CHAPTER 10

Sequential models

So far, we have limited our attention to domains in which each output y is assumed to have been generated as a function of an associated input x, and our hypotheses have been "pure" functions, in which the output depends only on the input (and the parameters we have learned that govern the function's behavior). In the next few weeks, we are going to consider cases in which our models need to go beyond functions.

- In *recurrent neural networks*, the hypothesis that we learn is not a function of a single input, but of the whole sequence of inputs that the predictor has received.
- In *reinforcement learning*, the hypothesis is either a *model* of a domain (such as a game) as a recurrent system or a *policy* which is a pure function, but whose loss is determined by the ways in which the policy interacts with the domain over time.

Before we engage with those forms of learning, we will study models of sequential or recurrent systems that underlie the learning methods.

1 State machines

A *state machine* is a description of a process (computational, physical, economic) in terms of its potential sequences of *states*.

The *state* of a system is defined to be all you would need to know about the system to predict its future trajectories as well as possible. It could be the position and velocity of an object or the locations of your pieces on a game board, or the current traffic densities on a highway network.

Formally, we define a *state machine* as $(\mathcal{S}, \mathcal{X}, \mathcal{Y}, s_0, f, g)$ where

- $\mathcal{S}$ is a finite or infinite set of possible states;
- $\mathcal{X}$ is a finite or infinite set of possible inputs;
- $\mathcal{Y}$ is a finite or infinite set of possible outputs;
- $s_0 \in \mathcal{S}$ is the initial state of the machine;
- $f : \mathcal{S} \times \mathcal{X} \to \mathcal{S}$ is a *transition function*, which takes an input and a previous state and produces a next state;

This is such a pervasive idea that it has been given many names in many subareas of computer science, control theory, physics, etc., including: *automaton*, *transducer*, *dynamical system*, *system*, etc.

There are a huge number of major and minor variations on the idea of a state machine. We'll just work with one specific one in this section and another one in the next, but don't worry if you see other variations out in the world!

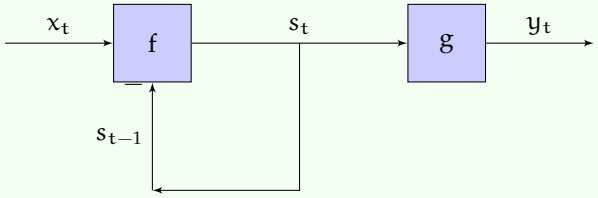
- $g : \mathcal{S} \to \mathcal{Y}$ is an *output function*, which takes a state and produces an output.

The basic operation of the state machine is to start with state s_0, then iteratively compute for $t \geqslant 1$:

In some cases, we will pick a starting state from a set or distribution.

$$s_t = f(s_{t-1}, x_t)$$
$$y_t = g(s_t)$$

The diagram below illustrates this process. Note that the "feedback" connection of s_t back into f has to be buffered or delayed by one time step—-otherwise what it is computing would not generally be well defined.



So, given a sequence of inputs $x_1, x_2, \ldots$ the machine generates a sequence of outputs

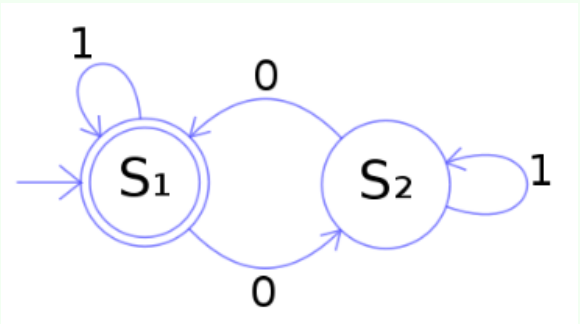
$$\underbrace{g(f(s_0, x_1))}_{y_1}, \underbrace{g(f(f(s_0, x_1), x_2))}_{y_2}, \ldots \quad .$$

We sometimes say that the machine *transduces* sequence x into sequence y. The output at time t can have dependence on inputs from steps 1 to t.

One common form is *finite state machines*, in which $\mathcal{S}$, $\mathcal{X}$, and $\mathcal{Y}$ are all finite sets. They are often described using *state transition diagrams* such as the one below, in which nodes stand for states and arcs indicate transitions. Nodes are labeled by which output they generate and arcs are labeled by which input causes the transition.

All computers can be described, at the digital level, as finite state machines. Big, but finite!

One can verify that the state machine below reads binary strings and determines the parity of the number of zeros in the given string. Check for yourself that all inputted binary strings end in state S_1 if and only if they contain an even number of zeros.



Another common structure that is simple but powerful and used in signal processing and control is *linear time-invariant (LTI) systems*. In this case, $\mathcal{S} = \mathbb{R}^m$, $\mathcal{X} = \mathbb{R}^l$ and $\mathcal{Y} = \mathbb{R}^n$, and f and g are linear functions of their inputs. In discrete time, they can be defined by a linear difference equation, like

$$y[t] = 3y[t-1] + 6y[t-2] + 5x[t] + 3x[t-2] \quad ,$$

(where $y[t]$ is y at time t) and can be implemented using state to store relevant previous input and output information.

We will study *recurrent neural networks* which are a lot like a non-linear version of an LTI system, with transition and output functions

$$f(s, x) = f_1(W^{sx}x + W^{ss}s + W_0^{ss})$$
$$g(s) = f_2(W^0 s + W_0^0)$$

defined by weight matrices

$$W^{sx} : m \times \ell$$
$$W^{ss} : m \times m$$
$$W_0^{ss} : m \times 1$$
$$W^0 : n \times m$$
$$W_0^0 : n \times 1$$

and activation functions f_1 and f_2. We will see that it's actually possible to learn weight values for a recurrent neural network using gradient descent.

2 Markov decision processes

A *Markov decision process* (MDP) is a variation on a state machine in which:

- The transition function is *stochastic*, meaning that it defines a probability distribution over the next state given the previous state and input, but each time it is evaluated it draws a new state from that distribution.
- The output is equal to the state (that is g is the identity function).
- Some states (or state-action pairs) are more desirable than others.

An MDP can be used to model interaction with an outside "world," such as a single-player game.

We will focus on the case in which $\mathcal{S}$ and $\mathcal{X}$ are finite, and will call the input set $\mathcal{A}$ for *actions* (rather than $\mathcal{X}$). The idea is that an agent (a robot or a game-player) can model its environment as an MDP and try to choose actions that will drive the process into states that have high scores.

Formally, an MDP is $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$ where:

- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a *transition model*, where

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

  specifying a conditional probability distribution;
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function, where $R(s, a)$ specifies how desirable it is to be in state s and take action a; and
- $\gamma \in [0, 1]$ is a *discount factor*, which we'll discuss in section 2.2.

A *policy* is a function $\pi : \mathcal{S} \to \mathcal{A}$ that specifies what action to take in each state.

Recall that stochastic is another word for *probabilistic*; we don't say "random" because that can be interpreted in two ways, both of which are incorrect. We don't pick the transition function itself at random from a distribution. The transition function doesn't pick its output *uniformly* at random.

There is an interesting variation on MDPs, called a *partially observable* MDP, in which the output is also drawn from a distribution depending on the state.

And there is an interesting, direct extension to two-player zero-sum games, such as Chess and Go.

The notation here uses capital letters, like S, to stand for random variables and small letters to stand for concrete values. So S_t here is a random variable that can take on elements of $\mathcal{S}$ as values.

2.1 Finite-horizon solutions

Given an MDP, our goal is typically to find a policy that is optimal in the sense that it gets as much total reward as possible, in expectation over the stochastic transitions that the domain makes. In this section, we will consider the case where there is a finite *horizon* H, indicating the total number of steps of interaction that the agent will have with the MDP.

2.1.1 Evaluating a given policy

Before we can talk about how to find a good policy, we have to specify a measure of the goodness of a policy. We will do so by defining for a given MDP policy π and horizon h, the "horizon h *value*" of a state, $V_\pi^h(s)$. We do this by induction on the horizon, which is the *number of steps left to go*.

The base case is when there are no steps remaining, in which case, no matter what state we're in, the value is 0, so

$$V_\pi^0(s) = 0 \ .$$

Then, the value of a policy in state s at horizon $h+1$ is equal to the reward it will get in state s plus the next state's expected horizon h value. So, starting with horizons 1 and 2, and then moving to the general case, we have:

$$\begin{aligned}
V_\pi^1(s) &= R(s, \pi(s)) + 0 \\
V_\pi^2(s) &= R(s, \pi(s)) + \sum_{s'} T(s, \pi(s), s') \cdot R(s', \pi(s')) \\
&\vdots \\
V_\pi^h(s) &= R(s, \pi(s)) + \sum_{s'} T(s, \pi(s), s') \cdot V_\pi^{h-1}(s')
\end{aligned}$$

The sum over s' is an *expected value*: it considers all possible next states s', and computes an average of their $(h-1)$-horizon values, weighted by the probability that the transition function from state s with the action chosen by the policy, $\pi(s)$, assigns to arriving in state s'.

Study Question: What is $\sum_{s'} T(s, a, s')$ for any particular s and a?

Then we can say that a policy π_1 is better than policy π_2 for horizon h, i.e. $\pi_1 >_h \pi_2$, if and only if for all $s \in \mathcal{S}$, $V_{\pi_1}^h(s) \geqslant V_{\pi_2}^h(s)$ and there exists at least one $s \in \mathcal{S}$ such that $V_{\pi_1}^h(s) > V_{\pi_2}^h(s)$.

2.1.2 Finding an optimal policy

How can we go about finding an optimal policy for an MDP? We could imagine enumerating all possible policies and calculating their value functions as in the previous section and picking the best one...but that's too much work!

The first observation to make is that, in a finite-horizon problem, the best action to take depends on the current state, but also on the horizon: imagine that you are in a situation where you could reach a state with reward 5 in one step or a state with reward 10 in two steps. If you have at least two steps to go, then you'd move toward the reward 10 state, but if you only have step left to go, you should go in the direction that will allow you to gain 5!

One way to find an optimal policy is to compute an *optimal action-value function*, Q. For the finite-horizon case, we define $Q^h(s, a)$ to be the expected value of

- starting in state s,

- executing action a, and
- continuing for $h-1$ more steps executing an optimal policy for the appropriate horizon on each step.

Similar to our definition of V^h for evaluating a policy, we define the Q^h function recursively according to the horizon. The only difference is that, on each step with horizon h, rather than selecting an action specified by a given policy, we select the value of a that will maximize the expected Q^h value of the next state.

$$\begin{aligned}
Q^0(s,a) &= 0 \\
Q^1(s,a) &= R(s,a) + 0 \\
Q^2(s,a) &= R(s,a) + \sum_{s'} T(s,a,s') \max_{a'} R(s',a') \\
&\vdots \\
Q^h(s,a) &= R(s,a) + \sum_{s'} T(s,a,s') \max_{a'} Q^{h-1}(s',a')
\end{aligned}$$

We can solve for the values of Q^h with a simple recursive algorithm called *finite-horizon value iteration* which just computes Q^h starting from horizon 0 and working backward to the desired horizon H. Given Q^h, an optimal finite-horizon policy π^*_h is easy to find:

$$\pi^*_h(s) = \arg\max_a Q^h(s,a) \quad .$$

There may be multiple possible optimal policies.

Dynamic programming (somewhat counter-intuitively, dynamic programming is neither really "dynamic" nor a type of "programming" as we typically understand it) is a technique for designing efficient algorithms. Most methods for solving MDPs or computing value functions rely on dynamic programming to be efficient.
The *principle of dynamic programming* is to compute and store the solutions to simple sub-problems that can be re-used later in the computation. It is a very important tool in our algorithmic toolbox.
Let's consider what would happen if we tried to compute $Q^4(s,a)$ for all (s,a) by directly using the definition:

- To compute $Q^4(s_i,a_j)$ for any one (s_i,a_j), we would need to compute $Q^3(s,a)$ for all (s,a) pairs.
- To compute $Q^3(s_i,a_j)$ for any one (s_i,a_j), we'd need to compute $Q^2(s,a)$ for all (s,a) pairs.
- To compute $Q^2(s_i,a_j)$ for any one (s_i,a_j), we'd need to compute $Q^1(s,a)$ for all (s,a) pairs.
- Luckily, those are just our $R(s,a)$ values.

So, if we have n states and m actions, this is $O((mn)^3)$ work—that seems like way too much, especially as the horizon increases! But observe that we really only have mnh values that need to be computed, $Q^h(s,a)$ for all h,s,a. If we start with $h=1$, compute and store those values, then using and reusing the $Q^{h-1}(s,a)$ values to compute the $Q^h(s,a)$ values, we can do all this computation in time $O(mnh)$, which is much better!

2.2 Infinite-horizon solutions

It is more typical to work in a regime where the actual finite horizon is not known. This is called the *infinite horizon* version of the problem, when you don't know when the game will be over! However, if we tried to simply take our definition of Q^h above and set $h = \infty$, we would be in trouble, because it could well be that the Q^∞ values for all actions would be infinite, and there would be no way to select one over the other.

There are two standard ways to deal with this problem. One is to take a kind of *average* over all time steps, but this can be a little bit tricky to think about. We'll take a different approach, which is to consider the *discounted* infinite horizon. We select a discount factor $0 < \gamma < 1$. Instead of trying to find a policy that maximizes expected finite-horizon undiscounted value,

$$\mathbb{E}\left[\sum_{t=0}^{h} R_t \mid \pi, s_0\right] \quad ,$$

we will try to find one that maximizes the expected *infinite horizon discounted value*, which is

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \mid \pi, S_0\right] = \mathbb{E}\left[R_0 + \gamma R_1 + \gamma^2 R_2 + \ldots \mid \pi, s_0\right] \quad .$$

Note that the t indices here are not the number of steps to go, but actually the number of steps forward from the starting state (there is no sensible notion of "steps to go" in the infinite horizon case).

There are two good intuitive motivations for discounting. One is related to economic theory and the present value of money: you'd generally rather have some money today than that same amount of money next week (because you could use it now or invest it). The other is to think of the whole process terminating, with probability $1 - \gamma$ on each step of the interaction. This value is the expected amount of reward the agent would gain under this terminating model.

2.2.1 Evaluating a policy

We will start, again, by evaluating a policy, but now in terms of the expected discounted infinite-horizon value that the agent will get in the MDP if it executes that policy. We define the infinite-horizon value of a state s under policy π as

$$V_\pi(s) = \mathbb{E}[R_0 + \gamma R_1 + \gamma^2 R_2 + \cdots \mid \pi, S_0 = s] = \mathbb{E}[R_0 + \gamma(R_1 + \gamma(R_2 + \gamma \ldots))) \mid \pi, S_0 = s] \quad .$$

Because the expectation of a linear combination of random variables is the linear combination of the expectations, we have

$$\begin{aligned} V_\pi(s) &= \mathbb{E}[R_0 \mid \pi, S_0 = s] + \gamma \mathbb{E}[R_1 + \gamma(R_2 + \gamma \ldots))) \mid \pi, S_0 = s] \\ &= R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') V_\pi(s') \end{aligned}$$

You could write down one of these equations for each of the $n = |\mathcal{S}|$ states. There are n unknowns $V_\pi(s)$. These are linear equations, and so it's easy to solve them using Gaussian elimination to find the value of each state under this policy.

This is *so* cool! In a discounted model, if you find that you survived this round and landed in some state s', then you have the same expected future lifetime as you did before. So the value function that is relevant in that state is exactly the same one as in state s.

2.2.2 Finding an optimal policy

The best way of behaving in an infinite-horizon discounted MDP is not time-dependent: at every step, your expected future lifetime, given that you have survived until now, is $1/(1-\gamma)$.

Study Question: Verify this fact: if, on every day you wake up, there is a probability of $1-\gamma$ that today will be your last day, then your expected lifetime is $1/(1-\gamma)$ days.

An important theorem about MDPs is: in the infinite-horizon case, there exists a stationary optimal policy π^* (there may be more than one) such that for all $s \in \mathcal{S}$ and all other policies π, we have

$$V_{\pi^*}(s) \geqslant V_\pi(s) \quad .$$

Stationary means that it doesn't change over time; in contrast, the optimal policy in a finite-horizon MDP is *non-stationary.*

There are many methods for finding an optimal policy for an MDP. We have already seen the finite-horizon value iteration case. Here we will study a very popular and useful method for the infinite-horizon case, *infinite-horizon value iteration*. It is also important to us, because it is the basis of many *reinforcement-learning* methods.

Define $Q^*(s,a)$ to be the expected infinite-horizon discounted value of being in state s, executing action a, and executing an optimal policy π^* thereafter. Using similar reasoning to the recursive definition of V_π, we can express this value recursively as

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q^*(s',a') \quad .$$

This is also a set of equations, one for each (s,a) pair. This time, though, they are not linear, and so they are not easy to solve. But there is a theorem that says they have a unique solution!

If we knew the optimal action-value function, then we could derive an optimal policy π^* as

$$\pi^*(s) = \arg\max_a Q^*(s,a) \quad .$$

Study Question: The optimal value function is unique, but the optimal policy is not. Think of a situation in which there is more than one optimal policy.

We can iteratively solve for the Q^* values with the infinite-horizon value iteration algorithm, shown below:

INFINITE-HORIZON-VALUE-ITERATION$(\mathcal{S}, \mathcal{A}, T, R, \gamma, \epsilon)$

for $s \in \mathcal{S}, a \in \mathcal{A}$:
    $Q_{\text{old}}(s,a) = 0$
while True:
    for $s \in \mathcal{S}, a \in \mathcal{A}$:
        $Q_{\text{new}}(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q_{\text{old}}(s',a')$
    if $\max_{s,a} |Q_{\text{old}}(s,a) - Q_{\text{new}}(s,a)| < \epsilon$:
        return Q_{new}
    $Q_{\text{old}} = Q_{\text{new}}$

2.2.3 Theory

There are a lot of nice theoretical results about infinite-horizon value iteration. For some given (not necessarily optimal) Q function, define $\pi_Q(s) = \arg\max_a Q(s,a)$.

- After executing infinite-horizon value iteration with parameter ϵ,

$$\|V_{\pi_{Q_{\text{new}}}} - V_{\pi^*}\|_{\max} < \epsilon \quad .$$

This is new notation! Given two functions f and f', we write $\|f - f'\|_{\max}$ to mean $\max_x |f(x) - f'(x)|$. It measures the maximum absolute disagreement between the two functions at any input x.

- There is a value of ϵ such that

$$\|Q_{\text{old}} - Q_{\text{new}}\|_{\max} < \epsilon \implies \pi_{Q_{\text{new}}} = \pi^*$$

- As the algorithm executes, $\|V_{\pi_{Q_{\text{new}}}} - V_{\pi^*}\|_{\max}$ decreases monotonically on each iteration.
- The algorithm can be executed asynchronously, in parallel: as long as all (s, a) pairs are updated infinitely often in an infinite run, it still converges to optimal value.

This is very important for reinforcement learning.