Machine Learning for the Quantified Self

Lecture 7

Reinforcement Learning

Overview

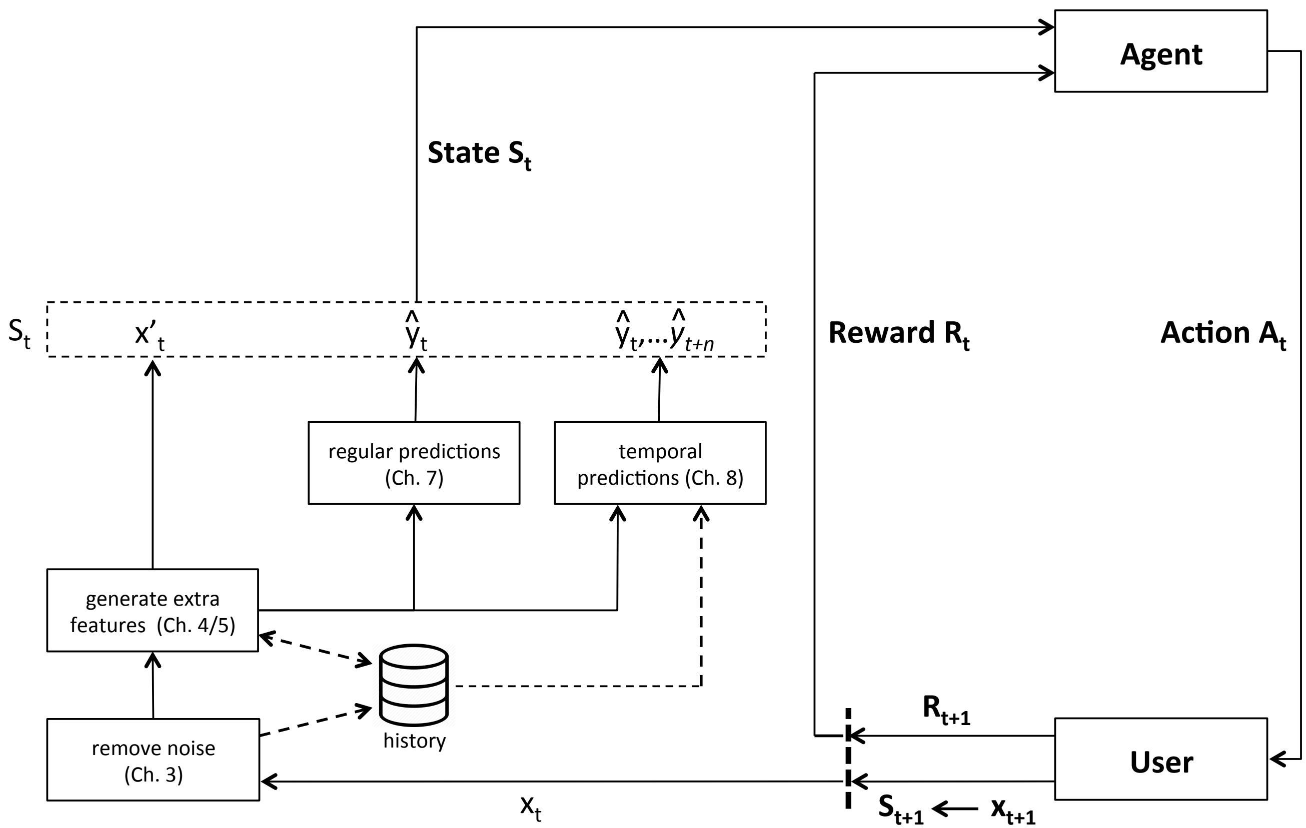
- We have been talking data and making predictions
- What if we want to influence the user?
- What intervention/action should we select?
- We can use reinforcement learning to figure this out
  - Note: not widespread in terms of applications for the quantified self yet
  - We discuss the basics, but many improvements have been made over the years

Basics of Reinforcement Learning (1)

- We assume two actors:
  - The *user* (mostly called the *environment* in RL) – the quantified selfs
  - The *agent* providing the support – the software entity we are aiming to create
  - The *agent* can observe the state of the user at time point *t*: $S_t \in \mathcal{S}$
  - We derive an action to perform based on this: $A_t \in \mathcal{A}(S_t)$
  - We obtain a reward: R_{t+1}

Basics of Reinforcement Learning (2)

- The loop:



Basics of Reinforcement Learning (3)

- We do not strive for immediate reward only, but rewards we accumulate in the future, called the *value function*
- A *policy* maps a state to an action (when to do what)
- We should balance *exploration* and *exploitation*

Basics of Reinforcement Learning (4)

- Let us define the G (value function) better:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{T-(t+1)} \gamma^k R_{t+k+1}$$

  – γ is the discount factor
  – T is the end time we consider (could be ∞)
- The Markov Properly underlies many of the algorithms we consider

Markov Property

- We can express a probability of ending up in a state with a reward based on the entire history:

$$\mathbf{Pr}\{R_{t+1} = r, S_{t+1} = s | S_0, A_0, R_0, \ldots, S_t, A_t, R_t\}$$

- We can also define it based on the previous state and action only:

$$\mathbf{Pr}\{R_{t+1} = r, S_{t+1} = s | S_t, A_t\}$$

- If the probabilities are equal we satisfy the property
  - True for the quantified self?

MDP (1)

- Let us assume the property is satisfied
- We can model our problem as a Markov Decision Process (MDP) with a finite number of states
- Transition probability from one state *s* tot a state *s'*: $p(s'|s,a) = \mathbf{Pr}\{S_{t+1} = s' | S_t = s, A_t = a\}$
- The expected reward for this transition:

$$r(s,a,s') = \mathbb{E}[R_{t+1} | S_{t+1} = s', S_t = s, A_t = a]$$

MDP (2)

- A policy selects a probability of an action in a state:

  $\pi(a|s)$
- For a policy π, the expected value in a state *s* given that we follow policy π thereafter is called the state-value function:

  $v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$
- And the expected return of a policy if we select action *a* in state *s* is called the action-value function:

  $q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$

MDP (3)

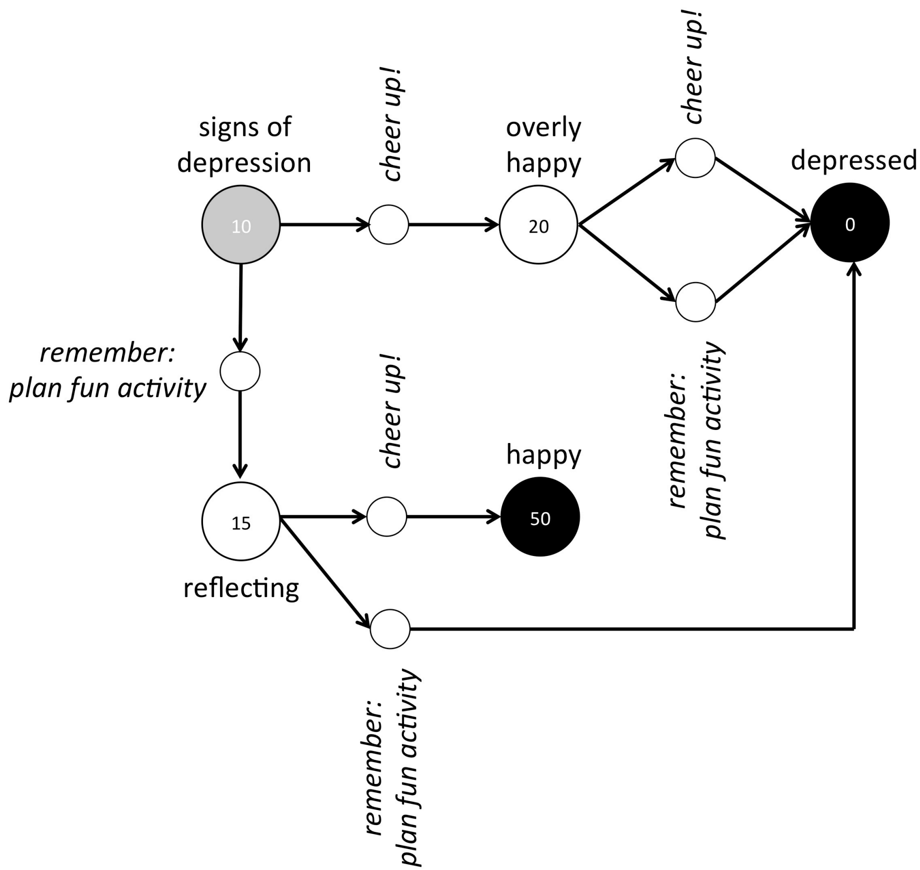
- We want to find policies with the highest state-value function over all states:

$$\forall s : v_{\pi_*}(s) \geq v_{\pi}(s)$$

- Similarly we want to find the optimal action-value function

MDP (4)

- Example:



One Step Sarsa (1)

- Let us start learning
- Let $Q(S_t,A_t)$ denote the learned action value function for a policy π
- Let us operationalize our goal:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

$$q_\pi(s,a) = \mathbb{E}_\pi[\sum_{k=0}^{T-(t+1)} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$$

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma \sum_{k=0}^{T-(t+2)} \gamma^k R_{t+k+2} | S_t = s, A_t = a]$$

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \lambda q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

One Step Sarsa (2)

- We update our value for the state as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

- We select the action based on these Q-values, an example is ε-greedy:

Algorithm 25: ε-greedy action selection given a state S

r = random number from $[0, 1]$
if $r < \varepsilon$ **then**
  return *a random action A from* $\mathcal{A}(S)$
else
  return $\text{argmax}_{A \in \mathcal{A}(S)} Q(S, A)$

One Step Sarsa (3)

- Or alternatively a softmax (τ is temperature):

$$p(A|S) = \frac{e^{\frac{Q(A,S)}{\tau}}}{\sum_{A' \in \mathcal{A}(S)} e^{\frac{Q(A',S)}{\tau}}}$$

One Step Sarsa (4)

- Complete algorithm

Algorithm 26: Evolving a policy π with Sarsa

$\forall S \in \mathcal{S}, A \in \mathcal{A} : Q(S,A)$ = random
$S = \text{derive\_state}(x_1)$
time = 1
Select an action A based from $\mathcal{A}(S)$ based on our set of $Q(S,A)$'s using a selection approach
while *True* **do**
  Perform action A
  time = time + 1
  $S' = \text{derive\_state}(x_{time})$
  R = observe reward
  Select an action A' from $\mathcal{A}'(S')$ based on our set of $Q(S',A')$'s using a selection approach
  Perform action A'
  $Q(S,A) = Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$
  A = A'
  S = S'
end

One Step Sarsa (5)

- For Sarsa, we pick our actions in the same way at each step
- This is called *on-policy*

Q-Learning (1)

- For Q-learning, we do not perform the next action before updating our Q-values
- We just assume that we select the highest value in the next state
- *Off-policy* approach

Q-learning (2)

- Complete algorithm

Algorithm 27: Evolving a policy with Q-learning

$\forall S \in \mathcal{S}, A \in \mathcal{A} : Q(S,A)$ = random
$S = \text{derive\_state}(x_1)$
time = 1
while *True* **do**
    Select an action A based from $\mathcal{A}(S)$ based on our $Q(S,A)'s$ using a selection approach
    Perform action A
    time = time + 1
    $S' = \text{derive\_state}(x_{time})$
    $Q(S,A) = Q(S,A) + \alpha(R + \gamma \max_{\mathcal{A}'(S')} Q(S',A') - Q(S,A))$
    S = S'
end

Multiple steps ahead (1)

- Previously we only looked one step ahead, and could not put credit to actions that contributed longer in the past
- Eligibility traces allow us to do this
- We use $Z_t(s,a)$ to denote an eligibility trace

$$Z_t(s,a) = \begin{cases} \gamma\lambda Z_{t-1} + 1 & \text{if } s = S_t \wedge a = A_t \\ \gamma\lambda Z_{t-1} & \text{otherwise} \end{cases}$$

Multiple steps ahead (2)

- And we include this in our update equations
  - The more frequently an action is selection, the more we update the values
  - For Sarsa:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot (R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \cdot Z_t(S_t, A_t)$$

  - For Q-learning:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha \cdot (R_{t+1} + \gamma \max_{\mathcal{A}'(S_{t+1})} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \cdot Z_t(S_t, A_t)$$

Approximate solutions

- We have assumed a value for each possible Q(S,A), but is this realistic?
  - Nope, there might be many
  - We can build a model that predicts the value for Q(S,A)
  - This is a "standard" learning problem we have seen so far, assume a certain model $\hat{f}(S_t, A_t, \mathbf{w})$ with weights **w** (e.g. a neural network)
  - We define the error as:

$$\sum_{S \in \mathcal{S}, A \in \mathcal{A}} \sqrt{(Q(S_t, A_t) - \hat{f}(S, A, \mathbf{w}))^2}$$

Handling continuous values in the state space (1)

- Final part:
  - We have states represented by continuous values, what then?
  - We have an infinite state space
  - We need to discretize this
  - We can use the U-tree algorithm for this purpose

Handling continuous values in the state space (2)

- How does it work?
  - We build a state tree, that maps our continuous values to a state
  - We start with a single leaf
  - We collect data for a while
  - For all attributes X_i we try different splits based on the (sorted) values we have collected
  - We test whether the splits result in a significant difference in Q-values using the Kolmogorov Smirnov test
  - We select the attribute with the lowest p-value and split on it (if below 0.05)
  - We continue collecting data again and repeat the procedure per leaf