Breakout Group

Reinforcement Learning

FABIAN RUEHLE (UNIVERSITY OF OXFORD)

String_Data 2017, Boston

12/01/2017

Outline

- Theoretical introduction (*30 minutes*)
- Discussion of code (*30 minutes*)
    - Solve version of grid world with SARSA
- Discussion of RL and its applications to String Theory (*30 minutes*)

How to teach a machine

- Supervised Learning (SL):
  - provide a set of training tuples $[(\mathrm{in}_0, \mathrm{out}_0), (\mathrm{in}_1, \mathrm{out}_1), \ldots, (\mathrm{in}_n, \mathrm{out}_n)]$
  - after training, machine predicts $\mathrm{out}_i \ \mathrm{from} \ \mathrm{in}_i$
- Unsupervised Learning (UL):
  - only provide set training input set $[\mathrm{in}_0, \mathrm{in}_1, \ldots, \mathrm{in}_n]$
  - give task to machine (e.g. cluster input) without telling it how to do this exactly
  - After training, the machine will perform self-learned action on in_i
- Reinforcement Learning (RL):
  - in between SL and UL
  - Machine acts autonomously, but actions are reinforced / punished

Theoretical introduction

Reinforcement Learning - Vocabulary

- Basic textbooks/literature [Barton, Sutton ’98 ‘17]
- The “thing that learns” is called *agent* or *worker*
- The “thing that is explored” is called *environment*
- The “elements of the environment” are called *states* or *observations*
- The “things that take you from one state to another” are called *actions*
- The “thing that tells you how to select the next action” is called *policy*
- *Actions* are executed sequentially in a sequence called *(time) steps*
- The “reinforcement” the agent experiences is called *reward*
- The “accumulated reward” is called *return*
- In *RL,* an *agent* performs *actions* in an *environment* with the goal to maximize its long-term return

Reinforcement Learning - Details

- We focus on discrete state and action spaces
- State space $S = \{\text{states in environment}\}$
- Action space
  - total: $A = \{\text{actions to transition between states}\}$
  - for $s \in S : A(s) = \{\text{possible actions in state } s\}$
- Policy π: Select next action for given state $\pi : S \mapsto A$

  $\pi(s) = a\,,\ a \in A(s)$

- Reward $R(s, a) \in \mathbb{R}$: Reward for taking action a in state s

  $R :\ S \times A \mapsto \mathbb{R}$

Reinforcement Learning - Details

- Return: The accumulated reward from current step t
  $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$, $\gamma \in (0, 1]$
- State value function $v_\pi(s)$: Expected return for s with policy π : $v_\pi(s) = \mathbb{E}[\, G_t \mid s = s_t]$
- Action value function $q(s, a)$: Expected return for performing action a in state s with policy π :
  $q_\pi(s, a) = \mathbb{E}[\, G_t \mid s = s_t, a = a_t]$
- Prediction problem: Given π , predict $v_\pi(s)$ or $q_\pi(s, a)$
- Control problem: Find optimal policy π_* that maximizes $v_\pi(s)$ or $q_\pi(s, a)$

Reinforcement Learning - Details

- Commonly used policies:
  - greedy: Choose the action that maximizes the action value function: $\pi'(s) = \operatorname{argmax}\ q(s, a)$
  - ε - greedy: Explore different possibilities

$$\pi'(s) = \begin{cases} \text{Choose greedy in } (1-\varepsilon) \text{ cases} \\ \text{Choose random action in } \epsilon \text{ cases} \end{cases}$$

- We take ε-greedy policy improvement
- On-policy: Update policy you are following (e.g. always ε-greedy)
- Off-policy: Use different policy for choosing next action a_{t+1} and updating $q(s_t, a_t)$

Reinforcement Learning - SARSA

- Solving the control problem: $\Delta v(s_t) = \alpha[G_t - v(s_t)]$
  - α : Learning rate ($\alpha = 0$ means no update to $v(s_t)$)
  - One step approximation: $G_t = r + \gamma v(s_{t+1})$
- Similar for action value function:

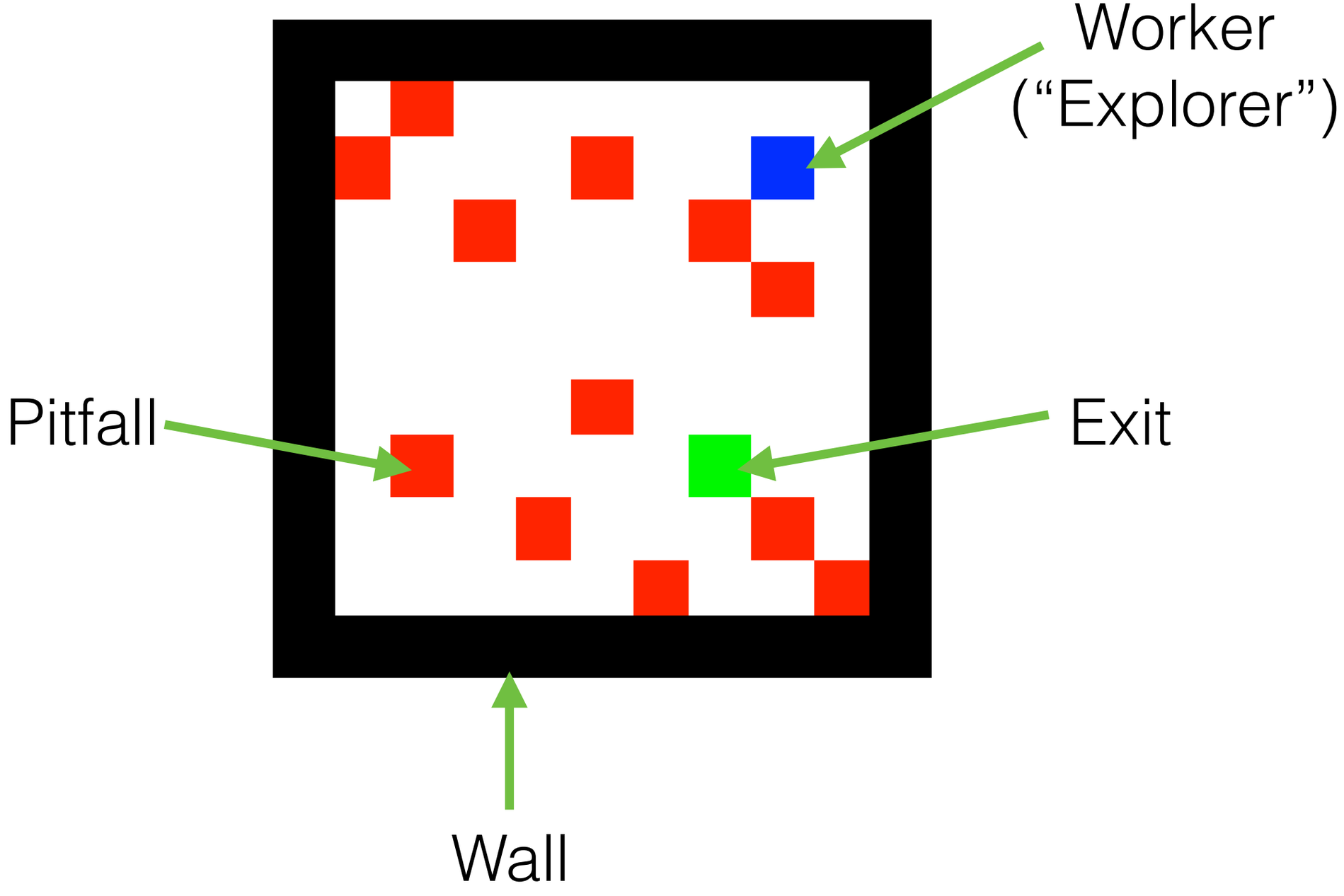
$$\Delta q(s_t, a_t) = \alpha[G_t - q(s_t, a_t)]$$
$$= \alpha[r + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t))]$$

  - Update depends on tuple $(s_t, a_t, r, s_{t+1}, a_{t+1})$
  - a_{t+1} is currently best known action for state s_{t+1}
- Note: SARSA is on-policy

Reinforcement Learning - Q-Learning

- Very similar to SARSA
- Difference in update:
  - SARSA: $\Delta q(s_t, a_t) = \alpha[r + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)]$
  - Q_Learning: $\Delta q(s_t, a_t) = \alpha[r + \gamma \max_{a'} q(s_{t+1}, a') - q(s_t, a_t)]$
- Note: This means that Q-Learning is off-policy
- SARSA is found to perform better
- Q-Learning is proven to converge to solution
- Combine with (deep NNs): Deep Q-Learning

Example - Gridworld



Example - Gridworld

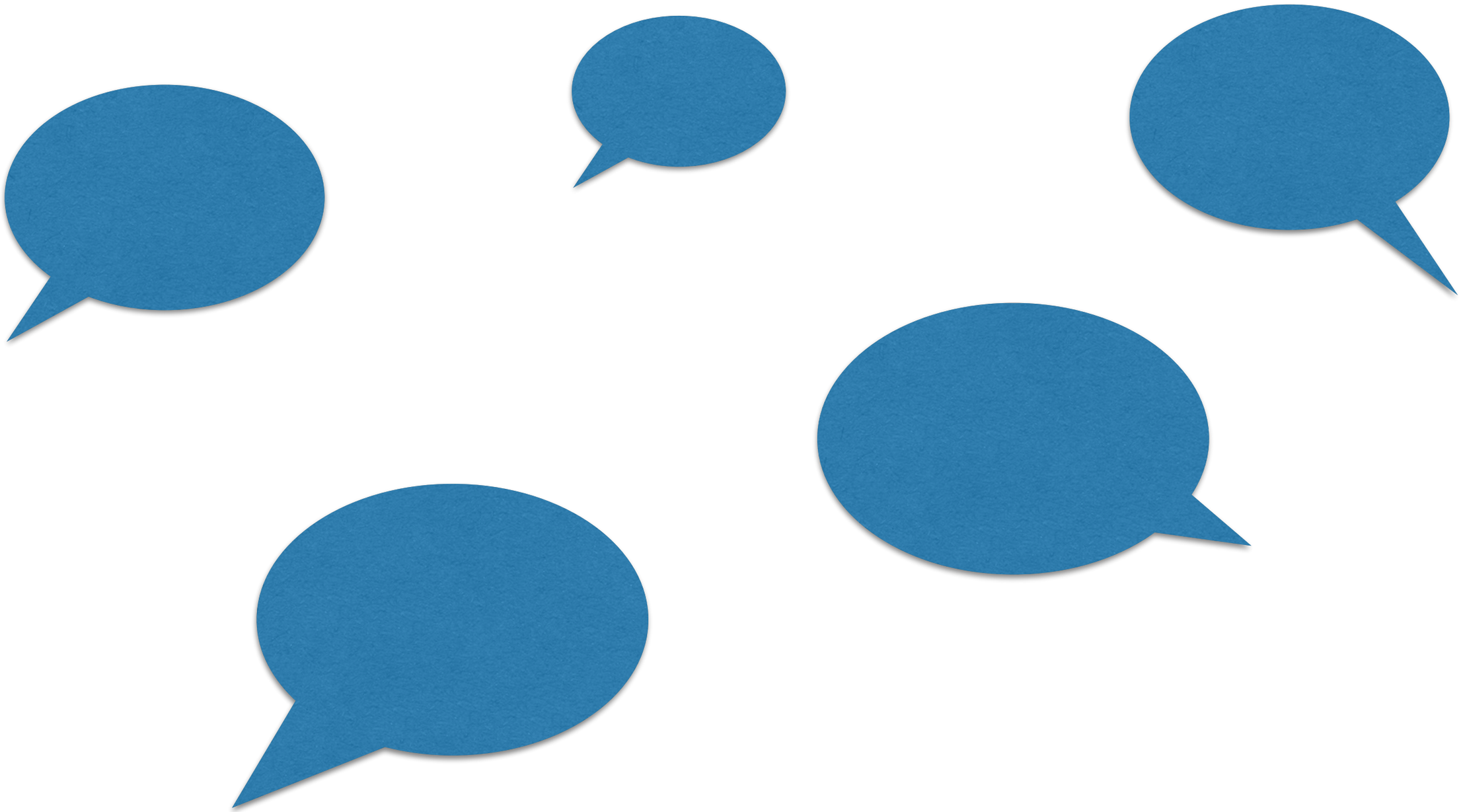
- We will look at a version of grid world:
  - Gridworld is a grid-like maze with walls, pitfalls, and an exit
  - Each state is a point on the grid of the maze
  - The actions are $A = \{\text{up}, \text{down}, \text{left}, \text{right}\}$
  - Goal: Find the exit (strongly rewarded)
  - Each step is punished mildly (solve maze quickly)
  - Pitfalls should be avoided (strongly punished)
  - Running into a wall does not change the state

Gridworld vs String Landscape

- Walls = Boundaries of landscape (negative number of branes)
- Empty square = Consistent point in the landscape which does not correspond to our Universe
- Pitfalls = Mathematically / Physically inconsistent states (anomalies, tadpoles, …)
- Exit = Standard Model of Particle Physics

```
plt.close('all')
t = 1.0  # count time
training_episodes = 30000
agent.max_steps = 200
alpha_W = 0.3
for i in range(training_episodes):
    if i % 1000 == 0:
        print "I'm playing game " + str(i) + " / "
    if i % 100 == 0:
        t += 0.01
    agent.reset()
    found_initial_move = False
    current_action = None
    current_state = agent.get_state()
    while not found_initial_move:
```

Coding



Discussion