

- The basic idea of EM in this context is to pretend that we know the parameters of the model and then to infer the probability that each data point belongs to each component
- After that, we refit the components to the data,
 - where each component is fitted to the entire data set with each point weighted by the probability that it belongs to that component
- The process iterates until convergence
- Essentially we are “completing” the data by inferring probability distributions over the hidden binary variables Z_{ij} :

$$Z_{ij} = \begin{cases} 1, & \text{if datum } \mathbf{x}_j \text{ was generated by the } i\text{:th component} \\ 0, & \text{otherwise} \end{cases}$$



- For the mixture of Gaussians, we initialize the mixture-model parameters arbitrarily and then iterate the following two steps:
- 1. E-step:** Compute the probability that datum \mathbf{x}_j was generated by component i , $p_{ij} = P(C = i | \mathbf{x}_j)$
 - By Bayes rule we have $p_{ij} = \alpha P(\mathbf{x}_j | C = i) P(C = i)$
 - The term $P(\mathbf{x}_j | C = i)$ is just the probability at \mathbf{x}_j of the i :th Gaussian
 - $P(C = i)$ is just the weight parameter w_i for the i :th Gaussian
 - Define $n_i = \sum_j p_{ij}$, the effective number of data points currently assigned to component i

- 2. M-step:** Compute the new parameter values:

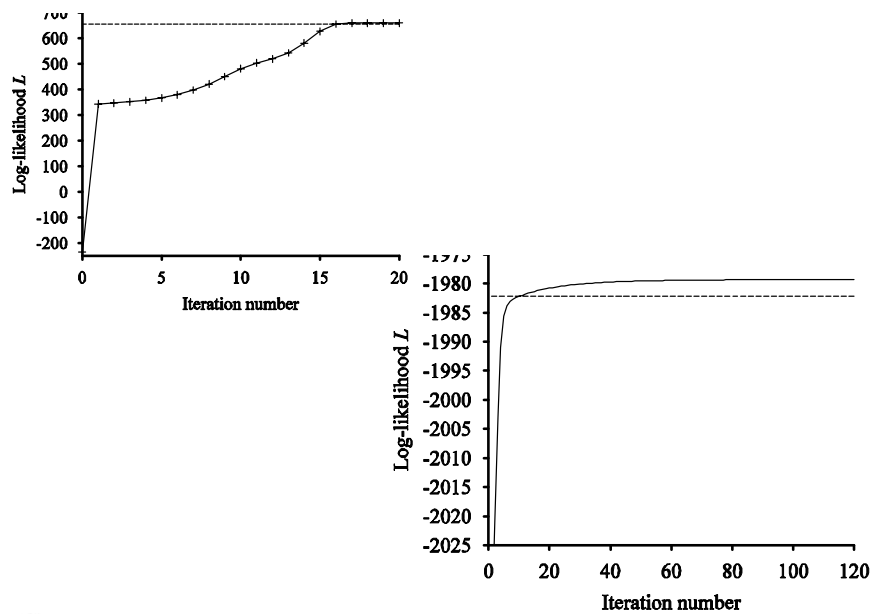
$$\mu_i \leftarrow \sum_j p_{ij} \mathbf{x}_j / n_i$$

$$\Sigma_i \leftarrow \sum_j p_{ij} (\mathbf{x}_j - \mu_i) (\mathbf{x}_j - \mu_i)^T / n_i$$

$$w_i \leftarrow n_i / N$$



- The E-step can be viewed as computing the expected values p_{ij} of hidden indicator variables Z_{ij}
- The M-step finds the new values of the parameters that maximize the log likelihood of the data, given the expected values p_{ij} of the hidden indicator variables
- EM increases the log likelihood of the data at every iteration
- Under certain (common) conditions, EM can be proven to reach a local maximum in likelihood (obs. no “step size”)
- Possible problems:
 - One Gaussian component may shrink to cover just one data point, variance = 0 \Rightarrow likelihood = ∞
 - Two components can “merge,” acquiring identical means and variances and sharing their data points



- In a Bayesian network hidden variables are the values of non-observed variables in each example
- In a hidden Markov model (HMM) the latent variables are the transition probabilities between states
- Hence, we get different instantiations of the EM algorithm for different probability models
- In its most general form the algorithm reduces to the update rule

$$\theta^{(i+1)} = \arg \max_{\theta} \sum_z P(\mathbf{Z} = \mathbf{z} \mid \mathbf{x}, \theta^{(i)}) L(\mathbf{x}, \mathbf{Z} = \mathbf{z} \mid \theta),$$

\mathbf{x} is all observed values in all the examples,
 \mathbf{Z} denotes all the hidden variables for all the examples,
 θ is all the parameters for the probability model
- The E-step is the computation of the summation, which is the expectation of the log likelihood
- The M-step is the maximization of this expected log likelihood with respect to the parameters



21 REINFORCEMENT LEARNING

- The task of reinforcement learning is to use the observed rewards to learn an optimal (or nearly optimal) policy for the environment
- A utility-based agent learns a utility function on states and uses it to select actions that maximize the expected outcome utility
 - Requires a model of the environment in order to make decisions, because it must know the states to which its actions will lead
 - E.g., a chess program must know what its legal moves are and how they affect the board position
- Q-learning
- An agent learns the expected utility of taking a given action in a given state
 - Now it is enough to know the moves, it is not necessary to know the board position



21.2 Passive Reinforcement Learning

- The agent's policy π is fixed: in state s , it always executes the action $\pi(s)$
- Its goal is simply to learn the utility function $U^\pi(s)$
- The agent does not know the transition model $P(s' | a, s)$ nor the reward function $R(s)$
- The agent executes a set of trials in the 4×3 grid
- In each trial, the agent starts in state $[1, 1]$ and experiences a sequence of state transitions until it reaches one of the terminal states
- The utility is defined to be the expected sum of discounted rewards obtained if policy π is followed:

$$U^\pi(s) = E \left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \right]$$



| | | | |
|---|---|---|----|
| → | → | → | +1 |
| ↑ | | ↑ | -1 |
| ↑ | ← | ← | ← |

| | | | |
|-------|-------|-------|-------|
| 0.812 | 0.868 | 0.918 | +1 |
| 0.762 | | 0.660 | -1 |
| 0.705 | 0.655 | 0.611 | 0.388 |



Direct utility estimation (DUE)

- Widrow & Hoff (1960)
- The utility of a state is the expected total reward from that state onward (*reward-to-go*)
- Each trial provides a sample of this quantity for each state visited
- For example the trial
 $[1,1]_{-0.04} \rightarrow [1,2]_{-0.04} \rightarrow [1,3]_{-0.04} \rightarrow [1,2]_{-0.04} \rightarrow [1,3]_{-0.04} \rightarrow [2,3]_{-0.04} \rightarrow [3,3]_{-0.04} \rightarrow [4,3]_{+1}$
provides a sample total reward of 0.72 for state [1,1], two samples of 0.76 and 0.84 for [1,2], two samples of 0.80 and 0.88 for [1,3], and so on
- At the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly
 - just by keeping a running average for each state in the table



- In the limit of infinitely many trials, the sample average will converge to the true expectation
- DUE is just an instance of supervised learning where each example has the state as input and the observed reward-to-go as output
- DUE, however, misses the fact that utilities of states are not independent
- The utility of each state equals its own reward plus the expected utility of its successor states (Bellman equations)
- For example if a trial reaches state [3,2] for the first time, transitioning to [3,3] – already visited and known to have high utility – should tell that also [3,2] is likely to have a high utility, like Bellman equations suggest immediately
- The algorithm often converges very slowly



Adaptive Dynamic Programming (ADP)

- An ADP agent solves the corresponding Markov decision process using a dynamic programming method
- Plugging the learned transition model and the observed rewards into the Bellman equations

$$U^\pi(s) = R(s) + \gamma \sum_{s'} P(s' | s, \pi(s)) U^\pi(s')$$

lets one calculate the utilities of states

- Because the policy is fixed, the transition model $P(s' | s, \pi(s))$ is easy to learn
- Just keep track of each action a occurs and estimate the transition probability $P(s' | s, a)$ from the frequency with which s' is reached when executing a in s
- The equations are linear (no maximization) and can be solved using any linear algebra package
- Intractable for large state spaces
 - E.g., backgammon $\approx 10^{50}$ equations and unknowns to solve



Temporal-difference learning (TD)

- Assume that $U^\pi(1, 3) = 0.84$ and $U^\pi(2, 3) = 0.92$
- If transition $[1, 3] \xrightarrow{-0.04} [2, 3]$ occurred all the time, then we would expect $U^\pi(1, 3) = -0.04 + U^\pi(2, 3) = 0.88$, so the current estimate 0.84 might be a little low and should be increased
- Use observed transitions to adjust the utilities of observed states
$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s)),$$
where α is the learning rate parameter
- The ideal equilibrium given by the Bellman equations is not reached with this update rule
- However, the average value of $U^\pi(s)$ will converge to the correct value
- If we change α to decrease with the number of times a state has been visited, then $U^\pi(s)$ itself will converge
- TD does not need a transition model at all!



- Whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore the consistency between the utility estimates U and the environment model P
- TD could use an environment model to generate several pseudoexperiences – imaginary transitions
- In this way, the resulting utility estimates will approximate more and more closely those of ADP
- Similarly, ADP could take into account only part of the transitions in adjusting the state utilities in order to come up with an efficient approximation algorithm
- **Prioritized sweeping** heuristic prefers to make adjustments to states whose likely successors have undergone a large adjustment in their own utility estimates
- Fast + efficient (time + training sequences)



21.3 Active Reinforcement Learning

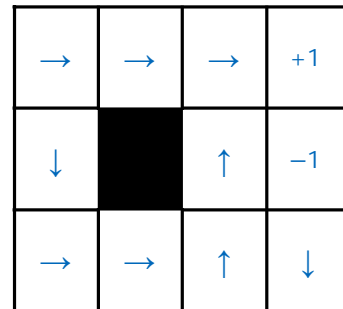
- As opposed to a passive agent, an active agent must determine:
 - What actions to take
 - What consequences does it have on the environment
 - How does it affect the rewards
- The utilities of the optimal policy obey the Bellman equations:

$$U(s) = R(s) + \gamma \cdot \max_a \sum_{s'} P(s' | s, a) U(s')$$
 and can be solved using the value iteration or policy iteration algorithms
- What to do at each step?
- Having obtained a utility function U that is optimal for the learned model, the agent should simply execute an optimal action (given by one-step look-ahead or policy)
- Or should it?



Exploration

- The optimal policy for the learned model is not necessarily the true optimal policy
- Sticking to the false policy means never learning utilities of other states and never finding the optimal route
- This agent is the greedy one
- Greedy agent very seldom converges to the optimal policy for the environment



- Actions do more than provide rewards according to the current learned model
- They also contribute to learning the true model by affecting the percepts that are received
- By improving the model, the agent will receive greater rewards in the future
- An agent therefore must make a tradeoff between exploitation to maximize its reward and exploration to maximize its long-term well-being
- Pure exploitation risks getting stuck in a rut
- Pure exploration to improve one's knowledge is of no use if one never puts that knowledge into practice
- With greater understanding, less exploration is necessary



- To promote exploration one can assign a higher utility estimate to relatively unexplored state-action pairs
- Essentially this amounts up to an optimistic prior over the possible environments
- Let $U^+(s)$ denote the optimistic estimate of the utility of state s , and let $N(s, a)$ be the number of times action a has been tried in state s
- Now the update rule can be written as

$$U^+(s) \leftarrow R(s) + \gamma \cdot \max_a f(\sum_{s'} P(s' | s, a) U^+(s'), N(s, a))$$
- $f(u, n)$ is called the exploration function. It determines how greed (u) is traded off against curiosity (n)
- The function should be increasing in u and decreasing in n
- A simple alternative:

$$f(u, n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

where R^+ is an optimistic estimate of the best possible reward and N_e is a fixed parameter



Exploration and Bandits



- A formal model of the exploitation/exploration dilemma
- An n -armed bandit has n levers
- The player must choose which lever to play on each successive coin
 - the one that has paid off best, or maybe one that has not been tried
- Differs from the expert setting in that we only get to know the payoff of the chosen lever
- Exploration is risky, expensive, and has uncertain payoffs
- On the other hand, failure to explore at all means that one never discovers any actions that are worthwhile
- In the bandit problem the aim is to maximize the expected total reward obtained over the agent's lifetime




Learning an action-utility function

- An alternative TD method called Q-learning learns an action-utility representation instead of learning utilities
- Let $Q(s, a)$ denote the value of doing action a in state s
- Q-values are directly related to utility values:
$$U(s) = \max_a Q(s, a)$$
- A TD agent that learns a Q-function does not need a model of the form $P(s' | s, a)$ either for learning or for action selection
- Therefore, Q-learning is called a model-free method
- At equilibrium, when the Q-values are correct, it must hold
$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' | s, a) \max_{a'} Q(s', a')$$
- This equation could be used directly as an update rule, but it would require a model of the environment
- The temporal-difference approach, on the other hand, requires no model of state transition



- The update equation for TD Q-learning is
$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$
which is calculated whenever action a is executed in state s leading to state s'
- A close relative to Q-learning is SARSA in which the update equation uses the action *actually taken* a' in the state reached s' rather than the best Q-value
$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma Q(s', a') - Q(s, a))$$
- For a greedy agent that always takes the action with best Q-value, the two algorithms are identical
- When exploration is happening, they differ significantly



- 
- Because Q-learning uses the best Q-value, it pays no attention to the actual policy being followed, whereas SARSA takes it into account
 - Q-learning is more flexible than SARSA; it can learn how to behave well even when guided by a random or adversarial exploration policy
 - On the other hand, SARSA is more realistic
 - If, for example, the overall policy is even partly controlled by other agents, it is better to learn a Q-function for what will actually happen rather than what the agent would like to happen
 - Whether to maintain a model or not is a fundamental question of the whole field of AI

