Learning to play a customized overhead-shooting

Haoming Hu
CS department, Viterbi
University of Southern California
Los Angeles, CA
haomingh@usc.edu

Haoqin Deng
ECE department, Viterbi
University of Southern California
Los Angeles, CA
haoqinde@usc.edu

Haoyun Zhu
CS department, Viterbi
University of Southern California
Los Angeles, CA
haoyunzh@usc.edu

Lihan Zhu
CS department, Viterbi
University of Southern California
Los Angeles, CA
lihanzhu@usc.edu

***Abstract*—We developed a customized overhead-shooting game and trained our Artificial Intelligent (AI) agent to play it. We applied several machine learning algorithms, including Genetic Algorithm (GA), Deep Q learning (DQN), Double Deep Q Learning (DDQN), Dueling DDQN, Deep Recurrent Q Learning (DRQN), Actor-Critic, and Proximal Policy Optimization (PPO), and compared their performances. In the end, our AI agents were able to achieve close to full score on our customized game environment and discovered unexpected but reasonable gaming strategies.**

***Index Terms*—Customized Overhead-shooting, GA, DQN, DDQN, Dueling DDQN, DQRN, Actor-Critic, PPO**

I. Introduction

The game of overhead shooting is one kind of popular computer game, which has large amounts of audience around the world. In the context of the game, the player usually controls an agent to move or attack and other operations against a computer or a real person. Taking down enough fighters while maintaining the players' health is the primary goal of the game and to get higher scores. Players will become more interested in this game if they can meet smart enough opponents. Recently, research on the application of artificial intelligence to computer games has attracted more and more attention, showing promising research value. Google's A.I. program, AlphaGo, defeated the world's best player Ke Jie, and caused a sensation throughout the world. In fact, there has been a lot of research on how to train an intelligent agent in 2D games. Genetic algorithms[1] borrows the idea from natural selection and can perform well on simple games such as Flappy Bird. However, it suffers from the problem of high training cost and the local-minimum problem. Q learning in a version of temporal difference algorithm. It keeps a Q table to register the desirability of state-action pairs. However, it is unscalable in a more complicated game environment because the overly large number of states requires a colossal size of Q table that is unrealistic to store. Deep Q Learning[2] resolves this issue by replacing the Q table with a Q network to achieve good interpolation ability. A number of variants of DQN, such as Deep Recurrent Q network[3], double DQN[4], dueling DQN[5] have been proposed later. Apart from these value-based algorithms, there are also policy-based algorithms, such as the actor-critic model[6] and its variant PPO algorithm[7]. These state-of-the-art algorithms can achieve high scores on Atari games in the open AI gym. Since our overhead-shooting game scenes resemble those Atari games, it is presumably suitable to be solved using the aforementioned machine learning algorithms. However, not much research has been done in comprehensively evaluating the best strategies in the customized 2d overhead shooting games. In this work, we fill this gap by investigating the performances of various mainstream models in our self-designed 2D overhead-shooting game. Our contribution will be two-folds: 1) design and develop a 2D overhead shooting game environment that is easily extensible and compatible with open AI gym environment and mainstream Python models. 2) implement various mainstream ML algorithms, compare and analyze their performances in our specific environment. By bringing intelligent AI into overhead shooting games, we can improve the entertainment to a new level—the players will enjoy a great deal of fun in this game when they battle smart opponents. This will be a great bonus feature of a game and will potentially attract more people to play. Such is the impact of our work.

II. Models and methods

A. Deep Q Learning

The Q-learning algorithm assigns a value $Q(A, S)$ to an action-state. A higher Q value indicates that the action at that state is expected to yield a higher return. The Q-learning algorithm iteratively updates the Q value using the temporal difference method:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \quad (1)$$

where $Q(s', a')$ is the Q value of the action a' in the next state s', r is the reward given by the environment, γ is diminishing return coefficient, α is the learning rate.

We are able to construct the Q table and update it in some simple environments. However, when the environment

is complicated, the number of states will be too large to be kept track of in the Q table, hence we need a neural network, parameterized by (θs) to approximate the Q table. Here, we iteratively update the Q network using the temporal difference method and minimize the loss function:

$$Q^*(s_t, a_t) = r + \gamma \max_{a'} Q(s_t + 1, a') \tag{2}$$

$$L(s, a|\theta_i) = (Q^*(s_t, a_t) - Q(s_t, a_t))^2 \tag{3}$$

$$\theta_{i+1} = \theta_i + \alpha \nabla_\theta L(\theta_i) \tag{4}$$

the agent starts off being exploratory and gradually becomes greedy in selecting actions. Its experiences are stored in the memory buffer and fed into the Q network to update js. In the end, the Q network will accurately evaluate the desirability of an action given a state input.

B. Double Deep Q Learning

The DQN method uses the same Q network to both select and estimate Q values. Such estimating from the estimation approach tends to maximize the bias, creating the overestimation problem. The DDQN method introduces a second Q' network to untangle the selection from evaluation. The target Q value is calculated as follows:

$$Q^*(s_t, a_t) = r_t + \gamma Q(s_t + 1, \max_{a'} Q'(s_{t+1}, a')) \tag{5}$$

The loss function and its minimization is the same as DQN. The parameters θ's of Q' network are periodically updated by copying ϑs of the Q network:

$$\theta' = \tau * \theta + (1 - \tau) * \theta' \tag{6}$$

C. Dueling Double Deep Q Learning

Sometimes edge states may not be important. The Dueling architecture takes this into account by designing an advantage function A that subtracts the state value $V^\pi(s)$ from the action value $Q^\pi(s, a)$. The DQN network is split into two streams to compute the action and the state value. The target Q value is computed using the following equation:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \max A(s, a; \theta, \alpha)) \quad a' \in |A| \tag{7}$$

D. Deep Recurrent Q Learning

The original DQN method takes four channels of consecutive frames as inputs in order to capture the temporal information such as movement. However, this method is known to perform poorly in the partially observable environment. In our game, the partial observation means the possible presence of frame flickering, which will hide some of the information about the state. Recurrent Neural Network (RNN), and one of its versions, Long-Short Term Memory (LSTM), inherently captures temporal information and is shown by[4] to be resistant to the frame flickering problem, DRQN replaces the linear layer in DQN with an LSTM layer. Instead of taking four frames as input, DRQN unrolls them into multiple time steps and processes one frame each step. There are two ways of sampling from memory buffer: randomly sampling an entire episode, and random sampling a sub-sequence of the entire episode. We implemented the latter approach.

E. Genetic Algorithm

GA simulates the mechanism of natural selection where the fittest survives in a highly random process of evolution.

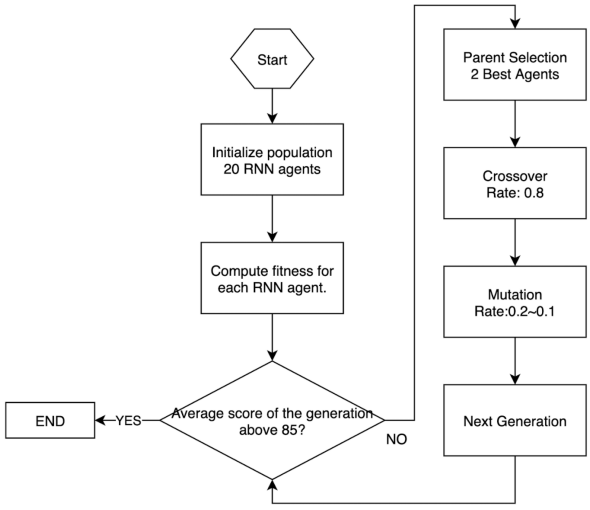

Figure 1. Basic GA Workflow

The advantage of the algorithm is the elegant simplicity compared to other theoretical methods in terms of understanding and implementation in practical applications. Also, it endures a relatively high amount of noise in the samples. This is because it treats the population as a whole which performs multiple parallel learning procedures at the same time. Combining with the mechanisms of "ranking selection" (which keeps only two best results from one generation), GA easily combs out alienated results and inherits the most suitable ones to the current environment. As a result, GA iterates by generations and evolves towards a local minimum at a high speed. Another main strength of GA we would care about is that it requires us a much lower threshold for the problem definition and environment inputs. This works well especially when inputs vary in a wide spectrum and strategies are not unique or clear to the goal which is, by the way, exactly our case.

However, the same knife cuts bread and fingers. The highly stochastic procedure limits the ability for agents to get out of local minimums. The algorithm does not guarantee a final convergence and is very likely to swing back and forth in the midway, taking up huge computational resources but giving out few constructive results. Thus, the unpredictability of the learning progress is another fatal weakness. It could be wandering at some low point one moment and pops out a historical best result the other moment because of a tiny mutation, vice versa. The mercurial results prevent us from terminating the training process immediately when it slips into an apparently

unreasonable result. Considering the computational power in our case, we decided to set the GA parameters as follows:

1) Population size: 20.
2) Mutation rate: 0.2 0.1 decreasing throughout the whole process.
3) Mutation Portion: 0.01, meaning 1% of a model would be modified when it mutates.
4) Mutation Scale: -10 10.
5) Crossover rate: 0.8, meaning that 8 out of 10 times, two parent models selected in each generation would exchange part of their weight vector to their off-springs.
6) An individual agent bases on a Recurrent Neural Network (RNN) as shown in figure below.
7) 500 maximum generations. Any results achieved over 500 repetitions would be meaningless for the low rate of return.

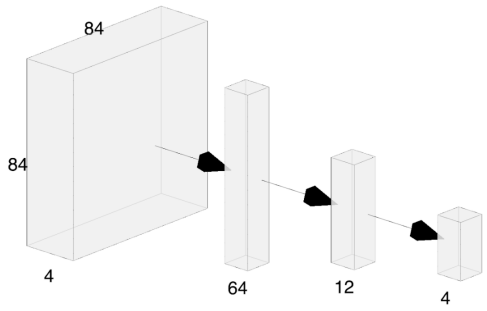

Figure 2. Structure of RNN Used in GA

F. Actor-Critic

Actor-critic algorithm is the combination of value-based and policy-based algorithms. The model consists of two networks: an actor choosing action based on current game state and a critic to calculate the Q value of the actor's action. The learning of the actor is using policy gradient while the learning of the critic is using temporal difference (TD).

The actor and the critic networks can share a large part of a neural network. Both of them need several convolution layers to extract features from the input images and one or two fully connected layers. The only difference is the final output layer. The actor network has an output size equal to the number of actions while the critic network has an output size of one (Q value).

The loss of actor-critic networks has two parts as well. The first part is policy loss, computed by policy gradient. And the second part is value loss, computed by temporal difference (TD).

G. Proximal Policy Optimization

Proximal Policy Optimization, one of popular reinforcement learning models in recent years, is an improvement of the policy gradient descent algorithm.

Algorithm 1 Pseudo-code: Actor-critic algorithm

while in each step of training **do**
  Observe the state.
  Randomly sample action according to $\pi(\cdot|s_t;\theta_t)$.
  Perform a_t and observe new state s_{t+1} and reward r_t.
  Update value network N_{value} with temporal difference (TD).
  Update policy network N_{policy} with policy gradient.
end while

Compared with previous models, PPO algorithm makes effective use of data and adopts the method of importance sampling. The sample data at one time can be used in several iterations in PPO while the same data need to be discarded after only one iteration in Vanilla policy gradient. It greatly improves the efficiency of using data and accelerates the training of the model. What's more, PPO can achieve satisfactory results on many classic reinforcement learning tasks. So it's necessary for our team to try PPO to our game task and see the performance.

PPO uses clipped surrogate objective to penalize large policy updates:

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)] \tag{8}$$

III. Experiments

A. Experimental Setup

We run all experiments on the Nvidia Tesla K80 GPU, n1-standard-4 CPU, in Ubuntu 16.04 on Google Cloud Platform (GCP).

B. Game Environment

The game is developed using PyGame. Figure 3 shows the conceptual design of the game. The game has a 2D overhead bird view. There are pre-scripted enemies and AI players that can shoot, move, or do nothing. The goal of the game is to shoot enemies and avoid their bullets. Note that the game can be easily extended to incorporate more complicated behaviors.

The game is wrapped to inherit the Environment class in open AI gym, so that it can directly interface with ML algorithms written in python. Table 1 summarizes the important functionalities:

Action Space	{NOOP, LEFT, RIGHT, FIRE}
Observation Space	screenshot of frames: (500, 600, 3) NumPy array
Reward	Hitting enemies: +10 Being hit by enemies: -10 Time passage: -0.005
Terminal condition	All enemies' health ≤ 0 AI's health insurance ≤ 0 Steps ≥ 3000
Score	Sum of reward

Table 1. Parameters

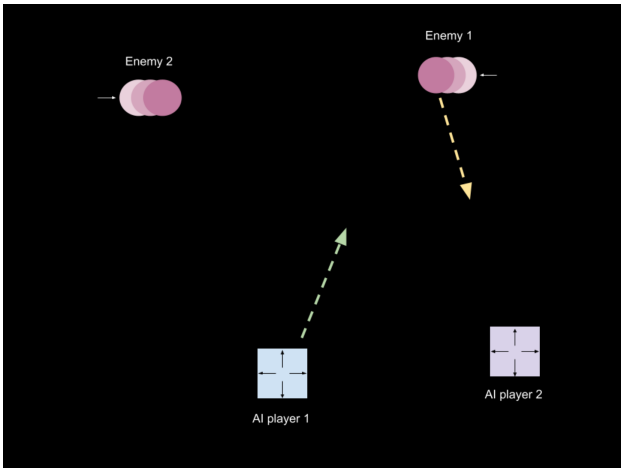

Figure 3. Conceptual design of our overhead-shooting game environments

C. Dataset

As the agent plays more games, it stores its experienced information into the memory buffer that contains tuples of (observation, reward, done, next_observation, action). The information inside the memory buffer is our training data.

Our observations are all screen shots from the game. Each piece of data is a 500 by 600 pixel RGB image. However, the color information is redundant for the agent to make decisions in this game. Original overhead shooting games only use black and white pictures while players will not be affected when making decisions in games. Therefore, to save computing resources and accelerate the convergence model, We reduce the dimension of image data by converting $3 \times 84 \times 84$ RGB images into $1 \times 84 \times 84$ gray-scale images.

We stack four down-sampled gray-scale images to form a $4 \times 84 \times 84$ inputs. Convolution neural networks then are used to extract features from these images as one part of input into reinforcement learning models that are in the training process.

D. Evaluation

In our game, the goal of the game design is to train an agent with sufficient intelligence to play games with players. We designed two indicators to measure the performance of the model:

Scores:

$$scores = \sum_{i=1}^{n} reward_{T=i}(WhenT = n, gameendsup)$$

$$scores \in [-100, 100] \quad (9)$$

We evaluate the model by summation of agent's reward scores.

Convergence speed:
It is well known that reinforcement learning requires enormous amounts of computational resources. Agents need a lot of interactive feedback with the environment to learn knowledge. Therefore, the model that can obtain similar performance through fewer training iterations should be given priority due to limited computational resources.

IV. Results

A. Performances

Figure 4 shows the performance of Genetic Algorithms. We see that after significant score increase in the first 200 generations, the model ceases to improve and oscillates at around 50 points. Note that the full score is 100 points.

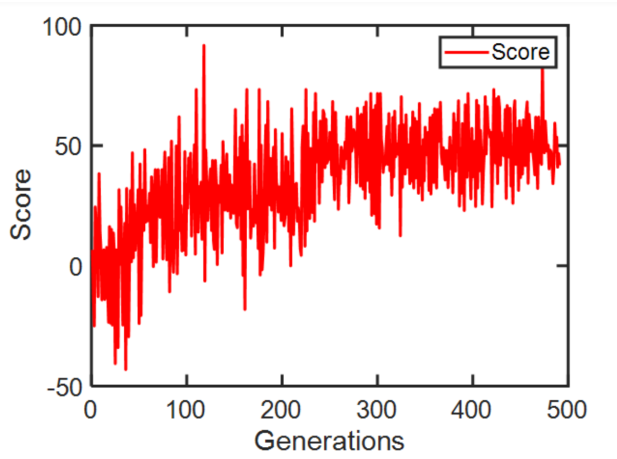

Figure 4. The training curve of Genetic Algorithm

Figure 5 shows the performances of the DQN model and its variants, DDQN, Dueling DDQN, and DRQN. We see that DQN and DRQN agents are able to achieve close to full score, 100 points. DDQN also approaches full score but oscillates more. Dueling DDQN does not converge.

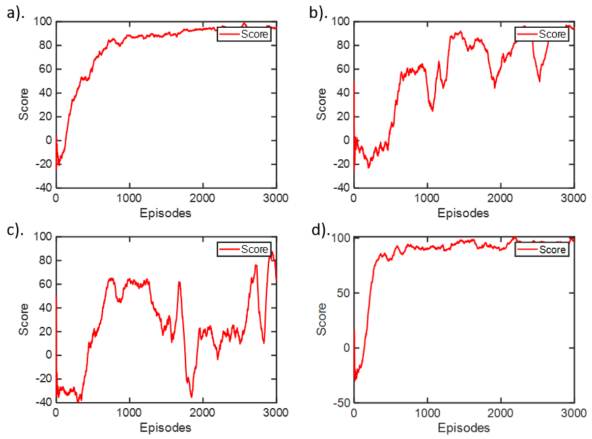

Figure 5. a). The training curve of DQN. b). The training curve of DDQN. c). The training curve of Dueling DDQN. d). The training curve of DRQN

Figure 6 shows the performances of the PPO model. It strongly oscillates around 0 scores and does not display a strong learning behavior beyond this point.

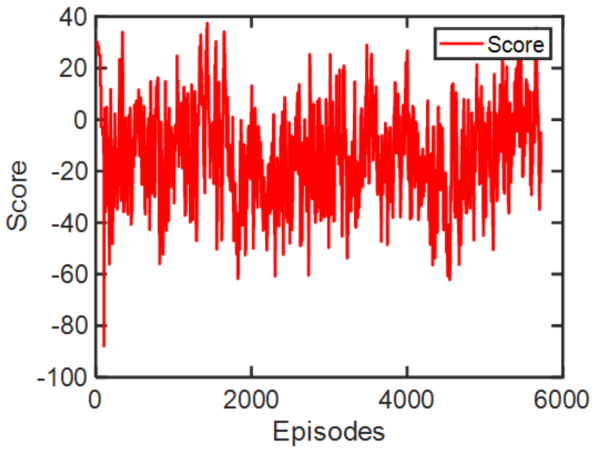

Figure 6. The training curve of the PPO model.

V. Discussion

We first diagnosed the causes of the relative poor performances of GA, PPO, and Dueling DDQN. We loaded each of these models and observed how AI behaves. We discovered that they all share one common syndrome—AI agents like to move to the corner and stay there forever (Figure 6.a). This happens because our pre-scripted enemy is always a few pixels away from the side of the window; therefore, staying at the corner can help AI avoid the enemy's bullets. This is certainly a safe policy, but it also makes it impossible for AI to shoot the enemy either, and the total score will always stay at around 0. We consider such a situation as the local minima, as it is a stable yet sub-optimal situation. One potential cure is to add another penalty if the AI agent is too close to the side of the window to force it to stay in the center.

We then loaded the models of DQN, which can achieve close to full scores, and observe the agent's behavior. We find the agent to be remarkably intelligent. The agent is not only able to follow the enemy but also able to predict its movement and fire bullets ahead of time. In addition, the agent can adjust its movement when bullets are close. What's more, the agent prefers to fire bullets consecutively to maximize the chance of hitting the enemy (Figure 7.b). Lastly, the agent actively moves in the center part of the window, rather than staying at the corner.

We also loaded the models of DRQN and observed a similar behavior as DQN. However, one difference is that DRQN runs noticeably more smoothly than DQN agent does. This is because while DQN takes in four stacked frames as inputs, DRQN only takes only one frame at each time step and can therefore process images much faster.

VI. Conclusion

Overhead shooting games were chosen as the environment for the application of machine learning. As a popular game, training an agent smart enough to play against the player can increase the players' interest in the game. We explored several

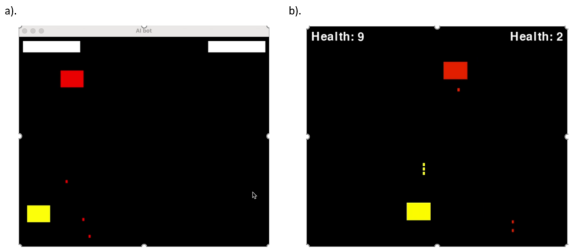

Figure 7. a). The screenshot of a GA agent playing the game. b). The screenshot of a DQN agent playing the game

mainstream methods of machine learning to train our agents to play the game.

GA, DQN, DDQN, PPO and other reinforcement learning models have been tested in the game environment. DQN and DRQN have better performance than other models. They can achieve nearly a full score. Compared with DQN, the DRQN inference speed is faster. PPO, GA, Dueling DQN do not converge after a long time of training and need further improvement.

VII. Future Work

Our future work includes:

1) Add more pre-scripted enemies.
2) Replace pre-scripted enemies with AI enemies.
3) Apply imitation learning to see if it can accelerate training.
4) Add multiple AI agents to train collaborations.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. (references)
[2] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).
[3] Hausknecht, Matthew, and Peter Stone. "Deep recurrent q-learning for partially observable mdps." 2015 aaai fall symposium series. 2015.
[4] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." Proceedings of the AAAI conference on artificial intelligence. Vol. 30. No. 1. 2016.
[5] Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." International conference on machine learning. PMLR, 2016..
[6] Konda, Vijay R., and John N. Tsitsiklis. "Actor-critic algorithms." Advances in neural information processing systems. 2000.
[7] Schulman, John, et al. "Proximal policy optimization algorithms." arXiv preprint arXiv:1707.06347 (2017)..