Midterm 2

Norman Rasmussen

Decemeber 2025

1 Problem

1.1 Notes

The files can be found here "https://github.com/normanlrasmussen/bank_rl_practice_problem". The bank and player files were coded completely by me. The bank_environment and main files were created with AI assistance. For the rl_players file, I adapted the algorithms I coded for the grid world assignment.

1.2 The Game BANK

Bank is a push-your-luck dice game where each round starts with three dice rolls whose sums form the starting "bank value." (In the first 3 rolls, 7's count as 70s). On your turn, you decide whether to roll or bank. Rolling adds the value of two dice to the bank—except if you roll a 7, you lose everything for that round and score 0, and if you roll doubles, the bank doubles instead of adding. You may keep rolling as long as you want, but at any time you can bank, which locks in the current bank value and adds it to your total score. In multiplayer versions, everyone starts the round together, and once a player banks they stop while others continue. After a set number of rounds, whoever has the highest total score wins.

Normally, this game would have an extremely large state space, however, we are going to simply it to make it viable for quick training using SARSA and Q-learning. The state you will be working with will be the score after 3 rolls. So a number between 6 and 210. The action space will be to choose a threshold to bank at. (A number between 50 and 230, with 20 in between every possible choice). You will be teaching a player to win against an agent that has a 30% chance of banking at every turn.

1.3 Problem 1

First you will code up the `SARSAAgnet` class. This will employ SARSA learning. You will implement the `train` method of class. Below is sudo code you can follow.

Please note that η here is used as a learning rate as a constant function. We also add a decaying epsilon term to encourage exploration in the beginning. Some useful methods to use from `self.env` are `reset()`, `choose_action()`, `step()` and `get_actions()`. You can find all of the uses of these functions in the `bank_environment.py` file.

Algorithm 1 SARSA: Training Procedure

function TRAIN(N episodes)
  for $k = 1$ to N **do**
    $\epsilon \leftarrow \max(\epsilon_{\min},\, \epsilon \cdot \epsilon_{\text{decay}})$
    Reset environment and observe initial state s_0
    Choose initial action $a_0 \leftarrow$ CHOOSEACTION(s_0)
    episode_reward $\leftarrow 0$

    while episode not terminated **do**
      Take action a_t, observe reward r_t and next state s_{t+1}
      if s_{t+1} is terminal **then**
        $Q(s_{t+1}, a_{t+1}) \leftarrow 0$
        $a_{t+1} \leftarrow 0$ {Dummy action}
      else
        $a_{t+1} \leftarrow$ CHOOSEACTION(s_{t+1})
      end if

      Compute TD error:

$$\delta_t \leftarrow r_t + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

      Update Q-value:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta\, \delta_t$$

      $s_t \leftarrow s_{t+1}, \quad a_t \leftarrow a_{t+1}$
      episode_reward $\leftarrow$ episode_reward $+ r_t$
    end while
    Store episode_reward
  end for
  return Q-table and reward history
end function
=0

1.4 Problem 2

Next you will code up the `QLearningAgnet` class. This will employ Q-learning learning. You will implement the `train` method of class. Below is sudo code you can follow.

Algorithm 2 Q-Learning Training Procedure

function TRAIN(N episodes)
  for $k = 1$ to N **do**
    $\epsilon \leftarrow \max(\epsilon_{\min}, \epsilon \cdot \epsilon_{\text{decay}})$
    Reset environment and observe s_0
    total_reward $\leftarrow 0$
    done $\leftarrow$ **False**
    while not done **do**
      Choose action a_t using ϵ-greedy policy
      Take action a_t, observe reward r_t and next state s_{t+1}
      total_reward $\leftarrow$ total_reward $+ r_t$
      $Q_{\text{current}} \leftarrow Q(s_t, a_t)$
      if s_{t+1} is terminal **then**
        $Q_{\text{max-next}} \leftarrow 0$
      else
        $Q_{\text{max-next}} \leftarrow \max_{a'} Q(s_{t+1}, a')$
      end if
      Compute TD error:

$$\delta_t = r_t + \gamma\, Q_{\text{max-next}} - Q_{\text{current}}$$

      Update Q-value:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta\, \delta_t$$

      $s_t \leftarrow s_{t+1}$
    end while
    Record total_reward for episode
  end for
  return Q-table and reward history
end function
=0

1.5 Problem 3

For the final part of this assignment you will develop the reward function for this game. Reward shaping plays a critical role in reinforcement learning. You can have the most effective algorithm, but if your reward design is poor, the agent will learning poorly. You will implement 4 different reward functions and compare them in order to gain a better grasp on their influence.

You will implement these in the `_calculate_reward()` method of the `BankEnviroment` class. The first reward function will be a "sparse" reward. This gives 1 if they win, 0 if they tie, and -1 if they lose. While this is an easy to implement method, it can often under perform because of the lack of frequent signals to learn from.

The next reward function will be "relative". This function should be the players current score minus the opponents score. This encourages the player to

stay ahead of their opponent.

After this you will implement the "score" reward function. This should use the score of the player as the reward to be given. So they will encouraged to increase their score.

Now, all the above reward functions are quite simple, and some of them are bad. For the last part implement a "custom" reward function, that makes use of a combination of the rewards above, or is of your own design.

(It might be useful to use the `bank.player_scores` attribute of the class to get the RL players score at index 0 and the opponents score at index 1).

1.6 Problem 4

Run the `main.py` file to compare the performance of your various reward functions and agents. Write about any observations you can make in comparing SARSA and Q-learning or different reward functions. Make sure to run at least 5,000 episodes.