Deep Reinforcement Learning Homework 2

DQN implementation for Super Mario Bros.

Department: ISA PhD Program
Student ID: 112065802
Name: 蔡睿翊

Preface

In this assignment, I implemented Double DQN to play Super Mario Bros. The environment settings, as well as the processes of training and testing, are introduced in detail as follows.

Training

By following the instructions in the tutorial of Pytorch and modifying the code in the tutorial for performance enhancement, the implementation of training in this assignment includes the following parts.

Environment Preprocessing

Since there are some segments that are not related to the game states in the frames, we use the preprocessing modules including `SkipFrame`, `GrayScaleObservation`, `ResizeObservation`, and `FrameStack` (the last one is imported from `gym.wrappers`) provided by OpenAI Gym to make the representation of observed states concise.

```python
# Environment preprocessing
class SkipFrame(gym.Wrapper):
    def __init__(self, env, skip):
        """Return only every `skip`-th frame"""
        super().__init__(env)
        self._skip = skip

    def step(self, action):
        """Repeat action, and sum reward"""
        total_reward = 0.0
        for i in range(self._skip):
            # Accumulate reward and repeat the same action
            obs, reward, done, info = self.env.step(action)
            total_reward += reward
            if done:
                break
        return obs, total_reward, done, info
```

```python
class GrayScaleObservation(gym.ObservationWrapper):
    def __init__(self, env):
        super().__init__(env)
        obs_shape = self.observation_space.shape[:2]
        self.observation_space = Box(low=0, high=255, shape=obs_shape, dtype=np.uint8)

    def permute_orientation(self, observation):
        # permute [H, W, C] array to [C, H, W] tensor
        observation = np.transpose(observation, (2, 0, 1))
        observation = torch.tensor(observation.copy(), dtype=torch.float)
        return observation

    def observation(self, observation):
        observation = self.permute_orientation(observation)
        transform = T.Grayscale()
        observation = transform(observation)
        return observation
```

```python
class ResizeObservation(gym.ObservationWrapper):
    def __init__(self, env, shape):
        super().__init__(env)
        if isinstance(shape, int):
            self.shape = (shape, shape)
        else:
            self.shape = tuple(shape)

        obs_shape = self.shape + self.observation_space.shape[2:]
        self.observation_space = Box(low=0, high=255, shape=obs_shape, dtype=np.uint8)

    def observation(self, observation):
        transforms = T.Compose([T.Resize(self.shape, antialias=True), T.Normalize(0, 255)])
        observation = transforms(observation).squeeze(0)
        return observation
```

Before the training process starts, we apply the aforementioned wrappers to the environment to make the observed states in 4 channels, with each channel having frames resized as 84 by 84 pixels.

```python
    # Apply Wrappers to environment
    env = SkipFrame(env, skip=4)
    env = GrayScaleObservation(env)
    env = ResizeObservation(env, shape=84)
    env = FrameStack(env, num_stack=4)
```

Double DQN Architecture

The architecture of the dueling DQN is implemented as follows. The architecture includes online and target networks. Each of the networks is built with 3 convolution layers (with ReLU layers in between) followed by the flatten operation and 2 FC layers (with a ReLU layer between them).

```python
# DQN Architecture
class D2QN(nn.Module):
    """mini CNN structure: input -> (conv2d + relu) x 3 -> flatten -> (dense + relu) x 2 -> output"""

    def __init__(self, input_dim, output_dim):
        super().__init__()
        c, h, w = input_dim

        if h != 84:
            raise ValueError(f"Expecting input height: 84, got: {h}")
        if w != 84:
            raise ValueError(f"Expecting input width: 84, got: {w}")

        self.online = self.__build_cnn(c, output_dim)

        self.target = self.__build_cnn(c, output_dim)
        self.target.load_state_dict(self.online.state_dict())

        # Q_target parameters are frozen.
        for p in self.target.parameters():
            p.requires_grad = False

    def forward(self, input, model):
        if model == "online":
            return self.online(input)
        elif model == "target":
            return self.target(input)

    def __build_cnn(self, c, output_dim):
        return nn.Sequential(
            nn.Conv2d(in_channels=c, out_channels=32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(3136, 512),
            nn.ReLU(),
            nn.Linear(512, output_dim),
        )
```

Agent

The agent is implemented as follows.

Parameters

- Memory size: 100000
- Gamma: 0.9
- Batch size: 32
- Exploration decay in epsilon greedy: 0.95
- Learning rate: 0.0025

```python
# Agent
class Agent:
    def __init__(self, state_dim, action_dim):
        self.state_dim = state_dim
        self.action_dim = action_dim

        self.use_cuda = torch.cuda.is_available()

        # Mario's DNN to predict the most optimal action - we implement this in the Learn section
        self.net = D2QN(self.state_dim, self.action_dim).float()
        if self.use_cuda:
            self.net = self.net.to(device='cuda')

        self.exploration_rate = 1
        self.exploration_rate_decay = 0.95
        self.exploration_rate_min = 0.1
        self.curr_step = 0

        self.save_every = 5e5  # no. of experiences between saving Mario Net
        self.memory = deque(maxlen=100000)
        self.batch_size = 32

        self.gamma = 0.9

        self.optimizer = torch.optim.Adam(self.net.parameters(), lr=0.00025)
        self.loss_fn = torch.nn.SmoothL1Loss()

        self.burnin = 1e4  # min. experiences before training
        self.learn_every = 3  # no. of experiences between updates to Q_online
        self.sync_every = 1e4  # no. of experiences between Q_target & Q_online sync

```

act

This function chooses an action via epsilon-greedy strategy with an exploration decay.

```python
    def act(self, state):
        """Given a state, choose an epsilon-greedy action"""
        # EXPLORE
        if np.random.rand() < self.exploration_rate:
            action_idx = np.random.randint(self.action_dim)

        # EXPLOIT
        else:
            state = state[0].__array__() if isinstance(state, tuple) else state.__array__()
            state = torch.tensor(state).cuda().unsqueeze(0) if self.use_cuda else torch.tensor(state).unsqueeze(0)
            action_values = self.net(state, model="online")
            action_idx = torch.argmax(action_values, axis=1).item()

        # decrease exploration_rate
        self.exploration_rate *= self.exploration_rate_decay
        self.exploration_rate = max(self.exploration_rate_min, self.exploration_rate)

        # increment step
        self.curr_step += 1
        return action_idx

```

cache

This function adds the experiences (states, actions, rewards and corresponding next states) to the memory.

```python
    def cache(self, state, next_state, action, reward, done):
        """Add the experience to memory"""
        def first_if_tuple(x):
            return x[0] if isinstance(x, tuple) else x
        state = first_if_tuple(state).__array__()
        next_state = first_if_tuple(next_state).__array__()

        state = torch.FloatTensor(state).cuda() if self.use_cuda else torch.FloatTensor(state)
        next_state = torch.FloatTensor(next_state).cuda() if self.use_cuda else torch.FloatTensor(next_state)
        action = torch.LongTensor([action]).cuda() if self.use_cuda else torch.LongTensor([action])
        reward = torch.DoubleTensor([reward]).cuda() if self.use_cuda else torch.DoubleTensor([reward])
        done = torch.BoolTensor([done]).cuda() if self.use_cuda else torch.BoolTensor([done])

        self.memory.append((state, next_state, action, reward, done))
```

recall

This function samples a batch of experiences (states, actions, rewards and corresponding next states) from the memory.

```python
    def recall(self):
        """Sample experiences from memory"""
        batch = random.sample(self.memory, self.batch_size)
        state, next_state, action, reward, done = map(torch.stack, zip(*batch))
        return state, next_state, action.squeeze(), reward.squeeze(), done.squeeze()
```

learn

This function first samples a batch of experiences (states, actions, rewards and corresponding next states) from the memory with function `recall`, estimates the current Q-value with function `td_estimate`, calculates the Q-value in the next state with function `td_target`, and updates the online network with function `update_Q_online` by calculating the TD error between the current Q-value and the Q-value in the next state.

Besides, this function synchronizes the target network and the online network with function `sync_Q_target` per 10000 steps and saves the model with function `save_model` per 500000 steps.

```python
    def td_estimate(self, state, action):
        current_Q = self.net(state, model="online")[np.arange(0, self.batch_size), action]  # Q_online(s,a)
        return current_Q

    @torch.no_grad()
    def td_target(self, reward, next_state, done):
        next_state_Q = self.net(next_state, model="online")
        best_action = torch.argmax(next_state_Q, axis=1)
        next_Q = self.net(next_state, model="target")[np.arange(0, self.batch_size), best_action]
        return (reward + (1 - done.float()) * self.gamma * next_Q).float()

    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())

    def learn(self):
        """Update online action value (Q) function with a batch of experiences"""
        if self.curr_step % self.sync_every == 0:
            self.sync_Q_target()

        if self.curr_step % self.save_every == 0:
            self.save_model()

        if self.curr_step < self.burnin:
            return None, None

        if self.curr_step % self.learn_every != 0:
            return None, None

        # 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)

        return (td_est.mean().item(), loss)
```

The training process is implemented with 10000 episodes as follows. The metric logger is implemented to monitor the training process (Note the details of the metric logger is omitted here since the main focus of this report is to explain the implementation of double DQN algorithm).

```
    mario = Agent(state_dim=(4, 84, 84), action_dim=env.action_space.n)

    logger = MetricLogger(save_dir)

    episodes = 10000
    for e in range(episodes):

        state = env.reset()

        # Play the game!
        while True:

            # Run agent on the state
            action = mario.act(state)

            # Agent performs action
            next_state, reward, done, info = env.step(action)

            # Remember
            mario.cache(state, next_state, action, reward, done)

            # Learn
            q, loss = mario.learn()

            # Logging
            logger.log_step(reward, loss, q)

            # Update state
            state = next_state

            # Check if end of game
            if done or info["flag_get"]:
                break

        logger.log_episode()

        if (e % 20 == 0) or (e == episodes - 1):
            logger.record(episode=e, epsilon=mario.exploration_rate, step=mario.curr_step)
            mario.save_model()
```

Testing

Since the trained model is to be loaded, the agent is implemented in a different way from the training process as follows.

- The observed states are preprocessed manually with functions of OpenCV in the function `act`.
- The best action with the maximum probability predicted by the loaded model is chosen in each iteration.

```python
# Agent
class Agent:
    def __init__(self):
        # self.use_cuda = torch.cuda.is_available()
        # Mario's DNN to predict the most optimal action - we implement this in the Learn section
        self.net = D2QN((4, 84, 84), 12).float()
        self.load('112065802_hw2_data')
        # self.net.load_state_dict(torch.load('112065802_hw2_data'), strict=False)
        self.frames = deque(maxlen=4)
        self.curr_step = 0
        self.memory = deque(maxlen=100000)

    def act(self, observation):
        preprocess_obs = cv2.cvtColor(observation, cv2.COLOR_RGB2GRAY)
        preprocess_obs = cv2.resize(preprocess_obs, (84, 84), interpolation=cv2.INTER_AREA)
        while len(self.frames) < 3:
            self.frames.append(preprocess_obs)
        self.frames.append(preprocess_obs)
        preprocess_obs = torch.from_numpy(np.array(self.frames) / 255).float().unsqueeze(0)
        # observation = observation[0].__array__() if isinstance(observation, tuple) else observation.__array__()
        # observation = torch.from_numpy(observation.copy()).unsqueeze(0)
        # print(f'shape of observation: {observation.shape}')
        action_values = self.net(preprocess_obs, model="online")
        action_idx = torch.argmax(action_values, axis=1).item()

        # increment step
        self.curr_step += 1

        return action_idx

    def cache(self, state, next_state, action, reward, done):
        """Add the experience to memory"""
        def first_if_tuple(x):
            return x[0] if isinstance(x, tuple) else x
        state = first_if_tuple(state).__array__()
        next_state = first_if_tuple(next_state).__array__()

        state = torch.FloatTensor(state)
        next_state = torch.FloatTensor(next_state)
        action = torch.LongTensor([action])
        reward = torch.DoubleTensor([reward])
        done = torch.BoolTensor([done])

        self.memory.append((state, next_state, action, reward, done))

    def load(self, load_path):
        ckp = torch.load(load_path)
        exploration_rate = ckp.get('exploration_rate')
        state_dict = ckp.get('model')

        print(f"Loading model at {load_path} with exploration rate {exploration_rate}")
        self.net.load_state_dict(state_dict)
        self.exploration_rate = exploration_rate
```

The testing process is implemented with 50 episodes as follows (with each episode iteratively performing the action, cache, and calculation of the cumulative reward). The average reward is calculated after finishing 50 episodes.

```python
if __name__=='__main__':
    env = gym_super_mario_bros.make('SuperMarioBros-v0')
    env = JoypadSpace(env, COMPLEX_MOVEMENT)

    env.reset()
    mario = Agent()

    total_reward = 0
    episodes = 50

    for e in range(episodes):
        state = env.reset()
        episode_reward = 0
        print(f'Episode {e}')
        while True:
            # env.render()
            action = mario.act(state)
            next_state, reward, done, info = env.step(action)
            mario.cache(state, next_state, action, reward, done)
            episode_reward += reward

            state = next_state

            if done or info['flag_get']:
                break

        print(f'Episode reward in episode {e}: {episode_reward}')
        total_reward += episode_reward

    avg_reward = total_reward/50
    print(f'Average reward: {avg_reward}')
```