reinforcement-learning-slides

May 24, 2022

1 Reinforcement learning

Steve Phelps

Certificate in Quantitative Finance, Fitch Learning

1.1 Overview

- recap of multi-armed bandits
- the exploitation-exploration trade-off
- exploration strategies: softmax versus epsilon-greedy
- risk-sensitivity in reinforcement-learning

1.2 Loss functions

- Nearly all machine-learning problem can be posed as optimisation problems.
- We define a loss function $L : (\mathbf{x}, F_{\mathbf{w}}) \rightarrow \mathbb{R}$
- where $\mathbf{x}$ is some data, and $F_{\mathbf{w}}$ is an arbitrary stochastic continuous function parameterised by numeric weights $\mathbf{w}$.
- F is often non-linear.
- We minimise the expected loss by choosing appropriate weights:

$$\underset{\mathbf{w}}{\operatorname{argmin}} E[L(\mathbf{x}, F_{\mathbf{w}})] \tag{1}$$

1.3 Control problems

- Reinforcement learning is technique for solving stochastic control problems.
- The function F defines a (stochastic) mapping between states of the world, and corresponding *actions* to take.
- The weights $\mathbf{w}$ typically specify propensities or probabilities for each action in each state.
- In the context of reinforcement-learning, we refer to the function F as a control *policy*.

- The loss function is the negative of the stochastic *return* obtained by taking the resulting actions.
- In contrast to traditional stochastic control,
- we do not have to obtain a closed-form solution for the dynamics of the environment.

1.4 Learning agents

- The entity taking the actions is called the agent.
- The agent repeatedly takes *actions* $a_t \in \mathbb{A}$ in discrete time periods $t \in \mathbb{N}$.
- When the agent chooses action a_t at time t, it obtains an immediate observable *reward* r_{t+1}.
- The *return* G_t at time t is some function f of the future rewards $G_t = f(r_{t+1}, r_{t+2}, \ldots)$
- Often we use $G_t = \sum_{i=t}^{\infty} \gamma^i r_{i+1}$ where $\gamma \in [0, 1]$ is the *time discounting* of the agent.

1.5 Markov Decision Processes

- The reward is a function of the action chosen in the previous state $s_t \in \mathbb{S}$.
- The s state and action a at time t determines the probability of the subsequent state s' and reward r.
- The probabilities are specified by the function $p(s', r|s, a)$.
- This specifies a finite Markov Decision-Process.
- The goal of the agent is to maximise the expected return $E[G]$.
- The agent follows a *policy* which specifies an action to take in each state: $\pi(s) \in \mathbb{A}$
- The optimal policy is denoted $\pi*$.

1.5.1 MDP example

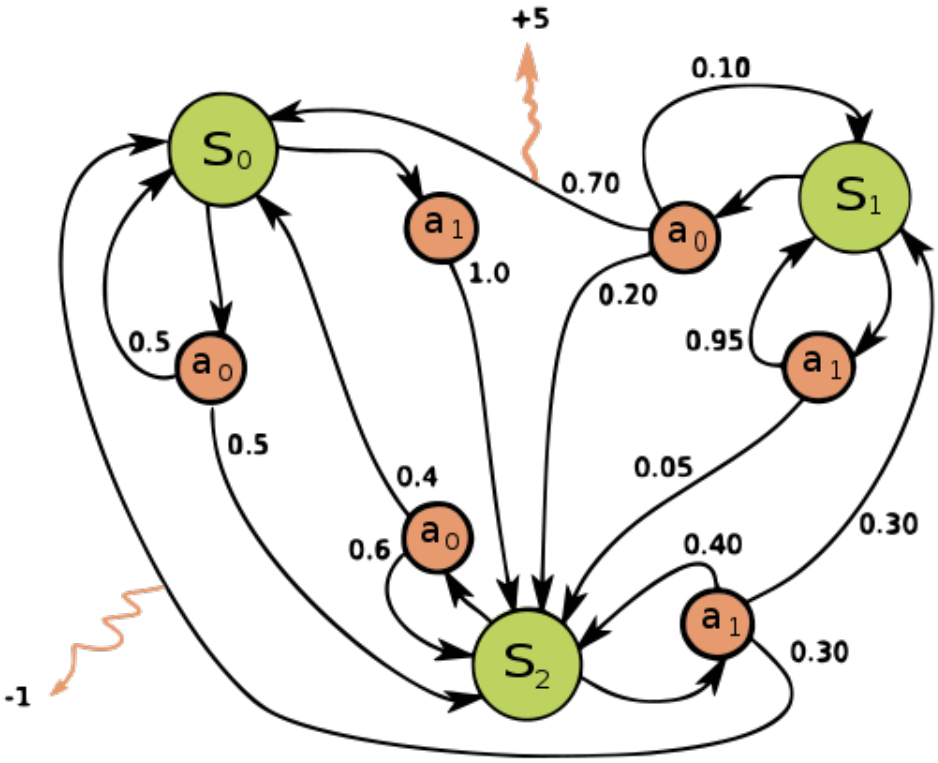


Image by Waldo Alvarez under CC-by-sa 4.0.

1.6 Multi-armed bandits

1.7 Multi-armed bandits as simple MDPs

- We can consider a multi-armed bandit as an MDP with a *single* state.
- The observed reward r_t is a realisation of an iid. random variable R_a whose distribution $F_a(r)$ depends on the action.
- At the beginning of the problem, the reward distributions F_a are *unknown*.
- We often assume that the reward distributions are *stationary*.
- The return for the agent over time period T is $G_T = \sum_{t=1}^{T} r_t$.
- Since the reward distribution depends on the actions, then similarly for the expected return $E[G]$.
- Therefore, we can solve a multi-armed bandit by finding the action with the highest expected reward.

1.8 The Dark Pool Problem

- As an example, consider a simple order-routing problem.
- We can submit orders to one of several dark-pools.
- A dark-pool does not display an order-book, and only accepts large market-orders (volume but no price).
- Each dark-pool reports the filled quantity, and the price obtained.
- Our reward function is the proportion of the order that is filled.
- For a constant volume of shares per unit time, this is a multi-armed bandit problem.
- This is called The Dark Pool Problem.

1.9 A multi-armed bandit in Python

- Consider an n-armed bandit where $R_a \sim N(a, 1)$.

[1]:
```
import numpy as np

def play_bandit(a, variance=1.0):
    """ Return the reward from taking action a """
    return np.random.normal(a, scale=np.sqrt(variance))
```

At $t = 1$, suppose we choose action $a = 1$, we then obtain realised reward r_2:

[2]:
```
r_2 = play_bandit(a=1)
r_2
```

[2]:
```
0.43537167258403053
```

1.10 Value functions

- We use the function $v(a)$ to denote the expected return for action a over the entire episode:

$$v(a) = E[G|a_t = a \; \forall_t] \tag{2}$$
$$= E[R_a] \tag{3}$$

- Typically, the function v is unknown to the agent.
- In this scenario, the agent performs *sequential decision making under uncertainty*.

1.11 The greedy action

- If we can compute $v(a)$, then the agent's optimal policy is simple.
- The agent simply chooses the action with the highest expectation:

$$a* = \underset{a}{\text{argmax}} v(a) \tag{4}$$

- We break ties arbitrarily.
- We call $a*$ the *greedy* action.

1.12 Learning as sampling

- The random variates r_t are directly observable.
- Therefore they are *samples* from the distribution $F_a(r)$.
- How can we estimate $v(a)$ given the observed rewards $r_1, r_2, \ldots, r_t$?

1.13 Value estimation

- We can use experience to "learn" the expectations $V(a)$.
- That is, we can use *sampling* to *estimate* V.

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{k_a}}{k_a} \tag{5}$$

- By the law of large numbers:

$$\lim_{k_a \to \infty} Q(a) = v(a) \tag{6}$$

1.14 Value estimation in Python

- We can translating the equation from the previous slide into code.

[3]:
```
def sample_from_bandit(a, k):
    rewards = []
    for t in range(k):
        r = play_bandit(a)
        rewards.append(r)
    return np.mean(rewards)
```

- To sample $k = 20$ times using action $a = 2$:

[4]:
```
q_2 = sample_from_bandit(a=2, k=20)
q_2
```

[4]:
```
1.65751781843102
```

1.15 Optimising the Python code

- We can make our code more scalable by using `map` instead of a `for` loop.

[5]:
```
def sample_from_bandit(a, k):
    return np.mean(map(lambda i: play_bandit(a), range(k)))
```

- or equivalently a comprehension:

[6]:
```
def sample_from_bandit(a, k):
    return np.mean([play_bandit(a) for t in range(k)])
```

- The code is simpler, and closer to mathematical notation.
- For large sample sizes, comprehensions can be faster to compute than for loops.
- Code in this form is more easily translated into TensorFlow.

1.16 Further code optimisation

- Our code is still not optimal though.
- For large sample sizes we need to allocate memory to hold all the previous samples.

1.17 Incremental update of estimates

- We can rewrite the previous update equation thus:

$$
\begin{aligned}
Q_{k+1} &= \frac{1}{k}\sum_{i=1}^{k} r_i && (7)\\
&= \frac{1}{k}\left[r_k + \sum_{i=1}^{k-1} r_i\right] && (8)\\
&= \frac{1}{k}\left[r_k + (k-1)\frac{1}{k-1}\sum_{i=1}^{k-1} r_i\right] && (9)\\
&= \frac{1}{k}\left[r_k + (k-1)Q_k\right] && (10)\\
&= \frac{1}{k}\left[r_k + kQ_k - Q_k\right] && (11)\\
&= Q_k + \frac{1}{k}\left[r_k - Q_k\right] && (12)\\
& && (13)
\end{aligned}
$$

1.18 Temporal difference learning

- We are adjusting an old estimate towards a new estimate based on more recent information.
- We can think of the coefficient $(k)^{-1}$ as a *step size* parameter.

$$Q_{k+1} = \frac{1}{k}\left[r_k - Q_k\right] \tag{14}$$

```
new_estimate = old_estimate + step_size * (target - old_estimate)
```

1.19 Incremental updates in Python

[7]:
```
def update_q(old_estimate, target_estimate, k):
    step_size = 1./(k+1)
    error = target_estimate - old_estimate
    return old_estimate + step_size * error

def sample_from_bandit(a, k):
    current_estimate = 0.
    for t in range(k):
        current_estimate = update_q(current_estimate, play_bandit(a), t)
    return current_estimate
```

[8]:
```
q_2 = sample_from_bandit(a=2, k=100000)
q_2
```

[8]:
```
1.9991323149375104
```

1.20 Using a constant step size

- Recall that with our previous update rule, the `step_size` parameter varies with each update.

$$Q_{k+1} = \frac{1}{k}\left[r_k - Q_k\right] \tag{15}$$

- Alternatively, we can use a *constant* step size $\alpha \in [0, 1]$:

$$Q_{k+1} = Q_k + \alpha\left[r_k - Q_k\right]. \tag{16}$$

1.21 Exponential moving-average of rewards

$$\begin{aligned}
Q_{k+1} &= Q_k + \alpha\left[r_k - Q_k\right] && (17)\\
&= \alpha r_k + (1-\alpha)Q_k && (18)\\
&= \alpha r_k + (1-\alpha)\left[\alpha r_{k-1} + (1-\alpha)Q_{k-1}\right] && (19)\\
&= \alpha r_k + (1-\alpha)\alpha r_{k-1} + (1-\alpha)^2 Q_{k-1} && (20)\\
&= \alpha r_k + (1-\alpha)\alpha r_{k-1} + (1-\alpha)^2\alpha r_{k-2} + \ldots && (21)\\
&\quad + (1-\alpha)^{k-1}\alpha r_1 + (1-\alpha)^k Q_0 && (22)\\
&= (1-\alpha)^k Q_0 + \sum_{k=1}^{k} \alpha(1-\alpha)^{k-i} r_i && (23)\\
& && (24)
\end{aligned}$$

1.22 Recency

- Thus we see that if we use constant step-size parameter, we are calculating an exponential (i.e. time-weighted) *moving* average.

- The moving average is an estimate of the expected reward.
- The parameter α is sometimes called the *recency*.
- This parameter is *not* learned, but instead is chosen by the scientist.
- Parameters that are chosen but not learned are called *hyper-parameters*.

1.23 Non-stationary environments

- If the first moment of the reward distribution is non-stationary,
- then we should choose a larger values of α.

1.24 Recency in Python

[9]:
```
def update_q(old_estimate, target_estimate, recency=0.2):
    error = target_estimate - old_estimate
    return old_estimate + recency * error
```

1.25 Action selection

- As discussed, if we have accurate estimates $Q(a) = v(a)$, then the optimal policy $\pi*$ is straightforward:

$$a_t* = \underset{a}{\text{argmax}} Q_t(a). \tag{25}$$

- However, is this the optimal policy if estimates are inaccurate?

1.26 Exploration

- Value forecasting and policy selection are *not independent*.
- If estimates $Q(a)$ are based on small sample sizes k_a, and we always choose the greedy action, then our policy will be suboptimal.
- Taking the greedy action is called exploitation.
- In order to learn, our agent must also *explore* the environment by sampling from alternative actions.
- This is true even when current information suggests that the alternative are suboptimal;
- current information may be inaccurate.

1.27 ϵ-greedy exploration

- There are many different strategies for exploration.
- With ϵ-greedy exploration with probability ϵ we choose an action randomly, otherwise we exploit.
- We introduce a new hyper-parameter $\epsilon \in [0, 1]$.

- Each time we choose action we draw a random variate $\eta_t \sim U(0,1)$.

$$\eta_t \leq \epsilon \implies a_t = a_t* \tag{26}$$
$$\eta_t > \epsilon \implies a_t \sim U(1,n) \tag{27}$$
$$\tag{28}$$

1.28 ϵ-greedy exploration

[10]:
```
def explore(q):
    return np.random.randint(len(q))

def exploit(q):
    greedy_actions, = np.where(q == np.max(q))
    return np.random.choice(greedy_actions)    # break ties randomly

def act(q, epsilon=0.99):
    if np.random.random() < epsilon:
        return exploit(q)
    else:
        return explore(q)
```

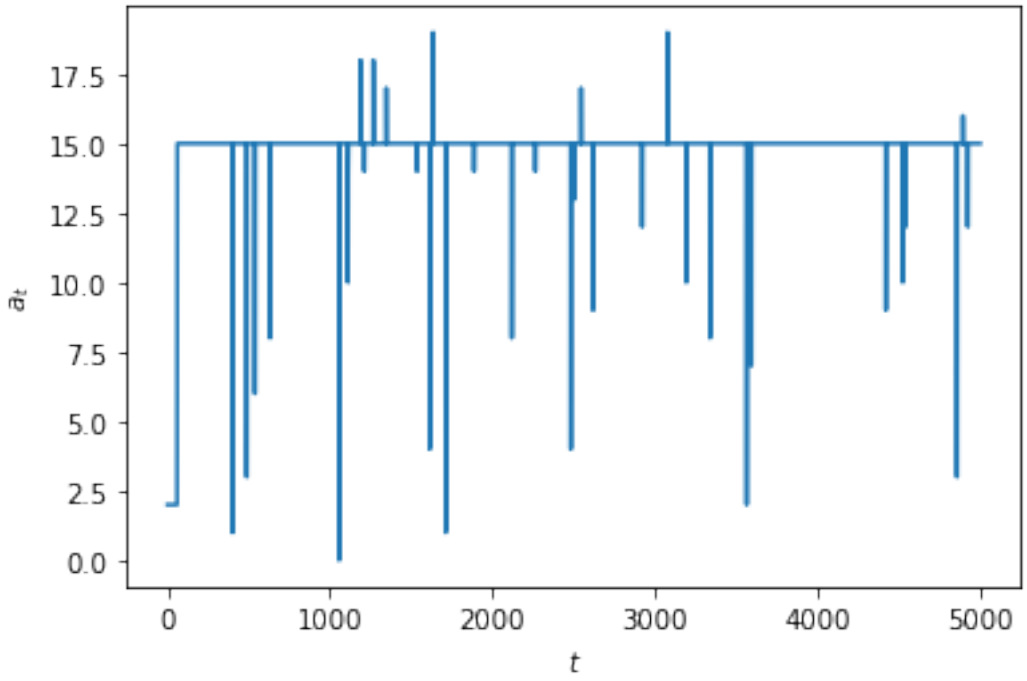
1.29 Complete example

[11]:
```
def simulate_agent(n = 20, T = 5000, epsilon=0.99, variance=1.0):
    q = np.zeros(n)
    actions = np.zeros(T); rewards = np.zeros(T)
    for t in range(T):
        a = act(q, epsilon)
        reward = play_bandit(a, variance)
        q[a] = update_q(q[a], reward)
        actions[t] = a
        rewards[t] = reward
    return q, actions, rewards
```

1.30 The time series of actions a_t

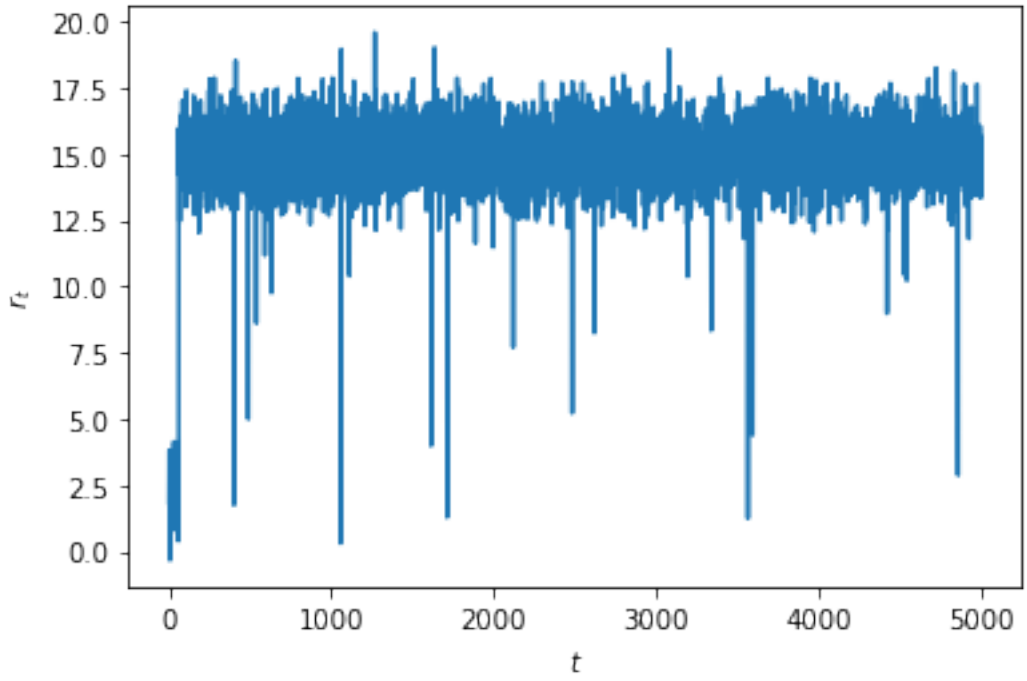
[12]:
```
import matplotlib.pyplot as plt

q, actions, rewards = simulate_agent()
plt.plot(actions)
plt.xlabel('$t$'); plt.ylabel('$a_t$');
plt.show()
```



1.31 The time series of rewards r_t

[13]:
```
plt.plot(rewards)
plt.xlabel('$t$'); plt.ylabel('$r_t$'); plt.show()
```



1.32 The expected reward

- From this single realisation of the model we can calculate a crude estimate of the expected reward from following our epsilon-greedy policy

[14]:
```
np.mean(rewards)
```

[14]:
```
14.829066119838421
```

- Although the policy improves over time, it is still below the value of the optimal policy $V_{a*=20} = 20$:

[15]:
```
np.mean(rewards[4500:])
```

[15]:
```
14.960628226935565
```

[16]:
```
np.var(rewards[4500:])
```

[16]:
```
1.2791383883771532
```

1.33 Tuning the exploration rate hyper-parameter

- What factors determine the optimal choice of ϵ?

1.34 The exploration/exploitation trade-off

- Exploration can be costly since there is an opportunity-cost from not taking the greedy action;
- provided that our value estimates are accurate.
- If estimates are inaccurate, we must explore in order to increase accuracy through sampling.

1.35 Standard error of the mean

- Our Q values are sample means, therefore their standard error is:

$$SE_Q = \frac{\sigma_r}{\sqrt{k_a}} \tag{29}$$

1.36 Uncertainty in rewards

- If we are certain about the rewards then there is no need to explore.
- Uncertainty can arise from:
  - *variance* in the reward distributions, and
  - non-stationarity in the underlying data-generation process for each reward distribution.
- Note that variance in reward distributions can be a modelling artifact;

- since sometimes we lack *knowledge* of, or the ability to *control*, extraneous variables which affect the reward.
- This latter uncertaintity relates to the concept of *states*, which we discuss later.

1.37 Increasing the exploration rate

- Let's rerun the simulation with a higher exploration of $1 - \epsilon$ by setting $\epsilon = 0.97$.

[17]:
```
q, actions, rewards = simulate_agent(epsilon=0.97)
```

[18]:
```
np.mean(rewards[4500:])
```

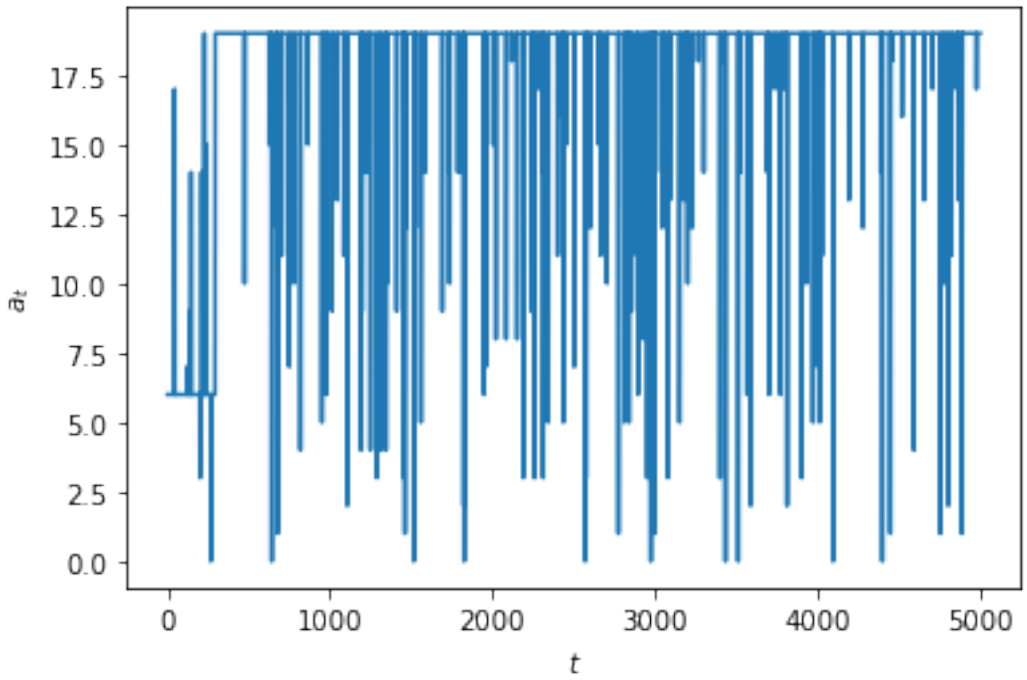
[18]:
```
18.82358252216539
```

[19]:
```
np.var(rewards[4500:])
```

[19]:
```
4.154283642220208
```

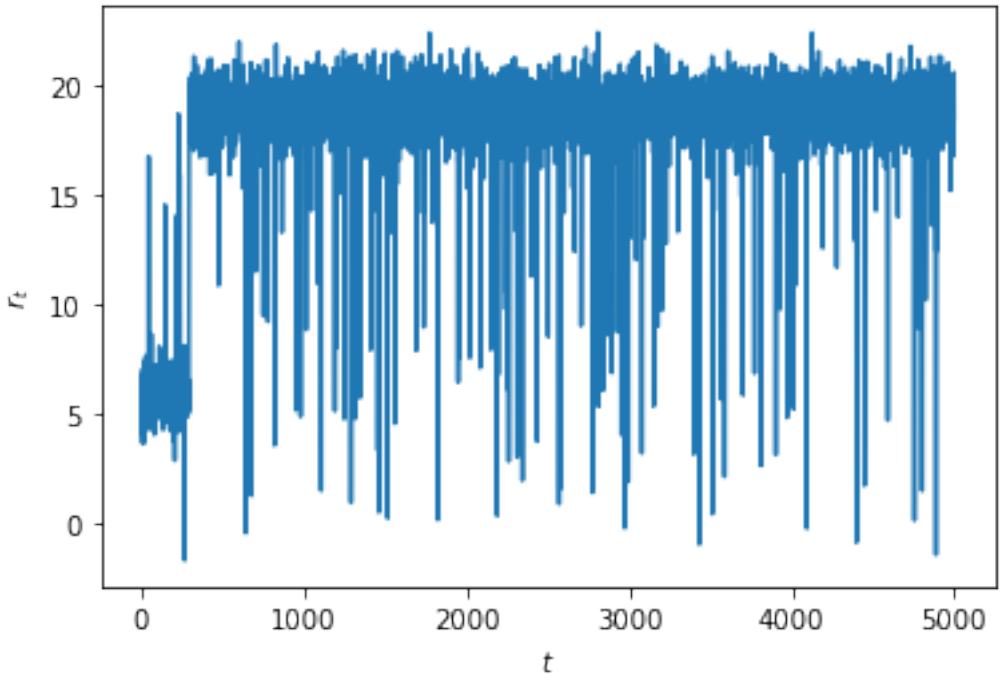
1.38 Time series of actions

[20]:
```
plt.plot(actions)
plt.xlabel('$t$'); plt.ylabel('$a_t$'); plt.show()
```



1.39 Time series of rewards

[21]:
```
plt.plot(rewards)
plt.xlabel('$t$'); plt.ylabel('$r_t$'); plt.show()
```



1.40 Softmax exploration

$$Pr\{A_t = a\} = \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{k} e^{Q_t(b)/\tau}} \tag{30}$$

where τ is a hyper-parameter denoting a positive *temperature*.

- Higher temperatures cause all actions to be nearly equiprobable.
- In the limit as $\tau \to 0$ softmax action selection is the same as greedy action selection.
- Exploration is biased towards those actions with higher currently-estimated return.

1.41 The Boltzmann distribution in Python

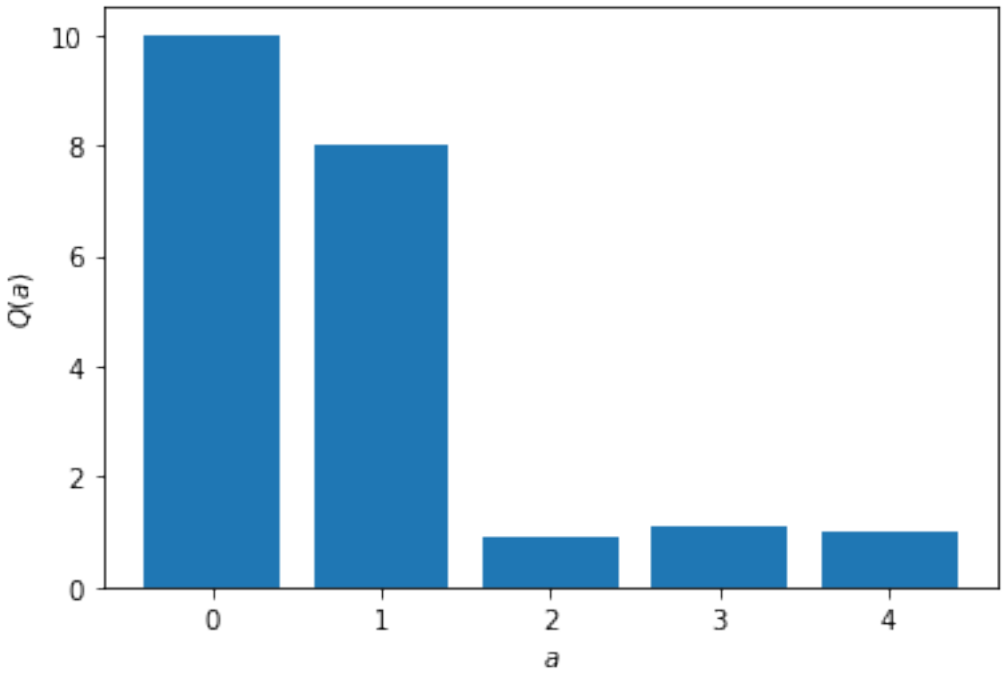
[22]:
```
def boltzmann(propensity, tau):
    return np.exp(propensity / tau)

def boltzmann_distribution(propensities, tau):
    x = np.array([boltzmann(p, tau) for p in propensities])
    total = np.sum(x)
```

```
    return x / total
```

1.42 Example Q values

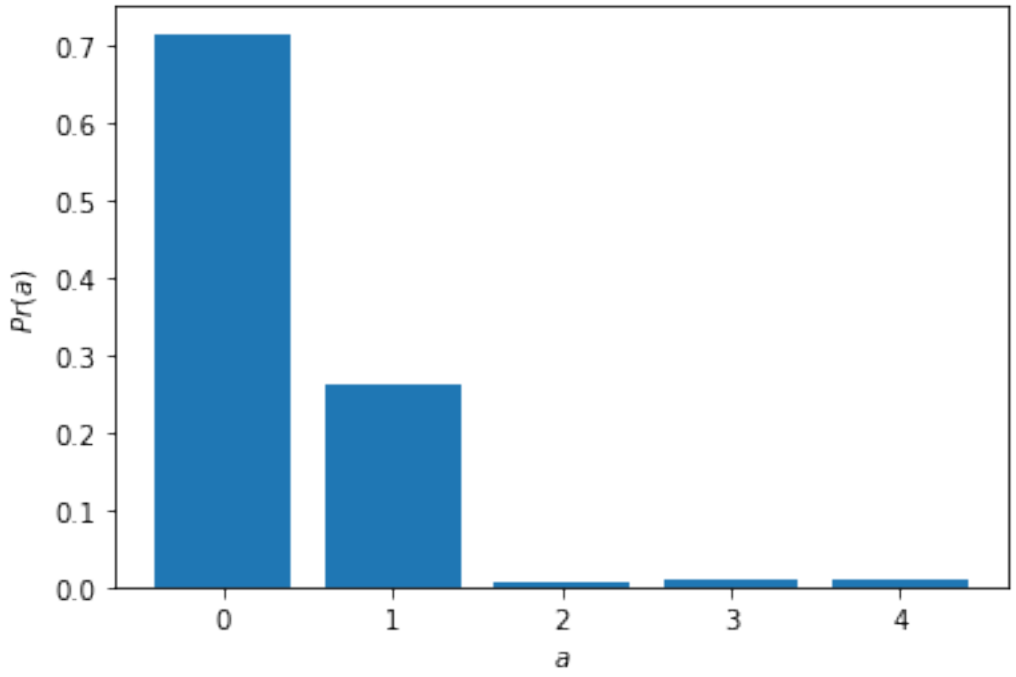
[23]:
```
Q = np.array([10., 8., 0.9, 1.1, 1.])
plt.bar(range(len(Q)), Q);  plt.xlabel('$a$'); plt.ylabel('$Q(a)$')
plt.show()
```



1.43 Softmax probabilities for $\tau = 2$

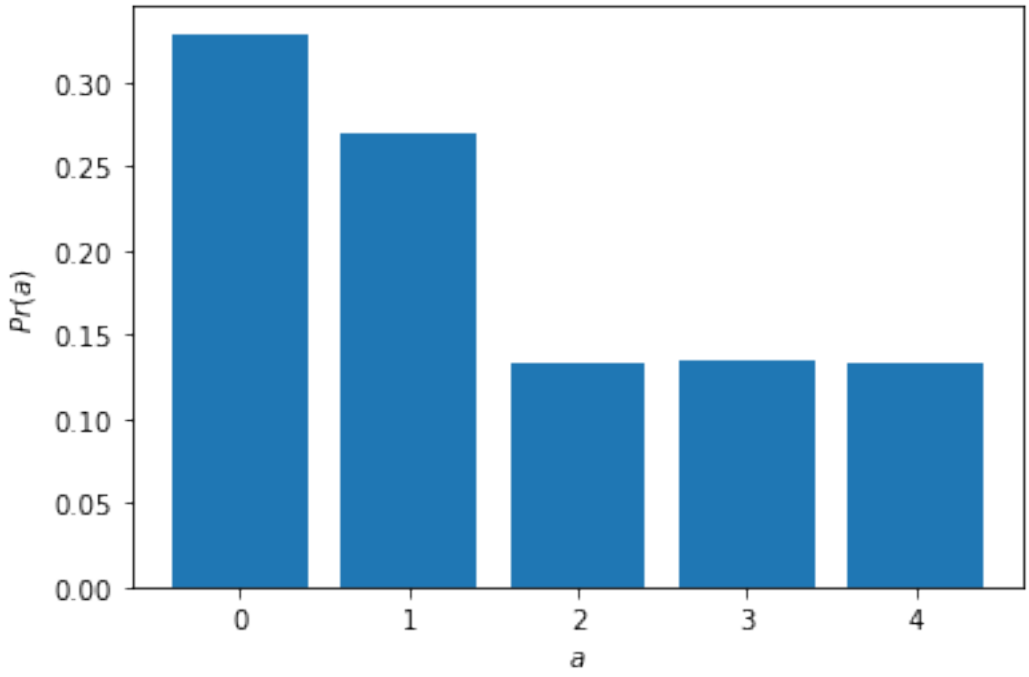
- We can compute $Pr(a)$ for a given temperature τ.
- Below we show the action selection probabilities for $\tau = 2$.
- As we increase the temperature, we reduce the skew of the distribution.

[24]:
```
plt.bar(range(len(Q)), boltzmann_distribution(Q, tau=2.))
plt.xlabel('$a$'); plt.ylabel('$Pr(a)$')
plt.show()
```



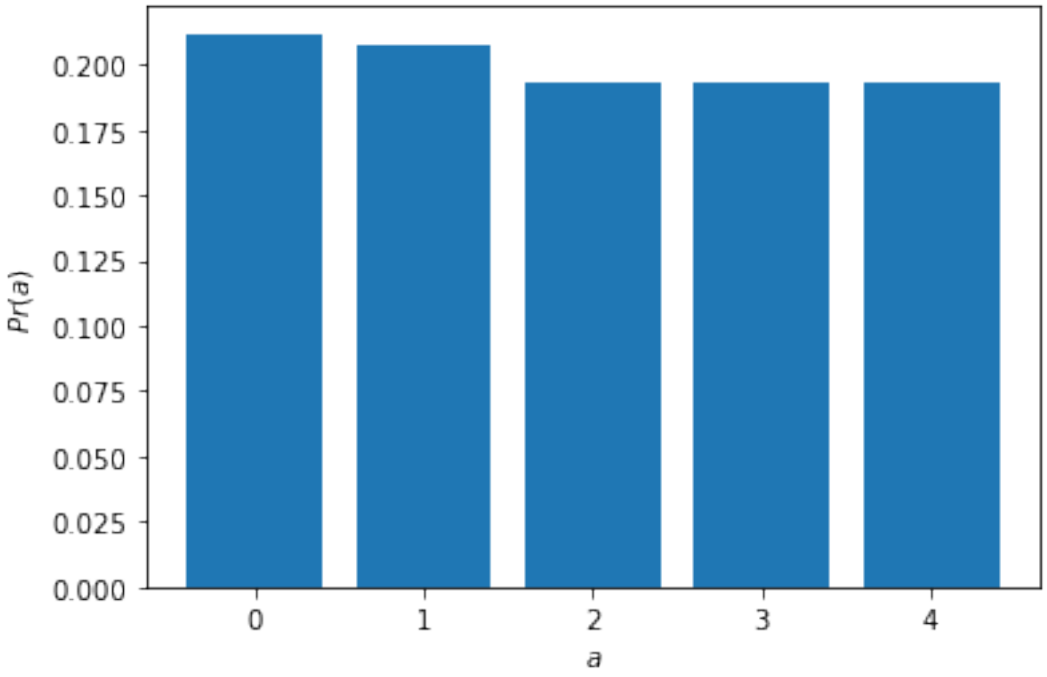
1.44 $\tau = 10$

[25]:
```
plt.bar(range(len(Q)), boltzmann_distribution(Q, tau=10.))
plt.xlabel('$a$'); plt.ylabel('$Pr(a)$')
plt.show()
```



1.45 $\tau = 100$

[26]:
```
Q = np.array([10., 8., 1., 1., 1.])
plt.bar(range(len(Q)), boltzmann_distribution(Q, tau=100.))
plt.xlabel('$a$'); plt.ylabel('$Pr(a)$')
plt.show()
```



1.46 Summary

- Reinforcement-learning is an example of machine-learning.
- It can be used to solve stochastic control problems.
- We do not require a model of the environment; we can learn directly from *experience*; e.g. a realised profit/loss.
- Successful application of this technique requires careful selection of hyper-parameters.
- We should consider uncertainty and risk when selecting exploration parameters.
- So far, we have considered problems with a single state.
- We will next extend this framework to multiple states using a case-study of a simple algorithmic trading strategy.