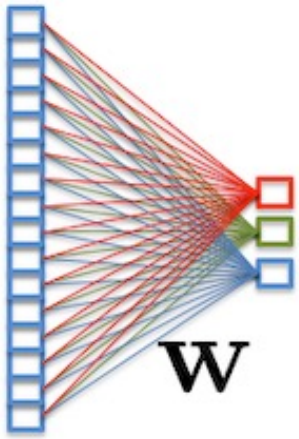
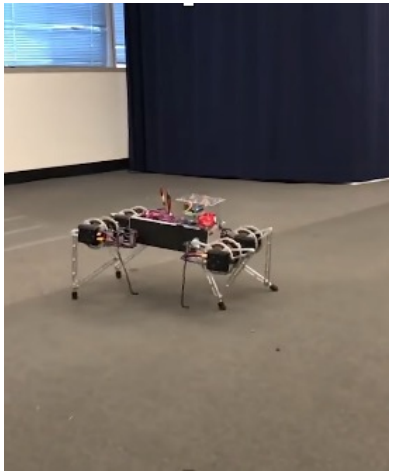
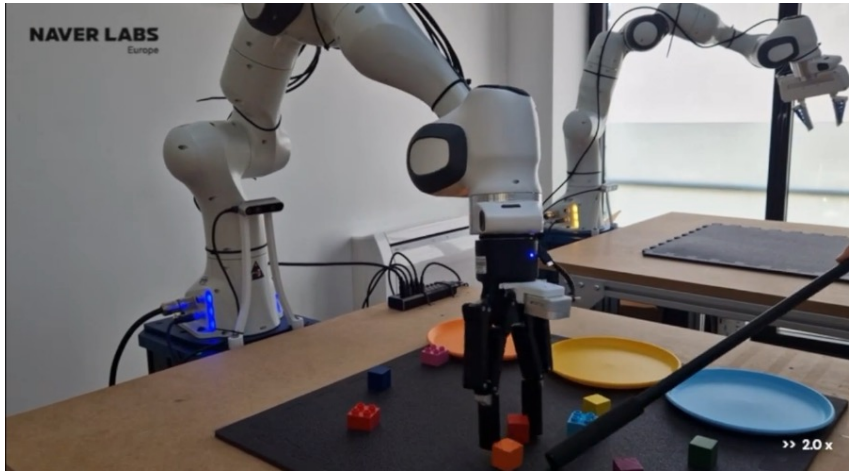
AI and Data Analytics

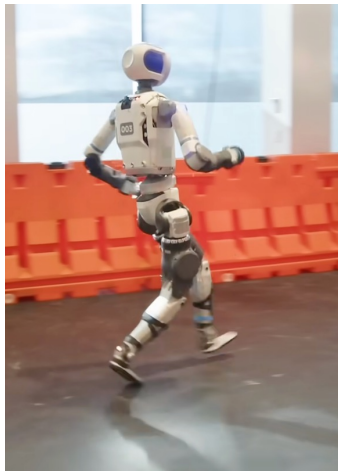
3.1 Imitation Learning

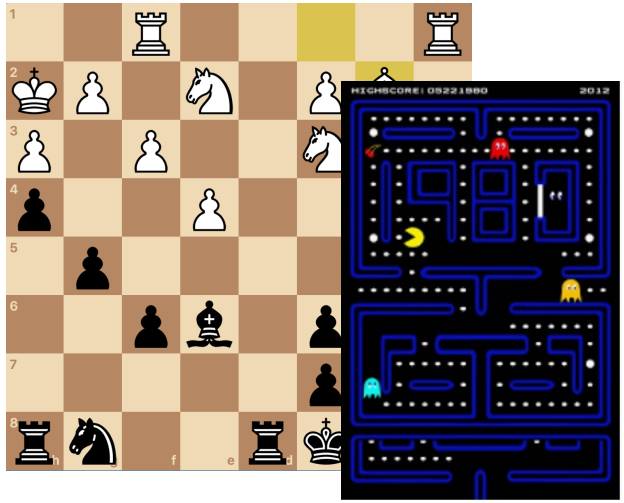


Christian Wolf

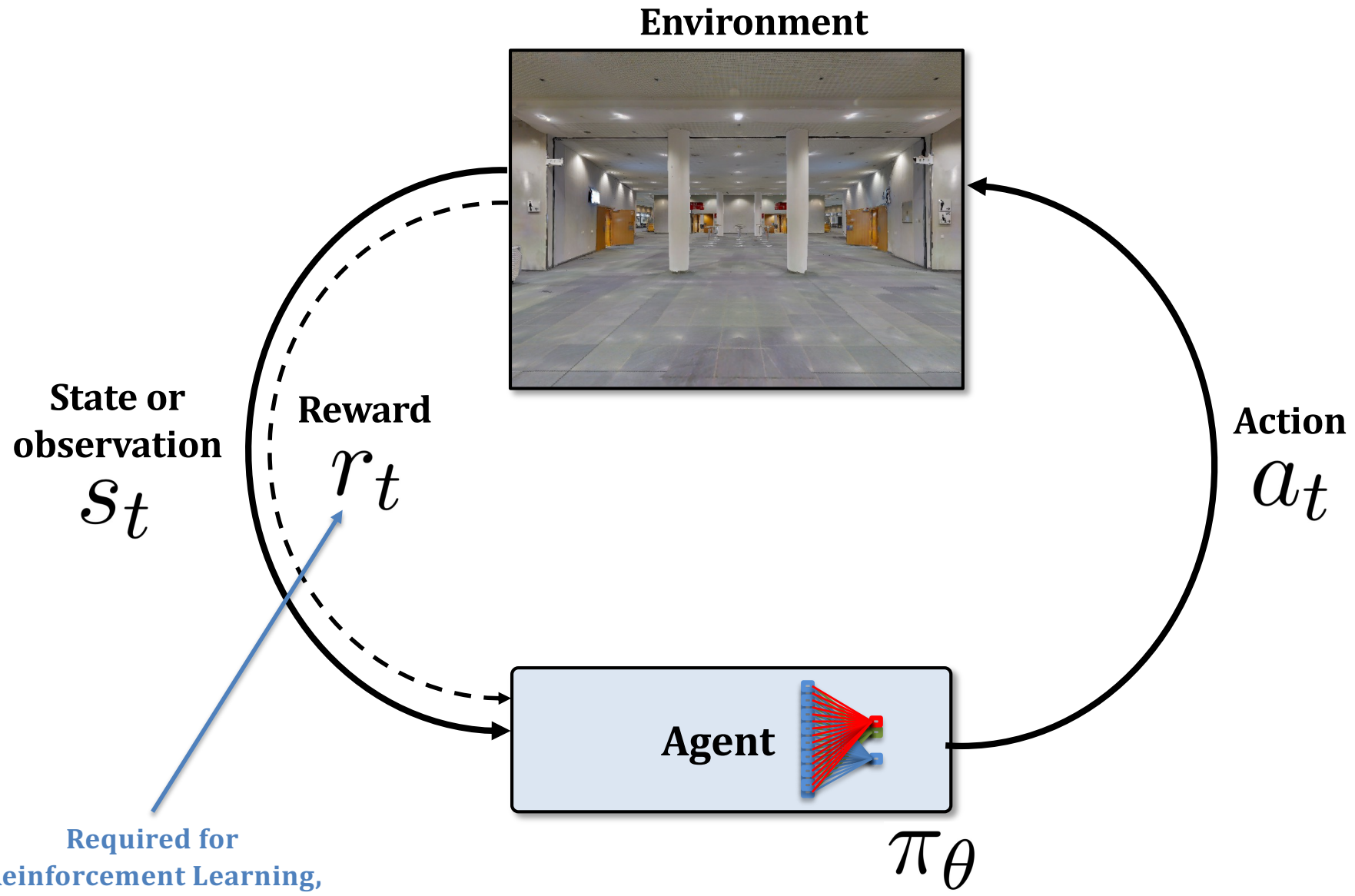
Learning to control



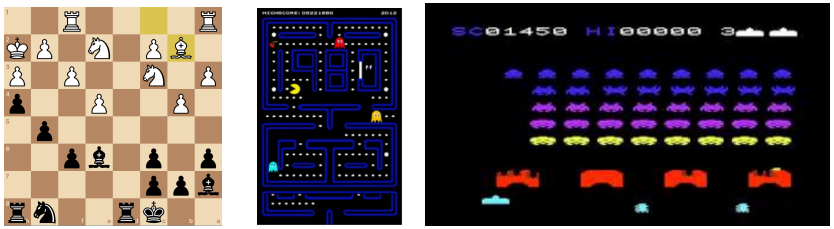
Interacting with an "environment"



States

An environment is in a given state s_t at any instant t.

Fully observable problems — the state is observable to the agent at any time, e.g. board games (chess, Go), 2D computer games with visibe full screen (Pacman).



Partially observable problems — only an observation is returned to the agent at each instant, not the full state. Examples: robot navigation; 3D computer games with ego-centric view.

Actions

At each time t, the agent takes an action a_t from an action space $\mathcal{A}$.

Discrete action spaces

- Chess: $\mathcal{A} = \{a1, Qf3, Ra1, bf3, \ldots\}$.
- Go: $\mathcal{A} = [1 \ldots 19] \times [1 \ldots 19]$.
- Dialogue: $\mathcal{A} =$ words or characters.

Continuous action spaces

- Robotic arm control: $\mathcal{A} =$motor commands
- Drone (UAV) control: $\mathcal{A} =$ motor commands

Policies

The policy π of an agent is a function which

- chooses an action given that the agent is in a current state;
- or, in the stochastic case, it assigns a probability to an action:

$$\pi(a_t|s_t)$$

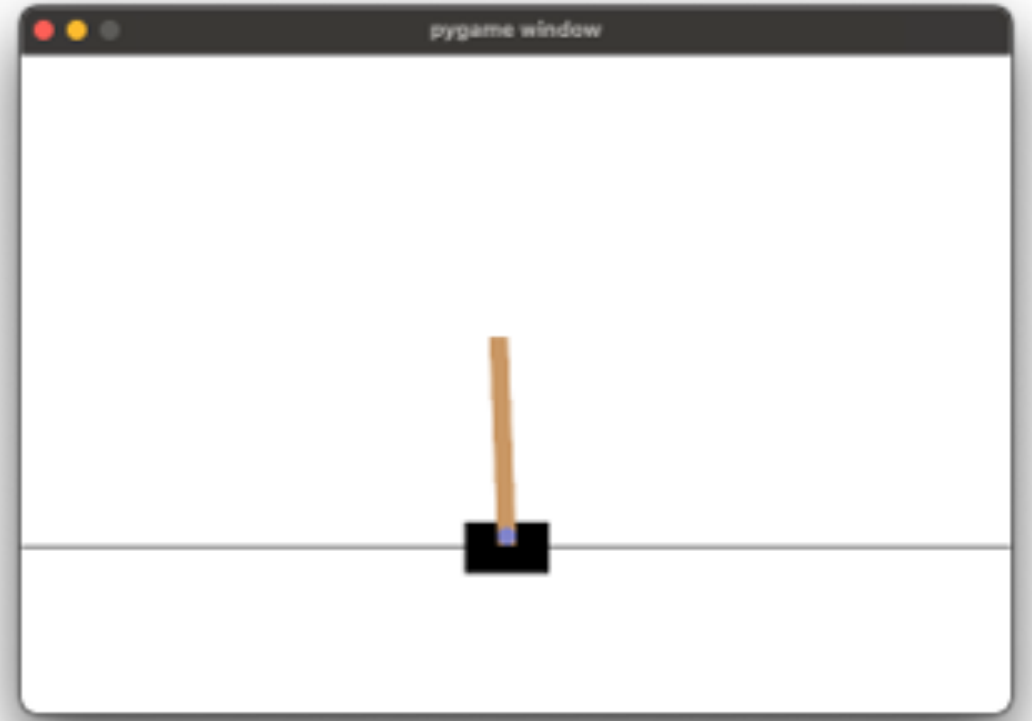
Policies are generally parametrized by a vector θ and implemented as a neural network, denoted as π_θ.

The Gym-Cartpol environment

A cart with an inverted pendulum is controlled with two different actions:

- 0 $\longrightarrow$ move to the left
- 1 $\longrightarrow$move to the right

The goal is to keep the pendulum/pole stable upright.

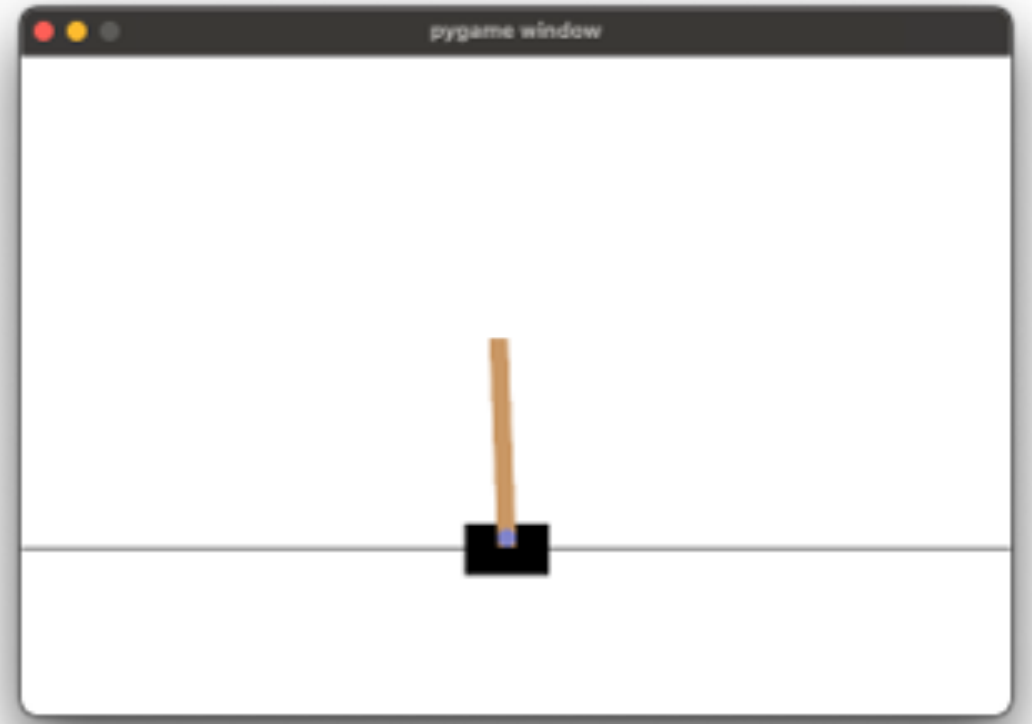


The Gym-Cartpol environment

Termination criteria:

- Angle too steep: failure
- Cart too far off the center: failure
- Time limit reached: success

Lecture demo source code: `cartpol_test.py`



The Gym-Cartpol environment

States space: 4 values
Action space: 2 values

$$f(\begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}) = \begin{bmatrix} \text{Score for Left} \\ \text{Score for Right} \end{bmatrix}$$

		min	max
x	Cart Position	-4.8	4.8
$\dot{x}$	Cart Velocity	$-\infty$	∞
θ	Pole Angle	$\approx -0.418 = -24°$	$\approx 0.418 = 24°$
$\dot{\theta}$	Pole Angular Velocity	$-\infty$	∞

Gym “Cartpol”: programming interface

```
# Initialize environment
env = gym.make("CartPole-v1", render_mode="human")
env.reset()

print("Left/Right arrows; 'ESC' to quit.")
running = True
while running:
  action = None
  keys = pygame.key.get_pressed()

  if keys[pygame.K_LEFT]:
    action = 0
  elif keys[pygame.K_RIGHT]:
    action = 1
  if action is not None:
    obs, reward, term, succ, info = env.step(action)

env.close()
pygame.quit()
```

Adding success/failure checking

```
1   if action is not None:
2     obs, reward, term, succ, info = env.step(action)
3
4     if term:
5       print("\n" + "="*20 + "\n  EPISODE FAILED!\n" +
      "="*20 + "\n" )
6
7       # Keep the window open for a second, then reset
8       time.sleep(1.0)
9       env.reset()
10
11     elif succ:
12       print("Time Limit Reached (Success!)")
13       env.reset()
14
15  # User wants to quit
16  for event in pygame.event.get():
17    if event.type == pygame.QUIT or (event.type ==
      pygame.KEYDOWN and event.key == pygame.K_ESCAPE):
18      running = False
```

Behavior cloning

"**Behavior cloning**" is a special case of "**Imitation Learning**":

- Use a dataset $\mathcal{D} = \{(s_i, a_i^*)\}$ of pairs of input states s_i and expert actions a_i^*.
- Use supervised learning: train a policy π to predict a_i^* when being in state s_i:

$$\min_\theta \sum_i \mathcal{L}(\pi_\theta(s_i), a_i^*)$$

- The loss function $\mathcal{L}$ depends on the action space. For instance, for discrete actions (Gym-Cartpol), a cross-entropy loss could be used.

Implementing the policy

```
class StudentNet(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, 32),
            nn.ReLU(),
            nn.Linear(32, action_dim)
        )
    def forward(self, x):
        return self.net(x)
```

Behavior Cloning in PyTorch

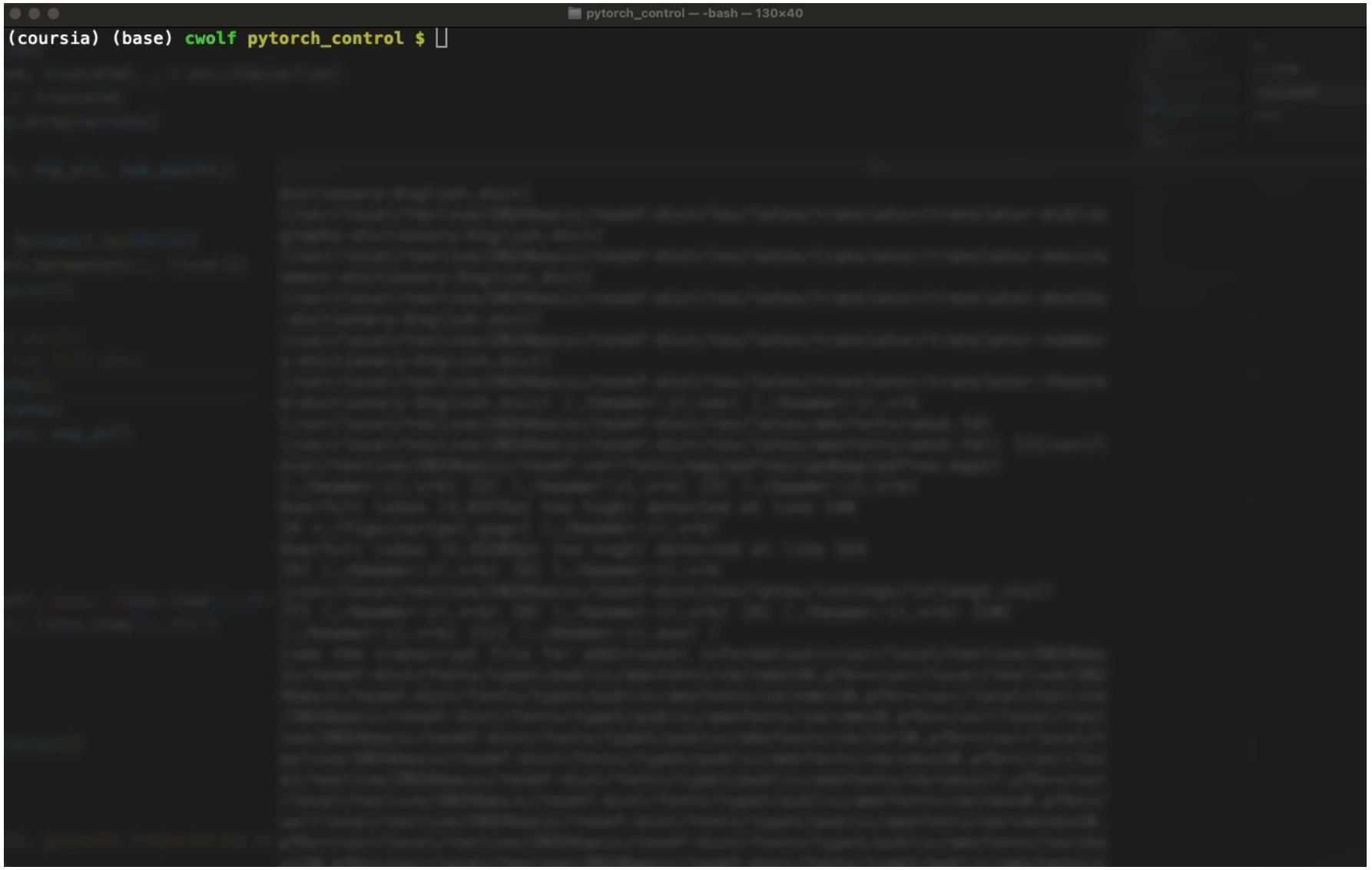
```
def behavior_cloning(exp_states, exp_act, num_epochs):
  States=4
  Actions=2
  model = StudentNet(States, Actions)
  optimizer = optim.Adam(model.parameters(), lr=1e-3)
  criterion = nn.CrossEntropyLoss()

  # For simplicity, we do not batch:
  # do one forward pass with the full data
  for epoch in range(num_epochs):
    pred_act = model(exp_states)
    loss = criterion(pred_act, exp_act)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

  return model
```

Demo



Lecture demo full code: beh_cloning.py