Classic Exploration Strategies

Christopher Mutschler

Agenda

- Motivation, Problem Definition & Multi-Armed Bandits
- **Classic Exploration Strategies**
  - Epsilon Greedy
  - (Bayesian) Upper Confidence Bounds
  - Thomson Sampling
- Exploration in Deep RL:
  - Count-based Exploration: Density Models, Hashing
  - Prediction-based Exploration:
    - Forward Dynamics
    - Random Networks
    - Physical Properties
  - Memory-based Exploration:
    - Episodic Memory
    - Direct Exploration
- Summary and Outlook

Random Exploration: ϵ-greedy

is this enough?

- Exploration at random: ϵ-greedy
- Recap & let's formulate:
  - Take the best action most of the time, but do random exploration occasionally
  - Action-values are estimated according to the past experience (by averaging rewards associated with the action up to time step T):

$$\hat{Q}_T(a) = \frac{1}{N_T(a)} \sum_{t=1}^{T} r_t \mathbb{I}[a_t = a],$$

  - where $\mathbb{I}$ is a binary indicator function and $N_t(a)$ is the action selection counter, i.e.:

$$N_t(a) = \sum_{t=1}^{T} \mathbb{I}[a_t = a].$$

  - With a small probability of ϵ we take a random action (explore) and with probability of $1 - \epsilon$ we pick the best action that we have learnt so far (exploit):

$$a_T^* = \arg\max_{a \in \mathcal{A}} \hat{Q}_T(a).$$

How to pick ϵ?

Random Exploration: ϵ-greedy

- Greedy may select a suboptimal action forever
  → Greedy has hence linear expected total regret

- ϵ-greedy continues to explore forever
  - with probability $1 - \epsilon$ it selects $a = \arg\max_{a \in \mathcal{A}} Q_T(a)$
  - with probability ϵ it selects a random action

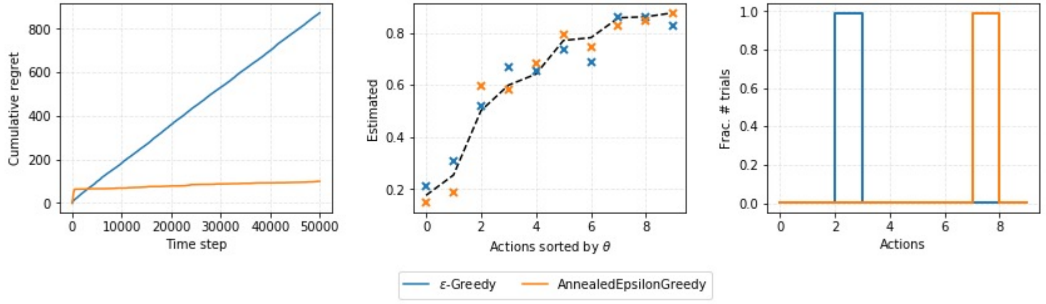
- Will hence continue to select all suboptimal actions with (at least) a probability of $\frac{\epsilon}{|\mathcal{A}|}$
  → ϵ-greedy, with a constant ϵ has a linear expected total regret

Random Exploration: ϵ-greedy (Demo)

In [1]:
```
import matplotlib  # noqa
#matplotlib.use('Agg')  # noqa

import matplotlib.pyplot as plt
import numpy as np
import time
from scipy.stats import beta
```

BernoulliBandit class

In [2]:
```
class BernoulliBandit(object):
    def __init__(self, n, probas=None):
        assert probas is None or len(probas) == n
        self.n = n
        if probas is None:
            np.random.seed(int(time.time()))
            self.probas = [np.random.random() for _ in range(self.n)]
        else:
            self.probas = probas

        self.best_proba = max(self.probas)

    def generate_reward(self, i):
        # The player selected the i-th machine.
        if np.random.random() < self.probas[i]:
            return 1
        else:
            return 0
```



In [ ]:

Random Exploration: ϵ-greedy

- Random exploration allows us to try out option that we have not much knowledge about yet
  - However: due to randomness, we end up exploring bad actions all over again!
  - What to do about it?

- **Option #1: decrease ϵ over course of training might work**
  - We saw in the demo that this helps
  - However, it is not easy to tune the parameters

- **Option #2: be optimistic with options of high uncertainty**
  - Prefer actions for which you do not have a confident value estimation yet
    → Those have a great potential to be high-rewarding!
  - This idea is called **Upper Confidence Bounds**

Upper Confidence Bounds

- Idea: estimate an upper confidence $U_t(a)$ for each action value, such that with a high probability we satisfy

$$Q(a) \leq \hat{Q}_t(a) + U_t(a)$$

- Next, we select the action that maximizes the upper confidence bound:

$$a_t^{UCB} = \arg\max_{a \in \mathcal{A}} [Q_t(a) + U_t(a)]$$

- The upper bound $U_t(a)$ is a function of the number of trials $N_t(a)$:
  - Small $N_t(a)$ → large bound $U_t(a)$ *(estimated value is uncertain)*
  - Large $N_t(a)$ → small bound $U_t(a)$ *(estimated value is certain/accurate)*
  - Central limit theorem[1]: the uncertainty decreases as $\sqrt{N_t(a)}$
    (as long as the variance of rewards is bounded)

→ How can we efficiently estimate the upper confidence bound?

[1] *https://en.wikipedia.org/wiki/Central_limit_theorem*

Upper Confidence Bounds

- Wait, let's put all the sidenotes on a single slide first:
  - We want to minimize $\sum_a N_t(a)\Delta_a$
  - If Δ_a is big → we want $N_t(a)$ to be small
  - If $N_t(a)$ is big → we want Δ_a to be small

  - Not all $N_t(a)$ can be small: their sum is (exactly) t
  - We know $N_t(a)$
  - We do not know Δ_a - *but what what can we learn about it?*

Theorem: Hoeffding's Inequality

- Let $X_1, \dots, X_n$ be i.i.d. random variables whose value are in $[0,1]$
- Let $\bar{X}_T = \frac{1}{t}\sum_{t=1}^{T} X_t$ be the sample mean
- Then (for any $u > 0$):

$$P(\mathbb{E}[X] \geq \bar{X}_t + u) \leq e^{-2tu^2}$$

- *Example: How likely is it to achieve an eye sum of at least 500 when rolling a dice for a hundred times?*
  - X is a random variable that describes the result of a roll, its mean is $\mathbb{E}[X] = 3{,}5$
    → $-2{,}5 \leq X - \mathbb{E}[X] \leq 2{,}5$
  - Hoeffding's Inequality:

$$P\left[\sum X \geq 500\right] = P\left[\sum (X - \mathbb{E}[X]) \geq 150\right] \leq e^{\frac{-2\cdot 150^2}{\sum(2{,}5+2{,}5)^2}} = e^{\frac{-45000}{100\cdot 25}} = e^{-18} \approx 1{,}523 \cdot 10^{-8}$$

see also https://en.wikipedia.org/wiki/Hoeffding%27s_inequality

Upper Confidence Bounds

- Let us apply Hoeffding's Inequality to bandits with bounded rewards
- Given one target action a, let us consider
  - $r_t(a)$ as the random variables
  - $Q(a)$ as the true mean
  - $\hat{Q}_t(a)$ as the sample mean
  - u as the upper bound confidence bound, $u = U_t(a)$
- From this follows:

$$P\left[Q(a) > \hat{Q}_t(a) + U_t(a)\right] \leq e^{-2tU_t(a)^2}$$

- We now want to pick a bound $U_t(a)$ so that with high chances the true mean lies below the sample mean + the upper confidence bound
  → $e^{-2tU_t(a)^2}$ should be a small probability
- Given a tiny threshold p and solve for $U_t(a)$:

$$e^{-2tU_t(a)^2} = p \;\; \rightarrow U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

Upper Confidence Bounds: one more thing

- **With collecting more and more samples, we will get more confident!**
- Let us now do a tiny little tweak: reduce p as we observe more rewards:
  - For instance: $p = \frac{1}{t}$

$$U_t(a) = \sqrt{\frac{\log t}{2N_t(a)}}$$

  - This ensures that we always keep exploring
  - But we select the optimal action much more often as $t \to \infty$

- The vanilla **UCB1** algorithm uses $p = t^{-4}$:

$$U_t(a) = \sqrt{\frac{2 \log t}{N_t(a)}} \quad \text{and} \quad a_t^{UCB} = \arg\max_{a \in \mathcal{A}} Q(a) + \sqrt{\frac{2 \log t}{N_t(a)}}$$

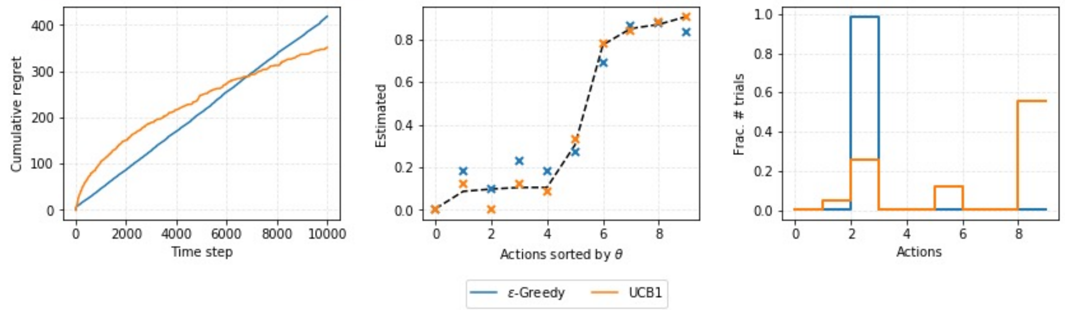
  - However, we could insert any hyper parameter c (here) to adjust this

→ UCB (with $c = \sqrt{2}$) has a logarithmic expected total regret

Upper Confidence Bounds: UCB1 (demo)

```python
class UCB1(Solver):
    def __init__(self, bandit, init_proba=1.0):
        super(UCB1, self).__init__(bandit)
        self.t = 0
        self.estimates = [init_proba] * self.bandit.n

    @property
    def estimated_probas(self):
        return self.estimates

    def run_one_step(self):
        self.t += 1

        # Pick the best one with consideration of upper confidence bounds.
        i = max(range(self.bandit.n), key=lambda x: self.estimates[x] + np.sqrt(
            2 * np.log(self.t) / (1 + self.counts[x])))
        r = self.bandit.generate_reward(i)

        self.estimates[i] += 1. / (self.counts[i] + 1) * (r - self.estimates[i])

        return i
```

```
/Users/mut/workspace/anaconda3/envs/py36/lib/python3.6/site-packages/ipykernel_launcher.py:44: MatplotlibDeprecation
Warning: Passing the drawstyle with the linestyle as a single string is deprecated since Matplotlib 3.1 and support
will be removed in 3.3; please pass the drawstyle separately using the drawstyle keyword argument to Line2D or set_d
rawstyle() method (or ds/set_ds()).
```



Extension: Bayesian UCB

- In UCB we did not assume any prior on the reward distribution
  - Hence, from Hoeffding's Inequality follows a relatively pessimistic bound
- Idea: prior knowledge on the distribution allows for a better bound!
- Example:
  - We expect the mean reward of the slot machines to follow (independent) Gaussians
  - We may set the upper bound to the 95% confidence interval by setting $\widehat{U}_t(a)$ to be twice the standard deviation
- Use the posterior to guide exploration!
  - UCB
  - Thompson Sampling (probability matching)

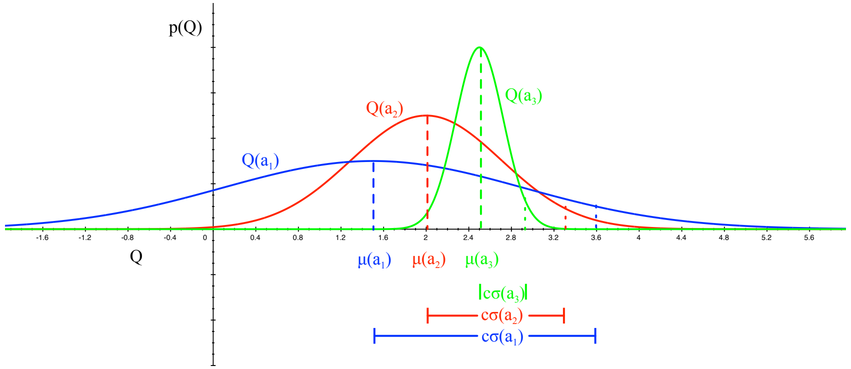


Image taken from UCL Course by David Silver – Lecture 9: XX.

Extension: Bayesian UCB

Example

- We again consider a Bernoulli distribution: rewards are either 0 or +1
- Prior: uniform on $[0,1]\ \forall a \in \mathcal{A}$ (each mean reward is equally likely)
- The posterior is a Beta distribution $\mathrm{Beta}(\alpha_a, \beta_a)$ with initial parameters $\alpha_a = 1$ and $\beta_a = 1$ for each action a
- Update the posterior:
  - $\alpha_{a_t} \leftarrow \alpha_{a_t} + 1$, if $r_t = 0$
  - $\beta_{a_t} \leftarrow \beta_{a_t} + 1$, if $r_t = 1$

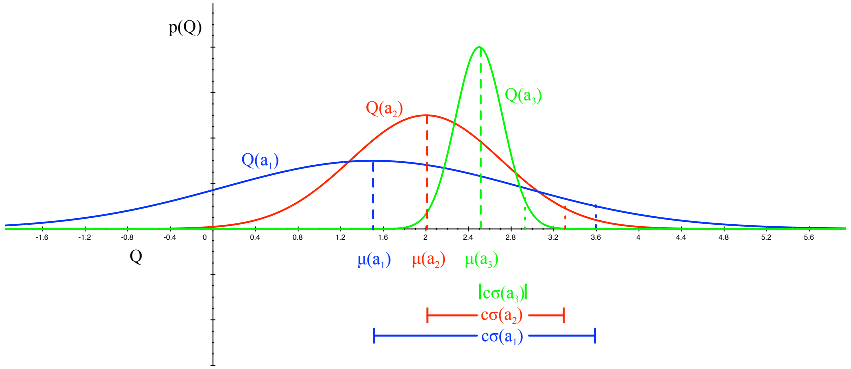


Image taken from UCL Course by David Silver – Lecture 9: XX.

Extension: Bayesian UCB

Example

- We again consider a Bernoulli distribution: rewards are either 0 or +1
- Prior: uniform on $[0,1]$ $\forall a \in \mathcal{A}$ (each mean reward is equally likely)
- The posterior is a Beta distribution $\mathrm{Beta}(\alpha_a, \beta_a)$ with initial parameters $\alpha_a = 1$ and $\beta_a = 1$ for each action a
- Update the posterior:
  - $\alpha_{a_t} \leftarrow \alpha_{a_t} + 1$, if $r_t = 0$
  - $\beta_{a_t} \leftarrow \beta_{a_t} + 1$, if $r_t = 1$
- Assume: $r_1 = 1, r_2 = 1, r_3 = 0, r_4 = 0$

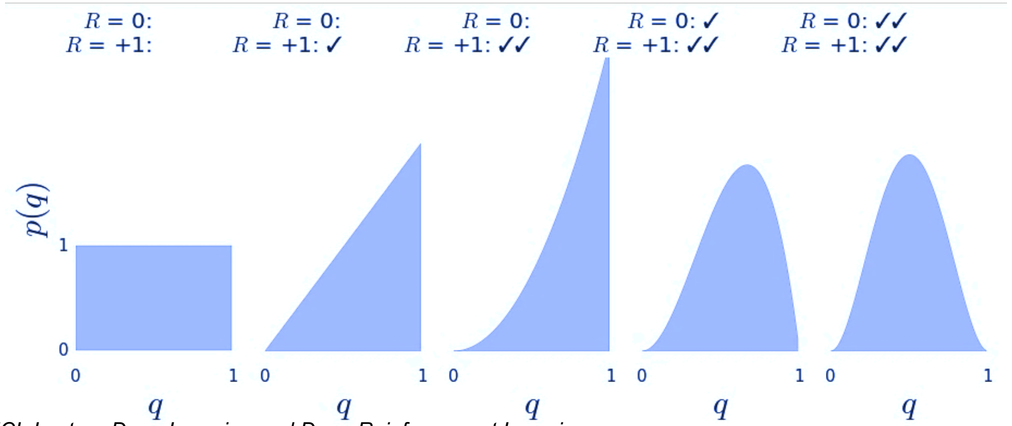
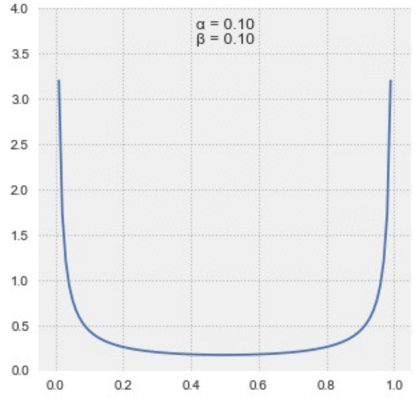


Image taken from Hado van Hasselt's UCL Lecture Deep Learning and Deep Reinforcement Learning

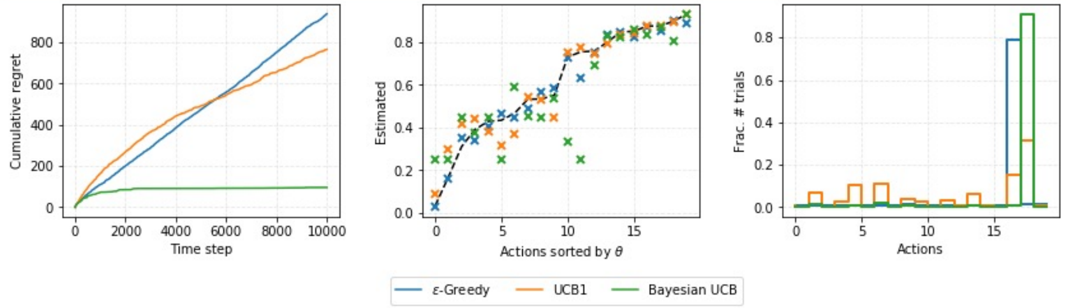


https://en.wikipedia.org/wiki/Beta_distribution

→ Pick action that maximizes
$$Q_t(a) + c\sigma(a)$$

Extension: Bayesian UCB (demo)

```python
class BayesianUCB(Solver):
    """Assuming Beta prior."""

    def __init__(self, bandit, c=3, init_a=1, init_b=1):
        """
        c (float): how many standard dev to consider as upper confidence bound.
        init_a (int): initial value of a in Beta(a, b).
        init_b (int): initial value of b in Beta(a, b).
        """
        super(BayesianUCB, self).__init__(bandit)
        self.c = c
        self._as = [init_a] * self.bandit.n
        self._bs = [init_b] * self.bandit.n

    @property
    def estimated_probas(self):
        return [self._as[i] / float(self._as[i] + self._bs[i]) for i in range(self.bandit.n)]

    def run_one_step(self):
        # Pick the best one with consideration of upper confidence bounds.
        i = max(
            range(self.bandit.n),
            key=lambda x: self._as[x] / float(self._as[x] + self._bs[x]) + beta.std(
                self._as[x], self._bs[x]) * self.c
        )
        r = self.bandit.generate_reward(i)

        # Update Gaussian posterior
        self._as[i] += r
        self._bs[i] += (1 - r)

        return i
```



Exploration via Probability Matching

We can also try the idea of directly sampling the action

- Select action a according to probability that a is the optimal action (given the history of everything we observed so far):

$$\begin{aligned}\pi_t(a|h_t) &= P[Q(a) > Q(a'), \forall a' \neq a|\ h_t] \\ &= \mathbb{E}_{r|h_t}\left[\mathbb{I}\left(a = \arg\max_{a \in \mathcal{A}} Q(a)\right)\right]\end{aligned}$$

Probability matching via Thompson Sampling:

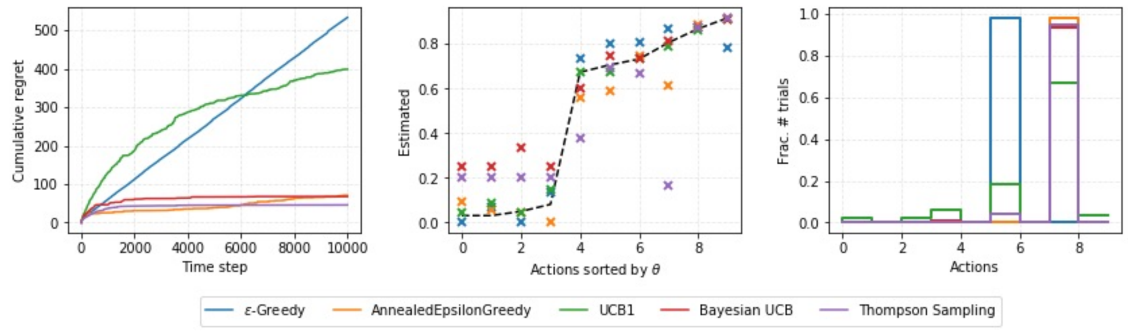
1. Assume $Q(a)$ follows a Beta distribution for the Bernoulli bandit
   - As $Q(a)$ is the success probability of θ
   - Beta(α, β) is within $[0,1]$, and α and β relate to the counts of success/failure
2. Initialize prior (e.g., $\alpha = \beta = 1$ or something different/what we think it is)
3. At each time step t we sample an expected reward $\hat{Q}(a)$ from the prior Beta(α_i, β_i) for every action
   - We select and execute the best action among the samples: $a_i^{TS} = \arg\max_{a \in \mathcal{A}} \hat{Q}(a)$
4. With the newly observed experience we update the Beta distribution:

$$\alpha_i \leftarrow \alpha_i + r_i \mathbb{I}[a_t^{TS} = a_i]$$

$$\beta_i \leftarrow \beta_i + (1 - r_i)\mathbb{I}[a_t^{TS} = a_i]$$

Exploration via Probability Matching (demo)

```python
class ThompsonSampling(Solver):
    def __init__(self, bandit, init_a=1, init_b=1):
        """
        init_a (int): initial value of a in Beta(a, b).
        init_b (int): initial value of b in Beta(a, b).
        """
        super(ThompsonSampling, self).__init__(bandit)

        self._as = [init_a] * self.bandit.n
        self._bs = [init_b] * self.bandit.n

    @property
    def estimated_probas(self):
        return [self._as[i] / (self._as[i] + self._bs[i]) for i in range(self.bandit.n)]

    def run_one_step(self):
        samples = [np.random.beta(self._as[x], self._bs[x]) for x in range(self.bandit.n)]
        i = max(range(self.bandit.n), key=lambda x: samples[x])
        r = self.bandit.generate_reward(i)

        self._as[i] += r
        self._bs[i] += (1 - r)

        return i
```



Classic Exploration Strategies: Summary

- We need exploration because information is valuable

No Exploration	→	Random Exploration	→	Smart Exploration
greedy		ϵ-greedy		UCB, Bayesian UCB, Thompson Sampling

- What did we not cover?
  - **Boltzman exploration:** the agent draws actions from a Boltzmann distribution (softmax) over the learned Q-values, regulated by a temperature parameter τ
  - When policies are approximated with neural networks:
    - **Entropy loss terms**: we can add an entropy term $H(\pi(a|s))$ into the loss function, encouraging the policy to take more diverse actions
    - **Noise-based Exploration:** add noise into the observation, action or even parameter space[1,2]

[1] *Meire Fortunato et al.: Noisy Networks for Exploration. ICLR 2018.*
[2] *Matthias Plappert et al.: Parameter Space Noise for Exploration. ICLR 2018.*