INFO 4000
Informatics III

Data Science Specialization - Advanced

MQTT and Reinforcement Learning

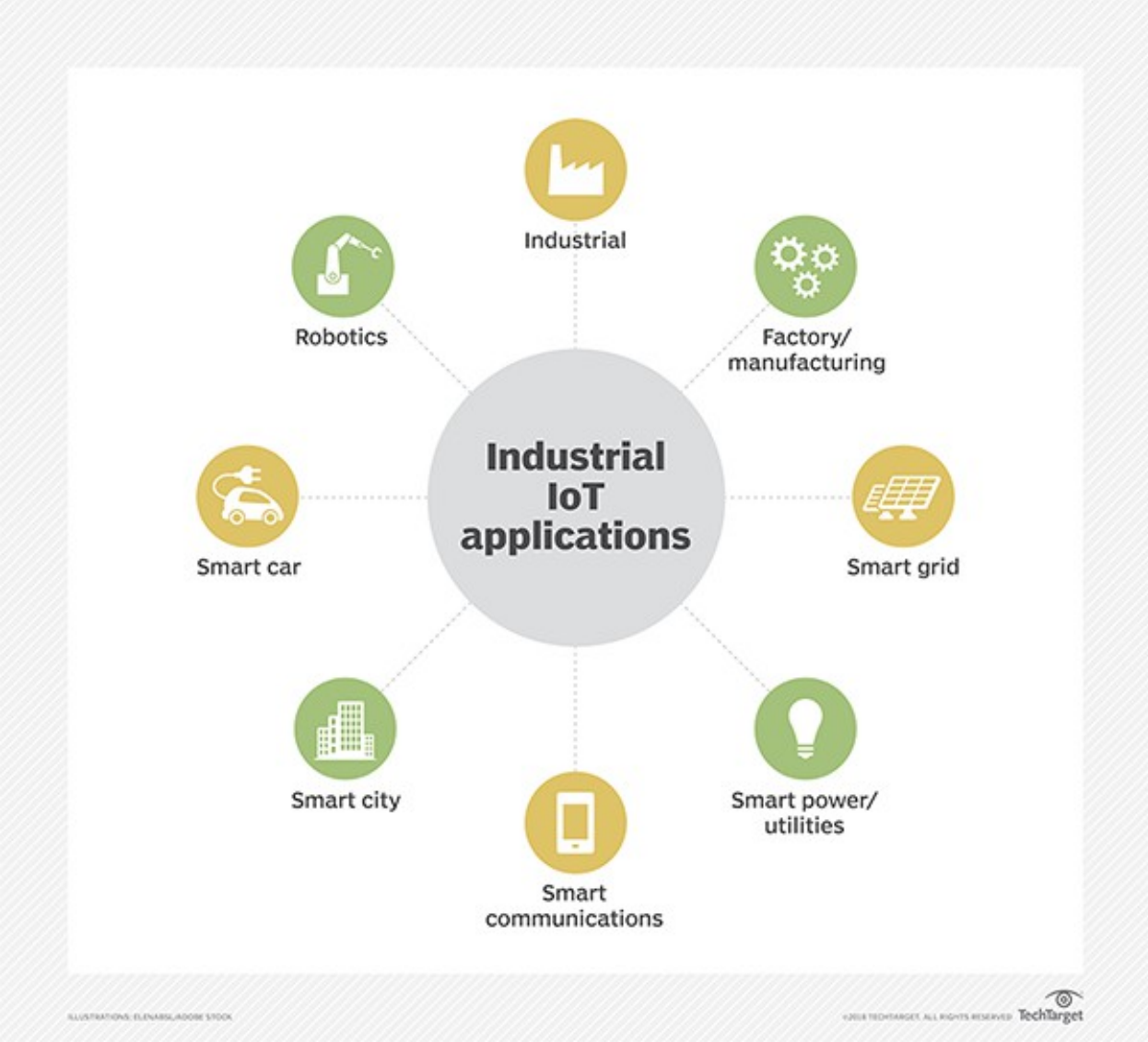
Course Instructor

Jagannath Rao

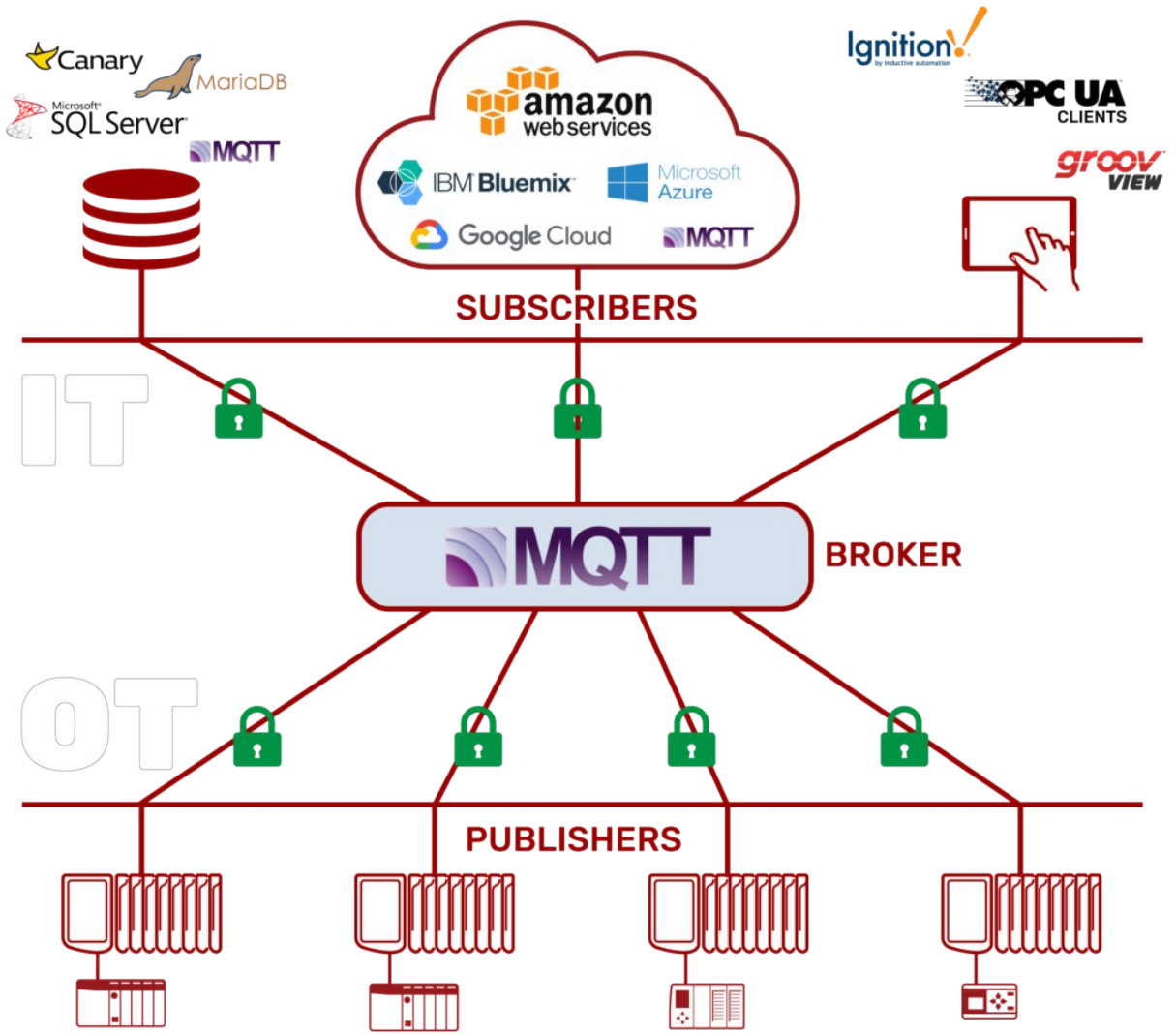
raoj@uga.edu

Device connectivity and communication – MQTT

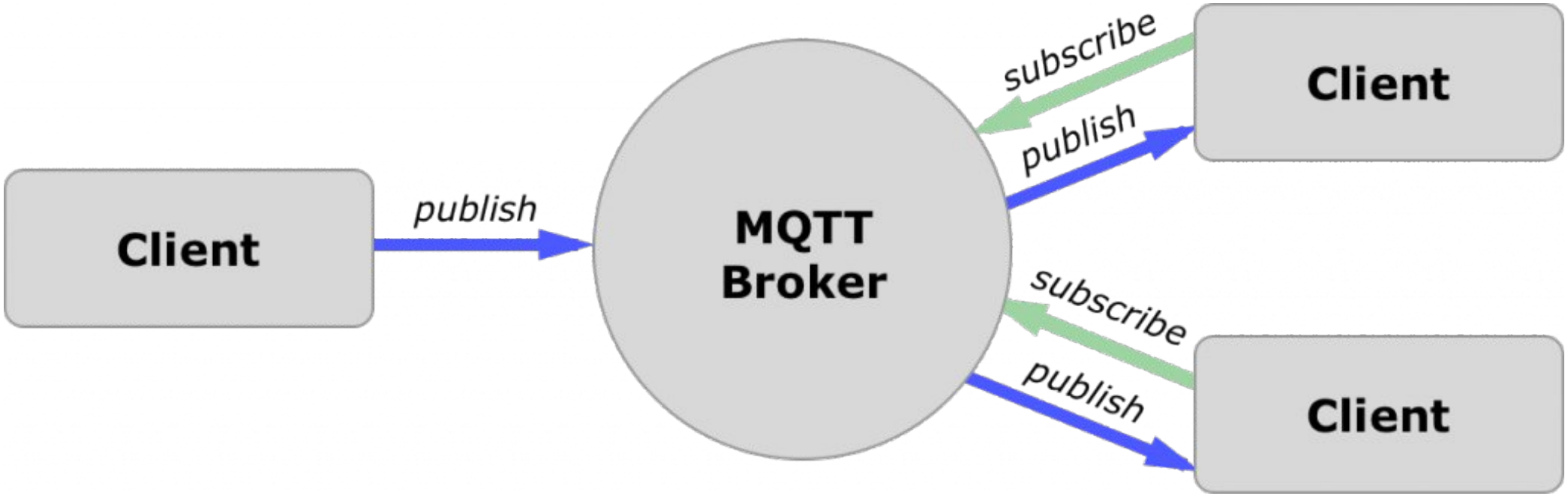
The realm of IIoT applications



Connectivity



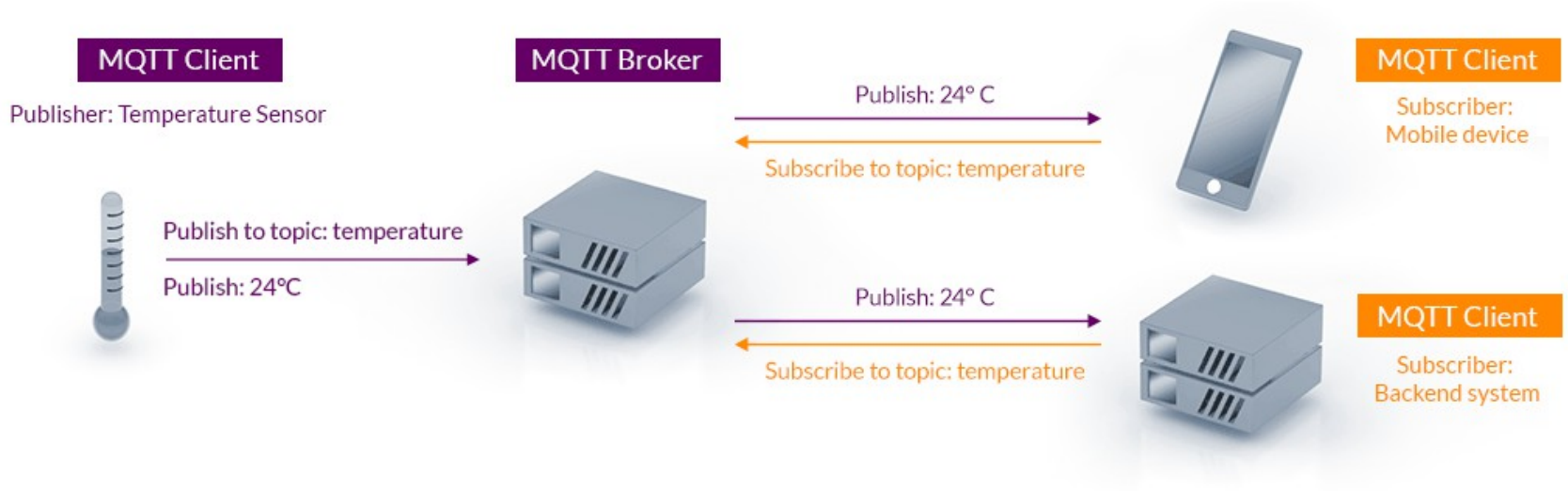
Broker, Publish (pub) and Subscribe (sub)



Communication protocols - MQTT

What is MQTT:

1. Message Queuing Telemetry Transport.
2. The Standard for IoT Messaging.
3. Lightweight and efficient.
4. Bi-directional communications
5. Scalable to a million connected things
6. Reliable message delivery.
7. Security enabled
8. Support for unreliable networks.



https://mqtt.org/getting-started/

Python and MQTT

1. While brokers are usually in the cloud (to enable global connectivity) and all of them are commercial.

2. The broker we use which is a popular one is called “Mosquitto” - https://mosquitto.org/ (you can download and install) and free.

3. We can also use their cloud version for connectivity beyond our PCs – ‘test.mosquito.org’ – this is good enough for our simple projects.

4. **MQTT client** - The paho MQTT python client from Eclipse supports MQTT, and works with Python 3.x.

5. Installing the MQTT client module –
   conda install -c conda-forge paho-mqtt (preferred)
   pip install paho-mqtt

6. With a broker and a client, we can now communicate between devices using MQTT.

How to implement MQTT - 1

1. The paho MQTT is a simple to use and well featured Python library for communicating using MQTT protocol.

2. We also need a subscriber to receive the messages published.

3. There are two functions that are essential. **on_connect** is called when the client connects to the broker successfully.

4. **on_message** is called when the broker pushes a message to the client.

5. The client registers these as callback methods and then connects to the broker. Once connected, we subscribe to the interest topic and start an infinite loop waiting for messages.

6. Callback functions in Python are a function calling a function and is often event driven – asynchronous.

```
import paho.mqtt.client as mqtt

client = mqtt.Client()
client.connect("localhost", 1883, 60)
topic = 'my_topic'
client.publish(topic, payload='on')
```

```
import paho.mqtt.client as mqtt

# The callback for when the client receives a CONNACK response from
the server.
def on_connect(client, userdata, flags, rc):
    print(f"Connected with result code {str(rc)}")

# The callback for when a PUBLISH message is received from the
server.
def on_message(client, userdata, msg):
    print(f"{msg.topic} {str(msg.payload)}")

if __name__ == '__main__':
    client = mqtt.Client()
    client.on_connect = on_connect
    client.on_message = on_message

    client.connect("localhost", 1883, 60)

    topic = 'my_topic'
    client.subscribe(topic)

    client.loop_forever()
```

MQTT related concepts

1. The 'payload' holds the data related to the packet being sent. For a PUBLISH packet, this is the actual message being sent.

2. If there are no subscribers listening when a message is sent to the broker, then the broker will discard this.

3. This is actually very useful as devices such as sensors can send a large number of messages over time. If there are no subscribers, then there’s no point storing all the messages and then overloading the subscriber when it connects.

4. Most likely in those cases, the older messages need to be ignored as newer measurements are more accurate and complete.

5. Considering the use case of low connectivity or high rate of connection loss between devices and broker, one nice feature to have would have been: to be able to recall the latest sensor value (or the last message) so we can restore any required state.

6. For this, the broker allows retaining the last message for any new subscribers to consume.

```
client.publish(topic, payload='on', retain=True)
```

MQTT Hands on

Explanation

1. **on_connect:** This function gets triggered when the client successfully connects to the MQTT broker. After connection, it subscribes to the MQTT topic.

2. **on_message:** This function is called whenever a new message is received on the subscribed topic. It prints the message payload and topic name.

3. **client.connect:** Connects to the broker at the specified host (test.mosquito.org is a public broker for testing) and port (1883 is the default for MQTT).

4. **client.loop_forever:** Keeps the subscriber running indefinitely to receive messages.

Intro to Reinforcement Learning(RL)

Definition of Reinforcement Learning (RL)

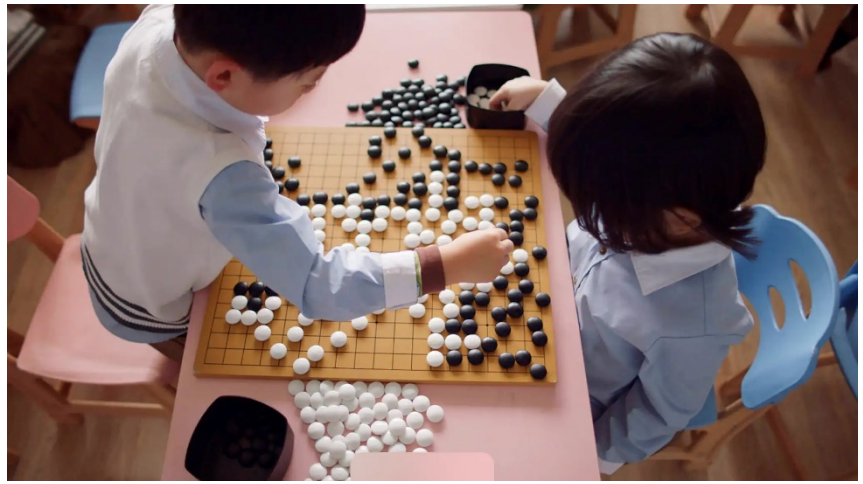
Reinforcement Learning is a way to teach machines to make decisions by rewarding good actions and penalizing bad ones.

OR

The method of ensuring that a system, when in its current state, can take the right action to maximize the returns it can get.

Making History

AlphaGo is the first computer program to defeat a professional human Go player, the first to defeat a Go world champion, and is arguably the strongest Go player in history.
Go originated in China over 3,000 years ago. Winning this board game requires multiple layers of strategic thinking.



Aloha: DeepMind has published research on using RL to train robot arms for dexterous manipulation tasks.

***Technique**: Agents learn to manipulate objects by controlling a robotic hand, often in simulation, to perform complex, fine-motor tasks.*
***Significance**: This research is critical for developing more capable robots for applications in manufacturing, logistics, and assistive technology.*

Real world applications

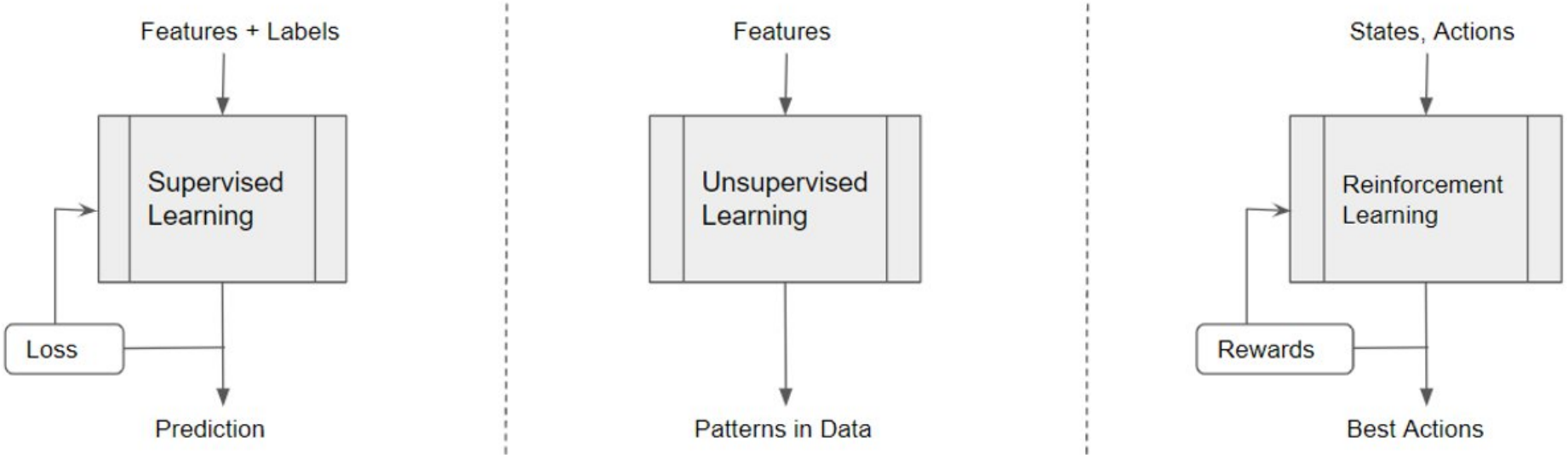
Data Center efficiency: DeepMind developed and deployed an AI-powered system, primarily using reinforcement learning, to optimize the energy efficiency of cooling systems in Google's data centers. This initiative aimed to significantly reduce the substantial energy consumption associated with keeping data center equipment at optimal operating temperatures.

Significant Energy Reduction:
The DeepMind AI system consistently achieved substantial reductions in the energy used for cooling, with reports indicating a 40% reduction .

Safety-First Approach:
The system is designed with a "safety-first" approach, ensuring that all adjustments maintain the data center within safe operating temperatures and conditions, preventing equipment damage or downtime.

Learning and Adaptation:
Unlike traditional rule-based systems, the reinforcement learning agent continuously learns and adapts to changing conditions, such as seasonal variations or fluctuating server loads, leading to ongoing improvements in efficiency.

Where does RL fit in?

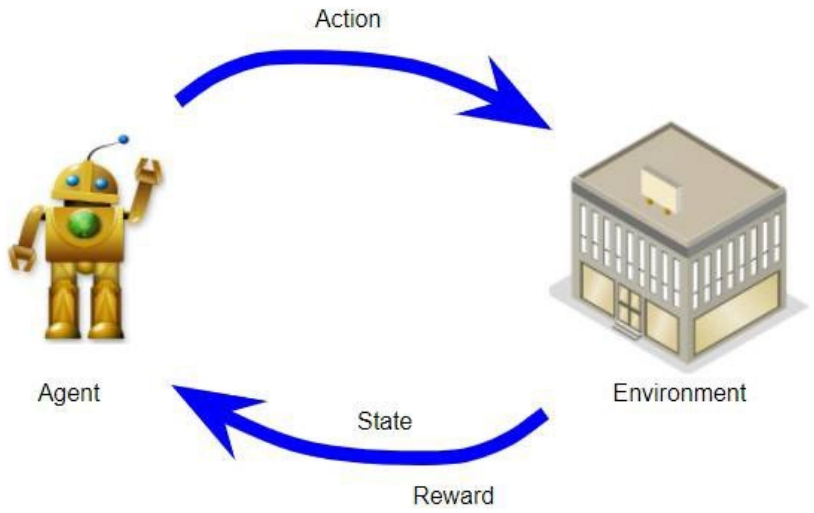


Rather than the typical ML problems such as Classification, Regression, Clustering and so on, RL is most used **to solve a different class of real-world problems, such as a Control task or Decision task,** where you operate a system that interacts with the real world.

- Operating a drone or autonomous vehicle
- Manipulating a robot to navigate the environment and perform various tasks
- Managing an investment portfolio and taking trading decisions
- Playing games such as Go, Chess, video games

What is RL?

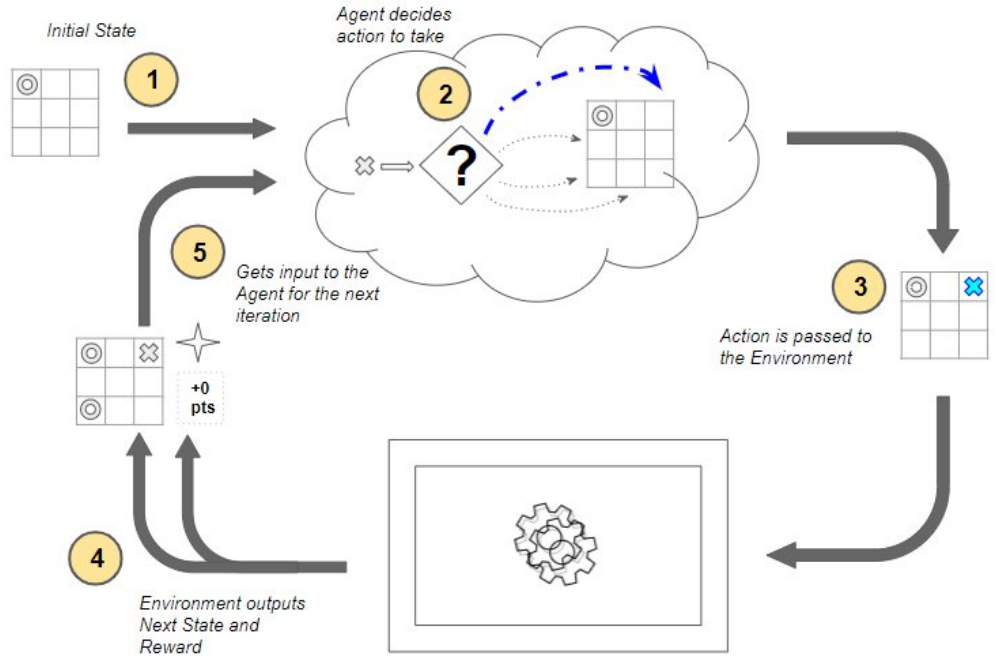
- In RL, the learning happens from experience by trial and error
  - A dog is trained to do various things
  - It does not learn unless there is a reward and no reward system (Treats)
  - The training happens in repeated trials
  - Once trained it remembers and it knows the policy
  - This is RL
- The agent that is being trained interacts with a real / simulated world and makes observations / get's rewards (+ve or –ve).



Tic Tac Toe – An example of how RL operates

The sequence starts with an initial state, which becomes the current state. For instance, your opponent, the environment has placed their token in a particular position, and that is the starting state for the game.

- The environment's current state is input to the agent.
- The agent uses that current state to decide what action it should take. It does not need a memory of the full history of states and actions that came before it.
- The agent decides to place its token in some position. There are many possible actions to choose from, so how does it decide what action to take? (later)
- That action is passed as input to the environment.
- The environment uses the current state and the selected action and outputs two things — it transitions the world to the next state, and it provides some reward.
- This reward from the environment is then provided as feedback to the agent as a consequence of the previous action.
- This completes one time-step and moves us to the next time-step. This next state now becomes the current state which is then provided to the agent as input, and the cycle repeats.
- Throughout this process, it is the agent's goal to maximize the total amount of rewards that it receives from taking actions in given states. It wants to maximize not just the immediate reward, but the cumulative rewards it receives over time.



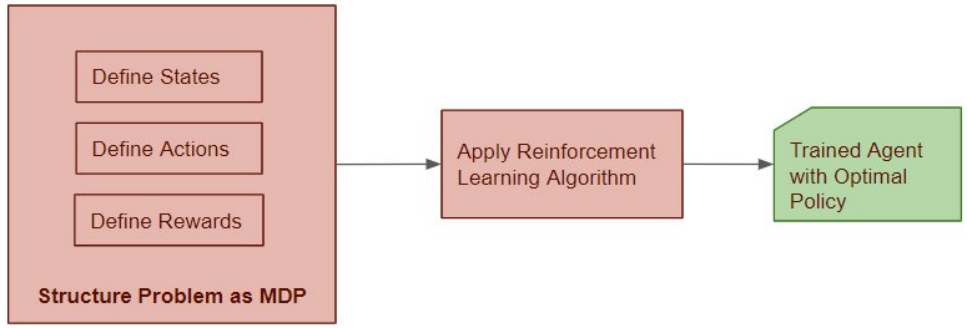
<u>Rewards System:</u>
Win = +10
Lose = -10
In between steps = 0

Key components of an RL system

- **Agent:** The learner or decision-maker. It could be a robot, a software program, a plant control system, autonomous vehicle or even a virtual character in a game.
- **Environment:** The world the agent interacts with. It could be a physical space, a simulated environment, or even a set of rules and constraints.
- **Actions:** The choices the agent can make. These could be movements, decisions, or any other type of interaction with the environment.
- **Rewards:** The feedback the agent receives after taking an action. Positive rewards encourage good behavior, while negative rewards (or penalties) discourage bad behavior.
- **State:** The current situation or context the agent is in. This could include the agent's position, the state of the environment, or any other relevant information.

Solving the RL Problem by Finding the Optimal Policy

- We structure our problem as an MDP (Markov Decision Process) and we can then solve this problem by building an agent.
- The agent operates in a way that it can make decisions about which action to take.
- The agent should do this in a way that maximizes Returns.
- In other words, we need to find the Optimal Policy for the agent. Once it has the Optimal Policy it simply uses that policy to pick actions from any state.
- We apply a Reinforcement Learning algorithm to build an agent model and train it to find the Optimal Policy.
- Finding the Optimal Policy essentially solves the RL problem.
- A lot of Reinforcement Learning problems with discrete actions are modeled as **Markov decision processes**,
  - with the agent having no initial clue on the next transition state.
  - the agent also has no idea on the rewarding principle,
  - So, it has to explore all possible states to begin to decode how to adjust to a perfect rewarding system.



Applications in real life

Robotics

- Reinforcement learning trains robots to perform tasks requiring fine motor skills, such as object manipulation, and more complex tasks, like autonomous navigation.
  - Pick and place robots which get negative rewards every time they pick a wrong part or put them in the wrong bin

Power Systems

- **Train electrical management system** (EMS) agent for optimizing operation of microgrid and maximize profit from PV power plants and batteries installed on households inside the observed microgrid.
  - An EMS agent gets positive rewards for right choice of action, quantified with profit in $.

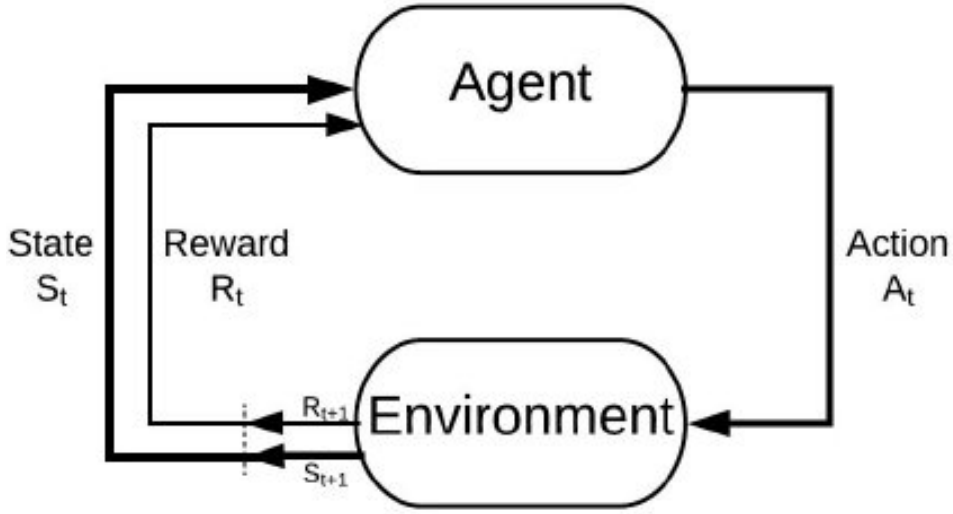
Autonomous Vehicles

- Reinforcement learning is crucial in developing autonomous vehicles, enabling them to learn from their environment
  - **Traffic Signal Adherence:** RL agents are trained to recognize and respond appropriately to traffic signals. They learn that stopping at a red light leads to a positive reward (safety) while running a red light leads to a negative reward (potential accident).

Reinforcement Learning Algorithms

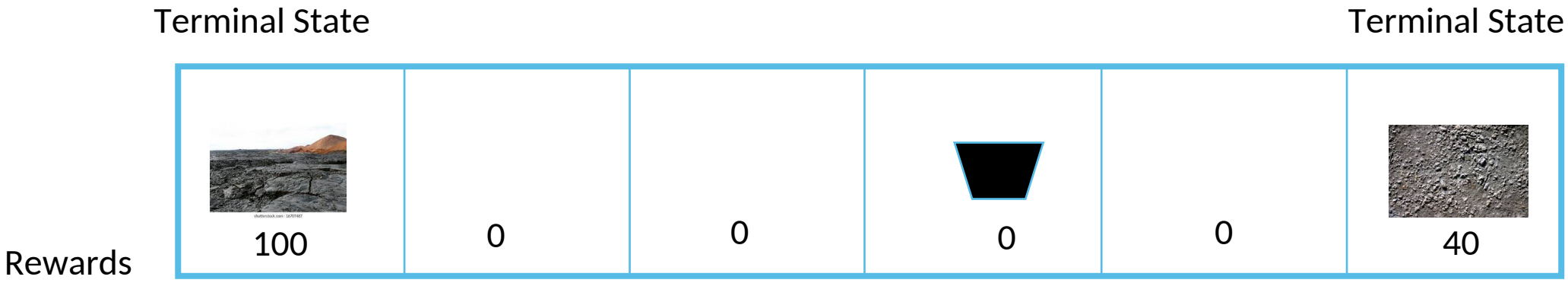
Algorithms in RL

- Model Based Algorithms (aka Planning)
  - Model-based approaches are used when the internal operation of the environment is known. In other words, we can reliably say what Next State and Reward will be output by the environment when some Action is performed from some Current State.
- Model Free Algorithms (aka Reinforcement Learning)
  - Model-free approaches are used when the environment is very complex, and its internal dynamics are not known. They treat the environment as a black-box.
- Think of it this way,
  - if the agent can predict the reward for some action before actually performing it thereby planning what it should do, the algorithm is model-based.
  - while if it actually needs to carry out the action to see what happens and learn from it, it is model-free.
- The algorithm we will learn, Q-Learning, is a model free type.
- The ‘Q’ in Q-learning stands for quality. Quality here represents how useful a given action is in gaining some future reward.



Returns - how to compute them

Example of Agent operating in an environment:
What should this mineral explorer do?



Some possible operations of the Agent and corresponding reward

Go left:

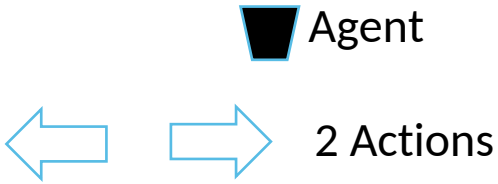
States:	4	3	2	1
Rewards:	0	0	0	100

Go right:

States:	4	5	6
Rewards:	0	0	40

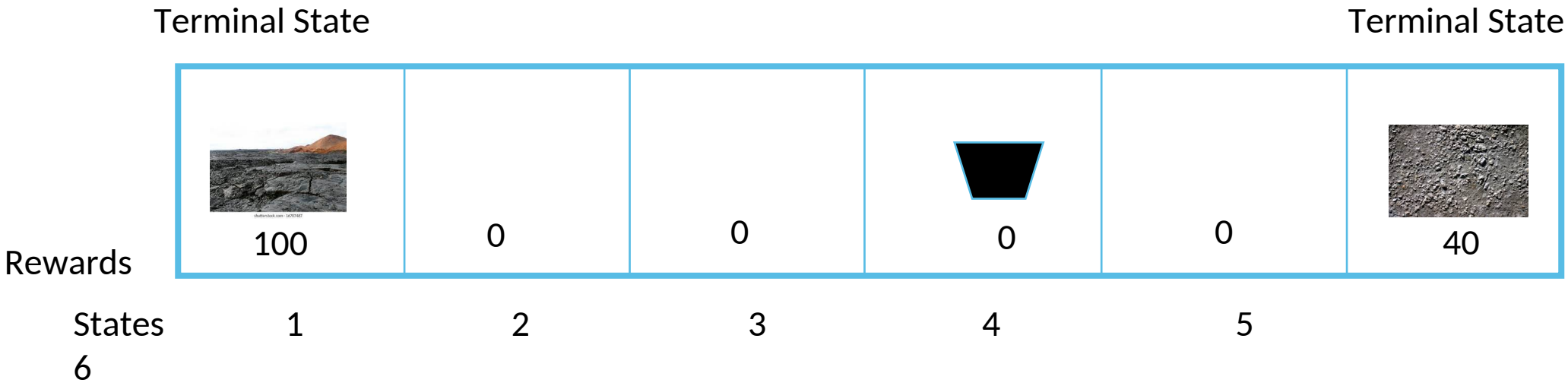
Other:

States:	4	5	4	3	2	1
Rewards:	0	0	0	0	0	100



Episode ends at Terminal State

Return and Reward are different in RL



- Reward is immediate
- Return is the long-term gain
- Like – cash in hand v/s in the future. Present Value of future return is computed as FV * discounting factor (example interest rate).
- What would prefer – get cash now or in the future?

Computing Return

Go left:

State: 4 3 2 1

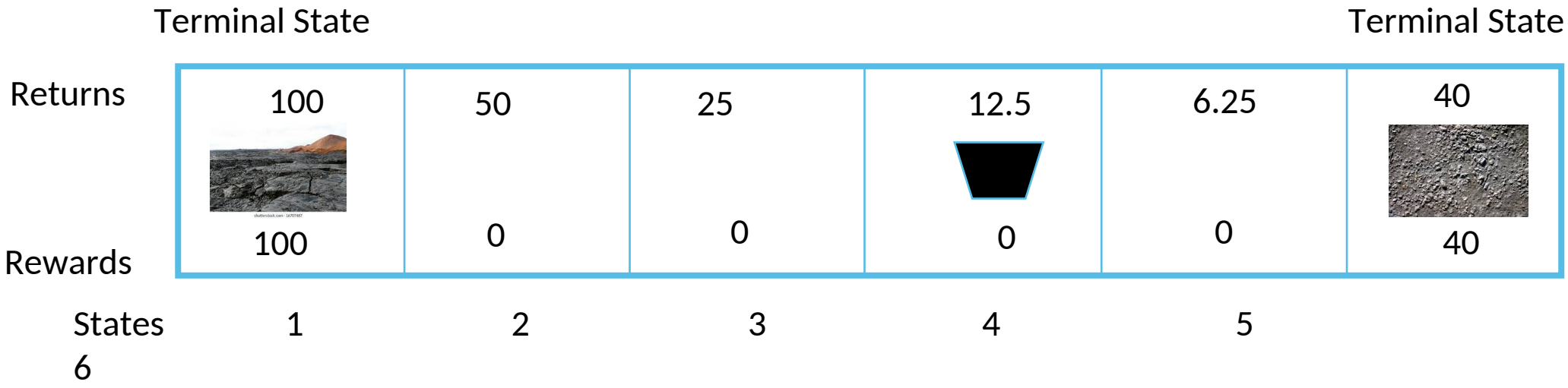
Return: 0 + (0.9)0 + (0)(0.9)^2 + (0.9)^3 100 = 72.9

Generally:

Return = $R1 + g\,R2 + g^2\,R3 + g^3\,R4$

where, ‘γ’ is the discount factor.

Return … Going Left

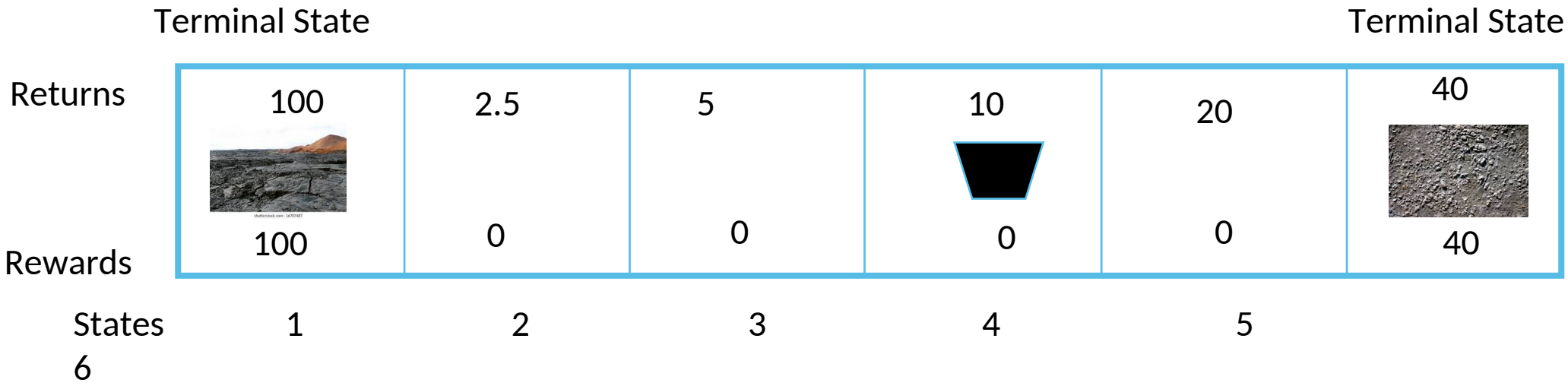


Computing Return

Go left from state 4 and g = 0.5:

Return: 0 + (0.5)0 + (0.5)^2(0) + (0.5)^3 (100) = 12.5

Return is different for different actions … Going Right



Computing Return

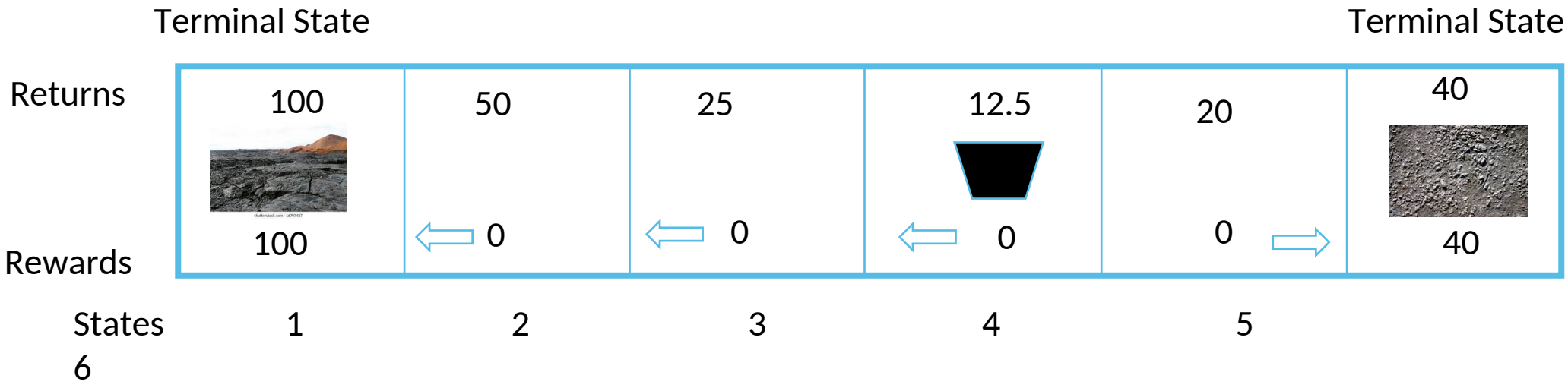
Go right from state 4 and g = 0.5:

Return: 0 + (0.5)0 + (0.5)^2(40) = 10

Always go to the right generates less returns in most states. Therefore, not the best option.

More optimal would be – if in 3 go left, if in 2 go left, if in 5 go right

Actions for optimal returns



Summary

- Example – if we start in state 5 and take the action right, the return in this state from taking the action is 20.
- In RL the return the agent gets is the sum of the reward the system gets but weighted by a discount factor.
- The discount factor when applied to negative rewards has the effect of pushing the reward more into the future
- Think about when you owe money to someone, and you postpone those payments – that money is worth less in the future.

Decision making using a Policy

- Goal of RL is to find a policy function which takes states as inputs and tells what action to take for max returns.
- The policy in RL is given the expression – p
- Mathematically: s ☾ p ☾ action
- In the states in the picture the policy would be for the discount factor 0.5:

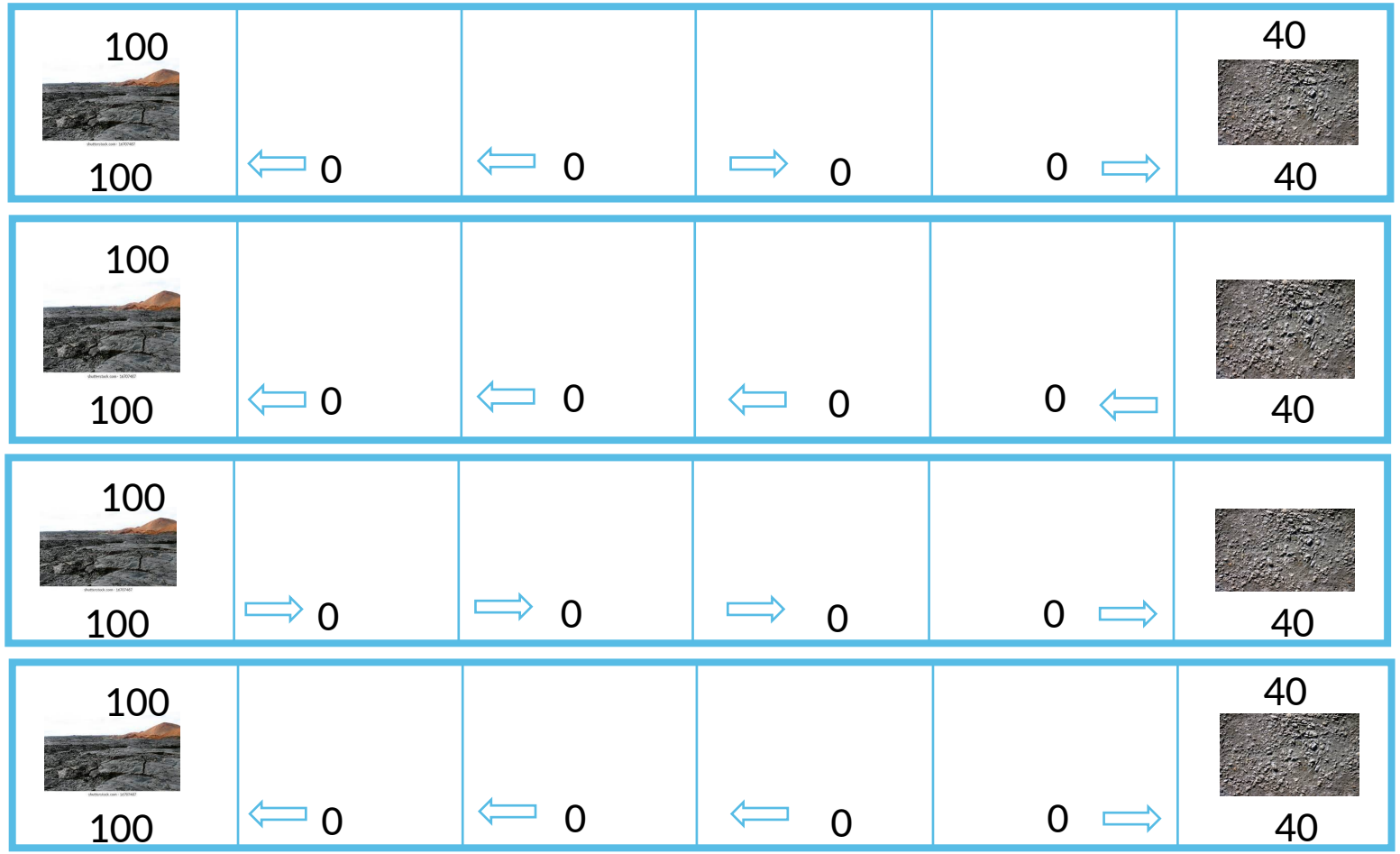
p (s) = a

p (2) = ⇝

p (3) = ⇝

p (4) = ⇝

p (5) = ☾



State Action-Value function (Q-function)

- **Q(s,a)** is called the state action-value function and is defined as the return you get if you :
  - Start in a state ‘s’
  - Take an action ‘a’ (once),
  - Then behave optimally after that

- Therefore, whatever action you take after that first step will always fetch the max return.

State Action-Value function (Q-function) - Example

100  100	50 12.5 0	25 0 0	12.5 0 0	0 0 0	40  40

- Let us compute Q(s,a) for starting in state 2 and taking the first action right and then going left.
- **Q(s,a) = 0 +0.5(0) + 0.5^2(0) + 0.5^3(100) ☾ 12.5**
- Let us compute going left from state 2: **Q(s,a) = 0 + 0.5(100) ☾ 50**
- In another case, start from state 4 and go left: **Q(s,a) = 0 + 0.5(0) + 0.5^2(0) +0.5^3(100) ☾ 12.5**
- Start 3 and go left : **Q(s,a) = 0 + o.5(0) + 0.5^2(100) ☾ 25**
- What would it be for Start 4 and go right?

Computing Q(s,a) and why it is important in RL - Example

100  100	50 12.5 0	25 6.25 0	12.5 10 0	6.25 20 0	40  40

- Now we have Q(s,a) for every state and every action.
- This function is also called, for obvious reasons, the Q function.
- It tells you what the returns are for an action taken in a state and you behave optimally after that.
- Notice that the max values, that we learned through trial and error, returned in every state matches with what we already determined as the best policy.
- Therefore, the best action for the agent to take in any state, is to take the action which gives the largest return.
- The best possible action to take in a given state is therefore **max [*Q(s,a)*].**

Bellman Equation – Our tool for calculation Q(s,a) values

- We come to realize that, if we can compute the Q(s,a) values for every state, then, we can take the best action in every state by picking the action which gives us the maximum return.
- The question is – how do we compute these values for different states and the different actions we can take in every state?
- The key to doing this are the Bellman equations and they are foundational in RL.
- The equation is a result of the theory of dynamic programming which was pioneered in the 1950s by Richard Bellman and coworkers.
- It computes the "value" of a decision problem at a certain point in time in terms of the payoff from some initial choices and the "value" of the remaining decision problem that results from those initial choices.
- The Bellman equation was first applied to engineering control theory and to other topics in applied mathematics and subsequently became an important tool in economic theory.

Bellman Equations – Terminology

- **Q(s,a)** is called the state action-value function and is defined as the return you get if you :
  - Start in a state ‘s’
  - Take an action ‘a’ (once),
  - Then behave optimally after that

- Terminologies used and sequence:

  s : current state

  a : current action

  s’ : state you get to after taking the action

  a’ : action you take in the state s’

Bellman Equation

- $Q(s,a) = R(s) + \gamma \max_{a'} Q(s'.a')$: 1st term is immediate reward, 2nd term is Return from behaving optimally
- Let us apply it on our running example and see if it checks out:

100  100	50 12.5 0	25 6.25 0	12.5 10 0	6.25 20 0	40  40

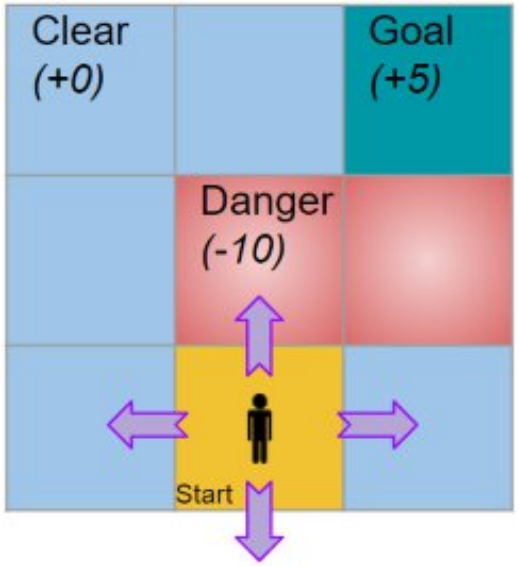
  - Q(2, ☾) = R(2) + 0.5 $\max_{a'}$ (3,a') ☾ 0 + 0.5(25) = 12.5
  - So, we can also say the max possible return from state s' is $\max_{a'}$(s',a') if we behave optimally.
  - We can write the equation also as: $R(s) + \gamma R2 + \gamma 2$ R3 + $\gamma 3$ R4 + ... OR R(s) + γ [R2 + γ R3 + $\gamma 2$ R4 + ...]
  - So, this term in the brackets is nothing but the max return you would compute if you started in s' and acted optimally.
  - Just applying the Bellman's equation will get an RL algorithm to work quite well.

Q – Learning and Q - Table

- The Q-learning algorithm uses a Q-table of State-Action Values (also called Q-values).
- Q-table has a row for each state and a column for each action.
- Each cell contains the estimated Q-value for the corresponding state-action pair.
- We start by initializing all the Q-values to zero.
- As the agent interacts with the environment and gets feedback, the algorithm iteratively improves these Q-values until they converge to the Optimal Q-values.
- It updates them using the Bellman equation.

Q – Learning and Q – Table

- Consider a 3x3 grid, where the player starts in the Start square and wants to reach the Goal square as their final destination.
- Reward when you reach the goal is 5 points.
- Some squares are Clear while some contain Danger, with rewards of 0 points and -10 points respectively.
- In any square, the player can take four possible actions to move Left, Right, Up, or Down.
- This problem has 9 states since the player can be positioned in any of the 9 squares of the grid.
- We start by initializing all the Q-values to 0.
- Q-learning finds the Optimal policy by learning the optimal Q-values for each state-action pair.



	Left	Right	Up	Down
(1,1)	0	0	0	0
(1,2)	0	0	0	0
(1,3)	0	0	0	0
(2,1)	0	0	0	0
(2,2)	0	0	0	0
(2,3)	0	0	0	0
(3,1)	0	0	0	0
(3,2)	0	0	0	0
(3,3)	0	0	0	0

Update rule and equation for Q values

- We said that we will update the Q values of a state using the Bellman equation.
- If we use the Bellman equation as it is, then we are not learning much from our previous experiences as the Q values are being refreshed to a new value and depending on the state these could be much higher or much lower than the previous update.
- We want to be learning incrementally and also learn from our experiences. We can modify our Bellman equation as follows:

$$Q(s,a) = Q(s,a) + \alpha\,(R(s) + \gamma \max_{a'} Q(s', a') - Q(s,a))$$

- We introduced a parameter called alpha, which is called the learning rate, and it typically has values around 0.1.
- We have also maintained the current Q(s,a) value of the state and the update step only incrementally changes the Q(s,a) update.
- You will also notice we subtract a part of the current Q(s,a) value from the return value.
- This modified equation ensures that we are headed in the right direction with Q value updates and incrementally and hence are sure of convergence.

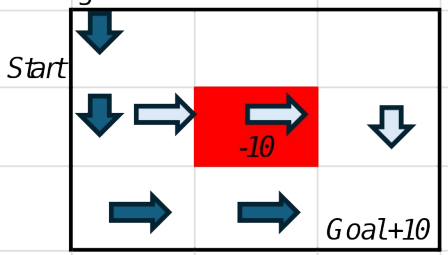
Updated Bellman equation – Why?

- We said that we will update the Q values of a state using the Bellman equation.
- If we use the Bellman equation as it is, then we are not learning much from our previous experiences as the Q values are being refreshed to a new value and depending on the state these could be much higher or much lower than the previous update.
- We want to be learning incrementally and also learn from our experiences. We can modify our Bellman equation as follows:

$$Q(s,a) = Q(s,a) + \alpha\,(R(s) + \gamma \max_{a'} Q(s', a') - Q(s,a))$$

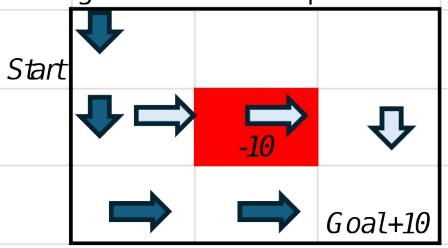
- We introduced a parameter called alpha, which is called the learning rate, and it typically has values around 0.1.
- We have also maintained the current Q(s,a) value of the state and the update step only incrementally changes the Q(s,a) update.
- You will also notice we subtract a part of the current Q(s,a) value from the return value.
- This modified equation ensures that we are headed in the right direction with Q value updates and incrementally and hence are sure of convergence.

Actions down, up, right, left
Rewards in all other states =0
gamma =0.9



EP1: Down - Down - Right-Right Q(1,1) =0 +0.9(0)+0.9^2(0)+0.9^3(0) +0.9^4(10) =6.51
Ep2: Down - Right - Right-Down Q(1,1) =0+0.9(0)+0.9^2(-10)+0.9^3(0)+0.9^4(10) =-1.54

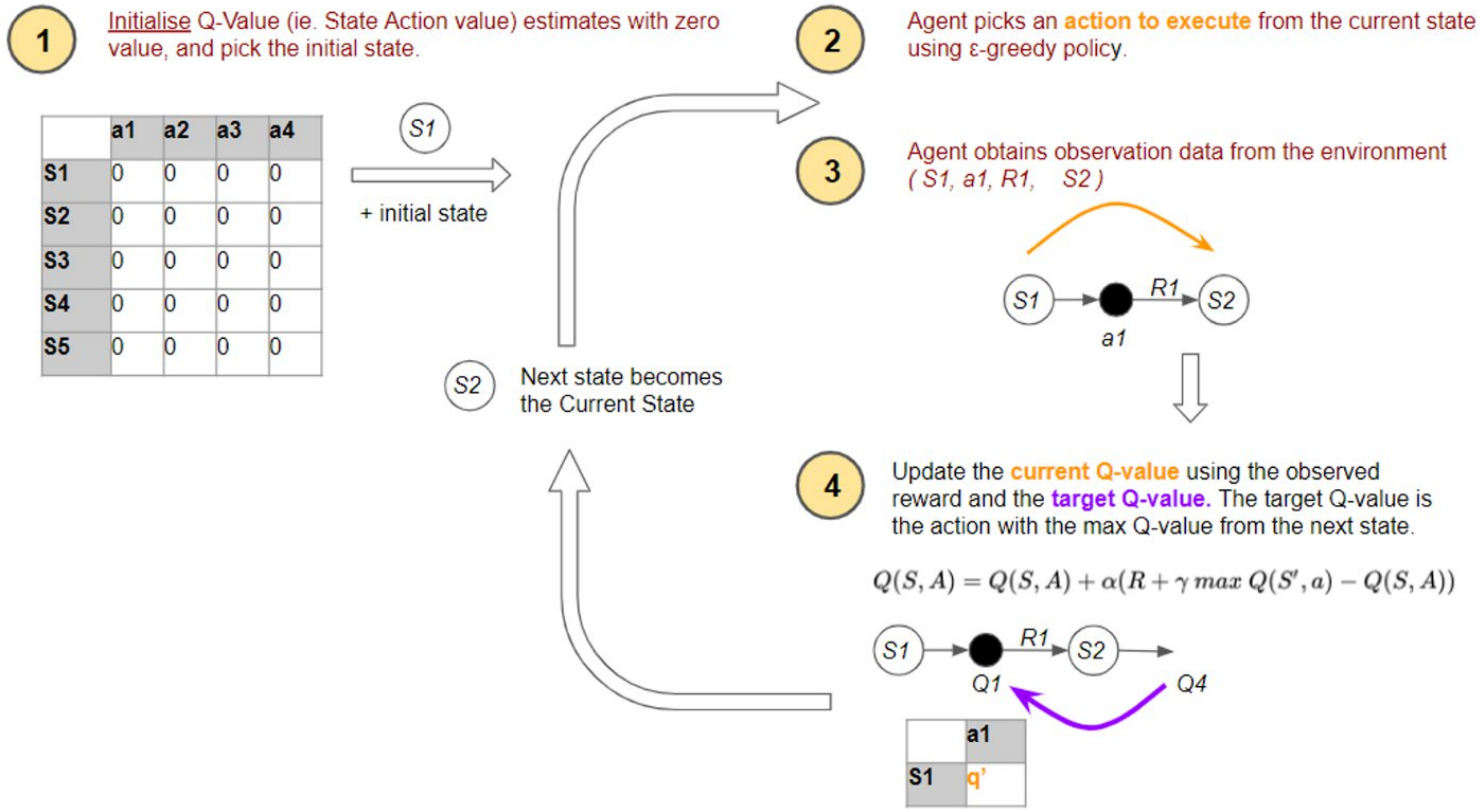
gamma =0.9 and alpha =0.1



EP1: Down - Down - Right-Right Q(1,1) =0 +0.1(6.5-0) =0.65
Ep2: Down - Right - Right-Down Q(1,1) =0.65+0.1(-1.54-0.65) =0.431

Overall flow of the Q-Learning algorithm

- Initially, the agent randomly picks actions.
- But as the agent interacts with the environment, it learns which actions are better, based on rewards that it obtains..
- It uses this experience to incrementally update the Q values.



Basic steps in training an agent using Q-learning

1. Agent starts in a state (s1) takes an action (a1) and receives a reward (r1)
2. Agent selects action by referencing Q-table with highest value (max) **OR** by random (epsilon, ε)
3. Update q-values $Q(s,a) = Q(s,a) + \alpha\,(R(s) + \gamma \max_{a'} Q(s', a') - Q(s,a))$
4. The reward system needs to be defined for each environment.
5. Terminal state / s need to be defined – when an episode ends after it has started
6. Run the training process for n number of episodes till the agent has learned the optimal policy and does what you want it to do every time.
7. Point to note is that the Q values are carried over to the next episode. This is required otherwise the policy will never converge to an optimum one.
8. Once you have the optimal policy, it can be deployed for normal use.

Exercise 5

Build the optimal policy for the Mineral explorer with Python

1. Follow the rule of taking a random first step and then following the optimal policy.
2. An episode is defined a – from start till the terminal state is reached.
3. Use the Bellman's equation
4. Implement the Q-table using the Bellman's equation and find the final optimal Q values.
5. Let gamma= 0.9 and alpha= 0.1.
6. Run for n episodes till the Q-values do not change anymore (optimal).
7. Print the final Q-values

How to verify your code

100  100	50 12.5 0	25 6.25 0	12.5 10 0	6.25 20 0	40  40

- We arrived at these Q-values by taking gamma = 0.5 and we did those computations manually.
- We did not use the updated Bellman equation, in other words, we assumed alpha = 1.
- So, if we coded the solution for Exercise 5 properly, and plugged in these value for alpha and gamma, then we should be able to verify this outcome and hence your code.

gymnasium: Python library
Simulated environments for RL

Gymnasium installing

- **Before install gymnasium you Must install: (for Windows users only)**
- conda install swig
- Microsoft C++ build tools - https://visualstudio.microsoft.com/downloads/ https://www.youtube.com/watch?v=gMgj4pSHLww

After that: (including mac and Linux users)

- conda install conda-forge::gymnasium-all

OR individually

- conda install -c conda-forge gymnasium
- conda install -c conda-forge gymnasium-box2d
- conda install -c conda-forge gymnasium-classic_control

OR

- pip install gymnasium
- pip install[box2d] and so on ..

gymnasium basics

1. The fundamental building block is the python class “Env”. It implements a simulation of the environment you want to train your agent in. There are many environments available.

2. The basic structure of the environment is described by the “observation_space” and the “action_space” attributes of the Gym Env class.

3. The observation can be different things - The most common form is a screenshot of the game. Other forms of observations are certain characteristics of the environment described in vector form.

4. The action_space, which describes the numerical structure of the legitimate actions that can be applied to the environment.

Environments in basic gymnasium

Gymnasium includes the following families of environments along with a wide variety of third-party environments:

Classic Control - These are classic reinforcement learning based on real-world problems and physics.

Box2D - These environments all involve toy games based around physics control, using box2d based physics and PyGame-based rendering

Toy Text - These environments are designed to be extremely simple, with small discrete state and action spaces, and hence easy to learn.

MuJoCo - A physics engine based environments with multi-joint control which are more complex than the Box2D environments.

Atari - A set of 57 Atari 2600 environments simulated through Stella and the Arcade Learning Environment that have a high range of complexity for agents to learn.

Third-party - A number of environments have been created that are compatible with the Gymnasium API.

Be aware of the version that the software was created for and use the “apply_env_compatibility” in “gymnasium.make“if necessary..

Interacting with the environment

- **reset:** This function resets the environment to its initial state and returns the observation of the environment corresponding to the initial state.
- **step :** This function takes an action as an input and applies it to the environment, which leads to the environment transitioning to a new state. The step function returns four things:
- **observation:** The observation of the state of the environment.
- **reward:** The reward that you can get from the environment after executing the action that was given as the input to the step function.
- **done:** Whether the episode has been terminated. If true, you may need to end the simulation or reset the environment to restart the episode.
- **info:** This provides additional information depending on the environment, such as number of lives left, or general information that may be conducive in debugging.

gymnasium basics - Example

End of Lesson