2024 AI - HW#5 Reinforcement Learning

生醫電資所 碩一 r12945040 郭思言

♦ Describe the Deep Q-Network

The DQN implementation uses a convolutional neural network to estimate Q-values for actions based on image input states. The agent interacts with the environment, stores experiences in a replay buffer, and periodically samples from this buffer to learn. The target network helps stabilize training by providing consistent Q-value estimates.

❖ Initialization `__init__`:

```python
class DQN:

    def __init__(
        self,
        state_dim,
        action_dim,
        lr=1e-4,
        epsilon=0.9,
        epsilon_min=0.05,
        gamma=0.99,
        batch_size=64,
        warmup_steps=5000,
        buffer_size=int(1e5),
        target_update_interval=10000,
    ):
        """
        DQN agent has four methods.

        - __init__() as usual
        - act() takes as input one state of np.ndarray and output actions by following epsilon-greedy policy.
        - process() method takes one transition as input and define what the agent do for each step.
        - learn() method samples a mini-batch from replay buffer and train q-network
        """
        self.action_dim = action_dim
        self.epsilon = epsilon
        self.gamma = gamma
        self.batch_size = batch_size
        self.warmup_steps = warmup_steps
        self.target_update_interval = target_update_interval
        self.network = PacmanActionCNN(state_dim[0], action_dim)
        self.target_network = PacmanActionCNN(state_dim[0], action_dim)
        self.target_network.load_state_dict(self.network.state_dict())
        self.optimizer = torch.optim.Adam(self.network.parameters(), lr)

        self.buffer = ReplayBuffer(state_dim, (1,), buffer_size)
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.network.to(self.device)
        self.target_network.to(self.device)

        self.total_steps = 0
        self.epsilon_decay = (epsilon - epsilon_min) / 1e6
```

1. Initializes the CNN model for action selection (`network`) and its target network (`target_network`).
2. Sets hyperparameters such as learning rate, epsilon for exploration, discount factor (gamma), batch size, etc.
3. Initializes the replay buffer and optimizer.
4. Sets device for computation (CPU/GPU) and initializes other variables like epsilon decay and total steps.

❖ Method `act`:

```python
    @torch.no_grad()
    def act(self, x, training=True):
        self.network.train(training)
        if training and ((np.random.rand() < self.epsilon) or (self.total_steps < self.warmup_steps)):
            # Random action
            action = np.random.randint(0, self.action_dim)
        else:
            # output actions by following epsilon-greedy policy
            x = torch.from_numpy(x).float().unsqueeze(0).to(self.device)

            # get q-values from network
            q_value = self.network(x)

            # get action with maximum q-value
            action = np.argmax(q_value.cpu().data.numpy())
        return action
```

1. Selects an action based on the epsilon-greedy policy.
2. Chooses a random action with probability epsilon, otherwise selects the action with the highest Q-value.

❖ Method `learn`:

```python
    def learn(self):
        # sample a mini-batch from replay buffer
        state, action, reward, next_state, terminated = map(
            lambda x: x.to(self.device), self.buffer.sample(self.batch_size)
        )

        # get q-values from network
        q_values = self.network(state)  # shape: [64, 9]
        next_q_values = self.target_network(next_state).detach()  # shape: [64, 9]

        # compute td_target
        # td_target: if terminated, only reward, otherwise reward + gamma * max(next_q)
        td_target = reward + (1 - terminated) * self.gamma * next_q_values.max(1)[0].view(-1, 1)  # shape: [64, 1]

        # gather the Q-values for the actions that were actually taken
        q_value = q_values.gather(1, action.long())  # use long to convert to int

        # compute loss with td_target and q-values
        loss = F.mse_loss(q_value, td_target)

        # initialize optimizer
        self.optimizer.zero_grad()
        # backpropagation
        loss.backward()
        nn.utils.clip_grad_norm_(self.network.parameters(), 10)

        # update network
        self.optimizer.step()
        return {"value_loss": loss.item()}  # return dictionary for logging
```

1. Samples a batch of experiences from the replay buffer.
2. Computes Q-values for the current states and next states using the network and target network, respectively.
3. Calculates the TD target, considering whether the episode has terminated.
4. Computes the loss between the Q-values of the taken actions and the TD target.
5. Performs backpropagation to update the network's weights.

❖ Method `process`:

```python
    def process(self, transition):

        result = {"value_loss": 0}
        self.total_steps += 1

        # update replay buffer
        # transition: (state, action, reward, next_state, terminated)
        self.buffer.update(*transition)

        if self.total_steps > self.warmup_steps:
            result = self.learn()

        if self.total_steps % self.target_update_interval == 0:
            # update target network
            "self.target_network.YOUR_CODE_HERE"
            self.target_network.load_state_dict(self.network.state_dict())

        self.epsilon -= self.epsilon_decay
        return result
```

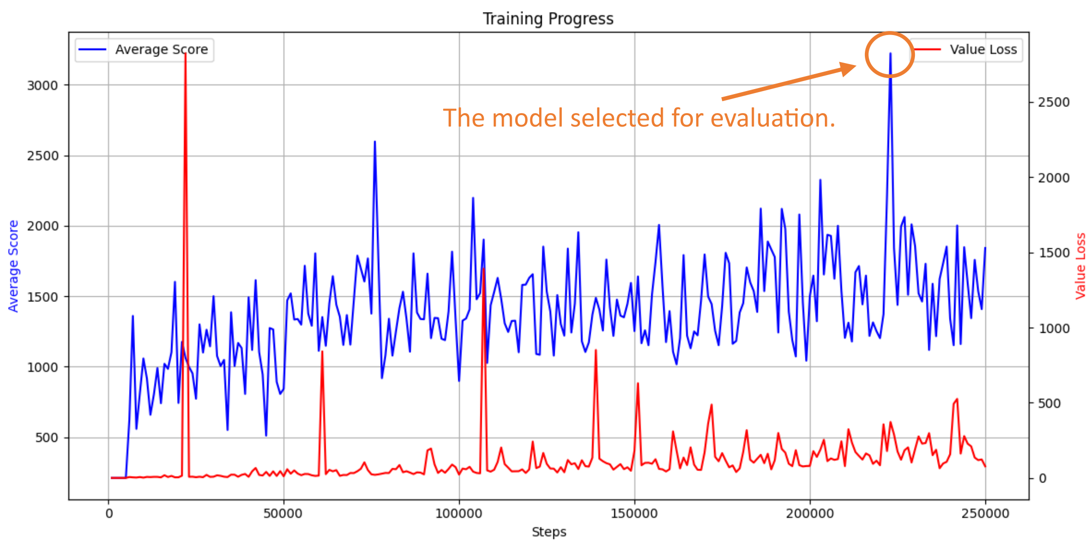
1. Processes a single transition, updating the replay buffer and performing learning if sufficient steps have been taken.
2. Updates the target network at regular intervals and decays epsilon for exploration.

◆ Describe the architecture of your PacmanActionCNN

```python
class PacmanActionCNN(nn.Module):

    def __init__(self, state_dim, action_dim):
        super(PacmanActionCNN, self).__init__()
        # build your own CNN model
        self.conv1 = nn.Conv2d(state_dim, 32, kernel_size=6, stride=2)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.bn2 = nn.BatchNorm2d(64)

        self.hidden1 = nn.Linear(64 * 19 * 19, 1024)
        self.hidden2 = nn.Linear(1024, 128)
        self.output = nn.Linear(128, action_dim)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        # print(x.shape)
        x = torch.flatten(x, start_dim=1)
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))
        return self.output(x)
```

- ❖ My PacmanActionCNN contains 2 2D Convolutional layers, both accompanied by Batch Normalization and ReLU activation. The first 2D Convolutional layer has a kernel size of 6 and a stride of 2, while the second 2D Convolutional layer has a kernel size of 4 and a stride of 2. These layers extract features from the input state, scaling the dimensions to 64x19x19.
- ❖ Following the convolutional layers, I apply a Multi-Layer Perceptron (MLP) with 2 hidden layers to transform the features into the final output dimension.

♦ Plot your training curve, including both loss and rewards



♦ Show screenshots from your evaluation video

("ALE/MsPacman-v5 has a total of three chances. Display the reward (score) each time you are caught.)