Learning to Play a Customized Overhead-shooting Game

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. For the simpler game environments, we applied several vanilla machine learning algorithms and compared their performances, 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). For more complex game environments, we modified DQN to take vector inputs instead of screenshots. We used YOLO-v3 to extract object positions from images. In addition, we applied pre-training technique to boost convergence speed. At the end, our AI agents were able to develop reasonable gaming strategies and achieve high score on our customized game environment.**

***Index Terms*—Customized Overhead-shooting, GA, DQN, DDQN, Dueling DDQN, DQRN, Actor-Critic, PPO, Vector Inputs, YOLO-v3, Pre-training**

I. Introduction

Overhead-shooting game is one of the most popular types of video games. In a typical overhead-shooting game, players usually control an agent to move, attack or perform other operations against a computer or real opponents. Taking down enemies while keeping the player's agents safe is the primary goal of the game. Players will become more challenged and attracted by the game if their enemies are highly intelligent.

Recently, researches on artificial intelligence applications in video games have attracted increasing attentions. Google's A.I. program, AlphaGo, defeated the world's best player Lee Sedol and caused a sensation throughout the world. To date, there have been a lot of researches on how to make an intelligent agent in 2D games. Genetic Algorithms [2] borrows the idea from natural selection and performs well on simple games such as Flappy Bird. However, it suffers from the problem of high training cost and is susceptible to getting stuck in local-minimum. Q-learning is a version of temporal difference algorithm. It keeps a Q table to register the desirability of state-action pairs. However, Q-learning is unscalable for more complicated game environments because it requires storing a colossal size of Q table. Deep Q Learning [6] resolves this issue by replacing the Q table with a Q network. Furthermore, a number of variants of DQN were proposed later, including Deep Recurrent Q network [4], double DQN [11] and dueling DQN [12]. Apart from these value-based algorithms, there are also policy-based algorithms, such as the Actor-critic model [5] and its variant PPO algorithm [9]. More complicated game environments demand more powerful techniques, such as pre-training methods [1]. They can boost a model's ability to converge in the initial stage of training. Object detection algorithms, such as Fast RCNN [3] and the YOLO family [7], [8] can be used to extract object positions from images. These state-of-the-art algorithms have been demonstrated to achieve high scores on Atari games wrapped in the Open AI Gym library. Since our overhead-shooting game scenes resemble those of Atari games, the aforementioned machine learning algorithms are presumed to be suitable.

On the other hand, little research has been conducted in comprehensively evaluating the best strategies in a customized 2D overhead-shooting game. 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 is two-fold: 1) design and develop a highly extensible 2D overhead-shooting game environment that is 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 incorporating intelligent AI with overhead-shooting games, we are able to bring entertainment to a new level—players will enjoy great fun when they battle smart AI enemies. This will be a bonus feature of a game and will potentially attract more players.

The rest of the paper is organized as follows. Section II introduces the background information of the machine learning models we used to train our agents. Section III introduces the data set for training and the features of the game environment. Section IV describes in details how we implemented the aforementioned machine learning algorithms to train AI agents to

play our customized game. Section V reports the performances of the models on different versions of our customized game. Section VI discusses the interpretation of our results. Section VII concludes our whole project. Section VIII discusses future work.

II. Backgrounds

A. Deep Q Learning

The Q-learning algorithm assigns a value $Q(A, S)$ to an action-state pair. 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 the diminishing return coefficient, and α is the learning rate.

Q table can be updated in some simple environments. However, when the environment is complicated, the number of states will be too large to be stored in the Q table. Hence, a neural network, parameterized by (θs), is used to approximate the Q table. Here, the Q network is iteratively updated with the temporal difference method to minimizes the loss function:

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

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

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

The agent starts off by 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 (θs). 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 an 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:

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

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

$$\theta' = \tau * \theta + (1 - \tau) * \theta' \quad (6)$$

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:

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

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 performs poorly in the partially observable environment [4]. In our game, partial observation means the possible presence of frame flickering, which hides some information of a state. Recurrent Neural Network (RNN) and one of its versions, Long-Short Term Memory (LSTM), inherently capture temporal information and are shown to be resistant to the frame flickering problem [4]. 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: (1) randomly sampling an entire episode or (2) randomly 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. The advantage of the algorithm is its elegant simplicity compared to other theoretical methods. Also, it endures a relatively high amount of noise in the samples because it treats the population as a whole and performs multiple parallel learning procedures at the same time. Combined with the mechanisms of "ranking selection" (which keeps only two best results from one generation), GA combs out alienated results and selects 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 is that it demands a much lower threshold for problem definition and environmental inputs. This works well especially when inputs vary in a wide spectrum and strategies are not unique or clear to the goal.

However, the highly stochastic procedure limits the ability for agents to get out of local minimum. The algorithm does not guarantee a final convergence and is 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 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 researchers from terminating the training process immediately when it slips into an apparently unreasonable result.

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 calculating the Q value of the actor's action. The actor learns with policy gradient while the critic learns by using the temporal difference method.

The actor and the critic networks share a large part of a neural network. Both of them need several convolution layers to extract features from input images and one or two fully connected layers. The only difference is the final output layer, in which 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. The second part is value loss, computed by the temporal difference method. The pseudo algorithm of actor critic is shown in Algorithm 1.

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

G. Proximal Policy Optimization

Proximal Policy Optimization is an improvement of the policy gradient descent algorithm.

Compared against previous models, PPO algorithm makes effective use of data and adopts the method of importance sampling. The sampled data 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 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.

PPO uses clipped a surrogate objective to penalize large policy updates:

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

H. Pre-training

Reinforcement learning often suffers from unstable convergence. Pre-training methods are therefore developed to address this issue. In the work of [1], both the model transfer and the experience transfer are utilized.

A pre-training data set is generated by letting a non-expert human player to play the game for a large number of episodes. From this data set, supervised training is used to find the optimal mapping between stacked-frame image inputs and corresponding actions. The network used in supervised training has the same topology as the Q network in the DQN algorithm. The weights of the trained network will be used to initialize the weights of the Q network in the actual DQN training.

All the state transitions occurring during human-play episodes are used to fill the initially-empty memory buffer. In addition, the initial exploration rate is lowered because human players already finish the exploration process.

I. YOLO

YOLO (You Only Look Once) is an object detection algorithm that can efficiently extract object positions from image inputs. To date, there are five versions of YOLO. YOLO v3 is used in this paper [8].

In YOLO v3, images first go through deep convolutional layers for feature extraction. The outputs are divided into three different sizes of grids to accommodate size variation of objects. To stabilize the initial stage of training, anchor boxes are pre-computed with k-means clustering method. Object positions are represented by vectors of normalized relative positions between bounding boxes and anchor boxes. The difference between the predicted object positions and the ground truth is computed as the loss function and gets iteratively minimized.

III. Data and Environment

A. 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 1 shows the conceptual design of the game. There are three versions of the game (Env v1.0, Env v2.0, Env v3.0), where higher version number represents increasing complexities. Table I summarizes their differences. 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 hit as high scores as possible. Generally, the player is rewarded when it hits an enemy or collects desirable objects (treasures, health packs); the player is punished when it is hit by bullets or run into undesirable objects(traps). Note that the game can be easily extended to incorporate more complicated features.

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 II summarizes the important functionalities:

C. Data Set

1) Reinforcement Learning: As the agent plays more games, it stores experienced information in its memory buffer that contains tuples of (observation, reward, done, next_observation, action). Observations are screenshots from

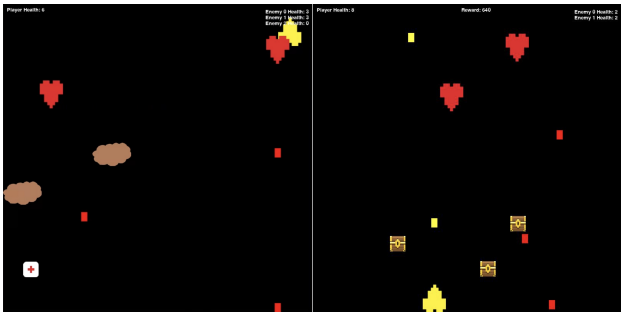

Figure 1: Scenes of our overhead-shooting game environments

Version	Action space	Features
Env v1.0	{NOOP, LEFT, RIGHT, FIRE}	One enemy, consecutive shooting enabled
Env v2.0	{NOOP, LEFT, RIGHT, FIRE}	Two enemies, consecutive shooting disabled
Env v3.0	{NOOP, LEFT, RIGHT, FIRE, UP, DOWN}	Two enemies, consecutive shooting disabled, treasures

Table I: Summary of game environments

the game, and each piece of data is a 600 by 600 pixel RGB image. The information inside the memory buffer is our training data.

In Env v3.0, we applied the pre-training method. In this case, all information in the memory buffer are experiences of a human player instead of an AI agent.

2) Object Detection: We let the agent play the game for 3000 frames and stored one screenshot every ten frames to increase the dissimilarities of the sampled images. The sampled 300 images are all labeled automatically such that each image is correctly associated with a list of positions of all objects, including the enemy spaceship, the player spaceship, bullets, and treasures.

IV. Methods

A. Overall Architecture

Figure 2 shows the overall architecture of the project. Pre-training is used to initialize network weights. YOLO v3 is used to extract object positions. CNN layers are used for extracting features of input images. An LSTM layer is used to retain past information and resolve the frame-skipping issue. All modules are weaved together into the temporal-difference architecture that covers a wide range of reinforcement learning algorithms that include DQN and PPO.

B. Env v1.0

1) Genetic Algorithm (GA): The overall workflow of our genetic algorithm is shown in Figure 3. All the training parameters are set as follows:

(i) Population size: 20.
(ii) Mutation rate: 0.2 - 0.1 decreasing throughout the whole process.
(iii) Mutation portion: 0.01, meaning 1% of a model would be modified when it mutates.

Action Space	{NOOP, LEFT, RIGHT, FIRE}
Observation Space	screenshot of frames: (600, 600, 3) NumPy array
Reward	1) Hitting enemies: +10 2) Being hit by enemies: -10 3) Time passage: -0.005 4) Collect treasures: +10
Terminal condition	1) All enemies' health ≤ 0 2) AI's health insurance ≤ 0 3) Steps ≥ 3000
Score	Sum of reward

Table II: Parameters

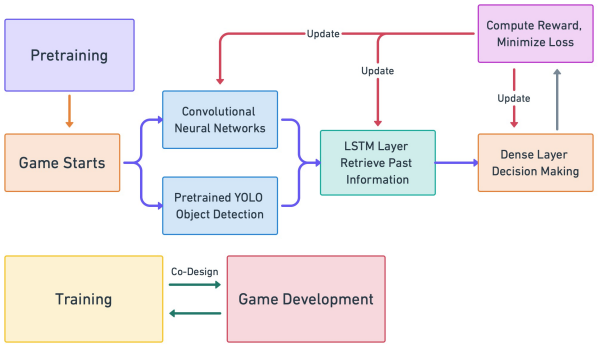

Figure 2: Project architecture

(iv) Mutation scale: -10 - 10.
(v) 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.
(vi) An individual agent bases on a Recurrent Neural Network (RNN) as shown in figure below.
(vii) 500 maximum generations. Any results achieved over 500 repetitions would be meaningless for the low rate of return.

2) DQN, DDQN, Dueling DQN: The three methods share the same deep Q network topology, as shown in Figure 4.a. The only difference between the three models are how their loss functions are calculated, which is describe in Section II and implemented accordingly.

3) Pre-processing: Observations are screenshots from the game, which are 600 by 600 pixel RGB images. The raw screenshot images are down-sampled and gray-scaled to 84 by 84 single-channel images. This compression of input data can boost training speed. At each time step, four most recent, consecutive frames of screenshots are stacked together to capture the temporal information such as velocity. As a result, the final input data after pre-processing have a shape of (84, 84, 4).

All the relevant parameters are summarized in Table III.

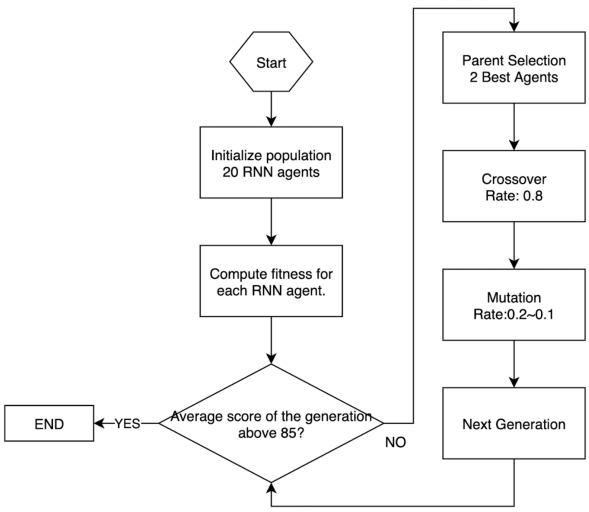

Figure 3: Genetic algorithm workflow

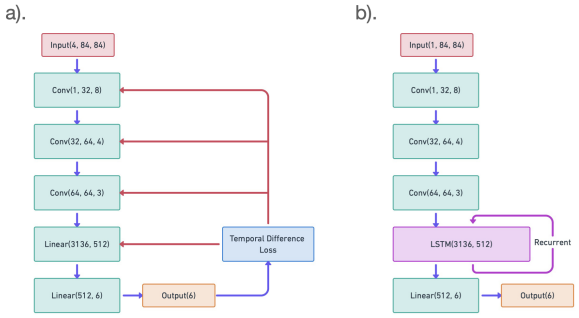

Figure 4: a). Network topology of DQN, DDQN, Dueling DQN. b). Network topology of DRQN

C. Env v1.1

Env v1.1 adds the frame skipping feature to Env v1.0. Here, one frame is skipped every eight frames.

1) DRQN, LSTM: We implemented a DRQN (deep recurrent Q network) by replacing the first linear layer of deep Q network with an LSTM layer, as shown in Figure 4.b. Incorporation of a recurrent layer (LSTM) brings two advantages: (1) An LSTM layer has the ability to retain past information. In theory it can be resilient to the frame-skipping issue because other past information can be retained to make up for the information loss of a skipped frame. (2) DRQN has faster inference speed because only a single frame, as opposed to stacked four frames, is used as the input of the neural network.

We applied both DQN and DRQN on the frame-skipping game environment and compared their performances.

D. Env v2.0

As described in section III-B , Env v2.0 presents a more difficult task since more enemies are present and the player cannot fire consecutively. We applied the same DQN model as we did in Env v1.0 to train the agent. We also used guided grad-CAM [10] algorithm to plot the heat map of the neural network.

Parameter	Value
Inputs	(84, 84, 4)
Outputs	actions
Number of episodes	2000 - 3000
Network architecture	Conv*3 $\rightarrow$ linear*2
Replay buffer size	30000
Rate of diminishing return	0.99
Epsilon delay	0.01
Batch size	32

Table III: Summary of Training parameters

E. Env v3.0

In this more complicated environment, the player has a larger action space as it can move up/down in addition to left/right.There are also more objects, which are treasures, that give the player additional rewards if collected. In this environment, we applied DQN-vector and pre-training in addition to the original DQN model.

1) DQN: The DQN model we used is the almost the same as the DQN model we used in Env v1.0 and Env v2.0, except that there are two more neurons in the output layer, which corresponds to the larger action space due to the additional UP/DOWN movements.

2) DQN-vector: In this model, instead of taking screenshot images as inputs, we put relevant information in a vector in order to capture the game state more accurately. The vector includes normalized coordinates of players, enemies, bullets, and treasures. The vector has a fixed size of 36, and empty objects are padded with (-1, -1). As before, to capture temporal information, we concatenated the four most recent vectors to represent the current state. Consequently, the state of the game is represented by a 1D vector of length 36*4. All the convolutional layers in the original DQN is replaced by one linear layer of shape (36*4, 512).

3) Pre-training: We let human players play the game for 10000 frames, and stored all the state transition information locally. It is used to initialize agents' memory buffer. We also applied supervised training on a neural network, which has the same topology as the deep Q network, to find the optimal mapping between states (inputs) and actions (classification) through a cross-entropy loss function. The trained weights will be used to initialize the weights of the deep Q network. We set the initial exploration rate ϵ to 0.1.

All other parameters are the same as the original DQN.

F. YOLO v3

YOLO v3 is used in our project mainly for pre-processing screenshots of states. It is used to extract position information from a image.

We used the baseline YOLO v3 implementation from [8]. It is applied on self-generated data set, described in section III-C. We trained for 96 epochs, until loss can no longer be decreased. The batch size is set to be 4.

G. Evaluation metrics

The performances of models are measure by two metrics:

1) **Scores:**

$$scores = \sum_{i=1}^{n} reward_i (n \ is \ the \ end \ of \ game) \tag{9}$$

The model is evaluated by the summation of agent's reward scores.

2) **Convergence speed:**
The model that can obtain similar performance through fewer training iterations should be considered better due to limited computational resources. Often, reinforcement learning requires enormous amounts of computational resources because agents need a lot of interactive feedback with the environment to learn.

V. Results and Analysis

A. Env v1.0

1) Genetic Algorithm: Figure 5 shows the performance of Genetic Algorithm. We see that after a significant score increment in first 200 generations, the model stops improving and oscillates around 50 points. Note that the full score is 100 points.

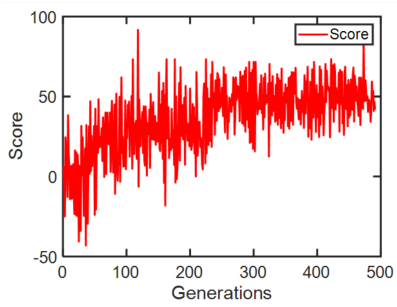

Figure 5: The training curve of Genetic Algorithm

2) DQN, DDQN, Dueling DQN, DRQN: Figure 6 shows the performances of the DQN model and its variants, DDQN, Dueling DDQN, DRQN. We see that the 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.

3) PPO: Figure 7 shows the performance of the PPO model. It intensively oscillates around 0 and does not show significant improvement beyond this point.

4) LSTM, frame-skipping: Figure 8 shows the performances of DQN and DRQN on Env v1.1, the frame-skipping environment. It can be observed that the DQN model collapsed due to the loss of information. On the other hand, the DRQN model only experiences a slight drop in scores. This experiment confirmed our hypothesis that the usage of LSTM layer can indeed preserve past information and is more resilient to loss of frames.

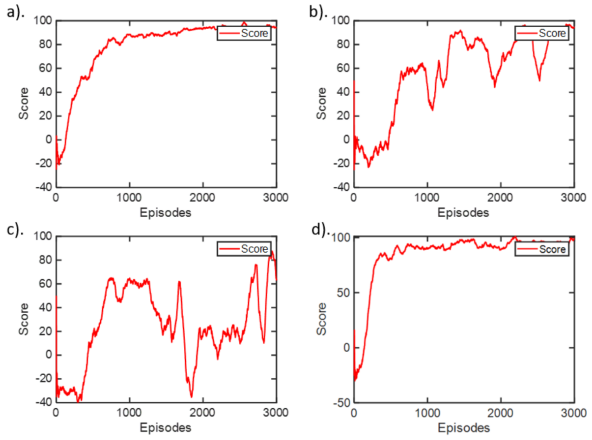

Figure 6: a). The training curve of DQN. b). The training curve of DDQN. c). The training curve of Dueling DDQN. d). The training curve of DRQN

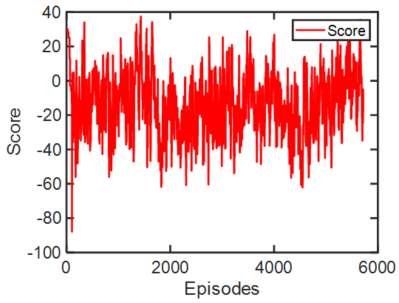

Figure 7: The training curve of the PPO model

B. Env v2.0

Figure 9.a shows the performance of DQN model on Env v2.0. It can be observed that in this more complex game environment, the DQN model is taking a longer time to converge and achieves a lower score than it does in Env v1.0. Figure 9.b shows the heat map of the trained deep Q network. We see that the network focuses more on the red regions where enemy and player spaceships are located; the network focuses less on the green regions where the bullets are more sparsely distributed; the network totally ignores the blue regions, where there is no objects at all. These results verify that our trained deep Q network looks at the right places on the image.

C. Env v3.0

Figure 10 shows the performances of image-input DQN, vector-input DQN, and pre-trained DQN on Env v3.0, the most complicated environment. We observe that vector-input DQN outperforms the image-input DQN, which makes sense because the former takes much more accurate information of the game state. The pre-trained DQN improves its score much more quickly than other models. However, its scores soon stabilize and stop increasing.

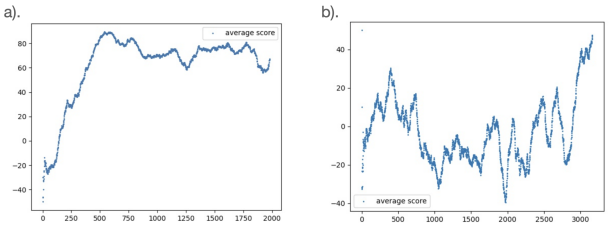

Figure 8: a). Performance of DRQN on frame-skipping environment b). Performance of DQN on frame-skipping environment

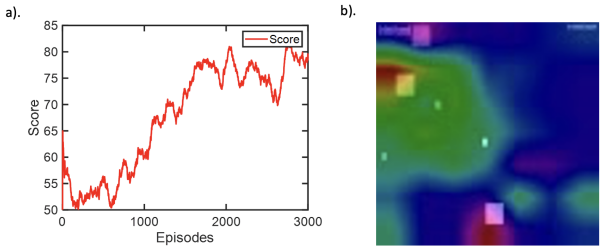

Figure 9: a). Performance of DQN on env v2.0. b). Heat map of DQN

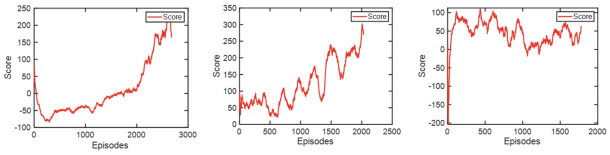

Figure 10: a). Performance of image-input DQN. b). Performance of vector-input DQN. c). Performance of DQN with pre-training

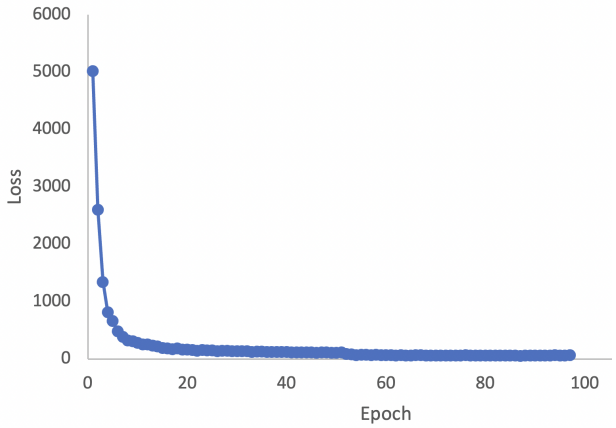

Figure 11: Training curve of YOLO v3 on self-generated data set

D. YOLO

Figure 11 shows the training curve of YOLO v3. It shows that the loss, which quantifies the difference between predicted and actual bounding boxes, is steadily decreasing.

VI. Discussion

First, we tried to diagnose the causation of relative poor performances of GA, PPO, and Dueling DDQN in Env v1.0. We loaded each of these models and observed how AI behaves. We discovered that they all share one common syndrome—AI agents tended to move towards a corner and stay there for the rest of a game (Figure 12.a). This happened because we set the bouncing box of our pre-scripted enemies a few pixels away along each side of the canvas; therefore, staying at a corner guarantees AI agents to avoid 100% of the enemies' bullets. This is certainly a safe policy, but it also prevents the agents from shooting the enemy either. As a result, the final score will always stay around 0. We consider such a situation as a local minimum, because it reached a stable yet sub-optimal situation. One potential solution 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 in Env v1.0, in which agents were able to achieve closely full scores, and then, observed the agents' behavior. We found the agent to be remarkably intelligent. More specifically, the agen were not only able to follow the enemies but also able to predict their movement and fire bullets ahead of time. In addition, the agent could adjust their movement when bullets were close. What's more, the agent preferred to fire bullets consecutively to maximize the chance of hitting the enemy (Figure 12.b). Lastly, the agent actively moved about the center part of the window, rather than staying at the corner.

We also loaded the models of DRQN in Env v1.0 and observed a similar behavior as DQN. However, the main difference is that DRQN ran noticeably more smoothly than DQN agent did. This is because while DQN took in four stacked frames as inputs, DRQN only took only one frame at each time step and could therefore process images at a higher rate. In addition, DRQN also showed its superior resilience to the frame-skipping issue. These results imply that under certain hardware conditions, such as resource-limited edge devices, we may want to switch to DRQN agents with less resource requirement and more robustness in low-quality data set.

In Env v2.0, we loaded the trained DQN model and observed its behavior. We observed that even though the consecutive shooting was disable, the agent was still capable of adjusting its position and shooting at appropriate times. We observed that when the enemies moved along the same direction, the agent could predict their movements and fire ahead of their arrival. The corresponding heap map corroborates our observation because the agent focused more on spaceships, less on bullets, and not at all on empty objects. However, when the enemies moved in opposition directions or went farther apart, the agent became a little disoriented. We think that this disorientation happened because the action values of chasing either enemy are similar and weaken the agent's decisiveness.

In Env v3.0, DQN-vector's superior performance over

vanilla DQN is expected, because the input vector can more accurately and reliably capture all relevant information of a state than CNN does. The pre-trained DQN showed a much faster convergence speed at the initial stage of training, which was expected because it had better-initialized model weights. What is not expected is that the score quickly stops increasing at some point; we think it is because of the low exploration rate that caused the agent to form a fixed pattern and settle in the local minimum point.

The YOLO training curve shows strong evidence of training. As Figure 13 shows, the final model is extremely good at detecting treasures but poor at detecting bullets. We think it is due to (1) the inherent difficulty of YOLO to detect ultra-small objects. We could add finer girding in YOLO to better capture ultra-small objects (2) the insufficient training data. While mainstream data sets, we only have 300 images as training data. A larger training data set may be beneficial.

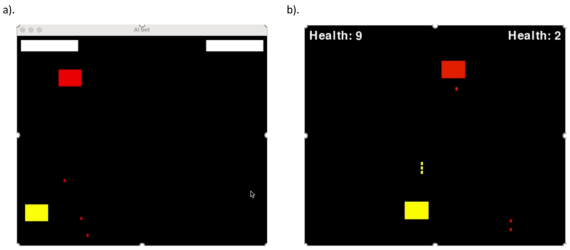

Figure 12: a). The screenshot of a GA agent playing the game. b). The screenshot of a DQN agent playing the game

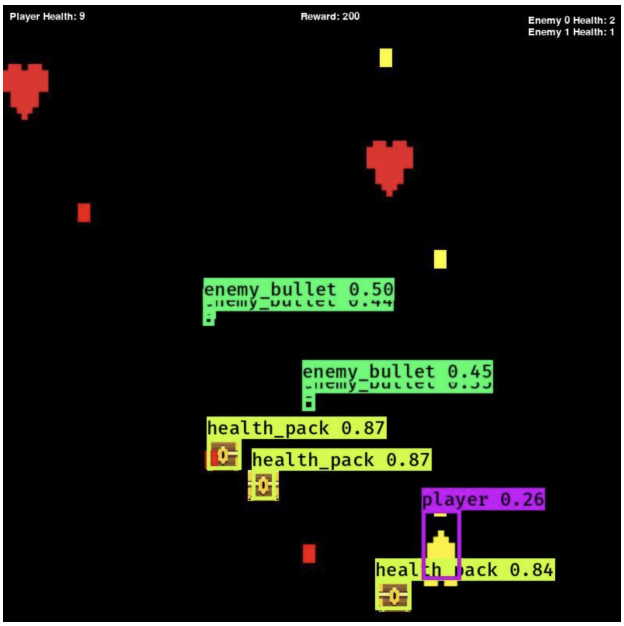

Figure 13: Trained YOLO v3 model labels an image

VII. Conclusion

Overhead-shooting games were chosen as the environment for the application of machine learning. We built different versions of the game (Env v1.0, Env v2.0, Env v3.0) with increasing complexity. The backbone of the game is finished, and it can be easily extended to include more features in future.

We explored several mainstream methods of machine learning to train our agents. GA, DQN, DDQN, PPO, YOLO and other reinforcement learning methods have been tested on different versions of our game. We found that DQN and DRQN have better performance than other models in our environment, while DRQN shows a much better resilience against the frame-skipping issue than DQN does. We also found that feeding the agent with properly processed vector of state information achieves better performance than feeding only pixel inputs. We observed both benefit and harm of using pre-training. Finally, we trained a YOLO object detector that directly extracts state information from images.

VIII. Future Work

Our future work includes:

1) Add more interesting features to the game.
2) Replace pre-scripted enemies with AI enemies.
3) Apply imitation learning in the more difficult environment.
4) Add multiple AI agents to train collaborations.

References

[1] Gabriel V Cruz Jr, Yunshu Du, and Matthew E Taylor. Pre-training neural networks with human demonstrations for deep reinforcement learning. *arXiv preprint arXiv:1709.04083*, 2017.

[2] George Eason, Benjamin Noble, and Ian Naismith Sneddon. On certain integrals of lipschitz-hankel type involving products of bessel functions. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 247(935):529–551, 1955.

[3] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[4] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*, 2015.

[5] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[8] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[9] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[10] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[11] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[12] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.