Reinforcement Learning

Brandon Shroyer

4/17/2012

HMM Homework

- Use likelihood function $P(X_1^N) = \sum_{Z_1^N} P(X_1^N, Z_1^N) = \sum_i \alpha(N, s_i)$
  - $P(X_1^N) = \sum_i \alpha(N, s_i) = \alpha(N+1, stop)$
  - $\Pi(s) = P(z_1 = s) = P(s|start)$
  - $Q(\theta, \theta^{old})$ = likelihood approximation given θ^{old}.
- Special end state is not always meaningful (with weather systems, for instance–they just keep going).
- To prevent α, β from going to zero, scale them by multiplying in a constant every ten frames or so.
- To remove scaling factor when plotting, take log and subtract log(const) from result.

Markov Decision Processes

A Markov Decision Process is an extension of the standard (unhidden) Markov model [1]. Each state has a collection of actions that can be performed in that particular state. These actions serve to move the system into a new state. More formally, the MDP's state transitions can be described by the transition function $T(s, a, s')$, where a is an action moving performable during the current state s, and s' is some new state.

As the name implies, all MDPs obey the *Markov property*, which holds that the probability of finding the system in a given state is dependent only on the previous state. Thus, the system's state at any given time is determined solely by the transition function and the action taken during the previous timestep:

$$P(S_t = s' | S_{t-1} = s,\ a_t = a) = T(s, a, s')$$

Each MDP also has a reward function $R : S \mapsto \mathbb{R}$. This reward function assigns some value $R(s)$ to being in the state $s \in S$. The goal of a Markov Decision Process is to move from the current state s to some final state in a way that a) maximizes $R(s)$ and b) maximizes R's potential value in the future.

Given a Markov Decision Process we wish to find a *policy* – a mapping from states to actions. The policy function $\Pi : S \mapsto A$ selects the appropriate action $a \in A$ given the current state $s \in S$.

Value Iteration

The consequences of actions (i.e., rewards) and the effects of policies are not always known immediately. As such, we need some mechanisms to control and adjust policy when the rewards of the current state space are uncertain. These mechanisms are collectively referred to as *reinforcement learning* [1].

One common way of trading off present reward against future reward is by introducing a *discount rate* γ. The discount rate is between 0 and 1, and we can use it to construct a weighted estimate of future rewards:

$$\sum_{t=0}^{\infty} \gamma^t r_t$$

Here, we assume that $t = 0$ is the current time. Since $0 < \gamma < 1$, greater values of t (indicating rewards farther in the future) are given smaller weight than rewards in the nearer future.

Let $V^{\Pi}(s)$ be the value function for the policy Π. This function $V^{\Pi} : S \mapsto \mathbb{R}$ maps the application of Π to some state $s \in S$ to some reward value. Assuming the system starts in state s_0, we would expect the system to have the value

$$V^{\Pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0 = s, \Pi\right]$$

Since the probability of the system being in a given state $s' \in S$ is determined by the transition function $T(s, a, s')$, we can rewrite the formula above for some arbitrary state $s \in S$ as

$$V^{\Pi}(s) = R(s) + \sum_{s'} T(s, a, s')\gamma V^{\Pi}(s')$$

where $a = \Pi(s)$ is the action selected by the policy for the given state s.

Our goal here is to determine the optimal policy Π*(s). Examining the formula above, we see that $R(s)$ is unaffected by choice of policy. This makes sense because at any given state s, local reward term $R(s)$ is determined simply by virtue of the fact that the system is in state s. Thus, if we wish to find the maximum policy value function (and therefore find the optimum policy) we must find the action a that maximizes the summation term above:

$$V^{\Pi^*}(s) = R(s) + \max_a \sum_{s'} T(s, a, s')\gamma V^{\Pi^*}(s')$$

Note that this formulation assumes that the number of states is finite.

Algorithm 1 Value Iteration

1. Initialize $V(s)$.
2. Repeat until converged:
   (a) for all $s \in S$:
      i. $R(s) = R(s) + \max_a \sum_{s'} T(s, a, s')\gamma V(s')$
      ii. $\Pi(s) = \operatorname{argmax}_a Q(s, a)$

The formula above forms the basis of the *value iteration* algorithm. This operation starts with some initial policy value function guess and iteratively refines $V(s)$ until some acceptable convergence is reached:

Each pass of the value iteration maximizes $V(s)$ and assigns to $\Pi^*(s)$ the action a that maximizes $V(s)$. The function $Q(s, a)$ represents the potential value for $V(s)$ produced by the action $a \in A$.

Q-Learning

The example of value iteration above presumes that the transition function $T(s, a, s')$ is known. If the transition function is not known, then the function $Q(s, a)$ can be obtained through a similar process of iterative learning, the aptly-named *Q-learning*.

The naive guess for a Q-learning formula would be one that closely resembles the policy value function, such as

$$Q(s, a) = R(s) + \gamma \left[\max_{a'} Q_{old}(s, a')\right]$$

This formula aggressively replaces old values of Q, though, which is not always desirable. For better results [1], use a weighted average learning function:

$$Q(s, a) = \eta Q_{old}(s) + (1 - \eta)\gamma \left[\max_{a'} Q_{old}(s, a')\right]$$

where η is a user-selected learning parameter.

This formula can be turned into an algorithm much like the value iteration algorithm above. The only difference is that the action is selected by a user-defined function $f(s)$, which returns the appropriate policy action $\Pi(s)$ most of the time, but occasionally selects a random action to blunt the effects of sampling bias.

The Q-learning algorithm below is from Ballard's textbook [1]:

Algorithm 2 One-Step Q-Learning

1. Initialize $\Pi(s)$ to $\underset{a}{\operatorname{argmax}}\, Q(s,a)$.
2. Repeat until Π converges:
   (a) For each $s \in S$:
      i. Select an action $a = f(s)$.
      ii. $Q(s,a) = \eta Q_{old}(s) + (1-\eta)\gamma \left[\max_{a'} Q_{old}(s,a')\right]$
      iii. $\Pi(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a)$.

Temporal Difference Learning

One disadvantage of value iteration is that it can take a long time for updates to later states to propagate back to earlier states. For instance, an MDP attempting to navigate a maze would see its reward function jump once it reaches the final stage N, but it would take N iterations for the effects of that jump propagate back to stage 1.

Temporal difference learning remedies this by modifying the output weights for each state with estimates of the policy value for the next state:

$$\Delta w = \eta \left(R(s_t) + \gamma V(s_{t+1}) - V(s_t)\right) \sum_{k=1}^{t} \lambda^{t-k} \frac{\partial V_k}{\partial w}$$

Where $0 < \lambda < 1$ and $V(s)$ is presumed to be a function of the weight w. The temporal difference learning algorithm below is adapted from [1]:

Algorithm 3 Temporal Difference Learning

1. Initialize all network weights to random values.
2. Repeat until Π converges:
   (a) Select current state s_t.
   (b) Use $\Pi(s_t)$ to obtain a new state s_{t+1} and calculate $V(s_{t+1})$.
   (c) If an actual reward is available, substitute that for estimate $V(s_{t+1})$.
   (d) $w = w + \eta \left(R(s_t) + \gamma V(s_{t+1}) - V(s_t)\right) \sum_{k=1}^{t} \lambda^{t-k} \frac{\partial V_k}{\partial w}$.

References

1. Ballard, D. H. *An Introduction to Natural Computation.* MIT Press, 1997.
2. Bishop, C. M. *Pattern Recognition and Machine Learning.* Springer, 2006.