# INFO 4000
# Informatics III

## Data Science Specialization - Advanced

## Week6 Reinforcement Learning contd...

Course Instructor

**Jagannath Rao**

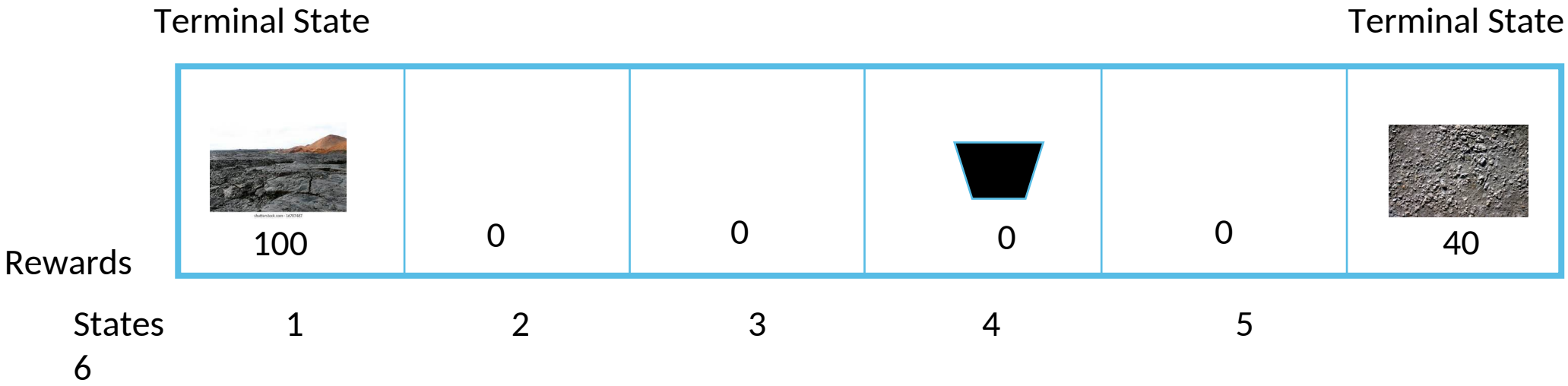
raoj@uga.edu

# Recap:
# Reinforcement learning:
## Discrete spaces and actions

# What is Reinforcement Learning (RL)

- An approach where,
  - an agent learns to make sequences of decisions through trial and error by interacting with an environment and ,
  - receiving feedback in the form of rewards or penalties to maximize its cumulative reward over time.
  - unlike supervised learning, there's no explicit supervisor
  - the agent independently discovers the optimal strategy without being given the correct actions, instead learning through experience how to achieve a goal
- Think of it like a child learning to walk –
  - they take steps (actions), sometimes fall (negative reward), and sometimes stay upright and move forward (positive reward).
  - through repeated attempts, they learn the sequence of muscular actions (policy) to achieve the goal of walking steadily

# Return and Reward 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.

# Q value function and the updated Bellman Equation

- **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

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

Terminologies used and sequence:

s : current state

a : current action

s’ : state you get to after taking the action

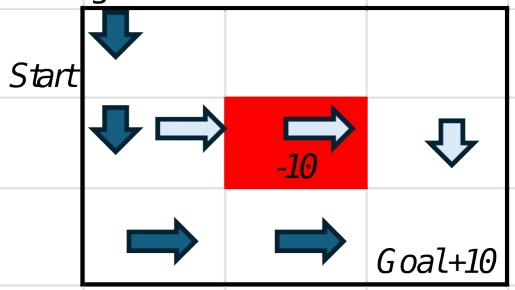
a’ : action you take in the state s’

# Updated Bellman equation – Why?

Actions down, up, right, left

Rewards in all other states =0

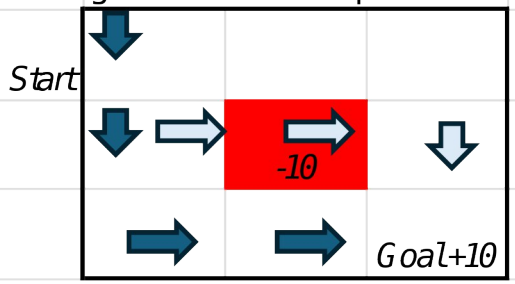
gamma =0.9




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

# 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.

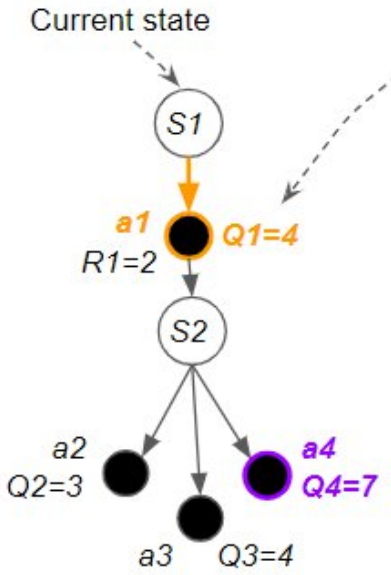
| | Left | Right | Up | Down |
|---|---|---|---|---|
| (1,1) | 0 | 0 | 0 | 0 |
| (1,2) | 0 | 0 | 0 | 0 |
| (1,3) | 0 | 0 | 0 | 0 |
| (2,1) | 0 | 0 | 0 | 0 |
| (2,2) | 0 | 0 | 0 | 0 |
| (2,3) | 0 | 0 | 0 | 0 |
| (3,1) | 0 | 0 | 0 | 0 |
| (3,2) | 0 | 0 | 0 | 0 |
| (3,3) | 0 | 0 | 0 | 0 |

# 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) and moves to state s2
2. Agent selects action by referencing Q-table with highest value (max) **OR** by random (epsilon, ε)
3. Update q-values $Q(s,a) = Q(s,a) + a\,(R(s) + \gamma \max_{a'} Q(s', a') - Q(s,a))$ where the term $\max_{a'} Q(s', a')$ **is the Q−Value associated with the action with the max Q-Value in the next state.**
4. Apply this rule in every state till you reach the terminal state.
5. The reward system needs to be defined for each environment. Terminal state / s need to be defined – when does an episode end 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.

# 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.




4b The current Q-value is updated

$$Q(S,A) = Q(S,A) + \alpha(R + \gamma\, max\, Q(S',a) - Q(S,A))$$

Q1 = Q1 + α (R1 + γ * Q4 - Q1)

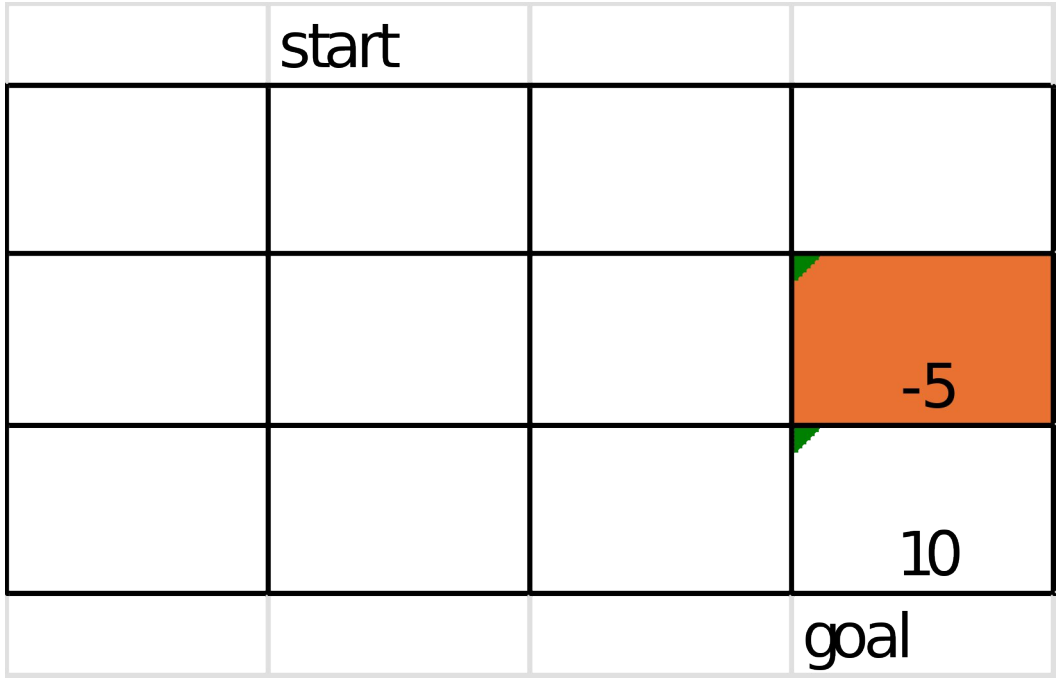
Q1 = 4 + (2 + 7 - 4) = 9

NB: taking α = γ = 1 for simplicity

# Let us do an exercise to understand the concept

Actions, Terminal state and Rewards:

- Up, down, left, right
- If you hit the side walls you remain there
- Episode end when goal is reached or 10 moves are made
- Rewards: 0 in all states except goal and dangerous state
- Gamma = 0.5, alpha = 0.5
- We shall do 3 episodes.



# ε – Greedy policy

# Trade off between Explore and Exploit strategy:

**Exploration**: It's about gathering more information about the environment. By exploring, the agent can learn about rewards associated with actions it has not tried before. Random actions.

**Exploitation**: Once the agent has gathered some knowledge about the environment, it can use that knowledge to take actions that it believes will maximize its reward. In other words, the agent uses the knowledge it has to get the best possible outcome.

Imagine playing a new video game and finding a strategy that works reasonably well early on. If you never try anything different, you might miss out on a much better strategy. Exploration ensures that the agent doesn't get trapped in local optima and has a chance to find the global optimum.

**Epsilon-greedy**: This is one of the most commonly used strategies. Here, with probability less than $\epsilon$ (where $\epsilon$ is a small positive value), the agent takes a random action (exploration), and with probability $1-\epsilon$, the agent takes the action that it believes to be the best based on its current knowledge (exploitation).

Achieving the right balance is essential. Too much exploration can result in the agent never really benefiting from what it learns (always trying new things and never capitalizing on known good actions). Conversely, too much exploitation can prevent the agent from discovering potentially better strategies.

# **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

# 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

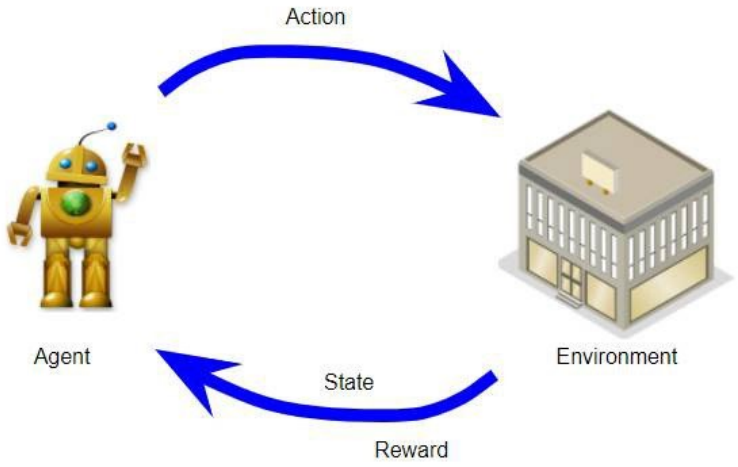
- **reset:** This function resets the environment to its initial state and returns the observation of the environment corresponding to the initial state.
- **step :** This function takes an action as an input and applies it to the environment, which leads to the environment transitioning to a new state. 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

# RL for continuous spaces

# Here is what we needed to do in discrete spaces

- We have 2 actions and 6 states (including 2 terminal states).
- Q-table has to be a 6 x 2 size table all initialized with zero except the terminal states.
- It updates them using the Bellman equation.
- Terminal states are assigned their fixed reward values and episodes end at the terminal states.




| | | | | | |
|---|---|---|---|---|---|
| 100<br>  <br>100 | 50 12.5<br>0 | 25 6.25<br>0 | 12.5 10<br>0 | 6.25 20<br>0 | 40<br>  <br>40 |

# Continuous spaces

- A q-table only works with a discrete state space, or when the agent can only be in a set number of states.
- Think a grid. A maze. A card game. Not the movements of a robot. Not for a racecar going around a track.
- Blackjack-v0: states are made up of the combinations of the players current card sum, the faceup card of the dealer, and if the player has a useable ace. This is discrete states.
- MountainCar: state is car position and car velocity. This means continuous states.
- A q-table assumes that each state is independent of each other. This clearly isn't true for continuous environments.
- Many states are very similar to each other, and we'd like our agent to behave similarly in those states.

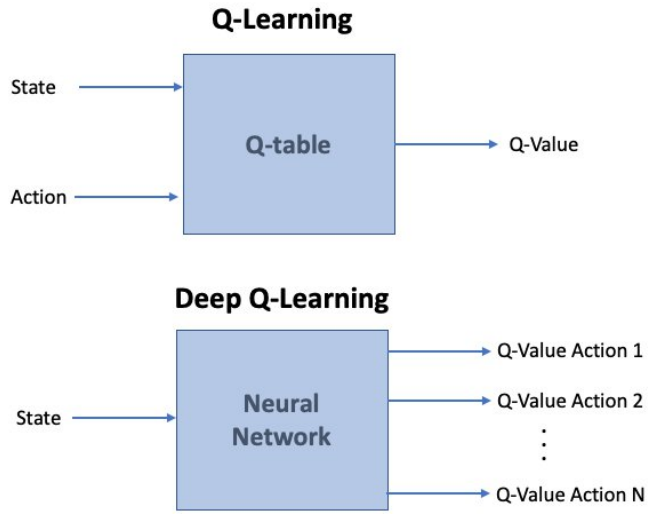
# Developing a function approximator

- What if we represented the q-values of each state with a model? Something where we could give as input our state and get as output the q-value for that state.
- The q-value is the thing we're trying to model (the y-variable, response variable, dependent variable), given the state (x-variables, independent variables, explanatory variables).
- So, we are in need to develop a function, $y = f(x)$ , this is best done with a Neural Network as a function approximator.
- If this were a straightforward supervised learning task, we'd use gradient descent and backpropagation to slowly adjust the values of **w** until we reach an optimum.
- While the true value of *q(s,a)* is a fixed value, we don't know what that true value is - we just have an estimate for the value.
- Our estimates (obtained using q-learning) will improve over time as the agent collects more information, but they will jump around in the process.
- So, the assumption that we have a fixed target to reach, which we know of, is invalid.
- Imagine you're playing golf, and every time you hit the ball towards the hole, the hole changed location – three hundred meters to the right, one hundred to the left, twelve meters behind you.

# Continuous Spaces Reinforcement Learning(RL) – Deep Q-Learning or DQN

# DQN - basics

- Since we use deep neural networks to create the function approximator we alluded to, it gets the name DQN or Deep Q Network.
- In a traditional reinforcement learning setup only one Q-value is produced at a time
- Deep Q-network is designed to produce in a single forward pass a Q-value for every possible action available in the Environment:
- This approach of having all Q-values calculated with one pass through the network avoids having to run the network individually for every action and helps to increase speed significantly..
- We can simply use this vector to take an action by choosing the one with the maximum value.
- This neural network can be a simple NN, a CNN or any other structure depending on what the input is.
- Training such a network requires a lot of data to even do a mini-batch gradient descent and we need to figure out how to provide that data.



# Bellman Equation for Continuous spaces

- $Q(s,a) = Q(s,a) + \ a\ (\ R(s) + \ \gamma \ \frac{max}{a'}\ Q(s', a')\ - Q(s,a))$ the normal Bellman equation.

- Bellman Equations for continuous spaces reduces to:
  - $Q(s,a) = R(s) + \frac{max}{a''}(Q(s',a'))$ .
  - This is because NN's have the learning rate factor built in and the changes to Q values will be incremental.

# Q-Network and Target Network

- Neural Network is a supervised learning algorithm and to train them and build a model, they need features and a fixed target value to compare with.
- The feature we provide the NN (Q-Network) is the current state and the output we should get is the predicted Q-Value of that state.
- If we knew the final Q-Value then we could train the network by comparing the predicted value with that. But we do not know that final value.
- So, the trick is to use a target network which provides that target even if they are intermediate targets. We make a 2$^{nd}$ NN which is a copy of the Q-Network to start with. This is called the Target Network.
- This network is not trained (no back prop). This is important to remember. Therefore, the Q values this network generates remains the same and hence is a good target value.
- After every so many episodes, the weights of the original network, which are being updated through gradient descent, is copied on to the Target Network. This enables a smooth progress towards optimal Q values.

# Technique for generating and feeding data – Experience Replay

- We are trying to approximate a complex, nonlinear function, Q(s, a), with a Neural Network.
- We need data containing features and we collect this in a “The replay buffer”, that contains a collection of tuples (*S*, *A*, *R*, *S´*).
- The buffer can be of a fixed size and is populated based on interactions with the environment.
- We can then sample training data from this, randomly, in mini-batches. This is also referred to as “ Experience Replay”.
- In order to store the Agent’s experiences, we used a data structure called a deque in Python’s built-in collections library. It’s basically a list that you can set a maximum size on so that if you try to append to the list and it is already full, it will remove the first item in the list and add the new item to the end of the list.
- The experiences themselves are tuples of *[observation*, *action*, *reward*, *done flag*, *next state]*.

# Example: How to create Experience Replay buffer

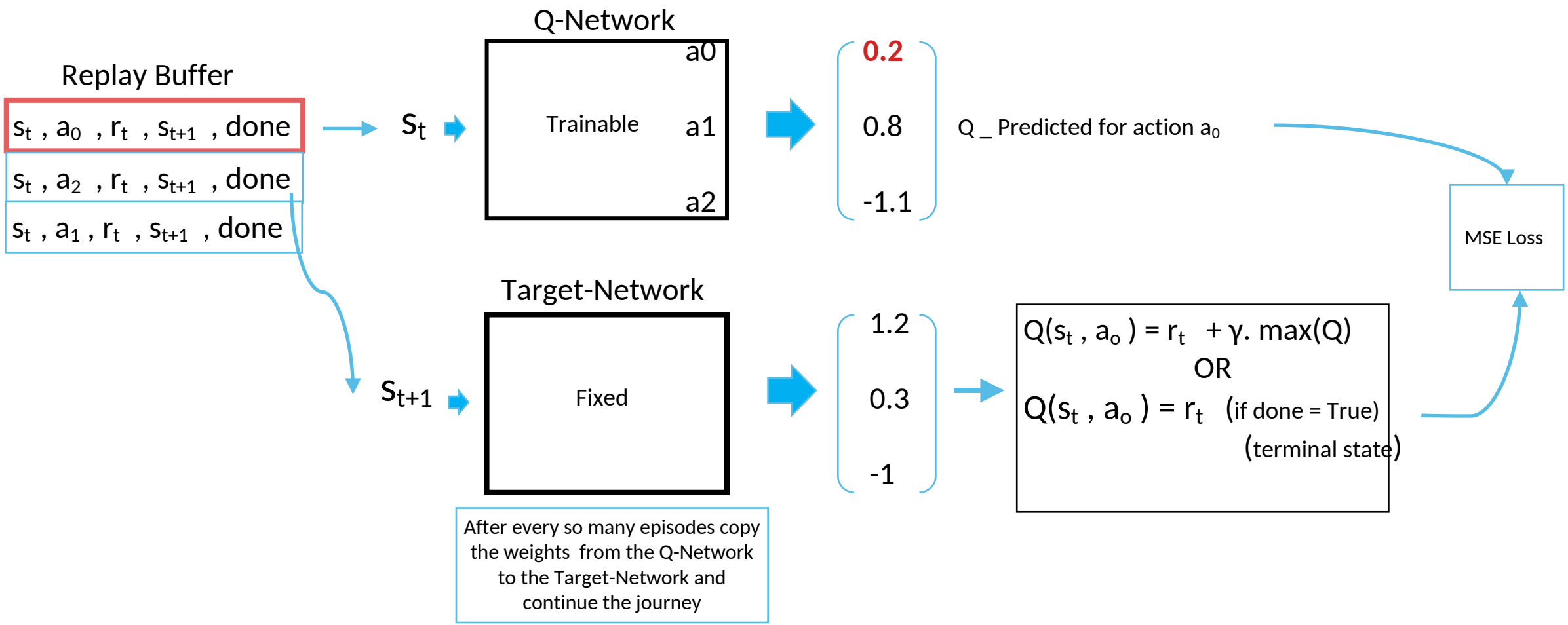
# DQN algorithm

Initialize network $Q$
Initialize target network $\hat{Q}$
Initialize experience replay memory $D$
Initialize the *Agent* to interact with the Environment
**while** *not converged* **do**

  `/* Sample phase`
  $\epsilon \leftarrow$ setting new epsilon with $\epsilon$-decay
  Choose an action $a$ from state $s$ using policy $\epsilon$-greedy($Q$)
  *Agent* takes action $a$, observe reward $r$, and next state $s'$
  Store transition $(s, a, r, s', done)$ in the experience replay memory $D$

  **if** *enough experiences in* $D$ **then**
    `/* Learn phase`
    Sample a random *minibatch* of $N$ transitions from $D$
    **for** *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ ***in*** *minibatch* **do**
      **if** $done_i$ **then**
        $y_i = r_i$
      **else**
        $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
      **end**
    **end**
    Calculate the loss $\mathcal{L}$ =1/N $\sum_{i=0}^{N-1}(Q(s_i, a_i) - y_i)^2$
    Update $Q$ using the SGD algorithm by minimizing the loss $\mathcal{L}$
    Every $C$ steps, copy weights from $Q$ to $\hat{Q}$
  **end**
**end**

# DQN training and flow



# Numerical example to illustrate previous slide details

Assumptions: actions = 4 and $\gamma = 0.95$

Replay buffer row content = [current state = 100, action = 0, reward = 5, next state = 102]

Q-Network output = [2.3, -1.5, 0.8, 0.1] Predicted Q-values for actions 0, 1, 2, 3 respectively

Target Network output = [3.1, 0.5, 2.2, -0.3] Predicted Q-values for actions 0, 1, 2, 3 in the next state

**Bellman Equation and Loss Calculation:**

max Q from Target network for next state = 3.1

Target Q value for the current state-action pair is : $Q(s,a) = R(s) + \gamma * \max Q(s',a')$ which is $5 + 0.95 * 3.1 = 7.945$

Predicted Q value from the Q-network is 2.3 for action 0

MSE = $1/2 * (7.945 - 2.3)^2$

# Examples:
## Discrete and Continuous spaces with discrete actions

# HW 2

# Part 1: MQTT

1. Write separate python scripts, for publishing data and to receive that data using MQTT.

2. You will write 2 separate publish scripts

   - one for temperature and humidity with numbers for temperature ranging between 50 to 52 F and numbers for humidity between 60 and 80 percentage.

   - the other publish file will be for rainfall with numbers ranging between 0 and 2 inches.

3. You will also write 2 separate subscriber scripts with

   - One subscriber subscribing to all the weather data

   - This script will start getting the data with client.loop_start() and when 50 data points are reached, it will stop subscribing using client.loop_stop()

   - The other subscriber will receive the temperate and humidity data and will loop_forever.

# Part 2: Build the DQN Model for the Lunar Lander

1. From the Farama Gymnasium library choose the version **Lunar Lander – V3.** This is the version with continuous spaces and discrete actions.

2. To complete the exercise, you will do the following:

   - Build main loop where you will take actions,
   - update Q-Values,
   - populate the replay buffer,
   - Call the Q-Network update function

3. You will write a separate function for the NN training and build a simple architecture for that.

4. Once the model achieves its goals (based on details given in the gymnasium documentation) save the model.

5. Use the saved model (the Policy) to run 5 complete episodes with flawless landings.

# Due data and grading

1. HW 2 will be due by October 5th midnight.
2. Grading criteria
   - All code is implemented as per instructions and runs error free
   - Questions given below are answered for each part
   - Code is well commented
3. Part 1 Questions:
   - What was you biggest takeaway from the MQTT exercise
   - What is different between looping forever and looping with start – stop? What would happen if you had a whole application and the MQTT code with loop_forever had to receive data , store it and in the later part of the same script, analyze that data?
4. Part 2 Questions:
   - Explain how DQN works in the application that you built? Answer should be specific to what you developed.

# LunarLander v3 details

## Description

This environment is a classic rocket trajectory optimization problem. According to Pontryagin's maximum principle, it is optimal to fire the engine at full throttle or turn it off. This is the reason why this environment has discrete actions: engine on or off.

There are two environment versions: discrete or continuous. The landing pad is always at coordinates (0,0). The coordinates are the first two numbers in the state vector. Landing outside of the landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt.

To see a heuristic landing, run:

```
python gymnasium/envs/box2d/lunar_lander.py
```

## Action Space

There are four discrete actions available:

- 0: do nothing
- 1: fire left orientation engine
- 2: fire main engine
- 3: fire right orientation engine

## Observation Space

The state is an 8-dimensional vector: the coordinates of the lander in `x` & `y`, its linear velocities in `x` & `y`, its angle, its angular velocity, and two booleans that represent whether each leg is in contact with the ground or not.

## Rewards

After every step a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode.

For each step, the reward:

- is increased/decreased the closer/further the lander is to the landing pad.
- is increased/decreased the slower/faster the lander is moving.
- is decreased the more the lander is tilted (angle not horizontal).
- is increased by 10 points for each leg that is in contact with the ground.
- is decreased by 0.03 points each frame a side engine is firing.
- is decreased by 0.3 points each frame the main engine is firing.

The episode receive an additional reward of -100 or +100 points for crashing or landing safely respectively.

An episode is considered a solution if it scores at least 200 points.

# Singular Value Decomposition(SVD):
## Get a deeper understanding of the data

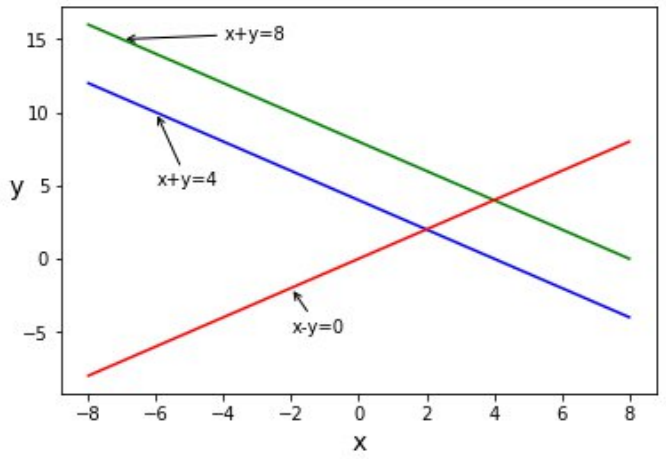
# 3 possible outcomes to system of Linear equations

## Matrix Representation

$$\begin{pmatrix} 2 & 1 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3 \\ -1 \end{pmatrix}$$

A x = b

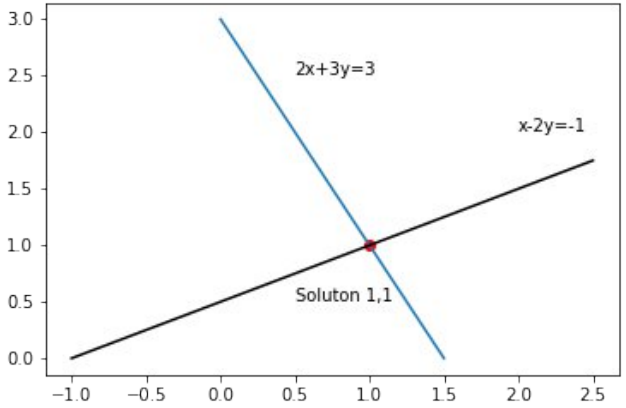
## No Solution



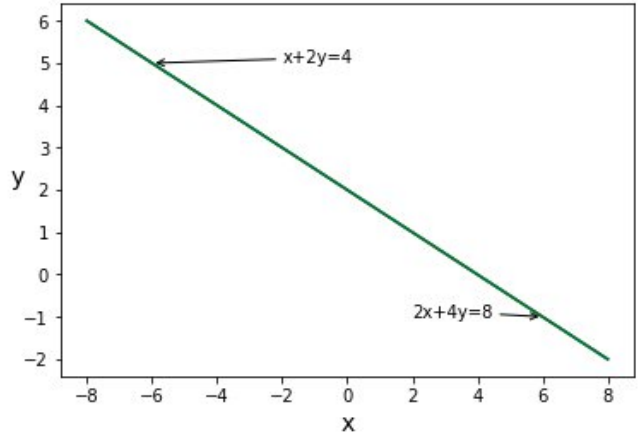

Machine Learning problems where we have 100's of data points can also be a system of "linear equations" and usually they do not have a solution. Therefore, we try to find the solution with the lowest error to build a model which represents the system

**The datasets can be represented in Matrix form as Ax=b.**

## Unique Solution




## Infinite Solutions



# What is matrix decomposition?:

## Here is an example to solve a system of linear equations

Solve the following system of equations:

$$2x_1 + 3x_2 = 5$$

$$4x_1 + 7x_2 = 10$$

We can write this system in matrix form:

$$A = \begin{pmatrix} 2 & 3 \\ 4 & 7 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 5 \\ 10 \end{pmatrix}$$

Thus, the system becomes:

$$A\mathbf{x} = \mathbf{b}$$

### Step 1: Perform LU Decomposition

LU decomposition breaks matrix $A$ into two matrices:

$$A = LU$$

where $L$ is a lower triangular matrix, and $U$ is an upper triangular matrix.

For our matrix $A$, the LU decomposition yields:

$$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & 3 \\ 0 & 1 \end{pmatrix}$$

Thus, $A = LU$.

Since $A = LU$, we can rewrite the system $A\mathbf{x} = \mathbf{b}$ as:

$$LU\mathbf{x} = \mathbf{b}$$

Let $\mathbf{y}$ be an intermediate vector such that:

$$L\mathbf{y} = \mathbf{b}$$

We first solve for $\mathbf{y}$, and then solve for $\mathbf{x}$ using $U\mathbf{x} = \mathbf{y}$.

**Step 2.1: Solve $L\mathbf{y} = \mathbf{b}$**

$$\begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 10 \end{pmatrix}$$

This gives us two equations:

$$y_1 = 5$$

$$2y_1 + y_2 = 10$$

Substitute $y_1 = 5$ into the second equation:

$$2(5) + y_2 = 10 \quad \Rightarrow \quad y_2 = 0$$

Thus, $\mathbf{y} = \begin{pmatrix} 5 \\ 0 \end{pmatrix}$.

**Step 2.2: Solve $U\mathbf{x} = \mathbf{y}$**

$$\begin{pmatrix} 2 & 3 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 0 \end{pmatrix}$$

This gives us two equations:

$$2x_1 + 3x_2 = 5$$

$$x_2 = 0$$

From $x_2 = 0$, substitute into the first equation:

$$2x_1 = 5 \quad \Rightarrow \quad x_1 = \frac{5}{2} = 2.5$$

Thus, $\mathbf{x} = \begin{pmatrix} 2.5 \\ 0 \end{pmatrix}$.

# Types of Matrix Decompositions

**LU Decomposition**: This decomposition breaks a matrix A into two matrices: a lower triangular matrix L and an upper triangular matrix U

**QR Decomposition**: This breaks a matrix A into an orthogonal matrix Q and an upper triangular matrix R, such that: $A = QR$. This is used in solving linear least-squares problems (Regression).

**Cholesky Decomposition**: For a symmetric positive-definite matrix A, this decomposition produces a lower triangular matrix L such that: $A = LL^T$. It is used in numerical simulations and optimization problems where the matrix is symmetric and positive-definite.

**Eigenvalue Decomposition (Spectral Decomposition)**: This expresses a matrix A as a product of its eigenvectors and eigenvalues: $A=V\Lambda V^{-1}$ where V is the matrix of eigenvectors, and $\Lambda$ is a diagonal matrix of eigenvalues. Is used in understanding the properties of matrices, diagonalizing matrices, and is critical in quantum mechanics, principal component analysis (PCA), and machine learning.

**Singular Value Decomposition (SVD)**: SVD factors any m×n matrix A into three matrices: $A=U\Sigma V^T$ ·
SVD is widely used in data compression, noise reduction, and dimensionality reduction techniques such as principal component analysis (PCA).

# What are eigen vectors and eigen values?

- By definition a vector is a projection along a unit vector in the same direction to some magnitude.

$$A\vec{x} = \lambda\vec{x}$$

- A is a matrix of row or column vectors, the vector ‘x’ is called the eigen vector and the ‘λ’ is the eigen value.
- We can compute the eigen values of any matrix A from the characteristic equation :

**Ax = λx ☾ Ax – λx = 0 ☾ (A – Iλ)x = 0**

and therefore

“**det(A – Iλ) = 0**”

# What does SVD do?

- SVD is a linear algebra operation which allows us to decompose any matrix as a combination of matrices.
- This decomposition is represented as : **$A=U\Sigma V^T$**

  A is the original matrix (which can represent different types of data such as images, text, etc.).
  U is the left singular matrix.
  Σ is the diagonal matrix of singular values.
  $V^T$ is the transpose of the right singular matrix.

- **Left Singular Matrix U:** Captures patterns across the rows of the matrix A. Each column of U corresponds to a direction in the row space of A
- For instance, in image data, these vectors might correspond to dominant spatial patterns, while in text data (like a term-document matrix), they might describe common themes or topics present across documents.
- **Right Singular Matrix $V^T$:** Captures patterns and structures related to the **columns** of A. Each column of $V^T$ corresponds to a direction in the column space of A.
- In the case of images, these could represent different frequency patterns in the columns, while in a term-document matrix, they might describe how individual terms relate to the topics represented by U

# Example 1 of SVD operation and what it tells us

- In this example we will use a simple 2 x 2 matrix
- We will see how to decompose it using SVD operator from the NumPy library.
- We will separate the 3 resulting matrices.
- We will then take 2 columns of the U matrix and project row 1 of A onto each of those columns
- We will then see what insight it gives us.

# SVD’s role in Image analytics

- Singular Value Decomposition (SVD) plays an important role in image analytics, where it is used for tasks such as:
  - image compression, by decomposing the image into its key components and discarding less significant information
  - denoising, helps by decomposing an image into components and allowing you to filter out noisy components
  - feature extraction, used to decompose an image into singular vectors and values, with the left singular vectors capturing spatial patterns, and the right singular vectors capturing frequency components. These extracted features can be used in tasks like object recognition or scene classification.
  - dimensionality reduction of an image data by identifying the most important singular values, which capture the dominant structure in an image.
- In conjunction with Deep Learning, SVD can offer complementary benefits in specific cases, often providing insights into the underlying structure of an image or enabling efficient computation where neural networks might be overkill.

# SVD in the context of audio

- A common approach is to transform the audio signal into the frequency domain using a spectrogram.
- This results in a matrix where rows represent time frames, and columns represent different frequencies.
- SVD decomposes this matrix into components that capture the essential information of the signal.
  - The left singular vectors (U) capture temporal patterns,
  - the singular values (Σ) represent the strength of those patterns,
  - and the right singular vectors ($V^T$) describe frequency components.
- In audio analytics, SVD can be used to reduce noise by discarding less important components (those with smaller singular values).
- For example, after performing SVD, you can reconstruct the matrix using only the largest singular values, thus retaining the dominant structures in the audio (such as voice or melody) while filtering out background noise.
- SVD is often used to extract features from audio data for further analysis, such as in speech recognition.

# End of Lesson