02465: Introduction to reinforcement learning and control

The finite-horizon decision problem

Tue Herlau

DTU Compute, Technical University of Denmark (DTU)

Lecture Schedule

Dynamical programming

1. **The finite-horizon decision problem**
   7 February
2. Dynamical Programming
   14 February
3. DP reformulations and introduction to Control
   21 February

Control

4. Discretization and PID control
   28 February
5. Direct methods and control by optimization
   7 March
6. Linear-quadratic problems in control
   14 March
7. Linearization and iterative LQR
   21 March

Reinforcement learning

8. Exploration and Bandits
   28 March
9. Bellmans equations and exact planning
   4 April
10. Monte-carlo methods and TD learning
    11 April
11. Model-Free Control with tabular and linear methods
    25 April
12. Eligibility traces
    2 May
13. Deep-Q learning
    9 May

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

Reading material:

- [Her25, Chapter 4] Introduction

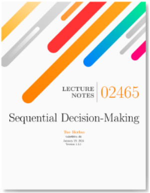
Learning Objectives

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

cweb

Course webpage

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



Search Ctrl + K

- Information
- Models and Environments
- Exercises
- Projects

Contents

Practicalities

Time and place:	Building B341, auditorium 21, 08:00–12:00
DTU Learn:	02465
Exercise code:	https://lab.compute.dtu.dk/02465material/02465students.git
Course descriptions:	kurser.dtu.dk
Discord:	Discord channel (invitation link)
Campus-wide python support:	pythonsupport.dtu.dk
Contact:	Tue Herlau, tuhe@dtu.dk.

Note

This page is automatically updated with typos, etc. I therefore recommend bookmarking it and using the newest version of the exercises.

Course schedule

The schedule and reading can be found below. Click on the titles to read the exercise and project descriptions.

#	Date	Title	Reading	Homework	Exercise	Slides
	Jan 26th, 2024	Installation and self-test	Chapter 1-3 ★, [Her24]		[PDF]	
1	Feb 2nd, 2024	The finite-horizon decision problem	Chapter 4, [Her24]	1, 2	[PDF]	[1x] [6x]
2	Feb 9th, 2024	Dynamical Programming	Chapter 5-6.2, [Her24]	1, 2	[PDF]	[1x] [6x]
3	Feb 16th, 2024	DP reformulations and introduction to Control	Section 6.3; Chapter 10-11, [Her24]	1, 2	[PDF]	[1x] [6x]
4	Feb 23th, 2024	Discretization and PID control	Chapter 12-14, [Her24]	1,2	[PDF]	[1x] [6x]
	Feb 29th, 2024	</> Project 1: Dynamical Programming				
5	Mar 1st, 2024	Direct methods and control by optimization	Chapter 15, [Her24]	1,	[PDF]	[1x] [6x]
6	Mar 8th, 2024	Linear-quadratic problems in control	Chapter 16, [Her24]	tbd	[PDF]	[1x] [6x]
7	Mar 15th, 2024	Linearization and iterative LQR	Chapter 17, [Her24]	tbd	[PDF]	[1x] [6x]
8	Mar 22th, 2024	Exploration and Bandits	Chapter 1; Chapter 2-2.7; 2.9-2.10, [SB18]	tbd	[PDF]	[1x] [6x]

Where and what

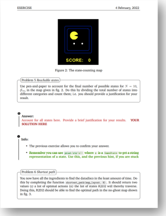
DTU Learn Announcements, assignment hand-ins, quizzes

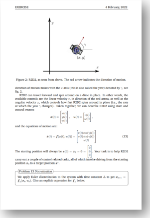
Course homepage Exercises, projects, slides, documentation, installation, etc. `https://02465material.pages.compute.dtu.dk/02465public`

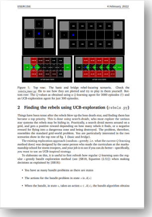
Off-hours QA Discord. See link on homepage.

- Exercises
  - Building B341, auditorium 21
  - Building B341, IT-015
  - Building B341, IT-019
- Ask **project-related question** online so that everyone has the same information (i.e. not in class)

Project work





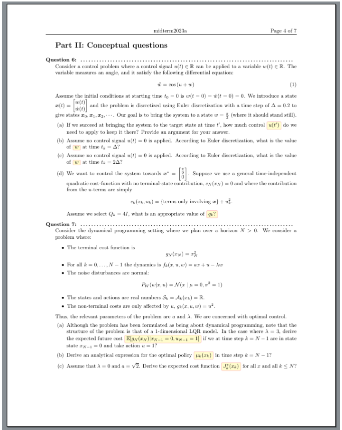


- Groups of 1, 2 or 3 students

  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 in Part 3.

Exam



midterm2023a Page 4 of 7

Part II: Conceptual questions

Question 6: ....

Consider a control problem where a control signal $u(t) \in \mathbb{R}$ can be applied to a variable $w(t) \in \mathbb{R}$. The variable measures an angle, and it satisfy the following differential equation:

$$\ddot{w} = \cos(u + w) \tag{1}$$

Assume the initial conditions at starting time $t_0 = 0$ is $w(t = 0) = \dot{w}(t = 0) = 0$. We introduce a state $\boldsymbol{x}(t) = \begin{bmatrix} w(t) \\ \dot{w}(t) \end{bmatrix}$ and the problem is discretized using Euler discretization with a time step of $\Delta = 0.2$ to give states $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \cdots$. Our goal is to bring the system to a state $w = \frac{\pi}{2}$ (where it should stand still).

(a) If we succeed at bringing the system to the target state at time t', how much control $u(t')$ do we need to apply to keep it there? Provide an argument for your answer.

(b) Assume no control signal $u(t) = 0$ is applied. According to Euler discretization, what is the value of w at time $t_k = \Delta$?

(c) Assume no control signal $u(t) = 0$ is applied. According to Euler discretization, what is the value of w at time $t_k = 2\Delta$?

(d) We want to control the system towards $\boldsymbol{x}^* = \begin{bmatrix} \frac{\pi}{2} \\ 0 \end{bmatrix}$. Suppose we use a general time-independent quadratic cost-function with no terminal-state contribution, $c_N(x_N) = 0$ and where the contribution from the u-terms are simply

$$c_k(x_k, u_k) = \{\text{terms only involving } \boldsymbol{x}\} + u_k^2.$$

Assume we select $Q_k = 4I$, what is an appropriate value of q_k?

Question 7: ....

Consider the dynamical programming setting where we plan over a horizon $N > 0$. We consider a problem where:

- The terminal cost function is
$$g_N(x_N) = x_N^2$$
- For all $k = 0, \dots, N-1$ the dynamics is $f_k(x, u, w) = ax + u - \lambda w$
- The noise disturbances are normal:
$$P_W(w|x, u) = \mathcal{N}(x \mid \mu = 0, \sigma^2 = 1)$$
- The states and actions are real numbers $\mathcal{S}_k = \mathcal{A}_k(x_k) = \mathbb{R}$.
- The non-terminal costs are only affected by u, $g_k(x, u, w) = u^2$.

Thus, the relevant parameters of the problem are a and λ. We are concerned with optimal control.

(a) Although the problem has been formulated as being about dynamical programming, note that the structure of the problem is that of a 1-dimensional LQR model. In the case where $\lambda = 3$, derive the expected future cost $\mathbb{E}[g_N(x_N)|x_{N-1} = 0, u_{N-1} = 1]$ if we at time step $k = N-1$ are in state state $x_{N-1} = 0$ and take action $u = 1$?

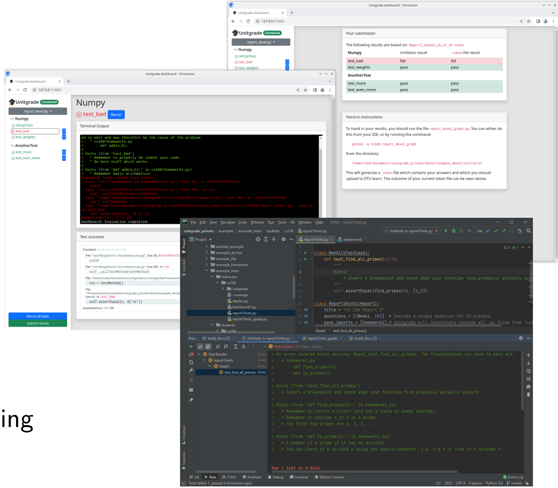
(b) Derive an analytical expression for the optimal policy $\mu_k(x_k)$ in time step $k = N-1$?

(c) Assume that $\lambda = 0$ and $a = \sqrt{2}$. Derive the expected cost function $J_k^*(x_k)$ for all x and all $k \leq N$?

- The 4-hour written exam will contain:
  - Multiple-choice questions
  - Written-answer questions
  - Programming questions
- Your evaluation is an overall assessment based on the written exam and project work
  - The project work is 20%.

N.b. the exam is planned to be in English and not Danish. You can request that I change the language to Danish. I don't think this is to anyones advantage since many terms don't have a good Danish equivalent, however, it is up to you. If you wish that the exam is translated please contact me before week 6 of the course.

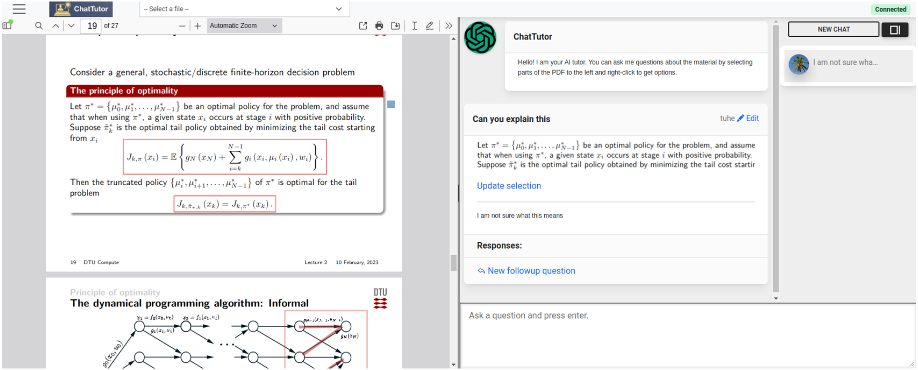
Creating handins



See 'Lecture 0' on the homepage

- Tests are available locally for debugging
- A grade script generate a handin
- Example usage:
  - `python -m irlc.project0.fruit_project_grade`
  - Hand in your code/scores by uploading the `.token` file

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!)
- Sign-up link: `https://chattutor.dk/s/112/bylink/jWOVynKVW6/as/ST/`

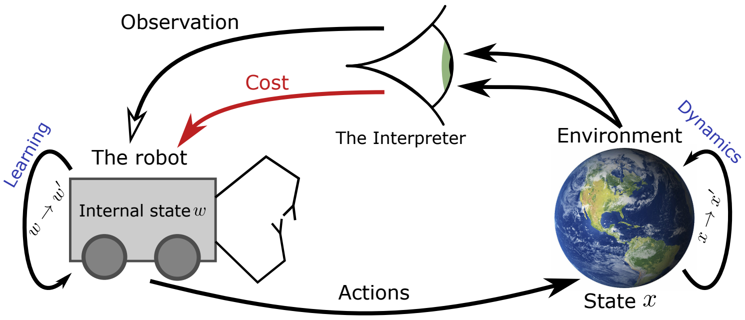
Welcome!

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

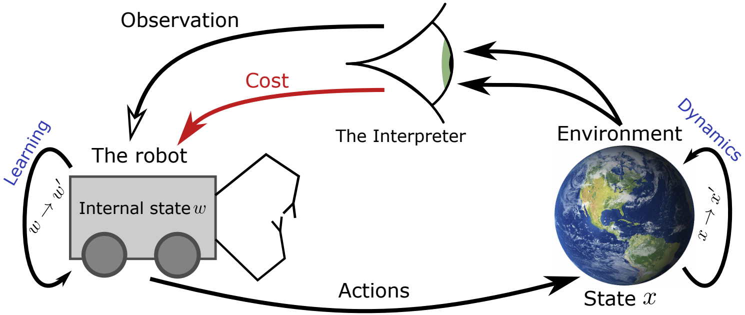


Make decisions, one after another, to bring about a desired outcome

- Observe the world
- Take action
- Obtain cost

Minimize total cost

`lecture_01_pacman.py`

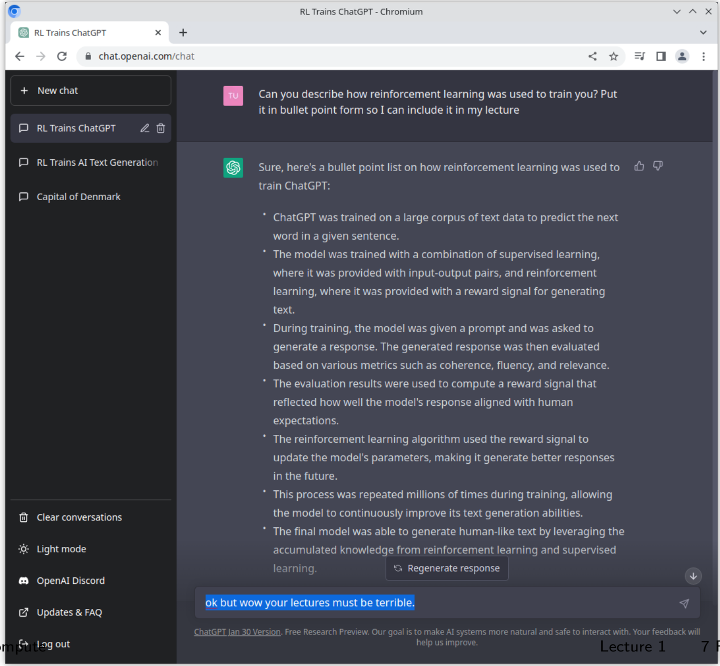


- Time is really important (sequential data)
- Must optimize behavior of dynamical systems using information that becomes progressively available as the systems evolve
- Future cost and state of the system will depend on current actions and state

Alpha-Go (2016)

- Self-learning Go supercomputer
- Defeated world champion Lee Sedol in 2016
- Notable mentions: Atari/Dota/Starcraft II learners

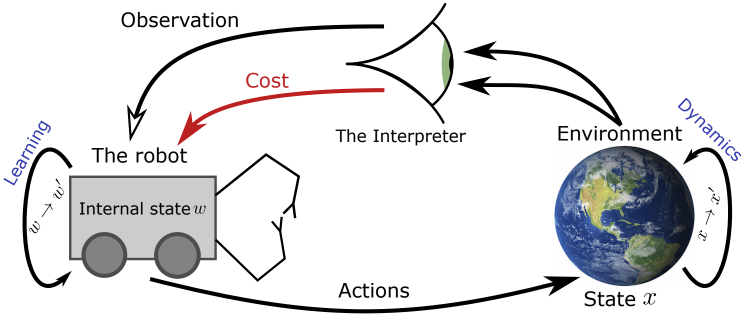
ChatGPT (2022)



How to address this problem

- Establish vocabulary
- Build a mathematical model
- Use the model to solve problems

The decision problem



State The configuration of the environment x

Action The robots output-signal

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

Example: Atari

States RAM memory state

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

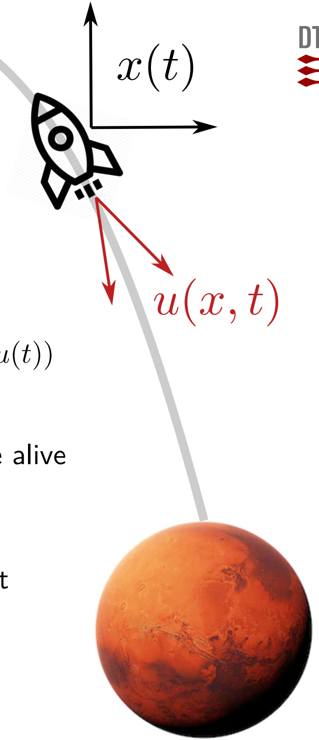
Actions Discrete joystick actions

Dynamics Discrete, stochastic (what the emulator does)

Cost High-score

Don't know dynamics; must learn from scratch

Example: Mars landing



Time Continuous

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

Dynamics A differential equation

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

Cost Land the right place
and use little fuel **and** keep everyone alive

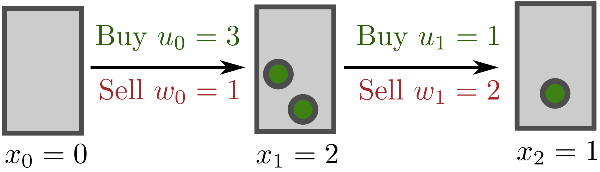
Constraints Thrusters deliver limited force,
ship cannot go into mars, etc.

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 $k = 0, \dots, N-1$ so as to meet a stochastic demand

$$\begin{aligned} x_k &\text{ stock at the beginning of the } k\text{th period,} \\ u_k \geq 0 &\text{ stock ordered at the beginning of the } k\text{th period.} \\ w_k \geq 0 &\text{ Demand during the } k\text{'th period} \end{aligned}$$

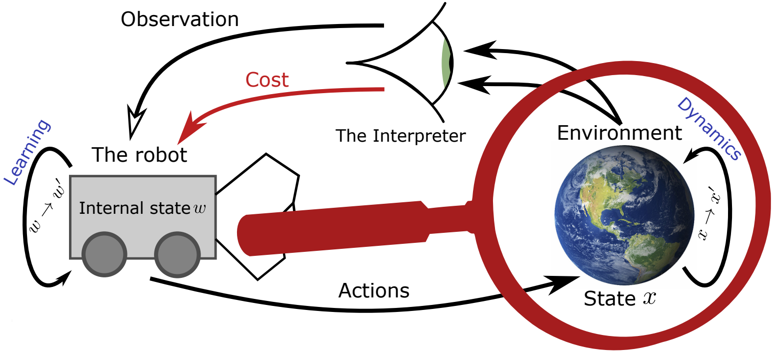
- Dynamics: $x_{k+1} = \min(\max(x_k + u_k - w_k, 0), 2)$
- Cost to minimize:

$$\underbrace{u_k}_{\text{cost-to-order items}} + \underbrace{(x_k + u_k - w_k)^2}_{\text{Satisfy demand + limit inventory size}}$$

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

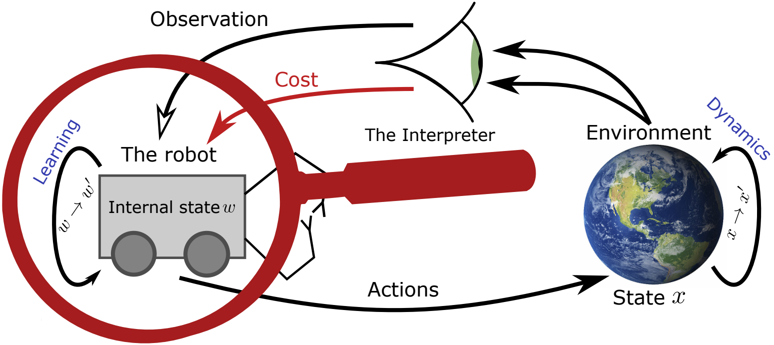
We want proven optimal rule for ordering

The environment



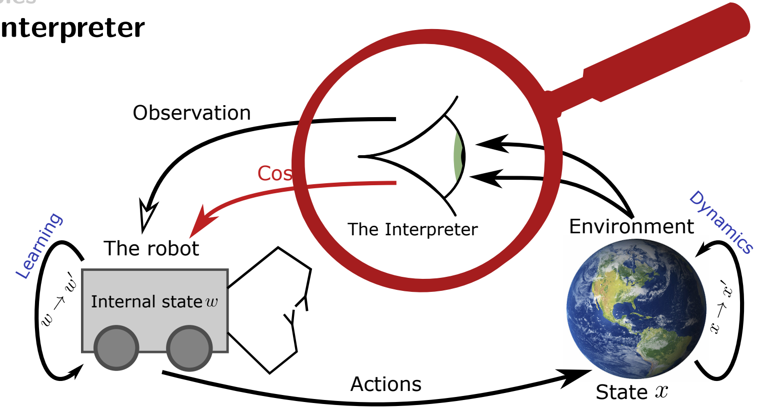
- Nature can be stochastic or deterministic
- The problem can be continuous-time or discrete-time
- We can know the dynamics or not

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** $x_k \in \mathcal{S}_k$

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

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

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 $\pi = \{\mu_0, \mu_1, \dots, \mu_{N-1}\}$

Rollout of policy Given x_0, select $u_k = \mu_k(x_k)$ to obtain a **trajectory** $x_0, u_0, x_1, \dots, x_N$ 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 $u_0, \dots, u_{N-1}$.
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:

$$u_0, \dots, u_{N-1} = \arg\min_{\mathbf{u}} c(\mathbf{u}).$$

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

Pre-semester quiz

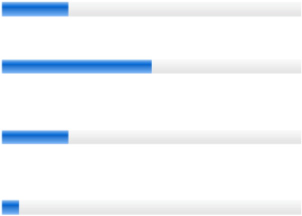

```python
# chapter1/lecture1_code.py
class MyClass:
    def __init__(self, a):
        self.my_variable = a

    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.

I have seen code like this before, but it is not something I have used. I think I can pick it up.

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

This is easy. I have written code like this before and can reason about what it does.



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 co-responsible for the new mandatory programming course (02002/3) in 2023

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`

```
>>> 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:

```
>>> 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

```
>>> 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 can
...         print("This dog is named", self.name, "please give me treats!")
...
>>> dog = BasicDog()
>>> dog.name
'Unnamed dog'
```

`self` refers to the class instance

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

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

```
>>> 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

```
>>> 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

```
>>> 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?

```
>>> 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'
```

```
# 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

```
>>> class Parrot:
...     def __init__(self):
...         self.words = ["Squack!"]
...     def learn(self, word):
...         self.words.append(word)
...     def speak(self):
...         return random.choice(self.words)  # Return a random word
...     def vocabulary(self):
...         return self.words
...
```

```
>>> parrot = Parrot()
>>> words = ["sugar", "sleep well", "(parrot noises)", "*honk*"]
>>> for word in words:
...     parrot.learn(word)
...
>>> for _ in range(3):  # Say three words
...     parrot.speak()
...
'Squack!'
'Squack!'
'sleep well'
>>> print("Vocabulary", parrot.vocabulary())
Vocabulary ['Squack!', 'sugar', 'sleep well', '(parrot noises)', '*honk*']
```

Inheritance

```
>>> class Parrot:
...     def __init__(self):
...         self.words = ["Squack!"]
...     def learn(self, word):
...         self.words.append(word)
...     def speak(self):
...         return random.choice(self.words)  # Return a random word
...     def vocabulary(self):
...         return self.words
...
```

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

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

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

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

Inheritance continued

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

```
>>> class Parrot:
...     def __init__(self):
...         self.words = ["Squack!"]
...     def learn(self, word):
...         self.words.append(word)
...     def speak(self):
...         return random.choice(self.words)  # Return a random word
...     def vocabulary(self):
...         return self.words
...
```

Where is the bug?

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

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

```
>>> class SqueekyParrot(Parrot):
...     def __init__(self, squeek="Quck!"):
...         super().__init__()   # Call the 'Parot' class __init__ method to set up the words-variable.
...         self.squeek = squeek # save the squeek variable
...     def speak(self):
...         word = super().speak() # Use the speak() function defined in the Parrot class.
...         return f"{self.squeek} {word} {self.squeek}"
...
>>> squeeky = SqueekyParrot(squeek="Kvak-Kvak")
>>> squeeky.learn("Good night!")
>>> squeeky.learn("Tell that damn bird to shut it's beak")
>>> squeeky.learn("Sugar!")
>>> squeeky.speak()
'Kvak-Kvak Good night! Kvak-Kvak'
>>> squeeky.speak()
'Kvak-Kvak Sugar! Kvak-Kvak'
```

Why classes in this course?

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 Inheritance allows us to re-use code

- In control theory, we will use inheritance to add simulation-functionality to all models

The inventory environment

```python
# inventory_environment.py
class InventoryEnvironment(Env):
    def __init__(self, N=2):
        self.N = N                                      # planning horizon
        self.action_space      = Discrete(3)            # Possible actions {0, 1, 2}
        self.observation_space = Discrete(3)            # Possible observations {0, 1, 2}

    def reset(self):
        self.s = 0                                      # reset initial state x0=0
        self.k = 0                                      # reset time step k=0
        return self.s, {}                               # Return the state we reset to (and an

    def step(self, a):
        w = np.random.choice(3, p=(.1, .7, .2))    # Generate random disturbance
        s_next = max(0, min(2, self.s-w+a))          # next state; x_{k+1} =  f_k(x_k,
        reward = -(a + (self.s + a - w)**2)          # reward = -cost      = -g_k(x_k,
        terminated = self.k == self.N-1              # Have we terminated? (i.e. is k=
        self.s = s_next                              # update environment state
        self.k += 1                                  # update current time step
        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:

```python
# inventory_environment.py
class RandomAgent(Agent):
    def pi(self, s, k, info=None):
        """ Return action to take in state s at time step k """
        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

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

As average over 1000 trajectories

```
# 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 ($x_1 = 0$)
  - With probability $\frac{3}{4}$ his friend gives him 12 kroner (aka one beer) as a thank you, and thus he will have $x_1 = 20 + 12 = 32$ 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:

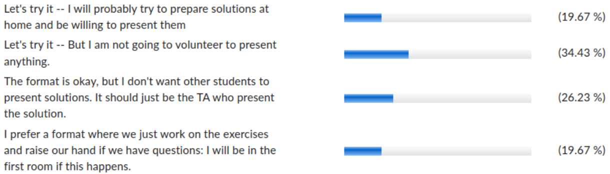
a. The state spaces are $\mathcal{S}_k = \{1, 2, \dots, 32\}$.

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



- 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), 2025.