# Technical Report

111550049 林德倫 111550151 徐嘉亨 111550177 吳定霖

## 1. Introduction

Offline RL learns a policy from a pre-collected dataset without further interactions with the environments. This is called extrapolation error.
In this setting, we often encounter extrapolation error, which is caused by the mismatch between the data in the batch and the state-action visitation of the current policy.BCQ restricts the current policy to match the state-action pairs in the batch by penalizing those who are not in the batch. This is called Batch-Constrained.

Because estimating the similarity between state-action pairs in the batch and of the current policy is difficult , BCQ uses a generative model Variational auto-encoder (VAE) to output an action which is the action most likely in the batch given a state.And we use a perturbation model as an actor to get n actions near the action generated by VAE. This increases the diversity of the action without sampling n times from VAE. Then using the critic Q theta to choose the highest value action among these actions. Finally, we use two Q networks to implement Clipped Double Q-learning to penalize high variance value estimation. It can avoid exploring uncertainty (just like the original use to penalize bias). However, BCQ modified the original Clipped Double Q-learning, you can see details in our github repository.

## 2. Baseline

Because they provide their github link in the paper, we just clone it and do some modifications to be our baseline. They first use DDPG to collect data(trajectories) in the replay buffer and use this buffer to train BCQ offline. But we don't use DDPG to collect data, we use off-the-shelf dataset D4RL. So we modify the part of placing data into the replay buffer to ours.  The below figure shows how we implement placing D4RL data in the replay buffer.

```
# Load buffer
replay_buffer = utils.ReplayBuffer(state_dim, action_dim, device)
dataset = d4rl.qlearning_dataset(env)
N = dataset['rewards'].shape[0]
for i in range(N):
        obs = dataset['observations'][i]
        new_obs = dataset['next_observations'][i]
        action = dataset['actions'][i]
        reward = dataset['rewards'][i]
        done_bool = bool(dataset['terminals'][i])
        replay_buffer.add(obs, action, new_obs, reward, done_bool)
```

And we show the NN acrhitectures they use.

- VAE: It contains encoder and decoder, and the hidden size is 750.

```
self.e1 = nn.Linear(state_dim + action_dim, 750)
self.e2 = nn.Linear(750, 750)

z = F.relu(self.e1(torch.cat([state, action], 1)))
z = F.relu(self.e2(z))
```

```
self.d1 = nn.Linear(state_dim + latent_dim, 750)
self.d2 = nn.Linear(750, 750)
self.d3 = nn.Linear(750, action_dim)

a = F.relu(self.d1(torch.cat([state, z], 1)))
a = F.relu(self.d2(a))
return self.max_action * torch.tanh(self.d3(a))
```

- Actor: Its hidden sizes are 400 and 300

```
self.l1 = nn.Linear(state_dim + action_dim, 400)
self.l2 = nn.Linear(400, 300)
self.l3 = nn.Linear(300, action_dim)

a = F.relu(self.l1(torch.cat([state, action], 1)))
a = F.relu(self.l2(a))
a = self.phi * self.max_action * torch.tanh(self.l3(a))
return (a + action).clamp(-self.max_action, self.max_action)
```

- Critic: They use two critics, and the hidden sizes are 400 and 300.

```
self.l1 = nn.Linear(state_dim + action_dim, 400)
self.l2 = nn.Linear(400, 300)
self.l3 = nn.Linear(300, 1)

self.l4 = nn.Linear(state_dim + action_dim, 400)
self.l5 = nn.Linear(400, 300)
self.l6 = nn.Linear(300, 1)
```

```
q1 = F.relu(self.l1(torch.cat([state, action], 1)))
q1 = F.relu(self.l2(q1))
q1 = self.l3(q1)

q2 = F.relu(self.l4(torch.cat([state, action], 1)))
q2 = F.relu(self.l5(q2))
q2 = self.l6(q2)
```

And we show some hyperparameters they use.

| Optimizer | Adam |
|---|---|
| Batch size | 100 |
| Discount factor | 0.99 |
| Tau | 0.005 |
| Lambda(used for Clipped Double Q-learning) | 0.75 |
| Phi(used for perturbation) | 0.05 |
| Learning rate(actor) | 0.001 |
| Learning rate(critic) | 0.001 |
| Learning rate(VAE) | 0.001 |

# 3. Main approach

In this part, we introduce our main approach.  We do six modifications on the original BCQ and for each modification, we don't change any hyperparameters or something else except for the modified part.

1.

Method: Replace the VAE with conditional generative adversarial net (CGAN).

We use CGAN to generate the action. CGAN will take state and noise as input, so here the condition is state.

```python
class Generator(nn.Module):
    def __init__(self, state_dim, action_dim, latent_dim, max_action, device):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(state_dim + latent_dim, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, action_dim)

        self.max_action = max_action
        self.action_dim = action_dim
        self.latent_dim = latent_dim
        self.device = device

    def forward(self, state, noise):
        x = torch.cat([state,
        x = F.leaky_relu(self.
        x = F.leaky_relu(self.fc2(x))
        action = torch.tanh(self.fc3(x))  # Assuming action space is normalized to [-1, 1]
        return action * self.max_action

    def generate_noise(self, num_samples):
        noise = torch.normal(0, .3, size=(num_samples, self.latent_dim)).to(self.device).clamp(-1., 1.)
        return noise
```

(variable) fc2: Linear

```python
class Discriminator(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(state_dim + action_dim, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 1)

    def forward(self, state, action):
        x = torch.cat([state, action], dim=1)
        x = F.leaky_relu(self.fc1(x))
        x = F.leaky_relu(self.fc2(x))
        validity = torch.sigmoid(self.fc3(x))
        return validity
```

We use leaky ReLU to avoid the problem of zero gradients during the training. And both generator and discriminator's learning rate are 0.001.

```
real_labels = torch.normal(0.9, .1, size=(batch_size, 1)).clamp(0.8, 1.).to(self.device)  # soft target labels
real_output = self.dis(state, action)
real_d_loss = F.binary_cross_entropy(real_output, real_labels)

# Generate fake actions, train the discriminator with fake batch
noise = self.gen.generate_noise(batch_size)
fake_action = self.gen(state, noise)
false_labels = torch.normal(0.1, .1, size=(batch_size, 1)).clamp(0., 0.2).to(self.device)  # soft target labels
fake_output = self.dis(state, fake_action.detach())
fake_d_loss = F.binary_cross_entropy(fake_output, false_labels)
D_loss = real_d_loss + fake_d_loss

self.dis_optimizer.zero_grad()
D_loss.backward()
self.dis_optimizer.step()

# Generator loss
fake_output = self.dis(state, fake_action)
g_loss = F.binary_cross_entropy(fake_output, real_labels)

self.gen_optimizer.zero_grad()
g_loss.backward()
self.gen_optimizer.step()
```

For the training part, we update the discriminator first. Use BCE loss between real labels and the score of data on the batch. And the BCE loss between fake labels and the score of state with action output by generator. Combine two losses to update the discriminator. And we use the technique of label smoothing to make our GAN not be overfitting. Finally, we update the generator. Use the updated discriminator to evaluate the score of action output by generator, and we use BCE loss between the score and real label to update the generator. The reason why we use real labels is because we want the generator to produce actions that the discriminator will classify as very similar to data in the batch.
Target: Try to use CGAN to get better performance.

2.

Method: Replace the modified Clipped Double Q-learning with modified Quadruple Q-learning (i.e. we will need 4 Q-networks instead of 2)

The original Clipped Double Q-learning is

$$y = r + \gamma \max_{a_i} \left[ \min_{j=1,2} Q_{\theta'_j}(s', a_i) \right]$$

and they convert it into their modified version. You can notice that if lambda is 0, then it is just the original Clipper Double Q-learning.

$$y = r + \gamma \max_{a_i} \left[ \lambda \min_{j=1,2} Q_{\theta'_j}(s', a_i) + (1 - \lambda) \max_{j=1,2} Q_{\theta'_j}(s', a_i) \right]$$

and we convert their into ours

$$y = r + \gamma \max_{a_i} \left[ \lambda \min_{j=1,2,3,4} Q_{\theta'_j}(s', a_i) + (1 - \lambda) \max_{j=1,2,3,4} Q_{\theta'_j}(s', a_i) \right]$$

So we use four Q-networks to achieve this.

```
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.l1 = nn.Linear(state_dim + action_dim, 200)
        self.l2 = nn.Linear(200, 150)
        self.l3 = nn.Linear(150, 1)

        self.l4 = nn.Linear(state_dim + action_dim, 200)
        self.l5 = nn.Linear(200, 150)
        self.l6 = nn.Linear(150, 1)

        self.l7 = nn.Linear(state_dim + action_dim, 200)
        self.l8 = nn.Linear(200, 150)
        self.l9 = nn.Linear(150, 1)

        self.l10 = nn.Linear(state_dim + action_dim, 200)
        self.l11 = nn.Linear(200, 150)
        self.l12 = nn.Linear(150, 1)


    def forward(self, state, action):
        q1 = F.relu(self.l1(torch.cat([state, action], 1)))
        q1 = F.relu(self.l2(q1))
        q1 = self.l3(q1)

        q2 = F.relu(self.l4(torch.cat([state, action], 1)))
        q2 = F.relu(self.l5(q2))
        q2 = self.l6(q2)

        q3 = F.relu(self.l7(torch.cat([state, action], 1)))
        q3 = F.relu(self.l8(q3))
        q3 = self.l9(q3)

        q4 = F.relu(self.l10(torch.cat([state, action], 1)))
        q4 = F.relu(self.l11(q4))
        q4 = self.l12(q4)
```

```
with torch.no_grad():
    # Duplicate next state 10 times
    next_state = torch.repeat_interleave(next_state, 10, 0)

    # Compute value of perturbed actions sampled from the VAE
    target_Q1, target_Q2, target_Q3, target_Q4 = self.critic_target(next_state, self.actor_target(next_state, self.vae.decode(next_state)))

    # Soft Clipped Double Q-learning
    target_Q = self.lmbda * torch.min(torch.min(target_Q1, target_Q2), torch.min(target_Q3, target_Q4)) + (1. - self.lmbda) * torch.max(torch.max(target_Q1, target_Q2), torch.max(target_Q3, target_Q4))
    # Take max over each action sampled from the VAE
    target_Q = target_Q.reshape(batch_size, -1).max(1)[0].reshape(-1, 1)

    target_Q = reward + not_done * self.discount * target_Q

current_Q1, current_Q2, current_Q3, current_Q4 = self.critic(state, action)
critic_loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(current_Q2, target_Q) + F.mse_loss(current_Q3, target_Q) + F.mse_loss(current_Q4, target_Q)
```

You can notice that we halve the hidden size from (400, 300) to (200, 150). This is because we don't have enough time and resources to train these large models.

Target: We hope it makes the total reward more consistent than the original BCQ.

3.

Method: Shared Layer

Make the Actor and Critic share the first layer of the neural network.

```
class Actor_and_Critic(nn.Module):
    def __init__(self, state_dim, action_dim, max_action, phi=0.05):
        super(Actor_and_Critic, self).__init__()
        self.shared_l1 = nn.Linear(state_dim + action_dim, 400)
        self.actor_l1 = nn.Linear(400, 300)
        self.actor_l2 = nn.Linear(300, action_dim)

        self.critic_l1 = nn.Linear(400, 300)
        self.critic_l2 = nn.Linear(300, 1)

        self.critic_l3 = nn.Linear(400, 300)
        self.critic_l4 = nn.Linear(300, 1)

        self.max_action = max_action
        self.phi = phi
```

```
    def forward(self, state, action):
        a = F.relu(self.shared_l1(torch.cat([state, action], 1)))
        a = F.relu(self.actor_l1(a))
        a = self.phi * self.max_action * torch.tanh(self.actor_l2(a))

        q1 = F.relu(self.shared_l1(torch.cat([state, action], 1)))
        q1 = F.relu(self.critic_l1(q1))
        q1 = self.critic_l2(q1)

        q2 = F.relu(self.shared_l1(torch.cat([state, action], 1)))
        q2 = F.relu(self.critic_l3(q2))
        q2 = self.critic_l4(q2)

        return (a + action).clamp(-self.max_action, self.max_action), q1, q2
```

Target: In the network, we can spend less computation time and parameters but still reach the same performance compared with the

original BCQ.

4.

Method: Removing Perturbation Model

Make BCQ directly use the action generated by VAE. The perturbation is implemented by the actor in BCQ, so we remove the whole actor perturbation model. Then we modified all parts which need to generate action from originally getting action by an actor perturbation model to getting action where directly generated by VAE.

```
def select_action(self, state):
    with torch.no_grad():
        state = torch.FloatTensor(state.reshape(1, -1)).repeat(100, 1).to(self.device)
        action = self.vae.decode(state)
        q1 = self.critic.q1(state, action)
        ind = q1.argmax(0)
    return action[ind].cpu().data.numpy().flatten()
```

```
# Critic Training
with torch.no_grad():
    # Duplicate next state 10 times
    next_state = torch.repeat_interleave(next_state, 10, 0)

    # Compute value of perturbed actions sampled from the VAE
    target_Q1, target_Q2 = self.critic_target(next_state, self.vae.decode(next_state))

    # Soft Clipped Double Q-learning
    target_Q = self.lmbda * torch.min(target_Q1, target_Q2) + (1. - self.lmbda) * torch.max(target_Q1, target_Q2)
    # Take max over each action sampled from the VAE
    target_Q = target_Q.reshape(batch_size, -1).max(1)[0].reshape(-1, 1)

    target_Q = reward + not_done * self.discount * target_Q
```

Target: By removing the perturbation model as known as the actor in BCQ, we can reduce computation costs, and make the model to choose the action most likely in the batch VAE thoughts.

5.

Method: Change the discount factor to 0.9

Target: We try a different gamma value to find the potential better model.

6.

Method: Change the batch size to 200

Target: We hope the model performs better by increasing batch size.

# 4. Evaluation & Metric

We first introduce how we evaluate the performance of each method. We just take the evaluation function from the original BCQ code on Github. Because we train each seed with 1M steps on one dataset and we evaluate current policy every 5000 steps, so we will get a total of 200 evaluations, and we save them as .npy. And we train each dataset over 3 random seeds, which are 0, 1, 2. So our final results are the average evaluations over 3 seeds on each dataset.

The result in these metrics is evaluated by the average in the range of training steps 500k steps to 545k steps(actually 10 evaluations) over 3 seeds. The reason why we choose the average in this range is because we notice that almost all methods will converge to a specific reward or having an oscillation curve but its average doesn't change a lot in this range on each dataset. The red score means that it's worse than baseline. The blue score means that it's better than baseline. The black score means that it has the same performance as baseline.

| | BCQ(origin) | BCQ(CGAN) | BCQ(quadruple) | BCQ(shared) | BCQ(no perturbation) | BCQ(gamma 0.9) | BCQ(batch size 200) |
|---|---|---|---|---|---|---|---|
| hopper-random | 326.5 | 304.4 | 325.0 | 309.55 | 303.3 | 315.0 | 326.6 |
| hopper-medium | 1713.2 | 251.7 | 1266.2 | 1151.016 | 987.2 | 899.7 | 1815.5 |
| hopper-expert | 3470.3 | 202.4 | 3498.3 | 1917.694 | 3500.1 | 2449.8 | 3166.6 |
| hopper-medium-replay | 925.6 | 134.3 | 962.1 | 987.233 | 918.0 | 302.9 | 908.9 |
| hopper-medium-expert | 3590.3 | 299.3 | 3489.7 | 3332.071 | 3555.4 | 245.4 | 3614.7 |
| walker2d-random | 243.8 | 313.2 | 233.7 | 222.877 | 105.7 | 250.5 | 239.0 |
| walker2d-medium | 2468.6 | 1187.0 | 2304.5 | 2367.923 | 962.5 | 941.5 | 2545.9 |
| walker2d-expert | 3826.4 | 325.6 | 4246.1 | 3553.091 | 3376.2 | 3804.5 | 3948.1 |
| walker2d-medium-replay | 871.7 | 719.7 | 695.7 | 755.087 | 483.2 | 455.1 | 631.8 |
| walker2d-medium-expert | 2275.3 | 491.1 | 2299.0 | 2633.07 | 1984.3 | 1183.0 | 2321.1 |
| maze2d-umaze | 82.3 | 36.2 | 79.4 | 2.4 | 66.8 | 45.4 | 85.0 |
| maze2d-medium | 46.5 | 21.3 | 92.8 | 1.5 | 86.3 | 31.2 | 52.0 |
| maze2d-large | 74.1 | 175.0 | 157.8 | 5.9 | 97.2 | 8.4 | 116.2 |
| antmaze-medium-diverse | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| antmaze-medium-play | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 |

Evaluation metric of step2

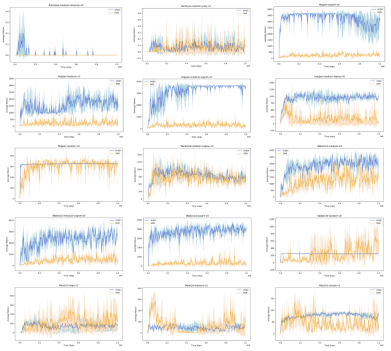
| | BCQ(origin) | BCQ(CGAN) | BCQ(quadruple) | BCQ(shared) | BCQ(no perturbation) | BCQ(gamma 0.9) | BCQ(batch size 200) |
|---|---|---|---|---|---|---|---|
| pen-human | 1811.9 | -0.2 | 1695.1 | 1716.8 | 1587.4 | 1802.4 | 2163.7 |
| pen-cloned | 573.1 | 215.4 | 1318.5 | 754.4 | 492.4 | 926.1 | 549.1 |
| pen-expert | 3651.3 | 1029.8 | 3621.7 | 3532.9 | 3669.7 | 3407.0 | 4010.2 |
| hammer-human | -139.9 | -248.2 | -218.3 | -114.8 | 53.5 | -171.6 | -226.1 |
| hammer-cloned | -228.2 | -232.7 | -227.8 | -185.7 | -148.9 | 30.9 | -227.7 |
| hammer-expert | 10659.3 | -194.8 | 15280.5 | 14408.3 | 15728.6 | 13432.0 | 15096.3 |
| door-human | -58.6 | -58.6 | -60.5 | -55.8 | 25.5 | -49.5 | -59.5 |
| door-cloned | -56.3 | -62.0 | -56.1 | -55.9 | -52.6 | -54.4 | -57.8 |
| door-expert | 2945.4 | -46.5 | 2767.5 | 2820.2 | 3023.5 | 2904.9 | 2918.3 |
| relocate-human | -8.4 | -14.9 | -7.9 | -8.0 | -5.9 | -7.6 | -9.3 |
| relocate-cloned | -17.3 | -17.8 | -17.8 | -17.6 | -16.9 | -16.5 | -17.5 |
| relocate-expert | 1998.5 | -18.6 | 1830.1 | 2081.8 | 4391.7 | 2107.8 | 2601.3 |

Evaluation metric of step3

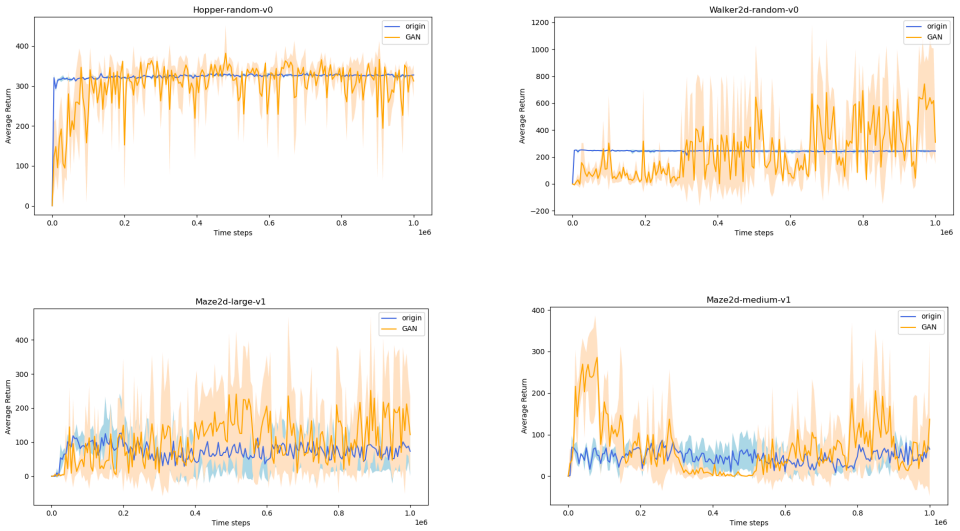
# 5. Result & Analysis

In this part, we will discuss some important results. We first show the overview of all results, the orange curve is our method. Please refer to the appendix if you want to see all the results.

## 1. CGAN (step2)



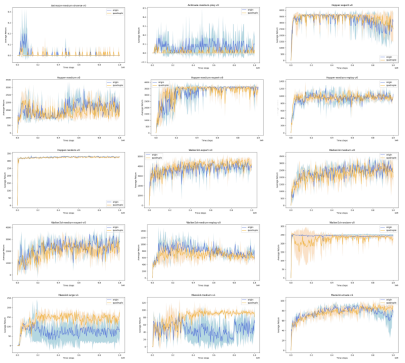
Overview of step2 CGAN results



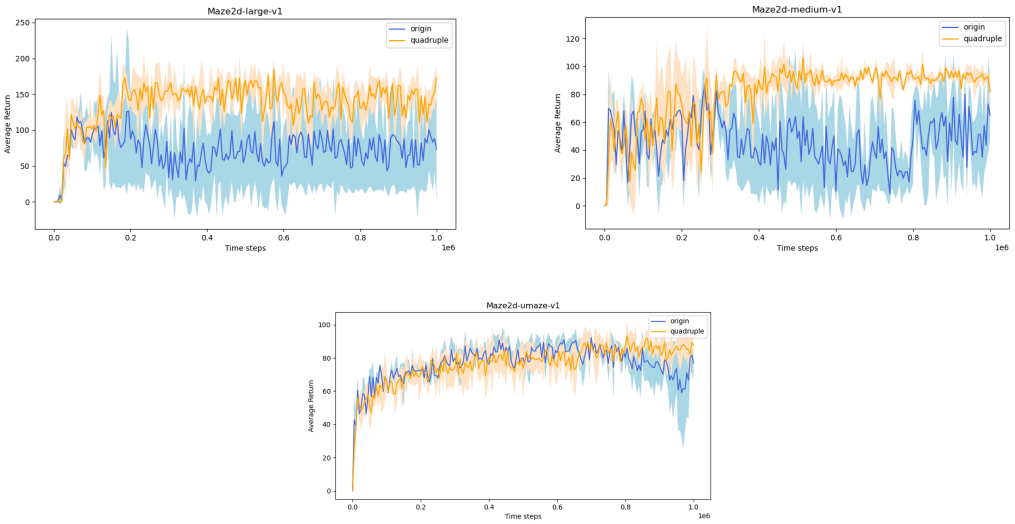

We choose Hopper-random, Walker2d-random, Maze2d-large, Maze2d-medium as our representatives. We notice that CGAN reaches near or better performance than baseline only in random and maze dataset. We think it is because random datasets are generated by random initialized policy, and BCQ will follow the data so it will not perform well. Which means it needs exploration to get better results(The exploration here means to increase the diversity of seen actions). And the

maze dataset also needs more exploration. We think CGAN will have more exploration during the contest between generator and discriminator, that is why it has oscillation training curves. Due to these reasons, we think the result we get in these dataset is reasonable. But in other datasets, they are generated by some expert or something else. So just following the data can get good results. Which means it doesn't need more exploration. So CGAN gets bad performance on those datasets.

2. Quadruple (step2)



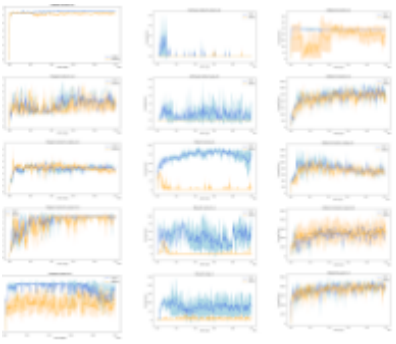
Overview of step2 Quadruple results



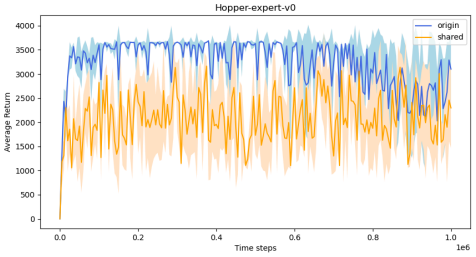

We choose maze datasets as our representatives. We notice that the use of Clipped Quadruple Q-learning reaches near performance in other datasets. But it has better performance in the maze datasets. We think the reason is that it might encounter some unknown states and the original

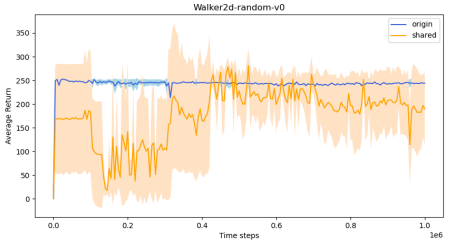
BCQ cannot estimate the corresponding actions accurately. But the use of Clipped Quadruple Q-learning indeed makes the total reward more consistent than the original BCQ even encountering some unknown states. And maze datasets contain many unique states, so it might be possible for BCQ to encounter more unknown states during training than other datasets.

3. Shared Layer (step2)



Overview of step2 Shared Layer results



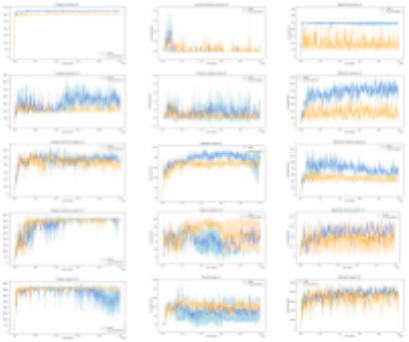




We choose Hopper-expert and Walker2d-random as our representatives. You can see that the shared first layer method in both graphs has worse performance than the original one. In Hopper-expert, original BCQ converges to a policy that can gain up to 3500 scores in the very beginnings, but the shared first layer method can only gain around 2000 scores. It also oscillates between 1500 to 2500 scores and cannot converge to an optimal policy. Same results in the Walker2d-random, original BCQ converges to an optimal policy with 250 scores, but the

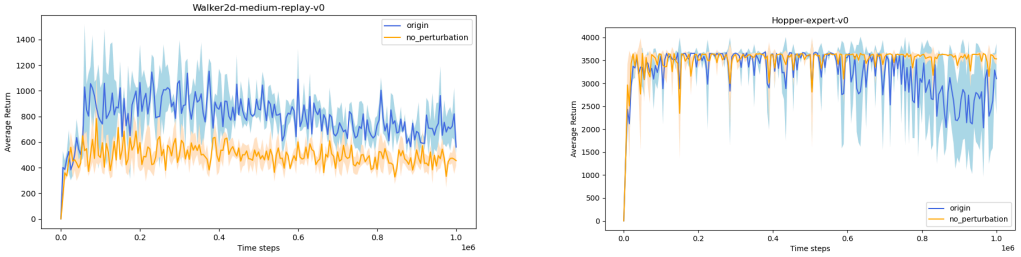
shared first layer method oscillates between 150 to 250 scores and cannot converge to an optimal policy.

The Possible reason may be that when we modify the original Actor and Critic into a shared first layer Actor and Critic, we reduce the numbers of parameters in the neural networks. So the complexity of the Actor Critic neural networks model decreases. If the task in the model is too complex for the shared layer model, then the model cannot fit well to the model behind Actor and Critic. Another possible reason is that if the Actor and Critic do not have that many features in common, then the first shared layer gives little helpful information to Actor and Critic. In conclusion, the shared first layer method does poorly in step2.

## 4. Removing Perturbation Model (step2)



Overview of step2 Removing Perturbation Model results

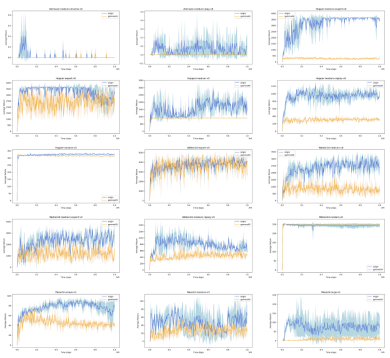



We choose Walker2d-medium-replay and Hopper-expert-v0 as our representatives. You can see that removing the perturbation model has worse performance in Walker2d-medium-replay but has better

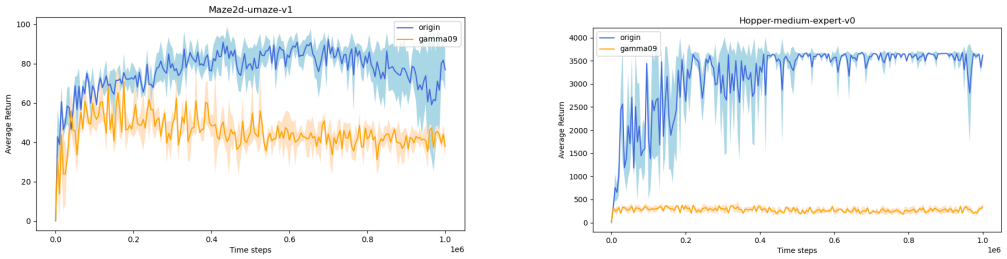
performance in Hopper-expert-v0. In Walker2d-medium-replay, removing the perturbation model method and original BCQ evaluation outcomes both oscillate around a stable average value, but removing the perturbation model method has a lower average evaluation value. In Hopper-expert-v0, for the first half time steps, removing the perturbation model method and original BCQ have similar evaluation outcomes, converging to an optimal policy with around 3500 scores. However, for the second half time steps, the evaluation outcomes of the original BCQ started to fall. Maybe this occurred because it explored some states and actions that did not belong to the expert batch. And removing the perturbation model method still stuck to the optimal policy so the evaluation results didn't change.

The possible reason is that the function of the perturbation model is increasing the diversity of actions, so the BCQ algorithm can make exploration more effective. When we remove the perturbation function, we can only get the action directly generated by VAE, that means we totally trust the action VAE generated. So in the environment that is more random, removing the perturbation model method may not explore the stares, actions in the batch enough, so it cannot find the optimal policy and can only converge to a local minimum point. In comparison, in the expert environment, removing the perturbation model method can be more restricted to the states and actions that are in a small portion in the batch. So it sticks to the expert trajectory, and gets a good performance.

5. Change gamma to 0.9 (step2)

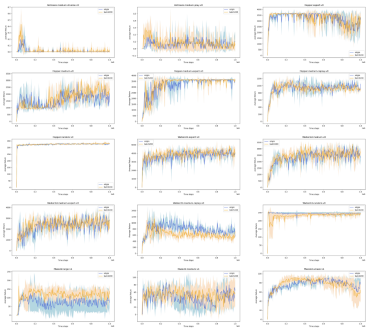


Overview of step2 Gamma 0.9 results

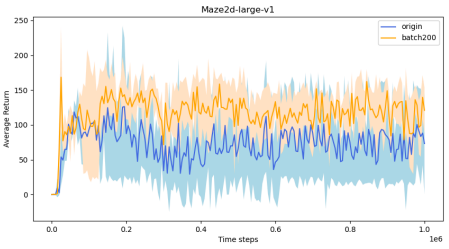


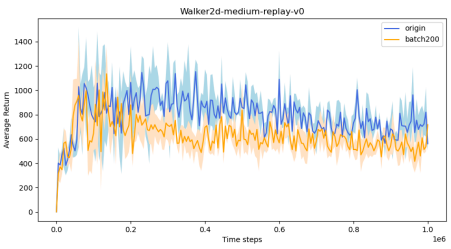

We choose Maze2d-umaze and Hopper-medium-expert as our representatives. According to the chart, we can see that the performance of the model trained under Gamma=0.9 is significantly worse than the original one. It seems that 0.99 is a more ideal discount factor under those tasks.

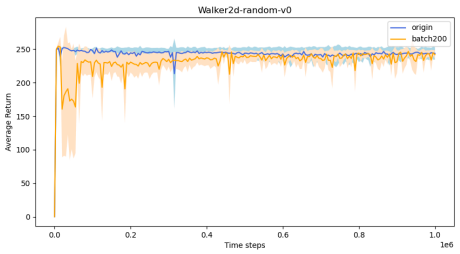
## 6. Change batch size to 200 (step2)



Overview of step2 batch size 200 results

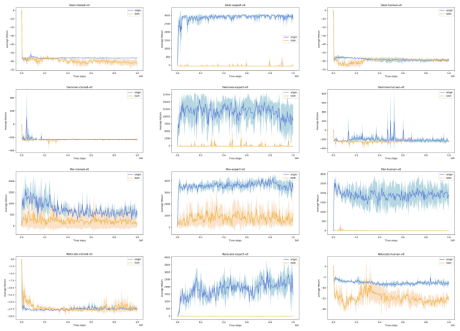


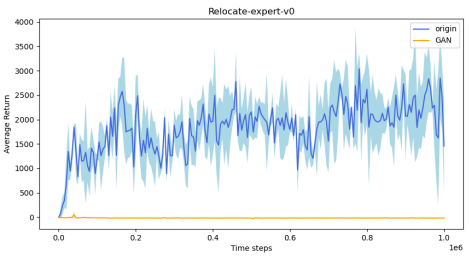



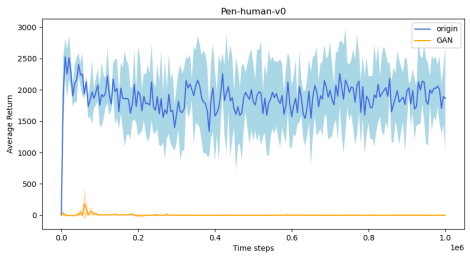

We choose Maze2d-large, Walker2d-random and Walker2d-medium-replay as our representatives. According to the chart, the performance of the model trained under batchsize=200 has a great improvement under Maze2d-large task, but performs poorly under Walker2d-medium-replay task. The effect of this modification is underterministic.

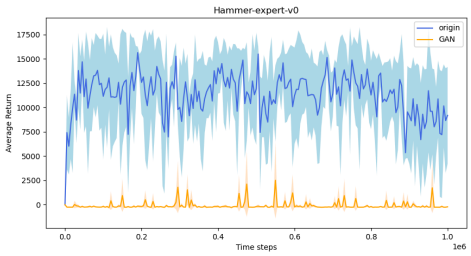
## 1. CGAN (step3)

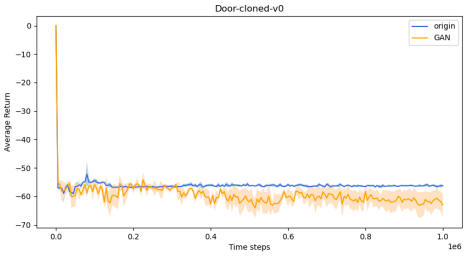


Overview of step3 CGAN results



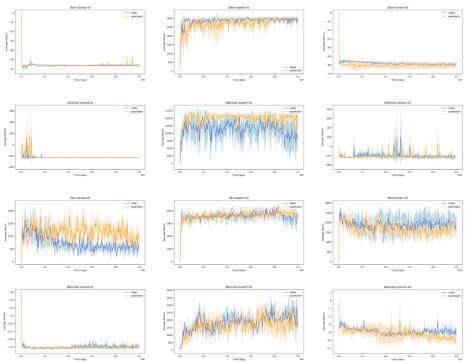





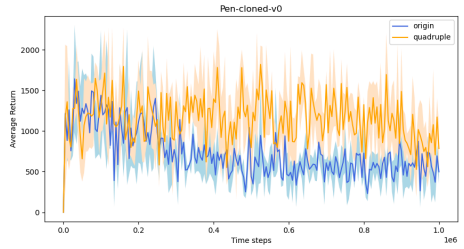



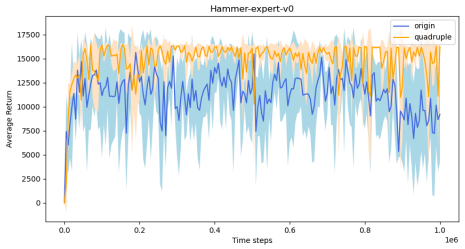

We choose Relocate-expert, Pen-human, Hammer-expert, and Door-cloned as our representatives. We notice that CGAN fails in almost all datasets. We think the reason is that exploration in these more complex tasks doesn't help at all. Which means a little change will result in the failures in these tasks. For example, in the door dataset if we change the expert action a bit(change the angle very small to open the door), it might fail to open the door. So it has such a bad performance.

2. Quadruple (step3)



Overview of step3 Quadruple results

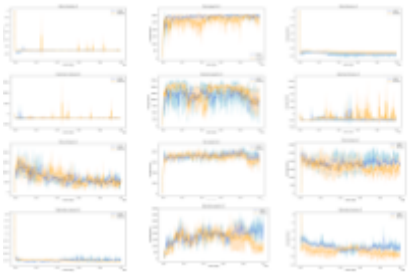


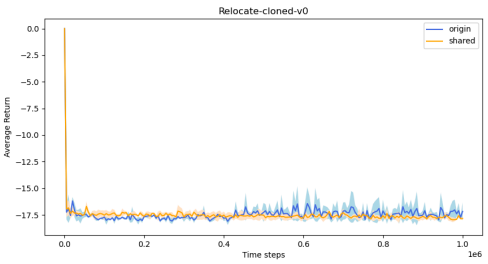

We choose Pen-cloned and Hammer-expert as our representatives. We notice that as the same in step2, the use of Clipped Quadruple Q-learning reaches near performance in other datasets. We think the reason is also

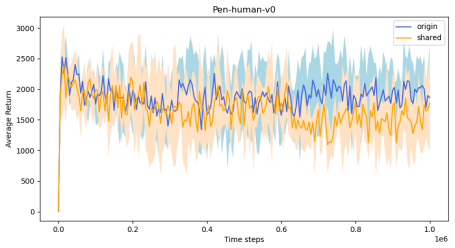
the same as step2. So the use of  Clipped Quadruple Q-learning indeed makes the total reward more consistent than the original BCQ even encountering some unknown states.

3. Shared Layer (step3)



Overview of step3 Shared Layer results



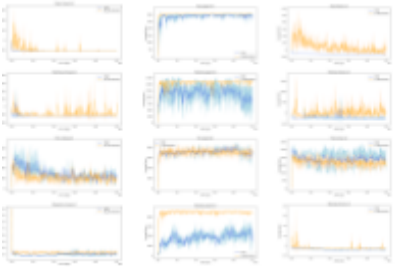




We choose Relocate-cloned and Pen-human as our representatives. You can see that in both graphs, the shared first layer method performs well at the first half time steps, and has worse performance than the original BCQ in the second half time steps. Although the above two examples have this phenomenon, evaluation results of other tasks may not have this kind of  results.

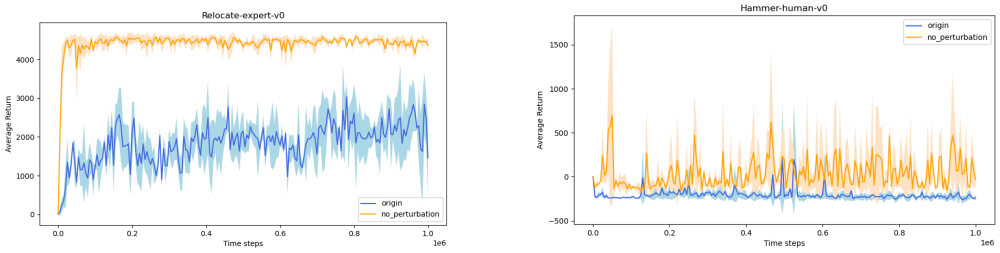
It is hard to explain the reason why it performs like this. I guess that the possible reason is that when we use the share first layer method, we reduce the number of parameters, so with fewer parameters, the policy can converge to some local minimum faster. Therefore, it performs better at the beginning. However, with fewer parameters, it cannot fit the complex model behind. So with enough time steps to update iterations, original BCQ can converge to a better policy than the share first layer

method. Therefore, the original BCQ performs better in the second half time steps.

## 4. Removing Perturbation Model (step3)



Overview of step3 Removing Perturbation Model results

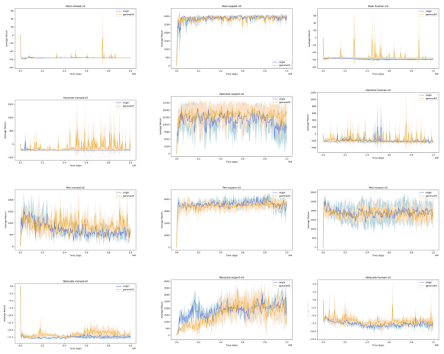



We choose Relocate-expert and Hammer-human as our representatives. In the small picture of "Overview of step3 Removing Perturbations Model results", we can see that almost all the tasks perform better than the original BCQ in the Adroit task, and removing perturbation model method performs extremely well in the expert task. In Hammer-human, you can see that removing the perturbation method always has a little bit better evaluation results than the original BCQ. In Relocate-expert, removing the perturbation method has a huge improvement than the original BCQ.

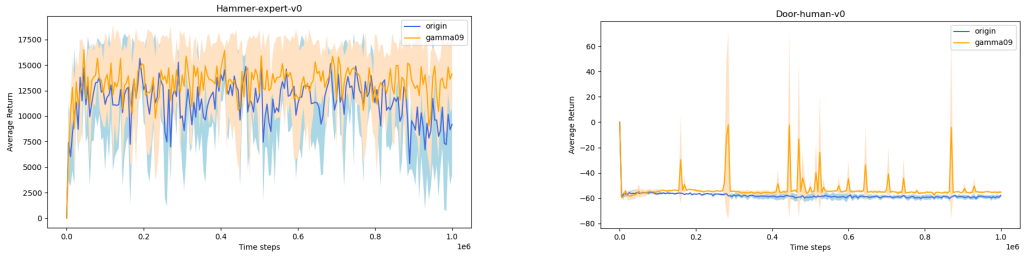
The possible reason is that the function of the perturbation model is increasing the diversity of actions, so the BCQ algorithm can make exploration more effective. When we remove the perturbation function, we can only get the action directly generated by VAE, that means we totally trust the action VAE generated. The Adroit tasks are more difficult and complex, so if the actor accidentally explores the states and actions

which do not belong to the batch, it may easily fall into very bad policy and never become better again. Therefore, removing perturbation and restricting the states and actions in a small portion that we can ensure that is the region without uncertainty is much safer although it may not be able to explore the optimal policy. In the expert environment, we can trust the states and actions in the batch which experts provided. So without perturbation, we can follow a safer and good performance policy in this kind of difficult and complex tasks.

5. Change gamma to 0.9 (step3)

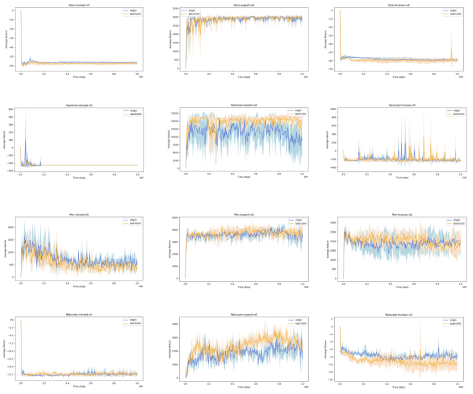


Overview of step3 Gamma 0.9 results

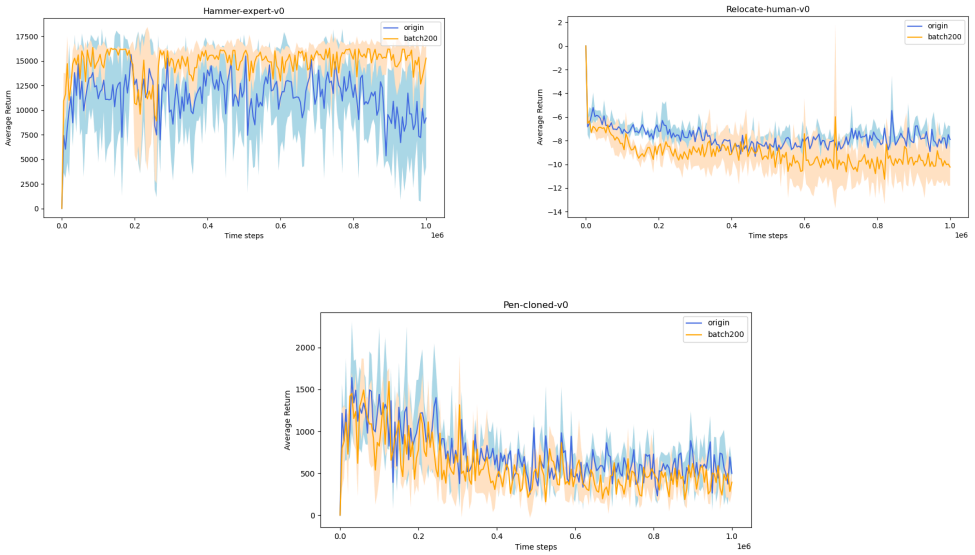



We choose Hammer-expert and Door-human as our representatives. According to the chart above, we can compare the performance between the models trained under Gamma=0.99 and 0.9. Surprisingly, the model trained under Gamma=0.9 performs better Gamma=0.99 one under these tasks. It seems that Gamma=0.9 is suitable for these tasks.

6. Change batch size to 200 (step3)



Overview of step3 Batch size 200 results




We choose Hammer-expert, Relocate-human, and Pen-cloned as our representatives. According to the chart above, we can compare the performance between the models trained under Gamma=0.99 and 0.9. We can see that the improvement of the modification of the batchsize is still uncertain.

# 6. Conclusion

By comparing all the methods with the original BCQ, we find that none of our methods can completely exceed the baseline. Although some methods perform well(ex. quadruple), but it is still slightly worse than baseline on some tasks. So using multiple methods on different tasks or using one method on all tasks might be a trade-off, we haven't found the algorithm, NN architectures, and hyperparameters that can surpass all the others. We also find a potential issue in the practical BCQ algorithm(we find it in the part of Clipped Quadruple Q-learning). That is, although the use of VAE and Perturbation model in BCQ addresses the problem of extrapolation error in offline settings, it still struggles to accurately fit some state-action pairs that are out-of-distribution (OOD) from the behavior policy distribution.

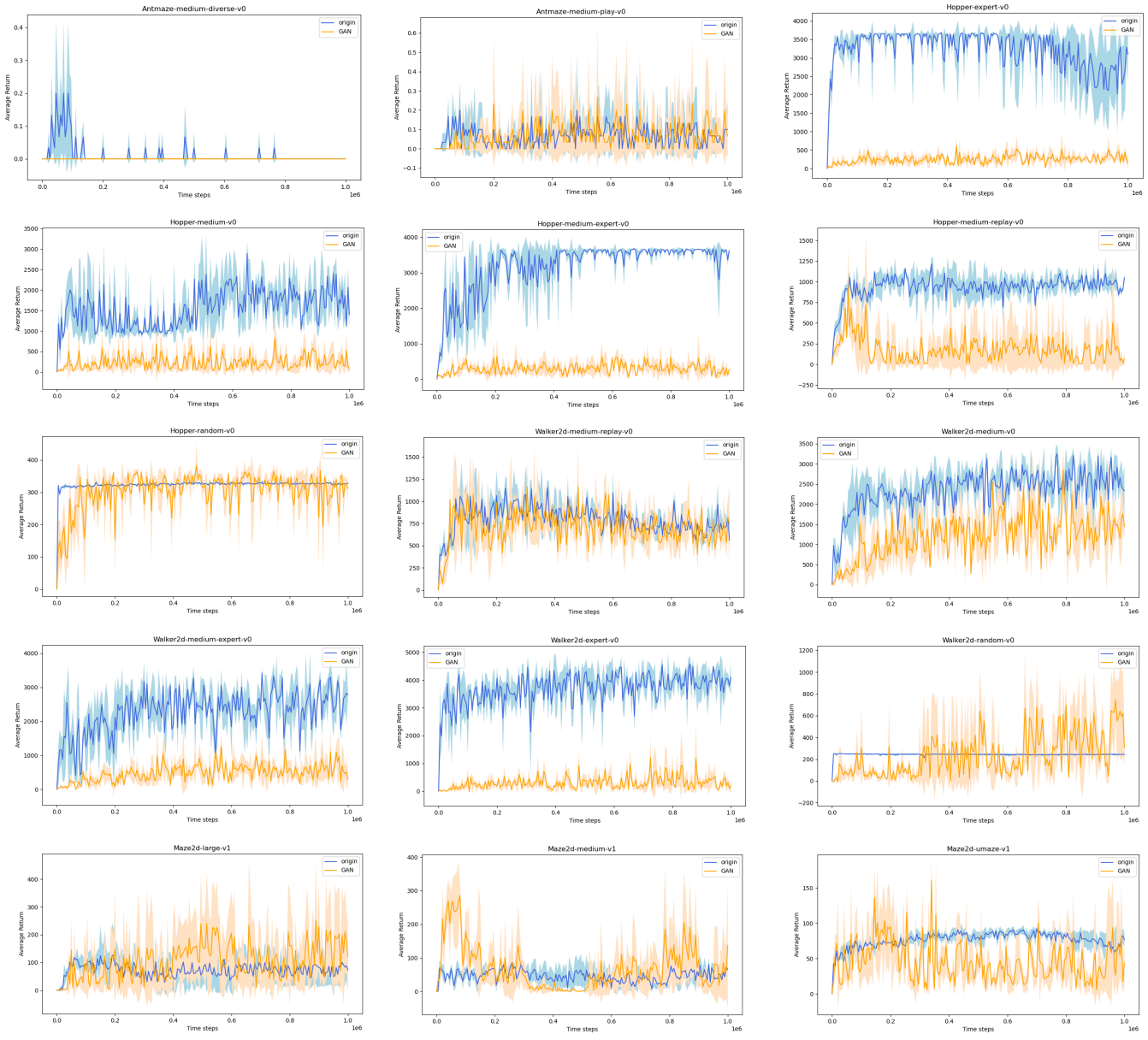
# 7. References


- https://arxiv.org/pdf/2004.07219
- https://arxiv.org/pdf/1812.02900
- https://arxiv.org/pdf/1802.09477v3
- https://github.com/sfujim/BCQ/tree/master
- https://blog.csdn.net/gsww404/article/details/123926753
- https://blog.csdn.net/weixin_43788106/article/details/123925913
- https://arxiv.org/pdf/1411.1784
- https://blog.csdn.net/yunlong_G/article/details/116375556
- https://blog.csdn.net/qq_24224067/article/details/104293409

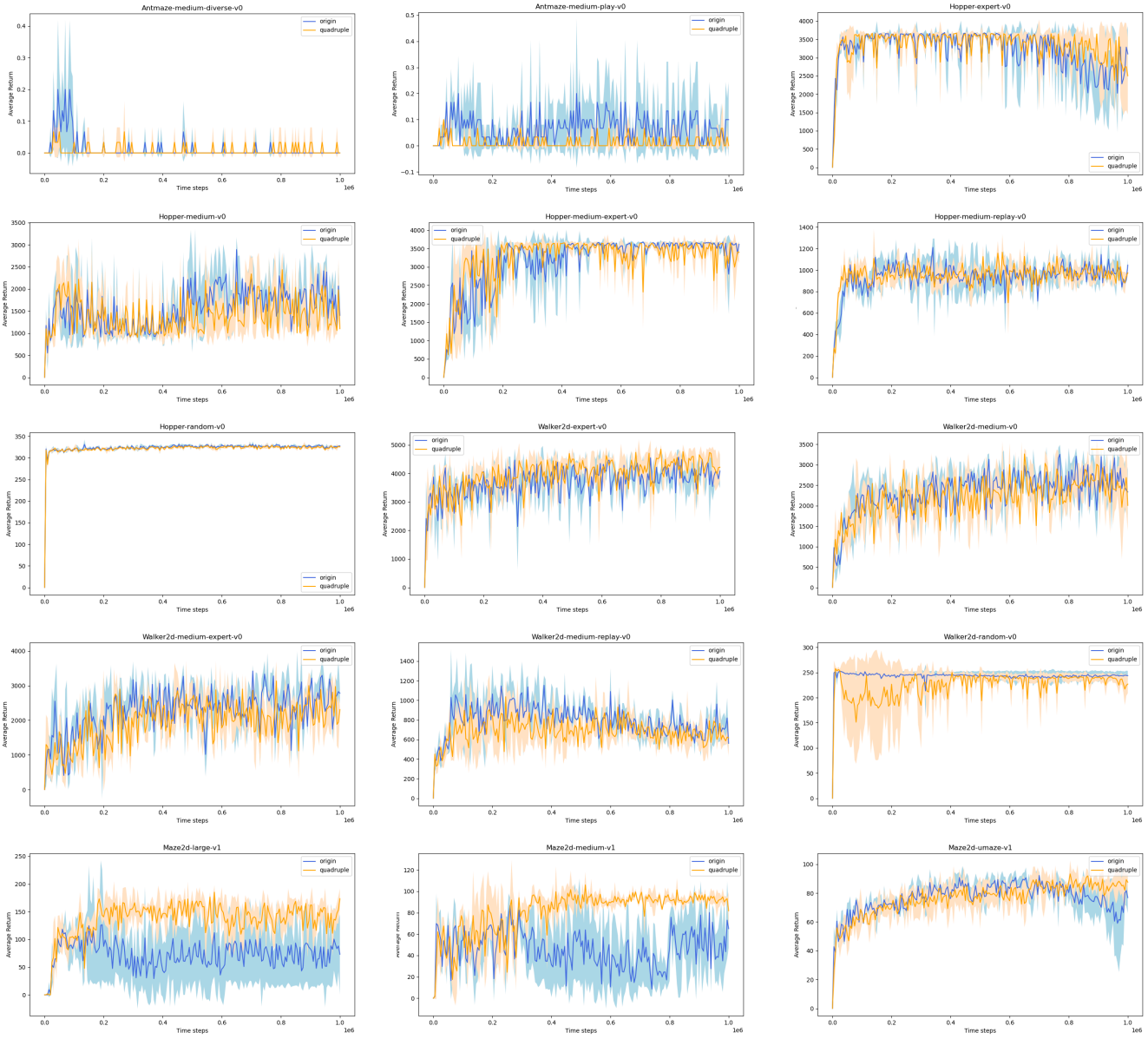
# 8. Appendix

We show all our result figures here.

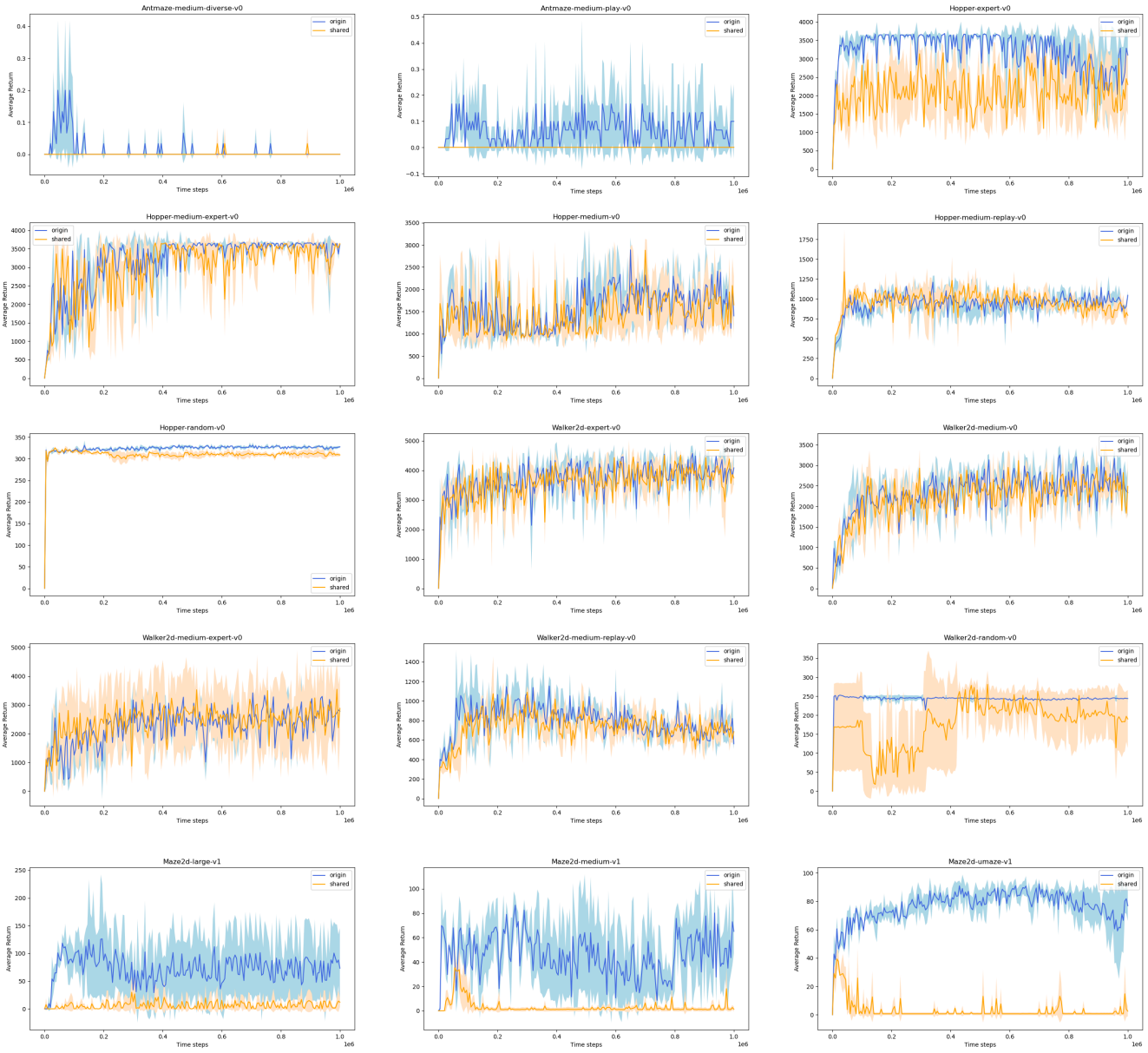
Only compare with baseline(step2):



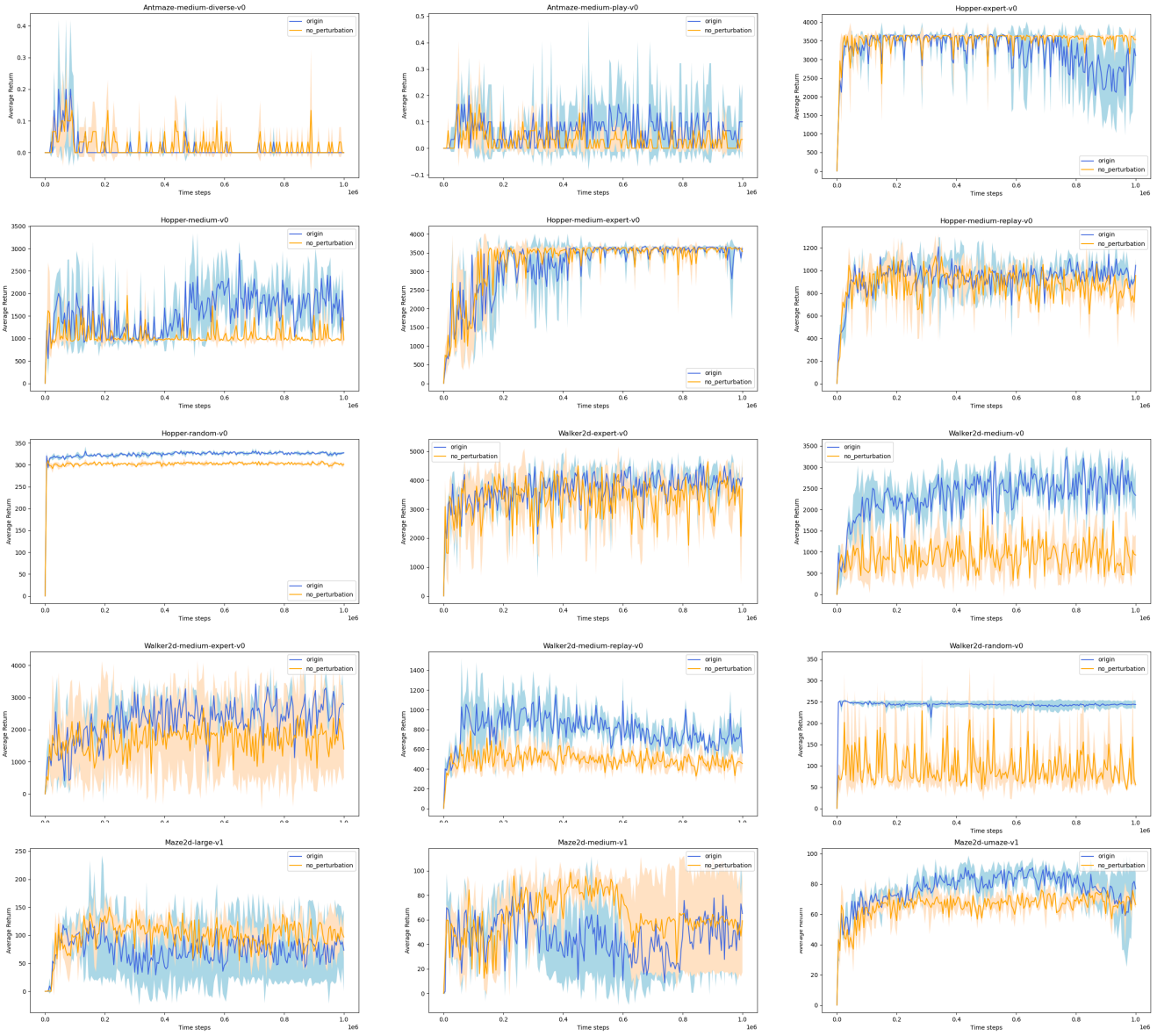

CGAN


Antmaze-medium-diverse-v0
Antmaze-medium-play-v0
Hopper-expert-v0
Hopper-medium-v0
Hopper-medium-expert-v0
Hopper-medium-replay-v0
Hopper-random-v0
Walker2d-expert-v0
Walker2d-medium-v0
Walker2d-medium-expert-v0
Walker2d-medium-replay-v0
Walker2d-random-v0
Maze2d-large-v1
Maze2d-medium-v1
Maze2d-umaze-v1
origin
quadruple
Average Return
Time steps
1e6

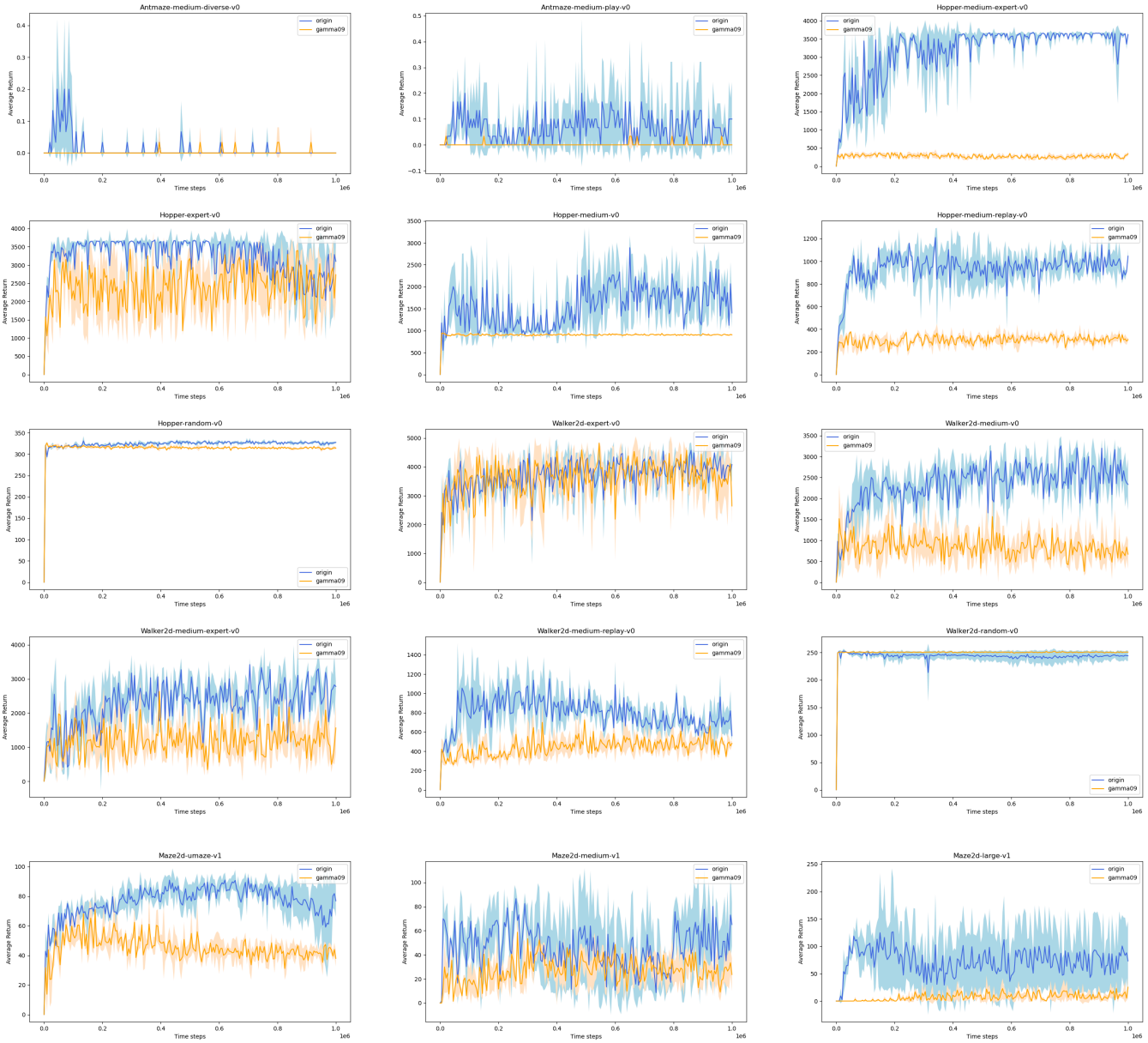
Quadruple


Antmaze-medium-diverse-v0
Antmaze-medium-play-v0
Hopper-expert-v0
Hopper-medium-expert-v0
Hopper-medium-v0
Hopper-medium-replay-v0
Hopper-random-v0
Walker2d-expert-v0
Walker2d-medium-v0
Walker2d-medium-expert-v0
Walker2d-medium-replay-v0
Walker2d-random-v0
Maze2d-large-v1
Maze2d-medium-v1
Maze2d-umaze-v1
origin
shared
Average Return
Time steps
1e6

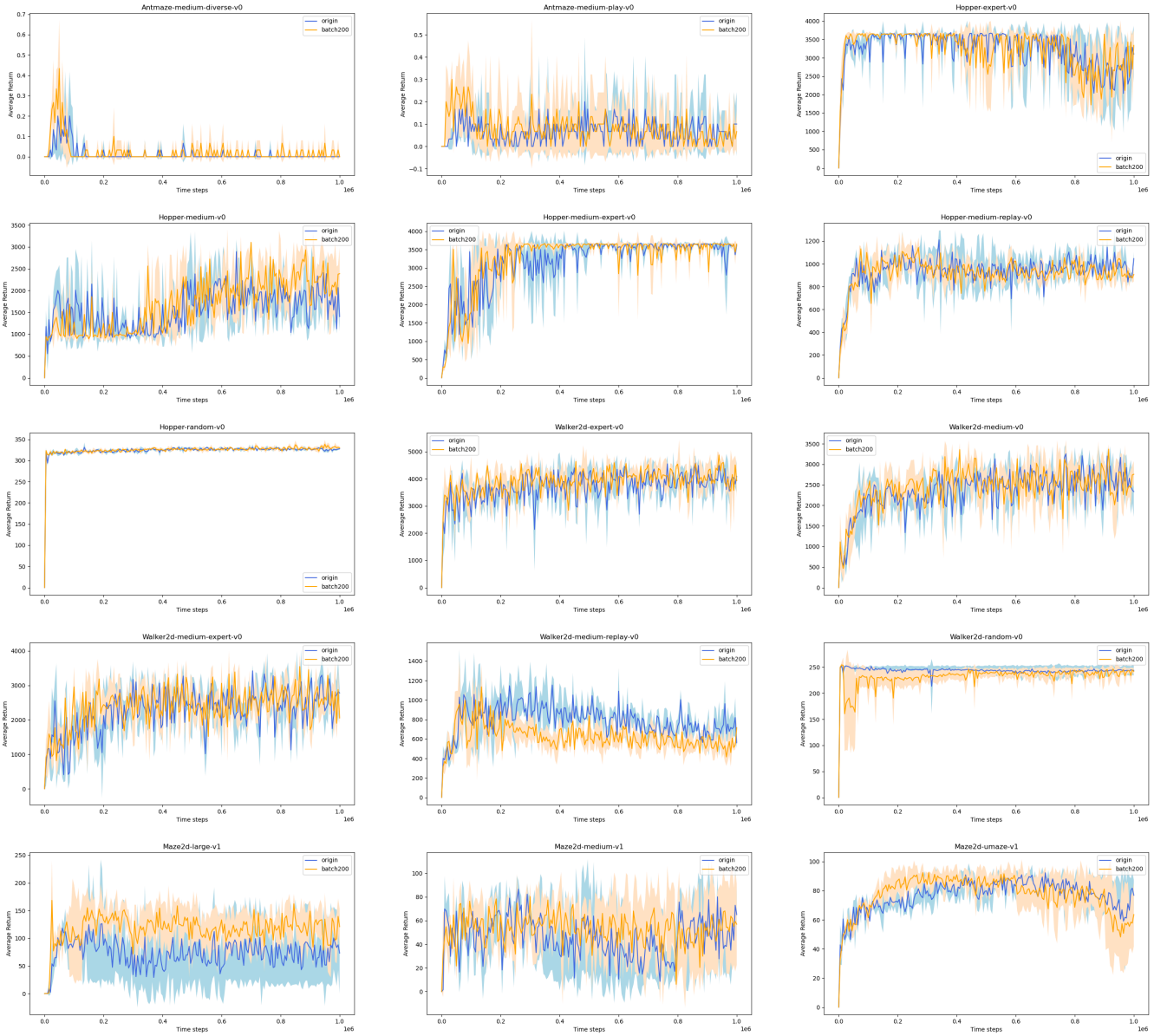
Shared




No perturbation

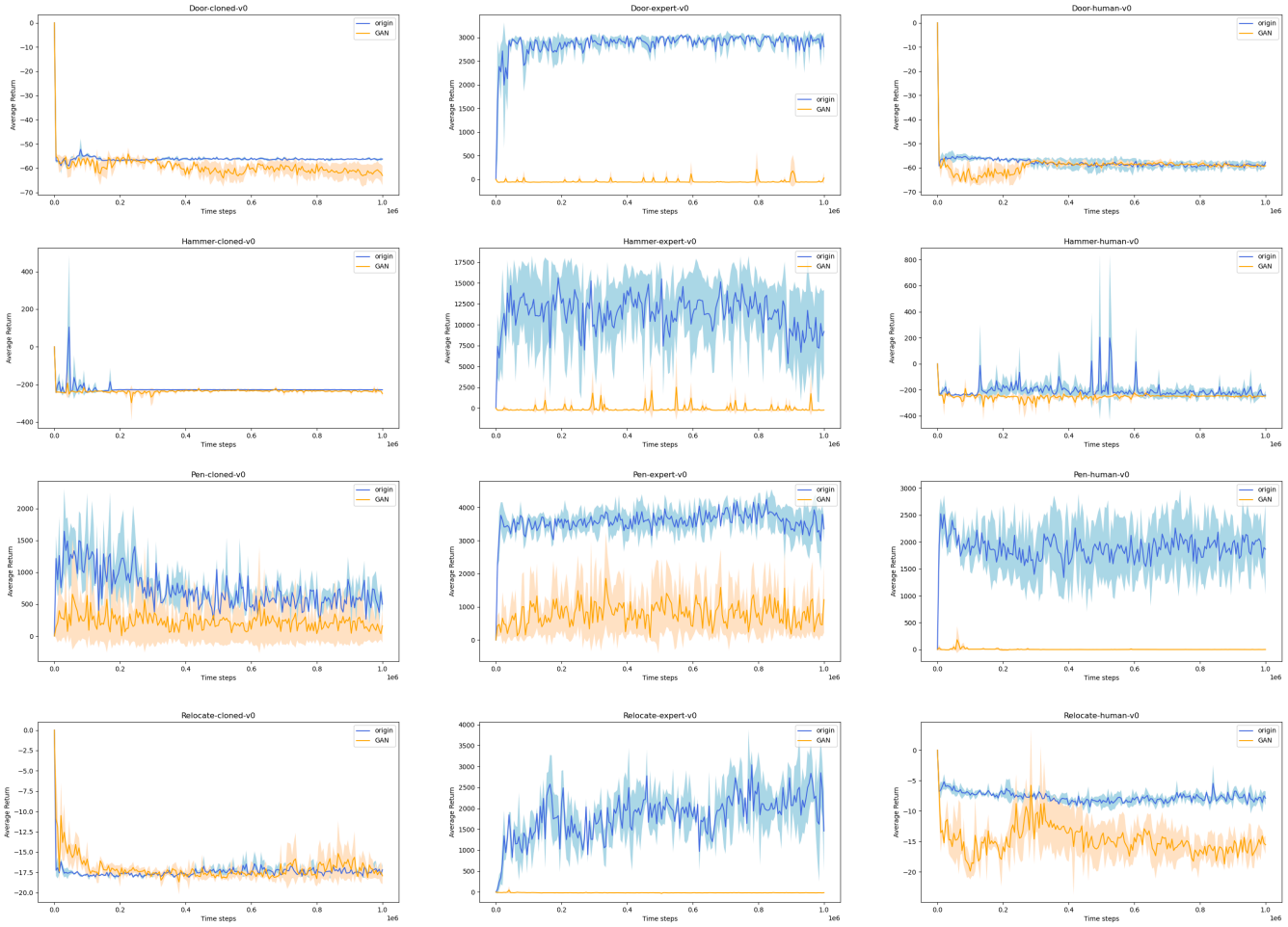

Antmaze-medium-diverse-v0
Antmaze-medium-play-v0
Hopper-medium-expert-v0
Hopper-expert-v0
Hopper-medium-v0
Hopper-medium-replay-v0
Hopper-random-v0
Walker2d-expert-v0
Walker2d-medium-v0
Walker2d-medium-expert-v0
Walker2d-medium-replay-v0
Walker2d-random-v0
Maze2d-umaze-v1
Maze2d-medium-v1
Maze2d-large-v1
origin
gamma09
Average Return
Time steps
1e6

Gamma = 0.9

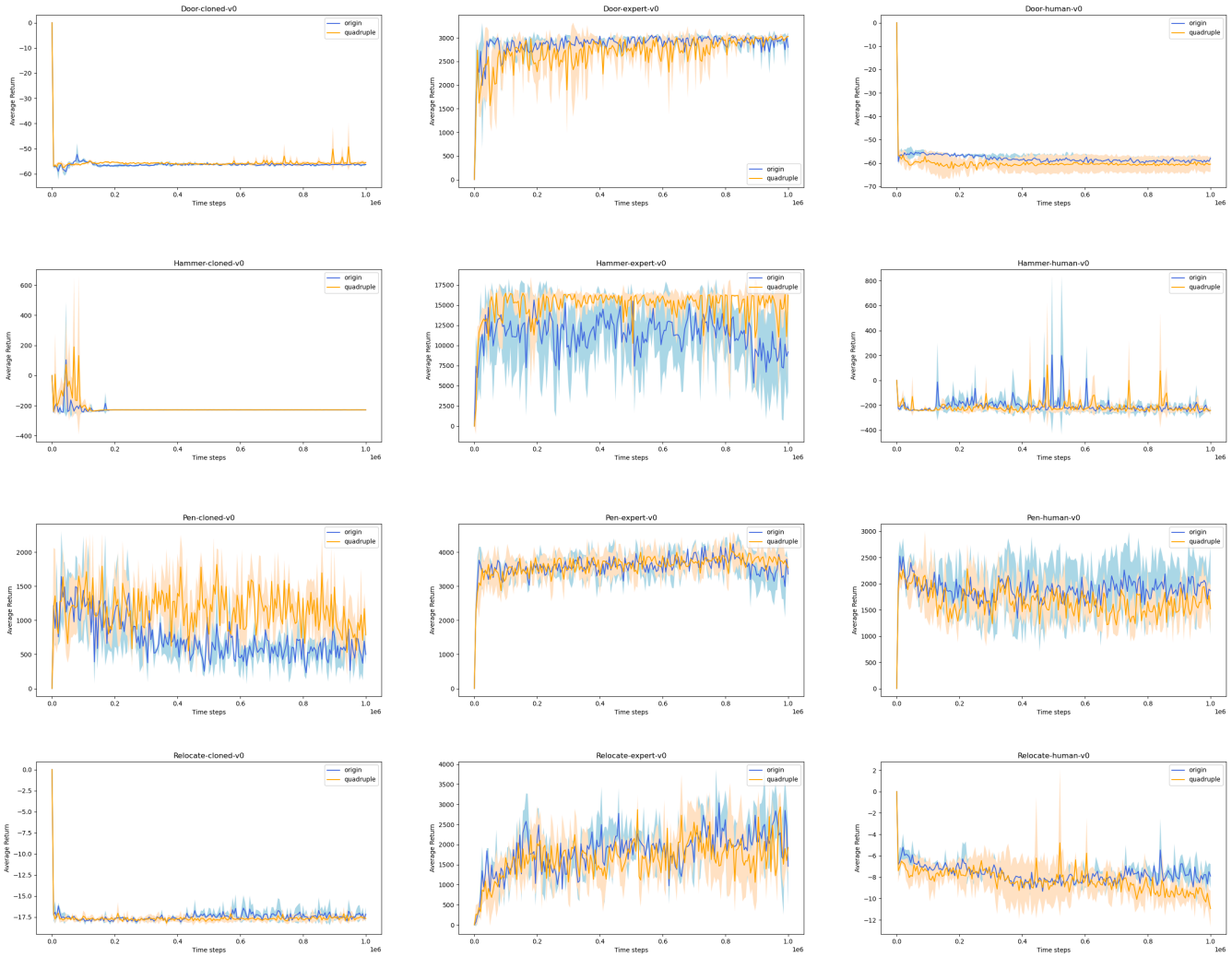



Batch size = 200

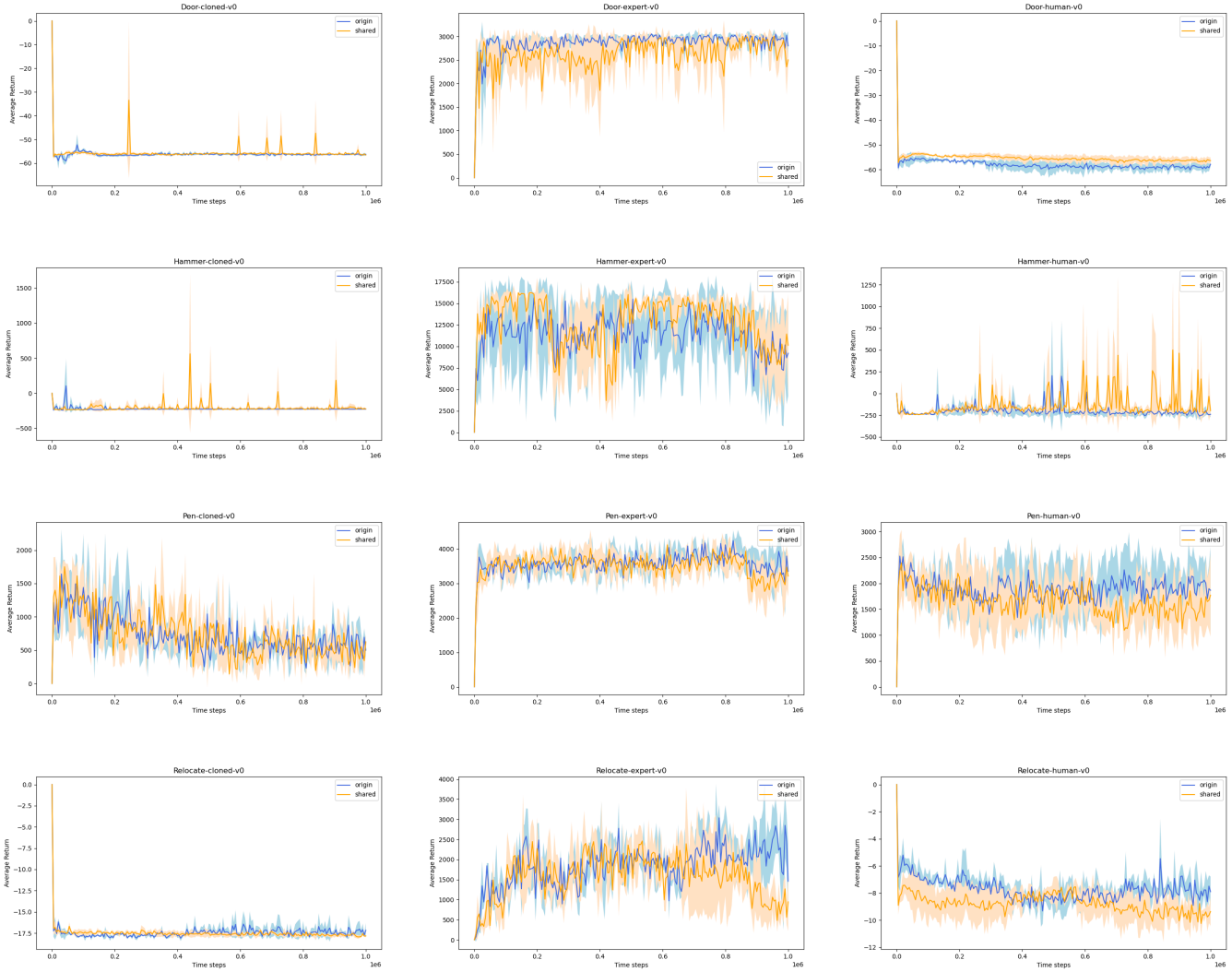
Only compare with baseline(step3):



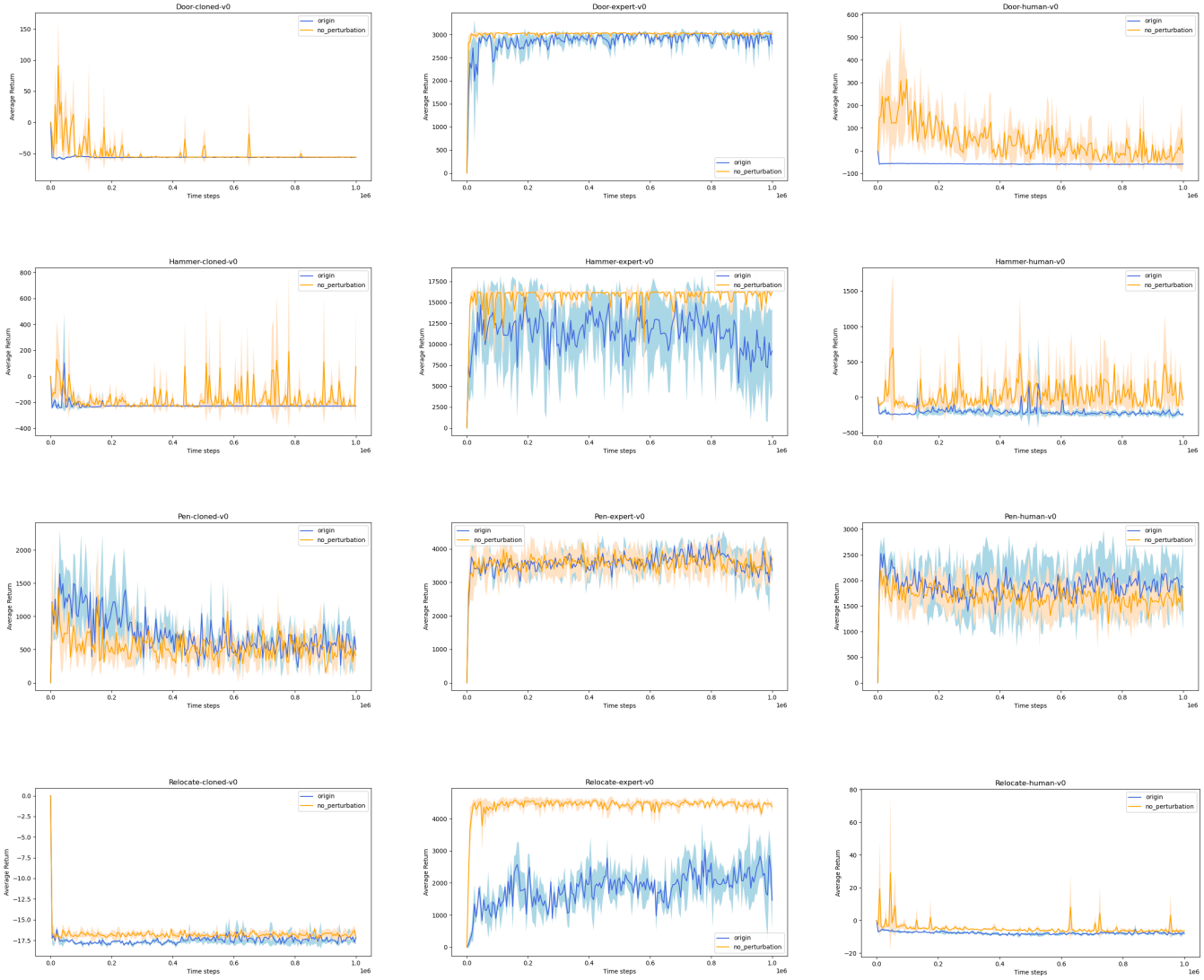

CGAN


Door-cloned-v0
Door-expert-v0
Door-human-v0
Hammer-cloned-v0
Hammer-expert-v0
Hammer-human-v0
Pen-cloned-v0
Pen-expert-v0
Pen-human-v0
Relocate-cloned-v0
Relocate-expert-v0
Relocate-human-v0
origin
quadruple
Average Return
Time steps
1e6

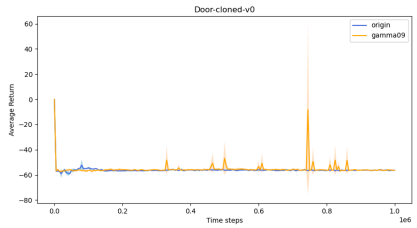
Quadruple

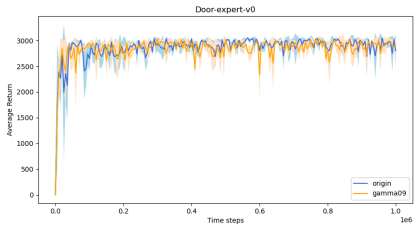

Door-cloned-v0
Door-expert-v0
Door-human-v0
Hammer-cloned-v0
Hammer-expert-v0
Hammer-human-v0
Pen-cloned-v0
Pen-expert-v0
Pen-human-v0
Relocate-cloned-v0
Relocate-expert-v0
Relocate-human-v0
origin
shared
Average Return
Time steps
1e6

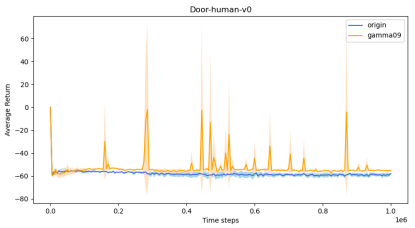
Shared

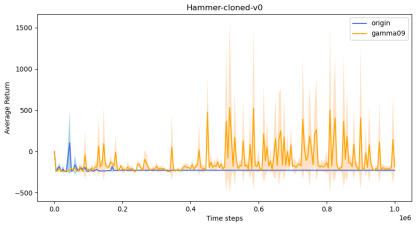


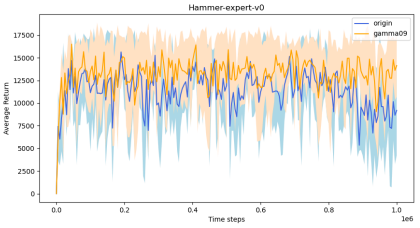

No perturbation

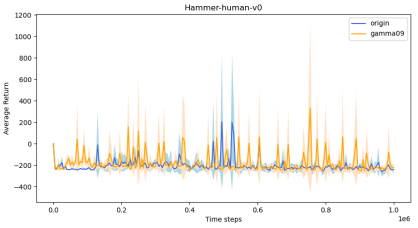

Door-cloned-v0
origin
gamma09
Average Return
Time steps
1e6

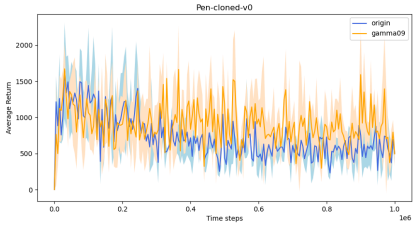

Door-expert-v0
origin
gamma09
Average Return
Time steps
1e6

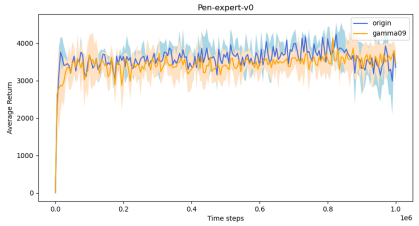

Door-human-v0
origin
gamma09
Average Return
Time steps
1e6

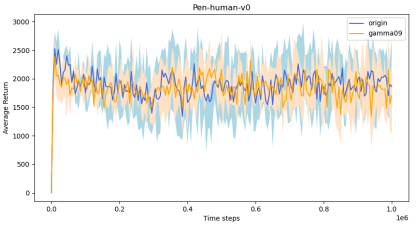

Hammer-cloned-v0
origin
gamma09
Average Return
Time steps
1e6

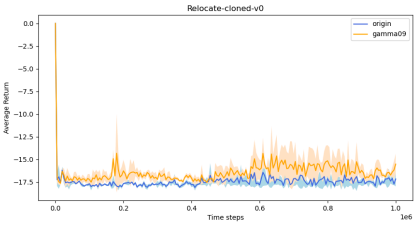

Hammer-expert-v0
origin
gamma09
Average Return
Time steps
1e6

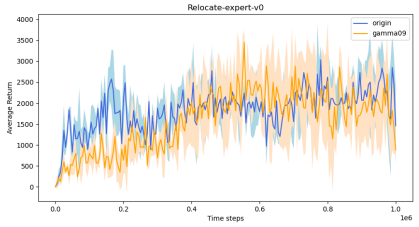

Hammer-human-v0
origin
gamma09
Average Return
Time steps
1e6

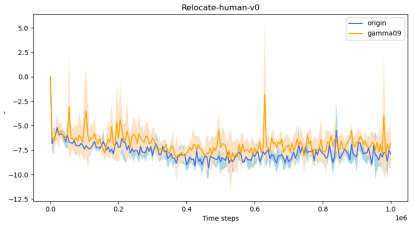

Pen-cloned-v0
origin
gamma09
Average Return
Time steps
1e6


Pen-expert-v0
origin
gamma09
Average Return
Time steps
1e6

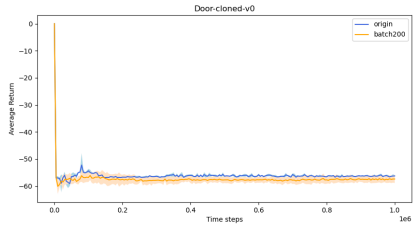

Pen-human-v0
origin
gamma09
Average Return
Time steps
1e6

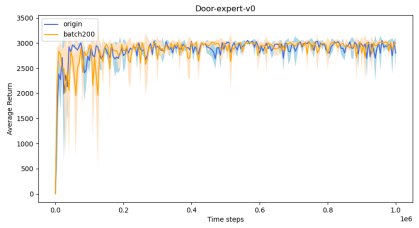

Relocate-cloned-v0
origin
gamma09
Average Return
Time steps
1e6

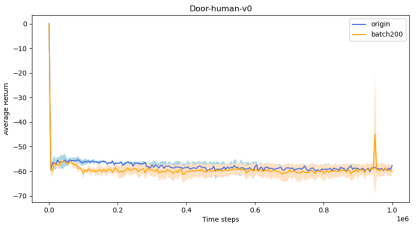

Relocate-expert-v0
origin
gamma09
Average Return
Time steps
1e6

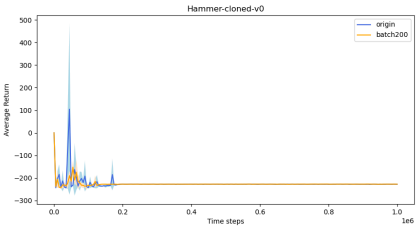

Relocate-human-v0
origin
gamma09
Average Return
Time steps
1e6

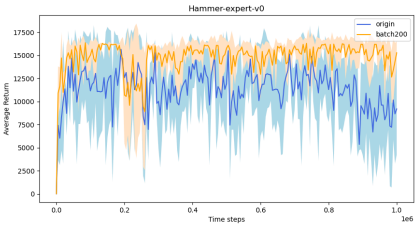
Gamma = 0.9

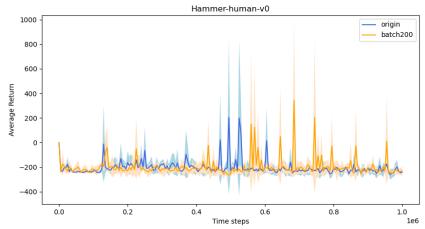


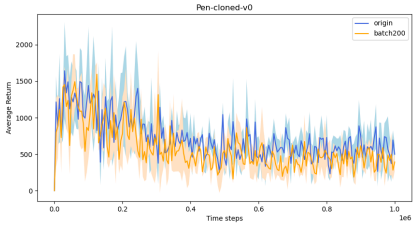



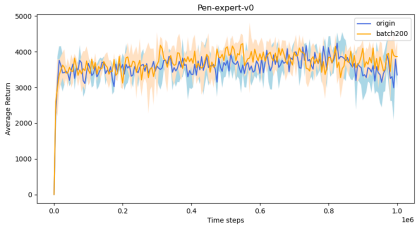



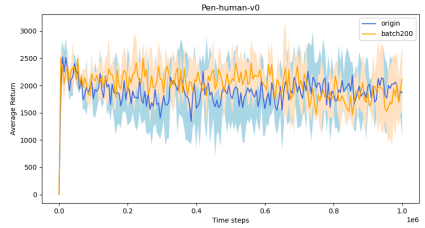



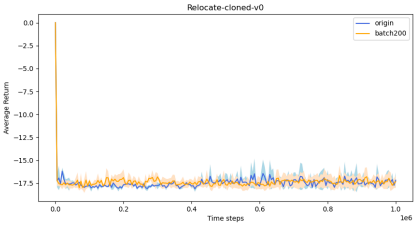



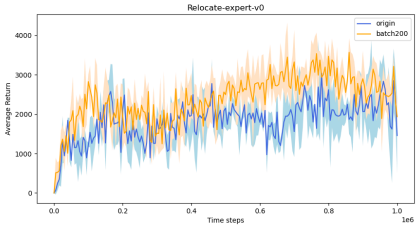



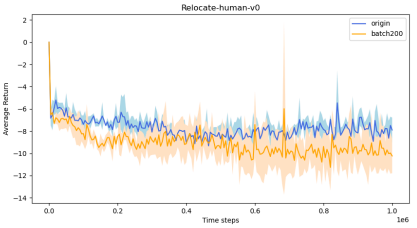



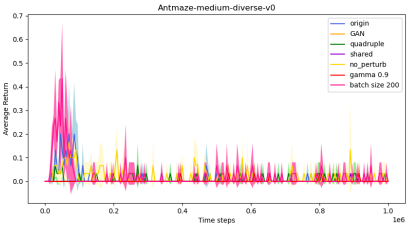



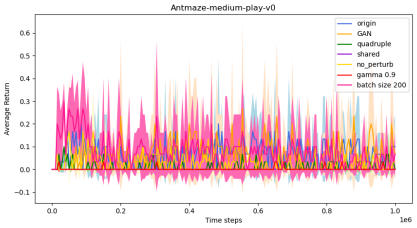



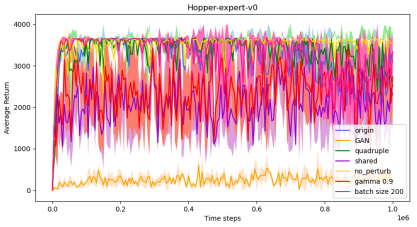



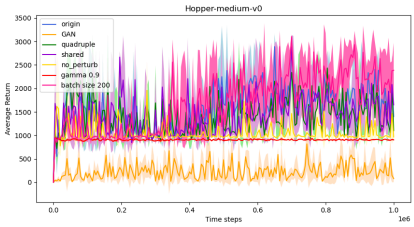

Batch size = 200

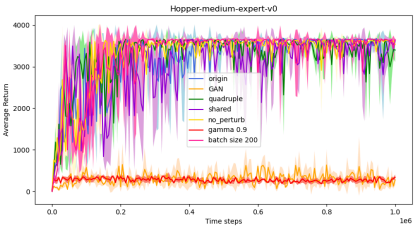
# Compare all method(step2):

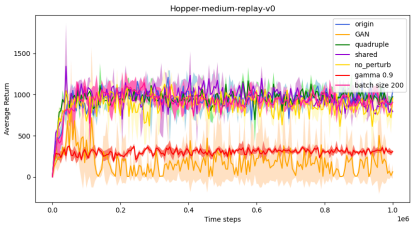


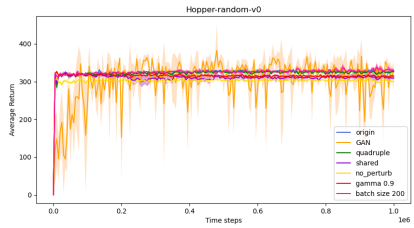



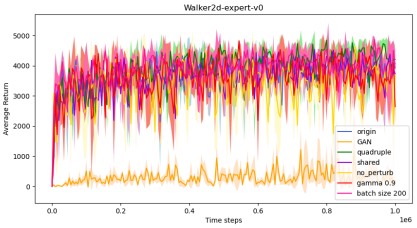



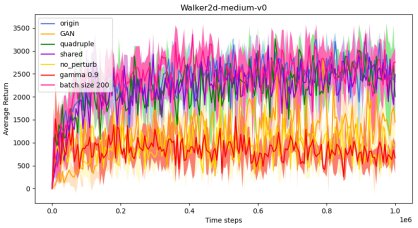



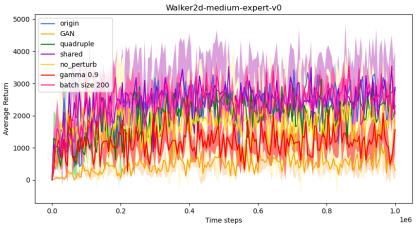



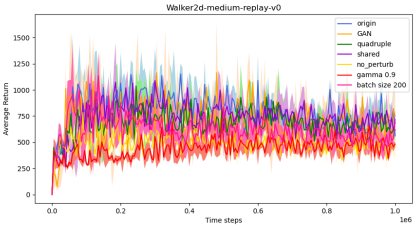



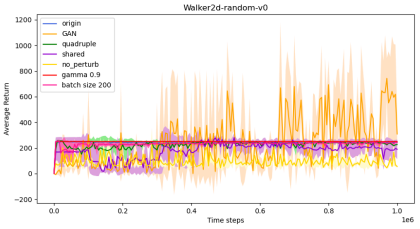



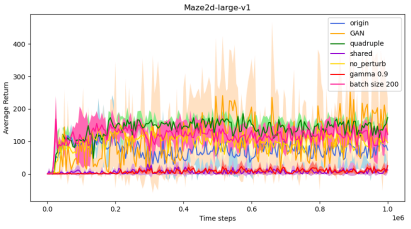



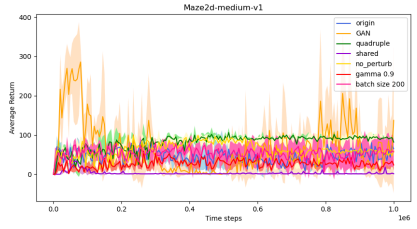



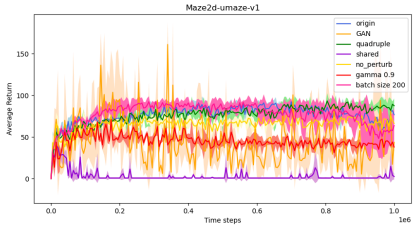



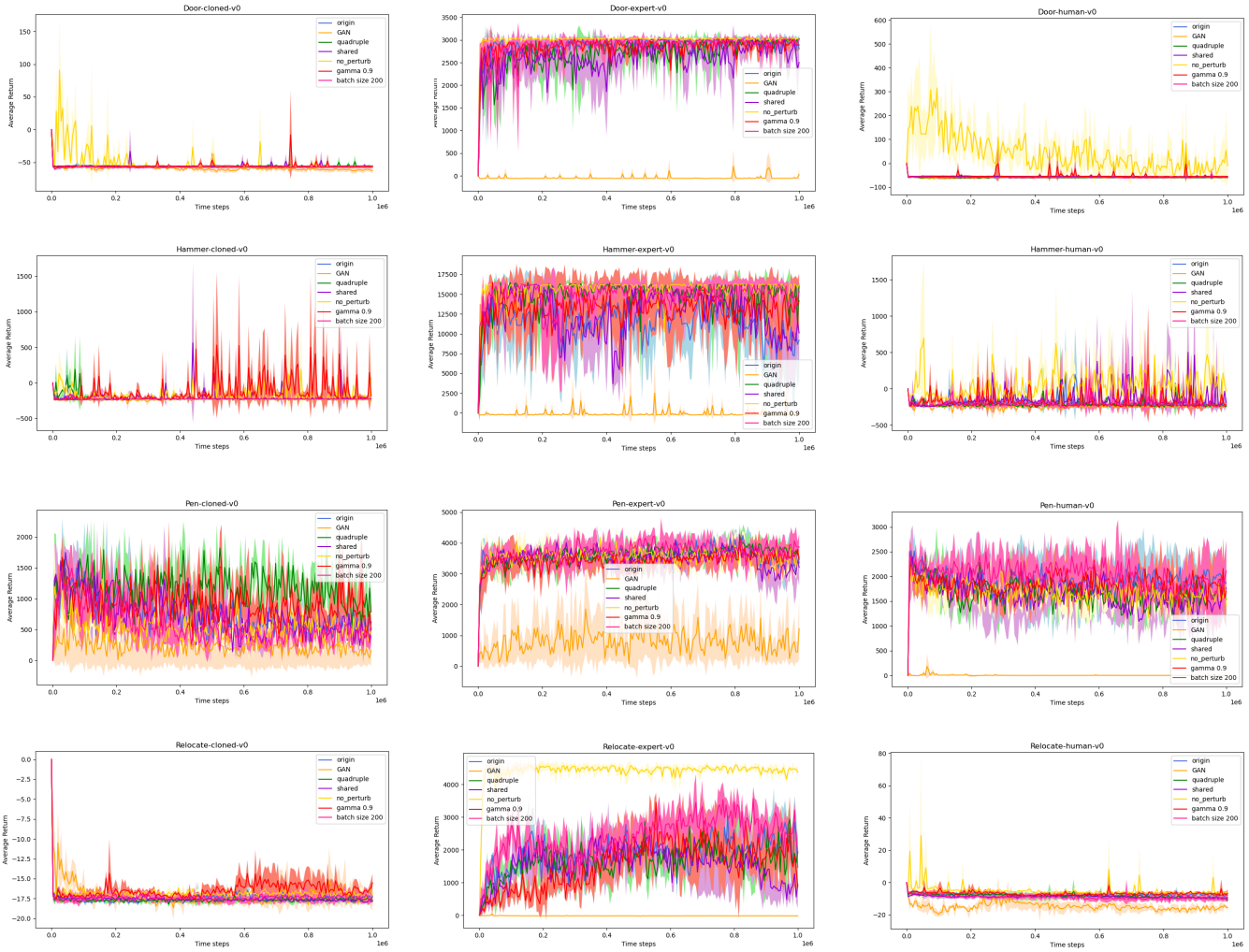



Compare all method(step3):




Here is our Github link: [RL_final_project]() It contains all our source code, results, and figures.