12. Bellman's principle

12.1. **States and Actions.** Let $x \in \mathcal{X}$ denote the state of an agent's environment, and let $\mathcal{U}(x)$ denote the space of allowable actions, given that your current state is x. For now both $\mathcal{X}$ and $\mathcal{U}(x)$ are finite sets. For $u \in \mathcal{U}(x)$, let

$$\text{next}(x; u) \in \mathcal{X}$$

denote the state which results from applying action u in state x, and

$$\text{cost}(x; u) \geq 0$$

the cost of applying action u in state x. A *policy* (also called a *control law* is a rule $u = \pi(x)$ for choosing an action, depending on which state you are in.

As an example, x may be the city where we are now, u the flight we take, $\text{next}(x; u)$ the city where that flight lands, and $\text{cost}(x; u)$ the price of the ticket. What is the cheapest way to your destination? An example of a "policy" could be "take the cheapest flight which lands closer to the destination than my current location." One can easily find examples in which this policy is not optimal. My dad recently found that the cheapest flight path from Columbia, SC to New York went through Atlanta, which is actually further from New York.

The solution to problems of the above type is derived starting from a beautiful and simple piece of intuition, now known as *Bellman's principle of optimality*:

> *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*
>
> — Bellman (1957)

Returning to the airline flight example, this says the following. Consider the optimal path from Columbia to New York. If we know that the first step is to Atlanta, then the rest of the steps must constitute the optimal path from Atlanta to New York.

This problem can be formalized as: find an action sequence $u_0, u_1, \ldots, u_{n-1}$ and corresponding state sequence $x_0, x_1, \ldots, x_{n-1}$ minimizing the total cost

$$J(\mathbf{x}; \boldsymbol{u}) = \sum_{k=0}^{n-1} \text{cost}(x_k; u_k)$$

where

$$x_{k+1} = \text{next}(x_k; u_k) \text{ and } u_k \in \mathcal{U}(x_k).$$

The initial state $x_0 = x_{\text{init}}$ is usually given, and considered fixed. In some variations of the problem, the destination state $x_n = x_{\text{dest}}$ may also be given as an additional constraint, while in other variations, the endpoint of the path is also something we have to solve for.

Optimal state-action sequences can be constructed by starting at the final state and extending backwards. Key to this procedure is the optimal value function

$$v(x) = \text{“minimal total cost for completing the task starting from state } x\text{”}$$

This function captures the long-term cost for starting from a given state. Moreover, if we knew the function $v(x)$, we could always find the optimal action to take for whatever current state we are in – a process which we formalize as Algorithm 12.1.

Algorithm 12.1. Consider every action $u \in \mathcal{U}(x)$ available at the current state x. Add the immediate cost of the action, $\text{cost}(x, u)$, to the optimal value $v(n)$ of the next state, $n := \text{next}(x, u)$, that results from taking that action. Choose an action for which the sum

$$\text{immediate cost} \;+\; \text{value of next state}$$

is minimal.

Algorithm 12.1 is called a *greedy algorithm* because it considers only the action with the least cost, rather than, say, randomizing over several nearby actions.

Formally, an *optimal control law* (also called an *optimal policy*) π satisfies

$$\pi(x) = \operatorname*{argmin}_{u \in \mathcal{U}(x)} \{\text{cost}(x, u) + v(\text{next}(x, u))\} \tag{12.1}$$

The minimum in (12.1) may be achieved for multiple actions in the set $\mathcal{U}(x)$, i.e. $\pi(x)$ may not be unique. However the optimal value function v is always uniquely defined, and satisfies

$$v(x) = \min_{u \in \mathcal{U}(x)} \{\text{cost}(x, u) + v(\text{next}(x, u))\} \tag{12.2}$$

Equations (12.1) and (12.2) are the *Bellman equations*. They mathematically encode Bellman's principle of optimality expressed in words above.

We can visualize this setting with a directed graph where the states are nodes and the actions are arrows connecting the nodes. In a directed graph, each edge goes in only one direction, and is drawn as an arrow. In a directed acyclic graph (DAG), there are no cycles. In a weighted DAG, a numerical weight is associated with each edge.

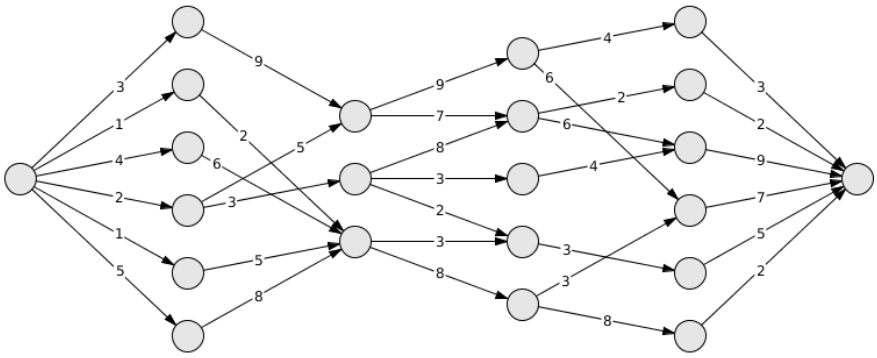

FIGURE 12.1. Weighted DAG example.

The Dynamic programming (DP) algorithm for weighted DAGs can be stated in the following way.

Forward pass:

Set value of each vertex to 0

For each vertex:

For each predecessor of the current vertex:

Let w = (predecessor vertex's value) + (weight of edge from predecessor vertex to current vertex)

Current vertex's value = minimum w over all predecessors

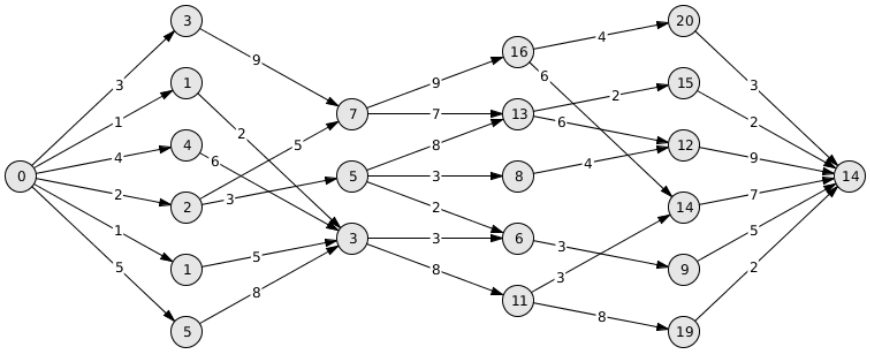

FIGURE 12.2. Example DAG after the forward pass.

Backward pass:

> Start with the smallest-value destination vertex
>
> Repeat:
>
> > Find the predecessor vertex such that (current vertex's value) = (predecessor vertex's value) + (weight of edge from predecessor vertex to current vertex)
> >
> > Go to that predecessor vertex
>
> Until you reach a source vertex.

The backward pass traces the minimum-weight path.

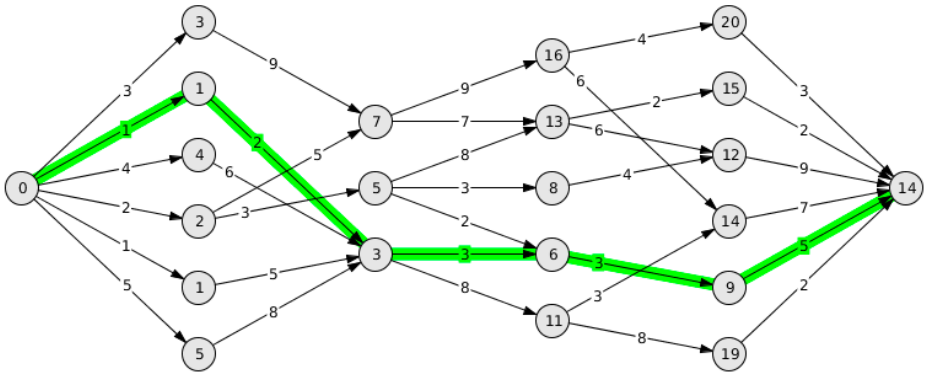

FIGURE 12.3. Example DAG after the backward pass.

The problems considered thus far are deterministic, in the sense that applying action u at state x always yields the same next state $\text{next}(x; u)$. Dynamic programming easily generalizes to the stochastic case where we have a probability distribution over possible next states:

$$p(y \mid x; u) = \text{probability that } \text{next}(x, u) = y$$

Then the value function becomes

$$v(x) = \min_{u \in \mathcal{U}(x)} \{\text{cost}(x, u) + \mathbb{E}\left[v(\text{next}(x, u))\right]\} \tag{12.3}$$

where

$$\mathbb{E}\left[v(\text{next}(x, u))\right] = \sum_{y \in \mathcal{X}} p(y \mid x, u) v(y).$$

12.2. **The Generalized Viterbi Algorithm.** We now discuss the *Viterbi algorithm* after Andrew Viterbi, who proposed it Viterbi (1967) as a decoding algorithm for convolutional codes over noisy digital communication links. However, it is really a straightforward application of Bellman's principle and the associated recursive

equations, which were already published in book form in 1957. The Viterbi algorithm for hidden Markov models is a special case of the general version we present below.

The setting of the Viterbi algorithm is as follows. Assume that we explicitly account for time, and each action u, which (potentially) results in us moving to a new state, takes one unit of time. The only possible actions are of the form "move to a new state" so rather than $\text{cost}(x, u)$ we can more simply write $\text{cost}(x, x')$ where u is the action "move to x'." The more general setup above is flexible enough to allow different actions with different costs which still effect the same state transition, eg. flying first class versus flying coach. Let S_t denote the space of states at time t, which we assume is finite. Let $K = \max_t |S_t|$ denote the maximum cardinality of any state space.

Algorithm 12.2.

(1) "Forward pass:"
Initialize with Let $\delta_0(i) = 0$ for all i. For $t = 1, 2, \ldots, T$, for $1 \leq j \leq K$, compute
$$\delta_t(j) = \min_i \left[\delta_{t-1}(i) + \text{cost}(x_{t-1}^{(i)}, x_t^{(j)})\right] \tag{12.4}$$
Let $\psi_t(j)$ be the value of i that minimizes the quantity in brackets. Save the values of $\psi_t(j)$ for all t, j.

(2) "Determine Final State:"
Let $i_T = \text{argmin}_i \, \delta_T(i)$.

(3) "Backward pass:"
For $t = T-1, T-2, \ldots, 1$, let $i_t = \psi_{t+1}(i_{t+1})$ and $x_t^* = x_t^{(i_t)}$.

The optimal path is $x_t^* = x_t^{(i_t)}$ for all t. Note that It is not necessary to save the values of $\delta_t(j)$ in order to compute the backward pass. It is only necessary to save one prior time-step $\delta_{t-1}(j)$ in order to handle the recursive calculation of $\delta_t(j)$ in (12.4). This is indeed equivalent to the Dynamic programming (DP) algorithm for weighted DAGs discussed previously.

By convention, $x_{t-1}^{(i)} = x_0$, the single initial state, for all i. Hence, for $t = 1$ the optimization in (12.4) degenerates to a single evaluation. Applications of this algorithm are numerous and include state estimation in hidden Markov models, multiperiod portfolio optimization, computing optimal flight connections, artificial intelligence for multi-stage planning and navigation tasks, and many others.

In the context of Hidden Markov models, this gives a way of estimating the maximum a-posteriori sequence (which we called the smoothed sequence in the lecture on Kalman filter/smoother), when the distributions involved are not Gaussian.

To apply the above to Hidden Markov models, simply replace every instance of "cost" with "negative log-probability." Then instead of a minimum-cost path, it finds a maximum log-probability path. The probability density for a hidden Markov model takes the form

$$p(x_0)\prod_{t=1}^{T} p(x_t \mid x_{t-1})p(y_t \mid x_t)$$

Hence the log-probability is the sum

$$\log p(x_0) + \sum_{t=1}^{T}[\log p(x_t \mid x_{t-1}) + \log p(y_t \mid x_t)]$$

One identifies the control u as making a transition from $x_{t-1} \to x_t$ with associated cost $-\log p(x_t \mid x_{t-1})$ and the total cost $\text{cost}(x, u)$ includes this transition cost plus a state cost $-\log p(y_t \mid x_t)$.

One would of course have to somehow create a finite state space in order to apply Algorithm 12.2. To do this, it's very important to generate states which are in the region of highest probability, and hence very likely to be near the maximum posterior region. In other words, one would need to sample state sequences from the posterior. There is a good general way to do this which we will discuss in a subsequent lecture, but this motivates why one would be interested.

If the distributions involved are Gaussian, one should of course continue to use the Kalman technology, since it will be computationally efficient, and is exact whereas the Viterbi method is limited by the mesh size of the grid.

12.3. **Applications to Machine Learning.** Many intelligent actions are deemed "intelligent" precisely because they are optimal interactions with an environment; an algorithm plays a computer game intelligently if it can optimize the score. A robot navigates intelligently if it finds a shortest path with no collisions.

Formulating an intelligent behavior as a reinforcement learning problem begins with identification of the state space $\mathcal{S}$. The relation with what statisticians call "state-space models" is direct and intuitive: underlying most reinforcement learning problems is a Markov Decision Process (MDP).

In reinforcement learning, the *environment* is defined to be everything outside the agent's *direct* control. The agent can still have arbitrarily complete knowledge about the environment, and the environment may be indirectly affected by the agent's actions. The term *state*, in reinforcement learning problems, usually refers to the *state of the environment*.

In a general reinforcement-learning setup, the agent observes the state, and chooses an action according to some policy. This choice influences both the transition to the next state, as well as the reward the agent receives. More precisely, there is assumed to be a distribution $p(s', r \mid s, a)$ for the joint probability of transitioning

to state $s' \in \mathcal{S}$ and receiving reward r, conditional on the previous state being s and the agent taking action a.

A *policy*, π, is a mapping from a state s to a probability distribution over actions: $\pi(a \mid s)$ is the probability of taking action a when in state s. The policy will be called *deterministic* if there is only one action with nonzero probability, in which case π is a mapping from the current state to the next action.

Following the notation of Sutton, Barto, and Williams (1992) and Sutton and Barto (1998), the sequence of rewards received after time step t is denoted

$$R_{t+1}, R_{t+2}, R_{t+3}, \ldots$$

The agent's goal is to maximize the expected cumulative reward, denoted by

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \tag{12.5}$$

The agent then searches for policies which maximize $\mathbb{E}[G_t]$. The sum in (12.5) can be either finite or infinite. The constant $\gamma \in [0, 1]$ is known as the *discount rate*, and is especially useful in considering the problem with $T = \infty$, in which case γ is needed for convergence.

According to Sutton and Barto (1998), "the key idea of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies." The *state-value function* for policy π is

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

where $\mathbb{E}_\pi$ denotes the expectation under the assumption that policy π is followed. Similarly, the *action-value function* expresses the value of starting in state s, taking action a, and then following policy π thereafter:

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

Policy π is defined to be at least as good as π' if

$$v_\pi(s) \geq v_{\pi'}(s)$$

for all states s. An *optimal policy* is defined to be one which is at least as good as any other policy. There need not be a unique optimal policy, but all optimal policies share the same optimal state-value function

$$v_*(s) = \max_\pi v_\pi(s)$$

and optimal action-value function

$$q_*(s, a) = \max_\pi q_\pi(s, a).$$

The state-value function and action-value function satisfy Bellman optimality equations

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma\, v_*(s')]$$

$$q_*(s, a) = \sum_{s',r} p(s', r \mid s, a)[r + \gamma \max_{a'} q_*(s', a')]$$

where the sum over s', r denotes a sum over all states s' and all rewards r. In a continuous formulation, these sums would be replaced by integrals.

If we possess a function $q(s, a)$ which is an estimate of $q_*(s, a)$, then the *greedy policy* is defined as picking at time t the action a_t^* which maximizes $q(s_t, a)$ over all possible a, where s_t is the state at time t.

To ensure that, in the limit as the number of steps increases, every action will be sampled an infinite number of times we use an ϵ-*greedy policy*: with probability $1 - \epsilon$ follow the greedy policy, while with probability ϵ uniformly sample the action space.

An important breakthrough in reinforcement learning came when Watkins (1989) and Watkins and Dayan (1992) suggested an iterative method which converges to the optimal action-value function q_*. The algorithm consists of the following steps. One initializes a matrix Q with one row per state, and one column per action. This matrix can be initially the zero matrix, or initialized with some prior information if available. Let S denote the current state.

Repeat the following steps until a pre-selected convergence criterion is obtained:

1. Choose action $A \in \mathcal{A}$ using a policy derived from Q (for example, the ϵ-greedy policy described above)
2. Take action A, after which the new state of the environment is S' and we observe reward R
3. Update the value of $Q(S, A)$: set target $= R + \gamma \max_a Q(S', a)$ and

$$Q(S, A) \mathrel{+}= \alpha[\underbrace{\text{target} - Q(S, A)}_{\text{TD-error}}] \tag{12.6}$$

   where $\alpha \in (0, 1)$ is recomputed each step as described below.

The update rule is of a form that occurs frequently. The general form is

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize} \cdot [\text{Target} - \text{OldEstimate}].$$

The expression [Target − OldEstimate] is interpreted as a error term. The error is presumably reduced by taking a step toward the "Target." The target is presumed to indicate a desirable direction in which to move, though it may be noisy.

Let $\alpha n(a)$ denote the step-size parameter used to process the reward received after the n-th selection of action a. The choice $\alpha_n(a) = 1/n$ results in the sample-average method, which is guaranteed to converge to the true action values by the law of large numbers. But of course convergence is not guaranteed for all choices of the sequence $\alpha_n(a)$. A result in stochastic approximation theory gives us the conditions required to assure convergence with probability 1:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \alpha_n(a)^2 < \infty.$$

The first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps become small enough to assure convergence. Note that both convergence conditions are met for the sample-average case, $\alpha_n(a) = 1/n$, but not for the case of constant step-size parameter, $\alpha_n(a) = \alpha$.

12.4. **Applications to Trading.** In trading problems, the environment should be interpreted to mean all processes generating observable data that the agent will use to make a trading decision. Let s_t denote the state of the environment at time t; the state is a data structure containing all of the information the agent will need in order to decide upon the action. This will include the agent's current position, which is clearly an observable that is an important determinant of the next action.

At time t, the state s_t may also contain the prices p_t, but beyond that, much more information may be considered. In order to know how to interact with the market microstructure and what the trading costs will be, the agent may wish to observe the bid-offer spread and liquidity of the instrument. If the decision to trade is driven by a *signal* – something which is supposed to be predictive of future returns – then that signal is part of the environment; the state would necessarily contain the signal. If the process is to be Markov, then the action decision must not depend on the whole history of states, hence the state itself must be a sufficiently rich data structure to make the optimal decision.

Fundamental Question 12.1. Can an artificial intelligence discover the *optimal* dynamic trading strategy in the presence of transaction costs, without being told what kind of strategy to look for?

In finance, *optimal* means that the strategy optimizes expected utility of final wealth:

$$\text{maximize: } \mathbb{E}[u(w_T)] = \mathbb{E}[u(w_0 + \sum_{t=1}^{T} \delta w_t)] \tag{12.7}$$

where

$$\delta w_t := w_t - w_{t-1}$$

is the change in wealth, and u is the investor's utility function.

Assume the asset returns follow an elliptical distribution (ie. one whose surfaces of equal probability are multi-dimensional ellipsoids). Then for any concave, increasing utility function u, there exists some $\kappa > 0$ such that the solution to:

$$\text{maximize}: \ \left\{\mathbb{E}[w_T] - \frac{\kappa}{2}\mathbb{V}[w_T]\right\} \tag{12.8}$$

is the same as the solution to

$$\text{maximize: } \mathbb{E}[u(w_T)]$$

This property is called *mean-variance equivalence*. It means that the utility function does not need to be the one that Professor Bernoulli suggested. The multivariate normal is elliptical. So are many heavy-tailed distributions (such as the multivariate Student-t). Do not let anyone tell you that mean-variance optimization is somehow connected with the normal distribution!

For a reinforcement learning approach to match (12.8), we need R_t to be an appropriate function of wealth increments, such that the following relation is satisfied:

$$\mathbb{E}[R_t] = \mathbb{E}[\delta w_t] - \frac{\kappa}{2}\mathbb{V}[\delta w_t]$$

One such function is,

$$R_t := \delta w_t - \frac{\kappa}{2}(\delta w_t - \hat{\mu})^2 \tag{12.9}$$

where $\hat{\mu}$ is an estimate of a parameter representing the mean wealth increment over one period, $\mu := \mathbb{E}[\delta w_t]$.

Estimation of the parameter $\hat{\mu}$ in this context is slightly circular: $\mathbb{E}[\delta w_t]$ depends on the optimal policy, which depends on the reward function, which depends on $\hat{\mu}$. We propose that, at least initially, one use the trivial biased estimator $\hat{\mu} = 0$. This will have the effect that we overestimate variance in the reward function during an initial burn-in period. This over-estimation will not be too large. Once the value function has sufficiently converged using the approximate reward function, one may then begin to estimate $\hat{\mu}$ by the sample average.

References

Bellman, Richard (1957). *Dynamic Programming*.

Sutton, Richard S and Andrew G Barto (1998). *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge.

Sutton, Richard S, Andrew G Barto, and Ronald J Williams (1992). "Reinforcement learning is direct adaptive optimal control". In: *IEEE Control Systems* 12.2, pp. 19–22.

Viterbi, Andrew J (1967). "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm". In: *Information Theory, IEEE Transactions on* 13.2, pp. 260–269.

Watkins, Christopher JCH (1989). "Q-learning". In: *PhD Thesis*.

Watkins, Christopher JCH and Peter Dayan (1992). "Q-learning". In: *Machine learning* 8.3-4, pp. 279–292.