

**University of Tehran**

**Faculties of Interdisciplinary Sciences and Technologies**

**AI Course**

**Homework 2**

**Babak Bayanian**

832203002

1404/03/02

­

[Question 1: 3](#_Toc198916441)

[Question 2 9](#_Toc198916442)

[1. Imports and Configuration 9](#_Toc198916443)

[2. State Discretisation 11](#_Toc198916444)

[3. Environment Initialisation 12](#_Toc198916445)

[4.Core Q-Learning Functions 13](#_Toc198916446)

[Epsilon-Greedy Strategy 13](#_Toc198916447)

[Q-Table Update Using Bellman Equation 13](#_Toc198916448)

[5. Training the Agent 14](#_Toc198916449)

[6. Visualising Learning 16](#_Toc198916450)

[7. Evaluation and GIF Creation 17](#_Toc198916451)

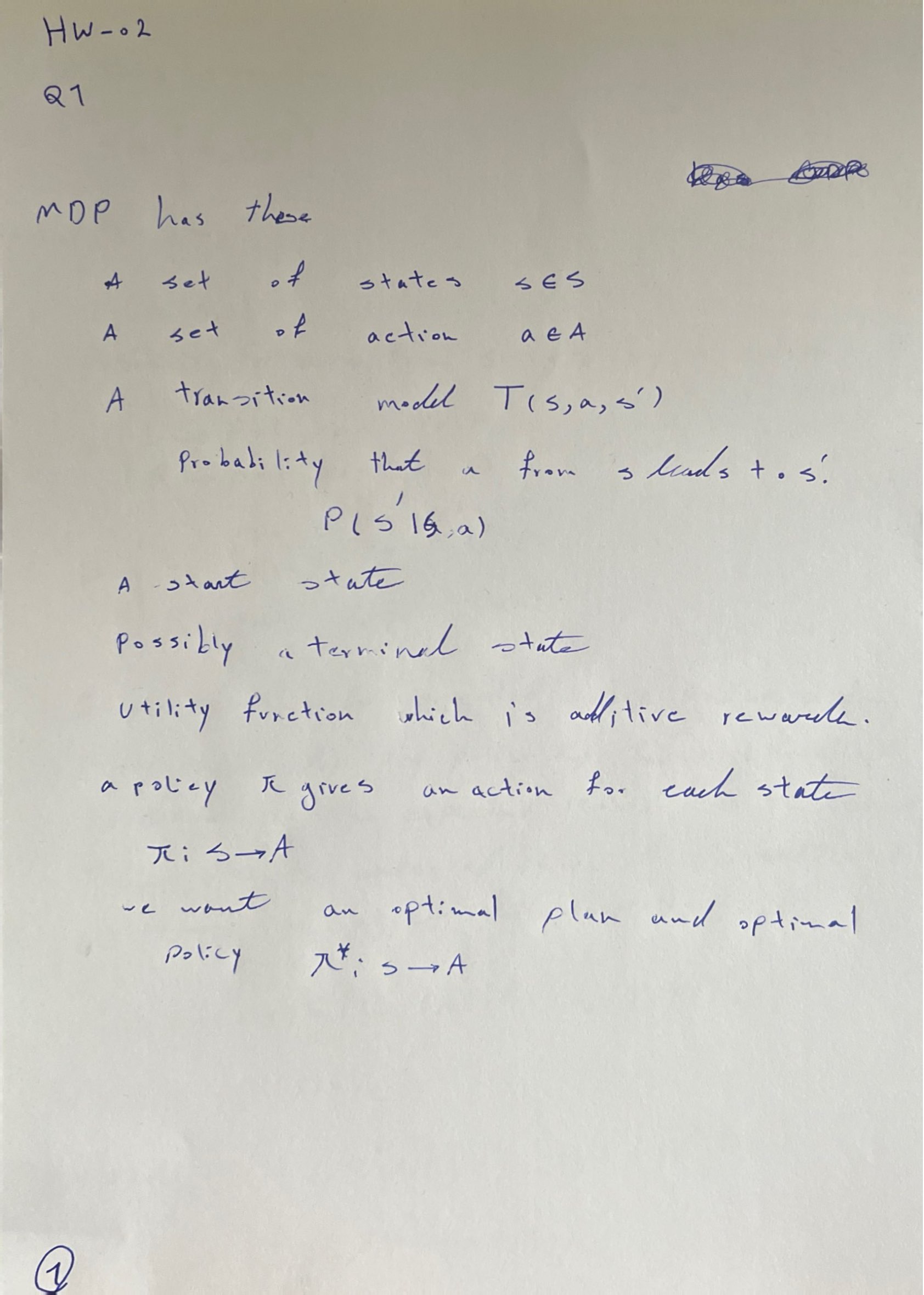
[8. Execution 19](#_Toc198916452)

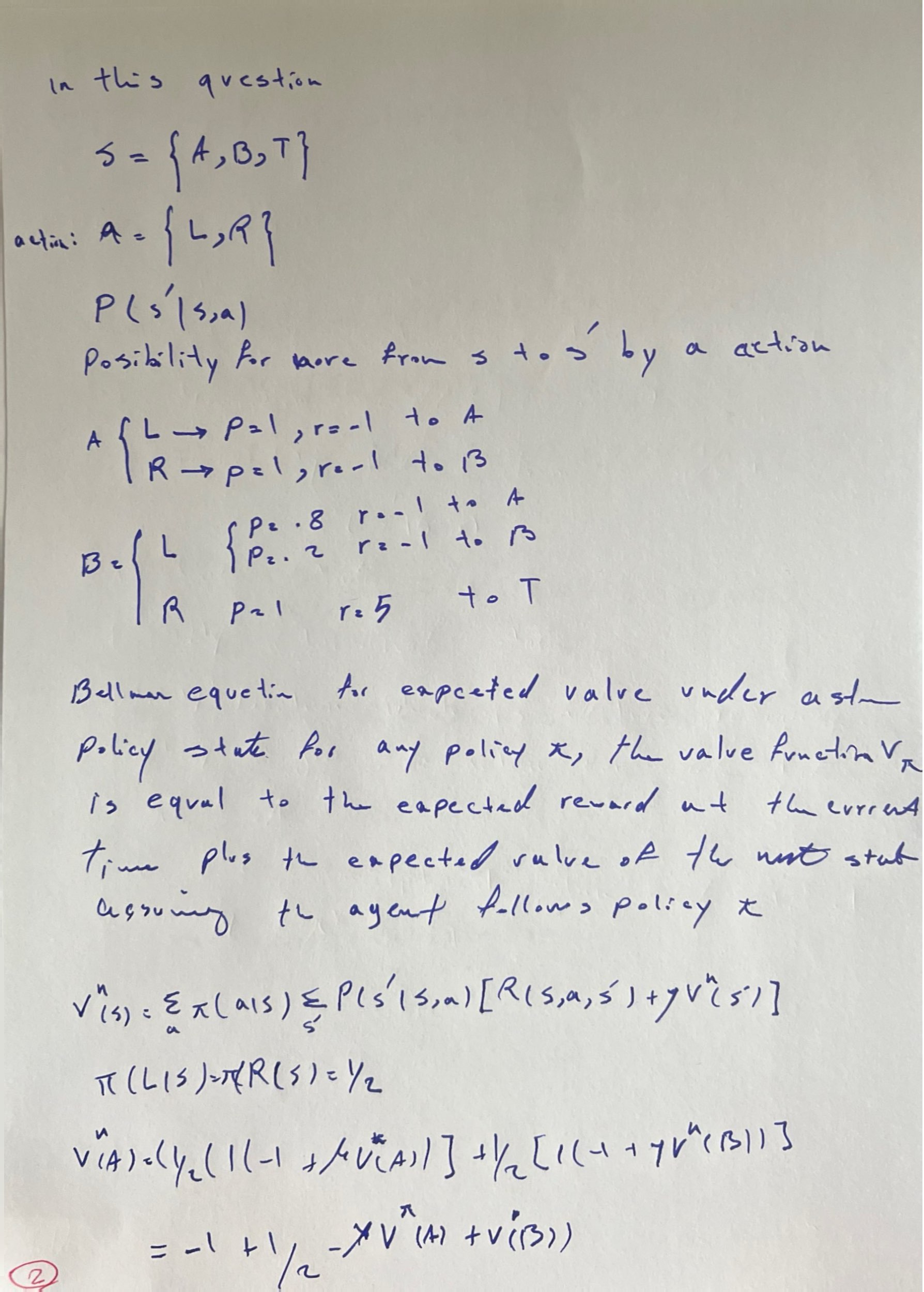
[9. Conclusion 20](#_Toc198916453)

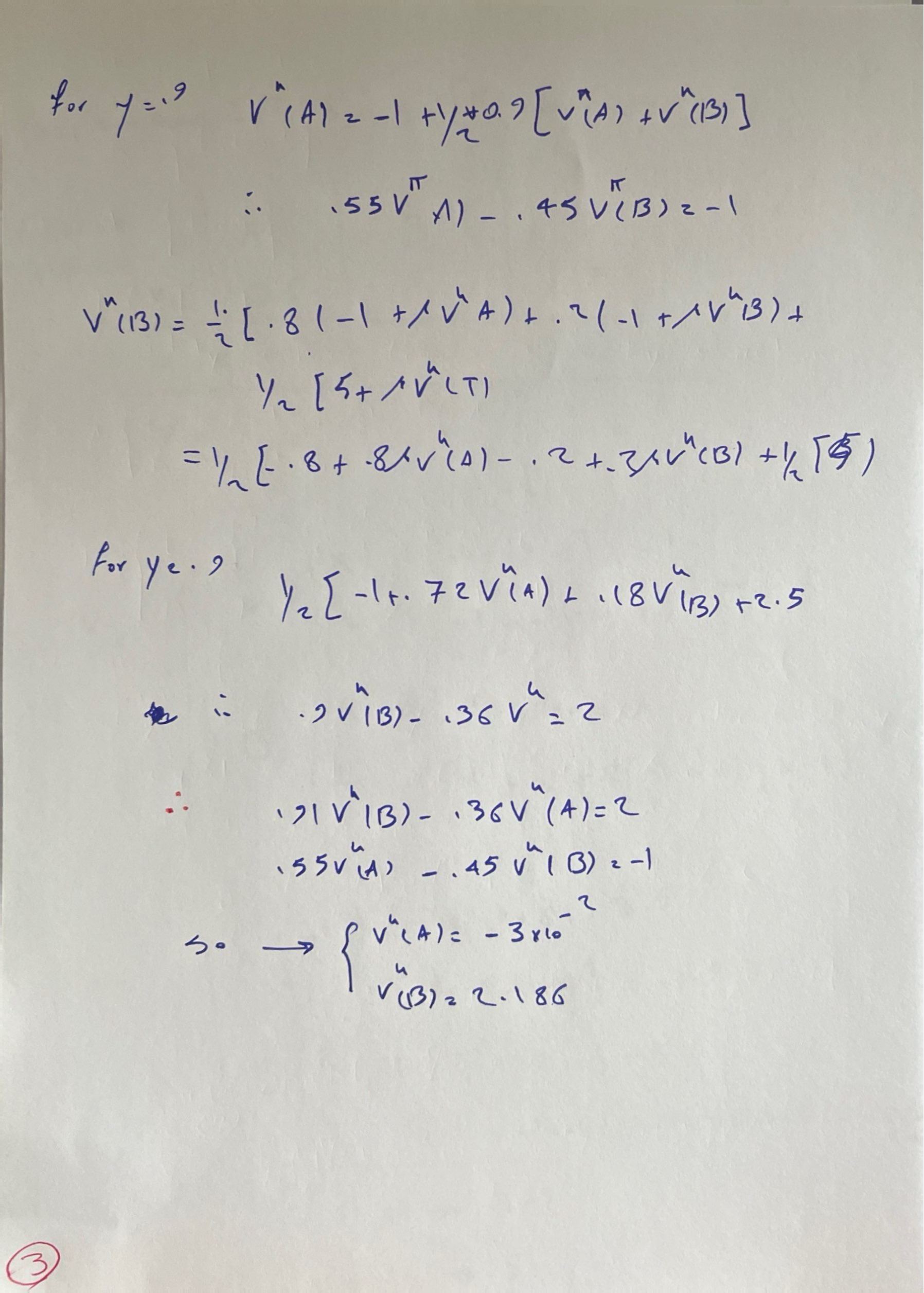
[1. Learning Curve Interpretation 21](#_Toc198916454)

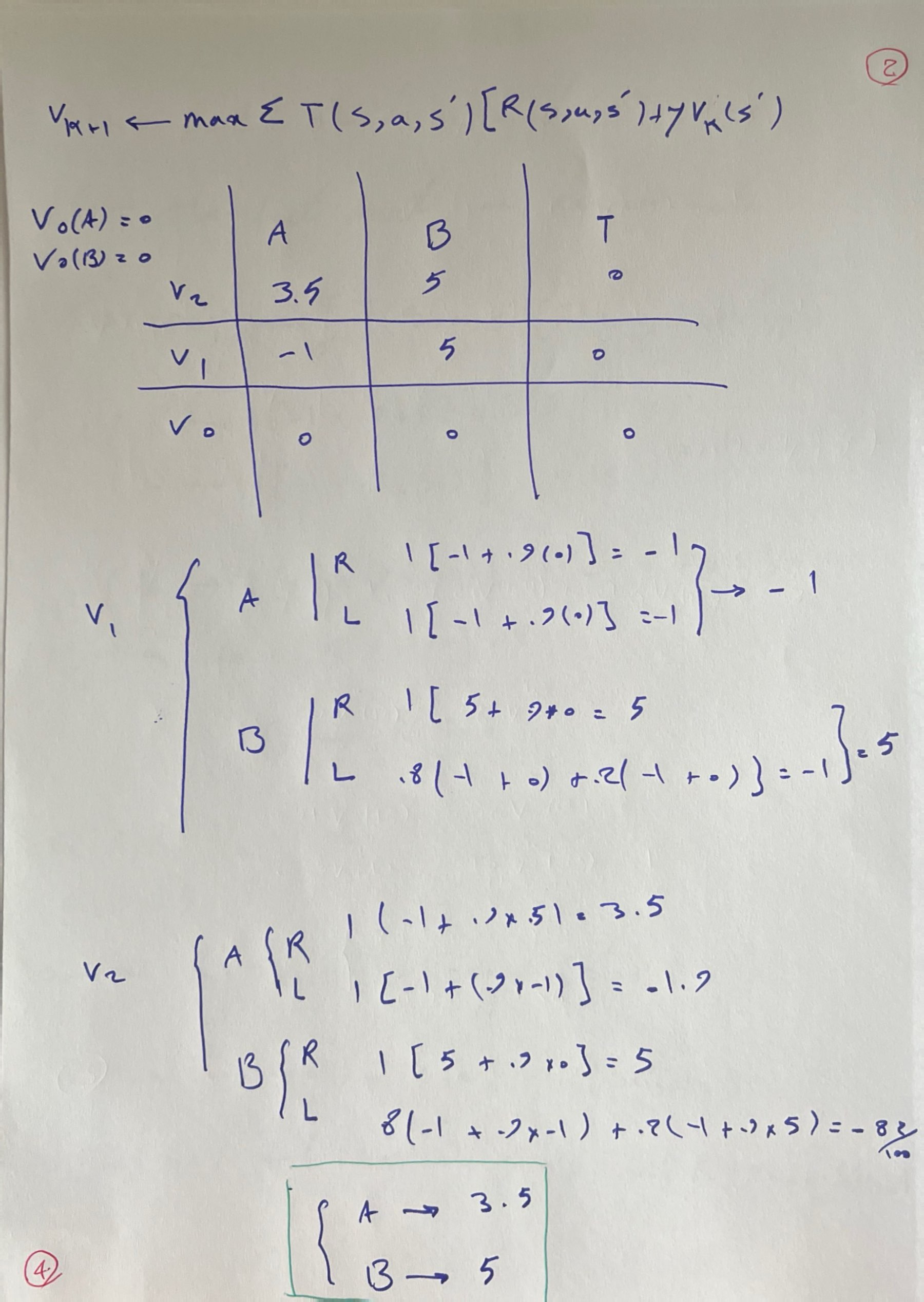
[2. Evaluation and GIF Output 21](#_Toc198916455)

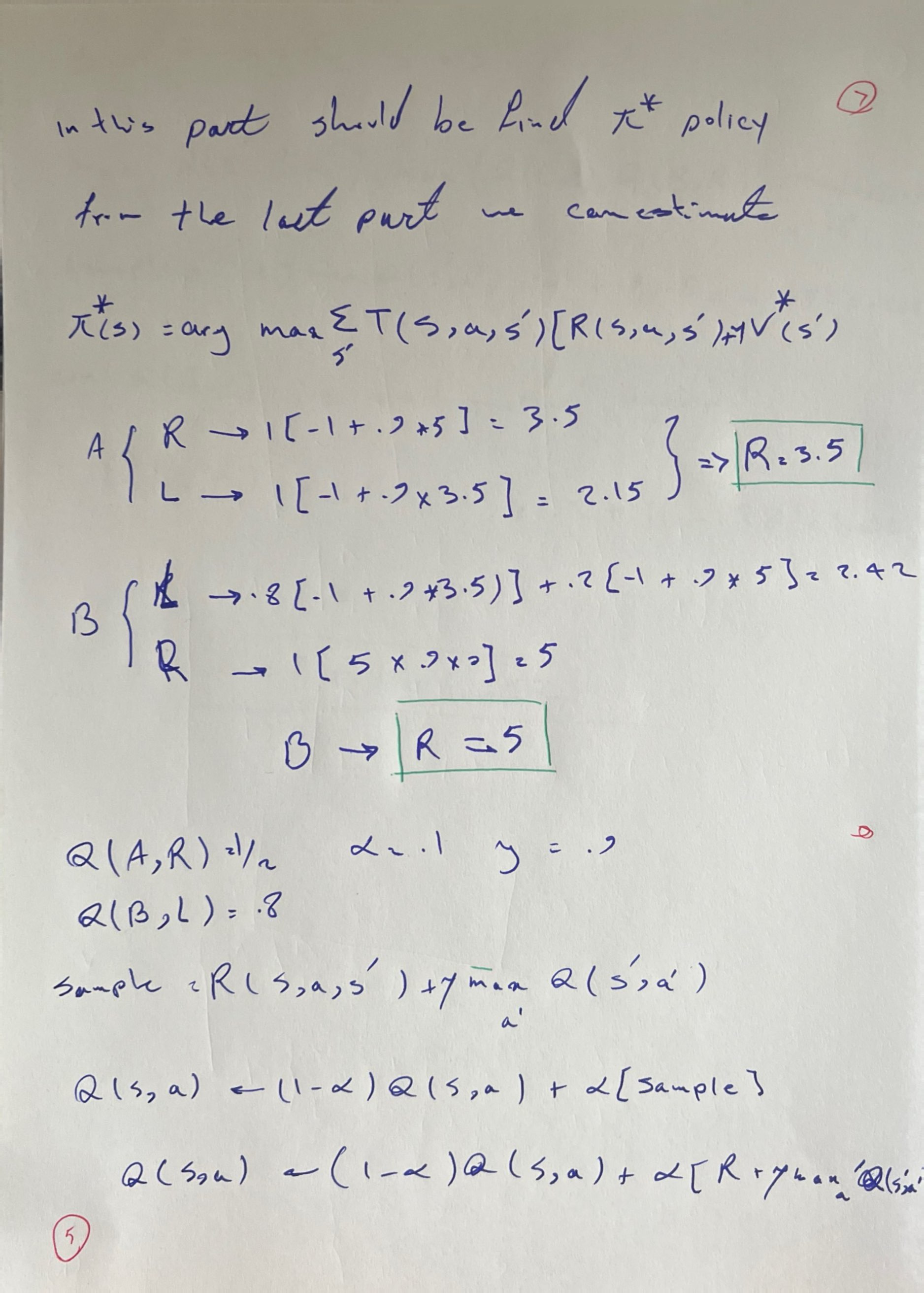
# Question 1:

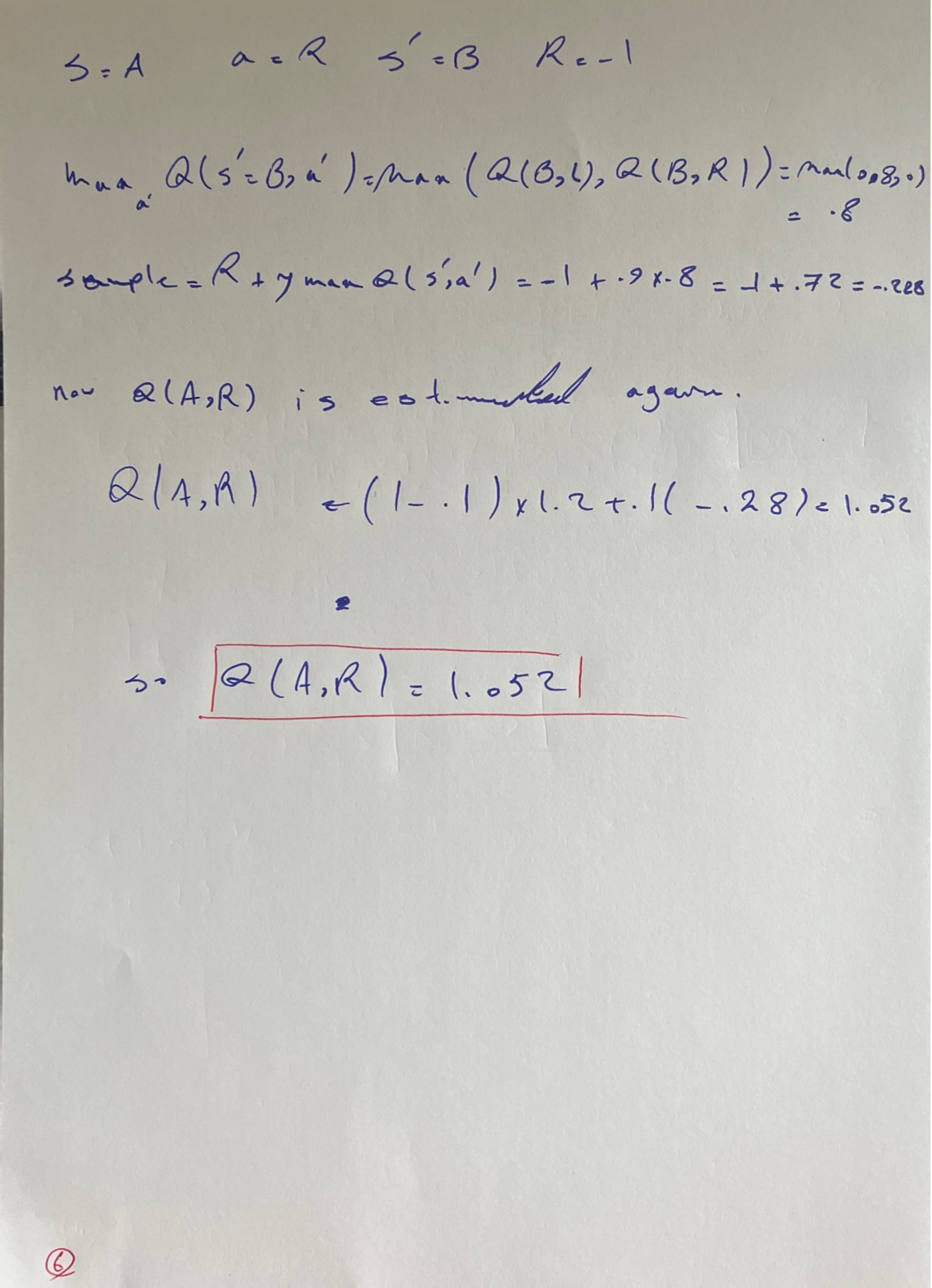










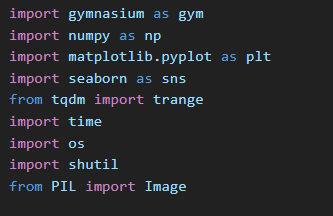


# Question 2

This tutorial provides a comprehensive explanation of a Q-learning-based agent trained to balance a pole on a cart using the CartPole-v1 environment from the Gymnasium library. The solution discretises the continuous state space, applies the epsilon-greedy strategy for exploration, and learns an optimal policy using the Q-learning algorithm.

## 1. Imports and Configuration

Essential libraries such as gymnasium, numpy, matplotlib, PIL, and tqdm are imported. Configuration settings include learning parameters like alpha (learning rate), gamma (discount factor), epsilon (exploration rate), and others like episode count and rendering preferences.



The Environment Setup is:



* "env\_name": "CartPole-v1"  
  Specifies the environment to use. In this case, it's the classic CartPole task.

These control the **Q-learning algorithm**:

* "alpha": 0.1  
  Learning rate — controls how much new information overrides old estimates.



* "gamma": 0.99  
  Discount factor — determines the importance of future rewards.  
  Closer to 1 → long-term rewards matter more.
* "epsilon": 1.0  
  Initial value for exploration probability — agent starts exploring randomly.
* "epsilon\_min": 0.01  
  Lower bound for epsilon — to ensure the agent doesn’t stop exploring entirely.
* "epsilon\_decay": 0.995  
  Decay rate of epsilon per episode — gradually favors exploitation over exploration.

These affect the training loop:

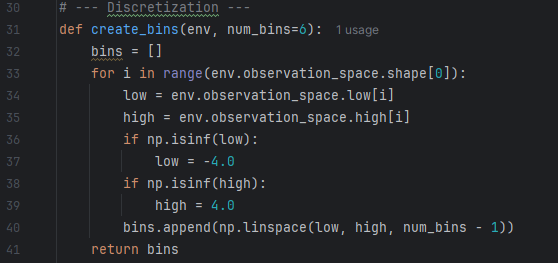
* "episodes": 1000  
  Total number of episodes to train.
* "max\_steps": 200  
  Max steps per episode — limits how long an episode can run.
* "reward\_threshold": 195.0  
  Used to determine early stopping — if the moving average of rewards reaches this value, training stops.

These settings affect output and display:

* "moving\_avg\_window": 50  
  The window size to compute the moving average of rewards for performance tracking.

## 2. State Discretisation

The CartPole environment provides continuous state variables. To use Q-learning, which requires a discrete state space, the 'create\_bins' function uses linspace to divide each dimension into equal bins. The 'discretize\_state' function then maps continuous observations to discrete indices using np.digitize.



 env.observation\_space.low[i] and high[i]: Get the min/max for each dimension.

 np.linspace(...): Divides each dimension into evenly spaced cut-points.

 Infinite bounds are clipped to -4.0 and 4.0 for numerical stability.

Converts a continuous state into a **tuple of bin indices** that represent a discrete state.



 np.digitize(...): For each element in the state, finds which bin it falls into.

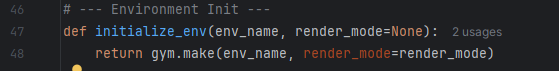
 zip(state, bins): Pairs each state component with its corresponding bin array.

 tuple(...): Converts the list of bin indices into a hashable tuple (used as a Q-table key).

## 3. Environment Initialisation

In reinforcement learning, the **environment** is the world the agent interacts with. For our CartPole example, we use the **Gymnasium** library (a modern version of OpenAI Gym) to simulate the CartPole environment.

The Function: initialize\_env



This function serves a simple but critical purpose: it **creates and returns an instance of the simulation environment** that the agent will interact with.

* **env\_name**: A string that tells Gym which environment to load.
  + Example: "CartPole-v1" is a classic control problem where the goal is to balance a pole on a cart by applying forces left or right.
* **render\_mode**: Optional string.
  + If set to 'rgb\_array', it enables frame capturing for later visualization (like a GIF).
  + If None, the environment won't render.

Internally, this line:

gym.make(env\_name, render\_mode=render\_mode)

calls the Gymnasium's **factory method** to create the environment. When this happens:

* The environment is initialised.
* The action space and observation space are defined.
* The state is set to its initial value.

1. **Separation of Concerns**: Isolating the environment setup into a function makes the code modular and reusable.
2. **Flexibility**: You can switch environments or render modes by changing a single config value.
3. **Support for Rendering**: Enables visualisation for debugging or presentation.

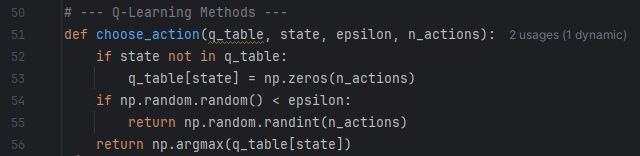
env = initialize\_env("CartPole-v1", render\_mode="rgb\_array")

This returns a CartPoleEnv object with rendering support enabled.

## Core Q-Learning Functions

### Epsilon-Greedy Strategy

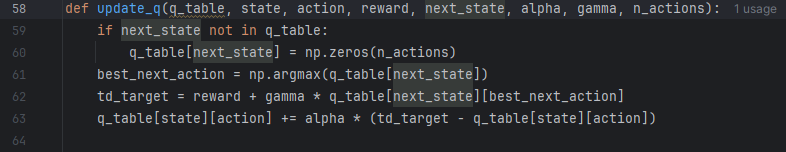
The 'choose\_action' function implements epsilon-greedy action selection. It explores randomly with probability epsilon or chooses the action with the highest Q-value otherwise. The 'update\_q' function applies the Bellman equation to update the Q-values using the reward and the maximum estimated value of the next state.



This function selects an action using the **epsilon-greedy policy** — a method that balances **exploration** (trying new actions) and **exploitation** (choosing the best-known action).

* If the state is not in the q\_table, it initialises it with a zero-value array.
* np.random.random() < epsilon: With probability epsilon, a **random action** is chosen (exploration).
* Otherwise, the **action with the highest Q-value** is chosen (exploitation).

### Q-Table Update Using Bellman Equation



 next\_state exists in q\_table.

 best\_next\_action: Find the best next action from the new state.

 td\_target: The estimated total reward for the current action:



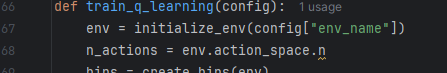
 td\_delta: The difference between target and current Q-value.

 Q-value is updated using the learning rate alpha

## Training the Agent

Training is the most critical phase in reinforcement learning. It is during this phase that the agent learns how to interact with the environment in order to maximise its reward. The training loop brings together all components—environment setup, action selection, Q-table updating, and performance monitoring—into a coherent learning process. Training in Q-learning involves letting the agent explore the environment repeatedly over multiple episodes. In each episode, the agent makes decisions based on its **Q-table**, receives feedback in the form of **rewards**, and updates its knowledge so that its decisions improve over time.

The function responsible for training is train\_q\_learning(config). In 'train\_q\_learning', the agent runs for several episodes. Each episode involves selecting actions, observing rewards, and updating the Q-table. Epsilon decays over time to shift from exploration to exploitation. Rewards and moving averages are tracked for monitoring performance.



 The environment is initialised using Gym.

 The continuous state space is divided into discrete bins for learning.

 An empty Q-table (dictionary) is created to store learned values.

for episode in range(config["episodes"]):

* The agent goes through multiple episodes (simulated trials).
* Each episode is a new attempt to complete the task.

state = discretize\_state(env.reset()[0], bins)

 The environment is reset to the start.

 The initial state is discretised for compatibility with the Q-table.

for \_ in range(config["max\_steps"]):

* The agent takes a series of actions in the environment.
* Each step involves decision-making, learning, and possibly ending the episode.

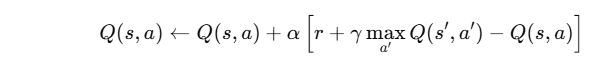
action = choose\_action(q\_table, state, epsilon, env.action\_space.n)

 The action is executed in the environment.

 The agent receives a reward and new state information.

update\_q(q\_table, state, action, reward, next\_state, alpha, gamma, ...)

The Q-table is updated using the Bellman Equation:

****

* This is the core of the learning process.

state = next\_state

total\_reward += reward

* The agent moves to the next state.
* It accumulates the reward received in this episode.

if terminated or truncated:

break

* The loop ends if the environment signals the episode is over (e.g., the pole fell).

rewards.append(total\_reward)

moving\_avg.append(np.mean(rewards[-window:]))

* Track how well the agent performed in each episode.
* Moving average gives a smoother performance measure.

epsilon = max(epsilon\_min, epsilon \* epsilon\_decay)

Reduces exploration over time, allowing the agent to rely more on its learning.

if avg\_reward >= reward\_threshold:

break

If the agent consistently performs well, training stops early to save time.

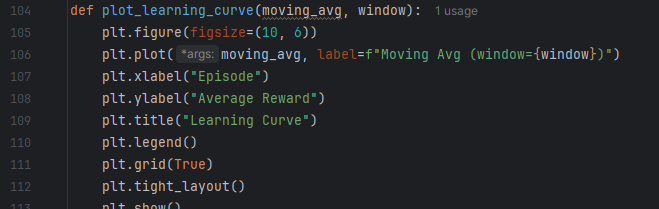
return q\_table, bins, rewards, moving\_avg

Returns everything needed for:

* Evaluating the agent.
* Plotting learning curves.
* Visualising the policy.

## Visualising Learning

The 'plot\_learning\_curve' function uses matplotlib to visualise the agent's learning progress over episodes. It shows the moving average of total rewards, helping identify convergence.

****

## Evaluation and GIF Creation

The 'evaluate\_agent' function tests the trained agent, rendering each step and optionally recording frames. It compiles these into a GIF using the PIL library, enabling a visual inspection of agent

behaviour.

env = initialize\_env(config["env\_name"], render\_mode="rgb\_array")

 Initializes the environment in **rendering mode**, allowing us to capture image frames.

 render\_mode="rgb\_array" tells Gym to return image data for each frame.

state = discretize\_state(env.reset()[0], bins)

done = False

frames = []

 Reset the environment to start the episode.

 Convert the initial continuous state into a discrete one using the bins.

 Prepare an empty list to store image frames.

while not done:

action = np.argmax(q\_table.get(state, np.zeros(env.action\_space.n)))

 Use the **greedy policy**: always pick the action with the highest Q-value.

 No exploration here—just exploitation of learned knowledge.

obs, reward, terminated, truncated, \_ = env.step(action)

state = discretize\_state(obs, bins)

 Perform the action and get the resulting observation and reward.

 Update the state.

if config["record\_frames"]:

img = env.render()

frames.append(Image.fromarray(img))

 If recording is enabled, capture the current frame.

 Convert the array (RGB data) into a PIL image for GIF creation.

if config["render\_sleep"]:

time.sleep(config["render\_sleep"])

* If render\_sleep is set, wait a little between frames (for visual effect).

if config["record\_frames"]:

if not os.path.exists(config["frame\_dir"]):

os.makedirs(config["frame\_dir"])

frames[0].save(

os.path.join(config["frame\_dir"], config["gif\_name"]),

save\_all=True,

append\_images=frames[1:],

duration=int(config["render\_sleep"] \* 1000),

loop=0

)

* After the episode ends, save all captured frames as a GIF.
* duration controls the time between frames.
* loop=0 makes the GIF repeat forever.

## Execution

The script ends by calling training, plotting, and evaluation functions. If configured, it generates a final animated GIF demonstrating the trained policy in action.

After defining all the individual components—discretization, Q-learning logic, training loop, evaluation, and visualisation—it’s time to **execute** the full pipeline. This final step **ties everything together** and runs the agent end-to-end from training to performance demonstration.

# Train the agent and get results

q\_table, bins, rewards, moving\_avg = train\_q\_learning(CONFIG)

# Plot the learning curve

plot\_learning\_curve(moving\_avg, CONFIG["moving\_avg\_window"])

# Evaluate and visualize if required

if CONFIG["render\_final"]:

evaluate\_agent(q\_table, bins, CONFIG)

q\_table, bins, rewards, moving\_avg = train\_q\_learning(CONFIG)

* Trains the agent using Q-learning.
* Returns:
  + q\_table: The learned Q-values.
  + bins: Discretization bins for mapping continuous states.
  + rewards: Episode-wise reward totals.
  + moving\_avg: Smoothed version of rewards for visualization.

plot\_learning\_curve(moving\_avg, CONFIG["moving\_avg\_window"])

* Uses matplotlib to plot a graph of the agent’s performance.
* Helps verify whether the agent improved over episodes.

if CONFIG["render\_final"]:

evaluate\_agent(q\_table, bins, CONFIG)

* If enabled in the config, runs the agent one last time in visual mode.
* Optionally records the frames to create a GIF (cartpole\_result.gif).

For better structure and modularity, you could wrap execution in a Python main() function:

def main():

q\_table, bins, rewards, moving\_avg = train\_q\_learning(CONFIG)

plot\_learning\_curve(moving\_avg, CONFIG["moving\_avg\_window"])

if CONFIG["render\_final"]:

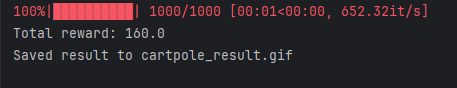
evaluate\_agent(q\_table, bins, CONFIG)

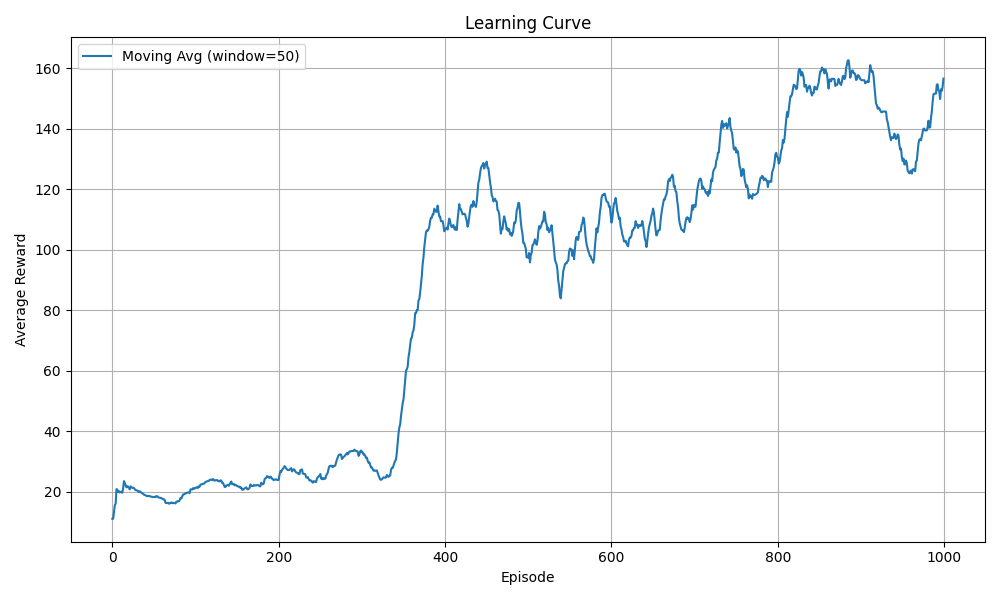
if \_\_name\_\_ == "\_\_main\_\_":

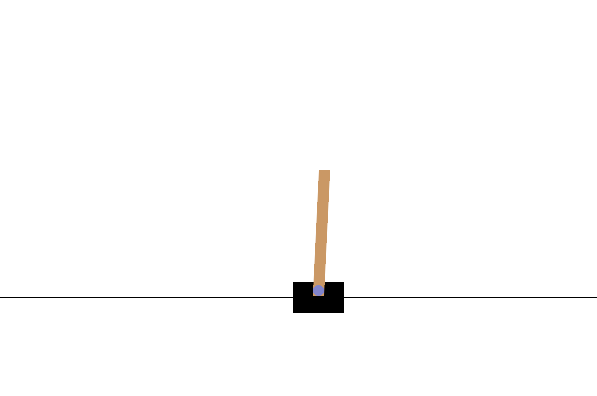
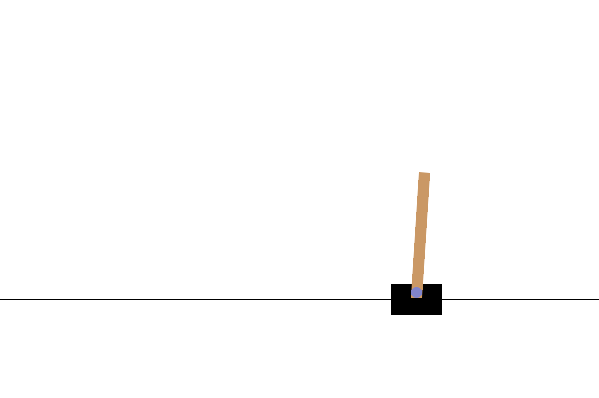
main()

## Conclusion

This implementation demonstrates a complete, educational approach to reinforcement learning using Q-learning in a discrete space. It offers valuable insights into environment interaction, state handling, exploration-exploitation trade-offs, and performance visualisation.





After executing the Q-learning agent for the **CartPole-v1** environment, we can now analyse the outcomes using the generated **learning curve** and **evaluation summary**.

### 1. Learning Curve Interpretation

The **Y-axis** represents the **moving average of rewards** over episodes (window = 50).

* The **X-axis** indicates the **episode number** (from 0 to 1000).
* At the beginning (episodes < 400), the average reward is low and noisy (around 20–40), indicating random exploration.
* Around episode **400**, there is a **sharp increase** in performance, suggesting the agent has started to learn a better policy.
* From episode **500 onwards**, the average reward stabilizes and gradually increases, approaching **160+**.
* This trend demonstrates that the agent consistently improves its ability to balance the pole for longer durations.
* The Q-learning strategy, combined with discretisation and epsilon-greedy exploration, has been **successful**.
* The agent was able to generalise and sustain high reward levels, though it did not fully reach the perfect threshold of **195.0** set in the configuration.

### 2. Evaluation and GIF Output

Final test episode produced a **total reward of 160.0**, meaning the agent kept the pole balanced for 160 steps.

* The agent's performance was **recorded and saved as a GIF** file: cartpole\_result.gif.
* The GIF serves as a **qualitative assessment** of the trained policy.
* You can visually confirm if the agent behaves as expected (e.g., stabilizing the pole and correcting quickly).
* This visualization is essential in reinforcement learning for verifying correctness beyond just numerical reward.

This exercise demonstrates the **full pipeline of building a Q-learning agent**, from **state discretisation** to **policy evaluation** and **visualisation**. This exercise reinforces key reinforcement learning principles and provides a strong foundation for applying RL to more complex problems.

Key takeaways:

|  |  |
| --- | --- |
| Aspect | Summary |
| Learning Progress | Agent improved steadily, reaching consistent performance (~160 reward). |
| Algorithm Effectiveness | Q-learning with discretisation and epsilon decay worked effectively. |
| Visualization | Learning curve and final GIF provided both numeric and visual insights. |
| Room for Improvement | While not perfect, further tuning or using Deep Q-Networks (DQN) could enhance performance. |