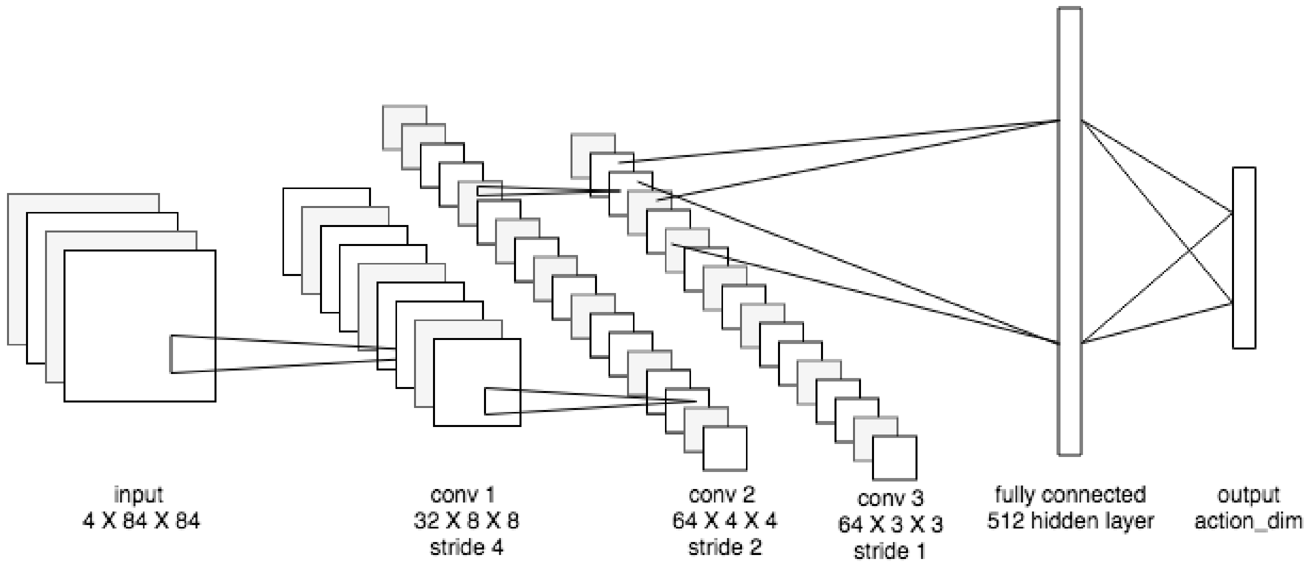

Fig. 1: Model Architecture of Deep Q Network

of Atari video games by using only raw sensory inputs (screen images) and reward signals [4].

When two or more agents are present in the environment, the problem becomes more challenging. The distributed nature of the learning environment offers challenge in finding good learning objective as well as convergence and stability of the algorithm [9]. Since now the rewards are affected by joint actions of the present agents in the environment, each of the agents need to keep track of all of the other agents.

4.2 Preprocessing of Atari Frames

As the inputs to the deep neural net are frames of the gameplay, it might be computationally difficult to train the network if we do not preprocess them before passing them into our network. This is because the number of pixels in the original frames is large (210 x 160 images with a 128-color palette).

First, for every single raw frame, we perform a pixel-wise max pooling operation over all 3 color channels of itself and the previous frame. This is to remove the flickering present in some games due to some objects only appearing in odd frames and some objects only appearing in even frames.

Next, we extract the Y component (luminance) of the pooled frames after converting their pixel values from RGB to YUV.

Finally, we rescale the single-channel frames linearly from 210 x 160 into 84 x 84. These preprocessing steps are all detailed in the original deep Q network paper in [4].

4.3 Experience Replay

As mentioned in original Deep Q Network paper, to solve the problem of correlated data and non-stationary distributions, the neural networks are trained using the experience replay mechanism [4] instead of sequential frames from the game. The experience, $e_t = (s_t, a_t, r_t, s_{t+1})$, of each agent is stored in a dataset $D = e_1, ..., e_N$, pooled over many episodes into a replay memory [11]. The randomization of the experience breaks these correlations and therefore reduces the variance of the update s [4].

4.4 Periodic Update

Iterative update that adjusts the action-values, $Q(s, a)$, towards target values that are only periodically updated, thereby reducing correlations with the target [4].

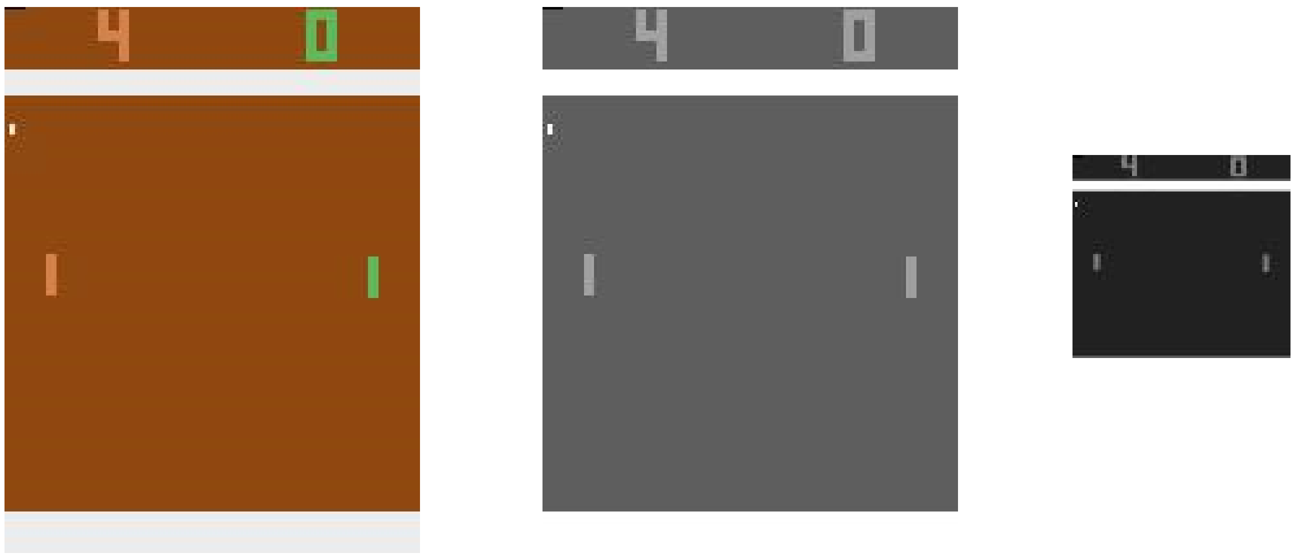

Fig. 2: Preprocessing of the raw Atari frames. On the left is a frame after max pooling. In the middle is the Y channel of the same frame. On the right is the same frame scaled down to 84 x 84.

4.5 Deep Q Network Architecture

The input to the neural network is a 84 $\times$ 84 $\times$ 4 tensor created by stacking 4 preprocessed images together [4]. The first hidden layer has 32 8 $\times$ 8 convolution filters with stride 4 with the input image and applies ReLu units. The second hidden layer has 64 4 $\times$ 4 convolution filters with stride 2, again followed by ReLu units. The third hidden layer has 64 3 $\times$ 3 convolution filters with stride 1, again followed by ReLu units.Lastly, The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully connected linear layer with the size of the number possible actions of a particular game. For both of the agents, we only consider the 4 valid moves that are available for the *Pong* game. Graphical representation of the architecture is shown in Fig. 1. In optimizing the loss function, we use RMSprop [4] as described in the original paper.

5 Experiment

5.1 Training Procedure

We replicated exactly how the paper conducted the experiments. All the hyperparameters used were the ones that were specified in [4].

All experiment were done in 50 epochs, with 250,000 steps in each epoch. The value of 50 epochs was chosen based on the convergence of Q-values for both agents [9].

5.2 Evaluation Metrics

In this paper, the authors measured specific game events in order to quantitatively measure the agents' behaviours in the *Pong* environment. We are able to collect these game statistics in our reimplementation as they are available in the C++ libraries that we ported over to Python.

At the end of each epoch of training, both agents play 10 games against each other and the statistics are measured. The measures used are:

Average paddle-bounces per point

This records the number of times the ball bounces between the agents before a point is scored.

Average wall-bounces per paddle-bounce

This records the number of times the ball bounces off the top and bottom walls when it travels from one agent to the other.

Average serving time per point

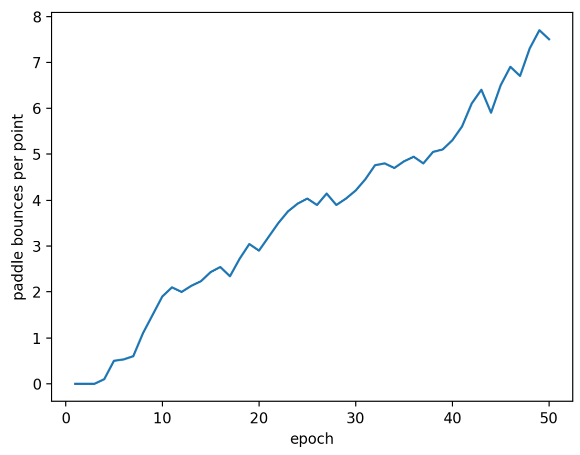

Fig. 3: Competitive Strategy: Paddle bounces per point

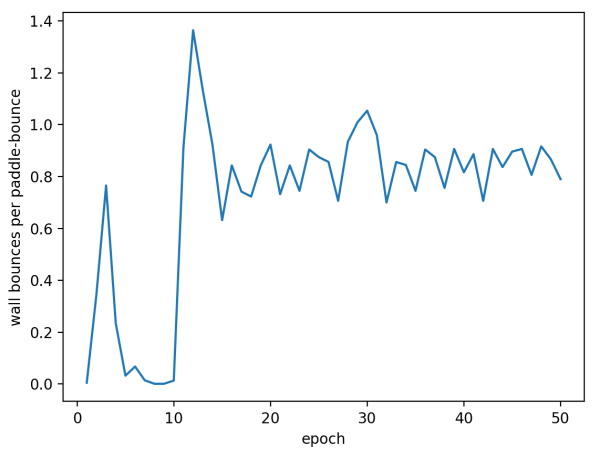

Fig. 4: Competitive Strategy: Wall bounces per point

This records how long it takes for agents to restart the game after a point is scored, in terms of number of frames.

We will explain the quantitative measures recorded for both cooperative and competitive behaviours in the results section below.

6 Results

6.1 Emergence of Competitive Strategy

In the fully competitive rewarding scheme, each agent obtains a reward of 1 when it scores and the losing agent obtains a reward of -1. We obtained very similar results to the ones published in the paper. Looking at the graphs above, we can see that there is a general upwards trend of average number of paddle-bounces per points as the number of epochs increases. This is because as the agents get better at the game, the number of rallies increases as it is not that easy to score against each other. For the first few epochs, the number is close to zero as both agents do not have knowledge of the game and may score points by pure luck, hence the Q values are still quite noisy. Moreover, the average bounces of the wall is around 0.8 per point. This might be because it is a good action to bounce the ball off the wall since it will be harder to predict where the ball is going.

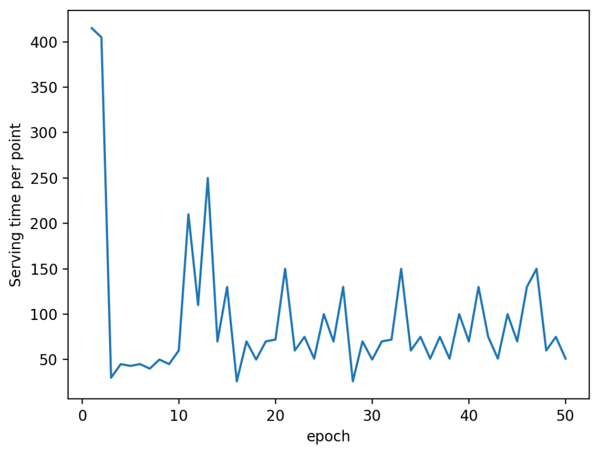

Fig. 5: Competitive Strategy: Wall bounces per point

We can also see that the agents have learned to start the game very quickly after a point is scored. This is seen in the abrupt decrease in average serving time per point early in the training of the agents.

6.2 Emergence of Cooperative Strategy

In the fully cooperative rewarding scheme, both agents obtain a reward of -1 whenever a point is scored, regardless of the agent who scored it. We obtained very similar trends to the ones published in the paper, but with a few differ-

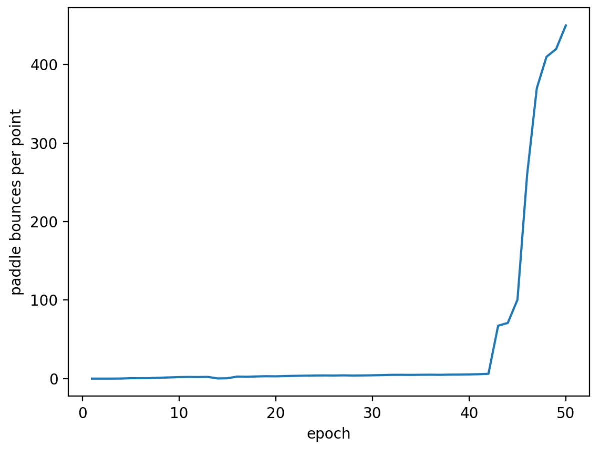

Fig. 6: Cooperative Strategy: paddle bounces per point

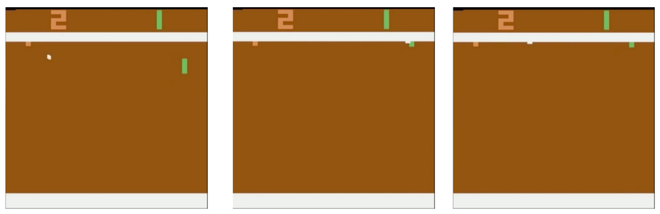
Fig. 7: Sample of cooperative strategy: the two player was able to hit the ball horizontally to keep the ball in the game

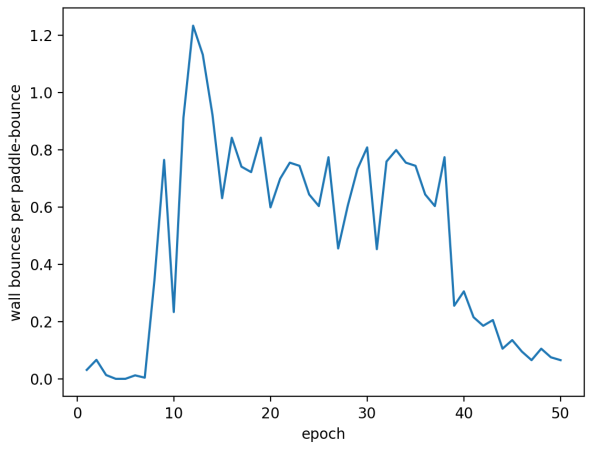

Fig. 8: Cooperative Strategy: Wall bounces per point

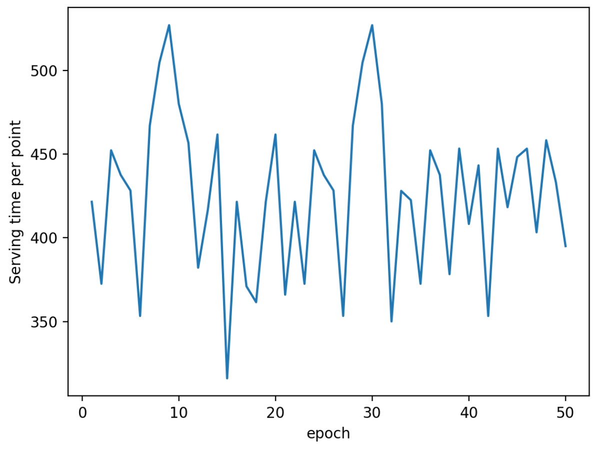

Fig. 9: Cooperative Strategy: Wall bounces per point

ences, such as the highest number of paddle-bounces per point is slightly less than that reported in the paper.

Looking at the graphs above, we can see that as compared to the competitive agents, the cooperative agents managed to have significantly larger number of average paddle-bounces per point as training goes along. This is because they are incentivized to keep the ball in the game, explaining the longer rallies. One strategy that the agents were able to keep the ball by hitting it horizontally shown in figure 7.

We can also see that the average wall-bounces per paddle-bounce is also generally lower than that of the competitive agents' case. This may be due to longer rallies from passing balls directly between agents instead of via wall bounces because the ball direction is easier to predict by the other agent. We can also see that the average serving time per point is generally higher than that of the competitive agents' case. Playing can only result in negative rewards, so the agents may have figured out that holding onto the ball for as long as possible is also a legitimate strategy to maximize rewards.

6.3 Progression from Competition to Collaboration

We observe very similar trends in the statistics that we collected when compared to that of the

ρ	Average paddle-bounces per point	Average wall-bounces per paddle-bounce	Average serving time per point
1.0	6.9 ± 1.2	0.87 ± 0.08	118.79 ± 45.15
0.5	5.543 ± 0.6	0.76 ± 0.07	139.76 ± 34.02
0.0	4.234 ± 1.3	0.884 ± 0.1	367.94 ± 87.79
-0.5	5.154 ± 1.1	0.45 ± 0.07	398.39 ± 45.42
-1.0	443 ± 154.45	0.02 ± 0.01	402.34 ± 234.49

TABLE 2: Behavioural statistics of agents as a function of their incentive to score

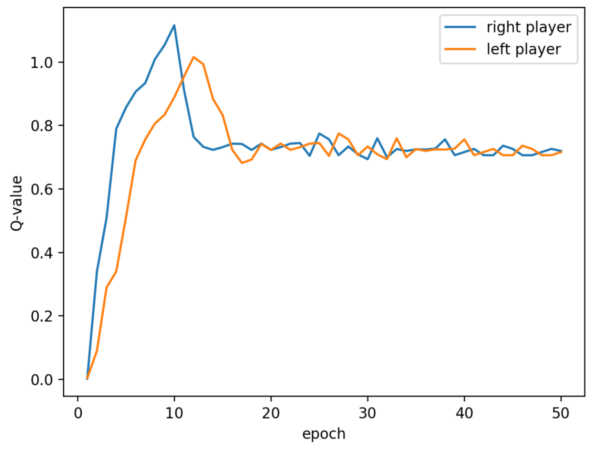

Fig. 10: q-value evolution

original paper, shown in Table 2. For the average number of paddle-bounces per point, we see only a sharp increase when ρ = -1. For the average number of wall-bounces per paddle-bounce, there is a clear decreasing trend, but with a sudden increase in the middle at ρ = 0. For the average serving time per point, there is a clear increasing trend.

7 Discussion

Here we were able replicated the results that were published in [9]. The results from the experiments indicates that Deep Q learning is suitable method for distributed multiagent learning. In particular, we were able to observe different interesting strategies emerge by manipulating the reward function.

Another interesting observation to be pointed out is that the progression of Q-value of the two agents in the competitive environment shown in Figure 10. The right player Q-value seems to gets higher faster and decreases faster than the left player Q-value, as if the left player's Q-value is following the right player Q-value. This can be interpreted as the left player agent learning from the right player as the right player gets better. A possible reason for the right player learning first is that for every episode of the game the right player gets to serve the ball first and due to the reward scheme of the competitive environment, the right player would get more positive rewards than the left player when both agents are still "bad" at the game (in early epochs). Although more rigorous testing and experimentation is required, this is an evidence of an agent learning from a "better" agent through environment observation in a multi-agent setting.

Another interesting observation that we found is that this situation is similar to situation in curriculum learning[1], where agents are presented with increasingly "difficult" learning task. In this case, left agent are learning from a slightly better opponent, the right player. At the same time, the right player also improves as its opponent becomes better and better.

However, this phenomenon does not happen in the fully cooperative setting because the reward systems does not reward any of the agent, thus serving first for every episode do not give any advantage in learning fully cooperative strategies.

Although we have shown that these agents can learn different strategies through DQN, there is a limitation that was shown by the convergence Q-value when both agents get to master the game. Ideally, two very competent agent would reach Nash Equilibrium, in which for every action that they would predict reward of 0.5. However, the plot seems to suggest that both of the agent Q-value was around 0.7. This is perhaps a result of a known limitation of Q learning that overestimate its prediction [2]. In the extension section, we will explain some strategies that we explored to mitigate this issue.

8 Extension

In order to mitigate the issue of over estimating the expected reward of particular actions mentioned in the discussion section, we explored 2 modifications to the vanilla DQN implementation.

8.1 Double Q-Learning

Double Q-Learning is a modification of the Q-learning algorithm that attempts to mitigate the overestimation phenomenon. The idea behind double-Q learning is to decouple the selection from the evaluation [2].

$$Y_{DoubleQ_t} = R_{t+1} + (S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t); \theta_t^{'})$$

Note that now, we still use the online set of weights. θ_t., to estimate the value of the greedy policy. However, we use the second set of weights, $\theta_t^{'}$, to fairly estimate the policy[2].

When integrating Double Q-Learning into the DQN network, we replace the second set of weights with the target network. Although the target network is not fully decoupled, target network is a natural candidate for the second value function, without having to introduce additional networks [2].

8.2 Dueling Networks

Dueling Networks provide two separate estimators, as opposed to the one in the Deep Q Network. They are able to separately estimate the state value function, $V^{\pi}(s)$, and the state-dependent action advantage function, $A^{\pi}(s,a)$. The functions are defined as:

$$V^{\pi}(s) = E_{a \sim \pi(s)}[Q^{\pi}(s,a)]$$
$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Intuitively, the value function V measures how "good" it is to be in a particular state and the advantage function A gives us a relative measure of the importance of each action for a given state. Hence, the dueling network is able to estimate the value of each state using the state value function estimator $V^{\pi}(s)$, without having to learn the effect of each action for each state. In [3], it is shown that for tasks with large action spaces, the $V^{\pi}(s)$ function that is estimated is shared across many similar actions at state s, leading to faster convergence.

8.3 Result

After applying the two improvements into the network, we ran the competitive setting again from scratch. The results show that it did not overshoot its Q-value estimation. We also discovered that the number of epochs needed to get to stable Q-value is lower compared to that of the vanilla DQN implementation.

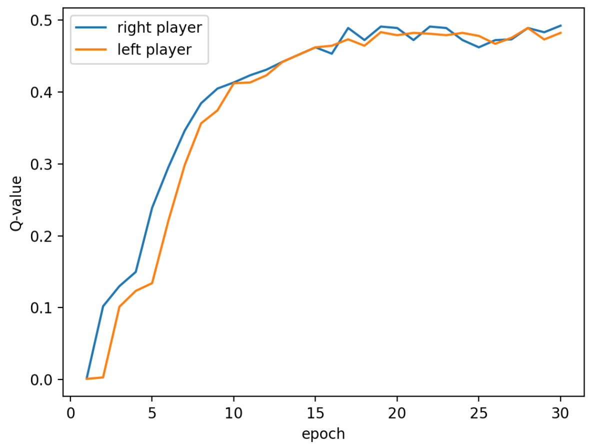

Fig. 11: q-value evolution

9 Limitation

This work is limited to using autonomous Deep Q-Networks to control independent agents. This could be extended by experimenting with other ways to train the Deep Q-Network. For instance, one could experiment with partial parameter sharing to simulate some forms of communication between all of the agents.

Moreover, this work could benefit more form extracting more varied metrics to evaluate the behaviour of the agents in different environments/reward settings. For example, average speed of the ball would allow us to analyze the "aggressiveness" of the agents depending on the reward configurations.

Lastly, to closely analyze and understand how each agent assesses the environment as well as other present agents, visualizations such as perturbation saliency map [12] and T-SNE embedding [13] would be extremely useful.

10 Tools

We decided to use Tensorflow to implement the network. Our reason of using Tensorflow is because it is popular framework that people use to do experiment. By creating Tensorflow version of this implementation, we hope that people would extend our work in future easily. In terms of computation power, we utilized a workstation equipped with NVDIA GTX 1080Ti as well as an AWS EC2 p3.8xlarge instance.

11 Lessons Learned

There are a few aspects where we learned during this process:

- We found that Deep Q-Learning is very brittle. We tried to optimize the hyperparameters, however, we did not get a very good result. We decided to stick with the parameter that is described in the original Deep Q Network paper.
- Deep Q-Learning takes a lot of time when training multiple agents from scratch. Out training time is significantly longer compared to the reported time when training a single agent against a competent agent. This might be because the competent agent gives better signal for the rewards, compared to our case where most of the rewards early on during training are due to random actions are taken.
- Understanding Deep Neural Network using visualization is tricky. When we tried to visualize the saliency map for the trained agents using gradient based saliency method[14], we found that the saliency map was not human-interpretable as shown in figure 12. After more researching, we should have gone with perturbation based saliency maps that reported was able to visualize ”attention” in Atari games[12].

12 Team Contribution

All of the work are divided equally, including writing code, training and writing the report.

In writing code, the work division is :

- Simon Zhuang : Together with Joshua Choo(a member from the joint project) wrote Ale wrapper to work with python, Tensorflow game statistic collection as well as saliency map
- Stefanus Hinardi: DQN code (+ experience replay, dueling and double q network) and multi-agent Training loop

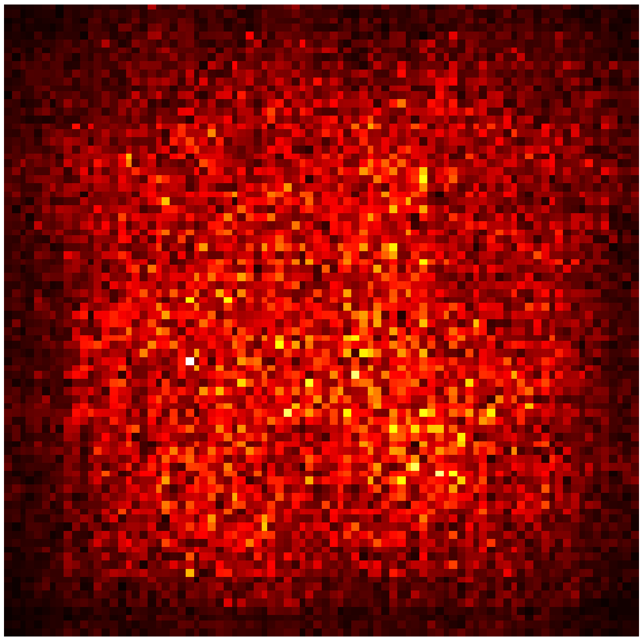

Fig. 12: Non human-interpretable gradient based saliency map

13 Conclusion

In conclusion, we managed to reproduce very similar results to what the original paper had published, using the Tensorflow framework. Also, in our project, we have shown that by additionally implementing Double Q-Learning and Dueling Networks on top of the vanilla Deep Q Network would give us shorter training times and faster convergence.

Our code reimplementation of this paper can be found at: https://github.com/simonzhuang/Tensorflow-DeepMind-Atari-Deep-Q-Learner-2Player.

A potential future extension of this project would be instead of training both agents from scratch, train them separately with a skilled AI agent first. We believe that this will greatly shorten the training time and speed up convergence as the first few epochs the agents are as good as random agents and actions may lead to good Q values purely by chance. Furthermore, experimentation with parameter sharing between agents could also be explored as a way of communication between the agents.

14 Acknowledgement

We would like to thank Joshua Choo as our third member of this project. Since this is a joint project with another class project, he is not enrolled in this class.

References

[1] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[2] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[3] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[5] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[6] Heriberto Cuayáhuitl, Simon Keizer, and Oliver Lemon. Strategic dialogue management via deep reinforcement learning. *CoRR*, abs/1511.08099, 2015.

[7] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[8] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013.

[9] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *CoRR*, abs/1511.08779, 2015.

[10] Christopher J. C. H. Watkins and Peter Dayan. Technical note: q-learning. *Mach. Learn.*, 8(3-4):279–292, May 1992.

[11] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[12] Sam Greydanus, Anurag Koul, Jonathan Dodge, and Alan Fern. Visualizing and understanding atari agents. *CoRR*, abs/1711.00138, 2017.

[13] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[14] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013.