DQN-Pong Review

雅达利游戏智能体设计报告

康振飞，张润泽

云南大学-新技术专题 [1]

2024 年 12 月 20 日

1. 课题背景
2. 研究现状
3. 研究内容
4. 训练结果

强化学习基础研究

- 验证算法性能：

强化学习基础研究

- 验证算法性能：
  - Atari Pong 是一个经典的强化学习基准任务，训练智能体玩 Pong 可以验证 DQN 算法的性能和鲁棒性。通过在 Pong 上的实验，可以测试和改进 DQN 的超参数、网络结构、经验回放机制等。

强化学习基础研究

- 验证算法性能：
  - Atari Pong 是一个经典的强化学习基准任务，训练智能体玩 Pong 可以验证 DQN 算法的性能和鲁棒性。通过在 Pong 上的实验，可以测试和改进 DQN 的超参数、网络结构、经验回放机制等。
- 算法优化：

强化学习基础研究

- 验证算法性能：
  - Atari Pong 是一个经典的强化学习基准任务，训练智能体玩 Pong 可以验证 DQN 算法的性能和鲁棒性。通过在 Pong 上的实验，可以测试和改进 DQN 的超参数、网络结构、经验回放机制等。
- 算法优化：
  - 通过实验可以发现 DQN 的瓶颈（如样本效率低、训练不稳定等），从而为后续算法的改进提供方向。

游戏智能体应用

- 游戏 AI 研究：

游戏智能体应用

- 游戏 AI 研究：
  - 通过训练 Pong 智能体，可以研究游戏 AI 的行为模式、策略选择和学习能力。

游戏智能体应用

- 游戏 AI 研究：
  - 通过训练 Pong 智能体，可以研究游戏 AI 的行为模式、策略选择和学习能力。
- 辅助理解架构知识：

游戏智能体应用

- 游戏 AI 研究：
  - 通过训练 Pong 智能体，可以研究游戏 AI 的行为模式、策略选择和学习能力。
- 辅助理解架构知识：
  - 通过调整超参和修改局部代码，可以进一步理解强化学习的概念（如状态、动作、奖励、Q 值等）。

1. 课题背景
2. 研究现状
3. 研究内容
4. 训练结果

基础网络架构[1]

- 输入层：Input 是一个 84*84*4 的图像。使用了最近的 4 帧画面，每帧 84*84 像素。
- 卷积层：
  - 1st：32 个 8*8 滤波器，步幅 4。从输入图像中提取基础特征，如边缘和简单形状。
  - 2nd：64 个 4*4 滤波器，步幅 2。这一层提取更复杂的特征，步幅减小以获得更详细的特征图。
  - 3rd：64 个 3*3 滤波器，步幅 1。这一层进一步提取精细特征，步幅更小以保持特征的细节。
- 全链接层：
  - 1st：nn.Linear(64*7*7, 512)。输入特征数：3136，输出特征数：512（神经元数）。
  - ReLU：负值置 0，正值不变。
  - 2nd：nn.Linear(512, action-space.n)。输入特征数：512（接上层），输出至动作空间。

[1] **模型参考** doi:10.1038/nature14236

1 课题背景

2 研究现状

3 研究内容

- 网络剖析
- 训练及超参设置

4 训练结果

1. 课题背景
2. 研究现状
3. 研究内容
   - 网络剖析
   - 训练及超参设置
4. 训练结果

DQN 设计[2]

```python
class DQN(nn.Module):
    def __init__(self,
                 observation_space: spaces.Box,
                 action_space: spaces.Discrete):
        super().__init__()
        assert type(
            observation_space) == spaces.Box, 'observation_space must be of type Box'
        assert len(
            observation_space.shape) == 3, 'observation space must have the form channels x width x height'
        assert type(
            action_space) == spaces.Discrete, 'action_space must be of type Discrete'

        self.conv = nn.Sequential(
            nn.Conv2d(in_channels=observation_space.shape[0], 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()
        )

        self.fc = nn.Sequential(
            nn.Linear(in_features=64*7*7 , out_features=512),
            nn.ReLU(),
            nn.Linear(in_features=512, out_features=action_space.n)
        )

    def forward(self, x):
        conv_out = self.conv(x).view(x.size()[0],-1)
        return self.fc(conv_out)
```

图 1: DQN Code

[2]完整代码：github.com/godsboy404/DQN-Pong

轮次更新策略 & 奖励处理

```python
class EpisodicLifeEnv(gym.Wrapper):
    def __init__(self, env):
        """视 end-of-life 为终止，但游戏结束时才真正重置环境。
        """
        gym.Wrapper.__init__(self, env)
        self.lives = 0
        self.was_real_done = True

    def step(self, action):
        obs, reward, done, info = self.env.step(action)
        self.was_real_done = done
        # check current lives, make loss of life terminal,
        # then update lives to handle bonus lives
        lives = self.env.unwrapped.ale.lives()
        if lives < self.lives and lives > 0:
            # for Qbert sometimes we stay in lives == 0 condtion for a few frames
            # so its important to keep lives > 0, so that we only reset once
            # the environment advertises done.
            done = True
        self.lives = lives
        return obs, reward, done, info

    def reset(self, **kwargs):
        """Reset only when lives are exhausted.
        This way all states are still reachable even though lives are episodic,
        and the learner need not know about any of this behind-the-scenes.
        """
        if self.was_real_done:
            obs = self.env.reset(**kwargs)
        else:
            # no-op step to advance from terminal/lost life state
            obs, _, _, _ = self.env.step(0)
        self.lives = self.env.unwrapped.ale.lives()
        return obs

class ClipRewardEnv(gym.RewardWrapper):
    def __init__(self, env):
        gym.RewardWrapper.__init__(self, env)

    def reward(self, reward):
        """Bin reward to {+1, 0, -1} by its sign."""
        return np.sign(reward)
```

图 2: LifeCycle & Rewards

回放缓冲

```python
class ReplayBuffer:

    def __init__(self, size):
        self._storage = []
        self._maxsize = size
        self._next_idx = 0

    def __len__(self):
        return len(self._storage)

    def add(self, state, action, reward, next_state, done):
        """
        Add a transition to the buffer. Old transitions will be overwritten if the buffer is full.
        :param state: 初态
        :param action: 当前行为
        :param reward: 行为对应奖励
        :param next_state: 次态
        """
        data = (state, action, reward, next_state, done)

        if self._next_idx >= len(self._storage):
            self._storage.append(data)
        else:
            self._storage[self._next_idx] = data
        self._next_idx = (self._next_idx + 1) % self._maxsize

    def _encode_sample(self, indices):
        states, actions, rewards, next_states, dones = [], [], [], [], []
        for i in indices:
            data = self._storage[i]
            state, action, reward, next_state, done = data
            states.append(np.array(state, copy=False))
            actions.append(action)
            rewards.append(reward)
            next_states.append(np.array(next_state, copy=False))
            dones.append(done)
        return np.array(states), np.array(actions), np.array(rewards), np.array(next_states), np.array(dones)

    def sample(self, batch_size):
        indices = np.random.randint(0, len(self._storage) - 1, size=batch_size)
        return self._encode_sample(indices)
```

图 3: Replay Buffer

1 课题背景

2 研究现状

3 研究内容
- 网络剖析
- 训练及超参设置

4 训练结果

超参

```python
hyper_params = {
        "env": "PongNoFrameskip-v4",  # name of the game
        "replay-buffer-size": int(5e3),  # replay buffer size
        "learning-rate": 1e-4,  # learning rate for Adam optimizer
        "discount-factor": 0.99,  # discount factor
        "num-steps": int(1e6),  # total number of steps to run the environment for
        "batch-size": 32,  # number of transitions to optimize at the same time
        "learning-starts": 10000,  # number of steps before learning starts
        "learning-freq": 1,  # number of iterations between every optimization step
        "use-double-dqn": True,  # use double deep Q-learning
        "target-update-freq": 1000,  # number of iterations between every target network update
        "eps-start": eps_start,  # e-greedy start threshold
        "eps-end": 0.01,  # e-greedy end threshold
        "eps-fraction": 0.1,  # fraction of num-steps
        "print-freq": 10
    }
```

图 4: params

训练脚本

```python
state = env.reset()
eps_timesteps = hyper_params["eps-fraction"] * \
    float(hyper_params["num-steps"])
episode_rewards = [0.0]

for t in range(hyper_params["num-steps"]):
    fraction = min(1.0, float(t) / eps_timesteps)
    eps_threshold = hyper_params["eps-start"] + fraction * \
        (hyper_params["eps-end"] - hyper_params["eps-start"])
    sample = random.random()

    if(sample > eps_threshold):
        # Exploit
        action = agent.act(state)
    else:
        # Explore
        action = env.action_space.sample()

    next_state, reward, done, info = env.step(action)
    agent.memory.add(state, action, reward, next_state, float(done))
    state = next_state

    episode_rewards[-1] += reward
    if done:
        state = env.reset()
        episode_rewards.append(0.0)

    if t > hyper_params["learning-starts"] and t % hyper_params["learning-freq"] == 0:
        agent.optimise_td_loss()

    if t > hyper_params["learning-starts"] and t % hyper_params["target-update-freq"] == 0:
        agent.update_target_network()

    num_episodes = len(episode_rewards)

    if done and hyper_params["print-freq"] is not None and len(episode_rewards) % hyper_params[
            "print-freq"] == 0:
        mean_100ep_reward = round(np.mean(episode_rewards[-101:-1]), 1)
        print("********************************************************")
        print("steps: {}".format(t))
        print("episodes: {}".format(num_episodes))
        print("mean 100 episode reward: {}".format(mean_100ep_reward))
        print("% time spent exploring: {}".format(int(100 * eps_threshold)))
        print("********************************************************")
        torch.save(agent.policy_network.state_dict(), f'checkpoint.pth')
        np.savetxt('results/rewards_per_episode.csv', episode_rewards,
                   delimiter=',', fmt='%1.3f')
```

图 5: train

优化误差

```python
def optimise_td_loss(self):
    """
    Optimise the TD-error over a single minibatch of transitions
    :return: the loss
    """
    device = self.device

    states, actions, rewards, next_states, dones = self.memory.sample(self.batch_size)
    states = np.array(states) / 255.0
    next_states = np.array(next_states) / 255.0
    states = torch.from_numpy(states).float().to(device)
    actions = torch.from_numpy(actions).long().to(device)
    rewards = torch.from_numpy(rewards).float().to(device)
    next_states = torch.from_numpy(next_states).float().to(device)
    dones = torch.from_numpy(dones).float().to(device)

    with torch.no_grad():
        if self.use_double_dqn:
            _, max_next_action = self.policy_network(next_states).max(1)
            max_next_q_values = self.target_network(next_states).gather(1, max_next_action.unsqueeze(1)).squeeze()
        else:
            next_q_values = self.target_network(next_states)
            max_next_q_values, _ = next_q_values.max(1)
        target_q_values = rewards + (1 - dones) * self.gamma * max_next_q_values

    input_q_values = self.policy_network(states)
    input_q_values = input_q_values.gather(1, actions.unsqueeze(1)).squeeze()

    loss = F.smooth_l1_loss(input_q_values, target_q_values)

    self.optimiser.zero_grad()
    loss.backward()
    self.optimiser.step()
    del states
    del next_states
    return loss.item()
```

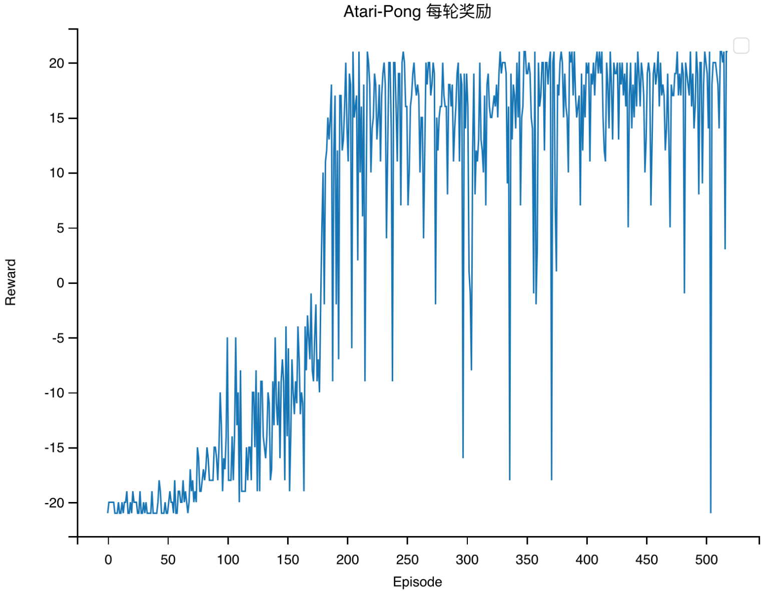
图 6: loss

采取动作 & 更新陪练网络

```python
def act(self, state: np.ndarray):
    """
    Select an action greedily from the Q-network given the state
    :param state: the current state
    :return: the action to take
    """
    device = self.device
    state = np.array(state) / 255.0
    state = torch.from_numpy(state).float().unsqueeze(0).to(device)
    with torch.no_grad():
        q_values = self.policy_network(state)
        _, action = q_values.max(1)
        return action.item()

def update_target_network(self):
    """
    Update the target Q-network by copying the weights from the current Q-network
    """
    self.target_network.load_state_dict(self.policy_network.state_dict())
```

图 7: action & network-updating


Atari-Pong 每轮奖励
Reward
Episode
20
15
10
5
0
-5
-10
-15
-20
0
50
100
150
200
250
300
350
400
450
500

Thanks for Watching!