

Engineering Design Document

AI Bot for Self-designed Overhead Shooter Game

Haoqin Deng

haoqinde@usc.edu

Haoyun Zhu

haoyunzhu@usc.edu

Haoming Hu

haomingh@usc.edu

Lihan Zhu

lihanzhu@usc.edu

TABLE OF CONTENTS

TABLE OF CONTENTS	2
1 INTRODUCTION	4
1.1 Overview of Game	4
1.2 Goals	4
1.3 Related Research	4
2 DESIGN OVERVIEW	5
2.1 System Architecture	5
3 GAME ENVIRONMENT DESIGN	6
3.1 Choosing the Right Tool	6
3.2 How PyGame + OpenAI Gym Works?	6
3.3 Building Up the Environment	8
3.3.1 PyGame Environment Setup	8
3.3.2 OpenAI Gym Environment Setup	10
3.3.3 Utilities	11
3.3.4 Parameters Setup	12
4 MODEL DESIGN	13
4.1 Genetic Algorithm	13
4.1.1 Implementation	13
4.2 DQN	15
4.2.1 Pseudocode	16
4.2.2 Implementation	16
4.3 DDQN	17
4.3.1 Implementation	17
4.4 Dueling DDQN	17
4.4.1 Implementation	18
4.5 DRQN	18
4.5.1 Implementation	18
4.6 Actor-Critic	18
4.6.1 Pseudocode	19
4.6.2 Implementation	19
4.7 PPO	20
4.7.1 Implementation	20
5 EVALUATION	22
6 TIMELINE	23
Phase 0 (Week 4)	23

Phase 1 (Week 5 - 6)	23
Phase 2 (Week 7 - 10)	23
Phase 3 (Week 11 - 14)	23
7 REFERENCES	24

1 INTRODUCTION

1.1 Overview of Game

The game is a 2D overhead shooter game similar to those old-school Atari games. Players can control their characters to fight against AI enemies. To win the game, players must eliminate all enemies and stay alive. The idea of this game came from the combination of MOBA and old-school video games. We would like the game to be simple enough for initial training and we planned to add more complexity throughout the semester.

1.2 Goals

The main goals of this project are to:

- 1) Construct the environment
- 2) Train our agent to play against an enemy or multiple enemies.
- 3) Compare the performance of different models and observe their behaviors

For each of our models, we will train the agent by playing against a pre-scripted enemy. Using different models, we hope to find their strengths and weaknesses. Along with training the agents, we expect to learn new strategies from our RL agents.

1.3 Related Research

Research was conducted to find out what environments and models were suitable for our project. We decided to design our own environment and apply as many models as possible.

Regarding the environment, we narrowed it down to an overhead shooter game. We were inspired by OpenAI's algorithms trained on Atari games. The overhead shooter was similar to those games but different in terms of the gameplay. We expected to discover new findings using such an environment.

We planned to try out as many different models as possible as it was part of our goals. From our research, we learned that many models were designed to play games of different kinds. The performance of different models was something we would like to know more about.

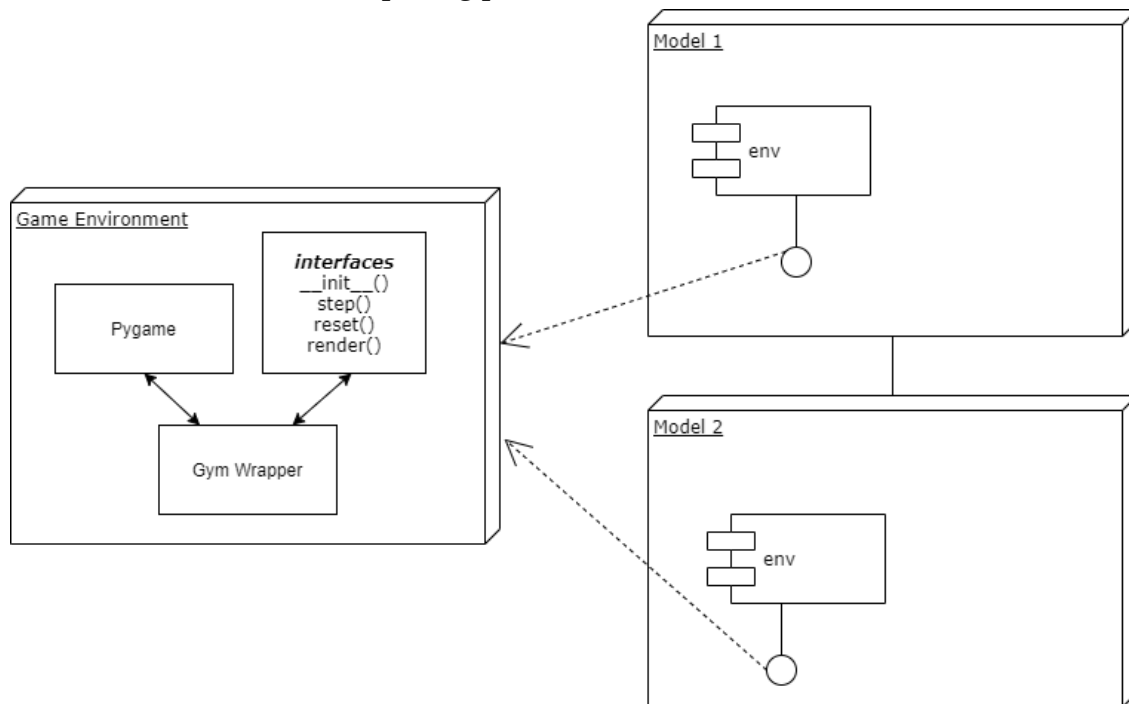
2 DESIGN OVERVIEW

Since all of the models require a standardized game environment, the project can be splitted into two components:

1. The game environment: it provides an interface for different models to use
2. The models: they utilize the game environment to perform training

2.1 System Architecture

The system architecture is shown in the below diagram. Each model will have a game environment instance and interact with it. This way, it is guaranteed that there is a standard environment for comparing performances.



Workflow of the system

3 GAME ENVIRONMENT DESIGN

3.1 Choosing the Right Tool

Prior to the project, multiple researches concerning different game development environments were conducted. We had come down to 2 main options and their features as following:

- 1) Pygame + Open AI Gym open code resources:
 - a) Direct, customizable interaction with Python. ✓
 - b) Similar syntax with SDL, easy to learn/use. ✓
 - c) Lightweight, small GPU usage. ✓
 - d) Basic game development tools with few integrated functions. ×
 - e) Less aesthetic graphic interface. ×
- 2) Unity + ML-agents:
 - a) Highly integrated game development tools. ✓
 - b) Delicate graphics quality. ✓
 - c) Indirect, less customizable interaction with Python. ×
 - d) Incompatible API with PyTorch baseline code for training. ×
 - e) Steep learning curve. ×
 - f) Taking up a lot of GPU memory, hindering training speed. ×

As the main purpose of this project being application of machine learning algorithms in game AIs, instead of making a relatively nice-looking facet, we prioritize the “intelligence” part of the game, digging into multiple algorithms and having them tested and analyzed in various environments. As a result, we decided not to put much effort into the actual gaming features, but to focus on the machine learning functionalities in which the combination of Pygame and Open AI Gym would give us more freedom in design.

3.2 How PyGame + OpenAI Gym Works?

The game will be built in Pygame, an open-source Python library for game development. In terms of the coding logic, the most important and principal functionality of Pygame is “Sprite” which essentially is the combination of a “game object” and an “image sprite”.

As shown in following diagram, a “pygame.sprite.Sprite” contains:

- 1) An image representing the object.
- 2) A “rect” variable storing the rectangular coordinates of the object. This is also where position information is stored.
- 3) All other customized parameters.

- 4) Function to display the game canvas.
- 5) Function to detect collisions between rectangular boxes of itself and other objects’.

Basic “sprite” Diagram:

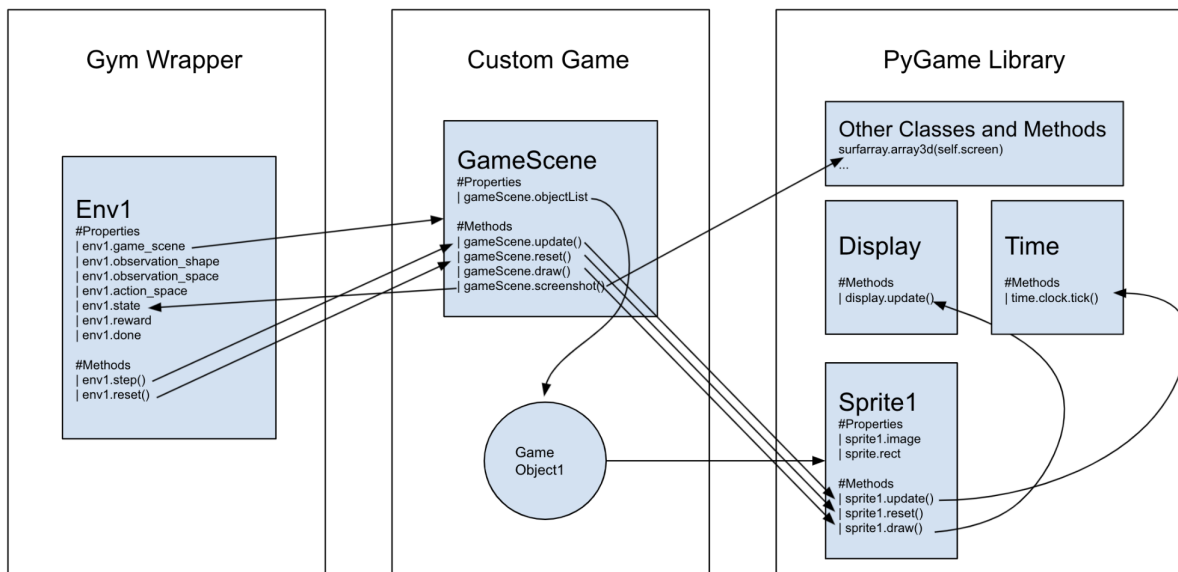
```
pygame.sprite
| self.image: pygame.image
| self.rect: pygame.rect
| self.foo: customizable data type
| self.draw() -> None
    #display on canvas
| self.collide_rect(sprite1: pygame.sprite, sprite2: pygame.sprite) -> bool
    #can be called as a static function
```

Open AI Gym is currently a prevalent API standard for implementing machine learning algorithms in a Pygame project. Simply by wrapping a pygame project inside a black box, developers are provided access to tons of open-source machine learning codebase. As long as the environment is configured accordingly, codes from variant resources can be properly integrated and run without a flaw.

Basic Gym “Env” Diagram:

```
gym.env
| self.game_scene: customized data type
    # a reference to the game project if needed
| self.observation_shape: Tuple
    # defines the size of an observation
| self.observation_space: gym.spaces.Box
    # defines a rudimental space of an observation
| self.action_space: gym.spaces.Discrete
    # defines all actions that can be performed
| self.state: customized data type
    # stores current state
| self.reward: customized data type
    # stores a customized evaluation score of current state
| self.done: bool
    # record status of the game
| self.info: dict
    # stores any additional information if needed
| step() -> None
    # takes an action and modifies self.state, self.reward,
    #   self.done and self.info
| render() -> None
    # displays the game canvas
```

| reset() -> None
starts over



Basic Workflow of PyGame + Gym

3.3 Building Up the Environment

The basic setup of our game will be two players on the screen: each player stays on their half of the screen and can fire bullets in the opposite direction. Each of them can only move left or right. The maximum number of bullets each player can fire is 3. And the starting health for both players is 10. These parameters are tunable and will be detailed in the “Parameters” section.

3.3.1 PyGame Environment Setup

The game scene is a human-playable Pygame class to be used by the environment wrapper.

GameScene

class GameScene:

Properties:

- self.screen: Pygame screen
- self.clock: Pygame clock to control the FPS
- self.player: Customized game objects as below classes

Methods:

- `__init__(self):`
 - Initializes Pygame display

- Initializes player and enemy
 - self.Reset()
- Play()
 - Take an action and get feedback from GameScene
- ScreenShot()
 - Get a vectorized array of screen pixel
- update()
 - Update the GameScene according to current action
- draw()
 - Display the canvas

Spaceship

class Spaceship(pygame.sprite.Sprite):

Properties:

- self.health: current health
- self.start_health: starting health
- self.start_x: starting x position
- self.start_y: starting y position
- self.color: color of this spaceship (used to fill the image)
- self.up_direction: a boolean whether the spaceship is facing upwards
- self.bullets: a pygame.sprite.Group for all its bullets

Methods:

- __init__(self, image: pygame.Surface, screen_rect: pygame.Rect, start_health: int, start_x: int, start_y: int, color: Tuple[int, int, int], up_direction: bool):
 - Initializes self.image and fill using color
 - Initializes self.rect.x and self.rect.y using x and y from arguments
 - Initializes other properties
- update(self, action: Action, others: List[pygame.sprite.Sprite]):
- fire(self):
- reset(self):
- is_dead(self) -> bool:

Bullet

class Bullet(pygame.sprite.Sprite):

Properties:

- self.vel_x: velocity in x direction
- self.vel_y: velocity in y direction
- self.screen_rect: the rect for the entire game screen

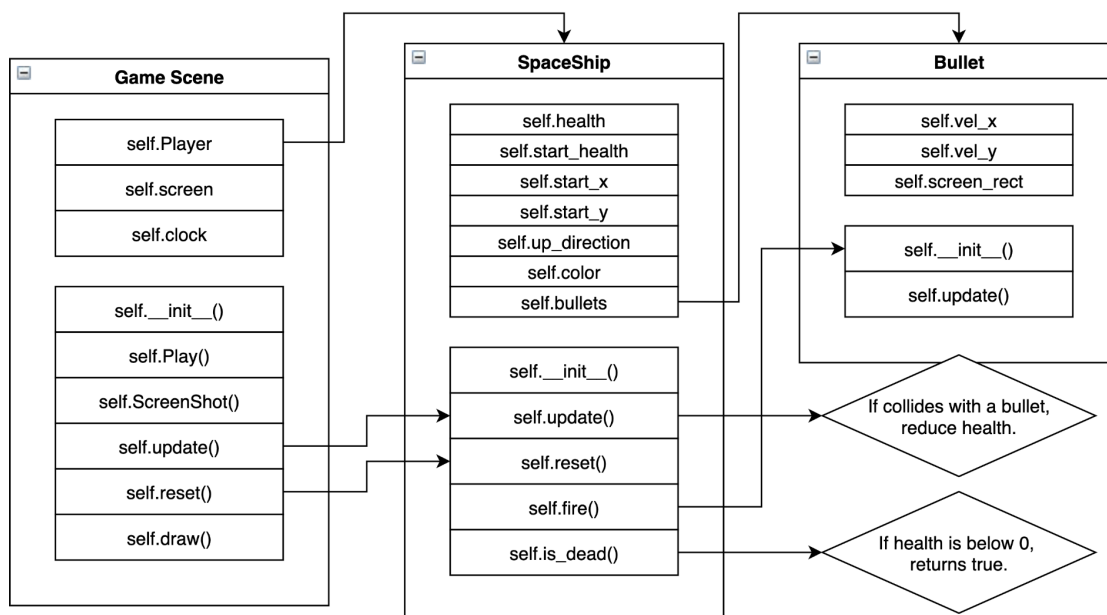
Methods:

- __init__(self, x: int, y: int, color: Tuple[int, int, int], vel_x: int, vel_y: int, screen_rect: pygame.Rect):
 - Initializes self.image and fill using color
 - Initializes self.rect.x and self.rect.y using x and y from arguments

- Initializes other properties
- `update(self)`:
 - Updates x and y position using `self.vel_x` and `self.vel_y`
 - Checks if it goes off screen:
 - If it does, kill the bullet sprite

Pre-scripted Enemy

The pre-scripted enemy is designed to facilitate the training of agents. The enemy will constantly move from left edge to right edge and move backward in a loop. It will fire randomly when possible.



Workflow of the PyGame Environment

3.3.2 OpenAI Gym Environment Setup

To use our game environment as inputs to learning agents, it is necessary to wrap the game in a `gym.Env` class.

```
class ShooterEnv(gym.Env):
```

Properties:

- `self.game_scene`: a `GameScene` class
- `self.observation_shape`: (WIDTH, HEIGHT, 3)
- `self.observation_space`: a screenshot of the game screen in RGB colors
- `self.action_space`: NOOP, LEFT, RIGHT, and FIRE
- `self.state`: a screenshot of the game screen in RGB colors

- `self.reward`: the current reward
- `self.done`: done status
- `self.info`: information

Methods:

- `__init__(self)`:
 - Initializes everything and defines observation and action space
- `step(self, action_num: int)`:
 - Performs the action
 - Returns done status, reward, and state from the game scene
- `reset(self)`:
 - Resets game scene
 - Returns state

3.3.3 Utilities

We defined some utility functions/classes that are used across different models

Methods:

- `def plot_learning_curve(x, scores, epsilons, filename, lines=None)`:
 - Plots stats using matplotlib
 - Saves the plot to a file
- `def make_env(env_name, shape=(84,84,1), repeat=4, clip_rewards=False, no_ops=0, fire_first=False)`:
 - Instantiates env using `env_name`
 - `env = RepeatActionAndMaxFrame(...)`
 - `env = PreprocessFrame(...)`
 - `env = StackFrames(...)`
 - Returns `env`

Classes:

`class RepeatActionAndMaxFrame(gym.Wrapper):`

- Wraps `reset()` and `step()` functions to repeat actions and only returns the max frame

`class PreprocessFrame(gym.ObservationWrapper):`

- Wraps `observation()` function to resize the frame to customized shapes (the most frequent one is 84 x 84)

`class StackFrames(gym.ObservationWrapper):`

- Wraps `reset()` and `observation()` functions to stack 4 frames

3.3.4 Parameters Setup

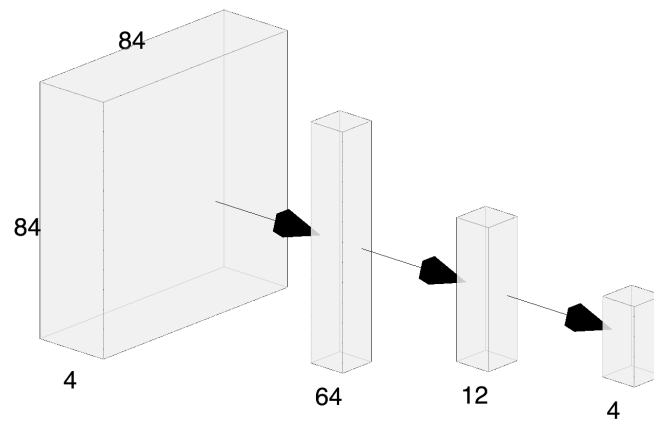
All tunable parameters are stored in one file for easy tuning. Some of the important ones are:

- PURE_COLOR_DISPLAY = True
- NEGATIVE_REWARD_ENABLED = True
- NEGATIVE_REWARD = 0.005
- REWARD.BULLET_HIT_ENEMY = 10
- REWARD.BULLET_HIT_PLAYER = -10

4 MODEL DESIGN

4.1 Genetic Algorithm

GA simulates an evolution situation where 20 separate learning agents count as one generation. Within one generation, models from different agents are combined through genetic processes such as mutation, crossover and inversion to form the next generation. In current progress, RNN is used as a single learning agent.



Structure of RNN used in GA

4.1.1 Implementation

Individual

Individual is a class for each “creature” or learning agent. A customized RNN is used to play the game and some genetic processes are included to perform evolution between parents and offspring.

```
class Individual
# Properties:
| self.nn: NeuralNetwork
    # stores a customized RNN class object
| self.fitness: float
    # stores the score of the RNN, in GA's terminology, fitness
| self.weight_biases: np.array
```

```

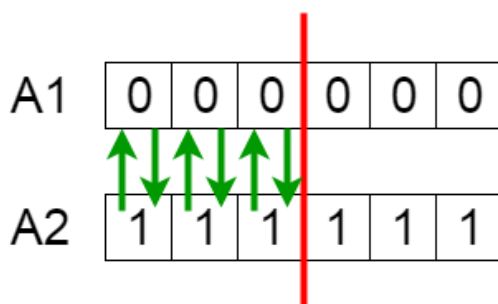
# Methods:
| self.__init__(input_size: int, hidden_size: int, output_size: int) -> None
    # initialization of a single learning agent
| self.calculate_fitness(env: gym.env) -> None
    # run a single episode of a game and get the feedback information
| self.update_model() -> None
    # update RNN mode weights with current result and biases

# Abstract methods for future definition
| self.get_model(input_size: int, hidden_size: int, output_size: int) ->
NeuralNetwork
    # method of getting a learning agent
| self.run_single(env: gym.env, episode: int, render: bool) -> Tuple[float, np.array]
    #method for a single episode game simulation

# Static Functions:
| crossover(parent1_weights_biases: np.array, parent2_weights_biases: np.array, p: float)
-> Tuple[np.array, np.array]
| inversion(child_weights_biases: np.array) -> np.array
| mutation(parent_weights_biases: np.array, p: float, scale: int) -> np.array
| ranking_selection(population: List[Individual]) -> Tuple[Individual, Individual]
| roulette_wheel_selection(population: List[Individual]) -> Individual
| statistics(population: List[Individual]) -> float, float, float

```

Genetic Crossover of a Vector



Genetic Mutation of a Vector

Before Mutation

A5 [1, 1, 1, 0, 0, 0]

After Mutation

A5 [1, 1, 0, 1, 1, 0]

Population

Population is a class for the “entire world” including 500 generations. It’s a generalization class for the GA model.

```
class Population
```

```

# Properties:
| self.pop_size: int
| self.max_generation: int
| self.p_mutation: float
    # probability of mutation
| self.p_crossover: float
    # probability of crossover
| self.p_inversion: float
    # probability of inversion

# Methods:
| self.__init__() -> None
    # initialization of a generation
| self.run()-> None
    # Run GA simulation until max generation is achieved.
    # Inside each generation, 20 individuals are called one by one to start a
    single game episode.

```

Basic Parameter

- RNN:
 - Input size: $84 * 84 * 3$
 - Hidden layer1 size: 40
 - Hidden layer2 size: 12
 - Output size: 4
- GA:
 - Max iteration in one game: 2000
 - Population size: 20
 - Max generation size: 500
 - Mutation rate: $0.2 \sim 0.1$
 - Crossover rate: $0.7 \sim 0.8$
 - Inversion rate: 0

Further modification and tests are needed to find out the best and most quickly-converging settings of the parameters.

4.2 DQN

DQN evolves from Q learning. Q learning keeps a Q table to evaluate the desirability of each state/action pair. The Q table is updated using the temporal different method. However, it is unscalable in large games because the number of states is too large, and the Q table will be intractable. So, we replace the Q table with Q network for generalization. Q network also updates itself by minimizing the loss function within the framework of temporal difference (TD) algorithm.

4.2.1 Pseudocode

Algorithm 1 Deep Q-learning with Experience Replay

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
  Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
  for  $t = 1, T$  do
    With probability  $\epsilon$  select a random action  $a_t$ 
    otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
    Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
    Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
    Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
    Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
    Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
    Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
  end for
end for
  
```

4.2.2 Implementation

class DeepQNetwork(nn.Module):

Properties:

- self.conv1, self.conv2, self.conv3: convolutional layers
- self.fc1, self.fc2: fully connected layers
- self.optimizer: RMSprop optimizer
- self.loss: MSE loss

Methods:

- forward(self, state):
 - Forwards the input state through each layer of the network
 - Returns action

class DQNAgent(object):

Properties:

- self.q_eval: a DeepQNetwork
- self.q_next: a DeepQNetwork
- self.memory: a replay buffer
- self.epsilon

Methods:

- choose_action(self, observation):
 - If np.random.random() > self.epsilon:
 - Chooses action using q_eval
 - else:
 - Chooses random action

- Return action
- store_transition(self, state, action, reward, state_, done):
 - Stores transition in self.memory
- sample_memory(self):
 - Samples memory from self.memory
 - Return states, actions, rewards, states_, dones
- replace_target_network(self):
 - Copies eval network to target_network for every 1000 steps
- decrement_epsilon(self):
 - Decreases self.epsilon by fixed amount

Main:

- agent = DQNAgent(...)
- env = make_env(...)
- for i in range(n_games):
 - done = False
 - observation = env.reset()
 - score = 0
 - while not done:
 - action = agent.choose_action(observation)
 - observation_, reward, done, info = env.step(action)
 - agent.store_transition(observation, action, reward, observation_, done)
 - agent.learn()
 - observation = observation_
 - Print out stats

4.3 DDQN

DQN uses the same Q network for both evaluation and selection, Such estimation creates a maximum bias. Double DQN alleviates the problem by introducing a separate Q' prime network solely for action selection in the max operator. The weights of Q' are periodically copied from Q.

4.3.1 Implementation

The only difference from DQN:

- Second Q' network

4.4 Dueling DDQN

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.

4.4.1 Implementation

The only difference from DQN:

- Uses an advantage function that subtracts the state value $V(s)$ from the action value $Q(s, a)$ as the output of the q_eval and q_next networks

4.5 DRQN

The original DQN learning captures temporal information by having 4 consecutive frames as inputs. DRQN has only one frame as input but can retain past information by using an LSTM layer to replace the linear layer in the original DQN. And we are sampling a sub-sequence from an episode.

4.5.1 Implementation

The only difference from DQN:

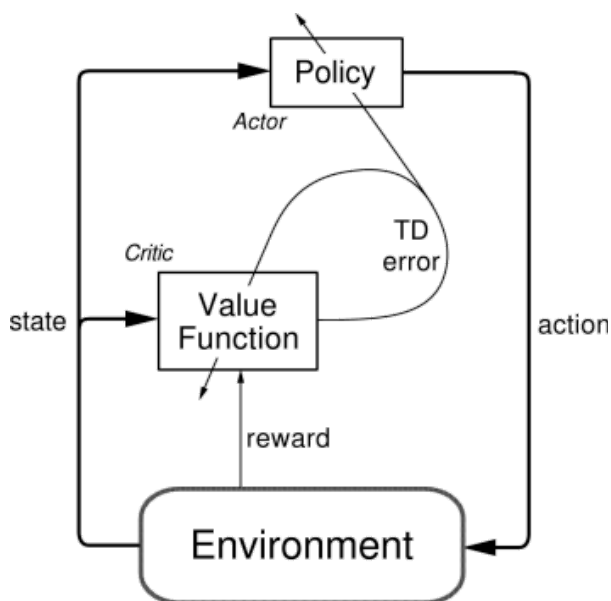
- One of the fully connected layer is replaced by an LSTM layer

4.6 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 convolutional 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).



Actor-critic workflow

4.6.1 Pseudocode

In each training step:

1. Observe the state
2. Randomly sample action a_t according to $\pi(\cdot | s_t; \theta_t)$
3. Perform a_t and observe new state s_{t+1} and reward r_t
4. Update ω (in value network) using temporal difference (TD)
5. Update θ (in policy network) using policy gradient

4.6.2 Implementation

```
class ActorCriticNetwork(nn.Module):
```

Properties:

- A neural network with two output layers:
 - self.pi: output for actor network
 - self.v: output for critic network

Methods:

- forward(self, state):
 - Given a game state, return pi and v

```
class ActorCriticAgent:
```

Properties:

- self.actor_critic: an ActorCriticNetwork
- self.gamma
- self.lr: learning rate

Methods:

- `choose_action(self, observation):`
 - Forwards `self.actor_critic` network to get action probabilities
 - Randomly samples an action and return it
- `learn(self, state, reward, state_, done):`
 - Forwards `self.actor_critic` network to get critic value
 - Calculates actor and critic losses
 - `(actor_loss + critic_loss).backward()`
 - Steps optimizer

Main:

- `agent = ActorCriticAgent(...)`
- `env = make_env(...)`
- `for i in range(n_games):`
 - `done = False`
 - `observation = env.reset()`
 - `score = 0`
 - `while not done:`
 - `action = agent.choose_action(observation)`
 - `observation_, reward, done, info = env.step(action)`
 - `score += reward`
 - `agent.learn(observation, reward, observation_, done)`
 - `observation = observation_`

4.7 PPO

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

PPO also uses actor-critic agents, but the difference is that it combines the idea of TRPO. It uses clipped surrogate objectives to penalize large policy updates.

4.7.1 Implementation

`class Policy(nn.Module):`

Properties:

- `self.base`: CNN base
- `self.dist`: distribution of size `num_outputs` to decide action

Methods:

- `act(self, inputs):`
 - Gets value and `actor_feature` from CNN
 - Returns return value, action, `action_log_probs`
- `get_value(self, inputs):`
 - Returns value from CNN
- `evaluate_actions(self, inputs, action):`

- Returns value, action_log_probs from CNN

class A2C_ACKTR:

Properties:

- self.actor_critic: a Policy network
- self.optimizer: optimizer

Methods:

- update(self, rollouts):
 - self.actor_critic.evaluate_actions(rollout buffer data)
 - Calculates value and policy loss
 - loss.backward()
 - self.optimizer.step()

Main:

- actor_critic = Policy(...)
- agent = A2C_ACKTR(actor_critic, ...)
- env = make_env(...)
- for j in range(num_updates):
 - for step in range(args.num_steps):
 - obs, reward, done, infos = envs.step(action)
 - Puts input to rollout buffer
 - next_value = actor_critic.get_value(...)
 - Determines reward based on action
 - agent.update(rollouts)

5 EVALUATION

1. Scores: we evaluate the model by summation of agent's reward scores.

$$scores = \sum_{i=1}^n reward_{T=i} \text{ (When } T = n, \text{ Game ends up)}$$

$$- 100 \leq scores \leq 100$$

2. The convergence speed of training the model.

6 TIMELINE

Phase 0 (Week 4)

- Construct game environment using PyGame

Phase 1 (Week 5 - 6)

- Train 1 AI player against 1 pre-scripted enemy
- DQN

Phase 2 (Week 7 - 10)

- Train 1 AI player against multiple pre-scripted enemies
- DQN + LSTM + Actor-critic + PPO

Phase 3 (Week 11 - 14)

- Train multiple AI player against multiple pre-scripted enemies
- Extend from 1 player model to include teammates' observation space

7 REFERENCES

Papers:

D. H. Ackley, "A Connectionist Machine for Genetic Hillclimbing," Kluwer Academic Publishers, Dordrecht, 1987

Y. Davidor, "A naturally occurring niche and species phenomenon: the model and first results. In Proceedings of the Fourth International Conference on Genetic Algorithms" Morgan Kaufmann, 1991

V. Mnih et al., "Playing Atari with Deep Reinforcement Learning", arXiv.org, 2013. [Online]. Available: <https://arxiv.org/abs/1312.5602>. [Accessed: 20- Oct- 2021].

M. Hausknecht and P. Stone, "Deep Recurrent Q-Learning for Partially Observable MDPs", arXiv.org, 2015. [Online]. Available: <https://arxiv.org/abs/1507.06527>. [Accessed: 20- Oct- 2021].

H. van Hasselt, A. Guez and D. Silver, "Deep Reinforcement Learning with Double Q-learning", arXiv.org, 2015. [Online]. Available: <https://arxiv.org/abs/1509.06461>. [Accessed: 20- Oct- 2021].

Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot and N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning", arXiv.org, 2015. [Online]. Available: <https://arxiv.org/abs/1511.06581>. [Accessed: 20- Oct- 2021].

Konda, Vijay R., and John N. Tsitsiklis. "Actor-critic algorithms." Advances in neural information processing systems. 2000.

Schulman, John, et al. "Proximal policy optimization algorithms." arXiv preprint arXiv:1707.06347 2017.

Bellemare M G, Naddaf Y, Veness J, et al. The arcade learning environment: An evaluation platform for general agents[J]. Journal of Artificial Intelligence Research, 2013, 47: 253-279.

Websites:

PyGame Documentation: <https://www.pygame.org/docs/>

OpenAI Gym Genetic Algorithm Tutorial:
<https://becominghuman.ai/genetic-algorithm-for-reinforcement-learning-a38a5612c4dc>

PPO: <https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail>

<https://inst.eecs.berkeley.edu/~cs188/sp20/assets/files/SuttonBartoIPRLBook2ndEd.pdf>