RL－Final Project Proposal

Super Agent: Conquering Super Mario with 3 RL strategies.

Teammates:

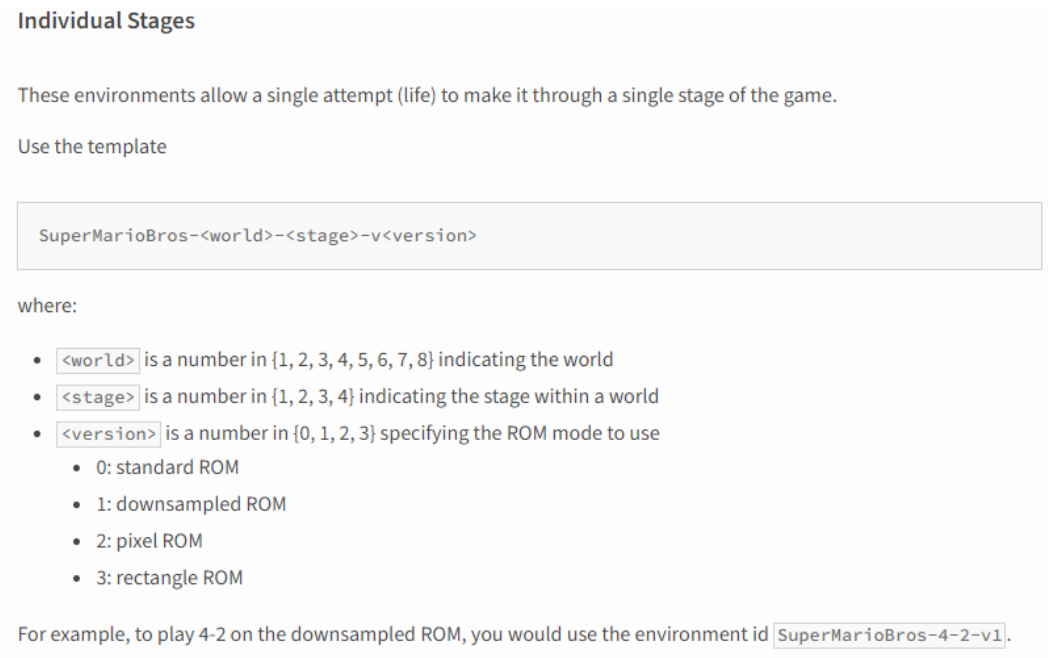
7111064803 張昆湧 7111064109 林軒宇 4108064040 鄭宇辰 4108064005 盧弘毅

Target:

We are going to use the super-mario-bros environment which provided by gym and try some RL strategies to train our agents. Aims to make our agents stronger to get higher scores with less game time. As gym help us covers lots of tedious work of building the envs, we hope to try around 3 different strategies to see how our agents behave in the same envs and the training episodes.

Super-Mario-Bros Environment:

As shown below, we can choose which stage our agent learn / play for.



Individual Stages

These environments allow a single attempt (life) to make it through a single stage of the game.

Use the template

```
SuperMarioBros-<world>-<stage>-v<version>
```

where:

- `<world>` is a number in {1, 2, 3, 4, 5, 6, 7, 8} indicating the world
- `<stage>` is a number in {1, 2, 3, 4} indicating the stage within a world
- `<version>` is a number in {0, 1, 2, 3} specifying the ROM mode to use
  - 0: standard ROM
  - 1: downsampled ROM
  - 2: pixel ROM
  - 3: rectangle ROM

For example, to play 4-2 on the downsampled ROM, you would use the environment id `SuperMarioBros-4-2-v1`.

✪ Since it provides 4 kinds of version, we'll choose one of them to implement because it depends on the ability of our computer in Lab.

Randomly select stages is also available:

```
gym.make('SuperMarioBrosRandomStages-v0', stages=['1-4', '2-4', '3-4', '4-4'])
```

Reward function is defined as below:

The reward function assumes the objective of the game is to move as far right as possible (increase the agent's *x* value), as fast as possible, without dying. To model this game, three separate variables compose the reward:

1. *v*: the difference in agent *x* values between states
   - in this case this is instantaneous velocity for the given step
   - *v = x1 - x0*
     - *x0* is the x position before the step
     - *x1* is the x position after the step
   - moving right ⇔ *v > 0*
   - moving left ⇔ *v < 0*
   - not moving ⇔ *v = 0*
2. *c*: the difference in the game clock between frames
   - the penalty prevents the agent from standing still
   - *c = c0 - c1*
     - *c0* is the clock reading before the step
     - *c1* is the clock reading after the step
   - no clock tick ⇔ *c = 0*
   - clock tick ⇔ *c < 0*
3. *d*: a death penalty that penalizes the agent for dying in a state
   - this penalty encourages the agent to avoid death
   - alive ⇔ *d = 0*
   - dead ⇔ *d = -15*

r = v + c + d

The reward is clipped into the range *(-15, 15)*.

RL strategies:

- **Q-Learning (off-policy TD control)**

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

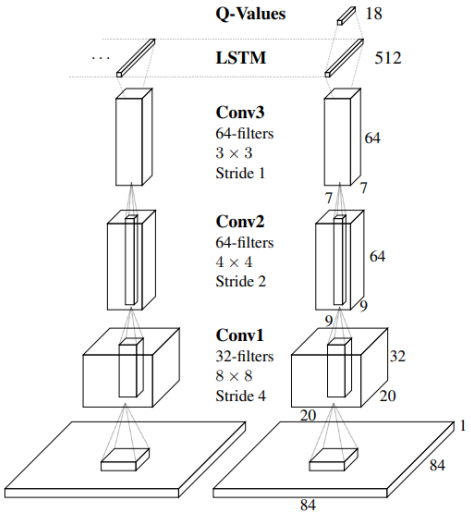
- **Deep Recurrent Q-learning / Q-Network (DRQN)**

A. Adding recurrency to a Deep Q-Network by replacing the first post-convolutional fully-connected layer with a recurrent Long Short Term Memory (LSTM).

B. Although capable of seeing only a single frame at each timestep, successfully integrates information through time and replicates DQN's performance on standard Atari games and partially observed equivalents featuring flickering game screens.

C. Given the same length of history, recurrency is a viable alternative to stacking a history of frames in the DQN's input layer and while recurrency confers no systematic advantage when learning to play the game, the recurrent net can better adapt at evaluation time if the quality of observations changes.

D. Architecture of a DRQN:



➢ Proximal Policy Optimization Algorithm (PPO)

A. It attains the data efficiency and reliable performance of Trust Region Policy Optimization (TRPO), while using only first-order optimization.

B. It proposes a novel objective with clipped probability ratios, which forms a pessimistic estimate (i.e. lower bound) of the performance of the policy.

C. To optimize policies, it alternates between sampling data from the policy and performing several epochs of optimization on the sampled data.

D. Main PPO Algorithm shown as below:

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Reference:

"Reinforcement Learning: An Introduction", 2nd Ed. R. Sutton & G. Barto, 2018 CH6

https://pypi.org/project/gym-super-mario-bros/ (Gym)

https://arxiv.org/abs/1507.06527 (Deep Recurrent Q-Learning for POMDPs)

https://arxiv.org/abs/1707.06347 (PPO)