MBD Sept 24

Session 10
Review Practice Assignments

jmanero@faculty.ie.edu

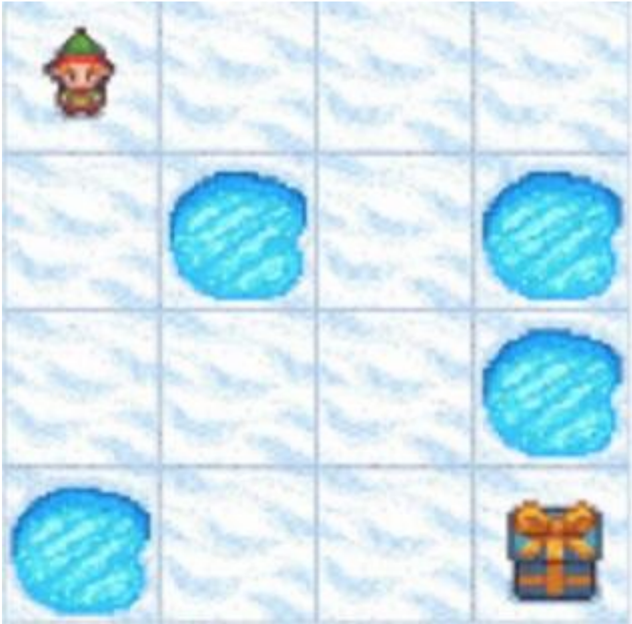
Let’s see where we are

- Assignment 1 – Jump start
- Assignment 2 – Dynamic Programming
- Assignment 3 – Model Free models
- Assignment 4 – DQN / DDQN
- Group Practice
- Mountain Cart Challenge

Assignment 3

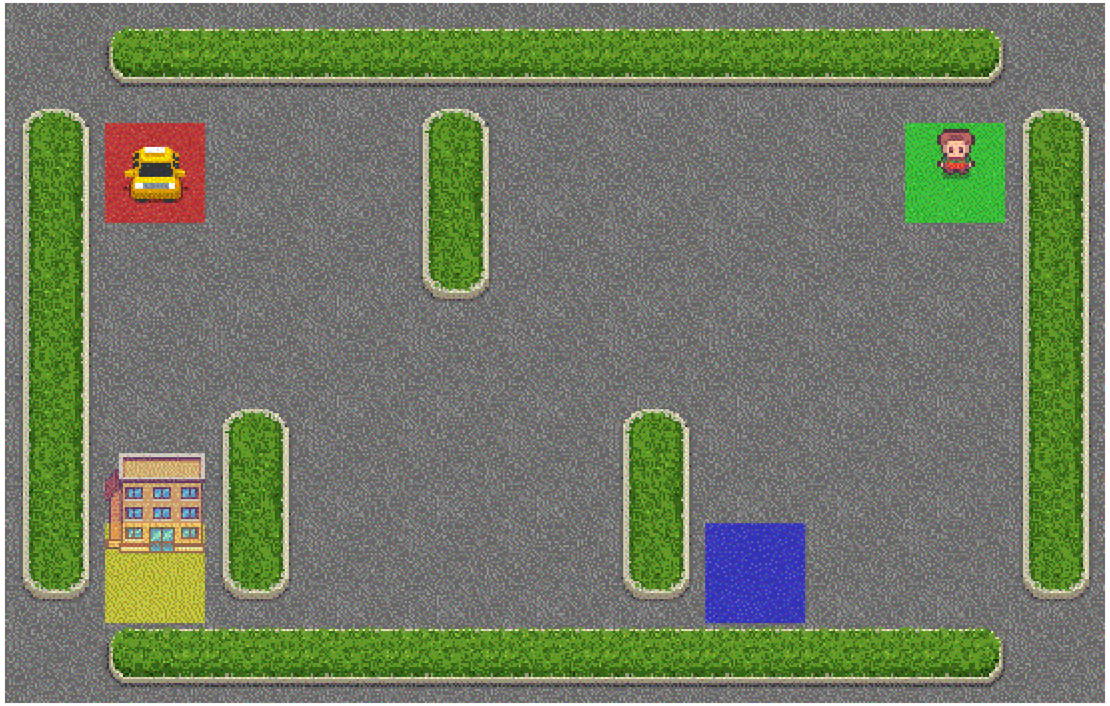
- All methods have different learning strategies
    - MC
    - TD
    - SARSA
    - Q-Learning

Frozen Lake



- 2 Dimensional
- Fixed start – end-point
- Stochastic and non-stochastic
- 8x8 large state space

Taxi



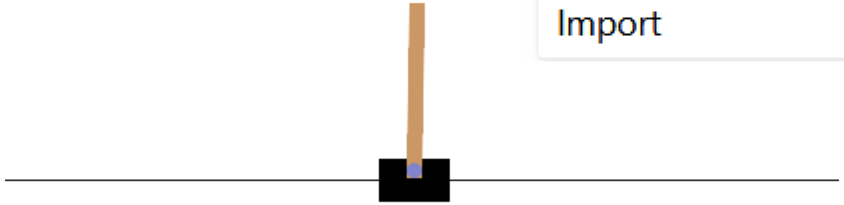
- Large observation space
- Grid 5x5
- State is encoded
  - Taxi row (5 possible positions)
  - Taxi column (5 possible positions)
  - Passenger location (5 possible values: at 4 locations or in the taxi)
  - Destination (4 possible values)
- 4 pick-up/drop-off places (red green yellow and blue)
- Taxi starts in any place
- Final cumulative reward around 8 if done properly

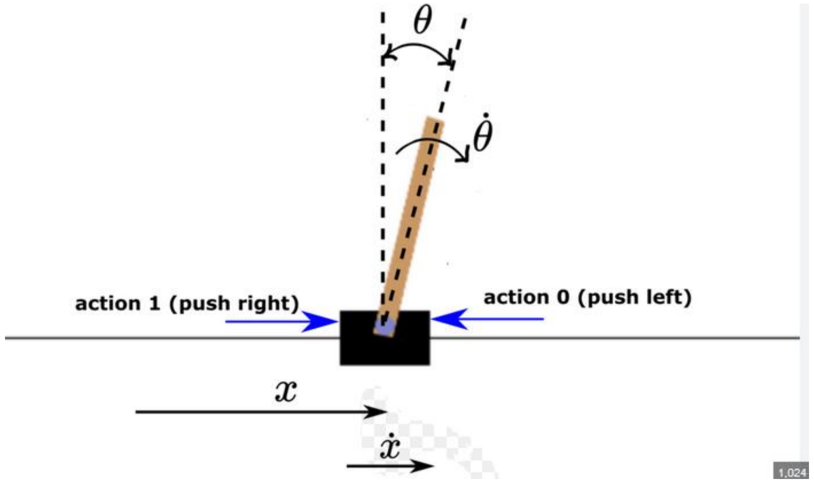
- The policy is difficult to visualize!!!

CartPole

Action Space	Discrete(2)
Observation Shape	(4,)
Observation High	[4.8 inf 0.42 inf]
Observation Low	[-4.8 -inf -0.42 -inf]
Import	`gym.make("CartPole-v1")`

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf





- Requires binning to use the continuous values in the observation for Q-Learning
- It may take time to converge
- Reward +1 for every step

Review Methods and environments

Monte Carlo

Feature	Experience	Conclusion
Speed to converge		
Use in several environments		
What wend well and what went bad		
Overall rating		

SARSA

Feature	Experience	Conclusion
Speed to converge		
Use in several environments		
What wend well and what went bad		
Overall rating		

Q-Learning

Feature	Experience	Conclusion
Speed to converge		
Use in several environments		
What wend well and what went bad		
Overall rating		

Remember

- SARSA shares similarities with Q-learning
- Both algorithms aim to learn the optimal action-value function and use similar update rules. However, the key difference lies in their learning strategies.
- While Q-learning is an off-policy algorithm that updates the Q-values based on the maximum expected future reward, regardless of the action taken, SARSA updates the Q-values based on the actual action taken by the current policy. This difference can lead to different learning dynamics and performance characteristics in certain problems.

What should we observe

Aspect	Q-Learning	SARSA
Policy Type	Off-policy	On-policy
Next Q-value Estimation	Uses max Q(s',a') (greedy action)	Uses Q(s', a') of the actual action taken
Exploration vs. Exploitation	Encourages more exploitation (greedy updates)	Encourages more exploration (policy-following updates)
Risk Sensitivity	More aggressive, assumes optimal future actions	More conservative, considers the actual exploration strategy
Convergence Speed	Typically, faster but less stable in stochastic environments	More stable but slower

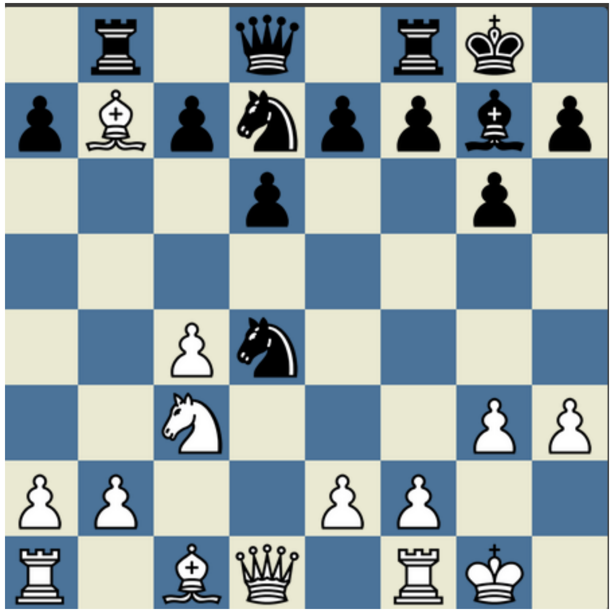
Rewards
Reward Shaping

Definition

is the use of small intermediate ‘fake’ rewards given to the learning agent that help it converge more quickly.

Gymnasium

- Each environment has its reward already shaped
- We may try ‘reward shaping’ to improve learning
- For instance, in Chess how do we evaluate the reward?

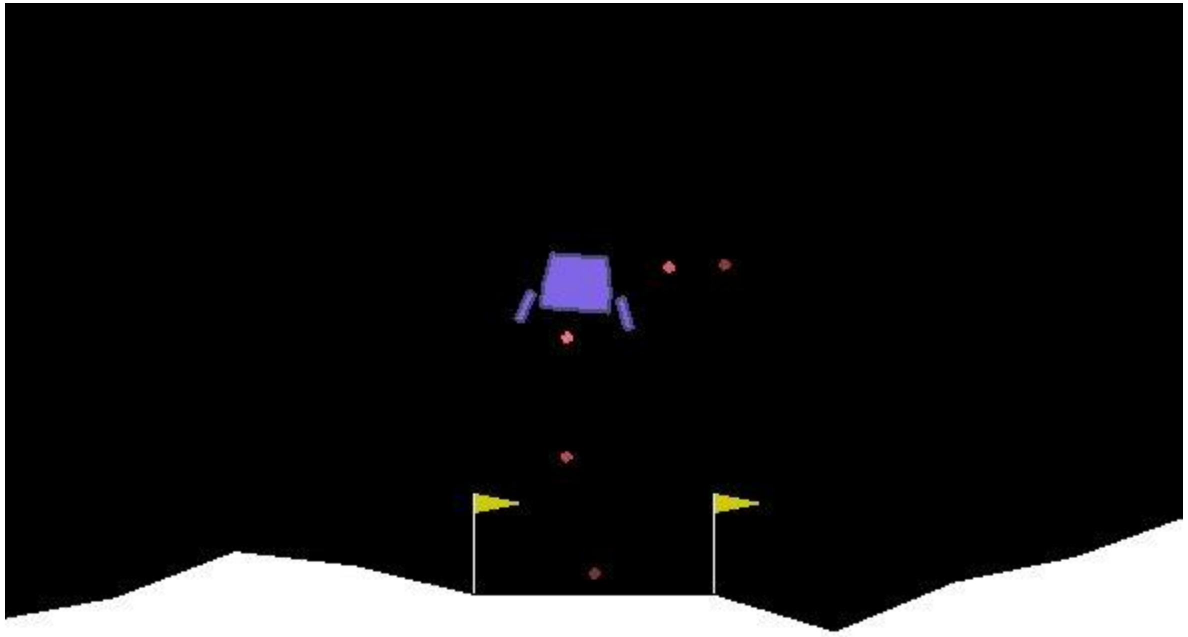


- King's safety.
- Material on the board.
- Pieces activity.
- Pawn structure.

But… How to define a quantitative measure?

Lunar Lander

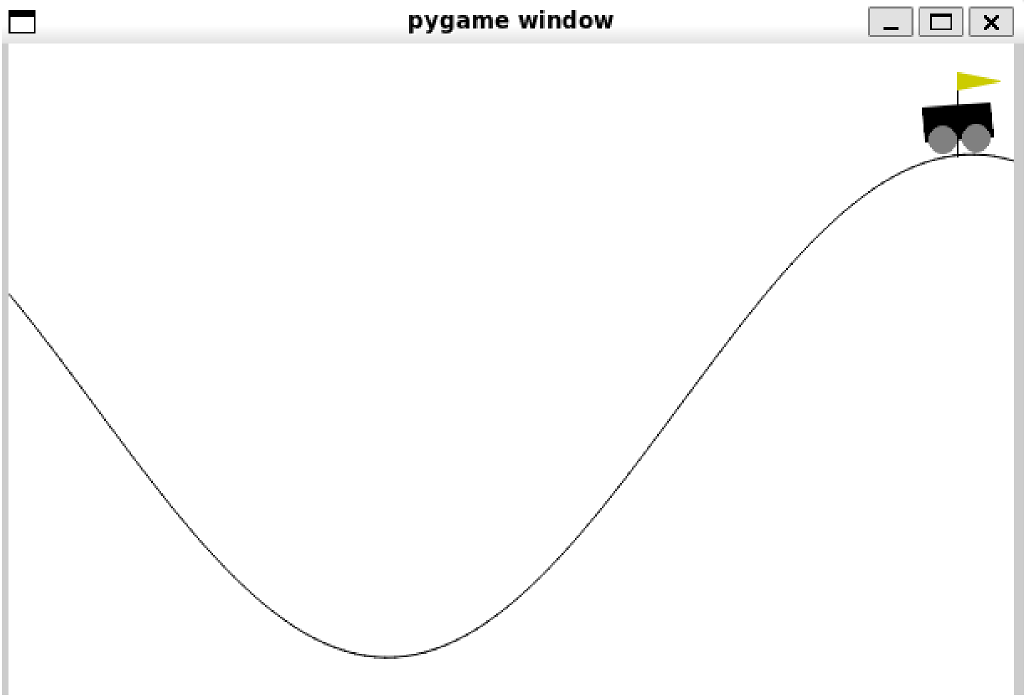
- Sparse rewards (big reward far away)
- No truncation of the episode / infinite fuel
- May end-up learning to hoover



Solutions

- Reward shaping
- Include fuel consumption
- Increase gravity

Shaping Rewards – MOUNTAIN-CART



This environment is part of the Classic Control environments which contains general information about the environment.

Action Space	`Discrete(3)`
Observation Space	`Box([-1.2 -0.07], [0.6 0.07], (2,), float32)`
import	`gymnasium.make("MountainCar-v0")`

For the Assignment MOUNTAIN CAR or CARTPOLE Discretization

```
#Support Funtions


# Function to create bins
def create_bins(interval, num):
    return np.linspace(interval[0], interval[1], num + 1)

# Updated intervals and bin sizes for discretization
intervals = [(-2.4, 2.4), (-3.0, 3.0), (-0.5, 0.5), (-2.0, 2.0)]
nbins = [12, 12, 24, 24]  # Increased bins for finer state representation
bins = [create_bins(intervals[i], nbins[i]) for i in range(4)]

# Function to discretize state variables into bins
def discretize_bins(x):
    return tuple(np.clip(np.digitize(x[i], bins[i]) - 1, 0, nbins[i] - 1) for i in range(4))
```

Reward Shaping in Mountain Car

```python

###### SHAPING REWARDS  #####

        shaped_reward = reward
        if done and step < 200:
          # If episode is ended the we have won the game. So, give some large positive reward
            shaped_reward = 250 + shaped_reward

        # Velocity is important, we give positive reward for velocity (is the sign correct?)
        velocity = next_obs[1]
        distance = next_obs[0]
        shaped_reward = shaped_reward + 1* abs(distance)
#        shaped_reward = shaped_reward + 10 * abs(velocity)

###### END SHAPING REWARDS  #####

```

2 Problems discretization and Shaping rewards in the Challenge

Discretization

- How many bins are ok?
- Hint: use this discretization strategy for the CARTPOLE Assignment !

Shaped Rewards

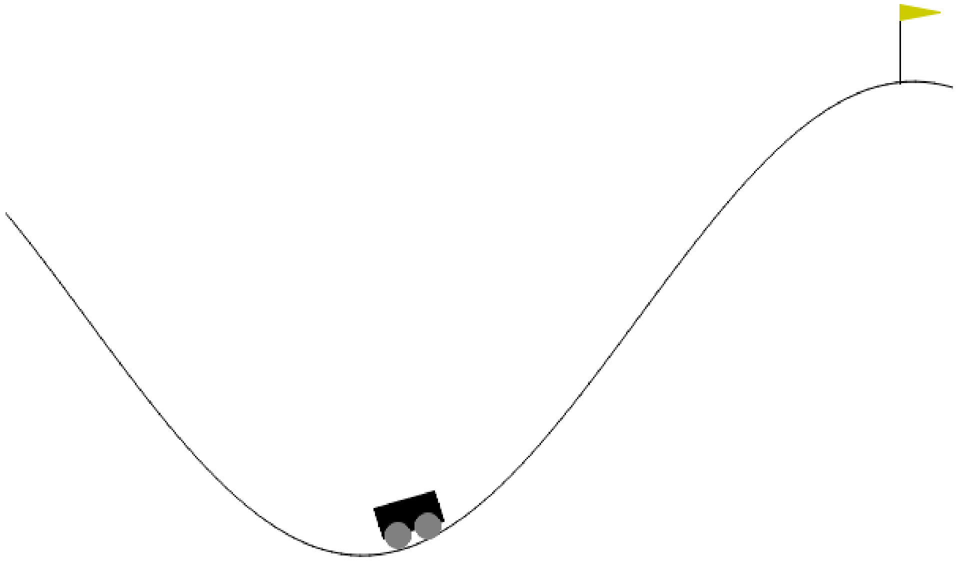
- -1 each timestep
- This reward does not allow good learning
- There is truncation and termination (careful)
- Can we define a shaped reward?

Objective

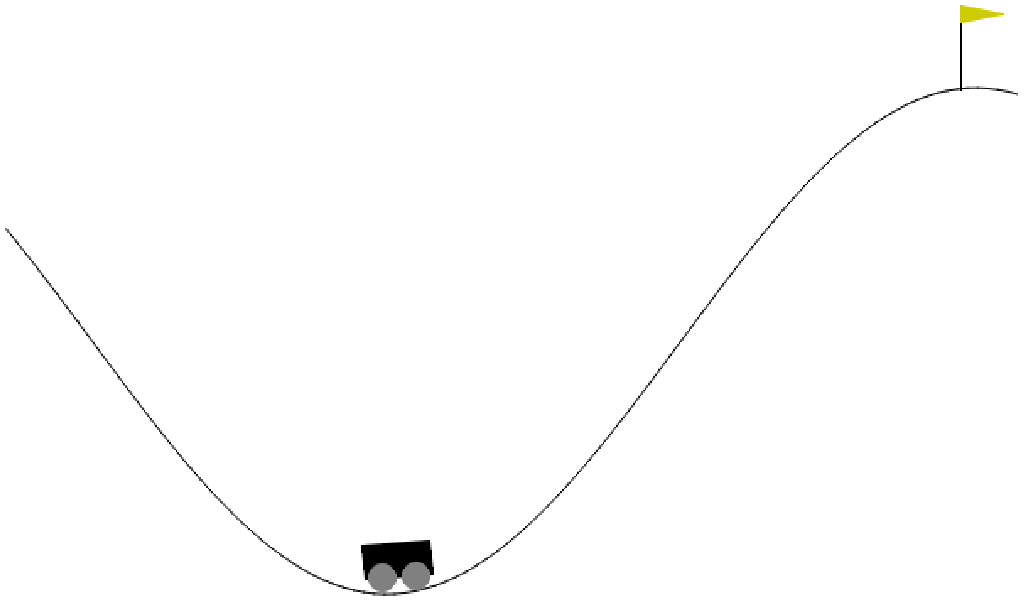
- **By changing BINS and Shaping reward obtain the BEST POSIBLE LEARNING (FAST-SHORT)**
- Use ChatGPT, internet, your ideas, whatever
- Don’t modify the code or the method. Just focus on shaped rewards

https://gymnasium.farama.org/environments/classic_control/mountain_car/
https://github.com/castorgit/RL_course/blob/main/021_Q_learning_MOUNTAIN_CAR.ipynb

Diference prioritizing velocity or distance



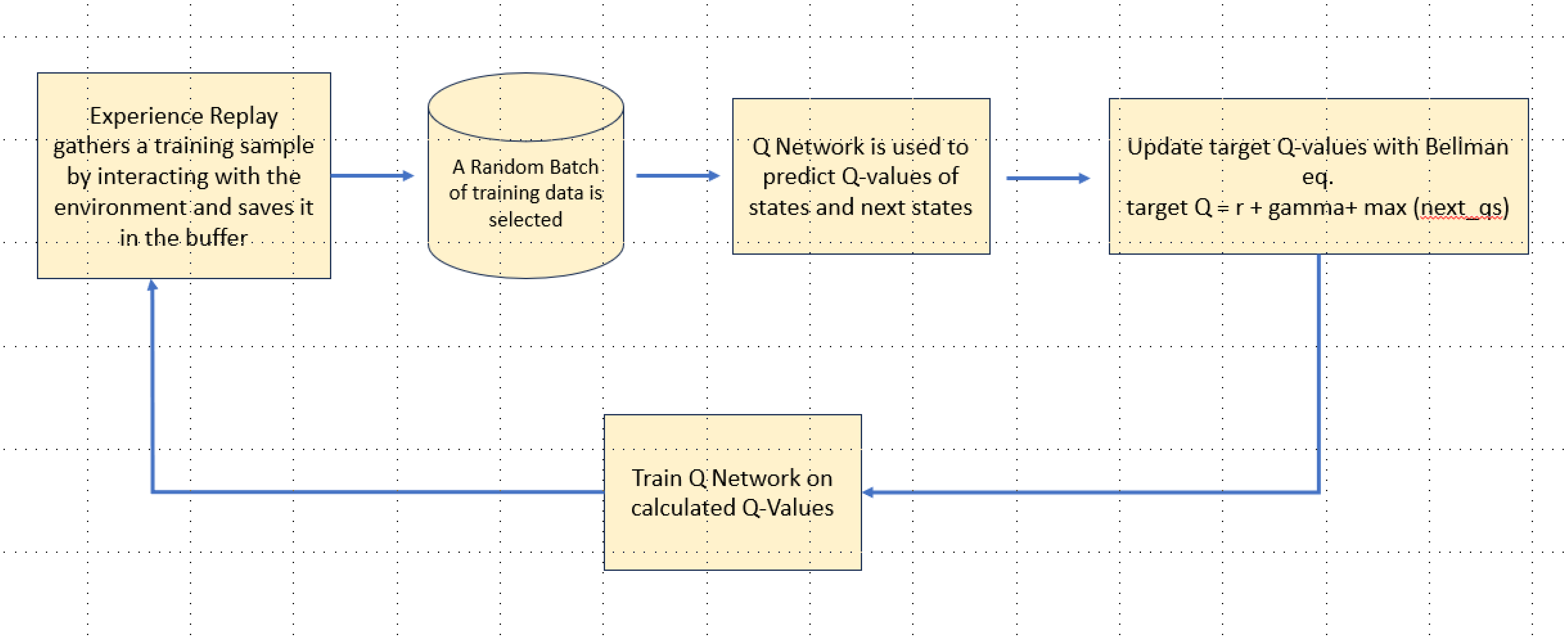
Reward Shaping – Velocity ++



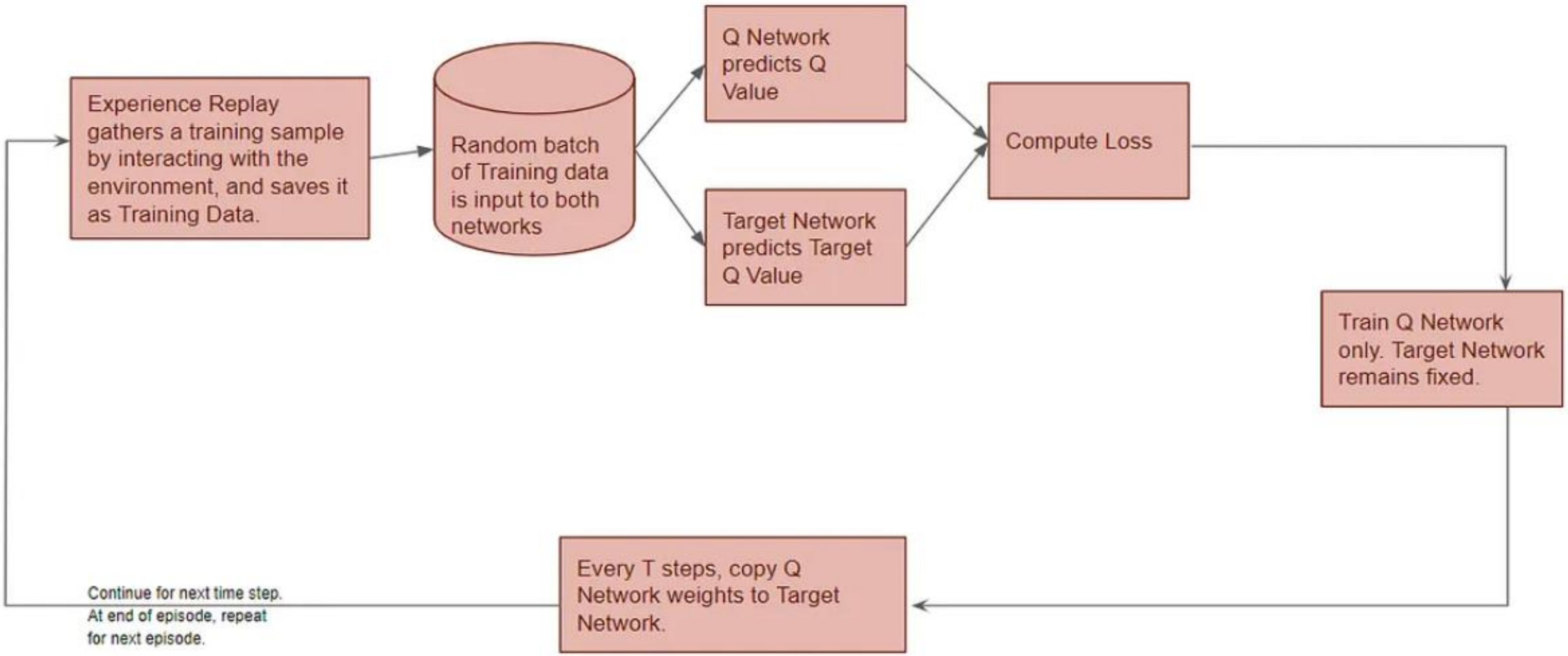
Reward Shaping – distance ++

DQN

General Workflow



Using 2 networks instead of 1



Hyperparameters

```
# Parameters to fine tune
# Try your own parameters
# Remember epsilon and gamma are very important


MAX_EPISODES = 300
ROLLING_WINDOW = 20
MEMORY_SIZE = 2000
MAX_STEPS = 500

gamma = 0.99                                    # discount rate
epsilon = 1.0                                    # exploration rate
epsilon_min = 0.01
epsilon_decay = 0.99
learning_rate = 0.001
batch_size = 64
solved_threshold = 195

verb = 0                                         # to see traces (verbosity)
```

Neural Network

Neural Network

```
###
# [CREATE YOUR NEURAL NETWORK TRY 16/32 or 24/24]
###

def build_model(state_size, action_size):
    inputs = Input(shape=(state_size,), name="state_input")
    x = Dense(1, activation='relu', name="dense_1")(inputs)
    x = Dense(1, activation='relu', name="dense_2")(x)
    outputs = Dense(action_size, activation='linear', name="output_layer")(x)

    model = Model(inputs=inputs, outputs=outputs, name="Q_Network")
    model.compile(loss='mse', optimizer=Adam(learning_rate=learning_rate))
    return model
```

Action greedy selection

```python
def select_action_greedy(state, DQN):
    """
    Selects the if agent takes a random action (explore) or a predicted action (exploit)
    """
    if np.random.rand() <= epsilon:
        return random.randrange(action_size)
    act_values = DQN.predict(state, verbose=verb)
    return np.argmax(act_values[0])  # returns action selected with greedy strategy
```

Sample experiences

```python
# Sample experiences from the replay buffer
def sample_experiences(batch_size):
    """
    Samples a batch_size of experiences from the Replay buffer.
    You MUST transform the data into numpy arrays as this accelerates the response time sensibily
    """
    indices = np.random.choice(len(replay_buffer), batch_size, replace=False)
    batch = [replay_buffer[i] for i in indices]
    states, actions, rewards, next_states, dones = zip(*batch)
    return (states, actions, rewards, next_states, dones)
###
# YOU MUST VECTIORIZE THE RETURN. TRANSFORM THE OUTPUTS IN np arrays otherwise it will be very slow
###
```

Experience Replay

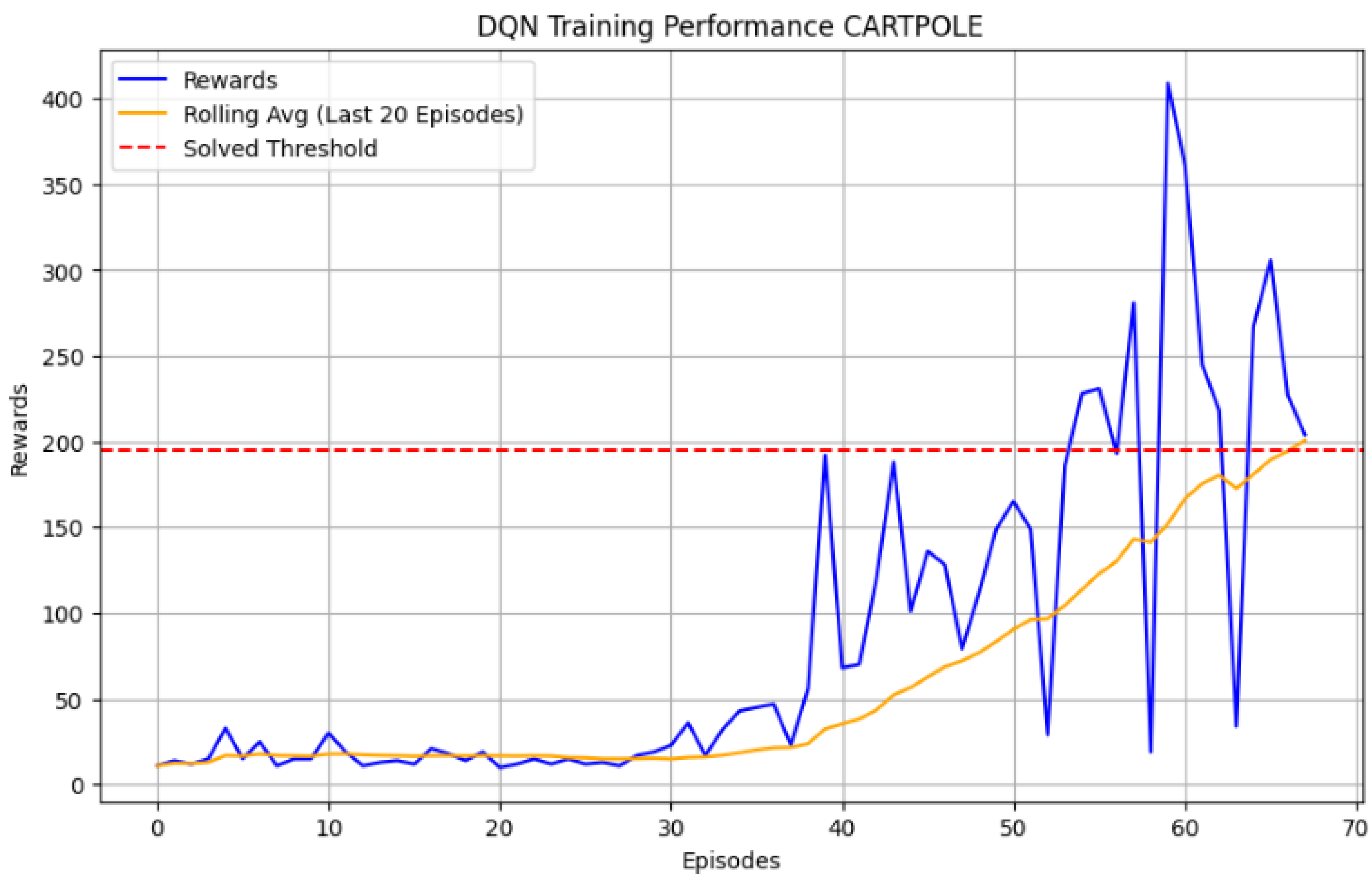
```
def experience_replay(batch_size, model, epsilon):
    """
    The critical function in the whole program
    1. gets a minibatch from replay buffer
    2. Predicts targets_qs from states (we need the full batch predicted but we'll avoid dones)
    3. predicts next_qs from next_states
    4. Bellman equation on next_qs obtains new target_qs $Q_{\text{target}}(s_t, a_t) = r_t + \gamma \cdot \max_

    6. Train DQN input states, output target_qs

    """
    if len(replay_buffer) < batch_size:
        return

    states, actions, rewards, next_states, dones = sample_experiences(batch_size)

    # Predict Q-values for current and next states using vectorized operations

####
#  [ Predict target_qs with model predict states, predict next_qs with model predict next states]
####
    target_qs =
    next_qs   =

    # Update target Q-values using standard DQN logic
    target_qs[np.arange(batch_size), actions] = rewards + gamma * np.max(next_qs, axis=1) * (1 - dones)

    # Train the model on the Q-values
    model.fit(states, target_qs, epochs=1, verbose=0)

```

Main Loop

In [ ]:

```python
rewards_per_episode= []

done = False
rolling_avg = 0
rolling_avg_rewards = []

start_time = time.time()

for e in range(MAX_EPISODES):                            # Should be While True, however we limit number of eps
    state, _ = env.reset()
    state = np.reshape(state, [1, state_size])
    total_reward = 0

    for step in range(MAX_STEPS):

        action = select_action_greedy(state, DQN)
        next_state, reward, done, truncated , _ = env.step(action)

        next_state = np.reshape(next_state, [1, state_size])
        store(state, action, reward, next_state, done)
        state = next_state
        total_reward = total_reward + reward
        if done:
            break

        if len(replay_buffer) > batch_size:
            experience_replay(batch_size, DQN, epsilon)

    epsilon = max(epsilon_min, epsilon * epsilon_decay)            # decay epsilon

    rewards_per_episode.append(total_reward)
    rolling_avg = np.mean(rewards_per_episode[-ROLLING_WINDOW:])   # append rewards
    rolling_avg_rewards.append(rolling_avg)
    print(f"Episode: {e+1:3}/{MAX_EPISODES}, Reward: {total_reward:+7.2f}, "
          f"Epsilon: {epsilon:.2f}, Rolling Avg: {rolling_avg:6.2f}, Steps: {step:3} Terminated: {done} ")


    # Check if environment is solved
    if rolling_avg >= solved_threshold:
        print(f"Environment solved in {e+1} episodes!")
#            model.save("lunarlander_ddqn_model1.keras")
        break

end_time = time.time()
testing_duration = (end_time - start_time) / 60  # Convert to minutes
print(f"Testing completed in {testing_duration:.2f} minutes")
```


· jmanero@faculty.ie.edu


DQN Training Performance CARTPOLE
Rewards
Rolling Avg (Last 20 Episodes)
Solved Threshold
400
350
300
250
200
150
100
50
0
Rewards
0
10
20
30
40
50
60
70
Episodes

END
Session 10

UNIVERSITY
SCHOOL OF SCIENCE & TECHNOLOGY