**Artificial Intelligence in Games**

**Session 9**

1. Learning a model: the naive approach
   1. If the one-step dynamics are not available, they can always be estimated by interacting with the environment randomly
   2. For any s,s’ 0 in S and a in A, the simplest (maximum likelihood) estimate for P^a\_{ss’} is given by  
        
      P^a\_{ss’} = N(s’,s,a) / N(s,a)  
        
      where N(s, a) > 0 is the number of times that action a was taken at state s, and N(s’,s,a) is the number of times that state s’ was observed after action a was taken at state s
   3. This estimates P^a\_{ss’} and R^a\_{ss’} can be combined with value iteration
   4. However, obtaining good estimates by interacting with the environment randomly may take too long.
2. Exploration and exploitation
   1. Exploration-exploitation trade-off: should the agent explore in order to learn about potentially better sources of reward or exploit the well-known sources of reward?
   2. This trade-off is relevant whenever an environment model is not available
   3. Excessive exploration typically leads to poor short-term performance, while excessive exploitation typically leads to poor long-term performance
3. Multi-armed bandits
   1. An agent interacts with an environment during a sequence of T discrete time steps
   2. At each time step t >= 0, the agent selects an action a\_t in A
   3. One time step later, the agent receives a reward r\_t+1 in R drawn from a fixed probability distribution with mean µat associated to action a\_t
   4. The interaction between the agent and the environment up to time step t can be represented by a trajectory tau\_t = a0,r1, a1,r2, . . . , at−2,rt−1
4. Bandit algorithms
   1. For each time step t, a bandit algorithm provides a policy pi\_t : A → [0, 1] such that pi\_t(a) is the prescribed probability for taking action a after observing the trajectory tau\_t
   2. The objective of an agent is to maximize the expected return given by  
        
      Expectation over { sum of (R\_t) }  
        
      where the expectation is (implicitly) over trajectories obtained by the agent in a specific environment
5. Regret
   1. The regret E(t) >= 0 of a bandit algorithm after t time steps is given by  
        
      E(t) = (t mu\_\*) - Expectation over { sum of (R\_t’) }  
        
      where mu\_\* = max\_a u\_a and the expectation is (implicitly) over trajectories obtained by the bandit algorithm in a specific environment
   2. The regret compares the expected return of the optimal agent for an environment with the expected return of a specific bandit algorithm
   3. Bandit algorithms differ in the rate of increase of their regrets  
        
      A bandit algorithm has linear graph regret if E(t) in O(t)   
      A bandit algorithm has logarithmic regret if E(t) in O(log(t))
6. -greedy exploration
   1. During the first |A| time steps, each action is selected once
   2. For each action a in A, the empirical mean ˆu\_a of its observed rewards is recorded
   3. For each time step t >= |A|, there is a parameter epsilon\_t in [0, 1]
   4. At each time step t >= |A|, the action with the highest empirical mean is selected with probability 1 − epsilon\_t and a random action is selected with probability epsilon\_t.
7. Subgaussianity
   1. A random variable X is sigma-subgaussian if, for all Lambda in R, Expectation of [exp(Lmabda X)] <= exp((Lmbda^2\*sigma^2) / 2)
   2. In simple terms, the tails of a sigma-subgaussian distribution decay approximately as fast as those of a Gaussian distribution with zero mean and variance sigma.
8. Epsilon-greedy exploration: regret
   1. Suppose that the reward for each action is a 1-subgaussian random variable
   2. Furthermore, suppose epsilon in (0, 1) and epsilon\_t = epsilon for every t
   3. In that case, if |A| >= 2, then  
        
      lim of t approaches infinity, E(t) / t = (epsilon / |A|) \* sum of (Delta\_a)  
        
      where, Delta\_a = u\_\* - u\_a is the so called suboptimality gap for action a
   4. Therefore, epsilon-greedy exploration with a fixed epsilon has linear regret
   5. This result can be generalised to sigma-subgaussian random variables by scaling rewards
   6. By choosing each epsilon\_t appropriately, epsilon-greedy exploration may achieve sublinear regret
9. Upper confidence bounds
   1. During the first |A| time steps, each action is selected once
   2. Each action a is associated with an upper confidence bound B\_a such that B\_a > u\_a with high probability
   3. At each time step t >= |A|, the action with the highest upper confidence bound is selected
   4. In simple terms, the upper confidence bound of every non-optimal action eventually falls below the upper confidence bound of every optimal action
   5. Upper confidence bounds employ the principle of optimism in the face of uncertainty
10. Upper confidence bound for 1-subgaussian variables
    1. Let X1, X2, . . . , X\_T be a sequence of independent 1-subgaussian variables with mean u
    2. If M = 1/T \* sum of(Xt) denotes the sample mean, then  
         
       P(u >= M + sqrt(2\*log(1/delta)/T)) <= delta  
       for any delta in (0,1)
    3. Intuitively, the result of adding a specific positive term to the sample mean almost certainly overestimates the underlying mean.
11. Upper confidence bounds
    1. Suppose that each action a in A has been selected at least once
    2. The upper confidence bound Ba for action a is given by  
         
       B\_a = u\_a + sqrt(2\*log(1/delta)/n\_a))  
       where u\_a is the empirical mean of the rewards observed for action a, n\_a is the number of rewards observed for action a, and delta is a parameter of the algorithm
    3. As mentioned before, at each time step t >= |A|, the action with the highest upper confidence bound is selected
    4. Intuitively, an action a is selected if u\_a is relatively high or if n\_a is relatively low
12. Upper confidence bounds: regret
    1. Suppose that the reward for each action is a 1-subgaussian random variable
    2. For any T >= |A|, if delta = 1/T^2 , then  
         
       E(T) <= 3\*sum of (Delta\_a) + sum of ((16 log T) / (Delta\_a))  
       u

where Delta\_a = u∗ − u\_a is the suboptimality gap for action a

* 1. Therefore, upper confidence bounds have logarithmic regret for appropriate delta
  2. This result can be generalized to sigma-subgaussian random variables by scaling rewards
  3. There are many other versions of upper confidence bounds that provide better performance in different settings

1. Posterior sampling
   1. Posterior sampling (Thompson sampling) is a Bayesian bandit algorithm that provides excellent performance in many settings
   2. Posterior sampling can be generalized to provide a reinforcement learning algorithm that learns a model of an environment while balancing exploration and exploitation
      1. The agent represents its knowledge by a distribution over models
      2. A single model is drawn from this distribution
      3. An optimal policy is found for this model
      4. This policy is used to interact with the environment for one episode
      5. The resulting data are used to update the distribution over models
      6. The process continues from Step 2
   3. Intuitively, increased certainty leads to decreased exploration
2. Monte Carlo control
   1. Monte Carlo control methods find an optimal policy without estimating the one-step dynamics by interleaving policy evaluation and policy improvement
   2. These methods require an episodic problem, where there is a transition to an absorbing state after a finite number of time steps
   3. Policy evaluation for pi consists of experiencing several episodes and averaging the returns that follow every possible state action pair (s, a) to obtain an estimate of Q^ pi (s, a).
   4. In practice, “policy improvement” based on π is performed before a reliable estimate of Q^ pi is available
   5. Monte Carlo control typically relies on -greedy policies to ensure that the environment is explored sufficiently
   6. For a given state s, an -greedy policy with respect to an estimate Q of the action value function chooses a random action with probability epsilon, and an action arg max\_a Q(s, a) with probability 1 – epsilon.
3. Monte Carlo control
   1. Algorithm 1 Monte Carlo control algorithm
   2. Input: set of states S, number of episodes N, probability of choosing random action epsilon.
   3. Output: deterministic policy pi, optimal when N → infinity.

1: for each s ∈ S do

2: for each action a ∈ A(s) do

3: Q(s, a) ← 0

4: n(s, a) ← 0

5: end for

6: end for

7: for each i in {1, . . . , N} do

8: Experience a new episode e following an -greedy policy based on Q.

9: for each state-action pair (s, a) in the episode e do

10: u ← return following (s, a) in the episode e.

11: n(s, a) ← n(s, a) + 1

12: Q(s, a) ← Q(s, a) + 1 n(s,a) [u − Q(s, a)]

13: end for

14: end for

15: for each state s ∈ S do

16: pi(s) ← arg maxa Q(s, a)

17: end for

1. Temporal difference
   1. Consider the tuple h\_t = (st , at ,rt+1,st+1, at+1) obtained by an agent using a policy pi to interact with an environment
   2. Let Q denote an estimate of the action value function Q^pi • The one-step return based on ht and Q is given by rt+1 + γQ(st+1, at+1)
   3. The temporal difference for (st , at) based on ht and Q is given by r\_t+1 + gamma\*Q(s\_t+1, a\_t+1) − Q(s\_t , a\_t)
   4. In other words, the difference between the immediate reward plus the (estimated) expected return from the next state and the (estimated) expected return for the current state
2. Sarsa control
   1. An algorithm bootstraps if it improves the estimate of the value of a state based on estimates of the values of other states
   2. Sarsa control is similar to Monte Carlo control, but it bootstraps based on temporal differences
   3. Sarsa control is comparatively more sample efficient, since it does not rely on the return that follows (s\_t , a\_t) after a single episode
   4. Given the tuple h\_t and the estimate Q, Sarsa control updates its estimate of Q(s\_t , a\_t) using Q(s\_t , a\_t) ← Q(s\_t , a\_t) + alpha[r\_t+1 + gamma\*Q(s\_t+1, a\_t+1) − Q(st , at)], where alpha is the so-called learning rate

Sarsa control

Algorithm 2 Sarsa control algorithm

Input: set of states S, number of episodes N, learning rate alpha, probability of random action epsilon, discount factor gamma.

Output: deterministic policy pi, optimal when N → infinity and alpha decays appropriately.

1: for each (s, a) in S × A do

2: Q(s, a) ← 0

3: end for

4: for each i in {1, . . . , N} do

5: s ← initial state for episode i

6: Select action a for state s according to an -greedy policy based on Q.

7: while state s is not terminal do

8: r ← observed reward for action a at state s

9: s 0 ← observed next state for action a at state s

10: Select action a 0 for state s 0 according to an epsilon-greedy policy based on Q.

11: Q(s, a) ← Q(s, a) + alpha[r + gamma\*Q(s’ , a‘ ) − Q(s, a)]

12: s ← s’

13: a ← a’

14: end while

15: end for

16: for each state s in S do

17: pi(s) ← arg max\_a Q(s, a)

18: end for

1. Q-learning
   1. An algorithm is off-policy if it learns about a policy that is different from the policy that it uses to act in the environment
   2. Q-learning learns about a greedy policy while acting using an epsilon-greedy policy
   3. Q-learning control is similar to Sarsa control: both algorithms bootstrap based on temporal differences
   4. Given the tuple ht and the estimate Q, Q-learning control updates its estimate of Q(st , at) using Q(st , at) ← Q(st , at) + alpha[rt+1 + gamma\*max Q(s\_t+1, a) − Q(st , at)] where alpha is the so-called learning rate

Q-learning control

Algorithm 3 Q-learning control algorithm

Input: set of states S, number of episodes N, learning rate alpha, probability of random action epsilon, discount factor gamma.

Output: deterministic policy pi, optimal when N → infinity and alpha decays appropriately.

1: for each (s, a) in S × A do

2: Q(s, a) ← 0

3: end for

4: for each i in {1, . . . , N} do

5: s ← initial state for episode i

6: while state s is not terminal do

7: Select action a for state s according to an epsilon-greedy policy based on Q.

8: r ← observed reward for action a at state s

9: s 0 ← observed next state for action a at state s

10: Q(s, a) ← Q(s, a) + alpha[r + gamma\* max Q(s’ , a’ ) − Q(s, a)]

11: s ← s’

12: end while

13: end for

14: for each state s in S do

15: pi(s) ← arg max Q(s, a)

16: end for