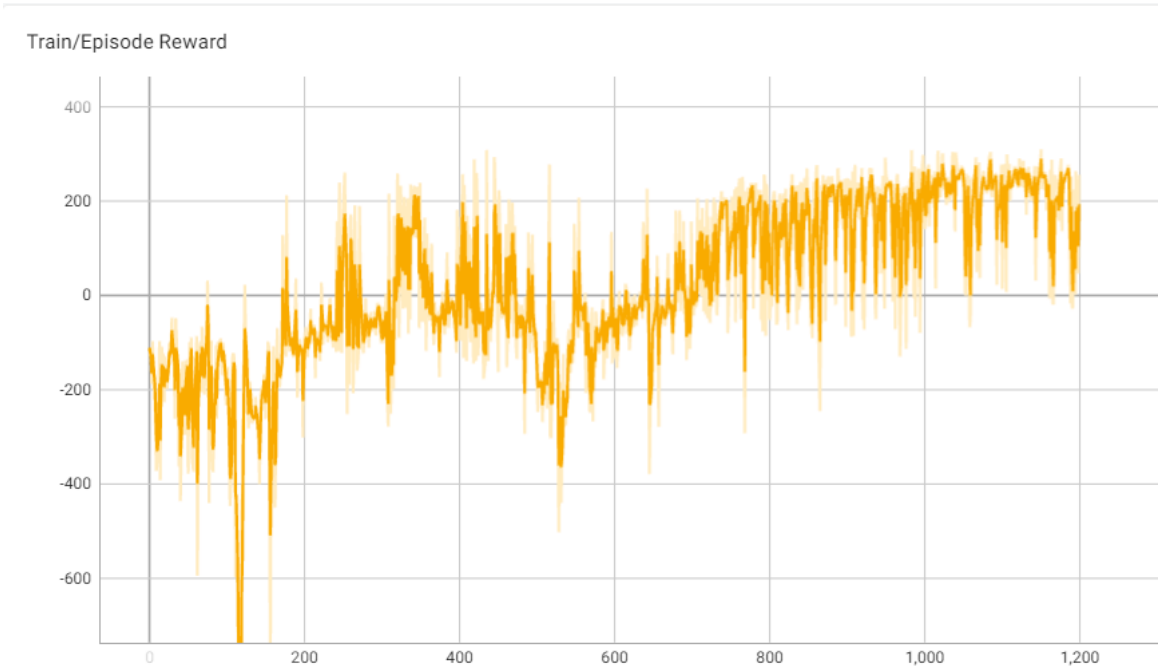
Deep Q-Network and Deep Deterministic Policy Gradient

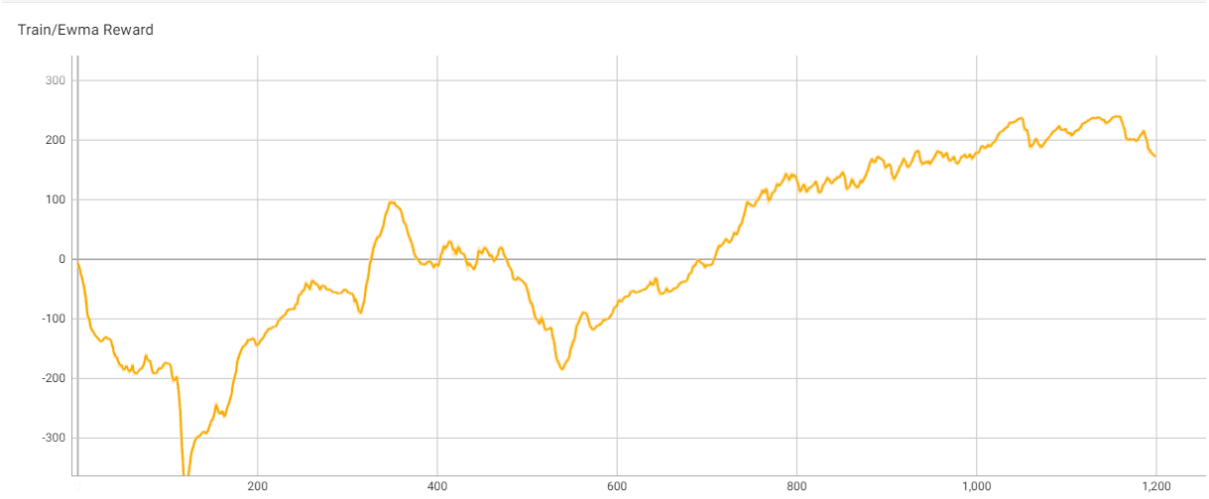
Experimental Results

DQN

Training reward



Training ewma reward

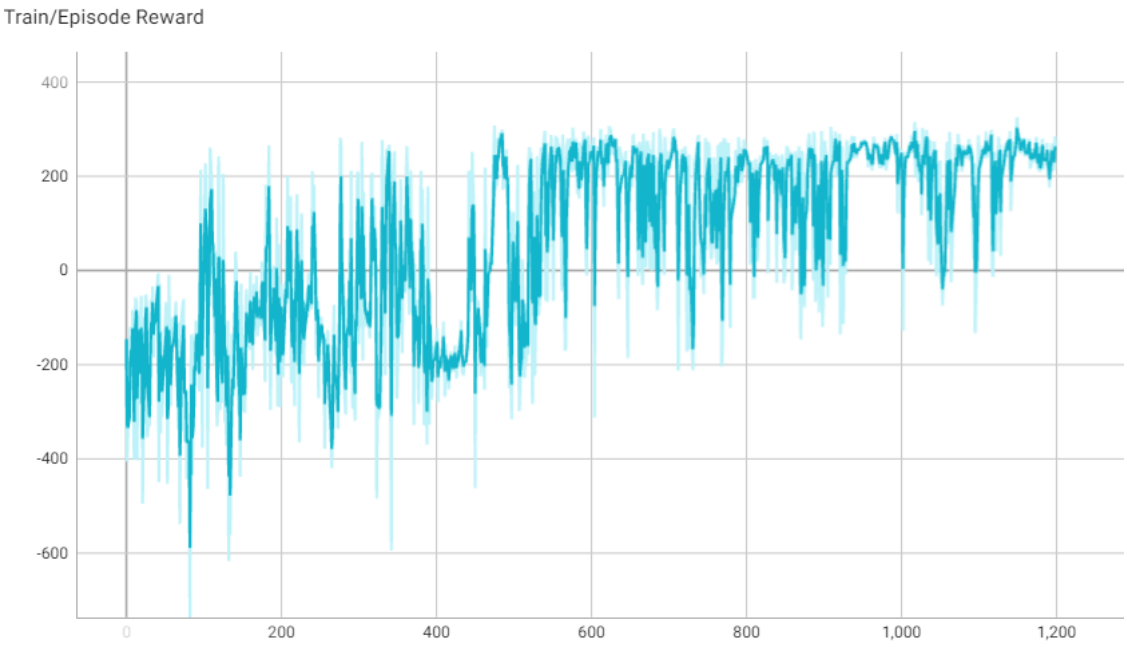


Testing result

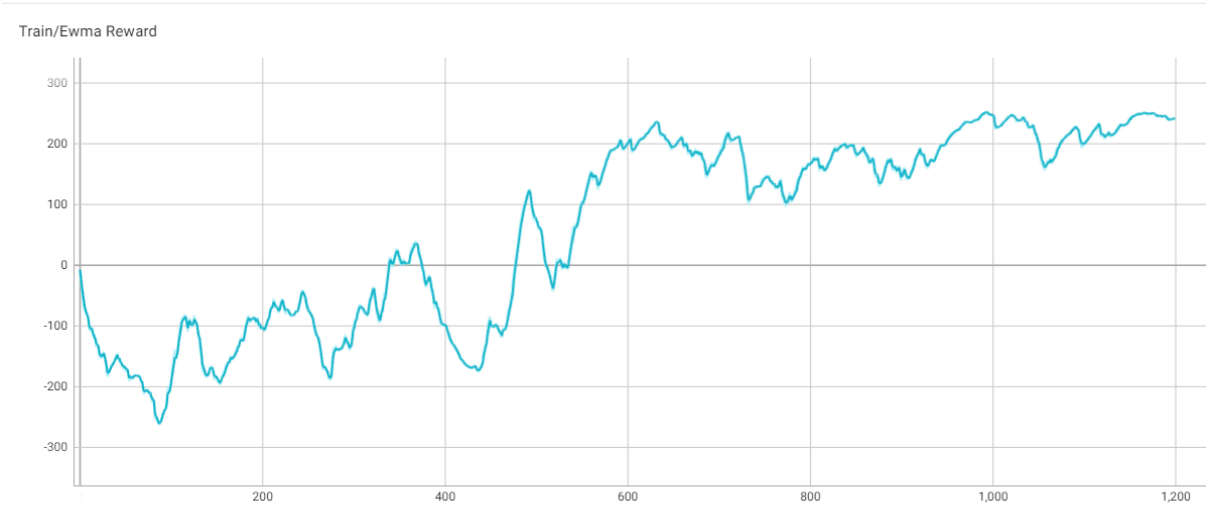
```
Start Testing
Episode: 0      Length: 252     Total Reward: 248.09
Episode: 1      Length: 257     Total Reward: 241.09
Episode: 2      Length: 273     Total Reward: 270.69
Episode: 3      Length: 281     Total Reward: 284.25
Episode: 4      Length: 315     Total Reward: 240.46
Episode: 5      Length: 278     Total Reward: 260.26
Episode: 6      Length: 899     Total Reward: 272.86
Episode: 7      Length: 205     Total Reward: 237.59
Episode: 8      Length: 362     Total Reward: 190.12
Episode: 9      Length: 278     Total Reward: 274.30
Average Reward 251.97099846611633
```

DDPG

Training reward



Training ewma reward

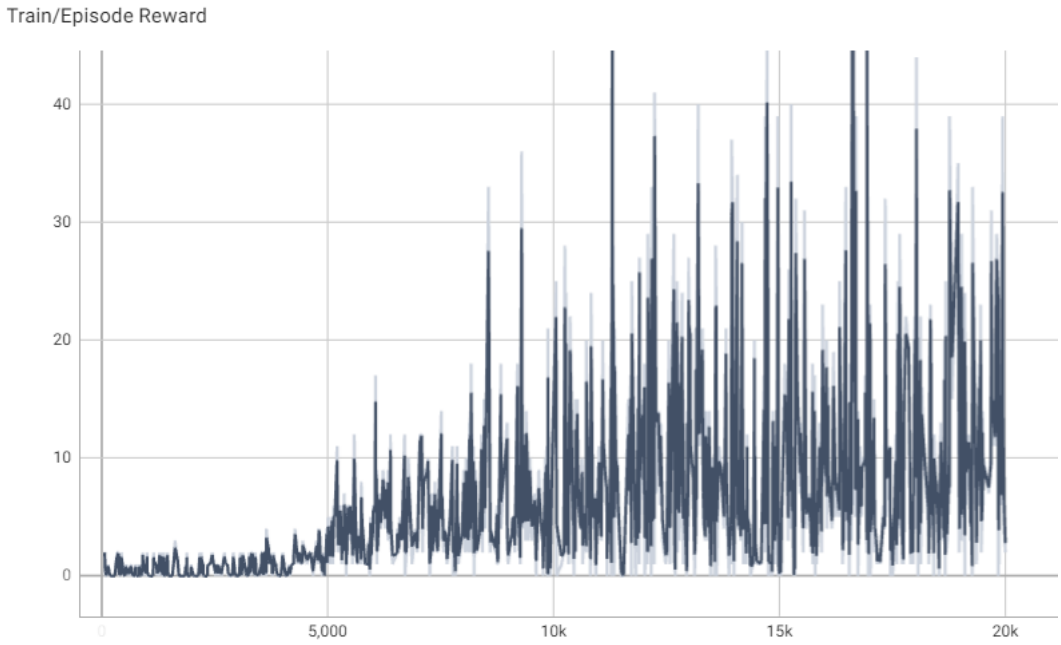


Testing result

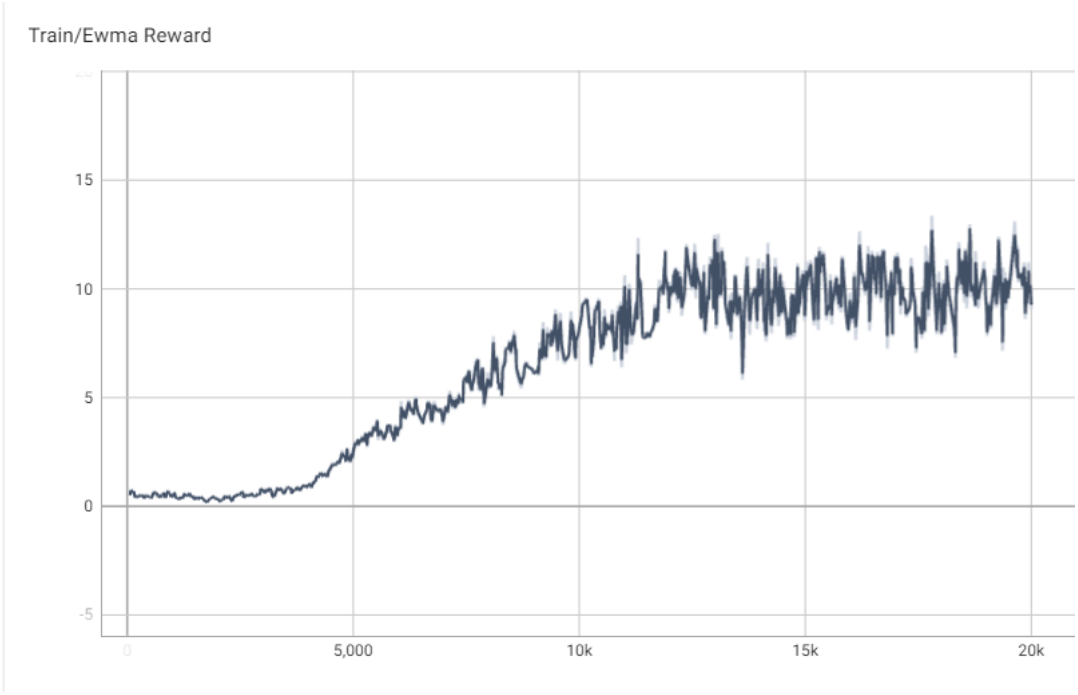
```
Start Testing
Episode: 0      Length: 200     Total reward: 240.63
Episode: 1      Length: 163     Total reward: 258.31
Episode: 2      Length: 249     Total reward: 269.89
Episode: 3      Length: 228     Total reward: 268.93
Episode: 4      Length: 330     Total reward: 239.87
Episode: 5      Length: 348     Total reward: 256.90
Episode: 6      Length: 389     Total reward: 225.57
Episode: 7      Length: 430     Total reward: 277.29
Episode: 8      Length: 218     Total reward: 258.57
Episode: 9      Length: 336     Total reward: 230.17
Average Reward 252.61280250790864
```

DQN-Breakout

Training reward



Training ewma reward



Testing result

```
Start Testing
episode 1: 282.00
episode 2: 275.00
episode 3: 380.00
episode 4: 356.00
episode 5: 283.00
episode 6: 372.00
episode 7: 293.00
episode 8: 279.00
episode 9: 313.00
episode 10: 199.00
Average Reward: 303.20
```

Questions

Implementation

DQN

```
def _update_behavior_network(self, gamma):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    ## TODO ##
    q_value = self._behavior_net(state).gather(1, action.long())
    with torch.no_grad():
       q_next = torch.max(self._target_net(next_state), 1)[0].view(-1, 1)
       q_target = reward + q_next * gamma * (1.0 - done)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
    # optimize
    self._optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()
```

random sample a minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ and let

$$targetQ = r_t + \gamma * max_{a'}\hat{Q}(\phi_{t+1}, a'; \bar{\theta}) * (1 - done)$$

then compute MSE loss between $targetQ$ and $Q(\phi_t, a_t; \theta)$ to update θ, then update $\hat{\theta} \leftarrow \theta$ every certain step.

DDPG

```
def _update_behavior_network(self, gamma):
    actor_net, critic_net, target_actor_net, target_critic_net = self._actor_net, self._critic_net, self._target_actor_net, self._targ
    actor_opt, critic_opt = self._actor_opt, self._critic_opt

    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    ## update critic ##
    # critic loss
    ## TODO ##
    q_value = critic_net(state, action)
    with torch.no_grad():
       a_next = target_actor_net(next_state)
       q_next = target_critic_net(next_state, a_next)
       q_target = reward + gamma * q_next * (1 - done)
    criterion = nn.MSELoss()
    critic_loss = criterion(q_value, q_target)

    # optimize critic
    actor_net.zero_grad()
    critic_net.zero_grad()
    critic_loss.backward()
    critic_opt.step()

    ## update actor ##
    # actor loss
    ## TODO ##
    action = actor_net(state)
    actor_loss = -critic_net(state, action).mean()
```

The actor network is trained using the critic network's Q-values in the following steps:

1. The actor network takes the current state as input and generates an action based on its policy. The objective is to select an action that maximizes the expected future rewards.
2. The critic network takes the current state and the generated action as input and predicts the corresponding Q-value. The Q-value represents the expected cumulative reward when taking that action in the given state.

- $s : current\ state$
- $Q : behavior\ critic\ network$
- $u : behavior\ actor\ network$

$$ActorLoss = -Q(s, u(s))$$

$$\nabla_{\theta^\mu} \mu|s_i \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|s_i$$

To update critic, firstly, we sample random minibatch from replay memory, and use actions of batch to calculate corresponding q value by critic network. We also need to calculate targe value.

$$\text{Set } y_i = r_i + \gamma Q'\left(s_{t+1}, \mu'\left(s_{t+1}\middle|\theta^{\mu'}\right)\middle|\theta^{Q'}\right)$$

$$\text{Update critic by minimizing the loss: } L = \frac{1}{N}\sum_i \left(y_i - Q\left(s_i, a_i|\theta^Q\right)\right)^2$$

Discount factor

By adjusting the discount factor, we can influence the agent's decision-making process. A smaller discount factor leads to a more myopic behavior, where the agent tends to prioritize short-term gains and immediate gratification. This can be useful in

scenarios where immediate rewards carry more significance or when the environment is highly volatile.

On the other hand, a larger discount factor encourages the agent to consider long-term consequences and prioritize actions that maximize cumulative rewards over time. This forward-thinking approach can be beneficial in situations where delayed rewards or strategic planning play a crucial role.

Epsilon-greedy

To ensure that the model explores different actions and doesn't always select the action deemed best for the current state, an exploration strategy called epsilon-greedy is commonly employed. Epsilon-greedy allows the model to randomly choose actions with a certain probability, denoted as ε.

Under the epsilon-greedy strategy, the model selects the action with the highest estimated value (exploitation), but with a probability ε, it chooses a random action (exploration) regardless of its value. This introduces randomness into the decision-making process, enabling the model to explore alternative actions and potentially discover better strategies or unforeseen rewards.

Target network

By employing a target network and periodically updating its weights from the behavior network, we can stabilize the training process and enhance the learning efficiency. This approach mitigates potential issues that may arise from using the same network for both the behavior and target Q-value estimations, resulting in more effective and stable training of the reinforcement learning agent.

Tricks in breakout

1. **stack frames**

   The use of stack frames helps address the challenge of capturing motion information in Atari games, where consecutive frames are crucial for understanding game dynamics and making accurate predictions. It allows the agent to perceive temporal changes in the game environment and facilitates learning strategies that rely on motion cues and dynamic interactions.

2. **clip reward**

   The purpose of reward clipping is to prevent extremely large or small rewards from disproportionately influencing the training process. Atari games often have varying reward scales, with some actions resulting in high positive rewards (e.g., obtaining points) and others leading to negative rewards (e.g., losing a life). These reward fluctuations can make the learning process more challenging and result in unstable training.

   By applying reward clipping, the range of rewards is constrained between -1 and 1. Any positive reward larger than 1 is truncated to 1, while any negative reward smaller than -1 is truncated to -1. This ensures that the agent receives rewards within a controlled range, allowing for more consistent and stable learning.