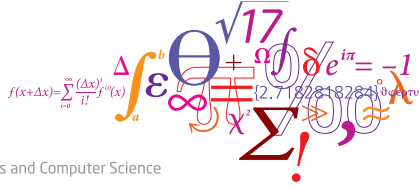**DTU**

02465: Introduction to reinforcement learning and control

The finite-horizon decision problem

Tue Herlau

DTU Compute, Technical University of Denmark (DTU)

$$f(x+\Delta x)=\sum_{i=0}^{\infty}\frac{(\Delta x)^i}{i!}f^{(i)}(x)$$

DTU Compute
Department of Applied Mathematics and Computer Science

DTU

Lecture Schedule

Dynamical programming

❶ **The finite-horizon decision problem**
31 January
❷ Dynamical Programming
7 February
❸ DP reformulations and introduction to Control
14 February

Control

❹ Discretization and PID control
21 February
❺ Direct methods and control by optimization
28 February
❻ Linear-quadratic problems in control
7 March
❼ Linearization and iterative LQR
14 March

Reinforcement learning

❽ Exploration and Bandits
21 March
❾ Policy and value iteration
4 April
❿ Monte-carlo methods and TD learning
11 April
⓫ Model-Free Control with tabular and linear methods
18 April
⓬ Eligibility traces and value-function approximations
25 April
⓭ Q-learning and deep-Q learning
2 May

Syllabus: https://02465material.pages.compute.dtu.dk/02465public
Help improve lecture by giving feedback on DTU learn

DTU

Reading material:

- [Her24, Chapter 4] Introduction

Learning Objectives

- Introduction and key definitions
- Python and object-oriented programming

---

cweb

Course webpage

`02465material.pages.compute.dtu.dk/02465public/index.html`

Where and what

DTU Learn Announcements, assignment hand-ins, quizzes

Course homepage Exercises, projects, slides, documentation, installation, etc. `https://02465material.pages.compute.dtu.dk/02465public`

Off-hours QA Discord. See link on homepage.

- Exercises
  - (Not confirmed) Building B341, auditorium 21
- Ask **project-related question** online so that everyone has the same information (i.e. not in class)
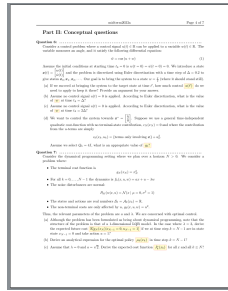

Project work

- Groups of 1, 2 or 3 students
  - Part 1 Dynamical programming **(available now)**
  - Part 2 Control
  - Part 3 Reinforcement Learning
- The projects are subject to DTUs rules of collaboration/Code of Conduct
  - This includes the individual programming.

Exam

- The 4-hour written exam will contain:
  - Multiple-choice questions
  - Written-answer questions
  - Programming questions
- Test exams will be online later
- Exercises emphasize code-questions as I believe they test more skills
- Your evaluation is an overall assessment based on the written exam and project work
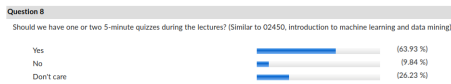 - The project work is 20%.

Creating handins
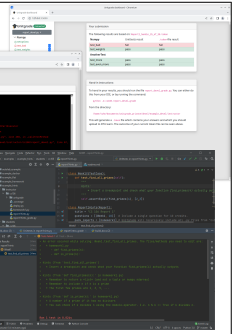
See videos for week 0

- I hope this can help you debug code
- Example usage:
  - `python -m irlc.project0.fruit_project_grade`
  - Hand in your code/scores by uploading the `.token` file

Quiz 0: answer on DTU Learn

Question 8

Should we have one or two 5-minute quizzes during the lectures? (Similar to 02450, introduction to machine learning and data mining)

Yes	(63.93 %)
No	(9.84 %)
Don't care	(26.23 %)

I will try to use quizzes this semester. You can find them under Quizzes on DTU Learn:

Do you use ChatGPT or a similar conversational AI tools in your studies?

- Yes
- No

ChatTutor

- ChatTutor allows you to ask questions to **both** TAs and an AI (ChatGPT)
- The platform will collect the data you put in (i.e., same as any other webpage!)
  - But please ask if you have questions!
- Optional offer:
  - Available from next week
  - Work in progress – you will have other options if it is too janky :-).

Types of machine learning

Supervised learning Learn a function $f(x_i) \mapsto \hat{y}_i$ to minimize a **loss**

Unsupervised learning Learn a **structure** to **summarize data**

Sequential decision making
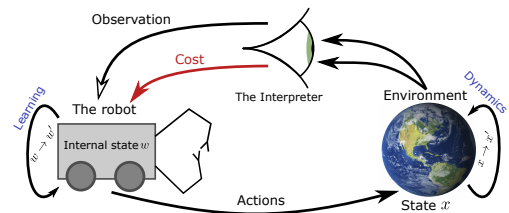
Make decisions, one after another, to bring about a desired outcome

- Observe the world
- Take action
- Obtain cost

Minimize total cost

🎮 `lecture_01_pacman.py`

- Time is really important (sequential, non-i.i.d data)
- Must optimize behavior of dynamical systems using information that becomes progressively available as the systems evolve
- Future cost and state of the system will depend on current actions and state

---

Alpha-Go

- Self-learning Go supercomputer
- Defeated world champion Lee Sedol in 2016
- Notable mentions: Atari/Dota/Starcraft II learners
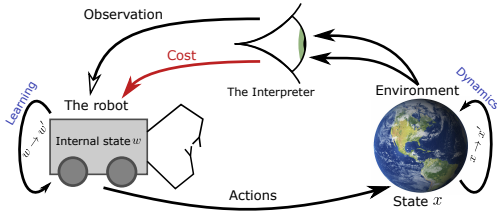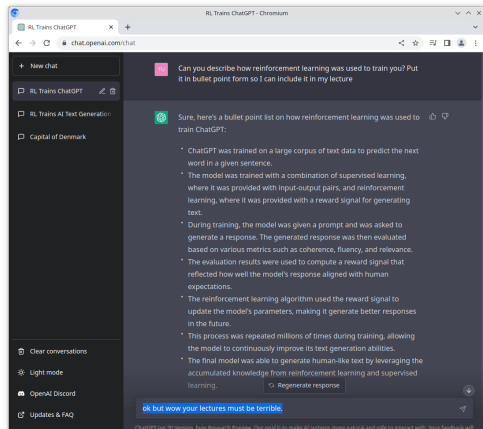- General approach: Reinforcement learning + Search trees

---

ChatGPT

The decision problem
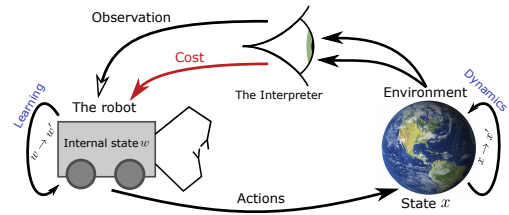
State The configuration of the environment x

Action Either discrete or a vector u

Cost/reward A number. Depends on state x and action u

Example: Mars landing

Time Continuous

State/Actions $x(t)$: (Position, velocity, fuel mass) $u(t)$: thruster outputs

Dynamics Smooth differential equation

$$\dot{x}(t) = f(x(t), u(t))$$

Cost Land the right place **and** use little fuel **and** don't kill anyone

Constraints Thrusters deliver limited force, ship cannot go into mars, etc.

Objective Determine $u(t)$ to minimize final cost

Really important constraints; no learning

🎮 `lecture_01_car_random.py`

Inventory control
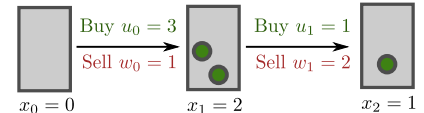
- We order a quantity of an item at period $k = 0, \ldots, N-1$ so as to meet a stochastic demand

  x_k stock at the beginning of the kth period,
  $u_k \geq 0$ stock ordered at the beginning of the kth period.
  $w_k \geq 0$ Demand during the k'th period

- Dynamics: $x_{k+1} = x_k + u_k - w_k$
- Cost per new unit c; cost to hold x_k units is $r(x_k)$

$$r(x_k) + cu_k$$

- Select actions $u_0, \ldots, u_{N-1}$ to minimize cost

We want proven optimal rule for ordering

Example: Atari

States RAM memory state

Observations Pixel-based snapshots $H \times W \times 3$

Actions Discrete joystick actions

Dynamics Discrete, stochastic (what the emulator does)

Cost High-score

Don't know dynamics; must learn from scratch
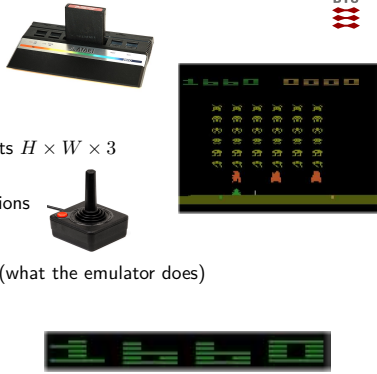
The environment
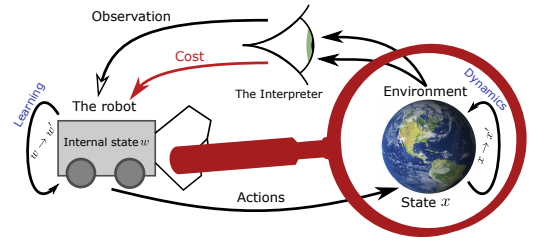
- Nature can be stochastic or deterministic
- The problem can be continuous-time or discrete-time
- We can know the dynamics or not

The agent
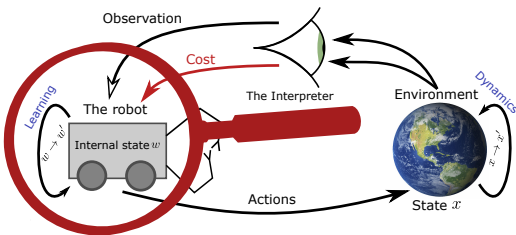
Policy How the robot chooses actions at given times/states

The interpreter
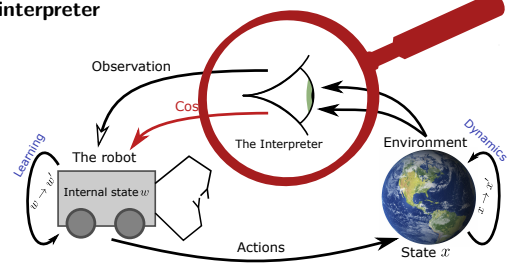
Reward The **immediate** evaluation of current step

Agents goal Maximize **cumulative** reward

Reward Hypothesis

Every desired behavior of the agent can be described by the maximization of expected cumulative reward

---

Making sense of these distinctions

- Why so many things in one course?
  - Study-line requirement
  - A single problem, and a single solution + tricks
  - A better overview (right tool for the job)
- Today, we will look at the problem

Basic control setup: Environment dynamics

Finite time Problem starts at time 0 and terminates at time N. Indexed as $k = 0, 1, \ldots, N$.

State space The states x_k belong to the **state space** $\mathcal{S}_k$

Control The available controls u_k belong to the **action space** $\mathcal{A}_k(x_k)$, which may depend on x_k

Dynamics
$$x_{k+1} = f_k\left(x_k, u_k, w_k\right), \quad k = 0, 1, \ldots, N-1$$

Disturbance/noise A random quantity w_k with distribution

$$w_k \sim P_k(W_k | x_k, u_k)$$

Cost and control

Agent observe x_k, agent choose u_k, environment generates w_k

Cost At each stage k we obtain cost

$$g_k(x_k, u_k, w_k), \quad k = 0, \ldots, N-1 \quad \text{and} \quad g_N(x_k) \text{ for } k = N.$$

Action choice Chosen as $u_k = \mu_k(x_k)$ using a function $\mu_k : \mathcal{S}_k \to \mathcal{A}_k(x_k)$

$$\mu_k(x_k) = \{\text{Action to take in state } x_k \text{ in period } k\}$$

Policy The collection $\pi = \{\mu_0, \mu_1, \ldots, \mu_{N-1}\}$

Rollout of policy Given x_0, select $u_k = \mu_k(x_k)$ to obtain a **trajectory** $x_0, u_0, x_1, \ldots, x_N$ and **accumulated cost**

$$\text{Cost-of-rollout} = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)$$

Expected return (approximate) Generate T rollouts according to π

$$J_\pi(x_0) \approx \frac{1}{T} \sum_{i=1}^{T} \{\text{Cost-of-rollout } i\}$$

Quiz 1: Discuss and answer on DTU Learn

How do you feel about this argument? Justify your answer:

Decision-making is about determining the appropriate sequence of actions $u_0, \ldots, u_{N-1}$.

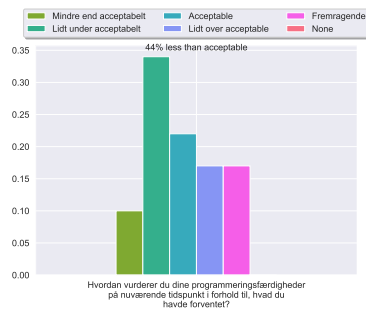Once executed, we get a total cost. Let's say that on average this is $c(\mathbf{u})$. Thus, decision-making is ultimately an optimization problem: Find the sequence that on average minimize the cost:
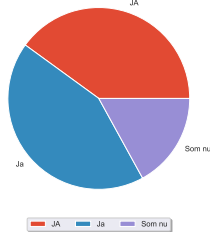
$$u_0, \ldots, u_{N-1} = \arg\min_{\mathbf{u}} c(\mathbf{u}).$$

a. It is computationally too complicated to solve such an optimization problem

b. It is infeasible to derive or learn the function $c(\mathbf{u})$

c. Actually nothing is wrong: It is just not a theoretically interesting/fruitful way to approach decision-making

d. Something else is wrong with the argument

e. Don't know

Programming: From KID study line evaluation

Hvordan vurderer du dine programmeringsfærdigheder på nuværende tidspunkt i forhold til, hvad du havde forventet?

Vil du gerne have mere programmering på 1. år?

Pre-semester quiz


```python
# chapter1/lecture1_code.py
class MyClass:
    def __init__(self, a):
        self.my_variable = a

    def some_function(self):
        print("The variable I got was", self.my_variable)

class MyOtherClass(MyClass):
    def __init__(self, a, b):
        super().__init__(a)
        print("I also got", b)
```

This is new -- I have not used class inheritance before. The code is mysterious.	(35.56 %)
I have seen code like this before, but it is not something I have used. I think I can pick it up.	(20 %)
I have written code that inherit from other classes (i.e., something like the second class). I am not an expert, but it is not something that worries me	(28.89 %)
This is easy. I have written code like this before and can reason about what it does.	(15.56 %)

Initiatives

What I have done:

- Re-structured the project work
- Simplification of exercises + videos
- Course notes on Python + online documentation
- This lecture
- Changed exam format
- Course responsible for the new mandatory programming course (02002/3)

What I hope you will do:

- Decide to learn this – you can!
- Set aside some time in the first block
- Don't give up:
  - Programming was not taught correctly – 100% valid criticism
  - You need to learn new programming techniques through your career

---

Pacman game loop (without objects)


```python
# chapter1/lecture1_code.py
walls = np.ndarray( ) # Initialize a walls-variable
food = np.ndarray( )
pacman_x = 4
pacman_y = 6

for k in range(10):
    # Use the walls and pacman_x, pacman_y to figure out what actions are available.
    available_actions = ... # compute using the walls-variable
    # Do some sort of planning (search?) by using the walls, pacman_x, pacman_y.
    # select the best possible action
    # Compute the outcome of the action:
    pacman_x = pacman_x + action_x
    pacman_y = pacman_y + action_y
    # Compute the reward
    # Let the agent learn based on the outcome and reward
```

(about 500 lines total)

Same with two agents and two environments

```python
# chapter1/lecture1_code.py
for k in range(10):
    if environment_type == 2:
        available_actions = ... # compute using the walls-variable
    else:
        available_actions = ... # This environment may differ
    if agent_type == 1: # Agent plan it's actions
        pass    # do planning of first type
    elif agent_type == 2:
        pass    # do planning of the second type
    if environment_type == 1: # Compute the outcome of the action:
        pacman_x = pacman_x + action_x
        pacman_y = pacman_y + action_y
        # Compute the cost-function
    else:
        pass # Updates relevant for second environment
        # Compute the cost function
    if agent_type == 2:  # Allow the agent to learn based on cost
        pass # Learning for the second agent
    else:
        pass # Learning method for the first agent
```

Using objects

```python
# chapter1/lecture1_code.py
env = InventoryEnvironment() # Create an instance of the inventory environment
agent = RandomAgent(env)     # Create an instance of a random-action agent
train(env, agent)            # Train the agent
```

Training-function:

```python
# chapter1/lecture1_code.py
def train(env, agent):
    s = env.reset()       # Reset and get first state, x_0
    for k in range(10):
        a = agent.pi(s) # The policy computes the action
        sp, r, done = env.step(a) # Environment computes next state, reward
        agent.train(s, a, sp, r, done)      # Let the agent train
```

(this is a very rough sketch. Well get to the real training function soon)

The simplest class

The smallest and friendliest `class`

```python
>>> class BasicClass:    # Classnames are usually upper-case
...     pass             # `pass` is a special keyword which does nothing
...
```

Each class **instance** function like it's own little box of variables:

```python
>>> a = BasicClass()   # Create an instance of the class
>>> a.name = "My first class" # You can write data to the class like this
>>> b = BasicClass()   # Another instance. a and b are not related and can store different data:
>>> b.name = "Another class"
>>>
>>> print("Class a:", a.name)
Class a: My first class
>>> print("Class b:", b.name)
Class b: Another class
```

A class with a function

```python
>>> class BasicDog:
...     name = "Unnamed dog"  # Each dog-instance will have the property name
...     def read_nametag(self):
...         # This is a class-function. Note we must pass it `self` as a first argument,
...         # instance of the class itself (i.e. the current object). This is how we ca
...         print("This dog is named", self.name, "please give me treats!")
...
>>> dog = BasicDog()
>>> dog.name
'Unnamed dog'
```

`self` refers to the class instance

```python
>>> dog.read_nametag()   # Invoke the read_nametag() function. Note we don't pass the ob
This dog is named Pluto please give me treats!
```

`def __init__` function is called when the class is created

```python
>>> class BetterBasicDog:
...     def __init__(self, name):
...         self.name = name
...         self.age = 0
...         print(f"The __init__() function has been called with name='{name}'")
...     def birthday(self):
...         self.age = self.age + 1
...         print("Hurray for", self.name, "you are now", self.age, "years old")
...
```

Arguments can be passed along like this

```python
>>> d1 = BetterBasicDog("Pluto")        # the __init__ function is now called
The __init__() function has been called with name='Pluto'
>>> d2 = BetterBasicDog(name="Lassie") # Also support named arguments
The __init__() function has been called with name='Lassie'
```

Functions can change the `state` of the class

```python
>>> d1.birthday()
Hurray for Pluto you are now 1 years old
>>> d1.birthday()
Hurray for Pluto you are now 2 years old
```

Quiz 2: What is the outcome of this code?

```python
>>> class BetterBasicDog:
...     def __init__(self, name):
...         self.name = name
...         self.age = 0
...         print(f"The __init__() function has been called with name='{name}'")
...     def birthday(self):
...         self.age = self.age + 1
...         print("Hurray for", self.name, "you are now", self.age, "years old")
...
>>> d1 = BetterBasicDog("Pluto")
The __init__() function has been called with name='Pluto'
```

```python
# chapter0pythonC/quiz.py
d1 = BetterBasicDog("Pluto")
d1.birthday()
d1.age = 5
d1.name = "Lassie"
d1.birthday()
```

a. Ignore changes and prints out `"Hurray for Pluto you are now 1 years old"`

b. Accept changes and prints out `"Hurray for Lassie you are now 6 years old"`

c. It gives an error – it is not possible to set the age.

d. It uses `name` but ignores `age`, so we get:
`"Hurray for Lassie you are now 1 years old"`

e. Don't know.

The parrot

```
1  >>> class Parrot:
2  ...     def __init__(self):
3  ...         self.words = ["Squack!"]
4  ...     def learn(self, word):
5  ...         self.words.append(word)
6  ...     def speak(self):
7  ...         return random.choice(self.words)  # Return a random word
8  ...     def vocabulary(self):
9  ...         return self.words
10 ...
```

```
1  >>> parrot = Parrot()
2  >>> words = ["sugar", "sleep well", "(parrot noises)", "*honk*"]
3  >>> for word in words:
4  ...     parrot.learn(word)
5  ...
6  >>> for _ in range(3):  # Say three words
7  ...     parrot.speak()
8  ...
9  '(parrot noises)'
10 'Squack!'
11 'sleep well'
12 >>> print("Vocabulary", parrot.vocabulary())
13 Vocabulary ['Squack!', 'sugar', 'sleep well', '(parrot noises)', '*honk*']
```

Inheritance

```
1  >>> class Parrot:
2  ...     def __init__(self):
3  ...         self.words = ["Squack!"]
4  ...     def learn(self, word):
5  ...         self.words.append(word)
6  ...     def speak(self):
7  ...         return random.choice(self.words)  # Return a random word
8  ...     def vocabulary(self):
9  ...         return self.words
10 ...
```

`ForgetfulParrot` : Is like the regular `Parrot` , except the learn-function

```
1  >>> class ForgetfulParrot(Parrot):
2  ...     # The Parot class is used as a template.
3  ...     # All functions in the Parot-class are therefore 'imported' as default, including 'self.words'
4  ...     def learn(self, word):  # This function overwrite the 'actual' learn function in the Parot class
5  ...         self.words = [word]  # This parrot only know a single word
6  ...
```

Inheritance: The functions are *"copy-pasted"* into the `ForgetfulParrot`

```
1  >>> old_parrot = ForgetfulParrot()
2  >>> old_parrot.learn("damn remote")
3  >>> old_parrot.learn("Jeopardy")
4  >>> print("Vocabulary", old_parrot.vocabulary())
5  Vocabulary ['Jeopardy']
```

Inheritance continued

More **inheritance**: Make a squeak before and after every word:

```
1  >>> class Parrot:
2  ...     def __init__(self):
3  ...         self.words = ["Squack!"]
4  ...     def learn(self, word):
5  ...         self.words.append(word)
6  ...     def speak(self):
7  ...         return random.choice(self.words)  # Return a random word
8  ...     def vocabulary(self):
9  ...         return self.words
10 ...
```

Where is the bug?

```
1  >>> class BadSqueekyParrot(Parrot):
2  ...     def __init__(self, squeek="Quck!"):
3  ...         self.squeek = squeek
4  ...     def speak(self):
5  ...         return f"{self.squeek} {random.choice(self.words)} {self.squeek}"
6  ...
7  >>> squeeky = BadSqueekyParrot(squeek="Kvak-Kvak")
8  >>> squeeky.learn("Good night!")
9  Traceback (most recent call last):
10   File "<console>", line 1, in <module>
11   File "<console>", line 5, in learn
12 AttributeError: 'BadSqueekyParrot' object has no attribute 'words'
```

Use `super()` to access functions in the **parent class**

```
1  >>> class SqueekyParrot(Parrot):
2  ...     def __init__(self, squeek="Quck!"):
3  ...         super().__init__()   # Call the 'Parot' class __init__ method to set up the words-variable.
4  ...         self.squeek = squeek  # save the squeek variable
5  ...     def speak(self):
6  ...         word = super().speak()  # Use the speak() function defined in the Parrot class
7  ...         return f"{self.squeek} {word} {self.squeek}"
8  ...
9  >>> squeeky = SqueekyParrot(squeek="Kvak-Kvak")
10 >>> squeeky.learn("Good night!")
11 >>> squeeky.learn("Tell that damn bird to shut it's beak")
12 >>> squeeky.learn("Sugar!")
13 >>> squeeky.speak()
14 'Kvak-Kvak Good night! Kvak-Kvak'
15 >>> squeeky.speak()
16 'Kvak-Kvak Sugar! Kvak-Kvak'
```

Consistency When we inherit from `Parrot` , we **know** the functions should be called `speak` , `learn` (and not `talk` , `practice`)
- `Env` : (`reset` , `step` , `action_space` and a few other)
- `Agent` : (`pi` , `train`)

Functionality By using `super().__init__` we saved a single line
- In control theory, we will use inheritance to add simulation-functionality to all models

The inventory environment

```
1  # inventory_environment.py
2  class InventoryEnvironment(Env):
3      def __init__(self, N=2):
4          self.N = N                                # planning horizon
5          self.action_space      = Discrete(3)      # Possible actions {0, 1, 2}
6          self.observation_space = Discrete(3)      # Possible observations {0, 1, 2}
7
8      def reset(self):
9          self.s = 0                                # reset initial state x0=0
10         self.k = 0                                # reset time step k=0
11         return self.s, {}                         # Return the state we reset to (and an
12
13     def step(self, a):
14         w = np.random.choice(3, p=(.1, .7, .2))   # Generate random disturbance
15         s_next = max(0, min(2, self.s-w+a))       # next state; x_{k+1} =  f_k(x_k,
16         reward = -(a + (self.s + a - w)**2)       # reward = -cost      = -g_k(x_k,
17         terminated = self.k == self.N-1           # Have we terminated? (i.e. is k=
18         self.s = s_next                           # update environment state
19         self.k += 1                               # update current time step
20         return s_next, reward, terminated, False, {}  # return transition information
```

Recall $x_{k+1} = x_k - w_k + a_k$ (clipped at 0 and 2) and e.g. $P(w = 0) = \frac{1}{10}$

The Agent:

```
1  # inventory_environment.py
2  class RandomAgent(Agent):
3      def pi(self, s, k, info=None):
4          """ Return action to take in state s at time step k """
5          return np.random.choice(3)  # Return a random action
```

- The policy $\mu_k(x_k)$ corresponding to `pi(x, k, info)`
- A training function which is given x_k, u_k and x_{k+1} plus obtained reward plus additional information
- In each exercise session, you will write at least one agent
- Look at the `Agent` -class
- `truncated=False` ; `info` is 'extra information' (see documentation)

The `train` -function

The train-function computes an episode as follows:

```python
# inventory_environment.py
def simplified_train(env: Env, agent: Agent) -> float:
    s, _ = env.reset()
    J = 0  # Accumulated reward for this rollout
    for k in range(1000):
        a = agent.pi(s, k)
        sp, r, terminated, truncated, metadata = env.step(a)
        agent.train(s, a, sp, r, terminated)
        s = sp
        J += r
        if terminated or truncated:
            break
    return J
```

Above computes the sum-of-reward for one episode:

```python
# inventory_environment.py
env = InventoryEnvironment()
agent = RandomAgent(env)
stats, _ = train(env,agent,num_episodes=1,verbose=False)  # Perform one rollout.
print("Accumulated reward of first episode", stats[0]['Accumulated Reward'])
```

Approximate value function

Approximate

$$J_\pi(x_0) = \mathbb{E}\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\right] \qquad (1)$$

As average over 1000 trajectories

```python
# inventory_environment.py
stats, _ = train(env, agent, num_episodes=1000,verbose=False)  # do 1000 rollouts
avg_reward = np.mean([stat['Accumulated Reward'] for stat in stats])
print("[RandomAgent class] Average cost of random policy J_pi_random(0)=", -avg_reward)
```
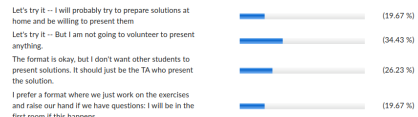

Quiz 3: Bobs friend

Bob has $x_0 = 20$ kroner. He can either:

- Action $u = 0$: Put them in the bank at a 10% interest, thereby ending up with 22 kroner.

- Action $u = 1$: Lend them to a friend.
    - With probability $\frac{1}{4}$ he looses everything
    - With probability $\frac{3}{4}$ his friend gives him 12 kroner (aka one beer) as a thank you, and thus he will have $20 + 12 = 32$ kroner total.

Bobs goal is to decide whether to put his money in the bank, or lend them to his friend. Which one of the following statements are correct:

a. The state spaces are $\mathcal{S}_k = \{1, 2, \ldots, 32\}$.

b. The dynamics is $f_0(x_0, u_0, w_0) = 1.1x_0 + \frac{3}{4}(x_0 + 12u_0)$.

c. The action space is $\mathcal{A}_0(x_0) = \{0, 1\}$.

d. It is not possible to determine an optimal policy since we don't know what Bobs friend will do.

Exercises

Let's try it -- I will probably try to prepare solutions at home and be willing to present them		(19.67 %)
Let's try it -- But I am not going to volunteer to present anything.		(34.43 %)
The format is okay, but I don't want other students to present solutions. It should just be the TA who present the solution.		(26.23 %)
I prefer a format where we just work on the exercises and raise our hand if we have questions: I will be in the first room if this happens.		(19.67 %)

- IT015: Passive exercises; installation problems

- Aud.21 + IT019: Interactive exercises.
  Try to prepare and present homework exercises.

1   Bobs financially challenged friend

☞   Bob has $x_0 = 20$ kroner. He can either:

- Action $u = 0$: Put them in the bank at a 10% interest, thereby ending up with 22 kroner.

- Action $u = 1$: Lend them to a friend.

  - With probability $\frac{1}{4}$ he looses everything
  - With probability $\frac{3}{4}$ his friend gives him 12 kroner (aka one beer) as a thank you, and thus he will have $20 + 12 = 32$ kroner total.

📄 Tue Herlau.
    Sequential decision making.
    (Freely available online), 2024.