Puddle World Problem - DQN System

Vihaan Akshaay Rajendiran - ME17B171

April 2, 2021

1 The Puddle World

The 'Puddle World' is a 12x12 grid world with 4 stochastic actions.

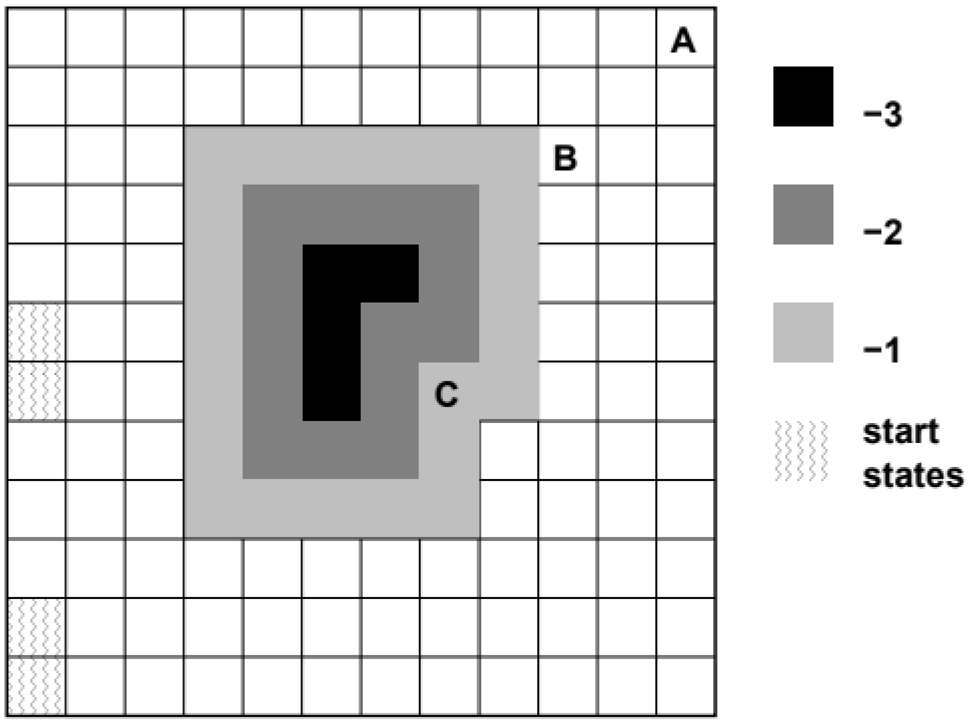

Figure 1: The Puddle world

1.1 States and Rewards

- The player spawns in one of the four start states in the first column of the man.
- The goal can be any of the three letters, 'A', 'B' or 'C' and the reward for reaching the state is +10.
- Certain positions of the map represent a puddle that the agent chooses to avoid. This is mathematically modelled by assigning a negative weight to all the puddle positions.(-1, -2 and -3 for gradients of gray)

1.2 Actions

- For a general state, the player can take 4 actions: Up, Down, Left and Right.
- Actions which will result in the agent going out of the map are ignored.
- For every action taken, the probability of that action effectively happening is 0.9. The rest of the actions take 0.1 probability equally spread amongt them
- There is an additional 'Wind' factor that pushes the player by one block to the east with a probability of 0.5 at the end of every action step

2 Notation

The Puddle Map is expressed as a (12x12) matrix and the rewards for each state is stored. Each state is a (1x2) vector and so is each action.

- $Transitions$ - A transition is a vector of the comprising of a state s, the action taken at this time step a, the next state s^i, the reward obtained for this action r, and a boolean value *bool* which declares if the s^i is the goal state.
- $Episode$ - An episode is the sequence of transitions from a start state to the goal state .
- $puddleWorld$ - This is a module in the code which has the map encoded along with its properties.
- *dqnagent* - This is the agent which has a deep neural network which we wish to train. The model used here is a dense neural network with two hidden layers(128 and 32 hidden nodes respectively) and the output layer with 4 nodes and linear activation to represent optimal q values for the input state.
- *epsilon* - The dilemma between exploration and exploitation is tackled by defining epsilon, which refers to the probability of exploration. We alter epsilon in such a way that the model to explore more in the initial phase and eventually start exploiting as it gains more experience and has seen more data.
- *replaymemory* - The order and type of subsequent interactions of the agent with the environment are highly influenced by the initial set of actions taken by the agent. Despite this, generally, two consequent actions and states are generally very close. This might affect the degree of generalisation of the function approximator. To tackle this, we generally store the experience in a deque and we pick actions in a random order to act as an unbiased dataset to train the neural network.

3 Applying DQN

3.1 Setting up the premise

For a given state s, there is a number(also called optimal q-value associated with every action a taken, given by $q^*(s,a)$, which refers to the highest cumulative discounted future reward if that action is taken from the state.

The goal of DQN is to use a neural network (with parameters θ) as a function approximator, which takes in the state and the action and returns the optimal Q-value. So, for obtaining the best path from a given start state, we would use the neural network at each state to obtain

Q- values at a state. We could consider all possible actions, and the action with the highest Q-value—the highest cumulative discounted future reward— would be the best choice at each state, following the optimal path/policy.

$q^*(s, a) = q(s, a, \theta)$

The challenging part is training the neural network to act as a good function approximator.

3.2 Target for training the neural network

We begin this section with the Q-value update equation derived from the Bellman equation:

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max Q(S_{t+1}, A_{t+1} - Q(S_t, A_t)]$

Here, the target of the neural network is taken to be $R_{t+1} + \gamma max Q(S_{t+1}, A_{t+1}$. One can argue that it is predicting its own value, but since R is the unbiased true reward, the network is going to update its gradient using backpropagation to finally converge.

3.3 Generating Data and Replay Memory

For every episode, we set a maximum upper limit on the number of steps and save every *transition* in a double-ended queue called *replaymemory*. This serves as a memory of experiences and for training, we collect samples from this memory. The reason why we do not feed the data directly into the neural network for training is that, these data points are very close to each other and won't generalise well for the neural network to learn from. 20 minibatches(each of size 64) were taken from the memory (in random order) so as to generalise the training data well, and was fed into the neural network. We also altered epsilon as we progressed with episodes so as to encourage the model to exploit more and explore less as it kept learning.

The above step was repeated for 5000 episodes and the neural network was trained.

3.4 Q - Actor Critic

The "Critic" estimates the value function. This could be the action-value (the Q value) or state-value (the V value). The "Actor" updates the policy distribution in the direction suggested by the Critic (such as with policy gradients).

For the whole training process, we have a target_model, parallel to the main neural network. For calculating the target for the network, we use this target_model as an actor and keep train the main neural network. Every once in 5 episodes, we update the target_model with the weights of the actual model so as to keep it competent and the algorithm effective.

3.5 Testing the agent

After the model is trained, we can simulate a new object of the environment class with the same goal. Assigning a start state and using the agent to determine the q values for each state allows us to choose the best action for a given state. This can be used in a loop to count the number of steps the agent takes to reach the goal.

3.6 Pseudocode

```
start with:
epsilon = 1
Repeat for 5000 episodes:

    • state = startState
```

- collect data till end of episode/ max number of 200 steps:
  action = choose epsilon greedy
  next state = findNextState(state,action)
  bool_done = if goal state is reached
  update transition = [state,action,next state,reward, bool_done] to *replay_memory*
  state = next_state
- Repeat for 20 mini-batches:
  - collect mini-batch(64 size) randomly from *replaymemory*
  - if terminal_state: target = reward
    else: target = reward + max q(next_state,actions)[prediction obtained from target_model]
  - train main_model
- reduce epsilon with small steps
- Update target_model weights with main_model weights every 5^{th} episode

4 Conclusion

The algorithm is coded on a colab notebook that can be found here. The neural network used was a dense network with input layer having 2 nodes [Representing the state of the agent], first hidden layer with 128 nodes and second hidden layer with 32 nodes. The output layer has 4 nodes with a linear activation which represents the q values for possible actions from the given state. A replay_memory double ended queue of size 50,000 was used, and the experiment was repeated for 5,000 episodes, with 20 mini batches of size 64 used in each episode. The actor network was updated every 5th episode and a stochastic gradient ascent algorithm with a learning rate of 0.01 was used to train the main network having a mean squared error loss. Puddle map with 'A' being the goal was used as the environment for this experiment.

The map spawns the player at one of the following 4 start states. Explicitly running the agent from each of these start states, the results are recorded. An average of 20 is taken so as to account for the stochastic nature of actions and states. For the start states [5,0],[6,0],[10,0],[11,0] the number of steps taken by the agent to reach the goal are respectively 31,32,26,24 This number can be improved by using a more sophisticated model(neural network with more dense layers and more hidden nodes or with convoluted neural networks) as well as, increasing the number of episodes, size of replay_memory, the number of training sessions with a smaller learning rate.

5 References

1. Playing Atari with Deep Reinforcement Learning - Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller
2. Reinforcement Learning: An Introduction
3. Understanding Actor Critic Methods and A2C
4. Deep Reinforcement Learning Blog - John Joo
5. Deep Q Network with TF-Agents