COMP3667: Reinforcement learning practical 2

Environments and multi-armed bandits

robert.lieck@durham.ac.uk

This is the version *with* answers!

Notebook with code

1 Overview

Welcome to the second reinforcement learning practical. This is a simple coding practical with the aim of getting early hands-on experience and developing intuition behind the environment and the reinforcement learning cycle. In this practical, we will:

- Define a basic multi-armed bandit environment.
- Build an agent solution from scratch intuitively and introduce the 'exploration vs exploitation' problem.
- Extend the multi-armed bandit problem to consider multiple states, and improve our agent's planning.

2 Basic environments

In the lecture we saw that the general setup of an environment consists of the step function, where

$$\text{observation}, \text{reward} = \text{env.step}(\text{action}).$$

Generally, we return the reward along with the next observation given the current environment's (internal) state and the action taken by the agent. There may also be some other functions such as for setting the environment's random seed, and for rendering the environment at the current state. The render function could simply print out some text for simple environments, or it could—for example—display a frame from a 3D scene.

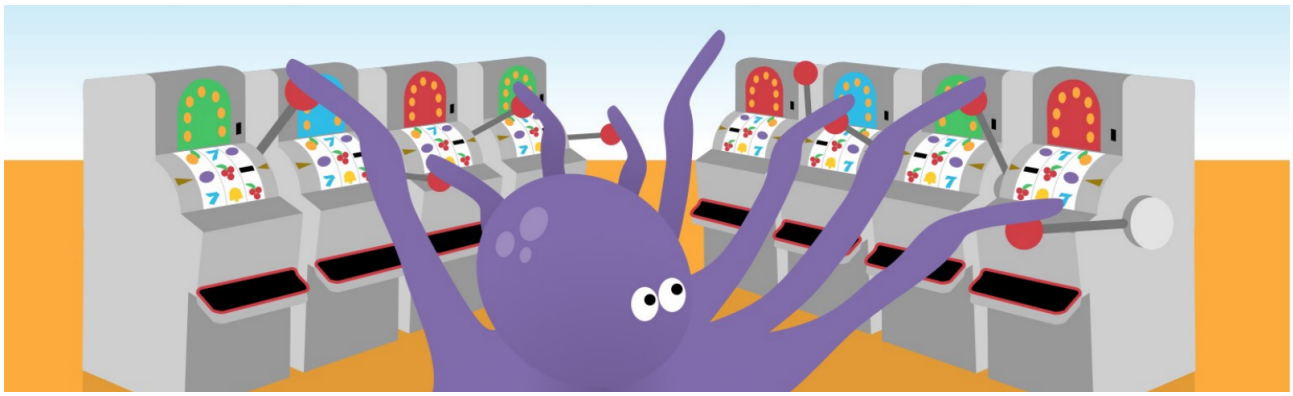
```
class Environment():
    def __init__(self):
        self.state = 0

    def step(self, a):
        self.state += 1
        reward = -1
        next_observation = 10 * self.state
        return next_observation, reward

    def reset(self):
        self.state = 0
        return self.state

    def render(self):
        print(self.state)
```

Sometimes you will see 'state' be used in stead of 'observation'. This relates to whether the agent has full- or partial-observability; often agents will only observe a small subset (e.g. limited camera or sensor data) of the full environment state (all the data required to simulate the full environment).

For the purpose of learning, we will restrict ourselves to a very simple environment and code it from scratch. However, the basic framework outlined today remains mostly the same for all reinforcement learning environments and problems.

3 Multi-armed bandits

The multi-armed bandit problem is a classical reinforcement learning challenge. Bandits are the slot machines you get in casinos. They are called bandits as the lever supposedly looks like a bandit holding a handgun, and they rob you of your money!



The game is simple—start using the machines (*exploring*) to find which slot machine gives you the most money, then *exploit* it repeatedly to maximise your profit. There is no internal state of the environment.

Some machines are more likely to give a monetary reward than others. The success rate of the individual machines is [0.4, 0.2, 0.1, 0.1, 0.1, 0.7] and the machines give the rewards [1, 0.1, 2, 0.5, 6, 70] accordingly. Therefore, the first machine has a 40% chance of winning £1 and the second machine has a 20% chance of winning 10p etc.

Incidentally, this introduces the first fundamental problem of reinforcement learning. How much should you explore different machines vs how much should you exploit a machine that you think gives good rewards?

Exercise

With only Python and NumPy (no Gym or Brax) implement a basic multi-armed bandit environment consisting of $N = 6$ slot machines. Therefore, the agent (octopus in the above illustration) has a discrete action space, where the action is an integer corresponding to the machine index $0, 1, \ldots, N-1$ of the lever that gets pulled at each environment step.

- Starting from the `Environment` class in the previous code listing, implement a multi-armed bandit environment in pure Python and NumPy, without using Gym or Brax.
  - Initialise the environment with:
    - `self.machine_probs = [0.4, 0.2, 0.1, 0.1, 0.1, 0.7]`
    - `self.machine_rewards = [1.0, 0.1, 2.0, 0.5, 6.0, 70.]`
    - `self.n_actions = 6`
  - The state in this simple environment does not change. You could imagine a casino with lots of rooms (states) containing different slot machines, but for now we will just have one room with ID 0.
  - Within the `step` function, which receives an action to pull a lever, set the initial reward to -£25; e.g. it costs £25 to use a machine.

Exercise solution

```
class BanditEnv():
    def __init__(self):
        self.state = 0
        self.n_actions = 6
        self.n_states  = 0
        self.probs   = np.array([0.4, 0.2, 0.1, 0.1, 0.1, 0.7])
        self.rewards = np.array([1.0, 0.1, 2.0, 0.5, 6.0, 70.])

    def step(self, a):
        reward = -25
        if np.random.uniform() < self.probs[a]:
            reward += self.rewards[a]
        return self.state, reward

    def reset(self):
        self.state = 0
        return self.state
```

- Now implement the reinforcement learning cycle (sample interactions with the environment) for a completely random agent.
  - You can use the following code for the random agent:

```
class Agent():
    def __init__(self):
        pass

    def sample_action(self, observation=0):
        return np.random.randint(env.n_actions)
```

- Run your random agent for 1,000 steps in this stateless environment, and calculate the answer to the following questions.
  - Approximately how much money does the agent make or lose overall?
  - Approximately how much money does the agent make or lose on average for each slot machine?
  - For clearer print statements, suppress scientific notation: `np.set_printoptions(suppress=True)`

Exercise solution

```
env = BanditEnv()
agent = Agent()
o = env.reset()
money = 0
money_per_machine = np.zeros(env.n_actions)
usage_per_machine = np.zeros(env.n_actions)
for episode in range(1000):
    a = agent.sample_action(o)
    o, r = env.step(a)
    money += r
    money_per_machine[a] += r
    usage_per_machine[a] += 1

print("about " + str(money))
print("about " + str(money_per_machine/usage_per_machine))
```

```
> about [-24.5, -24.98, -24.78, -25.02, -24.38, 23.65]
> about -£16,680 money lost (ouch!)
```

  - Calculate the average money $\boldsymbol{m}$ gained/lost for each machine exactly. Use the machine probabilities and machine rewards, remembering the £25 cost that you pay regardless of whether you win or loose—then compare your answer to what the agent estimated from its experience.

    The average is just a way to empirically estimate the expected value. So $\boldsymbol{m}$ is the expected value of

the reward over the distribution of possible outcomes (just two for each machine)

$$\begin{aligned}
\boldsymbol{m} &= \mathbb{E}_x[r_x] \\
&= \sum_x p_x\, r_x \\
&= p_0\, r_0 + p_1\, r_1 \\
&= (1-\boldsymbol{p})(-25) + \boldsymbol{p}\,(\boldsymbol{r}-25) \\
&= -25 + \boldsymbol{p}\,25 + \boldsymbol{p}\,\boldsymbol{r} - \boldsymbol{p}\,25 \\
&= \boldsymbol{r}\boldsymbol{p} - 25 \\
&\to [-24.6, -24.98, -24.8, -24.95, -24.4,\ 24].
\end{aligned}$$

– Now calculate exactly the average amount you would be expected to lose, following the random policy, and compare your answer.

$$\begin{aligned}
\mathbb{E}[\boldsymbol{x}] &= \sum_i \boldsymbol{x}_i\, p(\boldsymbol{x}_i) \\
&= \sum_i \boldsymbol{m}_i \frac{1}{6} \\
&= -£16.621\dot{6}.
\end{aligned}$$

Remark: *The agent produces an estimate of the state (or action) value by random sampling. This is a Monte Carlo method—obtaining numerical results without knowing the probabilities or rewards.*

3.1 Solving multi-armed bandits

As we've seen, in this trivial case you can calculate or even just eyeball which machine is best. Looking at the information, you can see the last machine (index 6) has 70% probability of giving £70—much higher than the others—so the optimal agent policy is to always choose $\pi(a|s) = [0, 0, 0, 0, 0, 1]$.

What have we just done intuitively? We have internally quantified the value of the action $q_*(a)$, for each slot machine, where we have imagined using the machine repeatedly to maximise our long-term reward over many Bernoulli trials, where (as very coarsely approximated in Sutton and Barto, chapter 2)

$$q_*(a) \approx \mathbb{E}[R_t \,|\, A_t = a]. \tag{1}$$

Then we have improved our policy immediately from an initial random exploratory (stochastic) policy to a deterministic policy—always choosing machine 6—by choosing the action that yields the highest action-value Q, where

$$A_t \approx \arg\max_a Q_t(a). \tag{2}$$

The problem is, when walking into a casino, we don't yet know the slot machine success probabilities or their associated reward distributions either. In other words you don't yet know the q_* value associated with the actions, and thus we need to learn q_* by our agent's experience within the environment.

Remark 1: *Later we will add to this definition of the action-value function q_* as it also depends on the environment state. For example, consider a casino with three rooms—each represented by a state. The machines in each room will be different, therefore each action's value depends on which room you're in. Further, consider the use of CCTV where casino operators change the reward distribution of the slot machines over time based on how well you're playing. This would imply that it is beneficial to update our action-value definition to consider not just the current reward, but the long-term cumulative rewards optimised under our policy.*

Remark 2: *In practice, we won't be able to immediately improve our initially stochastic (random) policy to the optimal policy. We will have to slowly learn to improve our policy until it converges on the optimal (deterministic) policy. To do this, we generally start out by exploring randomly, then slowly reduce the amount of stochastic exploration (noise or variance) until we converge on the optimal policy (exploiting the policy we have learnt to squeeze the most juice out of our environment).*

Exercise

- How would you find $q_*(a)$, assuming you had no previous knowledge about slot machines or the casino?

  You could try different machines randomly until you start to feel like you're winning, of course! More formally, you could calculate the expected action-value of each machine as the probability of winning (over k trials) times by the reward it gives. Remember to subtract the £25 cost of using the machine from the reward.

- Extend your random agent to learn the action-value function for each machine, under the random policy:
  - Have your agent initialise `self.q = np.zeros(env.n_actions)`
  - In the main outer loop, update `agent.q[a]` to be the value of the average reward when taking that action. You get a gold star for using an incremental mean to estimate this, moving the old estimate of `agent.q[a]` a little bit ($\alpha = 0.1$) in the direction of the target reward when taking that action.

$$\begin{aligned}
\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j && (3)\\
&= \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) && (4)\\
&= \frac{1}{k}(x_k + (k-1)\mu_{k-1}) && (5)\\
&= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}) && (6)\\
&\approx \mu_{k-1} + \alpha(x_k - \mu_{k-1}) && (7)\\
&= (1-\alpha)\,\mu_{k-1} + \alpha\, x_k && (8)\\
\text{new_estimate} &\approx \text{old_estimate} + \alpha(\text{target} - \text{old_estimate}). && (9)
\end{aligned}$$

Advanced: Equation equation 7 replaces the dynamically chaning k with a constant α. This exponentially smoothing average corresponds to the following recursion (stopping at $\mu_0 = 0$)

$$\begin{aligned}
\mu_k &= (1-\alpha)\,\mu_{k-1} + \alpha\, x_k && (10)\\
&= (1-\alpha)\,\left[(1-\alpha)\,\mu_{k-2} + \alpha\, x_{k-1}\right] + \alpha\, x_k && (11)\\
&= \quad \vdots && (12)\\
&= \alpha\sum_{i=0}^{k-1}(1-\alpha)^i x_{k-i} && (13)
\end{aligned}$$

That is

$$\begin{aligned}
\mu_0 &:= 0 && (14)\\
\mu_1 &= \alpha\, x_1 && (15)\\
\mu_2 &= \alpha\,(x_2 + (1-\alpha)\, x_1) && (16)\\
\mu_3 &= \alpha\,(x_3 + (1-\alpha)\, x_2 + (1-\alpha)^2\, x_1) && (17)\\
&\vdots && (18)
\end{aligned}$$

We see that this is not actually an unbised average of the observed x values (e.g. μ_1 is x_1 scaled by α, while it really should be just x_1). This is because the weights (which correspond to a truncated geometric series with common ratio $r = 1-\alpha$) only sum to 1 in the limit of large $k \to \infty$:

$$\alpha\sum_{i=0}^{k-1}(1-\alpha)^i = \alpha\frac{1-(1-\alpha)^k}{1-(1-\alpha)} = 1-(1-\alpha)^k \qquad (19)$$

But we can correct for this by normalising the weight (i.e., by dividing μ_k by the right-hand side of Equation equation 19) to get corrected estimates:

$$\bar{\mu}_k = \frac{\mu_k}{1-(1-\alpha)^k} \ . \tag{20}$$

  - Display your q values and check they are close to the exact solution found previously.

- What happens if you replace the sample_action function so that it simply always selects the action that brings the max q? For example always returns `return np.argmax(self.q)`

  The agent will exploit the best slot machine too early and won't explore. Therefore it won't converge on learning the true action-value funciton q. This also means the agent may not converge on learning the best policy (it will likely find the best policy in this simple environment, but in more complicated environments it almost certainly will not). For example, the agent may get stuck in a local environment minia—it may not discover the best casino rooms to explore or the best slot machines to play.

- What's a simple way to solve the above problem?

  Ensure that our policy (sample_action function) always includes some random exploration along with acting greedily—selecting the action that brings the max q. Control the amount of exploration with ϵ where $X > \epsilon$ results in acting greedily, and $X \leq \epsilon$ results in acting randomly. Sample X uniformly randomly. This is called an ϵ-greedy policy.

- Implement your solution and see that it converges on something close to the true action-value function q.

Exercise solution

```
class Agent():

    def __init__(self, env, alpha=None, epsilon=0):
        self.env = env
        self.alpha = alpha
        self.epsilon = epsilon
        self.k = np.zeros(self.env.n_actions)
        self.q = np.zeros(self.env.n_actions)

    @property
    def q_corrected(self):
        if self.alpha is None:
            return self.q
        else:
            return self.q / (1 - (1 - self.alpha)**self.k + 1e-8)

    def put_data(self, action, reward):
        self.k[action] += 1
        if self.alpha is None:
            # exact average
            if self.k[action] == 1:
                self.q[action] = r
            else:
                self.q[action] += (r - self.q[action]) / self.k[action]
        else:
            # smoothing average
            self.q[action] += self.alpha * (r - self.q[action])

            self.q[action] = (1 - self.alpha) * self.q[action] + self.alpha * r

    def sample_action(self, state=0, epsilon=0.4):
        if np.random.rand() < self.epsilon:
            return np.random.randint(env.n_actions)
        else:
            return np.argmax(self.q_corrected)

env = BanditEnv()
agent = Agent(env=env, alpha=0.1, epsilon=0.1)
s = env.reset()
for episode in range(1000):
    a = agent.sample_action(s)
    s, r = env.step(a)
    # learn to estimate the value of each action
    agent.put_data(a, r)

print(agent.k)
print(agent.q)
print(agent.q_corrected)

> about [ 19.  21.  15.  16.  18. 911.]
> about [-24.0397878  -24.67999842 -23.60210979 -24.13400123 -24.38846409  40.39368533]
> about [-27.79439863 -27.71224386 -29.7215035  -29.62324873 -28.69550519  40.39368493]
```

3.2 Multi-room multi-armed bandits

For fun, we can extend our multi-armed bandits environment to support multiple rooms.

- Add states to the environment. Imagine the casino you are in has two other rooms, both of which have no slot machines.
- Introduce an extra action $a = 0$ (there were 6 possible actions, now there are 7). When you take this action, you get no reward, but you advance the state to the next room. The rooms cycle, so when you're in the last room, you go back to the first room.

```
class BanditEnv():
    def __init__(self):
        self.n_actions = 6+1
        self.n_states  = 3
        self.state = 0
        self.probs = np.array([
            [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
            [0.0, 0.4, 0.2, 0.1, 0.1, 0.1, 0.7],
```

```
            [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
        ])
        self.rewards = np.array([
            [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ],
            [0.0, 1.0, 0.1, 2.0, 0.5, 6.0, 70.0],
            [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ],
        ])

    def step(self, a):
        r = -25

        # if first action taken, move to next state with no reward
        if a == 0:
            self.state = (self.state + 1) % self.n_states
            return self.state, 0
        # else pull the slot machine and get the reward, stay in current room
        if np.random.uniform() < self.probs[self.state, a]:
            r += self.rewards[self.state, a]
        return self.state, r

    def reset(self):
        self.state = 0
        return self.state

class Agent():
    def __init__(self):
        self.q = np.zeros([env.n_states, env.n_actions])

    def sample_action(self, state=0, epsilon=0.4):
        if np.random.rand() > epsilon:
            return np.argmax(self.q[state,:])
        else:
            return np.random.randint(env.n_actions)

env = BanditEnv()
s = env.reset()
agent = Agent()
for episode in range(1000):
    a = agent.sample_action(s)
    next_s, r = env.step(a)

    agent.q[s,a] += 0.1 * (r - agent.q[s,a])
    s = next_s

print(agent.q)
```

- Examine q. What is wrong with it? What is happening?

  The q values are currently only consider the action-value from the immediate reward; it doesn't plan ahead or look back at the consequences of past actions. When in the first room, there should be a higher q value associated with taking the action of moving to the next room, where we can get the nice juicy reward of using the last machine. Similarly, when in the last room, we should have a higher value of advancing to the first room, where we can then advance to the second room with the nice juicy reward of the last machine. So there needs to be some reccurrence relation, or a means to backtrack rewards through time.

- There are several ways to introduce a recurrence relation. One of the easiest ways is to improve the action-value function using a 'temporal difference' technique introduced in a future lecture:

  `agent.q[s,a] += 0.01 * (r + 0.99*np.max(agent.q[next_s,:])- agent.q[s,a])`

- While we won't cover the theory behind this today, implement the above and examine the q table. Does it look more reasonable now? What do you think has caused this?

  Yes, the table now gives a high value for the action of advancing to the slot machine room from the first empty room. Similarly, there should be a high (but not quite as high) value of advancing from the last room to the first room, so the agent can then proceed to the room with the slot machines. This is caused by `np.max(agent.q[next_s,:])` where we allow our q value to be influenced by looking ahead in time to the highest value that can be obtained from the next state. It's really clever when you think about it. It's like saying, "I'm in this state considering this action, what is the best value of doing this assuming I continue to behave optimally in the future?". This is like closing your eyes and imagining (planning) for the best possible future strategy.

4 More complicated environments

Today we've implemented and solved some very simple discrete environments. You may want to think how you would extend this framework to handle more complicated cases:

- Is it possible to have stochastic environment dynamics, e.g. transition between states based on an 'alcohol level' that influences the probability you end up in a random room, rather than the intended room?

  Yes, this is trivial to implement in the step function and it can be solved using the same methods.

- What types of environment will cause the q table to grow too large?

  Any environments with large branching factors in the state space. The branching factor for noughts and crosses is 9, the branching factor for chess is about 35, whereas the branching factor for Go is about 250. Continuous state spaces have an ∞ branching factor.

- How would you implement a continuous action and/or observation space?

  Rather than integers, your actions and/or observations would be real numbers accordingly.

- Can you think of challenges and possible solutions in using the agent's q table for continuous spaces?

  It would need to be infinitely large and therefore would not fit into memory. You could quantize the state space, but in practice its better to use a function approximator for the q table. Recall that the q table is just a function that maps from (s, a) to $\mathbb{R}$; you could therefore use linear function approximation or a deep neural network to learn that mapping instead of storing it in a massive table.

- If you've finished ahead of time, revisit the Colab code that accompanies the environments lecture and get a feel for running REINFORCE on the different discrete environments.
