Homework 2: Autonomous Agents

Josyula Gopala Krishna

1 INTRODUCTION

The problem of HW2 is to establish a sequence of actions for agents to reach a target in a grid world of 5x10, where the agents start at random positions on the grid and have to reach the target while maximizing their reward. The reward structure for the agents is they receive a reward of 20 for capturing a target, and there is a reward of -1 for every time step an agent is in this environment without capturing a target.

2 METHOD

For the assignment, I have experimented with Deep Q-Learning from the paper[1] with all three subproblems in the assignment. Further, I have used two variants of the DQN for experimentation and have coded the environment using matplotlib. The neural network for DQN has been written from scratch and implemented with the PyTorch learning library[2].

2.1 DQN

The RL problem is an Markov Decision Process(MDP) with a sequence of states and actions $s_t = x_1, a_1, x_2, \ldots, a_{t-1}, x_t$ and learn the game strategies depending on these sequences. The goal of the agent is to interact with the environment and select actions in a way that maximizes the future rewards, which are discounted by a factor γ per timestep $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ where T is the timestep at which game terminates.

An optimal action-value is $Q^*(s, a)$ is the maximum expected return achievable by following any strategy, after seeing some sequence 's' and then taking some actions 'a',

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\left[R_t \mid s_t = s, a_t = a, \pi\right]$$

where π is a policy mapping sequences to actions(or distributions over actions)

The optimal action-value function follows Bellman equation, which based on the postulate that an optimal strategy is to select optimal action a i.e. action which maximises the Q(s,a) at every state s'. this allows the $Q^*(sa,)$ to be written as

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*\left(s', a'\right) \mid s, a\right]$$

Therefore the idea behind the DQN is to estimate the action-value function by using the bellman equation as a iterative update

$$Q_{i+1}(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*\left(s', a'\right) \mid s, a\right]$$

DQN uses a function approximator $Q(s, a, \theta) \approx Q^*(s, a)$ Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i\left(\theta_i\right) = \mathbb{E}_{s,a \sim \rho(\cdot)}\left[\left(y_i - Q\left(s, a; \theta_i\right)\right)^2\right]$$

, , , .

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 1: Training DQN

where

$$y_i = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q\left(s', a'; \theta_{i-1}\right) \mid s, a\right]$$

is the target for iteration i, and the gradient is taken as

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_{i-1}\right) - Q\left(s, a; \theta_i\right)\right)\right.$$

$\nabla_{\theta_i} Q\left(s, a; \theta_i\right)$

After the DQN has been trained for several iterations, it can be used to estimate the actions which give maximum value at each state.

2.2 Training

For all the experiments in the report $\epsilon = 0.1$ and for training and $\epsilon = 0.5$ for testing. the DQN a replay memory buffer is used to store the sequence of (state, action, reward, next state) tuples to perform batch gradient descent the algorithm is shown in Fig.1 All parameters (pn is problem n):

$$\begin{aligned} \gamma &= 0.99 \\ learning\_rate &= 1e-4 \\ batch\_size &= 32 \\ time\_steps &= 600p1/10000(p2, p3) \\ replay\_buffer &= 10000p1/1000(p2, p3) \end{aligned}$$

3 EXPERIMENTS AND RESULTS

3.1 Problem 1

In this problem, it is assumed there's a single agent and single target, for this problem, I have used the state as the image from the environment for the DQN the structure of DQN is:

$$(conv1) : Conv2d(3, 32, kernel_size = (8, 8), stride = (4, 4))$$
$$(conv2) : Conv2d(32, 64, kernel_size = (4, 4), stride = (2, 2))$$
$$(conv3) : Conv2d(64, 64, kernel_size = (3, 3), stride = (1, 1))$$
$$(fc4) : Linear(in_features = 36864, out_features = 512, bias = True)$$
$$(fc5) : Linear(in_features = 512, out_features = 4, bias = True)$$

state is (320, 160, 3) sized RGB image to the DQN, the first three layers are convolution layers followed by two full connected layers and the output from fc5 is of shape (4,1), which represents the Q-values of actions in a state s, the replay memory buffer is of size 10000 and batch size for training is 32 and trained using the approach in Section 2 1. Reward for capturing target 20, and every step taken reward is -1.

Agent appears to have achieved optimal reward after 200 steps of training however, I continued training the agent for 500 steps the plot of rewards obtained during training is shown in 2

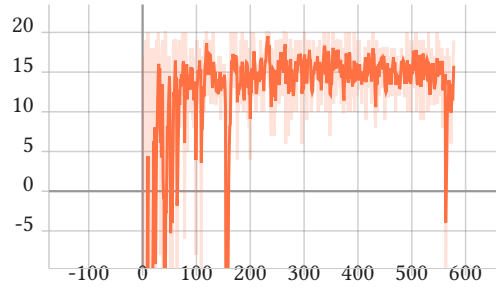

Figure 2: Timestep-vs-Rewards for problem 1

An agent starting at a random position and its path taken is shown in Fig 3. Agent always seems to finish the game in optimal number of moves.

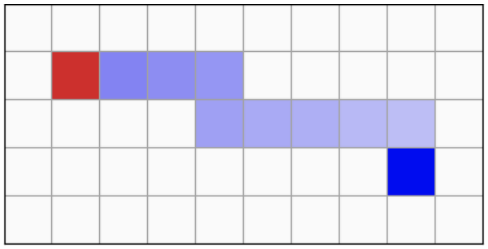

Figure 3: Random initial position and agent path to target for Problem 1

3.2 Problem 2

For this problem, the grid has two agents, and they both use two DQN networks trained simultaneously, using the reward scheme of 20 for capturing the target and -1 for every step taken.

Since parallel training of two DQN's with images is memory intensive, an alternative state representation has been chosen for this problem. state is the tuple $(agent_1position, agent_2position)$, the DQN architecture is

$$(fc1) : Linear(in_features = 4, out_features = 64, bias = True)$$
$$(fc2) : Linear(in_features = 64, out_features = 64, bias = True)$$
$$(fc3) : Linear(in_features = 64, out_features = 4, bias = True)$$

Which is composed of 3 fully connected layers with a ReLU activation function between each layer as non-linearity.The final layer has four output values which represent the Q-values for actions 'up', 'down', 'left', 'right'

The replay memory size is 1000 and the batch size is 32, the training is for 10,000 timesteps.

The rewards for each agent are shown below:

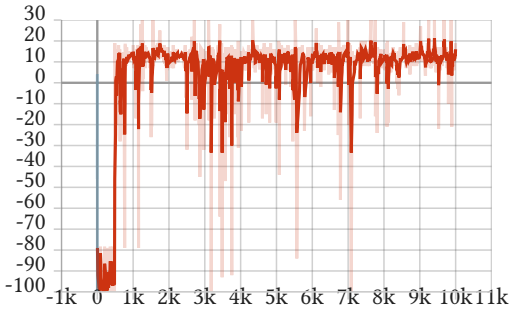

Figure 4: Timestep-vs-Rewards for Agent 1, problem 2

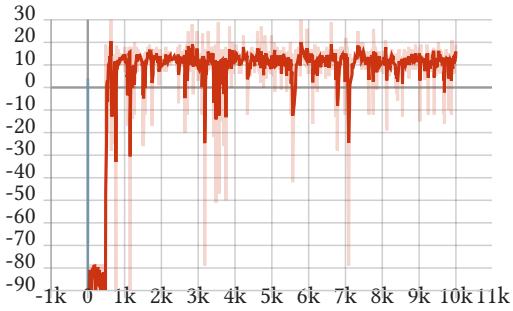

Figure 5: Timestep-vs-Rewards for Agent 2, problem 2

Observations :

When both agents start at (0,0), agents can be seen cooperating and reaching for separate targets and wouldn't go for the same target, same observation has been made over several runs, starting from random positions. This seems to be convincing evidence that the agents have learned explicit cooperation.

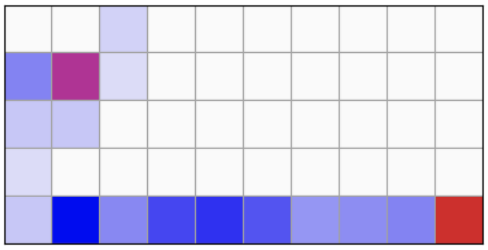

Figure 6: Problem 2: Both agents starting at (0,0) bottom left and take separate targets

However, this sometimes causes the other agent to wander around a lot and therefore causes its cumulative reward to drop until the other agent reaches the target.

3.3 Problem 3

The architecture of the DQN and state description is the same as Problem 2. in this setting, however, the reward for capturing the target is global. That is either of the agents reaches the target, both agents receive the reward.

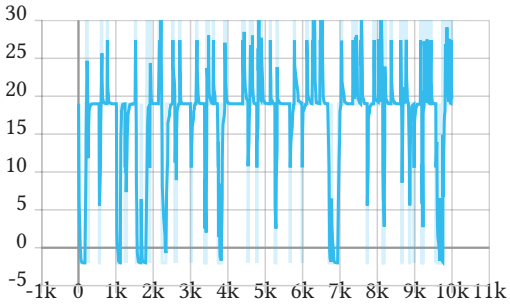

Figure 7: Problem 3: Timestep-vs-cumulative rewards of both agents.

Observations:

The agents seem to have learned quicker compared to the previous problem to maximize their cumulative reward, this can be observed in the graph in Fig. 7. The agents also seem to learn to implicitly cooperate, and the agents take considerably fewer steps to reach the target in this setting. It can be observed that once an agent reaches a target, both the agents try to go to the next target which was not previously observed in Problem 2. this shows cooperative behavior among the complete the game in as few time steps as possible to keep the cumulative reward higher.

4 CONCLUDING REMARKS

The agents in Problem 2, take a mean of 46.7 steps to finish the game over 100 games, while the agents in Problem 3, take an average of 22.3 steps over 100 games. This shows that the agents have learned to implicitly cooperate in Problem 3 while they observe non-competitive behavior in problem 2, yet this doesn't profit both the agents, in the long run. This can be observed in Fig.9

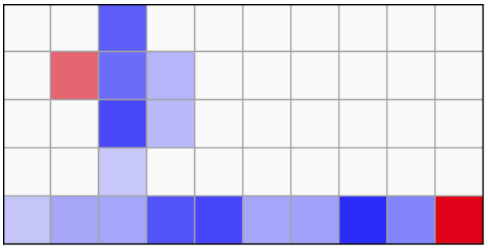

Figure 8: Problem 3: Both agents starting at (0,0) bottom left and take separate targets

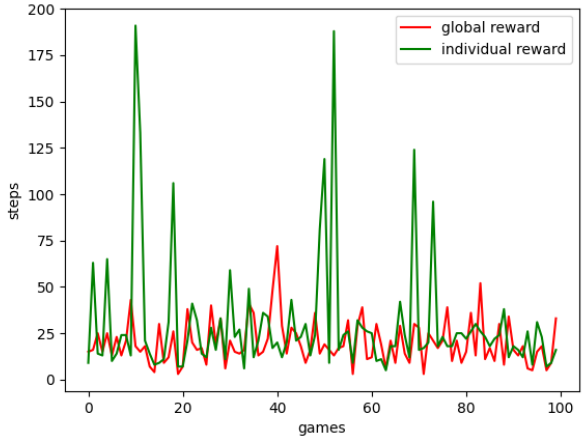

Figure 9: Number of steps taken to complete the game in Problem 2 and Problem 3, starting at the same random position in the grid in each game. It can be seen that global reward is a contributing factor

5 DETAILS OF RUNNING THE CODE:

The code is attached in the code.zip file, the code has three components

- Gridworld Rendering and State updates: window.py, rendering.py, grid.py
- DQN Training: my_dqn_img.py, dqn_p2_state.py, dqn_p3_state.py
- DQN Testing: load_and_test.py, load_and_test2.py

5.1 Problem 1

To reproduce results for Problem 1:

Training:

```
$ python my_dqn_img.py$
```

Test:

```
$ python load_and_test.pyy$
```

5.2 Problem 2 and 3

To reproduce results for Problems 2 and 3:
Training:

```
$ python dqn_p*_state.py$
```

Test:

```
$ python load_and_test2.pyy$
```

REFERENCES

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. (2013). http://arxiv.org/abs/1312.5602 cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf