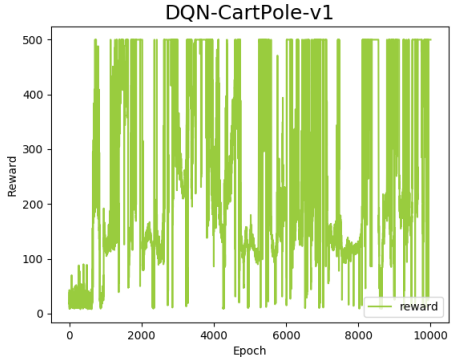
Label 6 :
Deep Q-Network and Deep Deterministic Policy Gradient

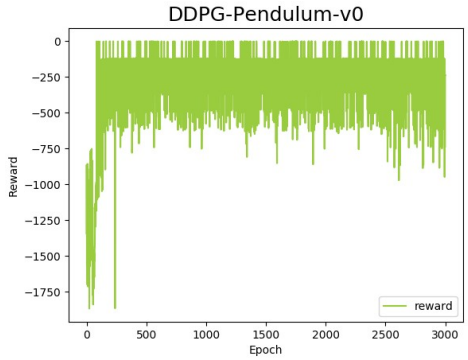
1. 訓練 DQN - CartPole-v1 的 10,000 epoch 時的分數與數據狀況

```
"e:9995,reward:500.0"
"e:9996,reward:500.0"
"e:9997,reward:500.0"
"e:9998,reward:500.0"
"e:9999,reward:500.0"
```



2. 訓練 DDPG - Pendulum-v0 的 3,000 epoch 時的分數與數據狀況

```
e:2995,Length:200,reward:-372.6622963674208
e:2996,Length:200,reward:-125.07216749615071
e:2997,Length:200,reward:-742.304199905537
e:2998,Length:200,reward:-233.448590134766
e:2999,Length:200,reward:-237.5385228384982
```



3. DQN 實作

a.DQN 網路架構

```
class DQN(nn.Module):
  def __init__(self, state_dim=4, action_dim=2, hidden_dim=24):
    super().__init__()
    self.net = nn.Sequential(
      nn.Linear(state_dim, hidden_dim),
      nn.ReLU(),

      nn.Linear(hidden_dim, hidden_dim),
      nn.ReLU(),

      nn.Linear(hidden_dim, action_dim),
    )

  def forward(self, x):
    return self.net(x)
```

b. optimizer 使用 Adam, 其中 betas 有額外改參數, criterion 的使用 MSE loss 才能符合公式。

```
optimizer = optim.Adam(behavior_net.parameters(), lr = args.lr, betas = (5e-1, 999e-3))
criterion = nn.MSELoss()
```

c. 選擇動作有機率是隨機以外, 因為輸出分別是往左或往右的機率, 所以選最大的並強轉 int 傳出去。

```
def select_action(epsilon, state, action_dim=2):
  """epsilon-greedy based on behavior network"""
  if random.random() < epsilon:
    return random.randint(0, action_dim-1)
  else:
    return int(behavior_net(state).max(0)[1])
```

d. 更新網路的部份，先將 replay buffer 的資料隨機的 sample 出來後，在用 behavior 網路輸出往左往右的機率，根據當時所選的動作產生 q_value，在使用不更新的 target 網路用 Q learning 的方式算出 q_next，最後在用 MSE 算誤差，其中很有意思的是有做 clip_grad_norm 來將一部分的 gradient 捨棄掉，來使訓練不容易收斂在一個不好的地方。

```
def update_behavior_network():
  def transitions_to_tensors(transitions, device=args.device):
    """convert a batch of transitions to tensors"""
    return (torch.Tensor(x).to(device) for x in zip(*transitions))

  # sample a minibatch of transitions
  transitions = memory.sample(args.batch_size)
  state, action, reward, next_state, done = transitions_to_tensors(transitions)

  q_value = behavior_net(state).gather(1, action.long())
  with torch.no_grad():
    q_next = reward + args.gamma * target_net(next_state).max(1)[0].view(args.batch_size, 1) * (1 - done)

  loss = criterion(q_value, q_next)

  # optimize
  optimizer.zero_grad()
  loss.backward()
  nn.utils.clip_grad_norm_(behavior_net.parameters(), 5)
  optimizer.step()
```

e. train 一開始會先等 replay buffer 存儲到一定的量後才開始讓DQN動作，每隔幾個步驟更新一次 behavior 網路，再每隔幾個步驟後直接完整更新 target 網路。

```
def train(env):
  print('Start Training')
  total_steps, epsilon = 0, 1.
  outputLog = ''

  for episode in range(args.episode):
    total_reward = 0
    state = env.reset()
    for t in itertools.count(start=1):
      # select action
      if total_steps < args.warmup:
        action = env.action_space.sample()
      else:
        state_tensor = torch.Tensor(state).to(args.device)
        action = select_action(epsilon, state_tensor)
        epsilon = max(epsilon * args.eps_decay, args.eps_min)
      # execute action
      next_state, reward, done, _ = env.step(action)
      # store transition
      memory.append(state, [action], [reward / 10], next_state, [int(done)])
      if total_steps >= args.warmup and total_steps % args.freq == 0:
        # update the behavior network
        update_behavior_network()
      if total_steps % args.target_freq == 0:
        target_net.load_state_dict(behavior_net.state_dict())

      state = next_state
      total_reward += reward
      total_steps += 1
      if done:
        outputLog += 'e:{},reward:{}\n'.format(episode, total_reward)
        print('Step: {}\tEpisode: {}\tTotal reward: {}\tEpsilon: {}'.format(
            total_steps, episode, total_reward, epsilon))
        break
  env.close()
  return outputLog
```

f. test 時就每一步驟都由 behavior 來掌控著。

```python
def test(env, render):
  print('Start Testing')
  epsilon = args.test_epsilon
  seeds = (20190813 + i for i in range(10))
  average = 0.0
  for seed in seeds:
    total_steps = 0
    total_reward = 0
    env.seed(seed)
    state = env.reset()

    for t in itertools.count(start=1):
      state_tensor = torch.Tensor(state).to(args.device)
      action = select_action(args.test_epsilon, state_tensor)
      next_state, reward, done, _ = env.step(action)

      state = next_state
      total_reward += reward
      total_steps += 1

      if done:
        average += total_reward
        print('Step:{},reward:{}'.format(total_steps, total_reward))
        break

  print("Average Reward:{}".format(average/10))
  env.close()
```

4. DDPG 實作

a. ActorNet 網路架構, hidden 維度調整成 512 x 256。

```
class ActorNet(nn.Module):
  def __init__(self, state_dim=3, action_dim=1, hidden_dim=(512, 256)):
    super().__init__()
    h1, h2 = hidden_dim
    self.net = nn.Sequential(
      nn.Linear(state_dim, h1, False),
      nn.ReLU(),

      nn.Linear(h1, h2, False),
      nn.ReLU(),

      nn.Linear(h2, action_dim, False),
      nn.Tanh()
    )

  def forward(self, x):
    return self.net(x)
```

b. CriticNet 網路架構, hidden 維度調整成 512 x 256, 以及多了一個 action_head, 將 action 維度拉高, 最後在進入 critic 時在與 cirtic_head 輸出串連起來。

```
class CriticNet(nn.Module):
  def __init__(self, state_dim=3, action_dim=1, hidden_dim=(512, 256)):
    super().__init__()
    h1, h2 = hidden_dim
    self.critic_head = nn.Sequential(
        nn.Linear(state_dim, h2, False),
        nn.ReLU(),
    )

    self.action_head = nn.Sequential(
      nn.Linear(action_dim, h2, False),
      nn.ReLU()
    )

    self.critic = nn.Sequential(
        nn.Linear(h1, h2, False),
        nn.ReLU(),
        nn.Linear(h2, action_dim, False)
    )

  def forward(self, x, action):
    x = self.critic_head(x)
    a = self.action_head(action)
    return self.critic(torch.cat([x, a], dim=1))
```

c. 參數設定改了 episode 的數量到 3,000 和 batch_size 為 256。

```
def parse_args():
  parser = argparse.ArgumentParser(description=__doc__)
  parser.add_argument('-d', '--device', default='cuda')
  # network
  parser.add_argument('-m', '--model', default='./DDPG.pt')
  parser.add_argument('--restore', action='store_false')
  # train
  parser.add_argument('-e', '--episode', default=3000, type=int)
  parser.add_argument('-c', '--capacity', default=10000, type=int)
  parser.add_argument('-bs', '--batch_size', default=256, type=int)
  parser.add_argument('--warmup', default=10000, type=int)
  parser.add_argument('--lra', default=1e-4, type=float)
  parser.add_argument('--lrc', default=1e-3, type=float)
  parser.add_argument('--gamma', default=.99, type=float)
  parser.add_argument('--tau', default=.001, type=float)
  # test
  parser.add_argument('--render', action='store_true')
  return parser.parse_args()
```

d. optimizer 選用 Adam 和 criterion 用 MSE。

```
actor_opt = optim.SGD(actor_net.parameters(), lr = args.lra)
critic_opt = optim.SGD(critic_net.parameters(), lr = args.lrc)
criterion = nn.MSELoss()
```

e. 選步的時候只有在 train 的時候會加上 OU, test 的時候不使用。

```
def select_action(state, low=-2, high=2, train = True):
  """based on the behavior (actor) network and exploration noise"""
  with torch.no_grad():
    action = actor_net(state)

  if train:
    action += random_process.sample()

  action = action.cpu()
  return max(min(action, high), low)
```

f. update netwark 的先更新 critic 網路, 符合更新公式, 但我有額外針對如果是 terminal 的時候, 只會剩下 reward, 讓公式比較符合 Q-Learning, actor 完全比照公式的方式, 其中有多乘上-1 來使梯度可以下降。

```
def update_behavior_network():
  def transitions_to_tensors(transitions, device=args.device):
    """convert a batch of transitions to tensors"""
    return (torch.Tensor(x).to(device) for x in zip(*transitions))

  # sample a minibatch of transitions
  transitions = memory.sample(args.batch_size)
  state, action, reward, state_next, done = transitions_to_tensors(transitions)

  ## update critic ##
  q_value = critic_net(state, action)
  with torch.no_grad():
    a_next = target_actor_net(state_next)
    q_next = reward + args.gamma * target_critic_net(state_next, a_next) * (1 - done)
  critic_loss = criterion(q_value, q_next)

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

  ## update actor ##
  action = actor_net(state)
  actor_loss = -1 * torch.mean(critic_net(state, action))

  # optimize actor
  actor_net.zero_grad()
  critic_net.zero_grad()
  actor_loss.backward()
  actor_opt.step()
```

g. 更新 target 網路的方式與公式相同。

```
def update_target_network(target_net, net):
  tau = args.tau
  for target, behavior in zip(target_net.parameters(), net.parameters()):
    target.data.copy_(tau * behavior.data + (1 - tau) * target.data)
```

h. test 的時候只要使用 actor 網路選動作即可。

```
def test(env, render):
  print('Start Testing')
  seeds = (20190813 + i for i in range(10))

  average = 0.0
  for seed in seeds:
    total_reward = 0
    total_steps = 0
    env.seed(seed)
    state = env.reset()

    for t in itertools.count(start=1):
      state_tensor = torch.Tensor(state).to(args.device)
      action = select_action(state_tensor, train=False)
      next_state, reward, done, _ = env.step(action)

      state = next_state
      total_reward += reward
      total_steps += 1

      if done:
        average += total_reward
        print('Step:{},Length:{},reward:{}'.format(total_steps, t, total_reward))
        break
  average /= 10
  print("Average Reward:{}".format(average))
  env.close()
```

5. 回答問題

a. DQN 與 DDPG 的不同在於, DDPG 實現了 Actor-Critic 的概念, 將 policy gradient 在 Actor 網路架構中去處理, Value based 則就在 Critic 網路架構中處理, 解決連續狀態問題, 基於 DQN 的 replay buffer 和 Target 網路的方式, 其中更新 Target 網路 DDPG 也不同於 DQN, 更新時會用 Soft 的方式來延遲更新。

b. discount factor 取決於對於未來的價值比例有多重, 當 discount factor 越大, 也就看下一個狀態所能得到的回饋就看得越高, 也更能夠知道選的這一動作是否能獲取更往好的目標前進。

c. epsilon-greedy 好處在於可以協助網路作到探索的功能, 如果一直都是只選最好的, 很容易會陷入到一個收斂瓶頸, 導致學習狀況沒有辦法有更好的表現, 那有機率去選擇到沒有看過得, 可以跳脫這瓶頸。

d. 因為 RL 不同 epoch 的 state 變畫會導致 Q-network 產生的結果分佈是非靜態的, 從而導致訓練出的結果不穩定, 那麼 target 網路就是為了解決這樣非靜態的問題, 來使訓練結果更加穩定。

e. Replay buffer 如果太小其實就近乎沒有 replay buffer 的意義, replay buffer 是為了要解決一直往某一個 gradient 的可能性, 也可說是模糊 model-based 和 model-free 的界線, 如果太大那要能學習到很好的狀態, 也要花費更多的時間去學習才行。

f. 因為 DDPG 是在處理連續環境下運作的, 那會與時序有很大的關聯, 那麼 OU 這個算法根據時序以及先前的 random 狀態去做相對性的調整, 會比 normal noise 來的更加平滑且穩定。

5. DQN - CartPole-v1 測試結果為平均 500.0

```
Step:500,reward:500.0
Step:500,reward:500.0
Step:500,reward:500.0
Step:500,reward:500.0
Step:500,reward:500.0
Step:500,reward:500.0
Step:500,reward:500.0
Step:500,reward:500.0
Step:500,reward:500.0
Step:500,reward:500.0
Average Reward: 500.0
```

6. DDPG – Pendulum-v0 測試結果為平均 -219.5

```
Step:200,Length:200,reward:-118.71331024169922
Step:200,Length:200,reward:-243.0944366455078
Step:200,Length:200,reward:-125.35185241699219
Step:200,Length:200,reward:-0.4142301678657532
Step:200,Length:200,reward:-125.93205261230469
Step:200,Length:200,reward:-347.29974365234375
Step:200,Length:200,reward:-508.76483154296875
Step:200,Length:200,reward:-125.37564849853516
Step:200,Length:200,reward:-474.3575439453125
Step:200,Length:200,reward:-125.69444274902344
Average Reward:-219.4998016357422
```