FitchLearning

Reinforcement learning

Author:
(c) Steve Phelps 2019
sphelps@sphelps.net

October 2021

Contents

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:

$$\underset{\mathbf{w}}{\operatorname{argmin}} E[L(\mathbf{x}, F_{\mathbf{w}})] \tag{1.1}$$

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.

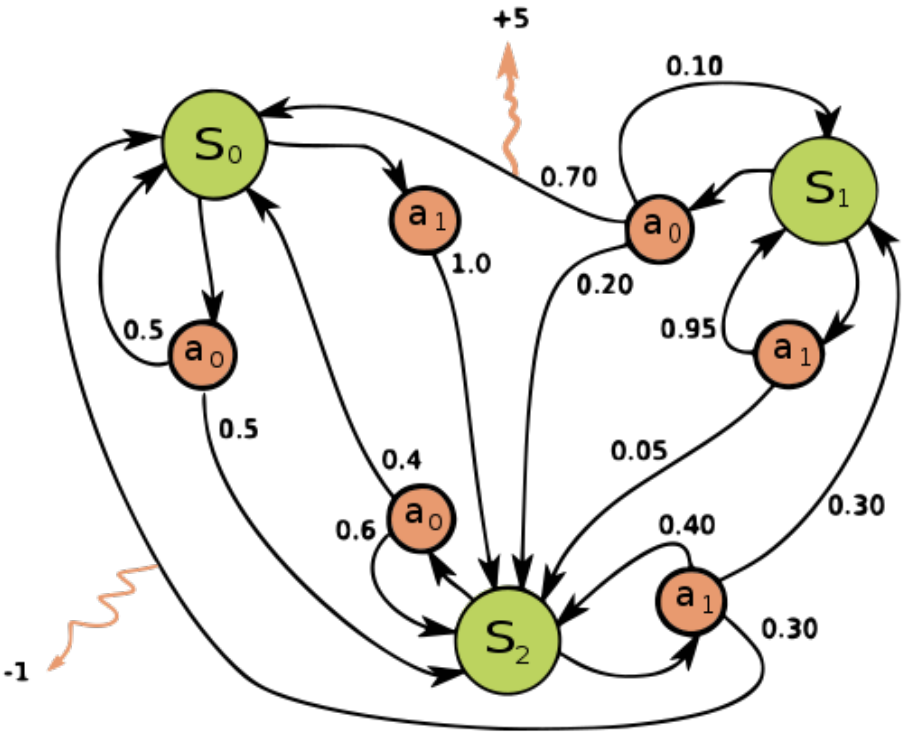


Figure 1.1: MDP

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

Figure 1.2: picture of bandit

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 $R_a \sim N(a, 1)$.

```
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:

```
r_2 = play_bandit(a=1)
r_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:

$$v(a) = E[G|a_t = a \; \forall_t] \tag{1.2}$$
$$= E[R_a] \tag{1.3}$$

- 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:

$$a* = \underset{a}{\text{argmax}} v(a) \tag{1.4}$$

- 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.

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{k_a}}{k_a} \tag{1.5}$$

- By the law of large numbers:

$$\lim_{k_a \to \infty} Q(a) = v(a) \tag{1.6}$$

1.14 Value estimation in Python

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

```
1 def sample_from_bandit(a, k):
2     rewards = []
3     for t in range(k):
4         r = play_bandit(a)
```

```
            rewards.append(r)
        return np.mean(rewards)
```

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

```
q_2 = sample_from_bandit(a=2, k=20)
q_2
```

```
1.65751781843102
```

1.15 Optimising the Python code

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

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

- or equivalently a comprehension:

```
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:

$$
\begin{aligned}
Q_{k+1} &= \frac{1}{k}\sum_{i=1}^{k} r_i && (1.7)\\
&= \frac{1}{k}\left[r_k + \sum_{i=1}^{k-1} r_i\right] && (1.8)\\
&= \frac{1}{k}\left[r_k + (k-1)\frac{1}{k-1}\sum_{i=1}^{k-1} r_i\right] && (1.9)\\
&= \frac{1}{k}\left[r_k + (k-1)Q_k\right] && (1.10)\\
&= \frac{1}{k}\left[r_k + kQ_k - Q_k\right] && (1.11)\\
&= Q_k + \frac{1}{k}\left[r_k - Q_k\right] && (1.12)\\
& && (1.13)
\end{aligned}
$$

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.

$$Q_{k+1} = \frac{1}{k}\left[r_k - Q_k\right] \tag{1.14}$$

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

1.19 Incremental updates in Python

```
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
```

```
q_2 = sample_from_bandit(a=2, k=100000)
q_2
```

```
1.9991323149375104
```

1.20 Using a constant step size

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

$$Q_{k+1} = \frac{1}{k}\left[r_k - Q_k\right] \tag{1.15}$$

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

$$Q_{k+1} = Q_k + \alpha\left[r_k - Q_k\right]. \tag{1.16}$$

1.21 Exponential moving-average of rewards

$$
\begin{aligned}
Q_{k+1} &= Q_k + \alpha\left[r_k - Q_k\right] && (1.17)\\
&= \alpha r_k + (1-\alpha)Q_k && (1.18)\\
&= \alpha r_k + (1-\alpha)\left[\alpha r_{k-1} + (1-\alpha)Q_{k-1}\right] && (1.19)\\
&= \alpha r_k + (1-\alpha)\alpha r_{k-1} + (1-\alpha)^2 Q_{k-1} && (1.20)\\
&= \alpha r_k + (1-\alpha)\alpha r_{k-1} + (1-\alpha)^2\alpha r_{k-2} + \ldots && (1.21)\\
&\quad + (1-\alpha)^{k-1}\alpha r_1 + (1-\alpha)^k Q_0 && (1.22)\\
&= (1-\alpha)^k Q_0 + \sum_{k=1}^{k} \alpha(1-\alpha)^{k-i} r_i && (1.23)\\
& && (1.24)
\end{aligned}
$$

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

```
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:

$$a_t* = \underset{a}{\text{argmax}} Q_t(a). \tag{1.25}$$

- 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 $\eta_t \sim U(0, 1)$.

$$\eta_t \leq \epsilon \implies a_t = a_t * \tag{1.26}$$
$$\eta_t > \epsilon \implies a_t \sim U(1, n) \tag{1.27}$$
$$\tag{1.28}$$

1.28 ϵ-greedy exploration

```
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)
```

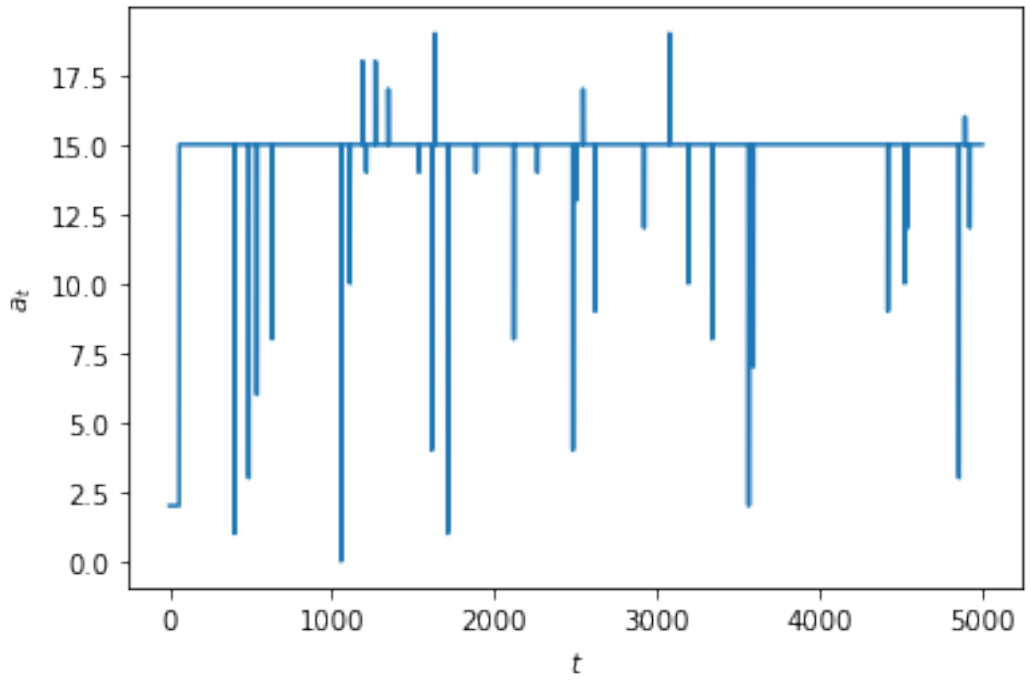
1.29 Complete example

```
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
```

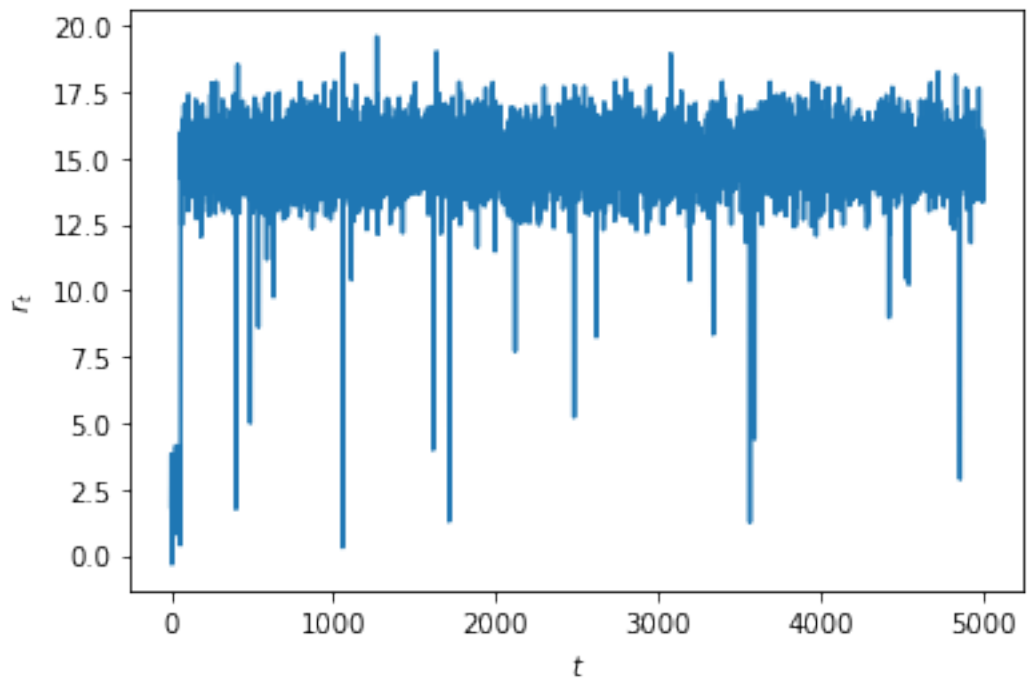
1.30 The time series of actions a_t

```
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

```
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

```
np.mean(rewards)
```

```
14.829066119838421
```

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

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

```
14.960628226935565
```

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

```
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:

$$SE_Q = \frac{\sigma_r}{\sqrt{k_a}} \tag{1.29}$$

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 $\epsilon = 0.97$.

```
q, actions, rewards = simulate_agent(epsilon=0.97)
```

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

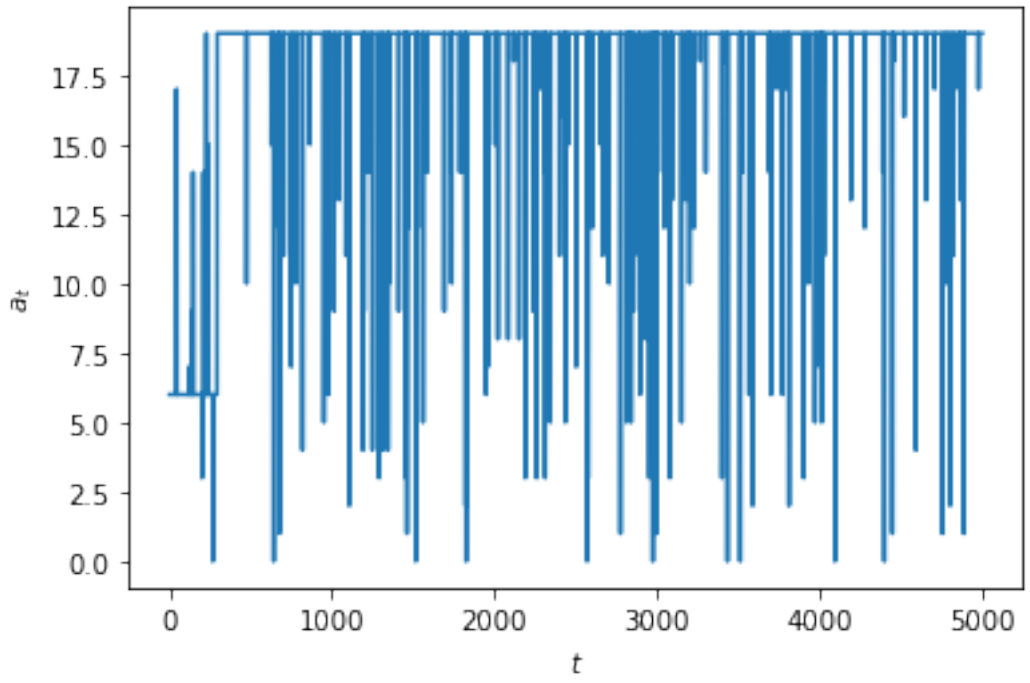
```
18.82358252216539
```

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

```
4.154283642220208
```

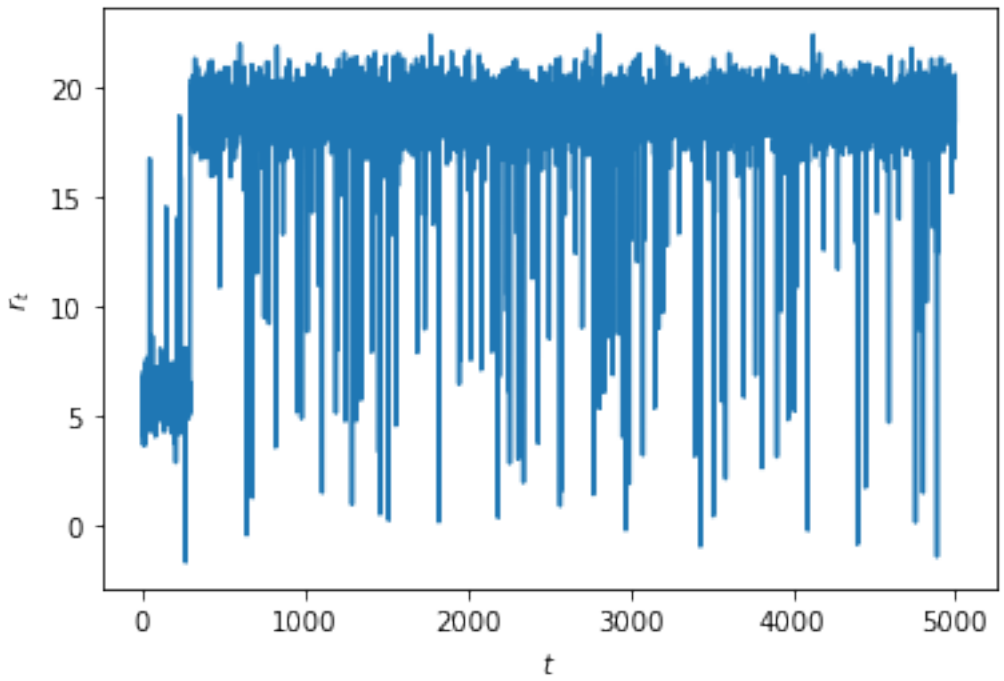
1.38 Time series of actions

```
plt.plot(actions)
plt.xlabel('$t$'); plt.ylabel('$a_t$'); plt.show()
```



1.39 Time series of rewards

```
plt.plot(rewards)
plt.xlabel('$t$'); plt.ylabel('$r_t$'); plt.show()
```



1.40 Softmax exploration

$$Pr\{A_t = a\} = \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{k} e^{Q_t(b)/\tau}} \tag{1.30}$$

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

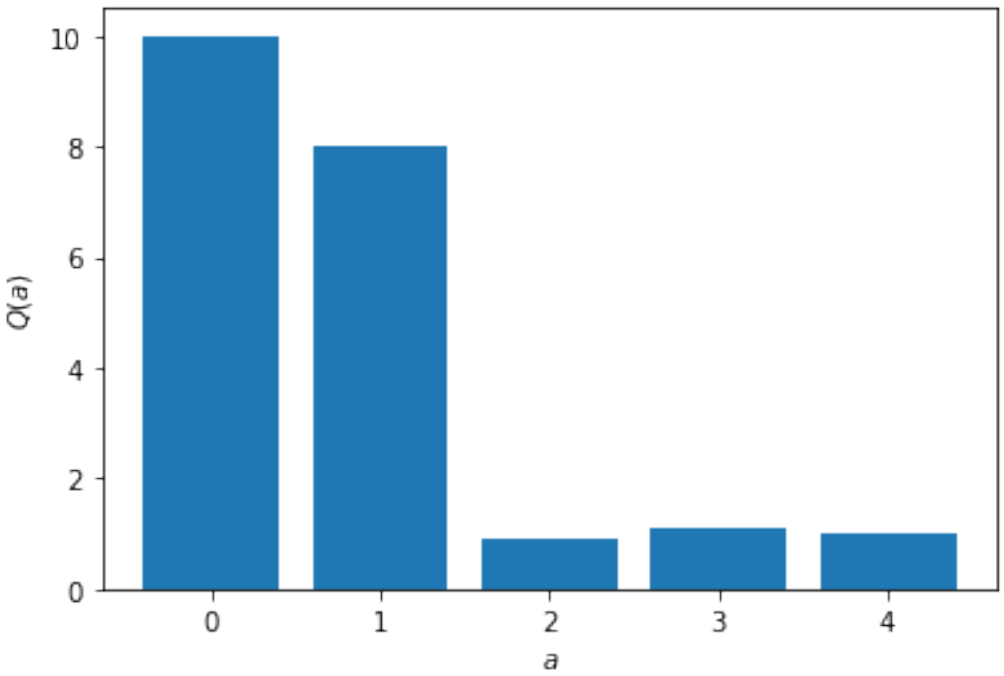
- Higher temperatures cause all actions to be nearly equiprobable.
- In the limit as $\tau \to 0$ softmax action selection is the same as greedy action selection.
- Exploration is biased towards those actions with higher currently-estimated return.

1.41 The Boltzmann distribution in Python

```
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

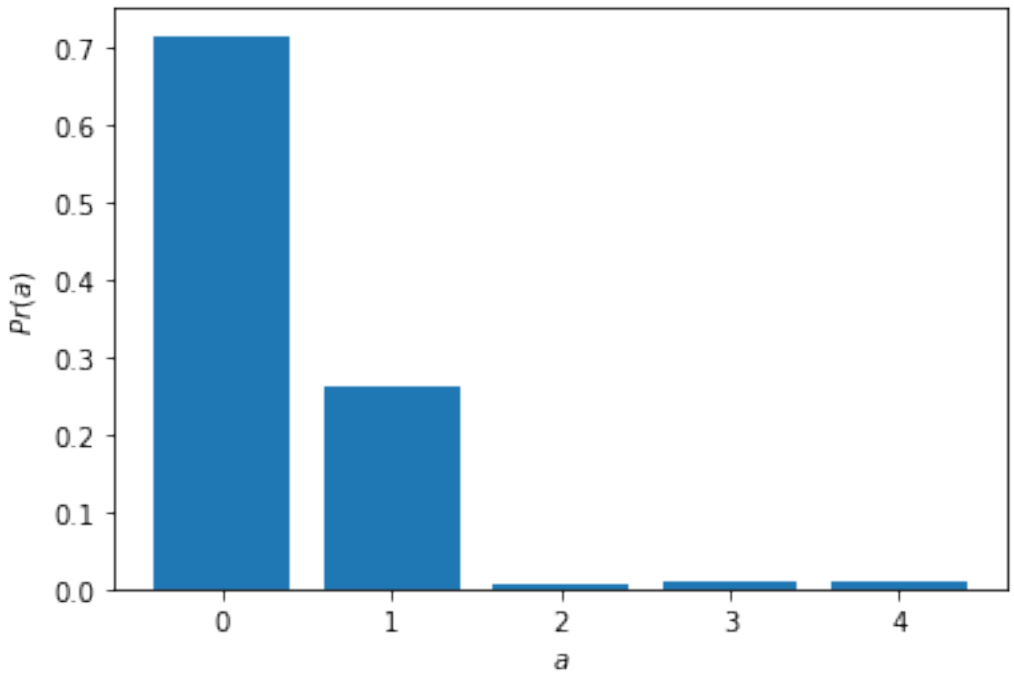
```
Q = np.array([10., 8., 0.9, 1.1, 1.])
plt.bar(range(len(Q)), Q);  plt.xlabel('$a$'); plt.ylabel('$Q(a)$')
plt.show()
```



1.43 Softmax probabilities for $\tau = 2$

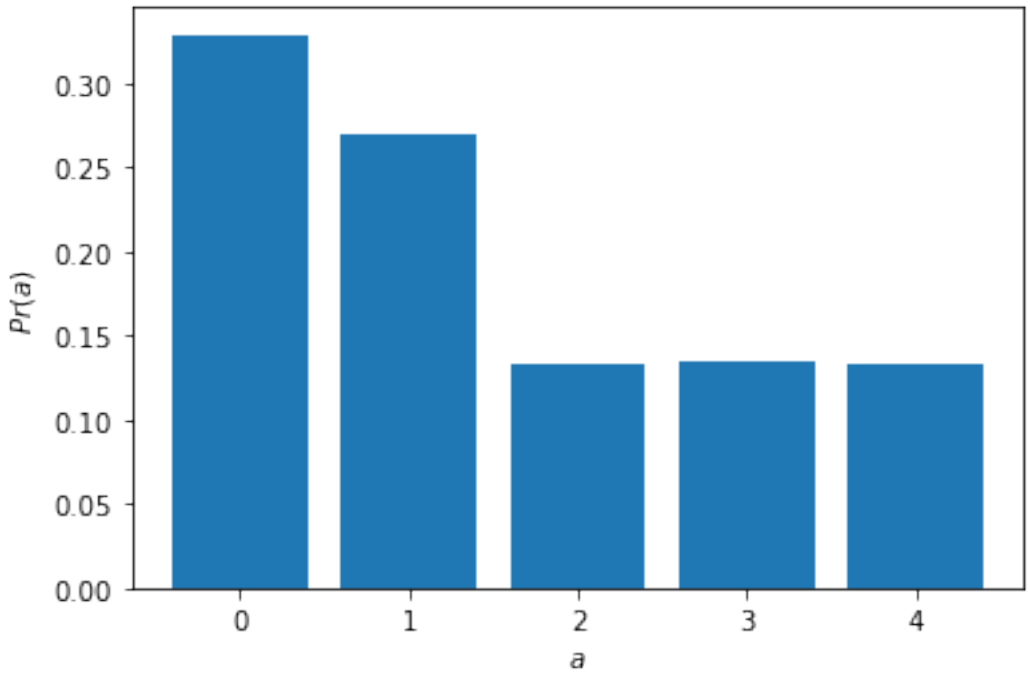
- We can compute $Pr(a)$ for a given temperature τ.
- Below we show the action selection probabilities for $\tau = 2$.
- As we increase the temperature, we reduce the skew of the distribution.

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



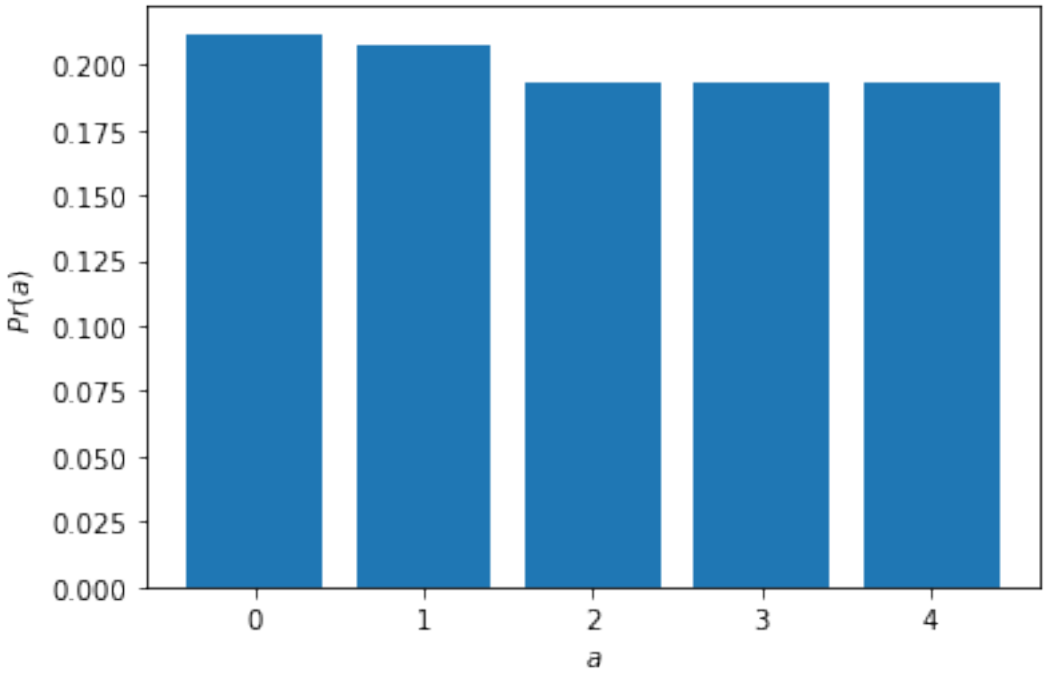
1.44 $\tau = 10$

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



1.45 $\tau = 100$

```
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.

2 Reinforcement-learning case-study

Steve Phelps

2.1 Overview

- We will implement a simplified algorithmic trading strategy in Python using reinforcement-learning.
- Our agent will attempt to earn profit from market-making in a high-frequency market.
- We will consider a simplified environment, but the problem can be scaled up to more realistic scenarios.
- It is based on the work of Chan and Shelton 2001; "An Electronic Market-Maker".
- A more structured, and fully commented, version of the code used in these notes is available on github.

2.2 Training an agent in a simulation

- Allowing an agent to learn in the real-world can be risky.
- We can train our agent in a simulation environment.
- The simulation environment is sometimes called a "gym".
- If the simulation is a good model of the real-world then the learned policy will still perform well in real-world.
- The agent can still adapt its policy online in the real-world even if the simulation is not a good model.

2.3 The market model

- We consider three types of agent:
  - informed traders,
  - uninformed traders, and
  - a single market-maker.
- Prices evolve intra-day in discrete time periods $t \in \mathbb{N}$.
- A single asset is traded.
- All trades and orders involve a single share of the asset.
- There is no order crossing between traders.
- The arrival of traders at the market follow a Poisson process.

2.3.1 The fundamental price

- The true price of the asset $p_t^* \in \mathbb{Z}$ follows a Poisson process.
- The parameter $\lambda_p \in [0, 1]$ is the probability of a discrete change in the price.

$$p_t^*(t) = p_0 + \sum_{i=1}^{t} \eta_t \tag{2.1}$$

where η_t is chosen i.i.d. from $(-1, +1, 0)$ with probabilities $(\lambda_p, \lambda_p, 1 - 2\lambda_p)$ respectively.

2.3.2 The market-maker

- In the simplest form of the model, the market-maker posts a single price p_t^m.
- The market-maker can adjust its price:

$$p_{t+1}^m = p_t + \Delta p_t \tag{2.2}$$

where the price changes are discrete and finite; $\Delta p_t \in \{-1, 0, +1\}$.

- The reward at time t is the change in the profit:
  - for a sell order: $r_t = p_t^* - p_t^m$.
  - for a buy order: $r_t = p_t^m - p_t^*$.
- More advanced versions of the model consider separate bid and ask prices, and corresponding spread.

2.3.3 Informed traders

- Informed traders have information about the fundamental price p_t^*.
- They can submit market orders for immediate execution at the market-maker's price p_t^m.
- They submit
  - a buy order iff. $p_t^* > p_t^m$.
  - a sell order iff. $p_t^* < p_t^m$.
  - no order iff. $p_t^* = p_t^m$.
- They arrive at the market with probability λ_i.

2.3.4 Uninformed traders

- Uninformed traders arrive at the market with probability $2\lambda_u$.
- They submit a buy order for $+1$ shares with probability λ_u, or a sell order for -1 shares with equal probability λ_u.

2.3.5 The overall process

- All Poisson processes are combined:

$$2\lambda_p + 2\lambda_u + \lambda_i = 1 \tag{2.3}$$

- There is an event at every discrete time period t.
- Trade occurs a finite period of time $t \in \{1, 2, \ldots, T\}$ where T is the duration of a single trading day.
- The market maker operates over many days.
- The initial conditions for every day are the same; they are independent *episodes*.

2.3.6 The market-maker as an agent

- We can consider the market-maker as an adaptive agent.
- The environment consists of the observable variables in the market.
- Initially the observable state is the total order-imbalance IMB_t.
- The variables are discrete, therefore there is a discrete state space.
- The market-maker chooses actions in discrete time periods t.
- The set of actions $\mathbb{A}$ available to the agent is $\Delta p \in \mathbb{A} = \{-1, 0, +1\}$.
- It can choose actions conditional on observations in order to maximise expected return $E[G]$ where $G = \sum_t \gamma^t r_t$.

2.3.7 Importing the required modules

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

2.3.8 Parameterising the model

- In the following examples we use the parameterisation of the model:

$$\begin{aligned} \lambda_p &= 0.2 \\ \lambda_u &= 0.1 \\ \lambda_i &= 0.4 \\ T &= 150 \\ p_0^* &= 200 \end{aligned}$$

2.3.9 Setting up the parameters in Python

```
INITIAL_PRICE = 200
MAX_T = 150

PROB_PRICE = 0.2

PROB_PRICE_UP = PROB_PRICE
PROB_PRICE_DOWN = PROB_PRICE

PROB_UNINFORMED = 0.1

PROB_UNINFORMED_BUY = PROB_UNINFORMED
PROB_UNINFORMED_SELL = PROB_UNINFORMED

PROB_INFORMED = 0.4

ALL_PROB = [PROB_PRICE_DOWN, PROB_PRICE_UP, PROB_UNINFORMED_BUY,
    ↪ PROB_UNINFORMED_SELL, PROB_INFORMED]
```

```
np.sum(ALL_PROB)
```

```
1.0
```

2.3.10 Representing events

```
EVENT_PRICE_CHANGE_UP     = 0
EVENT_PRICE_CHANGE_DOWN   = 1
EVENT_UNINFORMED_BUY      = 2
EVENT_UNINFORMED_SELL     = 3
EVENT_INFORMED_ARRIVAL    = 4
```

```
ALL_EVENT = \
    [EVENT_PRICE_CHANGE_DOWN, EVENT_PRICE_CHANGE_UP,
    ↪ EVENT_UNINFORMED_SELL,
         EVENT_UNINFORMED_BUY, EVENT_INFORMED_ARRIVAL]
```

2.3.11 Simulating the Poisson process

```
def simulate_events(probabilities=ALL_PROB):
     return np.random.choice(ALL_EVENT, p=probabilities, size=MAX_T
    ↪ )
```

```
events = simulate_events()
```

The first ten events:

```
events[:10]
```

```
array([2, 1, 0, 0, 1, 3, 0, 4, 0, 2])
```

2.3.12 Simulating the price process

```
fundamental_price_changes = np.zeros(MAX_T)
```

```
fundamental_price_changes[events == EVENT_PRICE_CHANGE_DOWN] = -1
fundamental_price_changes[events == EVENT_PRICE_CHANGE_UP] = +1
```

```
fundamental_price_changes[:10]
```

```
array([ 0., -1.,  1.,  1., -1.,  0.,  1.,  0.,  1.,  0.])
```

```
fundamental_price = \
    INITIAL_PRICE + np.cumsum(fundamental_price_changes)
```

```
fundamental_price[:10]
```

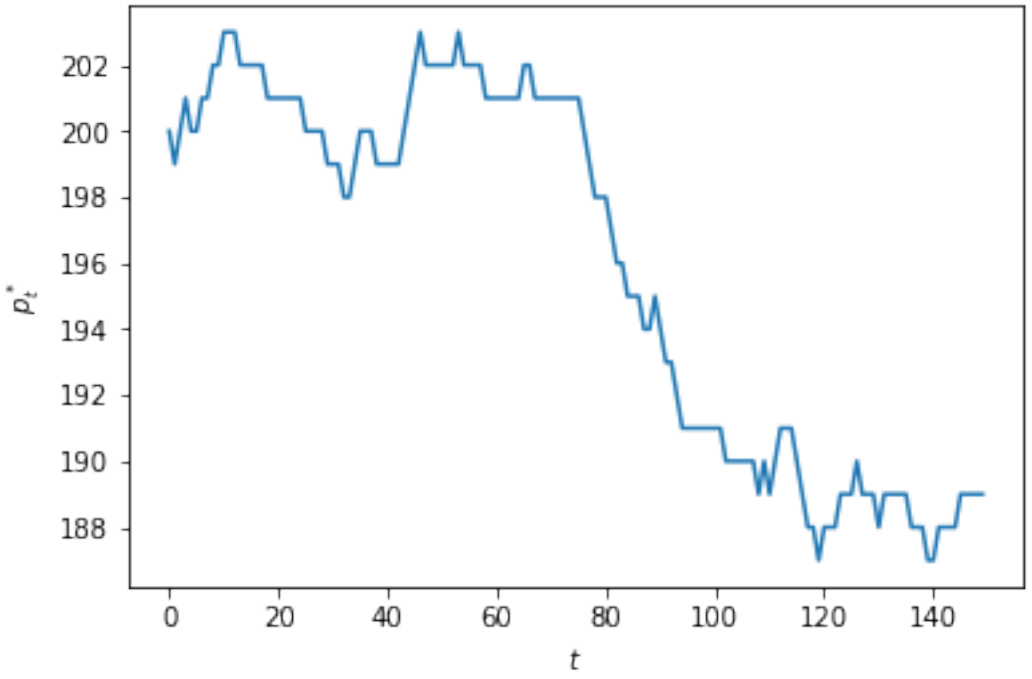
```
array([200., 199., 200., 201., 200., 200., 201., 201., 202., 202.])
```

2.3.12.1 As a function

```
def simulate_fundamental_price(events):
    price_changes = np.zeros(MAX_T)
    price_changes[events == EVENT_PRICE_CHANGE_DOWN] = -1
    price_changes[events == EVENT_PRICE_CHANGE_UP] = +1
    return INITIAL_PRICE + np.cumsum(price_changes)

fundamental_price = simulate_fundamental_price(events)
```

2.3.12.2 A single realisation of the price process.

```
plt.plot(fundamental_price)
plt.xlabel('$t$'); plt.ylabel('$p_t^*$')
plt.show()
```



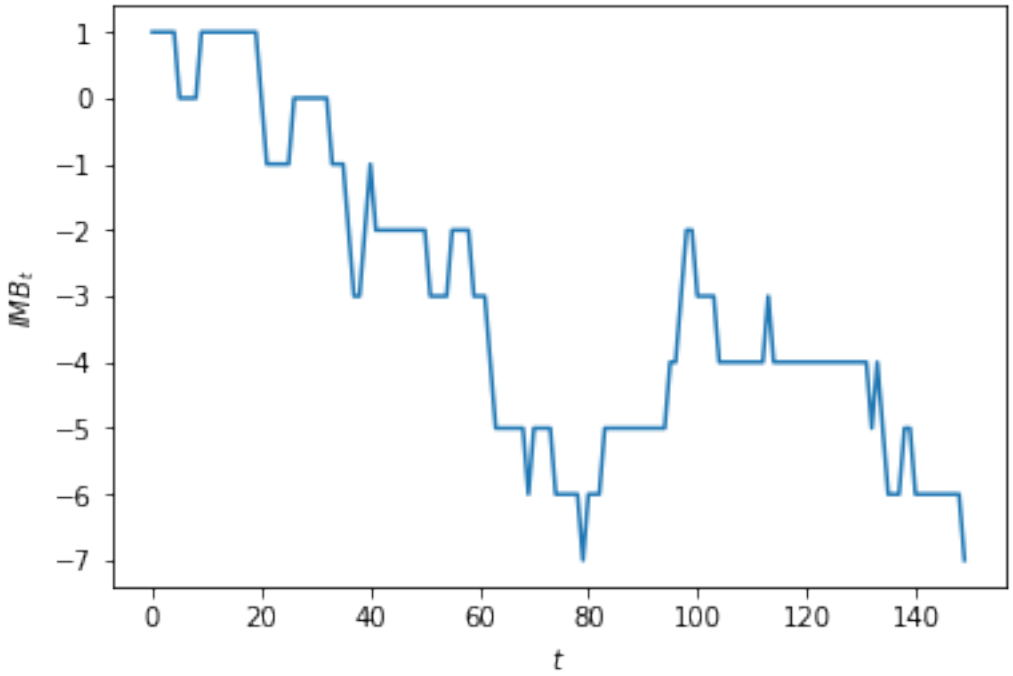
2.3.13 Uninformed traders

```
def simulate_uninformed_orders(events):
    orders = np.zeros(MAX_T)
    orders[events == EVENT_UNINFORMED_BUY] = +1
    orders[events == EVENT_UNINFORMED_SELL] = -1
    return orders

uninformed_orders = simulate_uninformed_orders(events)
uninformed_orders[:10]
```

```
array([ 1.,  0.,  0.,  0.,  0., -1.,  0.,  0.,  0.,  1.])
```

2.3.14 Uninformed order-imbalance

```
plt.plot(np.cumsum(uninformed_orders))
plt.xlabel('$t$'); plt.ylabel('$IMB_t$')
plt.show()
```



2.3.15 Informed traders

```
def informed_strategy(current_price, mm_price):
    if current_price > mm_price:
        return 1
    elif current_price < mm_price:
        return -1
    else:
        return 0
```

2.3.16 A simple market-making strategy

- Initially we consider a very simple policy for our market-making agent.
- The policy π_h is parameterised by a single threshold parameter h.
  - Increase the price by a single tick if the order-imbalance is $+h$.
  - Decrease the price by a single tick if the order-imbalance is $-h$.

```
def mm_threshold_strategy(order_imbalance, threshold=2):
    if order_imbalance == -threshold:
```

```
        return -1
    elif order_imbalance == +threshold:
        return +1
    else:
        return 0
```

2.3.17 The reward function

```
def mm_reward(current_fundamental_price, mm_current_price,
    ↪ order_sign):
    if order_sign < 0:
        return current_fundamental_price - mm_current_price
    elif order_sign > 0:
        return mm_current_price - current_fundamental_price
    else:
        return 0
```

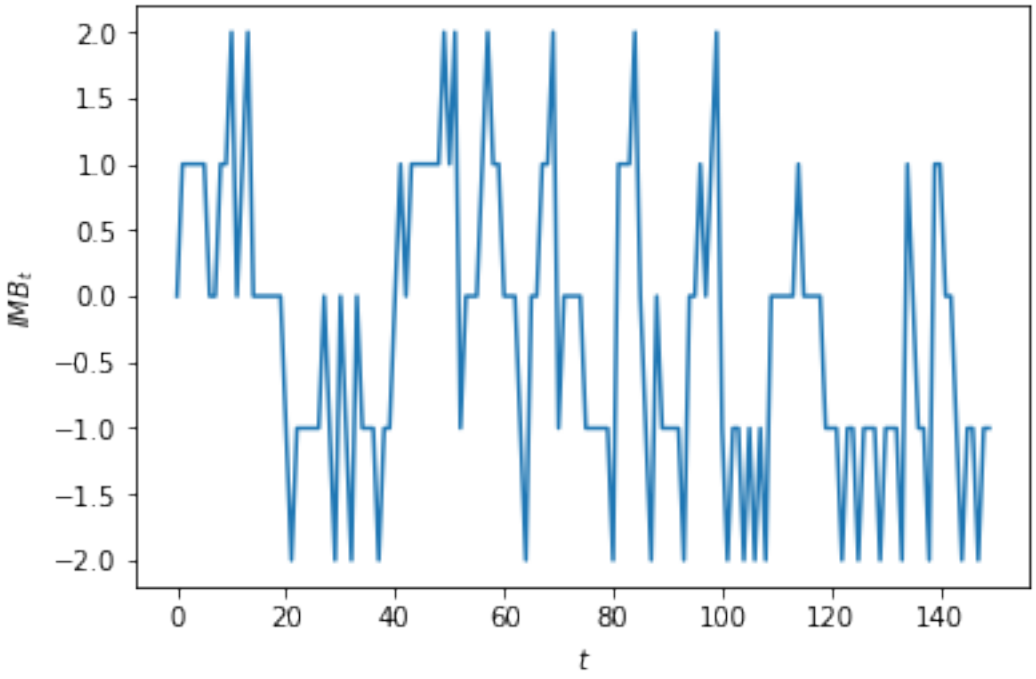
2.3.18 The market simulation

```
def simulate_market(events, uninformed_orders, fundamental_price,
                        mm_strategy=mm_threshold_strategy,
    ↪ threshold=1):

    mm_prices = np.zeros(MAX_T); order_imbalances = np.zeros(MAX_T)
    informed_orders = np.zeros(MAX_T); rewards = np.zeros(MAX_T);
    actions = np.zeros(MAX_T)
    t_mm = 0; mm_current_price = INITIAL_PRICE

    for t in range(MAX_T):

        if events[t] == EVENT_INFORMED_ARRIVAL:
            order = informed_strategy(fundamental_price[t],
    ↪ mm_current_price)
            informed_orders[t] = order
        else:
            order = uninformed_orders[t]

        imbalance = np.sum(informed_orders[t_mm:t] +
    ↪ uninformed_orders[t_mm:t])

        mm_price_delta = mm_strategy(imbalance, threshold)
        if mm_price_delta != 0:
            t_mm = t
            mm_current_price += mm_price_delta

        order_imbalances[t] = imbalance; mm_prices[t] =
    ↪ mm_current_price
        actions[t] = mm_price_delta;
        rewards[t] = mm_reward(fundamental_price[t],
    ↪ mm_current_price, order)

    return mm_prices, order_imbalances, rewards, actions
```

2.3.19 Order-imbalance time series

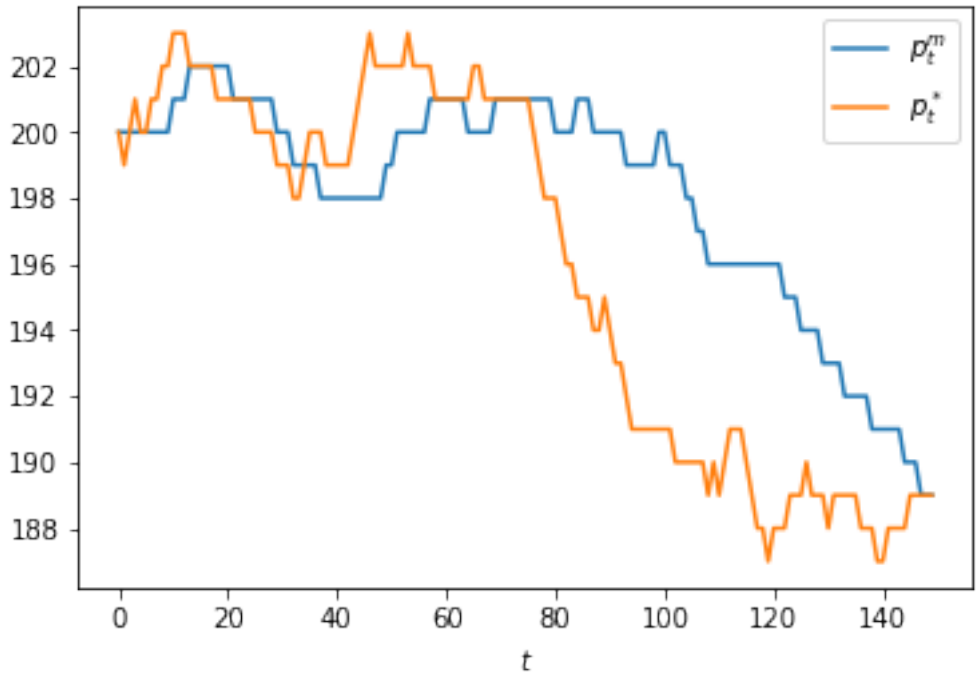
```
mm_prices, order_imbalances, rewards, actions = \
    simulate_market(events, uninformed_orders, fundamental_price,
    ↪ threshold=2)
```

```
plt.plot(order_imbalances); plt.xlabel('$t$'); plt.ylabel('$IMB_t$'
    ↪ )
plt.show()
```



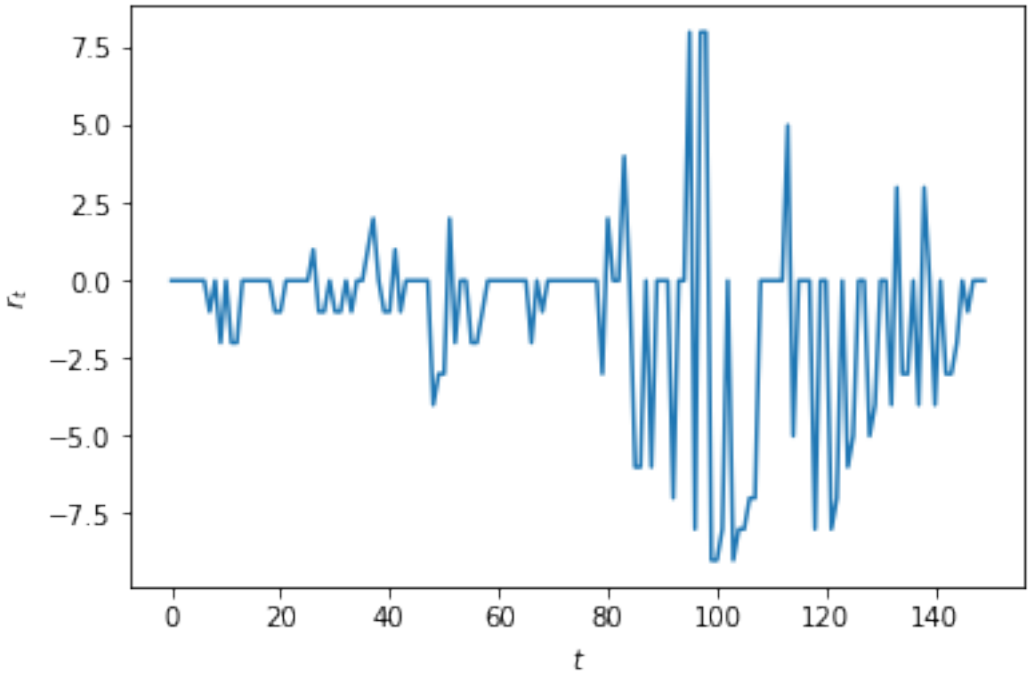
2.3.20 Price time series

```
plt.plot(mm_prices); plt.plot(fundamental_price)
plt.xlabel('$t$'); plt.legend(['$p^m_t$','$p^*_t$'])
plt.show()
```



2.3.21 Reward time series r_t

```
plt.plot(rewards)
plt.xlabel('$t$'); plt.ylabel('$r_t$')
plt.show()
```



2.4 Policy evaluation

- Each value of the threshold parameter h defines a policy π_h.
- In the experiments above we set $h = 2$.
- We can estimate the expected reward, and hence the expected return, across the episode.
- With no time-discounting, i.e. $\gamma = 1$, the return G is approximately:

```
np.mean(rewards)*MAX_T
```

```
-
165.0
```

- We could use this as an estimate of the expected return to the policy $E(G)$.
- However, we have only used a single sample.

2.5 Monte-carlo policy evaluation

- We will attempt to obtain more accurate value estimates using Monte-Carlo estimation.
- First we restructure our code so that we can easily rerun an entire simulation with given parameters.

```
def simulate_all(mm_strategy=mm_threshold_strategy, threshold=1,
                     probabilities=ALL_PROB):

    events = simulate_events(probabilities)
    fundamental_price = simulate_fundamental_price(events)
    uninformed_orders = simulate_uninformed_orders(events)

    return simulate_market(events, uninformed_orders,
    ↪ fundamental_price, mm_strategy, threshold)
```

```
mm_prices, order_imbalances, rewards, actions = simulate_all(
   ↪ threshold=1)
np.mean(rewards)
```

```
-
0.36
```

2.5.1 Using the sample-mean to estimate expected value

- We realise the model many times.
- Each realisation is a single episode or trajectory.
- We can consider each episode as a sample.
- We use the sample mean as the best estimator for the expectation.

```
def evaluate(policy, probabilities=ALL_PROB, samples=1000):
    return np.mean([np.mean(simulate_all(threshold=policy)[2])
                        for i in range(samples)])
```

2.5.2 Comparing policies

$v(\pi_1) \approx$

```
evaluate(policy=1)
```

```
-
0.4362533333333333
```

$v(\pi_2) \approx$

```
evaluate(policy=2)
```

```
-
0.5420799999999999
```

$v(\pi_3) \approx$

```
evaluate(policy=3)
```

```
-
0.6780799999999999
```

2.5.3 The value of states

- We can also estimate the value of a given state s assumming a fixed policy π
- For a threshold $h = 2$, i.e. $\pi = \pi_2$ the states are $s \in \{-2, -1, 0, +1, +2\}$.
- We first simulate a single episode.

```
mm_prices, order_imbalances, rewards, actions = simulate_all(
    ↪ threshold=2)
```

```
np.mean(rewards)
```

```
-
0.54
```

2.5.4 The value of states

- Now we estimate $v_{\pi_2}(s) \approx \bar{r}$ for those rewards obtained in the given state:

```
value_fn = {(state, np.mean(rewards[order_imbalances == state])) \
                    for state in [-2, -1, 0, +1, +2]}
value_fn
```

```
{(-2, -0.5833333333333334),
 (-1, -0.631578947368421),
 (0, -0.34782608695652173),
 (1, -0.4878048780487805),
 (2, -1.0769230769230769)}
```

- It's more elegant to represent this as a Python dictionary:

```
value_dict = dict(value_fn)
```

To compute $v_{\pi_2}(1)$ we can use the following

```
value_dict[1]
```

```
-
0.4878048780487805
```

2.5.5 The value of states

- Note that in the previous slide our estimate was based on a single episode.
- Below we extend the code on the previous slide to average over many episodes (samples).

```
def expected_reward_by_state(mm_strategy, threshold,
                                probabilities=ALL_PROB, samples
    ↪ =1000):

    states = range(-threshold, threshold+1)
    result = np.zeros((samples, len(states)))

    for i in range(samples):
        rewards = simulate_all(mm_strategy, threshold)[2]
        result[i, :] = [np.mean(rewards[order_imbalances == state])
    ↪  \
                            for state in states]

    return dict(zip(states, np.nanmean(result, axis=0)))
```

2.5.6 Policy comparision

$v_{\pi_1}(s) \approx$

```
expected_reward_by_state(mm_threshold_strategy, threshold=1)
```

```
{-1: -0.4366842105263162, 0: -0.4183260869565217, 1: -
0.4551463414634147}
```

$v_{\pi_2}(s) \approx$

```
expected_reward_by_state(mm_threshold_strategy, threshold=2)
```

```
{-2: -0.5339999999999994,
 -1: -0.5252894736842114,
 0: -0.5089347826086954,
 1: -0.5504634146341455,
 2: -0.5315384615384616}
```

2.5.7 The value of state action *pairs*

- Ideally we would like to compute $v_\pi(s,a)$.

```
def q_table(result, all_actions, all_states):
    return pd.DataFrame(result,
                        columns=["$\Delta p=%s$" % a for a in
    ↪ all_actions],
                        index=all_states)
```

```
def expected_reward_by_state_action(mm_strategy, threshold,
                                    probabilities=ALL_PROB,
    ↪ samples=1000):

    all_states = range(-threshold, threshold+1)
    all_actions = [-1, 0, +1]
    result = np.zeros((samples, len(all_states), len(all_actions)))

    for i in range(samples):

        _, states, rewards, actions = simulate_all(mm_strategy,
    ↪ threshold)

        result[i, :, :] = \
            np.reshape(
                [np.nanmean(rewards[(states == state) &
                                    (actions == action)]) \
                    for state in all_states for action in
    ↪ all_actions],
                (len(all_states), len(all_actions)))

    return np.nanmean(result, axis=0)
```

2.5.8 The value function obtained from π_2

```
Q = expected_reward_by_state_action(mm_threshold_strategy,
    ↪ threshold=2, samples=10000)
q_table(Q, [-1, 0, +1], range(-2, 3))
```

```
   $\Delta p=-1$  $\Delta p=0$  $\Delta p=1$-
2       -0.38593           NaN           NaN-
1            NaN     -0.567512           NaN
 0           NaN      -0.564312           NaN
 1           NaN      -0.567192           NaN
 2           NaN            NaN     -0.386951
```

2.5.9 Exploration of the state-space

```
def mm_exploration_strategy(order_imbalance, threshold=2, epsilon
    ↪ =0.025):
    if np.random.random() <= epsilon:
        return np.random.choice([-1, 0, +1])
    else:
        if order_imbalance == +threshold:
```

```
            return -1
        elif order_imbalance == -threshold:
            return +1
        else:
            return 0
```

2.5.10 Results from Monte-Carlo policy evaluation

```
Q = expected_reward_by_state_action(mm_exploration_strategy,
    ↪ threshold=2, samples=50000)
```

```
q_table(Q, [-1, 0, +1], range(-2, 3))
```

```
    $\Delta p=-1$  $\Delta p=0$  $\Delta p=1$-
2       -5.393120     -6.398448     -5.971927-
1       -5.820319     -6.092052     -6.168338
 0       -5.823990     -5.799513     -5.802399
 1       -6.240433     -6.094727     -5.838191
 2       -5.945665     -5.921463     -5.777265
```

The greedy policy:

```
dict({(s, np.where(Q[s+2, :] == np.max(Q[s+2, :]))[0][0] - 1) for s
    ↪ in range(-2, 3)})
```

```
{-2: -1, -1: -1, 0: 0, 1: 1, 2: 1}
```

2.6 Policy improvement

- We have estimated $Q_\pi(s,a) = \hat{v}_\pi(s,a)\ \forall s \in \mathbb{S}\ \forall a \in \mathbb{A}$ for a given policy π.
- If our estimates are accurate, we can use the function Q to find a better policy.
- To improve the policy we can simply take the greedy action for a given state:
  $\pi'(s) = \text{argmax}_a\, Q_\pi(s,a)$
- For the corresponding proof, see the *policy improvement theorem*.

2.7 Policy iteration

- Note, however, that if we change the policy from π to π', that our value estimates are outdated.
- Value estimates $Q_\pi(s,a)$ are obtained from following a given policy π.
- Value estimates and policies are not independent of each other.
- Therefore we should re-estimate $Q'_{\pi'}(s,a)$ for the new policy as $\hat{v}_{\pi'}(s,a)$
- We iterate until the policy and value estimates converge:

$$\pi(s,a) \to Q_\pi(s,a) \to \pi'(s,a) \to Q'_{\pi'}(s,a) \to \pi''(s,a) \ldots \tag{2.4}$$

2.7.1 Policy iteration figure

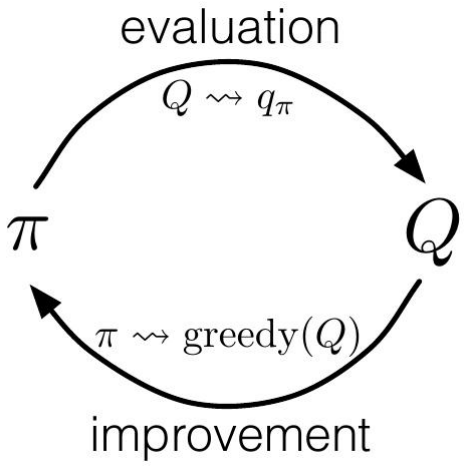


Figure 2.1: gpi

2.8 Generalised Policy Iteration

- Many reinforcement-learning interleaving policy improvement and value estimation;
  - improve policy based on incomplete samples, while
  - improving value estimates by following a (sub-optimal) policy.
- These techniques are called Generalised Policy Iteration (GPI) methods.

2.9 Temporal-difference (TD) learning

Recall that:

$$\begin{aligned} v_\pi(s) &= E_\pi[G_t|s_t = s] && (2.5)\\ &= E_\pi[r_{t+1} + \gamma G_{t+1}|s_t = s] && (2.6)\\ &= E_\pi[r_{t+1} + \gamma v_\pi(s_{t+1})|s_t = s] && (2.7)\\ & && (2.8) \end{aligned}$$

- When we use Monte-Carlo methods, G_t is unknown, so we estimate G_t from sampled returns.
- When we use dynamic-programming $v_\pi(s_{t+1})$ is unknown, so we use our existing estimate $\hat{v}_\pi(s_{t+1}) = V(s_{t+1})$ instead.
- Temporal difference learning combines both estimates:

$$V(s) \leftarrow V(s) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \tag{2.9}$$

- We can generalise this to Q:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.10}$$

2.10 Initialising the Q table

- We first define our action space $\Delta p_t \in \mathbb{A}$ and state space $\text{IMB}_t \in \mathbb{S}$

```
all_actions = [-1, 0, +1]
all_states = range(-2, +3)
```

- We will use a matrix to represent our current estimates Q

```
def initialise_learner():
    return np.zeros((len(all_states), len(all_actions)))
```

```
Q = initialise_learner()
q_table(Q, all_actions, all_states)
```

```
    $\Delta p=-1$  $\Delta p=0$  $\Delta p=1$-
2             0.0           0.0           0.0-
1             0.0           0.0           0.0
 0            0.0           0.0           0.0
 1            0.0           0.0           0.0
 2            0.0           0.0           0.0
```

2.11 Functions to manipulate Q

- We define the following functions to map from the state and action space into indices of the matrix.

```
def state(imbalance, all_states=range(-2, +3)):
    s = int(imbalance) - all_states[0]
    ms = len(all_states)-1
    if s > ms:
        return ms
    elif s < 0:
        return 0
    else:
        return s

def action(price_delta):
    return int(price_delta) + 1
```

- We then define functions to obtain Q values from specified actions and states

```
def q_values(Q, imbalance):
    return Q[state(imbalance), :]
```

```
def q_value(Q, imbalance, price_delta):
    return Q[state(imbalance), action(price_delta)]
```

2.12 Temporal-difference learning in Python

We can translate the following equation

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.11}$$

into Python:

```
def update_learner(s, a, r, s_, a_, Q, alpha=0.01, gamma=0.0):
        Q[state(s), action(a)] += alpha * (r + gamma * q_value(Q,
    ↪ s_, a_) - q_value(Q, s, a))
```

- Now we must modify our simulation code to provide feedback to the agent.

2.13 The market-maker as a reinforcement-learning agent

```
def simulate_learning_mm(Q, events, uninformed_orders,
    ↪ fundamental_price,
                             mm_policy):
    mm_prices = np.zeros(MAX_T, dtype=int); order_imbalances = np.
    ↪ zeros(MAX_T, dtype=int)
    informed_orders = np.zeros(MAX_T, dtype=int); rewards = np.
    ↪ zeros(MAX_T);
    actions = np.zeros(MAX_T, dtype=int);  mm_t_last_change = 0
    mm_current_price = INITIAL_PRICE
    for t in range(MAX_T):
        if events[t] == EVENT_INFORMED_ARRIVAL:
            order = informed_strategy(fundamental_price[t],
    ↪ mm_current_price)
            informed_orders[t] = order
        else:
            order = uninformed_orders[t]
        imbalance = np.sum(informed_orders[mm_t_last_change:t] +
                           uninformed_orders[mm_t_last_change:t])
        mm_price_delta = mm_policy(imbalance)
        if mm_price_delta != 0:
            mm_t_last_change = t
            mm_current_price += mm_price_delta
        order_imbalances[t] = imbalance; mm_prices[t] =
    ↪ mm_current_price
        actions[t] = mm_price_delta;
        rewards[t] = mm_reward(fundamental_price[t],
    ↪ mm_current_price, order)
        if t>0:
            update_learner(order_imbalances[t-1], actions[t-1],
    ↪ rewards[t-1],
                           imbalance, mm_price_delta, Q)
    return fundamental_price, mm_prices, order_imbalances, rewards,
    ↪  actions, Q
```

2.14 On-policy control

- Now we can combine policy improvement and policy estimation in a single step.
- This algorithm is called SARSA, which is named after the arguments to the function `update_learner`.
- We use TD learning to bootstrap Q values, and then form an ϵ-greedy policy using our value estimates.

```
def mm_learning_strategy(Q, s, epsilon=0.1):
    if np.random.random() <= epsilon:
        action = np.random.choice([-1, 0, +1])
```

```
    else:
        values = q_values(Q, s)
        max_value = np.max(values)
        action = np.random.choice(np.where(values == max_value)[0])
    ↪  - 1
    return action
```

```
def simulate_learning(Q, probabilities=ALL_PROB):

    events = simulate_events(probabilities)
    fundamental_price = simulate_fundamental_price(events)
    uninformed_orders = simulate_uninformed_orders(events)

    def sarsa(s):
        return mm_learning_strategy(Q, s)

    return simulate_learning_mm(Q, events, uninformed_orders,
    ↪ fundamental_price, mm_policy=sarsa)
```

2.15 Learning over a single episode

```
Q = initialise_learner()
```

```
fundamental_price, mm_prices, order_imbalances, rewards, actions, Q
    ↪  = simulate_learning(Q)
```

```
q_table(Q, all_actions, all_states)
```

```
    $\Delta p=-1$  $\Delta p=0$  $\Delta p=1$-
2        0.000000      0.000000       0.000000-
1       -0.401995     -0.266438      -0.260303
 0       -0.629396     -0.570168      -0.329424
 1       -0.764471     -0.882239      -0.889121
 2       -0.377703     -0.503057      -0.215830
```

2.16 Learning over many episodes

- We simply iterate over many trading days (i.e. episodes) in order to gradually learn the optimal policy.
- Each episode is independent, but notice that we re-use the Q-values from the previous episode.
- This ensures that we learn across episodes.

```
EPISODES = 5000

for i in range(EPISODES):
    fundamental_price, mm_prices, order_imbalances, rewards,
    ↪ actions, Q = simulate_learning(Q)
```

2.17 The results

The learned Q values:

```
q_table(Q, all_actions, all_states)
```

```
    $\Delta p=-1$  $\Delta p=0$  $\Delta p=1$-
2       -0.908272     -1.945495     -1.884303-
1       -0.404848     -0.941013     -1.036575
 0       -0.945292     -0.496658     -0.874284
 1       -1.331283     -1.248187     -0.343388
 2       -2.161748     -2.306827     -1.399997
```

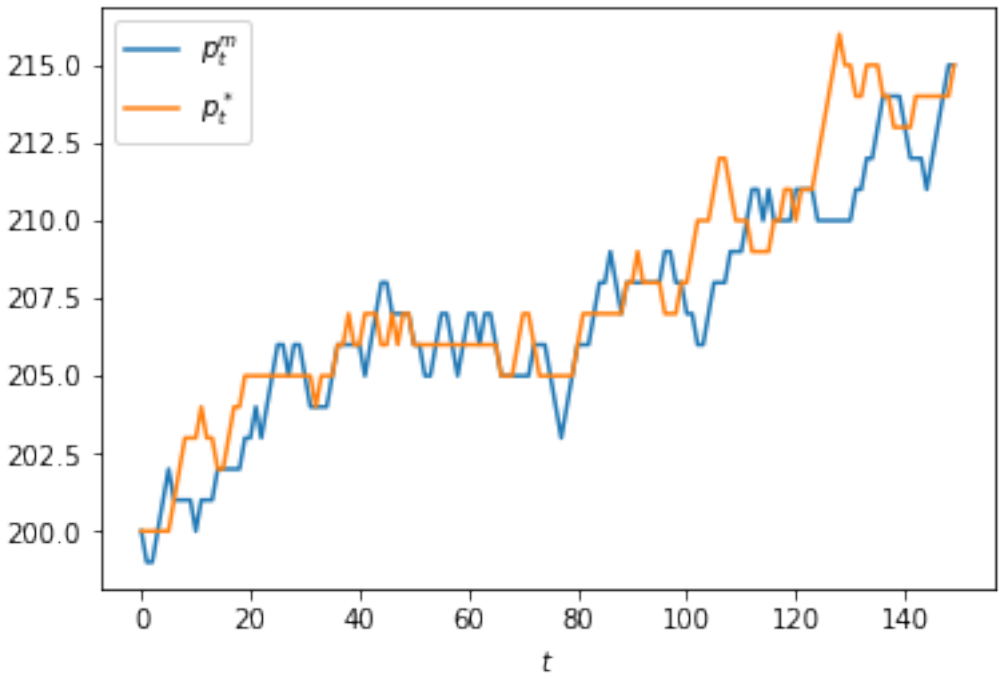
The greedy policy:

```
{(s, np.where(Q[state(s), :] == np.max(Q[state(s), :]))[0][0] - 1)
    for s in all_states}
```

```
{(-2, -1), (-1, -1), (0, 0), (1, 1), (2, 1)}
```

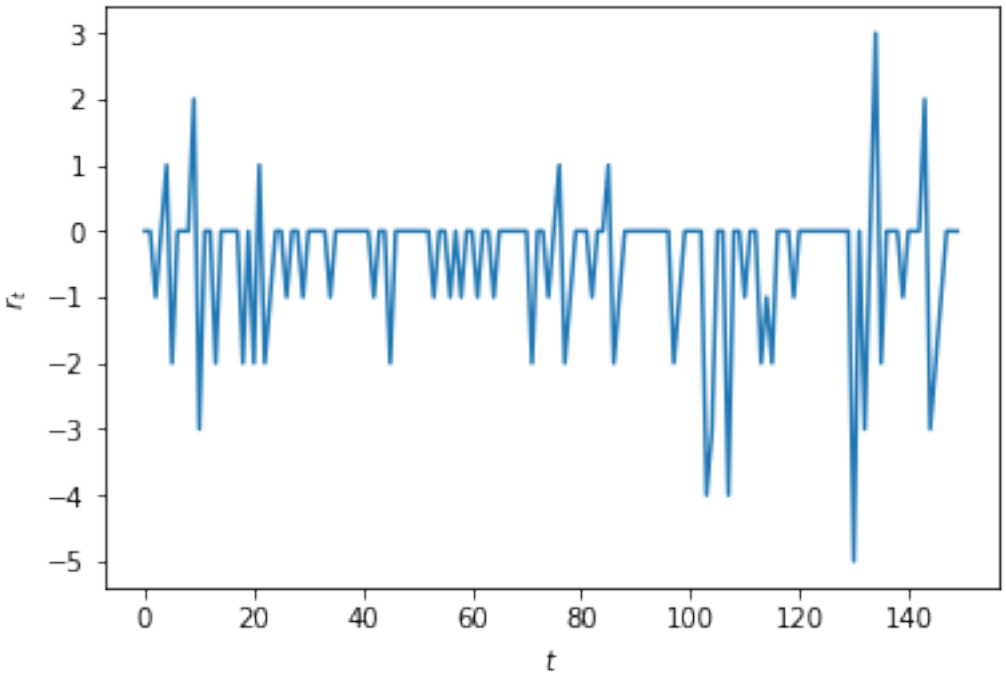
2.18 The prices from the final trading day

```
plt.plot(mm_prices); plt.plot(fundamental_price)
plt.xlabel('$t$'); plt.legend(['$p^m_t$','$p^*_t$'])
plt.show()
```



2.19 The rewards in the final day

```
plt.plot(rewards)
plt.xlabel('$t$'); plt.ylabel('$r_t$')
plt.show()
```



2.20 Conclusion

- We have implemented a very simple market-making strategy in a simplified model of a financial market.
- The framework is very flexible, and can be extended to more realistic applications, e.g. separate bid and ask qutes.
- We can use a simulation model to initially train our agent.
- However, the agent can learn directly from the environment in the absence of a model.
- Reinforcement-learning can be very useful in reducing model-risk.

3 Bibliography

Chan, N. T., & Shelton, C. (2001). An electronic market-maker.

Ganchev, K., Nevmyvaka, Y., Kearns, M., & Vaughan, J. W. (2010). Censored exploration and the dark pool problem. Communications of the ACM, 53(5), 99-107.

Mani, M., Phelps, S., & Parsons, S. (2019). Applications of Reinforcement Learning in Automated Market-Making.

Sutton, R. S., & Barto, A. G. (2011). Reinforcement learning: An introduction.

3.1 Further reading on agent-based modeling

Farmer, J. D., & Foley, D. (2009). The economy needs agent-based modelling. Nature, 460(7256), 685-686.

Lo, A. W. (2004). The adaptive markets hypothesis. The Journal of Portfolio Management, 30(5), 15-29.

Phelps, S. (2012). Applying dependency injection to agent-based modeling: the JABM toolkit. WP056-12, Centre for Computational Finance and Economic Agents (CCFEA), Tech. Rep.

Tesfatsion, L., & Judd, K. L. (Eds.). (2006). Handbook of computational economics: agent-based computational economics. Elsevier.

3.2 Links to software toolkits

- JABM
- JASA