RL - Reinforcement Learning

Exploration vs Exploitation

Prof. Riccardo Berta

2025.10.02

Contents

1 Exploitation vs Exploration **1**
1.1 Multi-armed bandit problem . . . 1
1.2 Estimation of the action value . . . 3
1.3 Basic exploration strategies . . . 6
1.3.1 Pure exploration and pure exploitation . . . 6
1.3.2 Epsilon greedy . . . 8
1.3.3 Decaying epsilon-greedy . . . 12
1.3.4 Optimistic initial values . . . 16
1.4 Strategic Exploration . . . 18
1.4.1 Softmax method . . . 19
1.4.2 Upper Confidence Bound method . . . 22
1.5 Comparison . . . 25

1 Exploitation vs Exploration

For an agent is really important **to explore** when interacting with **uncertain environments**, problems in which the MDP isn't available for planning. In these cases, the agent must **learn from experience** taking decisions about what actions to take. However, no matter how small and unimportant a decision may seem, every decision we make is a **trade-off between information gathering and information exploitation**. For example, when we go to our favorite restaurant, should we order our favorite dish, yet again, or should we request that dish we've been meaning to try? These kinds of questions illustrate the **exploration-exploitation dilemma** and are at the core of the reinforcement learning problem. It boils down to deciding **when to acquire knowledge** and **when to capitalize on knowledge previously learned**. It's a challenge to know **whether the good we already have is good enough**. In order to examine this dilemma, we will consider simplified environments that aren't sequential, but one-shot: the **multi-armed bandits** (MABs) problem.

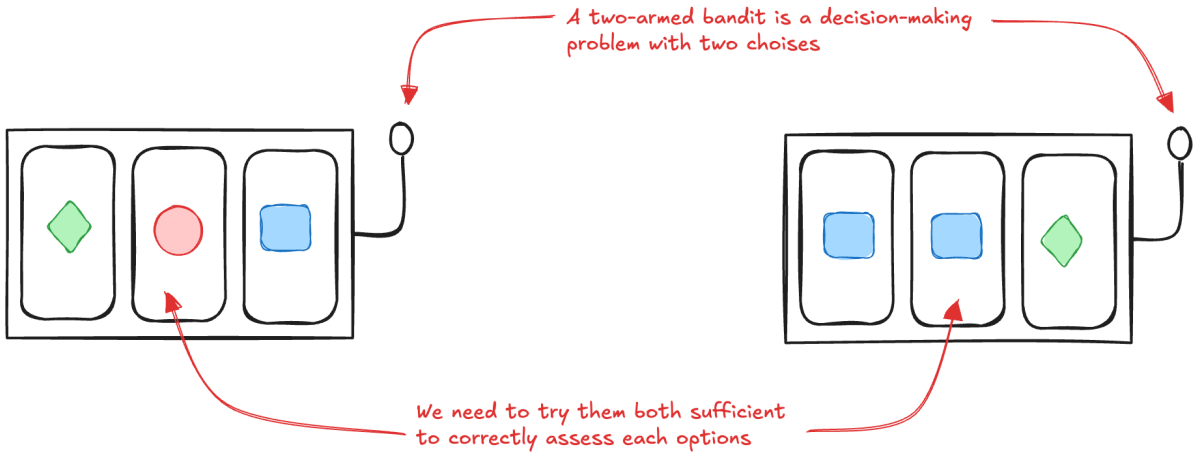
1.1 Multi-armed bandit problem

Multi-armed bandits (MAB) are a **special case of RL problem** in which the size of the state space and horizon equal to one. A MAB has **multiple actions**, a **single state**, and a **greedy horizon**. We can think of it as a "many-options, single-choice" environment (named by analogy to a slot machine). In formal terms we are faced repeatedly with a choice among k different

actions. After each choice we receive a reward chosen from a **stationary probability distribution** that depends on the selected action. The action-value function of action a is the expected reward given that a is sampled:

$q(a) = E[R_t|A_t = a]$

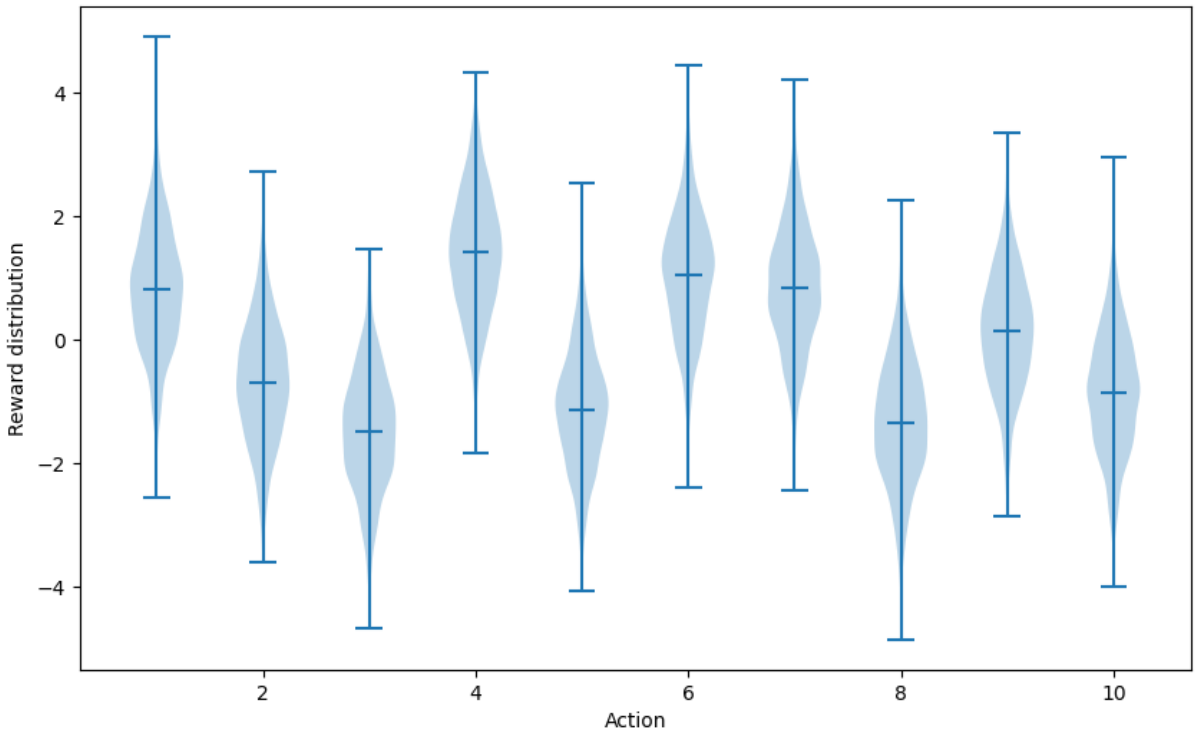
The goal of MAB is very similar to that of RL. In RL, the agent needs to maximize the expected "cumulative" and "discounted" reward, but in MABs we only have a single chance of selecting an action. Therefore, we can exclude the words that don't apply to the case: we remove "cumulative", because there's only a single time step; and "discounted", because there are no next states to account for. In MABs, the goal is to **maximize the expected reward**. Notice that the word "expected" stays because there's **stochasticity** in the environment.



We can make a numerical simulation with a 10-armed bandit problem. The true value of each of the ten actions is selected according to a normal distribution with zero mean and unit variance, and then the actual rewards are selected from a normal distribution with mean equal to the action value and unit variance:

```
import numpy as np

class BanditEnv:
    def __init__(self):
        self.size = 10
        self.means = np.random.randn(self.size)

    def step(self, action):
        return np.random.normal(loc=self.means[action])

env = BanditEnv()
```

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10,6))
plt.violinplot(np.random.randn(2000, 10) + env.means, showmeans=True)
plt.xlabel("Action")
plt.ylabel("Reward distribution")
plt.show()
```



If we knew the value of each action, then it would be trivial to solve the problem, we always select the action with highest value:

$a_* = \underset{a}{\text{argmax}}\, q(a)$

However, we assume to not know the action values with certainty, although we may have estimates.

1.2 Estimation of the action value

The estimation of the action-value function in MAB environments is pretty straightforward. Because MABs are one-step environments, to estimate the value of an action, we need to calculate the average reward obtained by selecting that action. In other words, the estimate of an action

value is equal to the total reward obtained when selecting the action, divided by the number of times action a has been selected:

$$q_t(a) = \frac{\sum_{i=1}^{t-1} R_i \cdot 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}}$$

We call this estimation the **sample-average method**, because each estimate is an average of the samples of relevant rewards.

The obvious implementation would be to maintain a record of all the rewards and then perform the computation whenever the estimated value was needed. However,the memory and computational requirements would grow over time as more rewards are seen. Each additional reward would require additional memory to store it and additional computation to compute the sum in the numerator. This is not really necessary. It is easy to devise an **incremental formulas** for updating averages with small, constant computation required to process each new reward. To simplify notation, we concentrate on a single action:

$$q_{t+1} = \frac{R_1 + R_2 + ... + R_t}{t} = \frac{1}{t}\sum_{i=1}^{t} R_i = \frac{1}{t}\left(R_t + \sum_{i=1}^{t-1} R_i\right) = \frac{1}{t}\left(R_t + (t-1)\frac{1}{t-1}\sum_{i=1}^{t-1} R_i\right) =$$

$$= \frac{1}{t}\left(R_t + (t-1)q_t\right) = \frac{1}{t}\left(R_t + tq_t - q_t\right) = q_t + \frac{1}{t}(R_t - q_t)$$

This implementation requires memory only for q_t and t, and only a small computation for each new reward. This update is of a form of the general rule:

NewEstimate <-- OldEstimate + StepSize (Target - OldEstimate)

The expression "Target−OldEstimate" is an error in the estimate, and it is reduced by taking a step toward the "Target", which is presumed to indicate a desirable direction in which to move, though it may be noisy. In our case above, the target is the reward and the step-size parameter changes from time step to time step, in processing the t reward for action a, the method uses the step-size 1/t.

The averaging method is appropriate for **stationary** bandit problems, in which the reward probabilities do not change over time. However, we often encounter problems that are **non-stationary**. In such cases, it makes sense to give more weight to recent rewards than to long-past rewards. One of the most popular ways of doing this is to use a constant step-size parameter:

$$q_{t+1} = q_t + \alpha(R_t - q_t)$$

where the step-size parameter $\alpha \in (0, 1]$ is constant. This results in q_t being a weighted average of past rewards and the initial estimate q_1:

$$q_{t+1} = q_t + \alpha(R_t - q_t) = \alpha R_t + (1-\alpha)q_t =$$

$= \alpha R_t + (1-\alpha)(\alpha R_{t-1} + (1-\alpha)q_{t-1}) =$

$= \alpha R_t + (1-\alpha)\alpha R_{t-1} + (1-\alpha)^2 q_{t-1} =$

$= \alpha R_t + (1-\alpha)\alpha R_{t-1} + (1-\alpha)^2 \alpha R_{t-2} + ... + (1-\alpha)^{t-1}\alpha R_1 + (1-\alpha)^t q_1$

$= (1-\alpha)^t q_1 + \sum_{i=1}^{t} \alpha(1-\alpha)^{t-1} R_i$

Sometimes it is convenient to vary the step-size parameter from step to step. Let $\alpha_t(a)$ denote the step-size parameter used to process the reward received after the t selection of action a. In that case the weighting on each prior reward is:

$q_{t+1} = q_n + \alpha_n(R_t - q_t) = \alpha_t R_t + (1-\alpha_t)q_t =$

$= \alpha_t R_t + (1-\alpha_t)(\alpha_{t-1}R_{t-1} + (1-\alpha_{t-1})q_{t-1}) =$

$= \alpha_t R_t + (1-\alpha_t)\alpha_{t-1}R_{t-1} + (1-\alpha_t)(1-\alpha_{t-1})q_{t-1} =$

$= \alpha_t R_t + (1-\alpha_t)\alpha_{t-1}R_{t-1} + (1-\alpha_t)(1-\alpha_{t-1})\alpha_{t-2}R_{t-2} + ... + (1-\alpha_t)(1-\alpha_{t-1})...(1-\alpha_1)q_1 =$

$= \prod_{i=1}^{t}(1-\alpha_i)q_1 + \sum_{i=1}^{t}\alpha_i \prod_{j=i+1}^{t}(1-\alpha_j)R_i$

The choice $\alpha_t(a) = 1/n$ results in the sample-average method, which is guaranteed to converge to the true action values by the law of large numbers. But of course convergence is not guaranteed for all choices of $\alpha_t(a)$. A result in stochastic approximation theory gives us the **conditions required to assure convergence**:

$$\sum_{t=1}^{\infty}\alpha_t(a) = \infty \qquad \sum_{t=1}^{\infty}\alpha_t^2(a) < \infty$$

The first condition is required to guarantee that the **steps are large enough** to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the **steps become small enough** to assure convergence. Note that both convergence conditions are met for the sample-average case, but not for the case of constant step-size parameter. In the latter case, the second condition is not met, indicating that the estimates never completely converge, but continue to vary in response to the most recently received rewards. **This is actually desirable in a non-stationary environment**.

All the methods we have discussed so far are biased (dependent to some extent) by their initial estimates. For the sample-average methods, the bias disappears once all actions have been selected at least once, but for methods with constant α, the bias is permanent, though decreasing over time. In practice, this kind of bias is usually not a problem and can sometimes be very helpful to supply some **prior knowledge** about what level of rewards can be expected.

It's essential to highlight that there are no differences in how strategies estimate the action-value function, the only difference is in how each strategy uses the estimates to select actions.

1.3 Basic exploration strategies

If we maintain the estimates of the action values, then at any time step there is at least one action whose estimated value is greatest. We call this the **greedy action**. When we select this action, we say that we are **exploiting our current knowledge** of the values of the actions. If there is more than one greedy action, then a selection is made among them in some arbitrary way, perhaps randomly.

$A_t = \underset{a}{\text{argmax}}\, q_t(a)$

Greedy action selection always **exploits** current knowledge **to maximize immediate reward**, it spends no time at all sampling apparently inferior actions to see if they might really be better. Instead, if we select one of the non-greedy actions, then we say we are **exploring**, because this enables to improve the estimate of the non-greedy action's value.

Exploitation is the right thing to do to maximize the expected reward on the one step, but exploration may produce a greater total reward in the long run.

Unfortunately, it is not possible both to explore and to exploit with any single action selection, one often refers to the **conflict between exploration and exploitation**. The need to balance exploration and exploitation is a distinctive challenge that arises in reinforcement learning.

Notice the relationship between a greedy strategy and time. If the agent only has one episode left, the best thing is to act greedily (if you know you only have one day to live, you'll do things you enjoy the most): this is a reasonable thing to do when we have limited time left. However, if we don't, we can trade-off immediate satisfaction for gaining of information that would allow us better long-term results.

Notice also that while there's only a single way to exploit, there are **multiple ways to explore**. Exploiting is nothing but doing what you think is best, exploring, on the other hand, is much more complex. It's obvious you need to collect information, but how is a different question. We could try gathering information to support your current beliefs. We could gather information to attempt proving ourself wrong. We could explore based on confidence, or based on uncertainty. The list goes on...

1.3.1 Pure exploration and pure exploitation

The **greedy strategy** (always exployt) and the **pure random strategy** (always explore) aren't really streategies, but two important baselines.

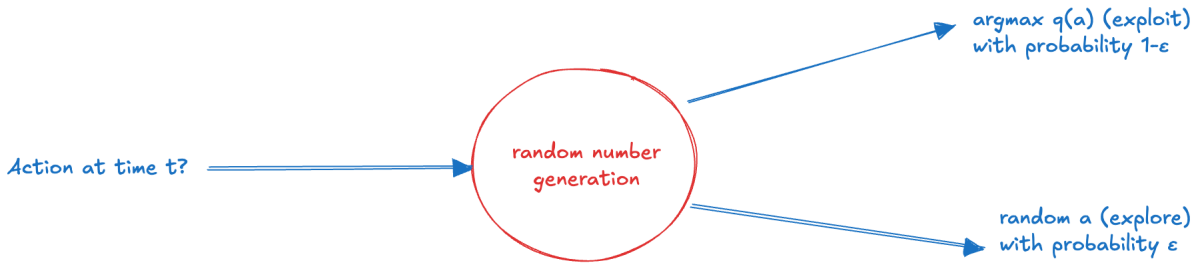
```python
def pure_exploitation(env, n_episodes=1000):

    # initialize the Q-function and the count array to all zeros
    q = np.zeros((env.size), dtype=float)
    n = np.zeros((env.size), dtype=int)

    # some variables to calculate statistics and not necessary
    qe = np.empty((n_episodes, env.size), dtype=float)
    returns = np.empty(n_episodes, dtype=float)
    actions = np.empty(n_episodes, dtype=int)

    # here we enter the main loop and interact with the environment
    for e in range(n_episodes):

        # select the action that maximizes the estimated Q-values
        action = np.argmax(q)

        # then, pass it to the environment and receive a new reward
        reward = env.step(action)

        # update the counts and the q-table
        n[action] += 1
        q[action] = q[action] + (reward - q[action])/n[action]

        # update the statistics and start a new episode
        qe[e] = q
        returns[e] = reward
        actions[e] = action

    return returns, qe, actions
```

```python
def pure_exploration(env, n_episodes=1000):

    # The baseline boilerplate is the same as before
    q = np.zeros((env.size), dtype=float)
    n = np.zeros((env.size), dtype=int)

    qe = np.empty((n_episodes, env.size), dtype=float)
    returns = np.empty(n_episodes, dtype=float)
    actions = np.empty(n_episodes, dtype=int)

    for e in range(n_episodes):
```

```
        # always selects an action randomly
        action = np.random.randint(len(q))

        reward = env.step(action)

        n[action] += 1
        q[action] = q[action] + (reward - q[action])/n[action]

        qe[e] = q
        returns[e] = reward
        actions[e] = action

    return returns, qe, actions
```

1.3.2 Epsilon greedy

We combine the two baselines, pure exploitation and pure exploration, so that the agent can exploit, but also collect information to make informed decisions. The **hybrid strategy** consists of **acting greedily most of the time and exploring randomly every so often**. This strategy is referred as the **epsilon-greedy strategy**.



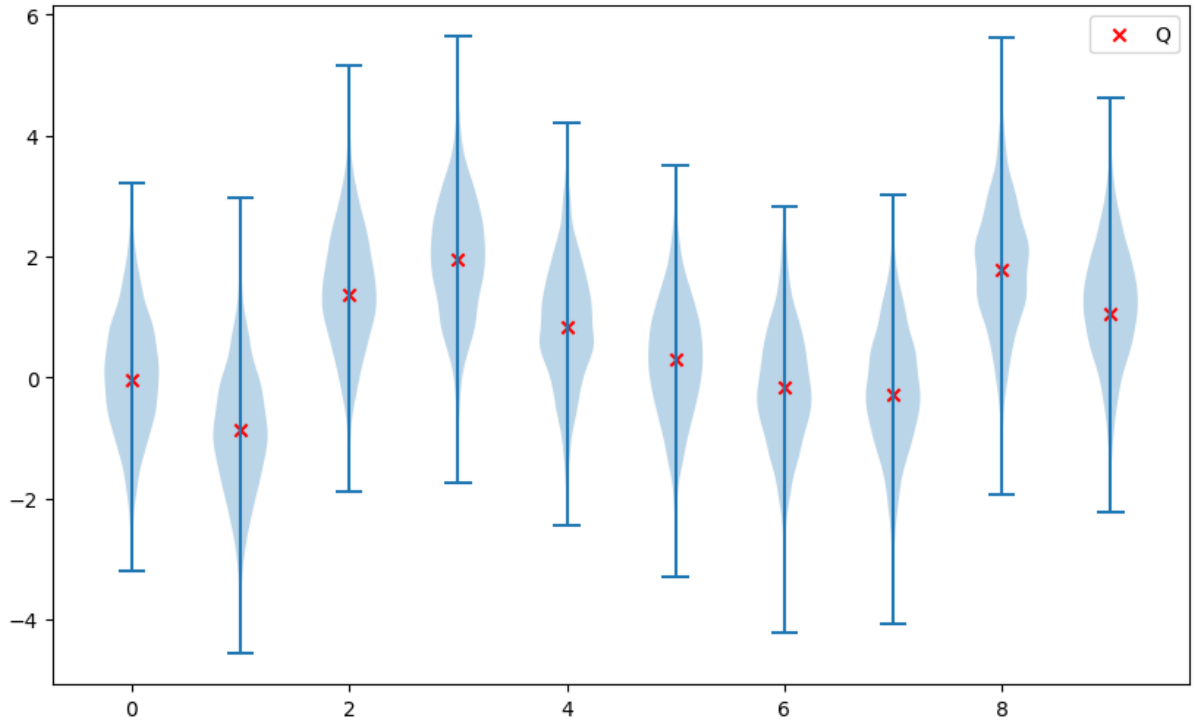

```
def epsilon_greedy(env, epsilon=0.01, n_episodes=1000):

    # The baseline boilerplate is the same as before
    q = np.zeros((env.size), dtype=float)
    n = np.zeros((env.size), dtype=int)

    qe = np.empty((n_episodes, env.size), dtype=float)
    returns = np.empty(n_episodes, dtype=float)
    actions = np.empty(n_episodes, dtype=int)

    for e in range(n_episodes):
```

```
        # draw a random number and compare to a hyperparameter epsilon
        if np.random.uniform() > epsilon:
            # if it is greater than epsilon, exploit:
            action = np.argmax(q)
        else:
            # otherwise, explore:
            action = np.random.randint(len(q))

        reward = env.step(action)

        n[action] += 1
        q[action] = q[action] + (reward - q[action])/n[action]

        qe[e] = q
        returns[e] = reward
        actions[e] = action

    return returns, qe, actions
```

We can plot action values in order to check if they are estimated correctly:

```
_, q, _ = epsilon_greedy(env=env, epsilon=0.5, n_episodes=10000)
```

```
plt.figure(figsize=(10,6))
plt.violinplot(np.random.randn(2000, 10) + env.means, positions=range(env.size))
plt.scatter(range(env.size), q[9999], color='red', marker='x', label='Q')
plt.legend();
```



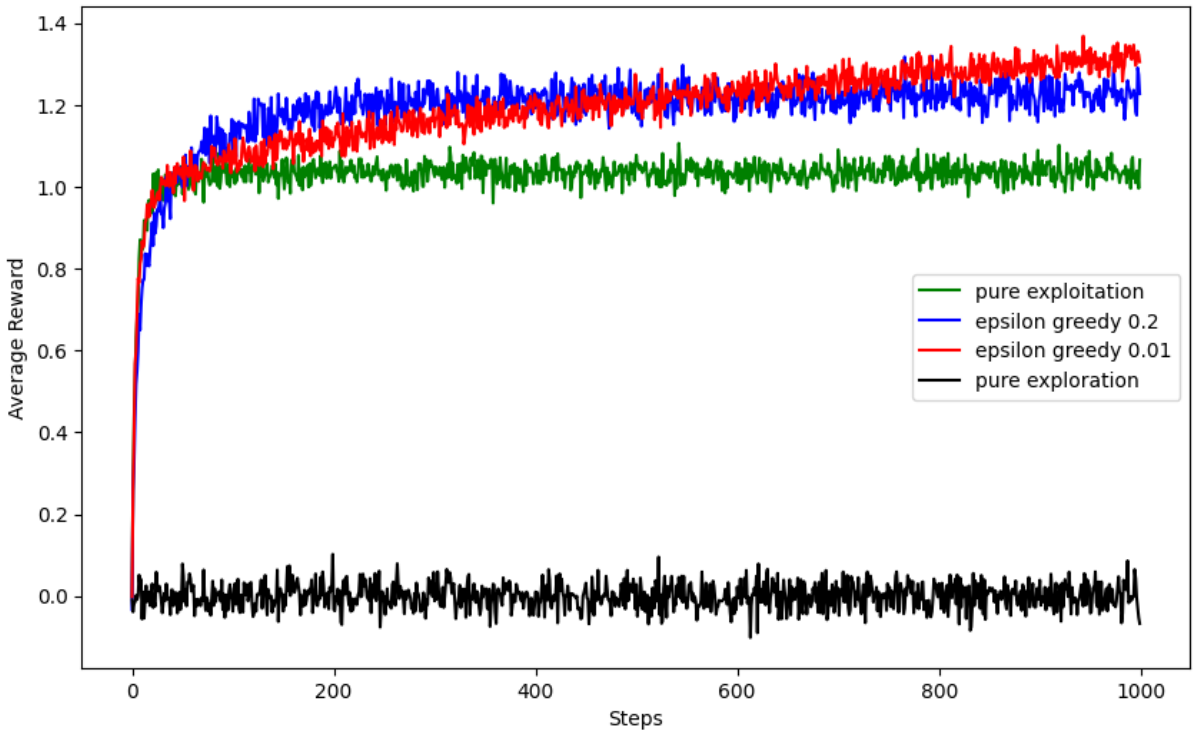
We can measure the performance and the behavior of the method over 1000 time steps when applied to a simulation of the 10-armed bandit problem. Repeating this for 2000 independent runs, each with a different bandit problem, we obtain a measure of the learning algorithm's average behavior. The following code compares methods with different values of epsilon:

```
returns_exploitation = [];  # pure exploitation
returns_epsilon_02 = [];    # epsilon=0.2
returns_epsilon_001 = [];   # epsilon=0.01
returns_exploration = [];   # pure exploration

repetitions = 2000

envs = []
for i in range(repetitions):
    envs.append(BanditEnv());

for i in range(repetitions):
    returns, _, _ = pure_exploitation(env=envs[i], n_episodes=1000);
    returns_exploitation.append(returns);

for i in range(repetitions):
    returns, _, _ = epsilon_greedy(env=envs[i], epsilon=0.2);
```

```
    returns_epsilon_02.append(returns);

for i in range(repetitions):
    returns, _, _ = epsilon_greedy(env=envs[i], epsilon=0.01);
    returns_epsilon_001.append(returns);

for i in range(repetitions):
    returns, _, _ = pure_exploration(env=envs[i]);
    returns_exploration.append(returns);

returns_exploitation = np.array(returns_exploitation);
returns_epsilon_02 = np.array(returns_epsilon_02);
returns_epsilon_001 = np.array(returns_epsilon_001);
returns_exploration = np.array(returns_exploration);
```

```
# Average reward
avg_reward_exploitation = np.average(returns_exploitation, axis=0)
avg_reward_epsilon_02 = np.average(returns_epsilon_02, axis=0)
avg_reward_epsilon_001 = np.average(returns_epsilon_001, axis=0)
avg_reward_exploration = np.average(returns_exploration, axis=0)
```

```
plt.figure(figsize=(10,6))
plt.plot(avg_reward_exploitation, color='green', label='pure exploitation')
plt.plot(avg_reward_epsilon_02, color='blue',  label='epsilon greedy 0.2')
plt.plot(avg_reward_epsilon_001, color='red',   label='epsilon greedy 0.01')
plt.plot(avg_reward_exploration, color='black', label='pure exploration')
plt.xlabel('Steps');
plt.ylabel('Average Reward')
plt.legend()
plt.show()
```



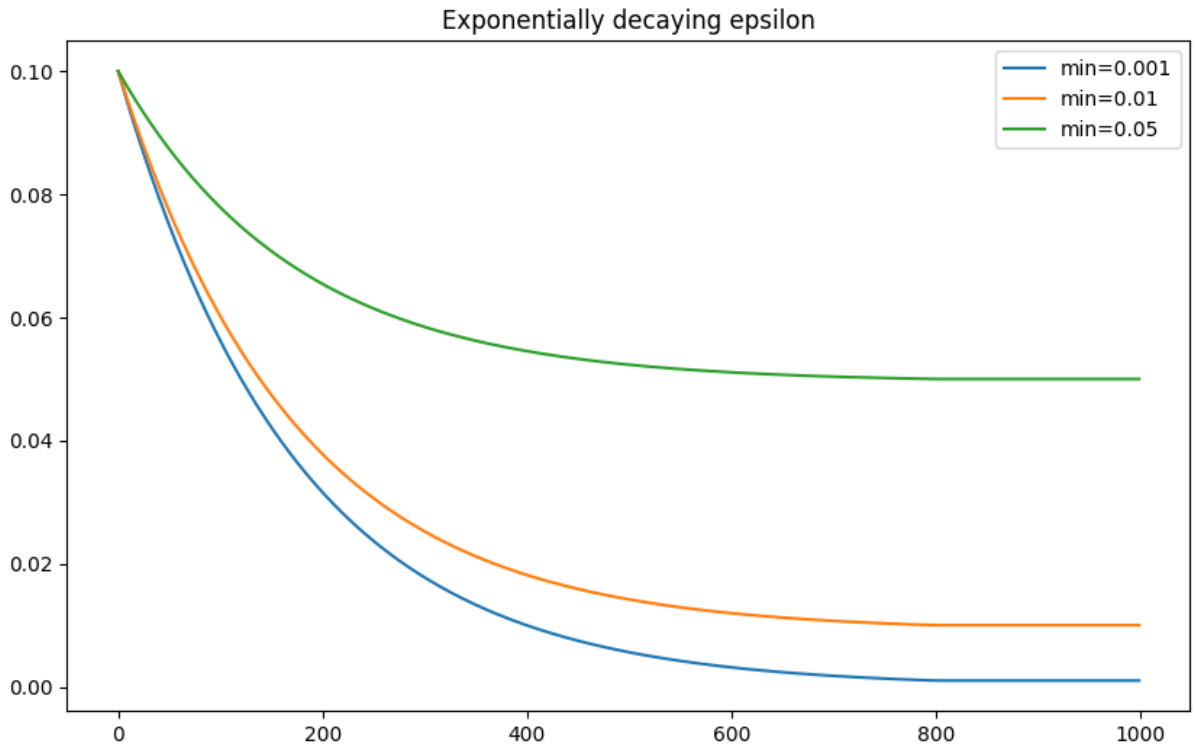
The pure exploitation method improved **slightly faster** than the others at the very beginning, but it performs significantly **worse in the long run** because it got stuck performing suboptimal actions. The greedy methods **performs better** because they **continued to explore** and to improve their chances of recognizing the optimal action. With epsilon=0.2 it exploreds more, and usually **found the optimal action earlier**, but it never selected that action more than 80% of the time, because it explores 20% of the time; with epsilon=0.01 method **improved more slowly** (it explores less), but eventually would perform better in the long run, because it explore only 1% of time. It is possible to **reduce epsilon over time** to try to get the best of both high and low values.

1.3.3 Decaying epsilon-greedy

Intuitively, **early on we can explore the most** (when the agent hasn't experienced the environment enough), **later we want the agent to exploit more and more** (as it obtains better estimates of the value functions). In the decaying method we start with a high epsilon and then we decay its value on every step. This strategy can take many forms depending on **how we change the value of epsilon** (linearly or exponentially). Notice that we can calculate all of the epsilon values at once, and only query an array of pre-computed values in the loop. The following is an exponential implementation

```python
def decay_epsilon(init_value, min_value, decay_episodes, max_episodes):

    # calculate the number of the remaining episodes after the decay
    rem_episodes = max_episodes - decay_episodes

    # logspace returns numbers spaced evenly on a log scale
    # base^start is the starting value of the sequence,
    # base^stop is the final value of the sequence
    # num is the number of values to generate
    # base is the base of the log space
    values = np.logspace(start=0, stop=-2, num=decay_episodes, base=10)

    # because the values may not end exactly at 0, given it's the log,
    # we change them to be between 0 and 1 so that the curve looks smooth and nice.
    values = (values - values.min()) / (values.max() - values.min())

    # linear transformation and get points between init_value and min_value
    values = (init_value - min_value) * values + min_value

    # repeats the rightmost value rem_step number of times
    values = np.pad(values, (0, rem_episodes), 'edge')

    return values
```

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(10,6))
plt.plot(decay_epsilon(0.1, 0.001, 800, 1000), label='min=0.001')
plt.plot(decay_epsilon(0.1, 0.01, 800, 1000), label='min=0.01')
plt.plot(decay_epsilon(0.1, 0.05, 800, 1000), label='min=0.05')
plt.title('Exponentially decaying epsilon')
plt.legend()
plt.show()
```

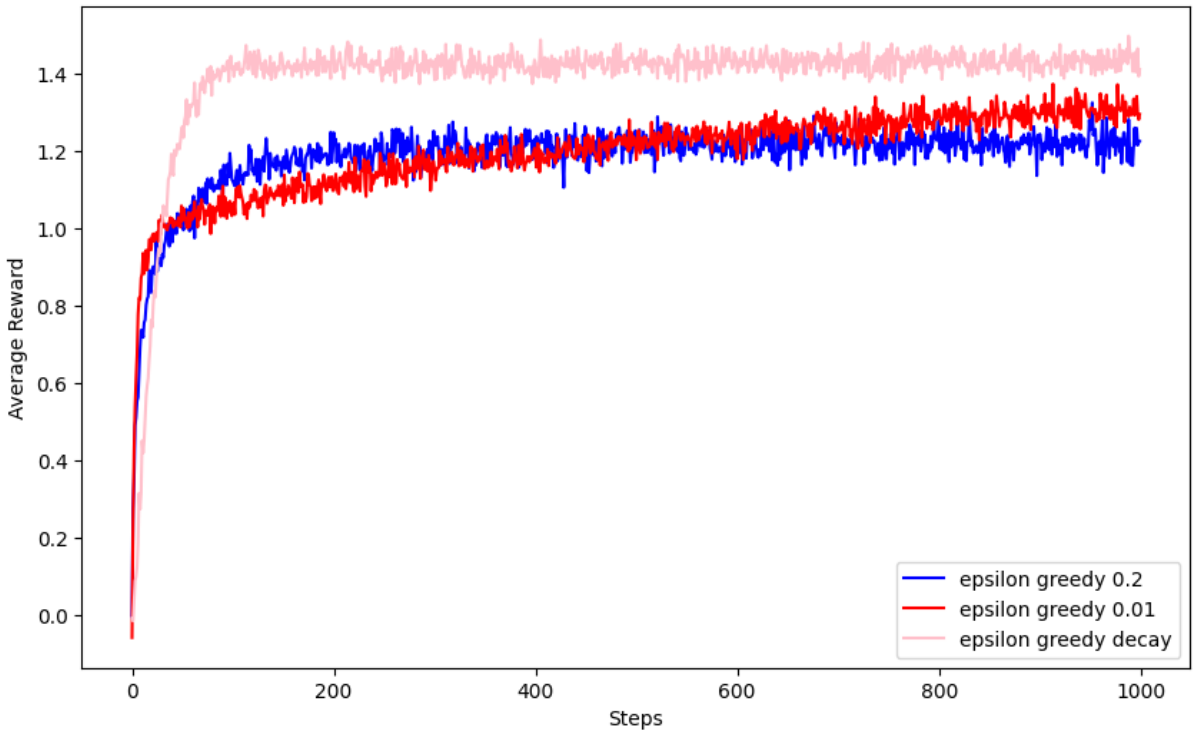



```python
def decay_epsilon_greedy(env, init_epsilon=1, min_epsilon=0.001, decay_episodes=100,
↪  n_episodes=1000):

    # The baseline boilerplate is the same as before
    q = np.zeros((env.size), dtype=float)
    n = np.zeros((env.size), dtype=int)

    qe = np.empty((n_episodes, env.size), dtype=float)
    returns = np.empty(n_episodes, dtype=float)
    actions = np.empty(n_episodes, dtype=int)

    # Calculate the exponentially decaying epsilons.
    epsilons = decay_epsilon(init_epsilon, min_epsilon, decay_episodes, n_episodes);

    for e in range(n_episodes):

        if np.random.uniform() > epsilons[e]:
            action = np.argmax(q)
        else:
            action = np.random.randint(len(q))

        reward = env.step(action)
```

```
        n[action] += 1
        q[action] = q[action] + (reward - q[action])/n[action]

        qe[e] = q
        returns[e] = reward
        actions[e] = action

    return returns, qe, actions
```

```
returns_epsilon_decay = [];

for i in range(repetitions):
    returns, _, _ = decay_epsilon_greedy(env=envs[i]);
    returns_epsilon_decay.append(returns);

returns_epsilon_decay = np.array(returns_epsilon_decay);
avg_reward_epsilon_decay = np.average(returns_epsilon_decay, axis=0);
```

```
plt.figure(figsize=(10,6))
plt.plot(avg_reward_epsilon_02, color='blue',  label='epsilon greedy 0.2')
plt.plot(avg_reward_epsilon_001, color='red',   label='epsilon greedy 0.01')
plt.plot(avg_reward_epsilon_decay, color='pink',   label='epsilon greedy decay')
plt.xlabel('Steps');
plt.ylabel('Average Reward')
plt.legend()
plt.show()
```



The advantage of the decaying approach is evident in the graph, it quickly finds optimal actions and stabilizes at a higher average reward compared to the fixed epsilon strategies. The reward curve for this strategy is smoother and consistently higher than the fixed epsilon strategies, indicating that it **effectively balances exploration and exploitation over time**. There are many other ways we can handle the decaying of epsilon. The bottom line is that **the agent should explore with a higher chance early and exploit with a higher chance later**. Early on, there's a high likelihood that value estimates are wrong. Still, as time passes and he acquires knowledge, the likelihood that the value estimates are close to the actual values increases, which is when the agent should explore less frequently so that he can exploit the knowledge acquired.

1.3.4 Optimistic initial values

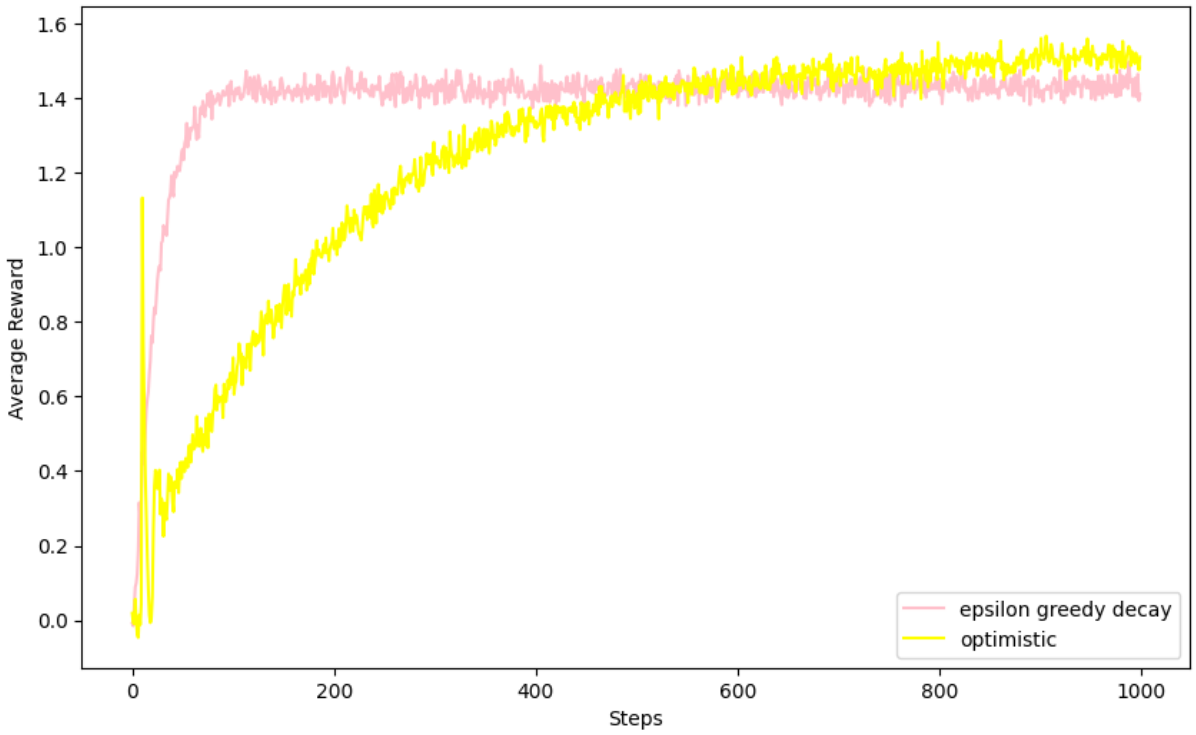
Initial action values can be used as a simple way to **encourage exploration**. Suppose to set an initial estimate in a **optimistic way**. Whichever actions are initially selected, the reward is less than the starting estimates and the learner switches to other actions, being **"disappointed"** with the rewards it is receiving. The result is that **all actions are tried several times** before the value estimates converge. The system does a fair amount of exploration even if greedy actions are selected all the time. We call this technique for encouraging exploration **optimistic initial values**. The mechanic of the strategy is straightforward: we initialize the action-value function to a high value and act greedily using these estimates:

```
def optimistic_initialization(env, optimistic_estimate=5.0, initial_count=10,
 ↪  n_episodes=1000):

    # start initializing the action-values to an optimistic value
    q = np.full((env.size), optimistic_estimate, dtype=float)

    # initialize the counts that will serve as an uncertainty measure
    # the higher the more certain.
    n = np.full((env.size), initial_count, dtype=int)

    qe = np.empty((n_episodes, env.size), dtype=float)
    returns = np.empty(n_episodes, dtype=float)
    actions = np.empty(n_episodes, dtype=int)

    for e in range(n_episodes):

        action = np.argmax(q)

        reward = env.step(action)

        n[action] += 1
        q[action] = q[action] + (reward - q[action])/n[action]

        qe[e] = q
        returns[e] = reward
        actions[e] = action

    return returns, qe, actions
```

```
returns_optimistic = [];

for i in range(repetitions):
    returns, _, _ = optimistic_initialization(env=envs[i])
    returns_optimistic.append(returns);

returns_optimistic = np.array(returns_optimistic)

avg_reward_optimistic = np.average(returns_optimistic, axis=0)
```

```
plt.figure(figsize=(10,6))
plt.plot(avg_reward_epsilon_decay, color='pink',   label='epsilon greedy decay')
```

```
plt.plot(avg_reward_optimistic, color='yellow',   label='optimistic')
plt.xlabel('Steps');
plt.ylabel('Average Reward')
plt.legend()
plt.show()
```



Initially, the optimistic method performs worse because it explores more. In the early part of the curve for the optimistic method we can see **oscillations and spikes**. If the initial action selected are by chance ones of the better choices then the action value estimates for these plays will be magnified resulting in an emphasis to continue playing this action. This results in large actions values being received on the initial draws and consequently very good initial play. In the same way, if the algorithm initially selects poor plays then initially the algorithm will perform poorly resulting in very poor initial play. We regard it as a **simple trick** that can be **quite effective on stationary problems**, but it is not well suited to non-stationary problems because its drive for exploration is inherently temporary.

1.4 Strategic Exploration

While humans explore, they don't explore randomly. Maybe imprecision is the source of randomness, but humans have a **strategic way of exploring**. We know that we're sacrificing short-

term for long-term satisfaction. **We know we want to acquire more information**. We explore by trying things **we haven't sufficiently tried**, but **have the potential to better our lives**. Perhaps, our exploration strategies are a combination of estimates and their uncertainty. For instance, we might prefer a dish that we're likely to enjoy, and we haven't tried, over a dish that we like okay, but we get every weekend. Perhaps we explore **based on our curiosity or our prediction error**. For instance, we might be more inclined to try new dishes at a restaurant that we thought would be okay-tasting food, but it resulted in the best food you ever had. That **prediction error** and that **surprise** could be our metric for exploration. We can consider more advanced exploration strategies that apply randomness in proportion to the current estimates of the actions or that take into account the confidence and uncertainty levels of the estimates. The epsilon-greedy strategy (and its decaying versions) is still the most popular exploration strategy in use today, perhaps because it per forms well, perhaps because of its simplicity. Balancing the exploration versus exploitation trade-off, the gathering and utilization of information is central to human intelligence, artificial intelligence, and reinforcement learning.

1.4.1 Softmax method

Exploration makes more sense if it takes into account action-value estimates: **if there is an action that has a really low estimate, we're less likely to try it**. The Softmax strategy **samples actions from a probability distribution over the action-value estimates**, such that the probability of selecting an action is proportional to its current action-value estimate. The difference between estimates will create a tendency to select actions with the highest estimates more often, and actions with the lowest estimates less frequently. We can also add a **hyperparameter to control the algorithm's sensitivity** to the differences in estimates. That hyperparameter, called the **temperature**, works in such a way that as it approaches infinity, the preferences over the action-values are equal (it samples uniformly), and as it approaches zero, the action with the highest estimated value will be sampled with probability of one. Also, we can decay this hyperparameter either linearly, exponentially, or another way.

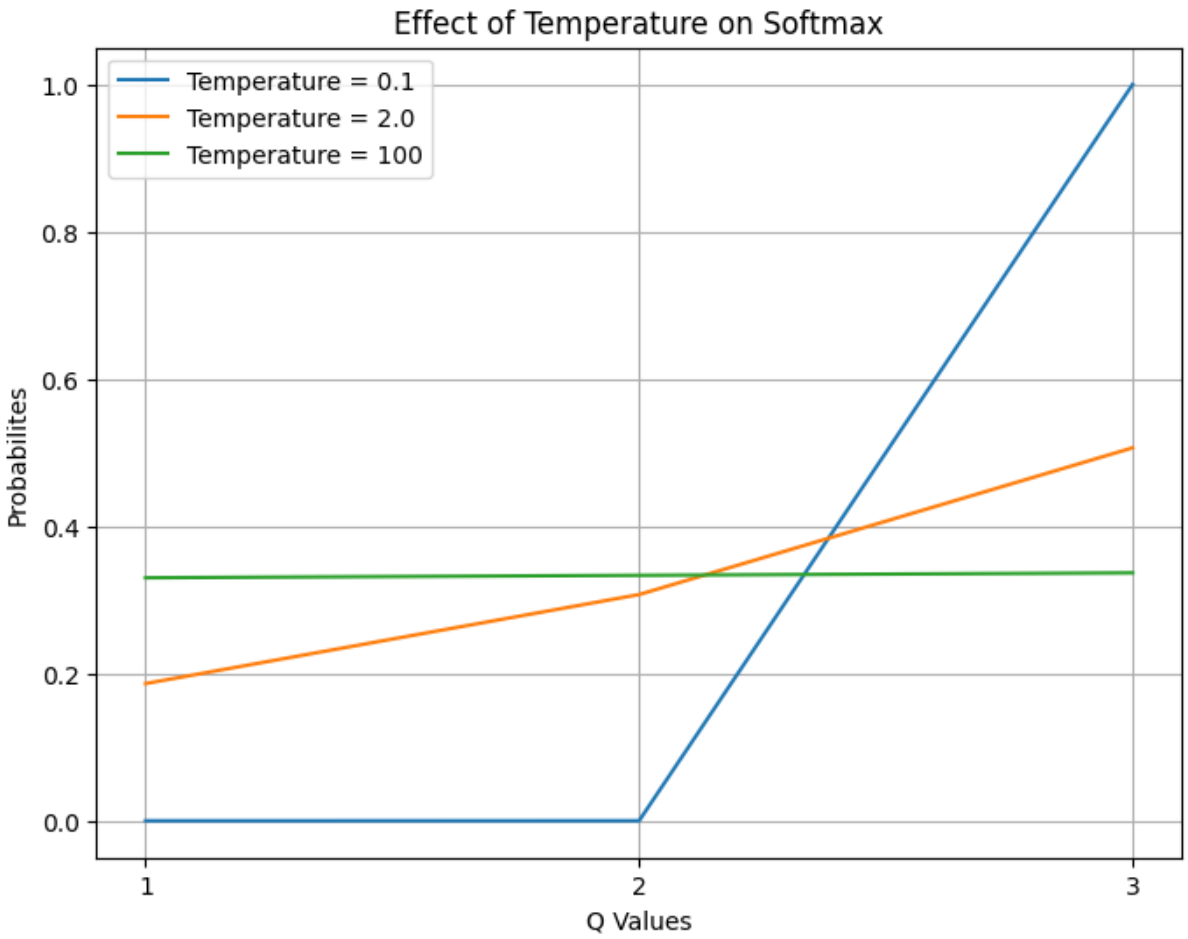
$$\Pr(a) = \frac{e^{\frac{q(a)}{\tau}}}{\sum_{b=1}^{B} e^{\frac{q(b)}{\tau}}}$$

```
def softmax_function(q, temperature=1.0):
    exp = np.exp(q / temperature);
    return exp / exp.sum(axis=0);
```

In order to show the temperature effect (lower temperatures make outputs closer to 1 or 0, while

higher temperatures spread out the probabilities more evenly), we're plotting the softmax output for different temperatures to show how this affects the distribution of the probabilities:

```
# define some input values
q = np.array([1.0, 2.0, 3.0])

# define a range of temperature values
temperatures = [0.1, 2.0, 100]

# plot the softmax output for different temperatures
plt.figure(figsize=(8, 6))

for temp in temperatures:
    probabilits = softmax_function(q, temperature=temp)
    plt.plot(q, probabilits, label=f'Temperature = {temp}')

# Plot settings
plt.title('Effect of Temperature on Softmax')
plt.xlabel('Q Values')
plt.ylabel('Probabilites')
plt.xticks(q)
plt.legend()
plt.grid(True)
plt.show()
```



Now we can write the exploitation-exploration trade-off usfine the softmax method:

```
def softmax(env, init_temp=1000, min_temp=0.1,
            decay_ratio=0.04,n_episodes=1000):

    q = np.zeros((env.size), dtype=float)
    n = np.zeros((env.size), dtype=int)

    qe = np.empty((n_episodes, env.size), dtype=float)
    returns = np.empty(n_episodes, dtype=float)
    actions = np.empty(n_episodes, dtype=int)

    for e in range(n_episodes):

        # calculate the linearly decaying temperature
        decay_episodes = n_episodes * decay_ratio
        temp = 1 - e / decay_episodes
```

```
        temp *= init_temp - min_temp
        temp += min_temp
        temp = np.clip(temp, min_temp, init_temp)

        # calculate the probabilities by applying the softmax function
        # to the estimates
        scaled_q = q / temp
        norm_q = scaled_q - np.max(scaled_q)
        exp_q = np.exp(norm_q)
        probs = exp_q / np.sum(exp_q)

        # select the action based on probabilities
        action = np.random.choice(np.arange(len(probs)), size=1, p=probs)[0]

        reward = env.step(action)

        n[action] += 1
        q[action] = q[action] + (reward - q[action])/n[action]

        qe[e] = q
        returns[e] = reward
        actions[e] = action

    return returns, qe, actions
```

1.4.2 Upper Confidence Bound method

The optimistic initialization strategy is a clever and efficient approach, however, it has two significant inconveniences. First, **we don't know the maximum reward the agent can obtain**. If we set the initial optimistic estimates to a value much higher than its actual value, the algorithm will perform sub-optimally for a long time, because the agent will take many episodes to bring the estimates near the actual values. Even worse, if we set the initial values lower than the maximum, the algorithm will no longer be optimistic, and it will no longer work. The second issue is that **the counts variable is a hyperparameter and it needs tuning**, but what we're trying to represent with this variable **is the uncertainty of the estimate**, which shouldn't be a hyperparameter. A better strategy, that follows the same principles of optimistic initialization, is the **upper confidence bound** (**UCB**) which use statistical techniques to calculate the **value estimates uncertainty** and uses that as a bonus for exploration. In UCB, we're still optimistic, but it's **a more a realistic optimism**. Instead of blindly hoping for the best, we look at the uncertainty of value estimates. **The more uncertain an estimate, the more critical it is to explore it**.

Notice that it's no longer about believing the value will be the "maximum possible", though it might be! The new metric that we care about here is uncertainty, we want **to give uncertainty the benefit of the doubt**. To select the action we add the estimates and an uncertainty bonus:

$$A_t = \underset{a}{\text{argmax}} \left[q_t(a) + c * u_t(a) \right]$$

whre q(a) **represents the value** term and u(a) is a term that **captures the uncertainty**, which could depend on factors like the number of times action has been selected. This formula reflects the general idea that we decide for a action with a combination of its value and an additional boost based on how uncertain the estimate of that value i. The uncertainty term can be expressed as:

$$u_t(a) = \sqrt{\left[\frac{\ln t}{N_t(a)} \right]}]$$

The idea is that each time an action is selected the uncertainty is presumably reduced (the count increments and the uncertainty term decreases). On the other hand, each time another action is selected, t increases but the count does not and the uncertainty estimate increases. The use of the logarithm means that the increases get smaller over time, but are unbounded. Composing the two contribution:

$$A_t = \underset{a}{\text{argmax}} \left[q_t(a) + c \sqrt{\left[\frac{\ln t}{N_t(a)} \right]} \right]$$

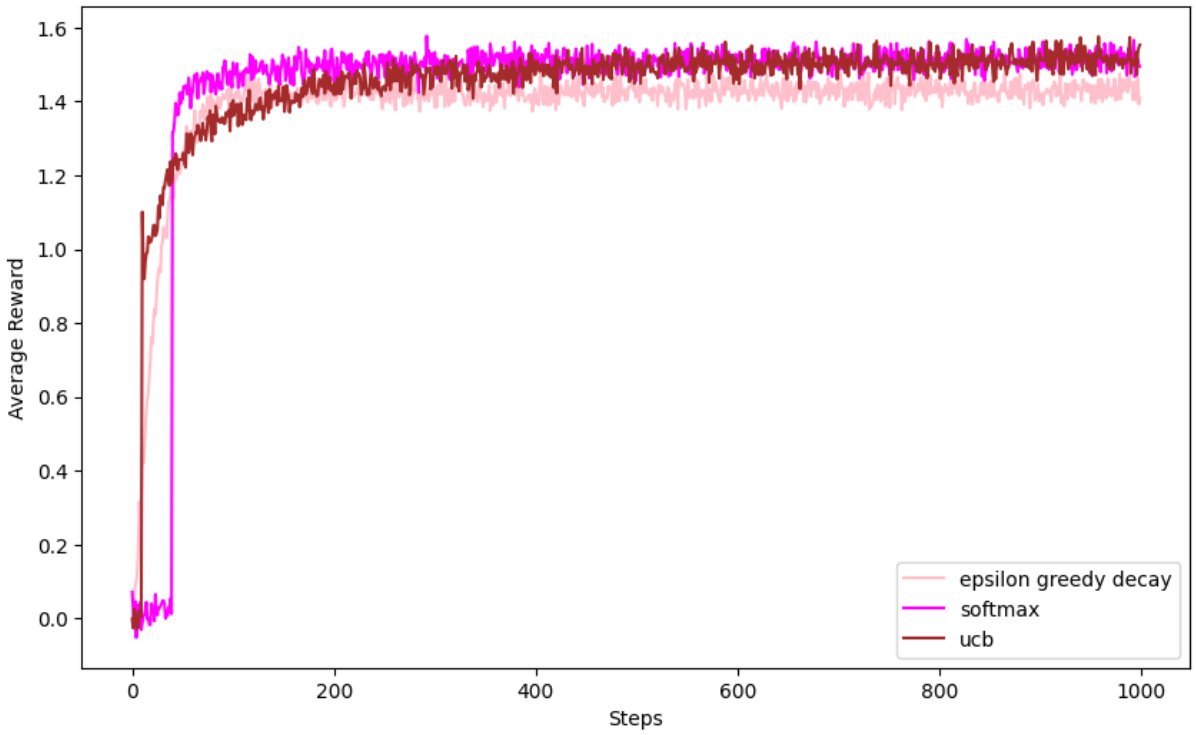
All actions will eventually be selected, but actions with lower value estimates or that have already been selected frequently, will be selected with decreasing frequency over time. UCB often performs well but is more difficult than epsilon-greedy to extend beyond bandits to the more general reinforcement learning settings.

```python
def upper_confidence_bound(env, c=2, n_episodes=1000):

    q = np.zeros((env.size), dtype=float)
    n = np.zeros((env.size), dtype=int)

    qe = np.empty((n_episodes, env.size), dtype=float)
    returns = np.empty(n_episodes, dtype=float)
    actions = np.empty(n_episodes, dtype=int)

    for e in range(n_episodes):
        # first select all actions once to avoid division by zero
        if e < len(q):
            action = e
        else:
            # proceed to calculating the confidence bounds
```

```
            u = np.sqrt(c * np.log(e)/n)
            # pick the action with the highest value with an
            # uncertainty bonus: the more uncertain the value of the action,
            # the higher the bonus
            action = np.argmax(q + u)

        reward = env.step(action)

        n[action] += 1
        q[action] = q[action] + (reward - q[action])/n[action]

        qe[e] = q
        returns[e] = reward
        actions[e] = action

    return returns, qe, actions
```

```
returns_softmax = [];
returns_ucb = [];

for i in range(repetitions):
    returns, _, _ = softmax(env=envs[i]);
    returns_softmax.append(returns);

for i in range(repetitions):
    returns, _, _ = upper_confidence_bound(env=envs[i]);
    returns_ucb.append(returns);

returns_softmax = np.array(returns_softmax);
returns_ucb = np.array(returns_ucb);

avg_reward_softmax = np.average(returns_softmax, axis=0)
avg_reward_ucb = np.average(returns_ucb, axis=0)
```

```
plt.figure(figsize=(10,6))
plt.plot(avg_reward_epsilon_decay, color='pink',   label='epsilon greedy decay')
plt.plot(avg_reward_softmax, color='magenta',   label='softmax')
plt.plot(avg_reward_ucb, color='brown',   label='ucb')
plt.xlabel('Steps');
plt.ylabel('Average Reward')
plt.legend()
plt.show()
```



Softmax exploration selects actions probabilistically based on their estimated value, favoring better actions but still allowing exploration. UCB balances exploration and exploitation by choosing actions based on their estimated reward and uncertainty. Both algorithms quickly reach a high reward and maintain strong performance over time. They performs better than the epsilon-greedy decay, however they are more complex and require more tuning.

1.5 Comparison

We have presented several ways of balancing exploration and exploitation. The epsilon-greedy methods choose randomly a small fraction of the time, UCB methods choose deterministically, but achieve exploration by subtly favoring at each step the actions that have so far received fewer samples. We can run simulations for all methods in order to compare their performances. A complication is that they **all have parameters**. To get a meaningful comparison we have to consider their performance as a function of their parameter. We summarize a complete learning curve by its average value over the 1000 steps and shows this measure for the various algorithms, each as a function of its own parameter shown on a single scale on the x-axis. This kind of graph is called a **parameter study**:

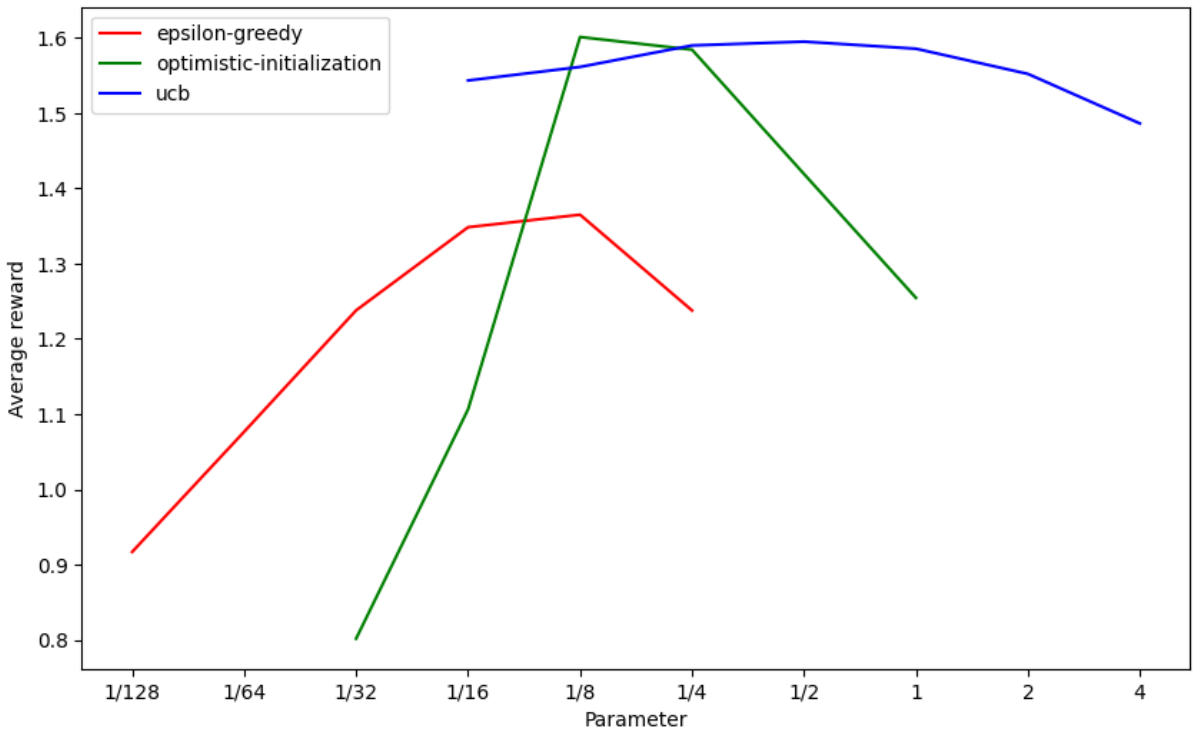
```
def run_experiment(envs, algorithm, param):
    returns = [];

    for i in range(2000):
        if algorithm == 'epsilon-greedy':
            r, _, _ = epsilon_greedy(env, epsilon=param, n_episodes=1000);
        elif algorithm == 'optimistic-initialization':
            r,_ ,_ = optimistic_initialization(env, optimistic_estimate=param,
 ↪  initial_count=10, n_episodes=1000);
        elif algorithm == 'softmax':
            r,_ ,_ = softmax(env, init_temp=param, min_temp=0.1,
 ↪  decay_ratio=0.04,n_episodes=1000);
        elif algorithm == 'ucb':
            r,_ ,_ = upper_confidence_bound(env, c=param, n_episodes=1000);
        else:
            raise ValueError('Unknown algorithm:', algorithm)
        returns.append(r);
    returns = np.array(returns);
    return returns.mean()
```

```
egreedy_x, egreedy_y = [], []
for param in [1/128, 1/64, 1/32, 1/16, 1/8, 1/4]:
    result = run_experiment(envs=envs, algorithm='epsilon-greedy', param=param);
    egreedy_x.append(param)
    egreedy_y.append(result)
```

```
opt_x, opt_y = [], []
for param in [1/4, 1/2, 1, 2, 4, 6]:
    result = run_experiment(envs=envs, algorithm='optimistic-initialization',
 ↪  param=param)
    opt_x.append(param)
    opt_y.append(result)
```

```
ucb_x, ucb_y = [], []
for param in [1/16, 1/8, 1/4, 1/2, 1, 2, 4]:
    result = run_experiment(envs=envs, algorithm='ucb', param=param)
    ucb_x.append(param)
    ucb_y.append(result)
```

```
plt.figure(figsize=(10,6))
plt.xticks(range(-7, 3), ('1/128', '1/64', '1/32', '1/16', '1/8', '1/4', '1/2', '1',
↪  '2', '4'))
plt.plot(range(-7, -1), egreedy_y, color='red', label='epsilon-greedy')
plt.plot(range(-5, 1), opt_y, color='green', label='optimistic-initialization')
plt.plot(range(-4, 3), ucb_y, color='blue', label='ucb')
plt.xlabel('Parameter')
plt.ylabel('Average reward')
plt.legend()
plt.show()
```



Overall, on this problem, **UCB seems to perform best**. Notice the characteristic **inverted-U shapes** of each algorithm's performance: all the algorithms perform best at an **intermediate value of their parameter**, neither too large nor too small. In assessing a method, we should attend not just to how well it does at its best parameter setting, but also to **how sensitive it is to its parameter value**. All of these algorithms are fairly insensitive, performing well over a range of parameter values varying by about an order of magnitude.