# INFO 4000
# Informatics III

## Data Science Specialization - Advanced

## Week7 – RL and Audio

Course Instructor

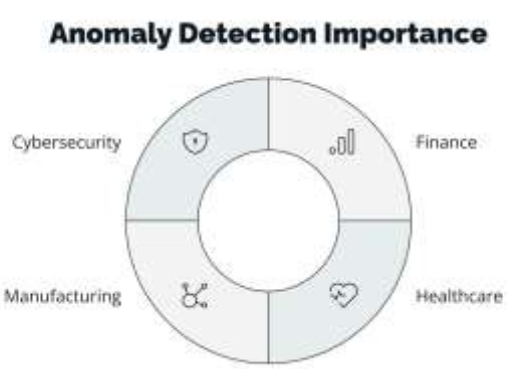
**Jagannath Rao**

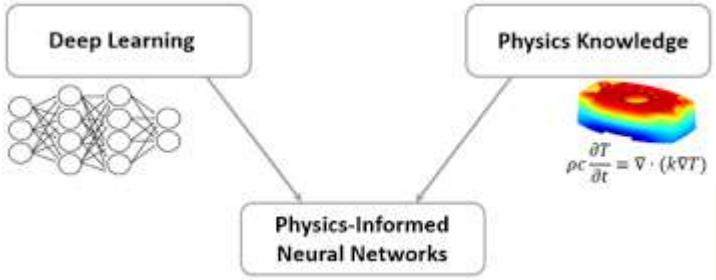
raoj@uga.edu

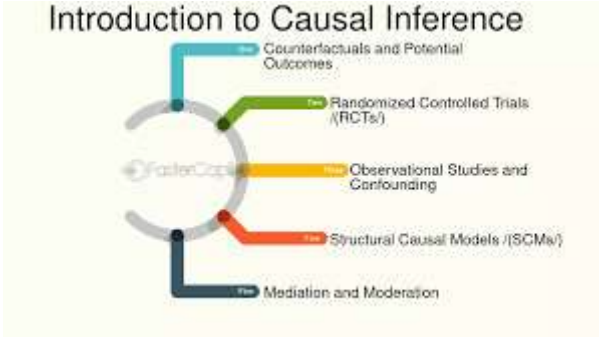
# Where are we in the course

# Course status

- **Topics to be completed:**
  - Audio analytics,
  - special algorithms (SVD and Anomaly Detection),
  - PINN (Physics informed Neural Networks) and
  - if possible Causal Inferencing.
- **Assignments:** Apart from exercises we have MP2 and MP3 to finish.
- **Test:** Will be held on the Thursday before Thanksgiving

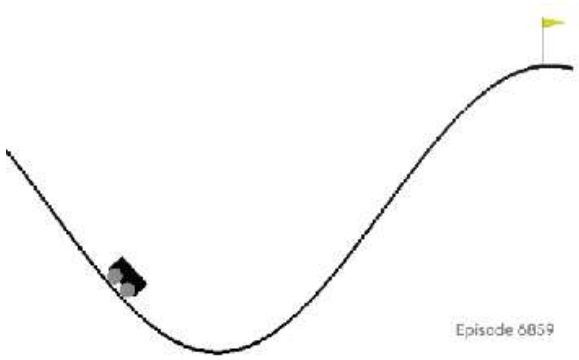


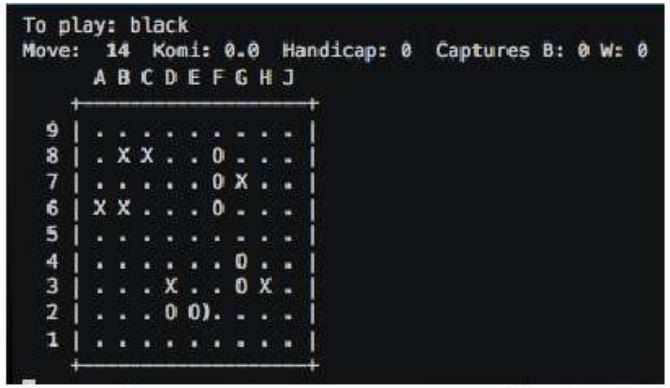



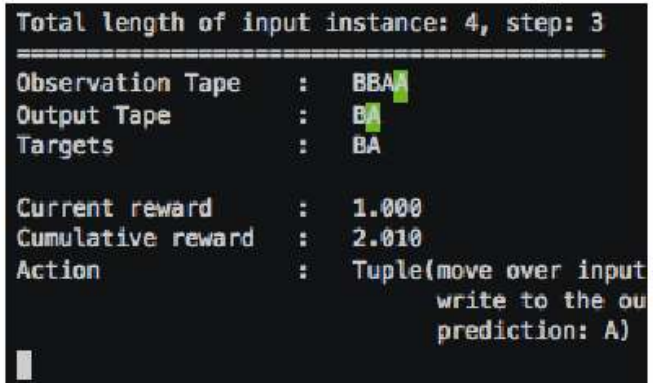
# Recap Reinforcement learning:
## Discrete Actions and Continuous Spaces

# Gymnasium – Python package with simulated environments

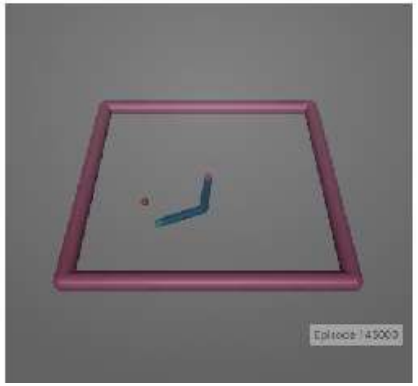




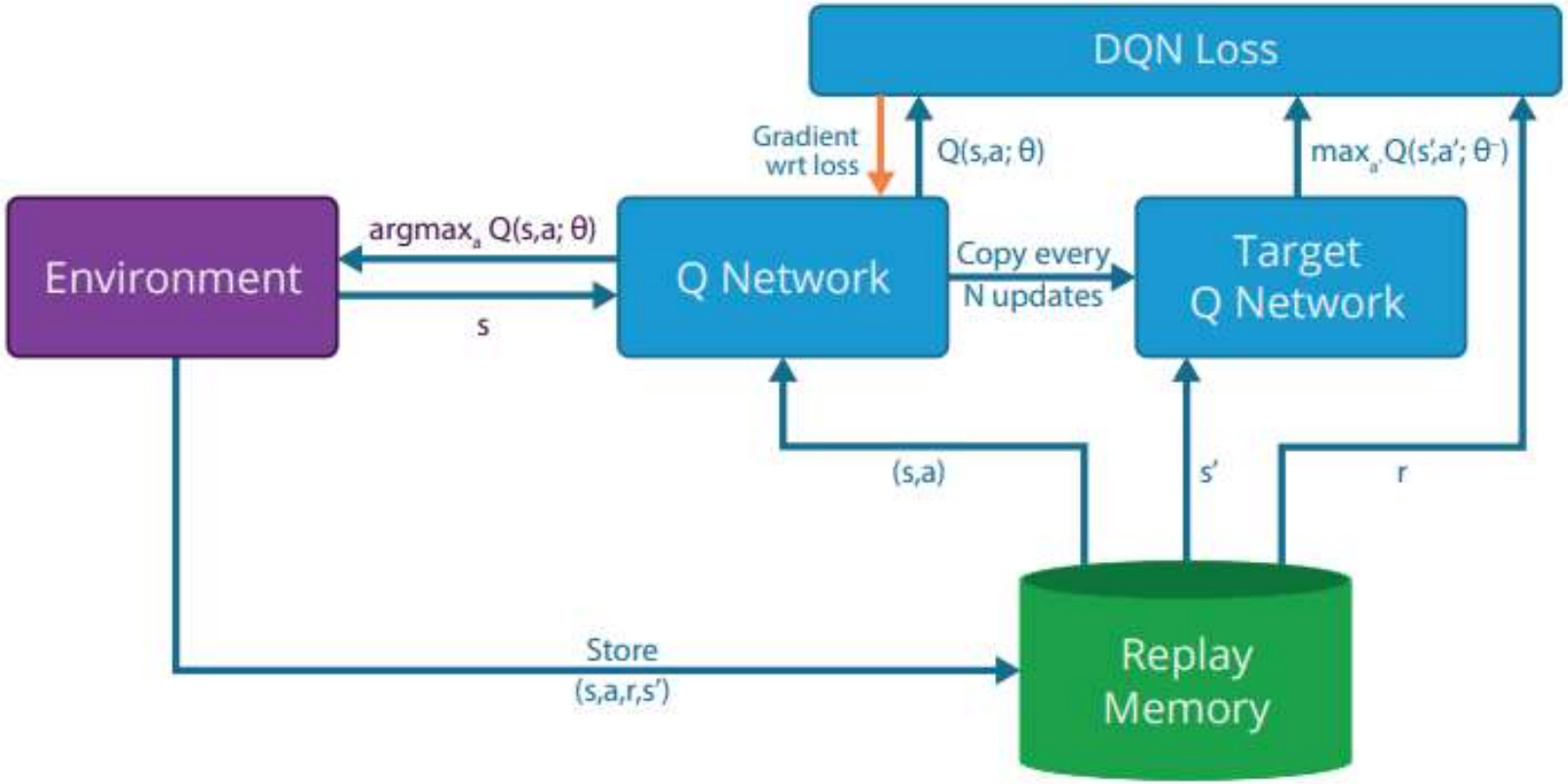




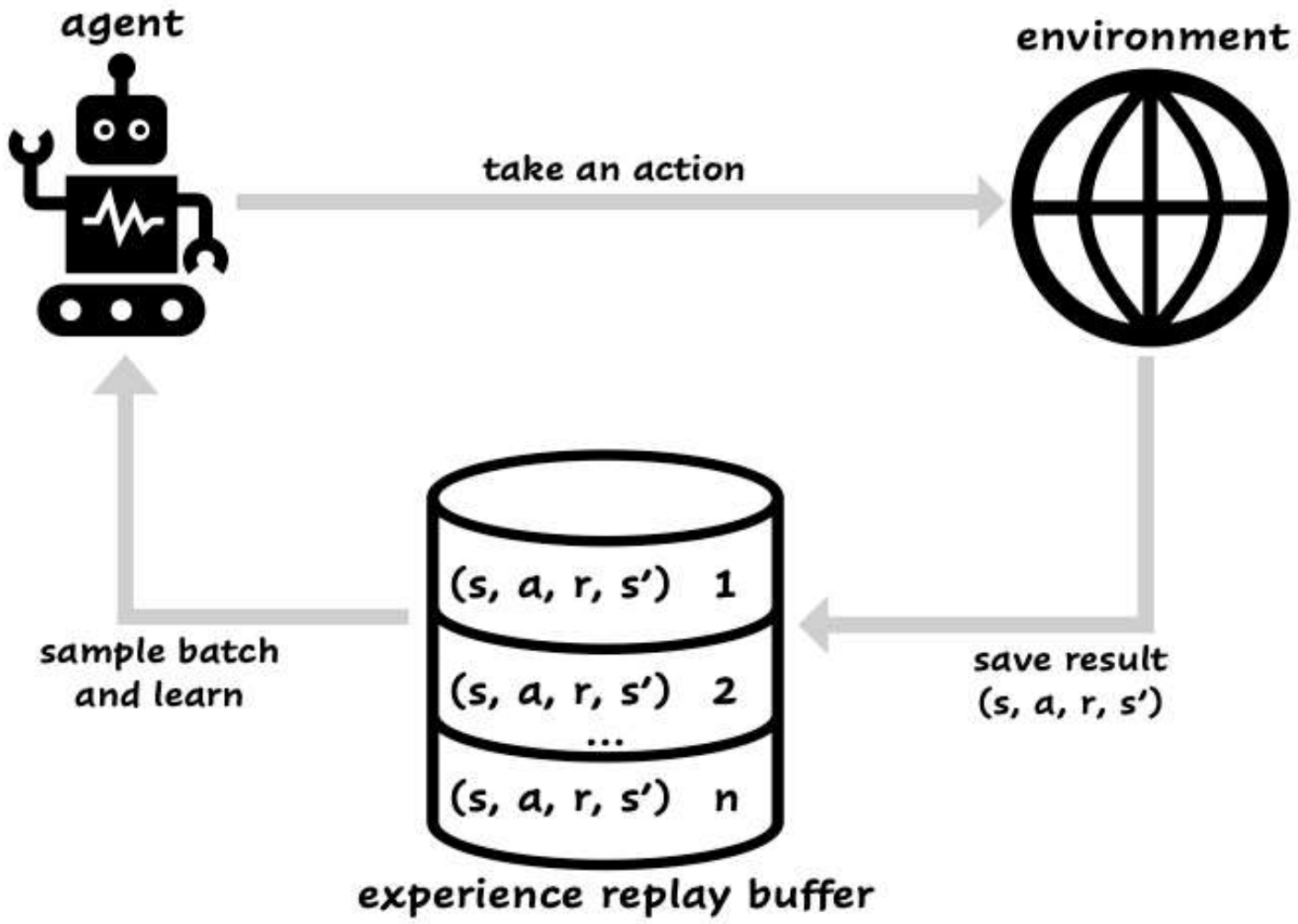
# Bellman Equation for Continuous spaces and discrete actions

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

# DQN training and flow



# Building a dataset for training the NN



# RL in Environments with:
# Continuous actions and continuous spaces

# What is the continuous action space?

**Example 1:**

1. When you're driving of your car and you turn the wheel, it is continuous, because you can control
   - How much you turn the wheel.
   - How much do you press the gas pedal?

2. This leads to a continuous action space:
   - e.g., for each positive real number x in some range, "turn the wheel x degrees to the right" is a possible action.

**Example 2:**

# Challenges in continuous space and action environments

Challenges of a robot learning to pick n place,

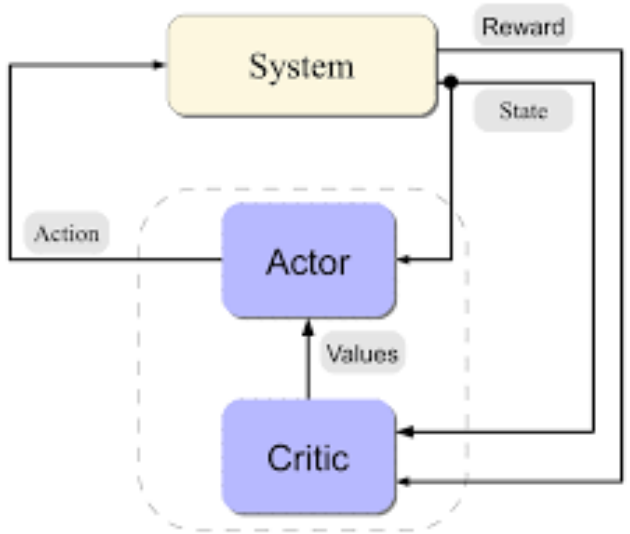
- It has countless joint angles (continuous state space) and
- It can apply varying forces to its motors (continuous action space).
- The number of possible states and actions explodes
- The robot must decide on precise values (e.g., rotate joint 32.5 degrees).
- All this requires more sophisticated decision-making.

# Question is, how do we solve the RL problem under these conditions

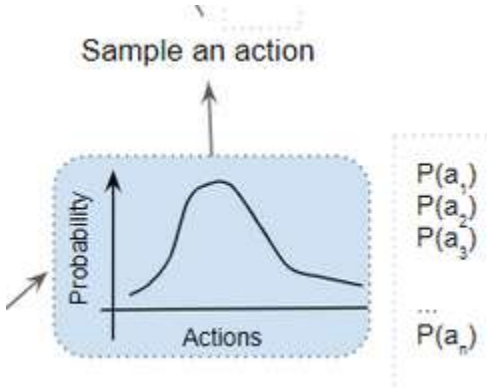
- **Policy Gradients:** These methods directly optimize the policy (the strategy for choosing actions) by adjusting it based on feedback.
  - A robot can slightly tweak its walking style after each step based on how well it's doing.
- **Actor-Critic Methods:** These combine value estimation (critic) with policy improvement (actor) for more efficient learning.
  - Think of it as the robot having a coach (critic) who provides feedback on its performance, helping it refine its walking technique (actor).

# Actor-Critic: How does it work?

They address the challenges by combining two key components:




- **Actor:**
  - This is a neural network that learns the optimal policy, mapping states to actions – unlike DQN which does states to Q-Values
  - Think of it as the "brain" of the robot, deciding what actions to take based on the current situation.
  - Because it is continuous action, the actions are picked up from a normal probability distribution.
  - In other words, the actor outputs parameters for a probability distribution - mean and standard deviation of a Gaussian




- **Critic:**
  - This is another neural network that estimates the value function,
  - Predicting the expected cumulative reward from a given state.
  - It acts as a "coach" or "evaluator," providing feedback to the actor on how good its actions are.
  - This helps the actor learn more efficiently by focusing on actions that lead to higher values.

# Proximal Policy Optimization (PPO) – Most popular variant of Actor-Critic What is it?

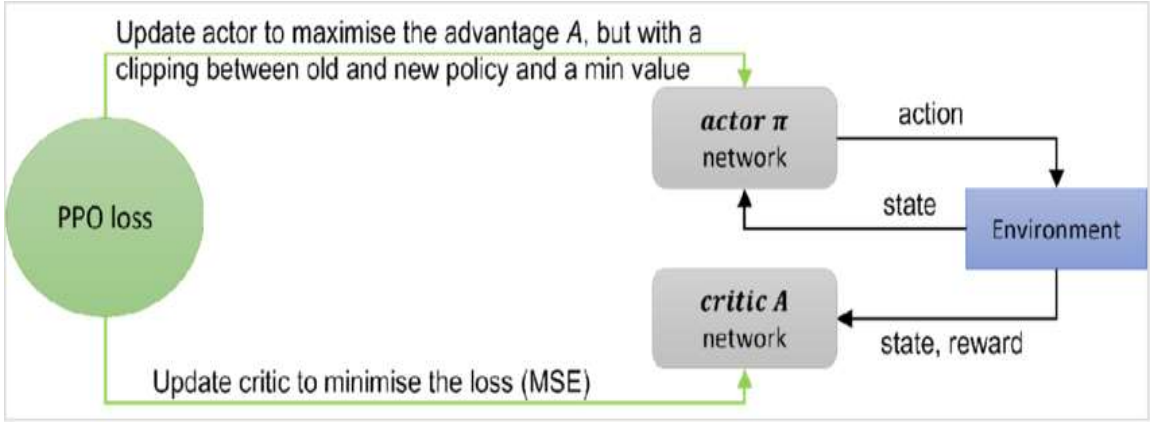
- Imagine teaching someone to **drive**.
  - The **actor** is the driver's hands and feet.
  - The **critic** is a calm instructor in the passenger seat saying, "that move was better/worse than average."
  - Training = lots of short drives. After each drive, you tweak the driver's habits.
- **PPO specific:**
  - You put **guardrails** on how much you're allowed to change the driver after each lesson.
  - Even if the instructor thinks "wow, turning hard like *that* worked great once," you **don't** overhaul the driver's whole style in one go.
  - You only allow **small, safe nudges** toward what seemed better.
  - So, If a suggested change is too big, PPO **clips** it — "that's past the guardrail; no extra credit for pushing further."
- **Why?**
  - Because sometimes a move looks amazing just by luck (traffic broke your way). If you overreact, you teach the driver bad habits.
  - PPO's guardrails keep each update **proportional but modest**, so learning is steady, not swingy.

# The “Advantage function”:
# The difference between actual reward and expected value

- Imagine you're training an agent to play a game, like a racing game or a simple Atari game:
  - The advantage function helps the agent figure out how *good* an action was in a particular situation,
  - It is measured as , compared to what it *expected* to be good.
- The advantage function provides a way to isolate:
  - how much better an action was than the average action in that situation.
  - It’s like, did the student perform above, same as or below the class average.
- The Importance of the Advantage computation
  - Without an advantage function, the agent would be constantly learning from noisy rewards. It's hard to tell if a reward was due to a good action or just random luck.
  - Focus on Improvement: The advantage function helps the agent focus on actions that actually improve its performance. It tells the agent which actions are worth learning.
  - Faster Learning: By focusing on good actions, the agent can learn much faster than if it had to learn from every single reward.

# Proximal Policy Optimization (PPO) – How does it work?

- Agent is in a particular state - s
- The actor network outputs actions from the probability distribution and from this action can be picked randomly or max value basis
- At the same time the critic network makes a prediction of the value of the current state – V(s ; θ)
- Based on the action taken by the actor the environment gives back (as always) a reward and the next state – s'
- The critic which predicted the value of state s can now predict the value of the next state and therefore compute what the real estimated value of s is using the Bellman equations as the foundation and the reward
  -> **V(s ; θ) = R(s) + γ V(s' ; θ)**
- Now the MSE can be computed between the predicted and estimated value as -> **MSE = (TD error)² = (R + γV(s'; θ) - V(s; θ))²**
- The actor network uses the same MSE loss, but we also add a couple of other terms to it (ratio of probability of previous to current action in the same state) and we clip the network update value to avoid big changes in actor parameter values.

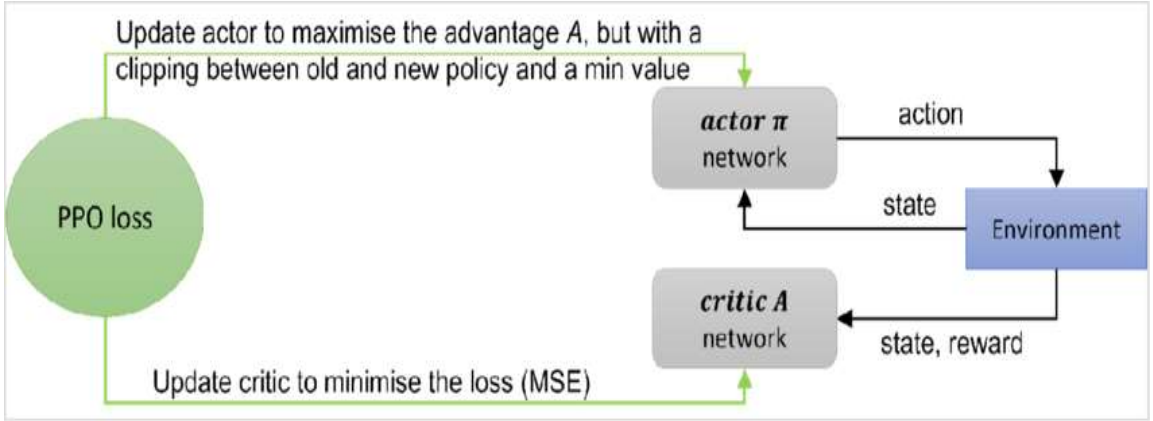



**Policy Network (actor**): This network is responsible for predicting the parameters of the policy. It takes the current state (`s`) as input and outputs the parameters that define the probability distribution over actions

**Value Network (critic):** This network estimates the value of being in a given state. It takes the current state (`s`) as input and outputs an estimate of the expected cumulative reward from that state onwards. This is often referred to as the state-value function.

# Proximal Policy Optimization (PPO) – Mathematically

- **Bellman-style target & error** (critic learning):
  - **Value of state after action:** $Q(st, a) = r_t + \gamma V_\phi(s_{t+1})$
  - **TD error (how surprised we were):** $\delta_t = Q(st, a) - V_\phi(s_t)$
- **Critic update:** make $V_\phi(s_t)$ closer to $Q(st, a)$)
- **Advantage** = "was that action better than expected?" → $A_t \approx \delta_t$
  - **If $A_t > 0$** then the action helped more than critic expected
  - **If $\hat{A}_t < 0$ :**it hurt.
- Actor update (policy learning) with PPO's guardrails:
  - increase the action's probability if $\hat{A}_t > 0$ and decrease if $\hat{A}_t < 0$, **but cap** the change so it can't move more than about $\pm\epsilon$ worth




**Intuition with examples**

$\hat{A}_t > 0, r_t = 1.2$" →nice, **slightly** prefer this action."
$\hat{A}_t > 0, r_t = 2.5$" →whoa, you're overreacting; **cap** the benefit."
$\hat{A}_t < 0, r_t = -0.4$" →ok, make it a bit rarer."
$\hat{A}_t < 0, r_t = -0.8$" →too harsh; **cap** the penalty."

# Applications of RL with PPO in industry

- **Optimizing Robotic Arm Control for Assembly Tasks**
  - Imagine a manufacturing line where robotic arms assemble intricate products, like smartphones or electronics. These robots need to perform precise movements in a continuous space to pick up components, align them correctly, and fasten them together.
  - **Electronics assembly:** Companies like Siemens and Foxconn are exploring using RL to optimize robotic assembly in electronics manufacturing.
  - **Automotive manufacturing:** RL is being used to improve the precision and efficiency of robotic arms for tasks like welding and painting in car factories.
- **When does training happen?**
  - **Offline Training:** Most of the training (simulation and potentially pre-training) happens offline before deployment.
  - **Fine-tuning and Adaptation:** Real-world fine-tuning and any ongoing learning occur after the robot is deployed.

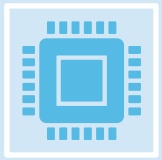
# Understanding the StableBaselines3 Library:

## Functionality and Applications in Reinforcement Learning

# What is StableBaselines3 (SB3)?

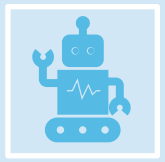
Stable Baselines is a popular Python library that provides implementations of various Reinforcement Learning (RL) algorithms

It is designed to help developers and researchers train RL models more easily and effectively and is user-friendly and modular.



This library helps streamline the development of RL models by offering tested and reliable algorithm implementations.

# Benefits of Using StableBaselines



Stable Baselines simplifies the process of setting up and training RL agents by providing pre-implemented, well-tested versions of various RL algorithms.

It abstracts away the low-level details of these algorithms, allowing users to focus on the higher-level aspects of RL, such as environment design, reward shaping, and agent evaluation.

Community Support: Extensive documentation and active community.

# Key Components of StableBaselines

Algorithms: Includes implementations of various RL algorithms
Supports various RL algorithms like DQN, PPO, A2C, etc.

Policies: Provides both standard and customizable policy architectures.

Environments: Integrates smoothly with Gymnasium environments.

Logging: Comes with built-in tools for tracking and visualizing training metrics.

# Getting Started with StableBaselines3

1. Installation: `pip install stable-baselines3`
2. Dependencies: Gymnasium, PyTorch
3. Please follow the instructions provided to get a working installation

Basic coding Setup example:

```
from stable_baselines3 import PPO
from gymnasium import make
env = make('CartPole-v1')
model = PPO('MlpPolicy', env, verbose=1)
model.learn(total_timesteps=10000)
```

Links: Stablebaselines3 , SB3-PPO , Custom-Policies , Getting-Started

# Model Evaluation and Visualization

Use `evaluate_policy` to assess model performance.

Visualize rewards using TensorBoard or Matplotlib.

# Advanced Functionalities

Hyperparameter Tuning: Adjusting parameters for better performance.

Callbacks: Creating custom training procedures.

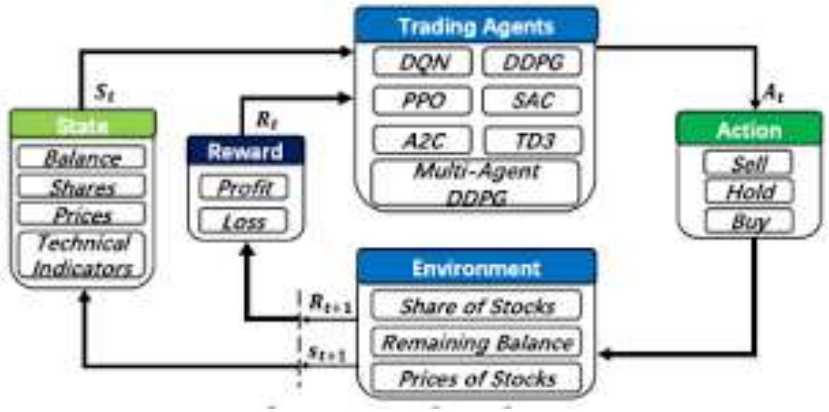
Vectorized Environments: Parallelizing training for efficiency.

# Let us train a “bipedal-walker” using SB3

# Building your own environment using gymnasium

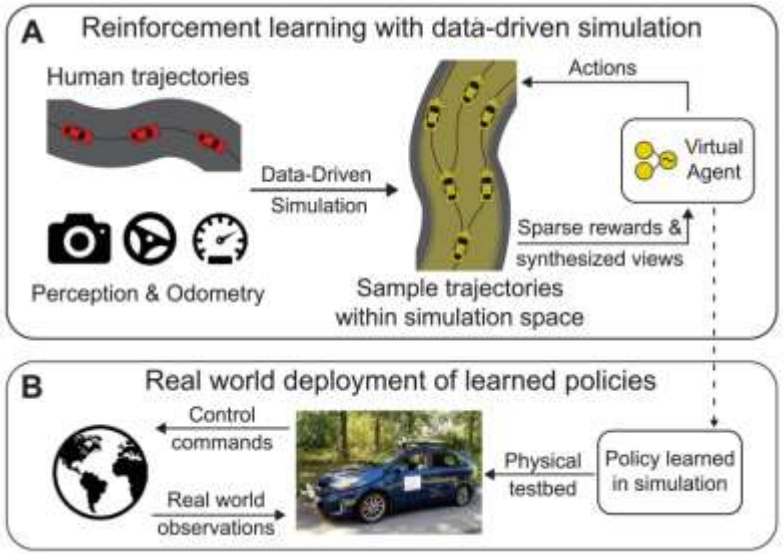
# Why would you build your own environment?

- **Unique Problem:** You might be working on a problem that has no readily available
- **Fine-grained Control:** Building your own environment gives you complete control over every aspect:
  - **State Space:** Define exactly what information the agent observes.
  - **Action Space:** Design the actions the agent can take, whether discrete (e.g., move left/right) or continuous (e.g., apply a specific force).
  - **Reward Function:** Craft a reward function that truly aligns with your desired agent behavior.
  - **Dynamics:** Determine how the environment changes in response to the agent's actions.



# Real world dynamics

- **Learning by Doing:** Implementing an environment from scratch forces you to deeply understand the underlying principles of RL and how environments interact with agents.
- **Simulating Real-World Scenarios:**
  - **Safe and Cost-Effective:**
    - RL environments allow you to simulate real-world scenarios that might be dangerous, expensive, or time-consuming to test in reality.
  - **Data Augmentation:** You can generate large amounts of synthetic data from your environment to supplement real-world data, improving the robustness of your RL agent.
  - **Research and Development:**
    - Novel Environments: Creating new environments can drive research in RL by introducing new challenges and benchmarks for the community.
    - Algorithm Testing: You can use your custom environment to test the performance and limitations of different RL algorithms.



# Gymnasium's ‘ENV’ Class provides us with the tools: Methods we can use

- **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.
- **spaces:**
  - **spaces.Discrete(n):** For a state that is a single number from \(0,…,n-1\).
  - **spaces.Box(low, high, shape, dtype):** For a state that is a continuous n-dimensional array, like a position and velocity.spaces.
  - **Dict({"key1": space1, "key2": space2}):** For a state composed of different, named components.

# Stepwise details

- **Create Environment Class:** Define a new Python class that inherits from "gymnasium.Env"
- **Initialize attributes:**
  - observation_space: Define the format of the state information the agent receives (e.g., using "gym.spaces.Box" for continuous values or "gym.spaces.Discrete" for discrete options).
  - action_space: Specify the possible actions the agent can take (e.g., using gym.spaces.Discrete for a set of actions).
- **Implement reset function:** This function is called at the start of each episode to set the environment to its initial state and return the initial observation.
- **Implement "step" function:**
  - Takes an action from the agent as input.
  - Updates the environment state based on the action.
  - Calculates and returns the new observation, reward, done flag (indicating episode termination), and any additional information.
- **Optional: Implement render function:** Visualizes the current state of the environment (can be a simple text output or a graphical display).

# Key points to remember

Flexibility:

- Design your environment to match the specific problem you want to solve in reinforcement learning.

Clear state representation:

- Ensure the observation space accurately reflects the information an agent needs to make decisions.

Reward function design:

- Carefully craft the reward structure to guide the agent towards the desired behavior.

Testing and debugging:

- Thoroughly test your custom environment to ensure it functions as expected

Video Link: Build your own Environment

# Let us code and build our own environment:

We shall use the same old Mineral Explorer

# Introduction to Sound or Audio as Data

# Sound as data – how should we understand it

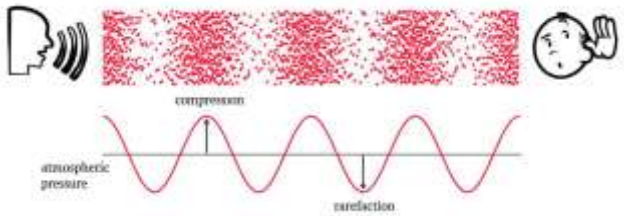
Easy way is to compare it with image as data -

- An image is a grid of:
  - tiny dots called **pixels,**
  - and each pixel has a number representing its color

- **Sound as data** is:
  - simply a long, ordered sequence of numbers,
  - each number represents how much the air pressure is vibrating (the loudness) at a specific instant in time.

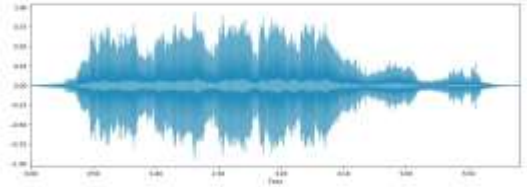
# What is audio?

- Audio, which is another form of unstructured data, is essentially **sound represented in a way that can be recorded, stored, transmitted, and reproduced**.

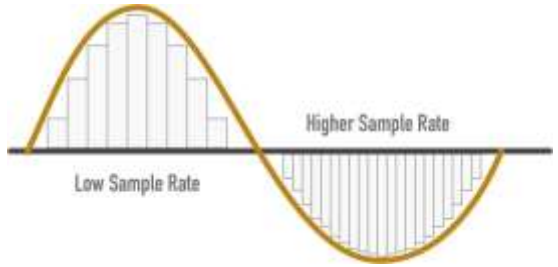



- It's the vibration of air molecules that our ears perceive as sound but captured and converted into a form that machines can understand.

- **Analog Representation:** This is the natural form of audio, where the sound waves are directly represented as continuous variations in a physical medium.



- **Digital Representation:**
  - This is how audio is stored and processed in computers and modern devices.
  - The continuous sound wave is converted into a series of discrete numbers (samples) through a process called sampling.
  - Each sample represents the amplitude of the sound wave at a specific point in time.

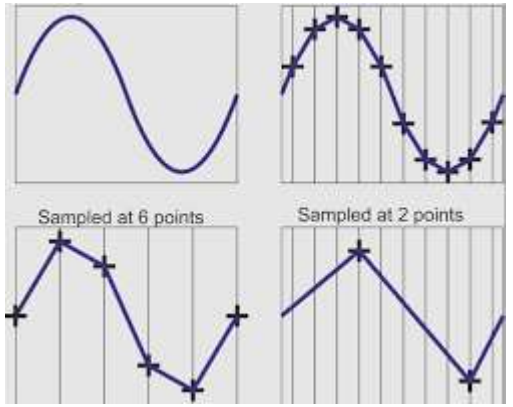



- Imagine a bird singing.
  - The bird's song itself is the audio.
  - **Analog representation:** Recording the song on Vinyl.
  - **Digital representation:** Recording the song on your smartphone as an MP3 file.

# Brief explanation of sampling

Think of a continuous audio signal as a smooth curve.

- Sampling is the process of taking discrete measurements of this curve at regular intervals.
- It's like taking snapshots of the curve at specific points in time.

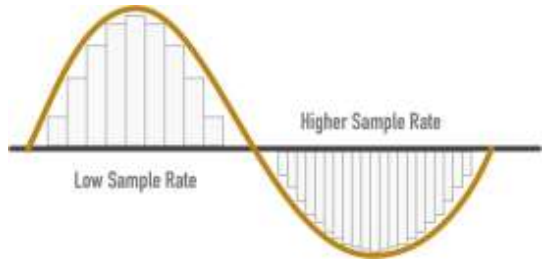



Why is Sampling Done?

- **Digital Representation:** Computers can only process digital data. Sampling converts the continuous analog audio signal into a digital format.
- **Storage and Transmission:** Sampled audio data can be easily stored and transmitted in digital form.
- **Feature Extraction:** Many feature extraction techniques, especially in classical machine learning, require discrete time-domain or frequency-domain representations.

How is Sampling Done?

- The sampling rate determines the number of samples taken per second.
- A higher sampling rate captures more detail of the original signal but also requires more storage space.




**Choosing the Right Sampling Rate:**

- The Nyquist-Shannon sampling theorem states that a signal must be sampled at a rate at least twice its highest frequency component to avoid losing information.
- For example, to accurately capture a signal with a maximum frequency of 20 kHz, you would need a sampling rate of at least 40 kHz.

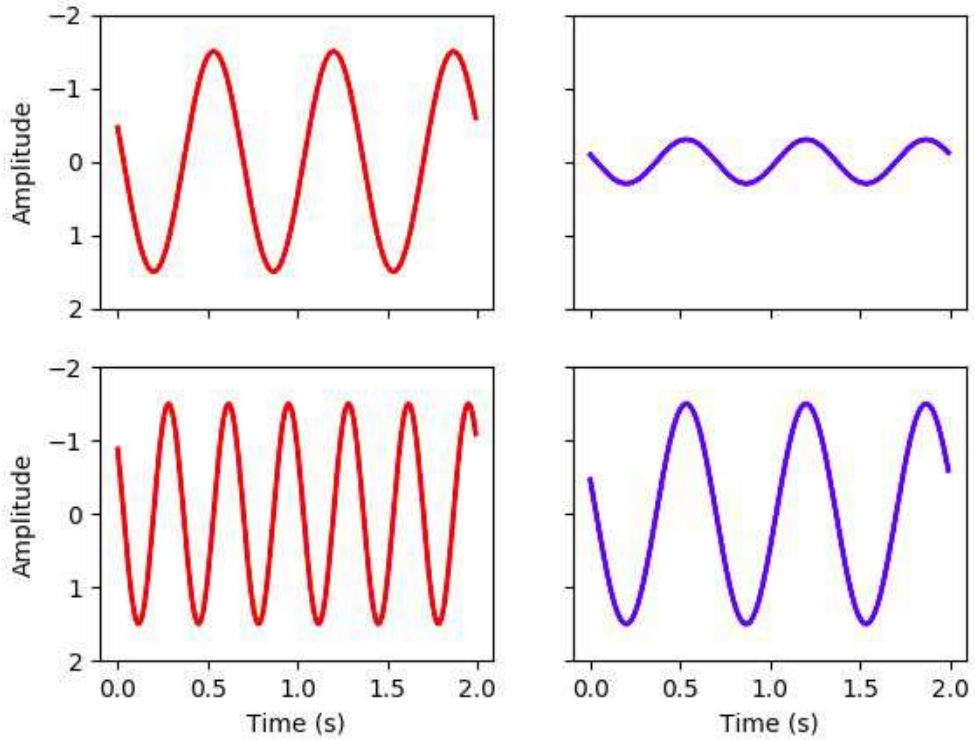
# Application with audio as data

- **Speech Recognition:** Relate this to how voice assistants like Siri or Google Assistant work. The audio is analyzed to understand the spoken words.
- **Music Genre Classification:** Think of how Spotify or Apple Music recommends songs. They analyze audio features to categorize music.
- **Sound Event Detection:** This is like how a smart home system recognizes the sound of a doorbell or a smoke alarm.
- **Sound classification:** Recognizing everyday urban sounds, identifying birds in nature from their tweets, studying bio-diversity in nature, etc.
- **Automatic Lyrics Alignment:** Implement a system that aligns lyrics with the sung audio in songs, useful for karaoke applications or music learning.
- **Bio-acoustic Monitoring:** Use audio deep learning for ecological studies, such as identifying bird calls or monitoring rainforest sounds to assess biodiversity.
- **Audio-based Health Diagnostics:** Develop tools for diagnosing health conditions through sound, such as analyzing cough sounds to detect respiratory diseases.

Let us start with an example of what is possible
**speech command recognition**

# Audia signal characteristics

# Frequency and amplitude




- *higher frequency -> higher pitch sound*
- *Measured in **Hertz (Hz)** (cycles per second).*
- *larger amplitude -> louder*
- *Measured using metrics like **Root Mean Square (RMS) energy***

**In analytics**: Complex sounds (like speech or music), we use a mathematical tool called a **Fourier Transform** to break the sound down and see exactly *which* frequencies are present at any given time – **spectrogram.**

# Sound Power and Intensity

***Power***

- *Rate at which energy is transferred*
- *Energy per unit of time emitted by a sound source in all directions*
- *Measured in watt (W)*

***Intensity***

- *Sound power per unit area*
- *Unit – $W/m^2$*
- *Threshold of hearing = $10^{-12}$* $W/m^2$.
- *Threshold of pain = 10 $W/m^2$*
- *Logarithmic scale*
- *Measured in decibels (dB)*

# Music related

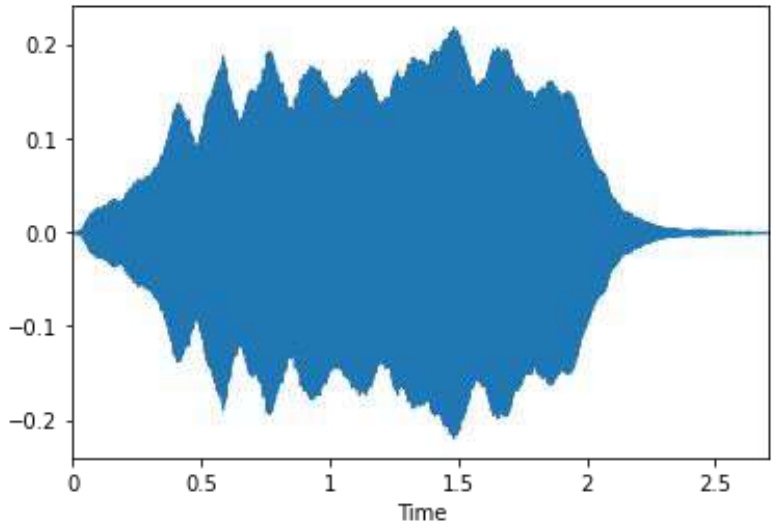
- Beat - rhythm
- Timbre – Sound quality and is a mix of different frequencies occurring simultaneously
- Pitch – high or low frequency
- Harmony – frequencies which are multiples of a fundamental
- many more

# ML needs features from audio data

# Audio data inherently has features in: time and frequency domains

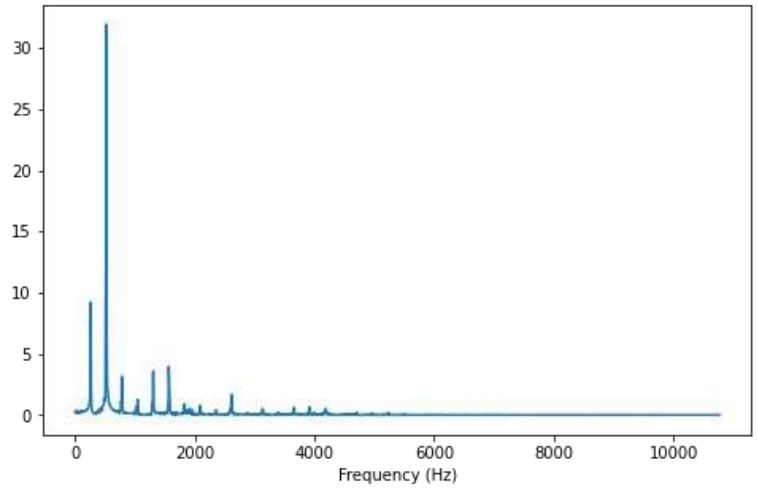
**Time-domain features:**

- Zero-crossing rate: The number of times the signal crosses the zero-amplitude line within a given time frame.
- Energy or Root mean square (RMS): The average loudness of the signal over a short period




**Frequency-domain features:**

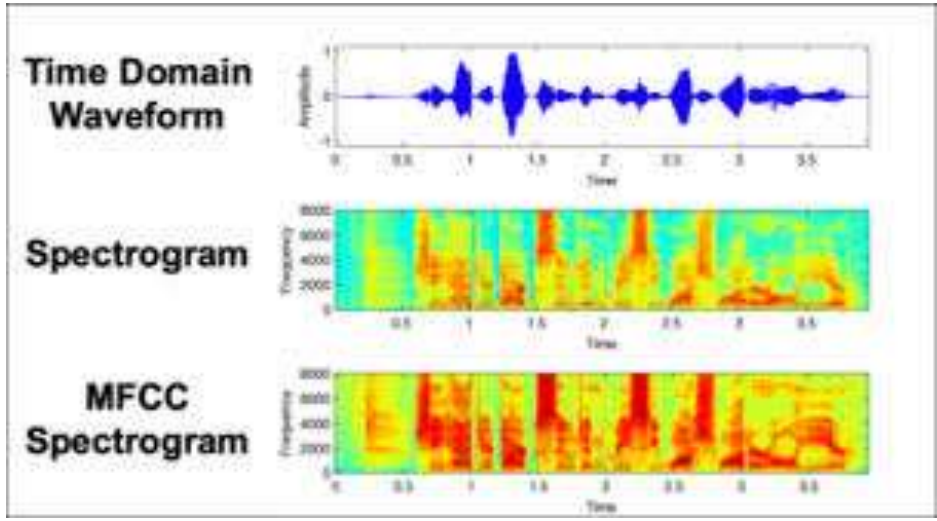
- Spectral centroid: The center of mass of the signal's spectrum.
- Spectral spread: A measure of how the spectrum is spread around the centroid.
- Spectral roll-off: The frequency below which a certain percentage of the signal's energy is contained.



# Audio data inherently has features in: Cepstral-Domain

**Cepstral-Domain features (gold standard):**

- Mel-Frequency Cepstral Coefficients(MFCCs)
- They are designed to model how the **human ear perceives sound** by separating the information about the sound source (vocal cords, instrument) from the information about the filter (vocal tract, instrument body).
- A set of coefficients that compactly represent the **spectral envelope** of a sound, based on the **Mel scale** (which is non-linear and approximates human hearing).
- The "fingerprint" of the sound's quality (timbre)
- **MFCCs** are the standard baseline feature for speech and audio classification.
- **Applications they are used in include:** Speech Recognition (speaker and word identification), environmental sound classification, and music genre classification.



In the next class we will study how to extract these features and the techniques available to build analytics around them

# Python packages we will use

- "librosa" - Python package designed for music and audio analysis
  - pip install librosa
  - pip install soundfile

- torchaudio – is a Python library built on top of PyTorch
  - Installed with PyTorch
  - designed for audio and signal processing, particularly within the context of machine learning.
  - primarily designed for preparing and processing audio data for deep learning models, rather than as a general-purpose signal processing library.

# End of Lesson