DeepFEH: Deep Q-Network Based AI For Turn-Based Strategy Games

Yuan Liu *
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
yuanl5@andrew.cmu.edu

Wenqi Mou *
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
wenqim@andrew.cmu.edu

Shijie Wu *
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
shijiew@andrew.cmu.edu

1 Introduction

In the development of a game AI, most AIs are hand implemented. However, a good AI is both hard to design and easy to be utilized by malicious players. This is caused by the inconsistency between the goal of the game and the goal indicated by the specified rules. Reinforcement learning alleviates this kind of difficulty by transfer the design of the rules to the craft of rewards. Though rewards for a specific environment is generally hard to be engineered, many turn-based strategy games are easy to have a reward which indicates the goal of the game.

Reinforcement learning is an approach that makes the AI outperform human without too much human-specified rules or even without human prior knowledge [Silver et al., 2017]. Q-learning [Watkins and Dayan, 1992] is a simple way for agents to learn how to act optimally in a decision process. Recent research shows that Deep Q-Network (DQN) [Mnih et al., 2015] is successful in reinforcement learning, especially in the learning process of classical environments like Atari [Mnih et al., 2013] and continuous control [Lillicrap et al., 2015]. In this project, we present an application of DQN in turn-based strategy games, which is usually ignored by researchers. We implement a simulated environment of a mobile turn-based strategy game as the environment, which in general conforms to the API of OpenAI Gym [Brockman et al., 2016]. We use a linear network without nonlinear activation as baseline. Then we strengthen the network with three hidden layers with ReLU activation [Nair and Hinton, 2010]. We expect to further explore the generalization ability of DQN by implementing more complex game environments. In order to evaluate our network, we propose three evaluation metrics based on the difficulty of the game.

Fire Emblem Heroes Mobile is a tactical role-playing game where you control a team of 4 heroes to fight against enemy team on an 8×6 map turn by turn. In player's turn, the player optionally moves his hero to a location, then, if possible, makes an attack to his enemy, and enemy will do the same in enemy's turn. The player wins when all enemies are defeated and loses when all heroes die. Since this game involves complex strategy based on various states, it poses as a challenging turn-based strategy problem for the agent.

*These authors contributed equally to this work

2 Approach

We implement a game simulator which simulates the Fire Emblem Heroes Mobile and conforms to the interface of gym. Our simulator outputs both an observation of current state of a game episode as well as a list of legal actions. We denote an observation of our environment as s, a legal action in s as a_s. In our DQN, we use both s and a_s as the input. And the corresponding value $Q(s, s_a)$ as output. In short, our DQN do not use the typical structure that outputs a list of Q-values since the valid actions given a specific state is dynamic.

2.1 Simulator

We divide the simulator into three parts. A simulator server, a map backend, and a battle simulator.

The battle simulator has four components which are battle interfaces, unit class, action class and skill class. The job of battle simulator is to calculate the result of a combat. The reason why we make a battle simulator is that a combat can be very complicated between different units who may have different skills and weapons. The input of battle simulator are two units instance, the result is updated info stored in those units. The unit class defines all the property of an unit, we plan to make our units versatile and user also can customize the unit themselves. Skill class defines all kinds of skills and weapons that unit can have. Action class just has few fields that specifies a particular action, it lists the source unit, the destination that unit wants to move, and the unit it wants to attack.

The map backend aims to maintain the map. The map is basically a m by n matrix where units are marked as 1 and all other parts are 0 for now, we may add more geographic elements like river or mountain in the future. The map backend is able to provide us a method to get action space and a method to execute the action. The way to get action space is to use depth first search algorithm to search for available destinations on the map of a given unit. After we store the destinations in a queue, we scan those destinations to calculate the distance of that destination to enemy units. If within attack range, we will add another action that combines move and attack into the final result indicating the unit can attack if move to that destination. So the return will have simple move action as well as move and attack action. The action method is simply carry out an action and update the result on the map. The return of that method should be a new state of map, the dead unit if there is any.

The simulator interface is in charge of communicating with agent. It has a step and a reset method which returns the state, reward, and done just like gym environment. We use two list to store living units of friendly and enemy, and get action space based on those units. If either of those lists become empty, we will check which side loses and return the reward and done to user. For now, we only have two sides and we set our enemy to be a silly one who only use random policy to move or attack. Our interface will be quite flexible and more teams could be added in the future if needed. Also each team has four members for now, but we can make it any number we want.

2.2 Network Structure

We design a Multilayer Perceptron with Experience Replay to learn the policy for this simulator, as shown in Figure 1. It takes the combination of states and actions as input and Q-value as output. It uses the same update scheme as Mnih et al. [2013]. As a baseline model, we also design a linear model without memory replay to solve this environment.

The input state s to our environment is a flattened vector of 48 units. The input action a_s is a vector of 4 units. We use 50 hidden units and ReLU activation for all three hidden layers. The output is a scalar, which is the estimated Q value given s and s_a. Since the input dimension is relatively low, we assume the training is not hard compared to prevalent convnets. Hence we don't use typical training techniques like batch normalization [Ioffe and Szegedy, 2015] and [Srivastava et al., 2014]. We use Adam optimizer [Kingma and Ba, 2014] to train the network. So momentums are used in the training process. We feed all the s and s_a pairs into our model sequentially and use an ϵ-greedy policy for selecting the action. Detailed parameter settings are illustrated in the experiment section.

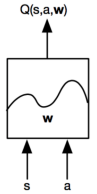


Figure 1: Network Inference Type

3 Experiment

We conducted the experiment on linear network and MLP network. The baseline is a random agent which chooses a random action on every step. The models are implemented using Keras. In future experiments, TensorFlow is considered for fine-grained customization.

3.1 Dataset

We implement a game simulator as the environment and an experience replayer in conjugation with the environment to provide training and testing data for our agent. The environment simulates a mobile turn-based strategy game Fire Emblem Heroes, whose episodes are deterministic MDPs. The dataset is generated by running the battle simulator. Each experience of $(sa\_pair, r, s'a\_pairs, done)$ is stored in experience replay. It is retrieved by sampling when agent is being trained.

3.2 Training

For the MLP agent, we train our three-layer model on a 4-core 2.5 GHz CPU for 200 update iterations, which takes about 30 minutes. We set the learning rate of Adam optimizer to 0.00005, and momentum parameters β_1 to 0.9, β_2 to 0.999 as the default setting. We set the experience replay size to 50000 samples and burn-in size to 10000 samples. We use an ϵ-greedy policy in the training process. The ϵ is initialized to 1 and decays to 0.05 after 200 episodes. The reward discount factor γ is set to 1.0, since our episode is finite-horizoed. The reward is set to be -1 for every step to penalize long episodes. The agent gets 100 if it wins or -100 if losses.

For the linear agent, we train our linear model for 1000 update iterations which takes about 30 iterations. We changed the learning rate of Adam to be 0.0005, every other parameter remains the same.

Another thing worth mentioning is the difficulty level of the enemy. The difficulty level is defined by the possibility of the enemy attacking the agent. The most difficult agent countered by our agent is the most aggressive agent which attacks its enemy whenever possible. So far the agent finds it hard to defeat, because this is nearly the most valuable move by now. But we expect the agent to learn the policy to focus their attacks on each enemy as more training is done. During training, a 0.05 difficulty enemy agent is used.

3.3 Testing

We use greedy policy in the testing process. Other hyper-parameters are the same as the training. In the linear baseline version, we test our model every 10 update iterations. In the three-layer DQN model, we test our model every 10 update iterations. In the test, we run 50 episodes and take the average winning rate as the performance metric. In future experiments, we develop a more systematic metric for measuring the learning process. We also use a 0.05 difficulty enemy when testing.

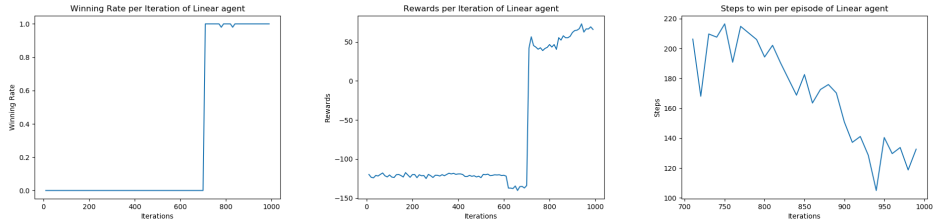
3.4 Preliminary Results

For any agent, we provide three metrics. First, the winning rate against agents with various difficulty levels. Second, the time steps for the agent to beat the enemy. This measures how efficient the agent is to achieve win the game. Third, the agents ability to defeat enemies stronger than itself. The enemy units are strengthened in hp to demonstrate the agent's ability level.

To show the effectiveness of our agents, we plot three figures for each one. The winning rate of the agent, the reward it achieves and the time steps it beats the enemy 2 3. From the graph we can see that, the MLP agent starts beating the random agent for sure at aroung 170 iterations. The linear agent, though it can finaly beat its opponent, only achieves this goal at around 700 iterations. In terms of the efficiency metric, the MLP also shows better efficiency than the linear agent in that it beats the enemy in 100 steps.

(a) Winning Rate (b) Rewards (c) Winning Steps

Figure 2: The performance of MLP agent through Q learning



(a) Winning Rate (b) Rewards (c) Winning Steps

Figure 3: The performance of Linear agent through Q learning

Furthermore, we tested the fully trained MLP agent on stronger enemies. The enemy has difficulty of 0.5 and its hp is increased according to figure 4. This figure shows surprisingly good result, since the fully trained agent could defeat opponents that has much stronger units than itself. This shows that learned agent performance is robust to changes in enemy units.

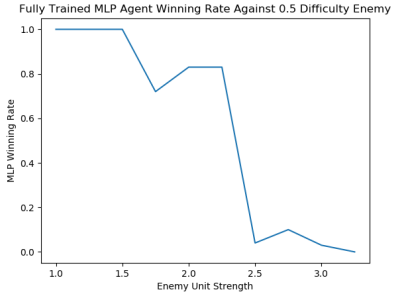


Figure 4: MLP Agent Against Enhanced Enemy Units

4 Conclusion

We implement a turn-based strategy game simulator and present a deep Q-learning based game AI on that. We use a three layer neural network as the Q function in the inference step. It outperforms a linear Q function in terms of iterations needed for convergence, overall training time and the average reward throughout training. In the next phase of our project, we continue the improvement of the network architecture and add complexity to the simulator to the full extent of the original game.

References

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456, 2015.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676): 354, 2017.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1): 1929–1958, 2014.

C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.