CONQUERING

SUPER MARIO

WITH RL STRATEGIES.

- 7111064803 張昆湧 Mario/マリオ 
- 7111064109 林軒宇 Luigi/ルイージ
- 4108064040 鄭宇辰 Toad/キノピオ
- 4108064005 盧弘毅 Yoshi/ヨッシー

Proximal Policy Optimization Algorithms


THANK YOU MARIO
YOUR QUEST IS OVER
WE PRESENT YOU A NEW QUEST
PUSH BUT
TO SELECT A WORLD
THANK YOU MARIO
YOUR QUEST IS OVER
WE PRESENT YOU A NEW QUEST
PUSH BUTTON B
TO SELECT A WORLD

MARIO KART

MARIO PARTY

MARIO FURY WORLD

SUPER MARIO MAKER

Double Deep Q-Network

AGENDA

- Overview
- Environment
- Agent
- Result

OVERVIEW

- We use Double Deep Q-Networks as our main Algorithm
- Two ConvNets - Q_{online} and Q_{target} that independently approximate the optimal action-value function
- Two values - TD Estimate and TD Target
- epsilon-greedy action

AGENDA

- Overview
- Environment
- Agent
- Result

ENVIRONMENT

- env = gym_super_mario_bros.make
- env = JoypadSpace(env, [["right"], ["right", "A"]])
- env = GrayScaleObservation(env)

[240, 256, 3] → [240, 256, 1]

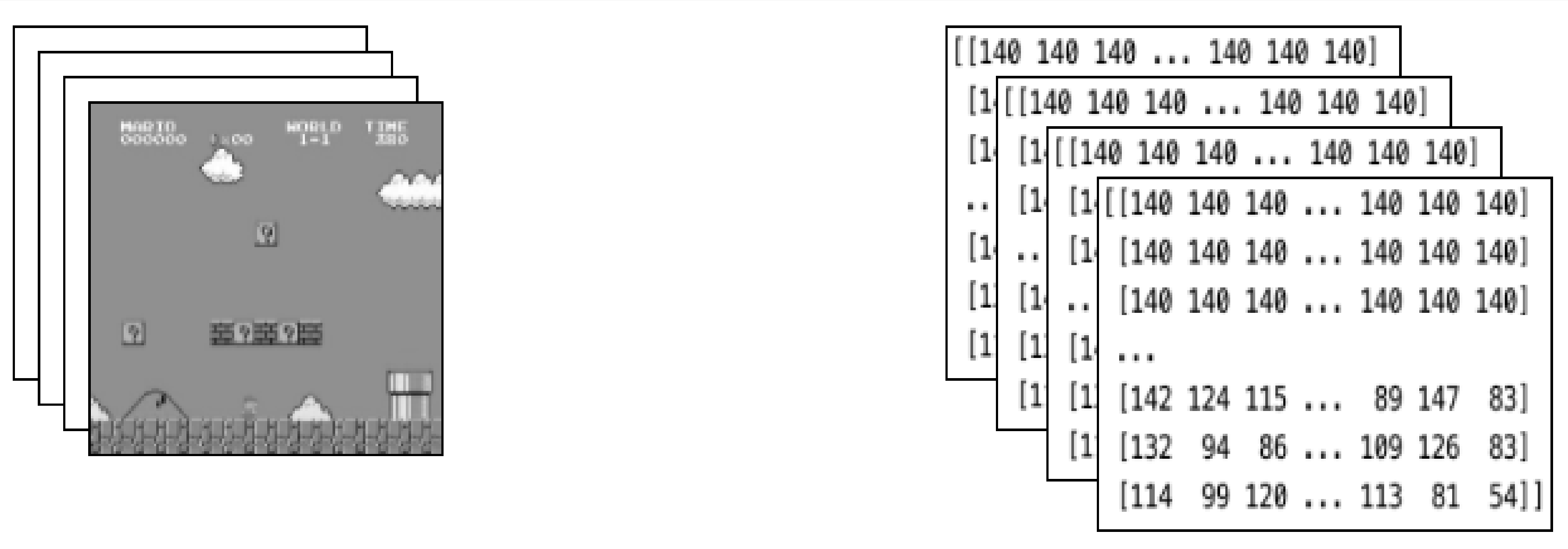
- env = ResizeObservation(env, shape=84)

[240, 256, 1] → [84, 84, 1]

- env = SkipFrame(env, skip=4)
- env = FrameStack(env, num_stack=4, new_step_api=True)

[84, 84, 1] → [4, 84, 84, 1]

ENVIRONMENT



AGENDA

- Overview
- Environment
- Agent
  - Act
  - Memory
  - Learn

1. Epsilon-greedy action

```
# EXPLORE
    if np.random.rand() < self.exploration_rate:
        action_idx = np.random.randint(self.action_dim)

# EXPLOIT
    else:
        action_values = self.net(state)
        action_idx = torch.argmax(action_values, axis=1).item()

# decrease exploration_rate
    self.exploration_rate *= self.exploration_rate_decay

# we still set a min probability for exploration
    self.exploration_rate = max(
       self.exploration_rate_min, self.exploration_rate)
```

mini CNN structure

input -> (conv2d + relu) x 3 -> flatten

-> (dense + relu) x 2 -> output

2. Memory

```
# cache(self, state, next_state, action, reward, done)

    state = torch.tensor(state, device=self.device)
    next_state = torch.tensor(next_state, device=self.device)
    action = torch.tensor([action], device=self.device)
    reward = torch.tensor([reward], device=self.device)
    done = torch.tensor([done], device=self.device)
    self.memory.append((state, next_state, action, reward, done,))

# recall

    batch = random.sample(
        self.memory, self.batch_size)  # just sample in normal distribution
    state, next_state, action, reward, done = map(torch.stack, zip(*batch))
```

3. learn

```
# Sample from memory
state, next_state, action, reward, done = self.recall()

# Get TD Estimate
td_est = self.td_estimate(state, action)

# Get TD Target
td_tgt = self.td_target(reward, next_state, done)

# Backpropagate loss through Q_online
loss = self.update_Q_online(td_est, td_tgt)
```

TD Estimate - the predicted optimal Q^* for a given state s

$$TD_e = Q^*_{online}(s, a)$$

TD Target - aggregation of current reward and the estimated Q^* in the next state s'

$$a' = argmax_a Q_{online}(s', a)$$

$$TD_t = r + \gamma Q^*_{target}(s', a')$$

As Mario samples inputs from his replay buffer, we compute TD_t and TD_e and backpropagate this loss down Q_{online} to update its parameters θ_{online} (α is the learning rate `lr` passed to the `optimizer`)

$$\theta_{online} \leftarrow \theta_{online} + \alpha \nabla (TD_e - TD_t)$$

θ_{target} does not update through backpropagation. Instead, we periodically copy θ_{online} to θ_{target}

$$\theta_{target} \leftarrow \theta_{online}$$

```
def __init__(self, state_dim, action_dim, save_dir):
    super().__init__(state_dim, action_dim, save_dir)
    self.optimizer = torch.optim.Adam(self.net.parameters(), lr=0.00025)
    self.loss_fn = torch.nn.SmoothL1Loss()

def update_Q_online(self, td_estimate, td_target):
    loss = self.loss_fn(td_estimate, td_target)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
    return loss.item()

def sync_Q_target(self):
    self.net.target.load_state_dict(self.net.online.state_dict())
```

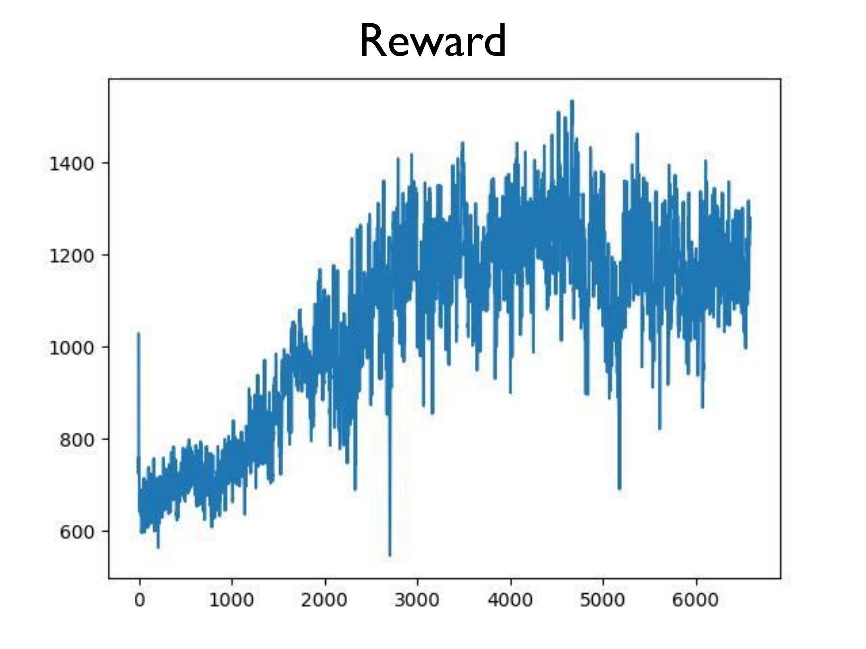
SMALL CONCLUSION

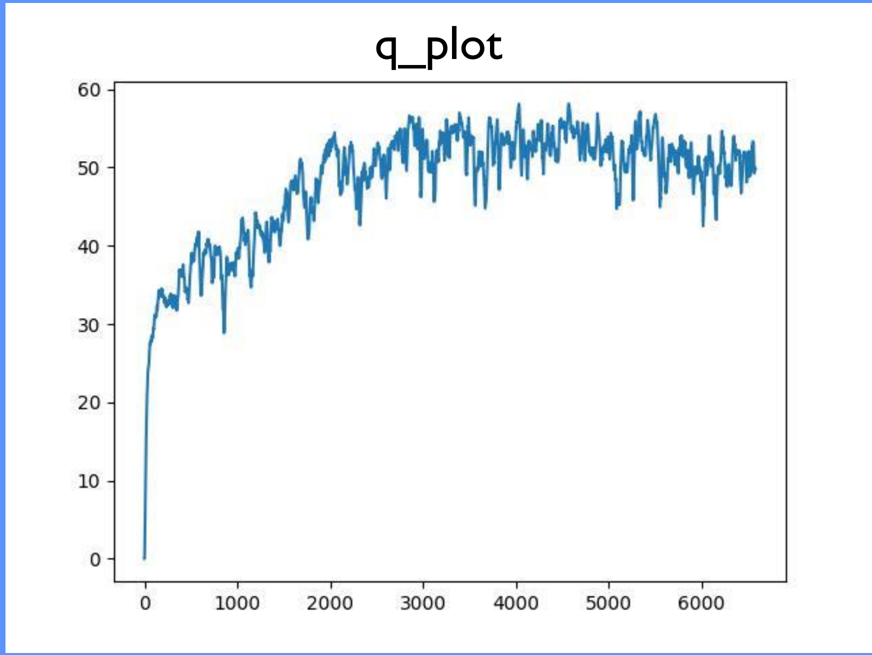
Algorithm 1 DDQN Algorithm

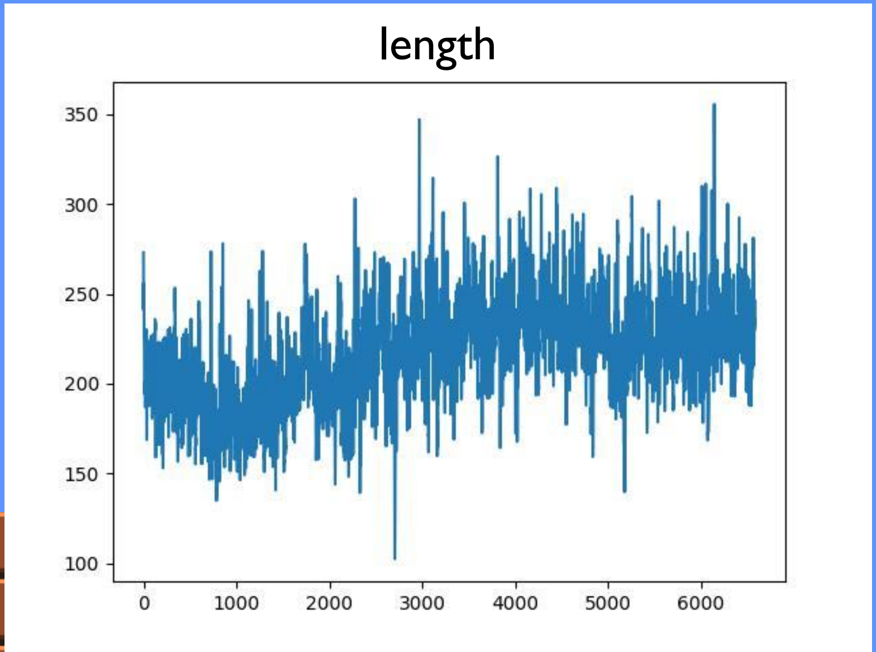
mario env Initialization
while $episode \leq episodes$ **do** ▷ episodes = 10000000
    env reset
    while True **do** ▷ Play the game
        $action \leftarrow mario.act(state)$ ▷ Run agent on the state
        $nextstate, reward, done, trunc, info \leftarrow env.step(action)$
        $mario.cache(state, nextstate, action, reward, done)$
        $q, loss \leftarrow mario.learn$ ▷ learn with DDQN
        $state \leftarrow nextstate$ ▷ update state

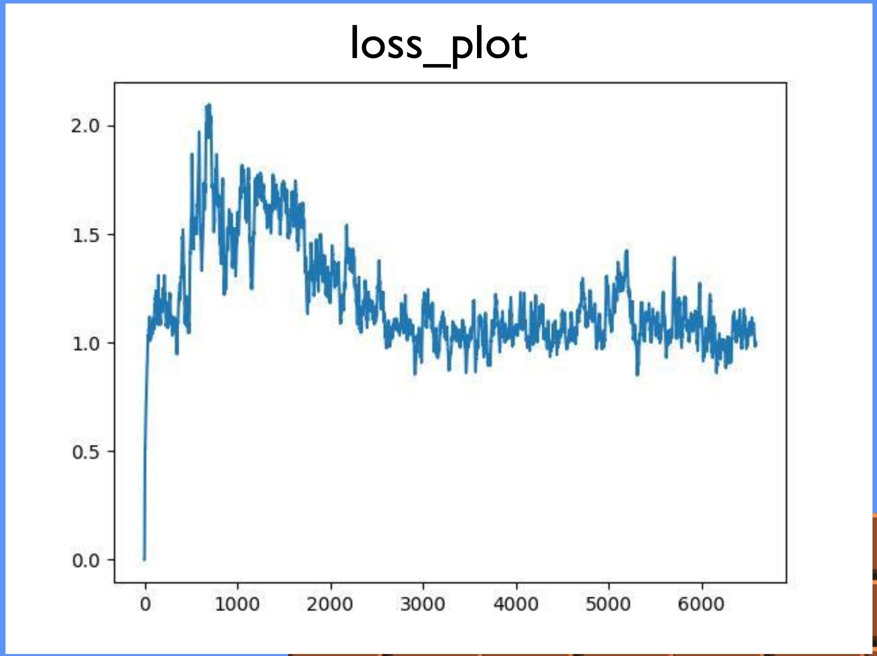
AGENDA

- Overview
- Environment
- Agent
- Result


Reward
1400
1200
1000
800
600
0 1000 2000 3000 4000 5000 6000


q_plot
60
50
40
30
20
10
0
0 1000 2000 3000 4000 5000 6000


length
350
300
250
200
150
100
0 1000 2000 3000 4000 5000 6000


loss_plot
2.0
1.5
1.0
0.5
0.0
0 1000 2000 3000 4000 5000 6000

WE DIDN'T CONQUER THE GAME

500000 steps

3500000 steps

COMING SOON