TRAINING VISION-BASED AGENT WITH THE ACTOR CRITIC MODEL IN AN ONLINE ENVIRONMENT

Liu, Ting-Wei, B03901170@ntu.edu.tw

Wen, Ming-Hao B03901179@ntu.edu.tw

Department of Electrical Engineering, National Taiwan University

ABSTRACT

In this report, we present the result of training a vision-based agent for Slither.io, an online massively multiplayer browser game that is partially supported by Universe [OpenAi], using Reinforcement Learning (RL) algorithms. The framework we used is based on the Actor-Critic models [1][2][3][4], combining with convolutional neural networks (CNN). During training, we apply several techniques to encourage exploration and keeping our agent at a high entropy state, successfully avoiding the dilemma of having a highly-peaked policy function ($\pi(a|s)$) towards a few actions, a known problem with on-policy models. The agent we trained requires only raw frames from the screen and game states from the AI side, without using opponents' information. Therefore, **the technique applied is general and suitable for training computer agents in other environments which uses raw frames directly.** Our agent is capable of playing against other human players online and survive in this massively multiplayer game, and is proficient at performing tricky moves upon the encounter of enemies, including intensive sharp turns, high speed twist, and circulations.

Index Terms— Slither.io, Universe, Reinforcement Learning, Actor-Critic, Convolutional Neural Network

1. INTRODUCTION

Deep reinforcement learning has dominated over controllable environments, where agents are trained against manipulable and predictable computer opponents, for example the first-person shooter game Doom [5], and Atari Games [6]. In these controllable environments, the difficulty of the environment is often tunable, offering a simple training approach: start training from simple environment and gradually try harder ones. Also, the pattern of the game is more predictable and repeatable, so the agents learn more easily. However, to train an agent directly in an online environment where massive human players are involved remains an open challenge. Since the behaviors of human players are more unpredictable, the environment and difficulty the agent faces every time is inconsistent and can vary through a wide range. Direct application of the Actor-Critic model is nontrivial due to the sparse and long-term reward, and mainly because the unpredictable and inconsistent environment.

The environment that we choose to train our agent in is Slither.io, an online massively multiplayer browser game, where players control an avatar resembling a snake. These snakes can consume multicolored pellets from other players or ones that naturally spawn on the map to grow in size. A snake only dies when its head collides with another snake's body, when a snake dies it transforms into a humorous number of pellets which other players can consume. The goal of the game is to grow the largest snake in the server. Previous work on this game is based on hand-tuned state machines and privileged information (player's precise location), where several circular range are preset around the snake's head to signify alert level, and the snake will turn its head in an increasingly dramatic fashion as other player steps closer to its head. The actions performed by this state machine are predictable and monotonic, also this snake is incapable to kill. State machine tuning in complex situations requires manually designed rules and can be very time consuming, also, it does not operate like human who only rely on visual input.

In this report, we design a vision-based agent that trains in Slither.io by playing directly against other human players online with the Reinforcement Learning approach, using a framework based on the Actor-Critic models [1][2][3][4] combining with convolutional neural network. This model takes the current game frame as input, and predicts the next most desirable action and the value of the current state. We apply four techniques to overcome the lack of exploration problem of the Actor-Critic model, where the agent refuses to explore new actions as one action is being repeatedly chosen due to the highly-peaked policy function ($\pi(a|s)$). As a result, our agent is capable of playing and surviving against human players, with special tactics to maneuver upon engagement of enemies, which no one taught it explicitly and was learned by the agent itself. Our training process doesn't require any opponent's information or any other internal information of the game, instead our training relies only on raw frames and a reward signal provided by the game, just like a human player playing with pure vision and knowing the game score.

Figure 1: Screen Processing.

2. SLITHER.IO AS A TRAINING ENVIRONMENT

We run Slither.io with the aid of Universe [OpenAI], in which the environment is set up with two parts, the client and the remote environment server. The client part of the environment runs locally, and this is where the agent train, receives observations, and choose actions. Most importantly the client is responsible to communicate with the server through a defined protocol. The server part sets up a container using a Docker, and our game Slither.io runs within it through a Chrome browser. The server and the client communicates through two important ports, the VNC port and the rewarder port. The VNC port is responsible for connecting the client to the VNC viewer, where the VNC viewer delivers pixels of observation from the game back to the client. Also, the client can deliver keyboard and mouse inputs through the VNC port. The rewarder port runs under the Universe-specific bi-directional JSON protocol, it allows the agent to submit control commands to the environment, and to receive rewards and other structural information such as latencies and performance timings. Currently, the Universe has full environment support over flash games and Atari games, except internet games, like Slither.io. As internet games requires a different type of client to communicate to the server. The environment wrapper of Slither.io is only partially supported, with the client part unfinished, this requires us to look into some documents and modified the environment wrapper by ourselves.

The choice of our training environment: Slither.io, is an online massively multiplayer browser game, this makes our training process extremely difficult. Not only did the agent have to face human players, but the environment was also extremely inconsistent. When a snake spawns in Slither.io, it will be randomly placed in a circular map, where players and pellets are denser towards the center of the map, and sparse out towards the outer rim. We observe that sometimes our agent would spawn near the outer rim, where it would just wander around without encountering any enemies and pellets, which is extremely inefficient. In other extreme instances, it would spawn at the center of the map, where it is surrounded by massive number of human players, causing it to die in a short time. This extremely random and inconsistent environment is a major drawback to our training, as we lack

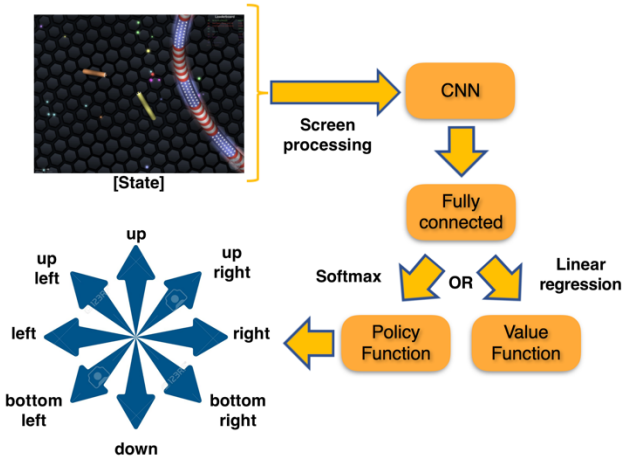

Figure 2: The Network Structure.

control over the environment, and there are too many possibilities for the agent to explore. However, we did overcome this problem and gain some success through the tuning of our training process.

3. THE ACTOR-CRITIC MODEL

As all Reinforcement Learning algorithms seek, the goal is to train an agent that maximizes expected future reward and minimizes penalties, this can be partially achieved by learning one of the following two crucial functions: the value function $V(s)$ that estimates the expected value of the current state, and the policy function $\pi(a|s)$ that yields the probability distribution on the candidate actions a given the current state s. The Actor-Critic models [1][2][3][4] jointly estimates these two functions, $V(s)$ and $\pi(a|s)$, together. The Critic is responsible for learning the value function $V(s)$ while the Actor learns to act by learning the policy function $\pi(a|s)$. The Critic updates its value function by using the reward r as ground truth, and updates toward the calculated squared temporal-difference error (TD error: $r + \gamma V(s_{t+1}) - V(s_t)$). The Actor updates the policy function by encouraging actions that lead to high reward and discourage actions that lead to low reward, which couples the value function by using an advantage (TD error) guided loss. Given a state, we "encourage" the log probability of the chosen action by an amount guided by the TD error. We decide to choose a minimization algorithms, therefore we should add a minus sign to this "encouragement" to the policy function. since $\max(\pi(a|s) = \min(-\pi(a|s))$. This leads to the two loss functions as follows:

$$loss_\pi = -(r + \gamma V(s_{t+1}) - V(s_t)) \log \pi(a|s)$$

$$loss_V = (r + \gamma V(s_{t+1}) - V(s_t))^2$$

Where γ is the gamma discount factor, used to discount the next states value, and the minus sign before $loss_\pi$ is because

we want to minimize the negative of this expression, therefore maximizing the original positive expression. Gradients descents are applied on these two loss functions, then the value function and policy function are updated accordingly. It can be seen that this model is prone to converge, as the two functions $V(s)$ and $\pi(a|s)$ reinforces each other: a correct $\pi(a|s)$ leads to high rewards, which updates $V(s)$ towards the desire direction, and a correct $V(s)$ emphasizes the correct action for $\pi(a|s)$ to strengthen. On-policy models like Actor-Critic converges faster since it updates in every time step, however, it is destined to converge at a bad local minimum. Critic updates on rewards that is gained by the Actor, while the Actor updates on the Critic's value, the same reinforcement behavior will always lead to highly-peaked policy function towards a (or a few) action(s) [5], because it is always easy for both the Actor and Critic to repeat the same action and optimize on that single action. Consider the Actor choosing an action that accidently leads to a high reward, then it will continue to choose that action as the Critic encourages it to do so, causing the Actor and Critic to over-optimize on a few actions and refuses to explore the rest of the environment. When we first train our agent directly using the Actor-Critic model, we experience great failure since a single action is always peaked with a chosen probability of 100%, and our agent would only move in a straight line without reaction. To overcome this problem, we have applied several techniques to encourage the agent to explore and urge it to choose diverse actions, which will be mentioned in a later section.

4. SCREEN PROCESSING

A vision based agent observes the environment state directly through game frames, and this can be done by extracting the screen through a convolutional neural network. However, some preprocessing and adjustments must be performed on the screen image before we feed it to the agent. Since Slither.io is an online browser game, we open the game instance through a Chrome browser, which we will first crop the game screen from the browser and ignore the rest of the screen. Resulting in a 300×500 RGB image representing the game screen, this image is then rescaled to 50% of its original size, giving us a 150×250 screen in RGB. Therefore each frame is now represented by an array with the shape of (150, 250, 3). Next, we process this resized screen in to a grayscale image, the screen is now represented by an array with the shape of (150, 250, 1), with each pixel represented by an integer ranging from 0 to 255. These resize and grayscale transform steps significantly ease up computational power, which we find to be crucial in our case due to the limitation of our hardware resource (Macbook Pro, 2015b, without GPU). As a final step, we normalize the screen by dividing each pixel by 255. Which we also find to be crucial as the convolutional neural network and the Adam [7] optimizer that were used operates at its best when the input feature varies through a smaller range of 0.0 to 1.0, rather than 0 to 255. Moreover, we find this normalization step significantly increases the convergence speed of our model. This screen processing flow is summarized in Figure 1. At each time step, we run the above procedure once before feeding the screen into the convolutional network. Note that resizing and turning the screen into grayscale is only necessary when the device running the training process is not capable of running with original resolution and RGB (as in our case), we believe that training with original size and full color would greatly aid the performance of the agent.

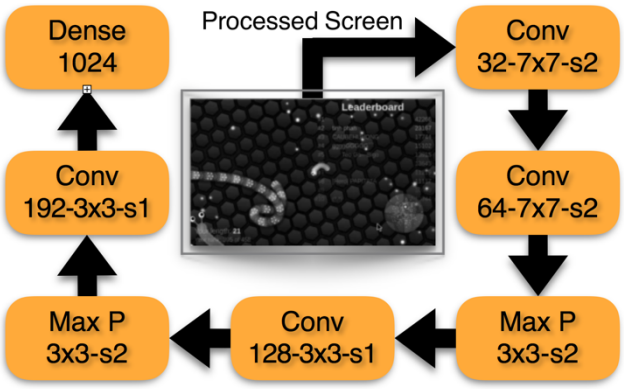

Figure 3: The Network Parameters.

5. NETWORK ARCHITECTURE

We use convolutional neural networks to extract information from the game frames, the Actor and the Critic each has its own but identical network. The network we used is consist of a 6 layer CNN followed by a fully connected dense layer. Despite the structural equivalence, the Actor network and Critic network differs with each other by having a different output function. The Actor's CNN is attached with a fully connected layer that outputs the policy function $\pi(a|s)$ by regular softmax. The policy network will give 8 actions, namely *up, up right, right, bottom right, bottom, bottom left, left*, and *up left*, each resembling the cursor pointing in the corresponding direction. After the policy network outputs the probability distribution over all actions, the agent than randomly chooses one action with respect to the probabilities. These cursor directions are preset as coordinates on the screen, which will lead the snakes towards the corresponding directions. On the other hand, the Critic's CNN is joined with a fully connected layer that outputs the value function $V(s)$ by linear regression. The network structure is shown in Figure 2. The network parameters are shown in Figure 3, where Conv 32-7x7-s2 represents a convolutional layer with 32 filters, each with a 7x7 kernel, stride 2, and Max P 3x3-s2 represents a max pooling layer with a pool size of 3x3, stride 2. Each convolutional layer is followed by a ReLU, except the output layer. These network parameters are based on the network used in [5] and has shown good performance on vision-based agents.

6. EXPLORATION TUNNING

Our model initially suffers from the dilemma of having a highly-peaked policy function ($\pi(a|s)$) that favors towards a few actions, and refuses to choose and explore other options. It would continue to choose the same single action, causing it to move in a straight line. It would not react to any snakes as it hasn't had the chance to explore other actions yet. To have a clearer understanding of the agent's behavior, imagine at the start of the training, the agent chooses an action a that accidently gain some reward r by crossing a random pellet. This will make the Critic think that the current state has a value of r, then the Critic encourages the Actor to continue to choose the action a. The Critic and Actor will continue this process and over-optimize on this action without having a chance of seeing the whole picture. We discovered this problem by observing the output of the policy network, and find that a single action always has a probability distribution of 100%, causing the agent to choose that action continuously. To avoid this lack of exploration issue, we utilize 4 techniques that greatly encourages the agent to explore:

1) We add an entropy term to the loss function of the policy network to encourage diversity, which was suggested in [5] and [8], and we later find to be crucial. The entropy term we used is defined as follow:

$$\mathrm{H}\big(\pi(\cdot\,|s)\big) = -Sum[\pi(\cdot\,|s)\log(\pi(\cdot\,|s) + \rho)]$$

Where ρ is a small entropy constant that we find to be necessary to add, as it avoids taking log on zero probabilities. This entropy term is maximal when the probability distribution over all action candidates are uniform, and tends to approach zero when further they are from uniform. Therefore by maximizing the entropy term, we are encouraging uniform action probability, hence increasing exploration. As a result, we simply modify the original loss function by adding the entropy term to it. Intuitively, the entropy term encourages an action additionally when the probability distribution is uniform, and it wears off when the probability distribution tends to diverge from uniform. Then again, we maximize this final encouragement by taking a minus sign to the whole expression and running it under a minimization algorithm, the final loss functions are listed as follow:

$$loss_\pi = -[(r + \gamma V(s_{t+1}) - V(s_t))\log\pi(a|s) + \beta\mathrm{H}\big(\pi(\cdot\,|s)\big)]$$
$$loss_V = (r + \gamma V(s_{t+1}) - V(s_t))^2$$

Where β is the entropy discount factor, used to adjust the effect of the entropy term.

2) In addition to the entropy term, we further design our agent's action policy. After the policy network outputs the probability distribution by softmax, actions are not chosen by selecting the action with the greatest probability, but instead chosen randomly with respect to these probabilities. For example, if the policy network outputs the following probability distribution over the 8 actions: (0.1, 0.2, 0, 0.3, 0.2, 0, 0.1, 0.1), then the first action has a 10% chance of being chosen, the third action has zero chance of being chosen, and so on. In this manner, we greatly increase the chance for the agent to explore on different actions. Moreover, we further deploy a constant ε-greedy exploration policy, where the agent will select random actions uniformly at probability ε, and select action using its policy network with a probability of $1-\varepsilon$. This ε-greedy exploration policy is crucial as it greatly increases exploration, and forces the agent to learn new actions. This ε-greedy exploration policy is only activated during training, and is deactivated otherwise, so the agent chooses action by its own will during demonstration.

3) We further use the technique of reward shaping, as it has been shown to be an effective technique when applying Reinforcement Learning in a complicated environment [9]. In Slither.io, rewards are sparse and delayed, hence in addition to basic rewards earned by consuming pellets on the map, we apply immediate rewards listed as follow:

Condition	**Reward, *r***
$r > 0$	$r \times= 2$
repeated action >3	$r -=$penalty, penalty+=0.25/repeated action
action != action'	$r +=1$
die	$r = -10$

The above reward shaping is applied on every training time step. If the agent receives positive rewards, we further double its reward. We penalize agents who keeps choosing the same action, this is done by keeping track of an action history. At each time step, we check the latest 3 actions, if the past 3 actions are duplicated actions, then the agent receives an incrementing penalty for every additional repeated action. This penalty starts at 0.25 and increases by 0.25 for every further repeated action, the penalty resets back to 0.25 once the repetition ends and the agent starts choosing a different action. Furthermore, the agent gets an additional bonus just by choosing a different action every time step. The application of reward shaping enhances the effect of the exploration and penalizes repeated actions, which helps the agent learn that it is applicable to choose different actions. Lastly, the agent receives a big penalty (-10) at the terminal state (when it dies).

4) The final change we've made to encourage exploration is to multiple the policy network's output by an exploration factor (this factor can range from 0.1 to 0.9)

just before applying softmax, a method proposed in [5]. This exploration factor scales down the effect of the highly-peaked policy function as it makes the output of the softmax more uniform, which we find to aid early stage training significantly.

The above four measures are designed in a sense to keep the search alive, they prevent the agent from converging to a single action, especially when there are several other options that will lead to the same reinforcement result. Intuitively, early convergence discourages exploration, if an action is over reinforced too early, we may be missing something that made another action better. So it is best for our agent to stay at high entropy, hence seeking action uniform, to avoid local minimum. By applying these four techniques, we were successful in avoiding the highly-peaked policy function ($\pi(a|s)$) dilemma, and the agent uses all 8 actions rapidly and resourcefully to maneuver through the course of the game. We conclude that these tuning and adjustments are the major reason of our success. As a result, we were able to produce playable agents even in a difficult training environment.

7. TRAINING PIPELINE

Our model and training procedure is implemented with TensorFlow [10] and in Python. When the environment is ready, the initial state (processed screen) is feed to the Actor, which will choose an action according to its action policy. The agent then steps into the next state with this action, and the environment responds by yielding the next state and the reward of the current state. The Critic then updates its parameters by performing gradient descent with respect to its loss function using the reward, and providing the Actor with the calculated TD error. The Actor learns the value of its previously chosen action by using the TD error to guide its action behavior. Then the Actor chooses its next action, and the whole cycle repeats. This training pipeline is summarized in Figure 4. We call a complete cycle of this training process a time step, in each time step the model will update its parameters once. As mentioned in [5] and [11], care should be taken for frame skips, as small frame skips will introduce a strong correlation in the training set, and huge frame skips will result in inefficient training. In our case, the environment continues to run during a training cycle (when the model is updating), causing frames to be lost naturally, as a result this reduces the game experience correlation, and the model trained is less biased. For our training settings, we use Adam [7] as our optimizer with a learning rate $\alpha = 0.001$, gamma discount factor $\gamma = 0.9$, entropy discount factor $\beta = 0.7$, entropy constant $\rho = 1\times10^{-10}$, ε-greedy exploration factor $\varepsilon = 0.2$, and finally the exploration factor $= 0.2$. The training procedure runs on a Macbook Pro, 2015b without the use of GPU. It takes a few hours to obtain applicable results. Our final model is trained from scratch for 118 episodes which is equivalent to 1118262 training time steps (updates), the training runtime is approximately 4 hours.

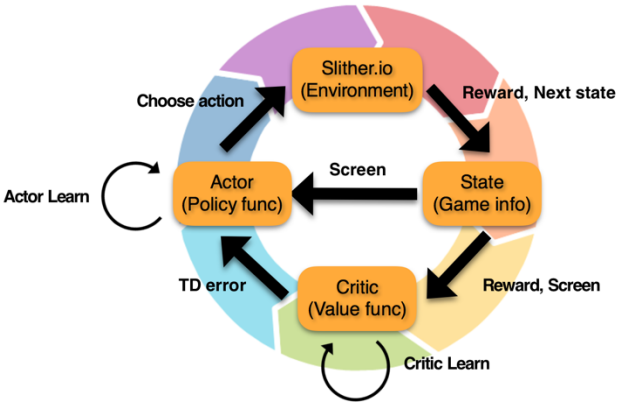

Figure 4: The Training Pipeline.

7. RESULT

The training curve for our final model is shown in Figure 5. Where the blue curve represents the reward the agent earns in each episode, and the red curve represent the number of time steps the agent took in that episode. The number of time step can be a symbolic representation on the survivability of the agent, showing how long it survives in each episode. Remember that Slither.io is an online massively multiplayer browser game, which means that it is an extremely unstable environment, as in each episode the agent has to face different human players whose behaviors may vary very different. This explains the rough and spiking training curve, but despite the extreme of the environment, it is still obvious to observe that the performance of the agent is gradually improving throughout the course. Figure 6 shows the average reward gained by the agent per time step, this is calculated by dividing the reward earned in each episode by the time step the agent took in each episode. Our agent shows improvement on its behavior as the curve tends to grow more stable after 60 episodes. As the behavior and gameplay of our agent, currently our agent is proficient of responding to most of the encountered enemies, where it can maneuver in a special fashion to avoid collision with other snakes. Our agent is capable of performing sharp turns, high speed twist, and circulations. It has learned how to survive by avoiding to run its head into other snakes, and to consume pellets on the map to grow in size. However, it has not yet learned how to kill enemies with its body aggressively, which is a difficult task to do because it is very unlikely for a not-yet-well-trained agent to kill a normal human player. However, with its very unexpected twist it is sometimes able to trick its opponent and kills them passively. Occasionally when our agent runs into an enormous large snake it is also confused and will eventually die by not reacting properly. These flaws can be easily overcome with the use of a stronger Reinforcement Learning algorithm which we did not use in this report due to game environment and computational limitations (This part will be discussed in the next section), or with an enormous

amount of additional training time. For those who are interested, we have assembled a highlight video of our agent's game play, which is available on YouTube: https://youtu.be/8iRD1w73fDo.

8. CONCLUSION

In this report, we show the training procedure and result of our vison-based agent, which combines the classic Actor-Critic model with the convolutional neural network. This agent is trained for the game Slither.io, a rather hard environment for Reinforcement Learning due to its inconsistency and lack of support by Universe [OpenAI], which causes us some problems with internet connection issues. We were initially experiencing difficulties as our agent continues to choose the same action, but with the 4 techniques mentioned above, we were able to maintain the entropy of our agent, and keep the exploration momentum during the training phase. The greatest constraint and difficulty in our research is mainly due to the harsh environment we've choose and the limited hardware resources. If we had better computational resources, we would have select the much more advanced, state-of-the-art approach, the Asynchronous Advantage Actor-Critic (A3C) model [8], which runs multiple independent game instances in parallel using the Actor-Critic model. All these game instances will together update the main model, theses asynchronous updates will greatly reduce the correlation of game experience and dramatically increase convergence speed and performance. Unfortunately, our hardware cannot afford to run multiple parallel game instances, which denied us the option of using the A3C model. However, we did try to mimic the effect of A3C by using multi-threading and Tensorflow [10] training coordinator, where we open up multiple game instances and run these instances in turn. Each time only running one game instance with the agent for a short amount of time steps, while giving up control on other game instances (By not receiving actions and stepping into the environment). After each game instance is visited once we collect the game experience and update them in batch. This way we don't really need multiple workers (duplicated models) working at the same time, but only one worker working, while obtaining uncorrelated game experiences. In other words, we trade training time with computational power. This indeed ease up the computational power, but unfortunately the TCP connection of Slither.io will break when we switch between these different game instances, and this attempt results in a failure too. This suggest that we select a different game environment next time (one without internet connection) if we want to delve into the powerful and almighty A3C model. However, with our tuning during the training phase, using the Actor-Critic model alone still shows promising results. **The research on this topic establishes a firm foundation for further studies**, as we have learned how to set up Reinforcement Learning environments using Universe [OpenAI], write our own environment wrapper,

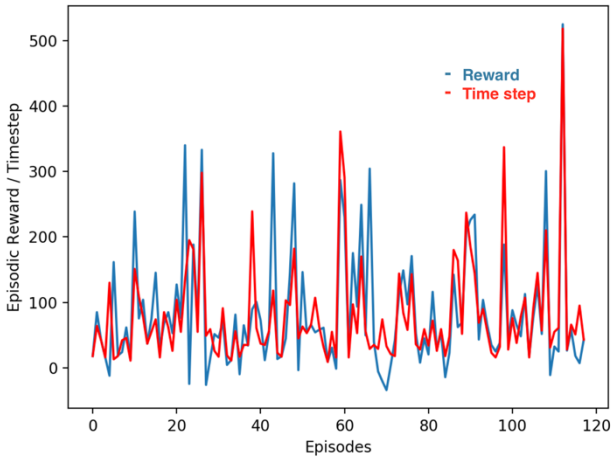

Figure 5: The Training Curve.

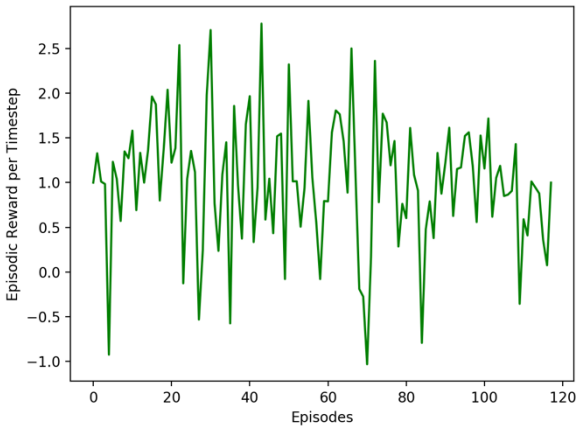

Figure 6: The Reward per Time Step Curve.

design and tune the Actor-Critic models using TensorFlow [10], and furthermore a thorough understanding on the topic of agent exploration dilemma. Currently, our agent is still a visual responsive bot that lacks global tactic and aggressiveness. Ideally, the agent should learn how to attack other players aggressively and intentionally, not just surviving, however we shall leave this part to future works. As a final remark, the architecture of our method is general, it can be expanded and apply on other environments that are even more complex, and is suitable in the sense of training more intelligent computer agents that are capable of interacting with humans in an intellectual way.

12. REFERENCES

[1] Barto, Andrew G, Sutton, Richard S, and Anderson, Charles W, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[2] Sutton, Richard Stuart, "Temporal credit assignment in reinforcement learning," 1984.

[3] Konda, Vijay R and Tsitsiklis, John N, "Actor-critic algorithms," In *NIPS*, volume 13, pp. 1008–1014, 1999.

[4] Grondman, Ivo, Busoniu, Lucian, Lopes, Gabriel AD, and Babuska, Robert, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *IEEE Transactions on Systems, Man, and Cybernetics*, Part C (Applications and Reviews), 42(6):1291–1307, 2012.

[5] Yuxin Wu, Yuandong Tian, "Training agent for first-person shooter game with Actor-Critic curriculum learning,", *ICLR*, 2017.

[6] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al, "Human- level control through deep reinforcement learning," *Nature*, 518(7540):529–533, 2015.

[7] Kingma, Diederik and Ba, Jimmy, "Adam: A method for stochastic optimization," *arXiv* preprint, arXiv:1412.6980, 2014.

[8] Mnih, Volodymyr, Badia, Adria Puigdomenech, Mirza, Mehdi, Graves, Alex, Lillicrap, Timothy P, Harley, Tim, Silver, David, and Kavukcuoglu, Koray, "Asynchronous methods for deep reinforcement learning," *arXiv* preprint, arXiv:1602.01783, 2016.

[9] Ng, Andrew Y, Harada, Daishi, and Russell, Stuart, "Policy invariance under reward transformations: Theory and application to reward shaping," In *ICML*, volume 99, pp. 278–287, 1999.

[10] Abadi, Mart´ın, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Cor- rado, Gregory S., Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Goodfellow, Ian J., Harp, Andrew, Irving, Geoffrey, Isard, Michael, Jia, Yangqing, Jo´zefowicz, Rafal, Kaiser, Lukasz, Kudlur, Manjunath, Levenberg, Josh, Mane´, Dan, Monga, Rajat, Moore, Sherry, Murray, Derek Gordon, Olah, Chris, Schuster, Mike, Shlens, Jonathon, Steiner, Benoit, Sutskever, Ilya, Talwar, Kunal, Tucker, Paul A., Vanhoucke, Vincent, Vasudevan, Vijay, Vie´gas, Fernanda B., Vinyals, Oriol, Warden, Pete, Wattenberg, Martin, Wicke, Martin, Yu, Yuan, and Zheng, Xiao- qiang, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, abs/1603.04467, 2016. URL http://arxiv.org/abs/1603.04467.

[11] Kempka, Michał, Wydmuch, Marek, Runc, Grzegorz, Toczek, Jakub, and Jas´kowski, Wojciech, "Vizdoom: A doom-based ai research platform for visual reinforcement learning," *arXiv* preprint arXiv:1605.02097, 2016.