Module 2: Reinforcement Learning

2-1 Reinforcement Learning (RL) vs Supervised Learning (SL):

The Machine Learning implementations you have all seen up until now all fall under the category of **Supervised Learning**, which differs from **Reinforcement Learning** in that:

Supervised Learning (SL):

- The **Goal** is to learn a mapping $f(x) \to y$ from **Labeled Examples**
- **Feedback** is **Immediate** and **Complete**, determined by the target y
- You train a **Model** to minimize a **Loss Metric**.
- ***Data** is **Pre-Existing** and fed directly to the model.

Reinforcement Learning (RL):

- The **Goal** is to learn a **Policy/Strategy** to maximize **Performance/Reward**.
- **Feedback** is **Delayed** and **Incomplete**, we do not have a direct **Loss Metric** or **Targets**
- You train an **Agent** to interact with an **Enviorment** and maximize a **Reward**
- **Data** has to be gathered through the **Agent** interacting with the **Enviorment**

Supervised Learning	Reinforcement Learning
Has outcome information ("labels")	Makes decisions based on trial and error
Finds patterns that relate to those outcomes	Decision-making algorithm is constantly refined based on "rewards"
Uses patterns to predict outcomes not yet known	Excels in complex situations

2-2 Markov Decision Processes (MDPs):

A **Markov Decision Process** is a basic framework for solving **RL** problems where the learner and decision maker is called the **Agent**, which interacts with the **Environment**, which composes everything outside of it. **MDPs** involve the following terminology:

- **Agent:** The learner or decision maker.
- **Environment:** Everything the **Agent** interacts with; provides feedback in the form of **Rewards**.
- **State:** A representation of the current situation or configuration of the **Environment**.
- **Action:** The set of all possible moves the **Agent** can make.
- **Reward:** Feedback from the **Environment**.
- **Policy π:** Strategy used by the **Agent** to determine the next **Action** based on the current **State**
  Collection of conditional probabilities $P(action = a|state = s)$.
- **Value Function:** Estimates expected sum of all future **Rewards** from a **State** or **State-Action** pair.
- **Q-Value (Action Value):** Expected sum of all future **Rewards** for taking a specific **Action** in a given **State** and then following a given **Policy**.

This framework goes far beyond the **RL** we will touch in this class, so we will just focus on the **Agent**, **Environment**, **State**, **Action**, **Reward**, and **Policy**.

When determining what must be contained withing the **State** of your system, ask yourself the following questions:

- What information does the **Agent** need to properly learn?

- Which **Variables** in this information are not constants of the **Environment**?
- What possible combinations of these **Variables** exist?

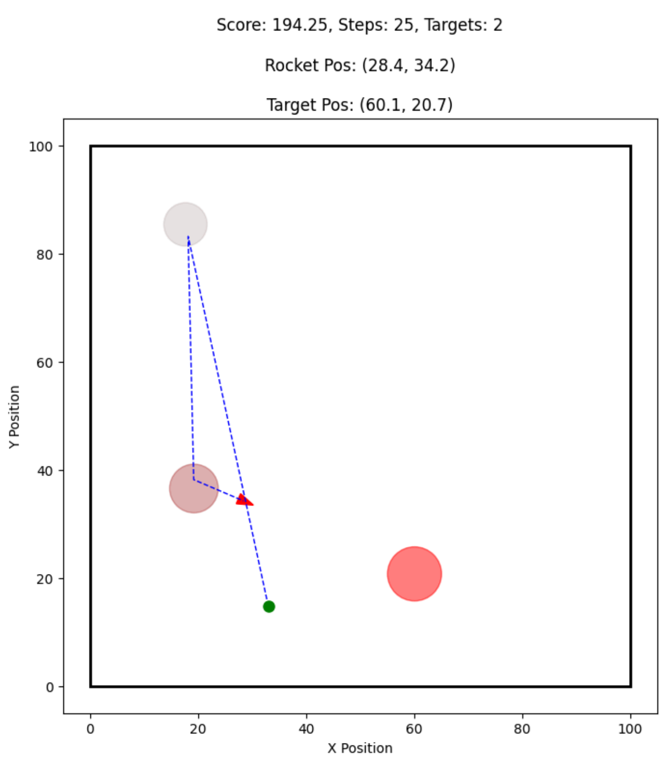
When determining **Rewards** for your system, ask yourself the following questions:

- What situations should be **Encouraged/Discouraged**? (positive/negative reward)
- Do the **Relative Magnitudes** of the **Rewards** make sense?
- Would a reasonable acting **Agent** end up with a total **Reward** that is centered around 0?
- Is enough pressure to learn being applied to the **Agent** through the **Rewards**? Is it too much?

Example: You are tasked with using **Reinforcement Learning** to train a Rocket Ship to navigate towards a target. Assume the ship can only change its heading and the speed at which it travels. Cap its speed to a predefined value. Assume the ship is bounded to a rectangular area of space in 2D and that if it crosses over the boundary it is clipped back to the bounds and its velocity is reflected off the boundary. Every time it comes within a set radius of the target, it destroys it and another appears. Define all of the relevant terms for **RL** in this class as it pertains to this task.

```
Agent = The Rocket Ship

Environment = The Region of Space and Targets

State = Target's current position and the Rocket Ship's current position and

Actions = The headings and speeds the rocket ship can select (up to a maximum speed)

Rewards = Defined rewards for hitting the target (positive), hitting the bounds (negative),
attempting to exceed max speed (negative), and taking a valid step (negative).
You could also consider additional rewards to encourage higher speeds, getting closer to the target, etc...

Policy = The way in which the Rocket Ship selects its desired heading and speed given its current position and the position of the target.
```

Code Implementation: Code the **Environment** for the previous example (Generation code in the matching notebook).



Note that the number of possible **State-Action** combinations can scale insanely fast, with this problem having:

$$N = (num\_of\_rocket\_positions) * (num\_of\_target\_positions) * (num\_of\_headings) * (num\_of\_speeds)$$

Say that instead of the continuous implementation above we use a grid of $(100 * 100)$ and speeds of $(0 - 5)$ in 0.1 increments with a heading resolution of 1°.

This results in the number of possible **State-Action** combinations being given by:

$$N \approx (100^2) * (100^2) * (360) * (51) \approx 1,836,000,000,000$$

2-3 Epsilon Greedy (EG) Algorithm:

The **Epsilon Greedy (EG)** Algorithm is a **Policy** for **MDPs**. Recall that a **Policy** is defined by a collection of conditional probabilities which describe the probability that the **Agent** takes some **Action** a given that it is currently in some **State** s:

$$\pi \rightarrow P(action = a|state = s)$$

Epsilon Greedy utilizes a hyperparameter ε to form a **Policy** which selects randomly from all valid **Actions** at a given **State** with a probability of ε (**Explore**), and selects the **Action** at that **State** with the highest average **Reward** with a probability $1 - \varepsilon$ (**Exploit**):

When implementing **EG** and other simple **MDPs**, it is typically most convenient to utilize two matrices of identical shape:

- **Wins:** To keep track of the total **Reward** received by each **Action-State** combination.
- **Visits:** To keep track of the number of times each **Action-State** combination has been selected.

Similarly, there are two ways to handle **Initialization** of such processes:

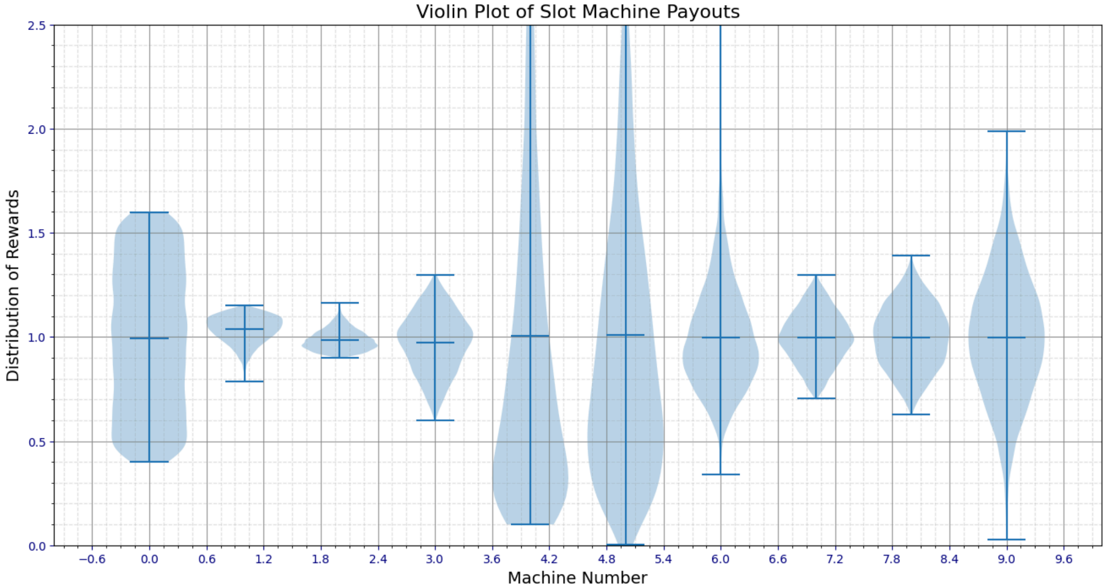
- **Full-Initialization:** Simulate the **Agent** selecting every **Action-State** combination and update the **Wins** and **Visits** matrices.
- **Initialization-Free:** Utilize some small $eps \rightarrow (10^{-8} - 10^{-6})$ to avoid any division by 0 issues when the **Visits** value of an **Action-State** combination is 0.

Typically, we only use **Full-Initialization** when the number of **Action-State** combinations is small and they are easy to simulate.

Example: Code a **Class** which uses the **EG** Algorithm with $\varepsilon = 0.25$ to determine the distribution combination (no double selection) with the highest average product. Use **Full-Initialization**. A violin plot of the distributions can be seen below alongside statistics regarding each distribution's mean (Generation code in the matching notebook).

Mean Data:

```
Mean #0:   0.999838         1 - Mean #0:   0.000162
Mean #1:   1.035575         1 - Mean #1:   -0.035575
Mean #2:   0.985782         1 - Mean #2:   0.014218
Mean #3:   0.973406         1 - Mean #3:   0.026594
Mean #4:   1.000928         1 - Mean #4:   -0.000928
Mean #5:   1.000079         1 - Mean #5:   -0.000079
Mean #6:   0.999407         1 - Mean #6:   0.000593
Mean #7:   0.999641         1 - Mean #7:   0.000359
Mean #8:   0.999878         1 - Mean #8:   0.000122
Mean #9:   1.000198         1 - Mean #9:   -0.000198
```



Example Code:

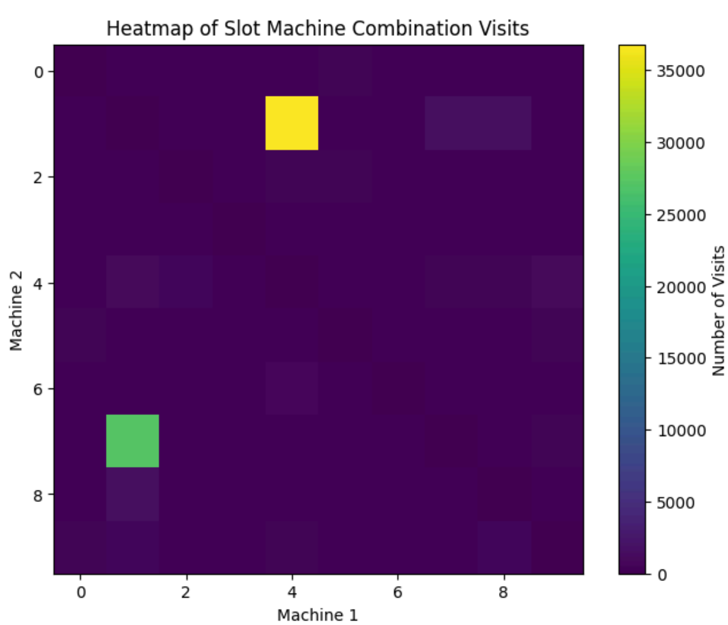
```python
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

class EGSlotCombos:
    def __init__(self, num_machines, dist_list):
        #Store number of slot machines, if trained
        self.num_machines = num_machines
        self.dist_list = dist_list
        self.trained = False

    def __play_combo(self, combo, N):
        '''Play a given slot machine combination N times and return the average product'''
        mean_product, _, _ = DistCombos(combo, N, self.dist_list, check_valid=True)
        return mean_product

    def train(self, max_plays=10_000, epsilon=0.25):
        '''Train the epsilon-greedy algorithm on the slot machine combinations'''
        #Initialize variables
        print(f"Training with max_plays={max_plays}, epsilon={epsilon}\n")
        self.wins = np.zeros((self.num_machines, self.num_machines)) #Includes double sel. to simplify
        self.visits = np.zeros((self.num_machines, self.num_machines)) #Includes double sel. to simplify
        plays = 0

        #Initialize by playing each combination once
        for i in range(self.num_machines):
            for j in range(self.num_machines):
                if i != j:
                    self.wins[i,j] += self.__play_combo((i,j), 1)
                    self.visits[i,j] += 1
                    plays += 1

        #Continue playing until max_plays is reached
        while plays < max_plays:
            if plays % 10_000 == 0:
                print(f"Plays: {plays}/{max_plays}")
            #Decide whether to explore or exploit
            if np.random.rand() < epsilon:
                #Explore: choose a random valid combination
                i, j = np.random.choice(self.num_machines, size=2, replace=False)
            else:
                #Exploit: choose the best known combination
                avg_rewards = np.divide(self.wins, self.visits, out=np.zeros_like(self.wins), where=self.visits!=0)
                np.fill_diagonal(avg_rewards, -np.inf) #Exclude double selections
                i, j = np.unravel_index(np.argmax(avg_rewards), avg_rewards.shape)

            #Play the chosen combination and update wins and visits
            self.wins[i,j] += self.__play_combo((i,j), 1)
            self.visits[i,j] += 1
            plays += 1

        #Mark as trained
        print("\nTraining complete.\n\n")
        self.trained = True

    def results(self):
        '''Return and plot the results of the training'''
        #Check if trained
        if not self.trained:
            raise ValueError("The model must be trained before retrieving results.")
```

```
        #Print the most visited combination and its average reward
        avg_rewards = np.divide(self.wins, self.visits, out=np.zeros_like(self.wins), where=self.visits!=0)
        np.fill_diagonal(avg_rewards, -np.inf) #Exclude double selections
        best_i, best_j = np.unravel_index(np.argmax(self.visits), self.visits.shape)
        print(f'Best combination: Machine {best_i} & Machine {best_j} with {int(self.visits[best_i, best_j])} visits and average reward {av

        #Display the visits as a heatmap
        plt.figure(figsize=(8,6))
        plt.imshow(self.visits.T, cmap='viridis', interpolation='nearest')
        plt.colorbar(label='Number of Visits')
        plt.title('Heatmap of Slot Machine Combination Visits')
        plt.xlabel('Machine 1')
        plt.ylabel('Machine 2')
        plt.show()
```

```
#Run the epsilon-greedy slot machine combination algorithm
EGSlot = EGSlotCombos(num_machines=10, dist_list=dist_list)
EGSlot.train(max_plays=100_000, epsilon=0.25)
EGSlot.results()
```

Output:

```
Training with max_plays=100000, epsilon=0.25

Plays: 10000/100000
Plays: 20000/100000
Plays: 30000/100000
Plays: 40000/100000
Plays: 50000/100000
Plays: 60000/100000
Plays: 70000/100000
Plays: 80000/100000
Plays: 90000/100000

Training complete.


Best combination: Machine 4 & Machine 1 with 36794 visits and average reward 1.036464
```



Note that the best combination will fluctuate between all valid combinations containing Game $\#1$ and any other game that is not $\#2$ or $\#3$.

2-4 Upper Confidence Bound (UCB) Algorithm:

The **EG** Algorithm can be very inefficient, as it does not have any notion of a **Confidence Level** in its selections. Thus, the algorithm's decision to **Explore/Exploit** is completely random and only dependent on ε and its evaluation of the **Best Action** when **Exploiting** does not take into account the amount of data currently stored. Improvements can be made upon this implementation by introducing a quantifiable **Confidence Level** when determing the **Best Action** based upon current data.

According to the **CLT**, as the number of samples from a random variable X increases above $n \approx 30$, The mean of these samples will be approximately **Normally Distributed**. Using this, we can define a **Confidence Level** $(1-\alpha)$ where the probability that the **Sample Mean** $\bar{X}$ and the **True Mean** μ are within a specified **Margin of Error (MOE)** d is given by:

$$Pr(|\bar{X}-\mu| < d) > (1-\alpha)$$

Where:

$$\bar{X} - d < \mu < \bar{X} + d$$

The **MOE** d is calculated by:

$$d = z_{\alpha/2}\frac{\sigma}{\sqrt{n}}$$

OR

$$d = t_{\alpha/2}\frac{s}{\sqrt{n}}$$

Where $z_{\alpha/2}$ and $t_{\alpha/2}$ are pulled from the **Standard Normal** and **Standard** t distributions. $t_{\alpha/2}$ is used when the true standard deviation of the distribution σ is unknown (which is almost always the case).

The interval $(\bar{X}-d, \bar{X}+d)$ is know as a **Confidence Interval** and can be interpreted as saying that the **True Mean** μ can be said to be within the interval with $(1-\alpha)$ confidence,95% at $\alpha = 0.05$ for example.

These ideas are expanded upon in the **Upper Confidence Bound (UCB)** Algorithm to construct an optimistic upper bound estimate for the **Average Reward** of each **State-Action** combination as follows:

$$UCB_i = \mu_i = \bar{X}_i + c\sqrt{\frac{\ln(k)}{n}}$$

Where c is a tunable **Hyperparameter** (typically 2), k is the total number of times any **Action** has been selected at the current **State**, and n is the total number of times the **State-Action** combination has been selected.

The **UCB** algorithm uses these values as its metric with which to score all **State-Action** combinations, selecting the best one as follows:

$$i_{star} = argmax(UCB_i)$$

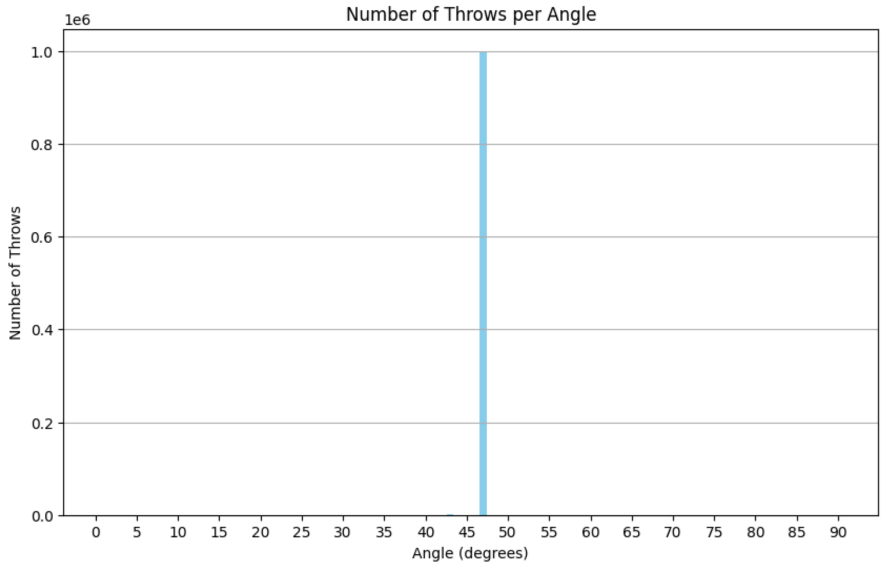
Additionally, **UCB** will typically only **Exploit**, with the **Explore** structure from **EG** not being used with **UCB** in this class.

Example: Using **UCB** and an **Initialization-Free** approach, code a **Class** to train an agent to throw a ball as far as possible, allowing it to throw at a set speed and at any angle from $(1^\circ - 90^\circ)$ above the horizontal in increments of 1°. Add some percentage of uniform noise to the throwing speed for each throw.

```python
import numpy as np
import matplotlib.pyplot as plt

class BallThrower:
    def __init__(self, v0=100, g=9.81):
        #Store initial velocity
        self.v0 = v0  #Initial velocity (m/s)
        self.g = g    #Acceleration due to gravity (m/s^2)

    def __distance(self, angle):
        '''Calculate the distance a rocket flies given an angle'''
        #Calculate distance using the range formula
        v0i = np.random.uniform(0.9*self.v0, 1.1*self.v0)  #Initial velocity with noise
        angle_rad = np.deg2rad(angle)  #Convert angle to radians
        distance = (v0i**2*np.sin(2*angle_rad))/self.g  #Range formula
        return distance

    def train(self, num_throws=1_000_000, c=2, eps=1e-8):
        '''Train the ball thrower using UCB'''
        #Initialize state matrices and variables
        self.wins = np.zeros(90)  #Total distance for each angle
        self.visits = np.zeros(90)  #Number of throws for each angle
        self.k = 0

        #Training Loop
        for i in range(num_throws):
            self.k += 1
            #Select angle using UCB
            angle = np.argmax(self.wins/(self.visits + eps) + c*np.sqrt(np.log(self.k)/(self.visits + eps)))

            #Throw the ball and update wins and visits
            distance = self.__distance(angle + 1) #+1 to convert index to angle in degrees
            self.wins[angle] += distance
            self.visits[angle] += 1

    def results(self):
        '''Return and plot the results of the training'''
        #Print the best angle and its average distance
        avg_distances = np.divide(self.wins, self.visits, out=np.zeros_like(self.wins), where=self.visits!=0)
        best_angle = np.argmax(self.visits) + 1  #+1 to convert index to angle in degrees
        print(f'Best angle: {best_angle}° with average distance {avg_distances[best_angle-1]:.2f} meters\n')

        #Display the visits as a bar chart
        plt.figure(figsize=(10,6))
        plt.bar(range(1, 91), self.visits, color='skyblue')
        plt.title('Number of Throws per Angle')
        plt.xlabel('Angle (degrees)')
        plt.ylabel('Number of Throws')
        plt.xticks(range(0, 91, 5))
        plt.grid(axis='y')
        plt.show()


#Run the UCB ball thrower algorithm
thrower = BallThrower(v0=100, g=9.81)
thrower.train(num_throws=1_000_000, c=2, eps=1e-8)
thrower.results()
```

Output:

```
Best angle: 47° with average distance 1020.47 meters
```



2-5 EG & UCB Differences/Limits:

There are certain considerations you have to account for when deciding between **EG** and **UCB** for an **RL** problem, some of which are as follows:

EG:

- Can easily handle **Reward** systems which change over time.
- Converges very inefficiently, spending time on clearly subpar **Actions**.
- Computations are much quicker for small numbers of possible **Actions**.

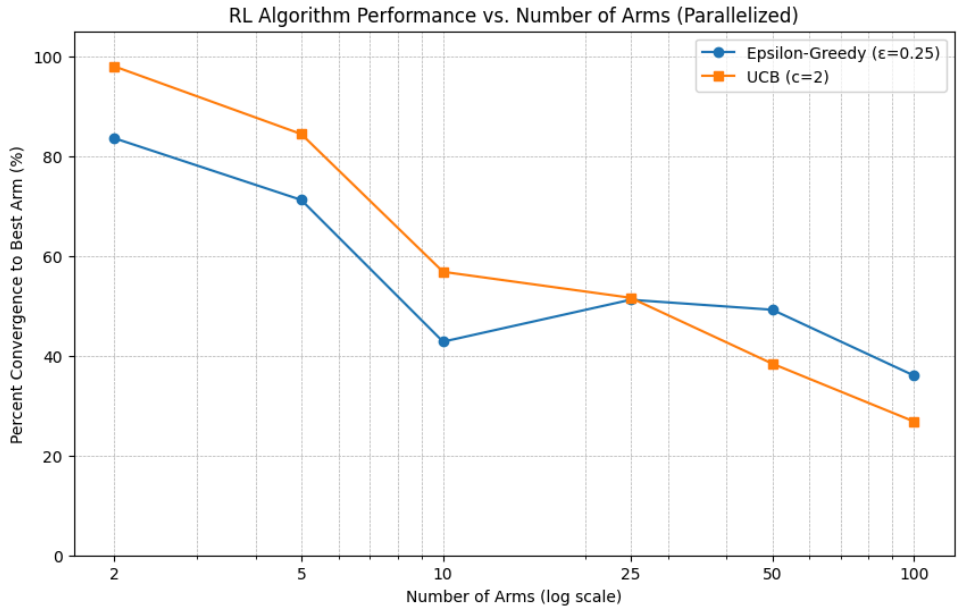
UCB:

- Requires **Discount Values** to handle **Reward** systems which change over time.
- Converges more efficiently, only **Exploring** to under-sampled **Actions**.
- Can get stuck when the number of possible **Actions** is high, preventing convergence.
- Much more computationally intensive for small numbers of possible **Actions**.

Both of these algorithms are considered very simple **MDPs** and have severe limitations, causing them to be ineffective when:

- The number of possible **Actions** is immense (such as in the rocket ship example).
- The potential **Actions** are not discrete and finite (e.g. the throw angle without a resolution).
- You need to generalize across different **States** (such as learning to evaluate the quality of a chess game).
- You have **Adversarial Components** (such as an opponent in a game whose behavior cannot be fully know by the **Agent**).

Below is an example of how **EG** and **UCB** could converge on a specific **RL** task as the number of possible **Actions** increases (Generation code in the matching notebook):


RL Algorithm Performance vs. Number of Arms (Parallelized)
Epsilon-Greedy (ε=0.25)
UCB (c=2)
Percent Convergence to Best Arm (%)
Number of Arms (log scale)
0
20
40
60
80
100
2
5
10
25
50
100