Reinforcement Learning **Spring Semester, 2018/9**

Lecture 8: May 1, 2019

Lecturer: Yishay Mansour *Scribe: ym*

This lecture starts looking at the case where the MDP models are large. In the current lecture we will look at approximating the value function. In the next lecture we will consider learning directly a policy and optimizing it.

When we talk about a large MDP, it can be in one of a few different reasons. The most common is having a large state space. For example, Backgammon has 10^{20} states, Go has 10^{170} and robot control has continuous state space. Another dimension is the action space, which can be even continuous in many applications (say, robots). Finally, we might have a complex dynamics which are hard to describe succinctly. The main challenge is that we need our algorithm to scale up to this enormous spaces.

Today we will consider function approximation for large MDPs, mainly large state spaces. Previously, we had a look-up table for the value function $V^\pi(s)$ or $Q^\pi(s,a)$, which we updated every time we encountered the state. In a large state space this will be infeasible. Not only that we do not have the memory, but even more importantly, we are unlikely to observe states re-occurring. We will need to make (implicit) assumptions about the MDP, which can be viewed similarly to our assumption in learning a classifier.

Specifically, we will have the value functions parameterized by *weights* w, namely, $\hat{V}(s;w) \approx V^\pi(s)$ and $\hat{Q}(s,a;w) \approx Q^\pi(s,a)$. We would like to guarantee generalization from the observed states and trajectories to unobserved states and trajectories. In the process we will update the weights w, using some learning procedure, most notably using MC or TD learning.

When considering a value function there are a few interpretations what exactly we mean. It can either be: (1) mapping from a state s to its expected return, i.e., $\hat{V}^\pi(s;w)$. (2) mapping from state-action pairs (s,a) to their expected return, i.e., $\hat{Q}^\pi(s,a;w)$. (3) mapping from states s to expected return of each action, i.e., $\langle \hat{Q}^\pi(s,a_i;w) : a_i \in A \rangle$. All the interpretations are valid, and our discussion will not distinguish between them. (Actually, for $\langle \hat{Q}^\pi(s,a_i;w) : a_i \in A \rangle$ we implicitly assume that the number of actions is small.)

We now need to discuss how will we build the approximating function. For this we can turn to the rich literature in Machine Learning and consider the popular hypothesis classes. For example: (1) Linear functions, (2) Neural networks, (3) Decision trees, (4) Nearest neighbors, (5) Fourier or wavelet basis, etc. We will concentrate on linear functions and neural networks, mainly since gradient based methods apply to them very naturally.

8.1 Basic Approximation

Before we start the discussion on the learning methods, we will do a small detour. We will discuss the effect of having an error in the value function we learn, and its effect on the outcome. Assume we have a value function V such that $\|V - V^*\|_\infty \leq \epsilon$. Let π be the greedy policy with respect to V, namely,

$$\pi(s) = \arg\max_a [r(s,a) + \gamma E_{s' \sim p(\cdot|s,a)}[V(s')]$$

We will consider in this lecture (mainly) the discounted return with a discount parameter $\gamma \in (0,1)$.

Theorem 8.1. *Let V such that $\|V - V^*\|_\infty \leq \epsilon$ and π be the greedy policy with respect to V. Then,*

$$\|V^\pi - V^*\|_\infty \leq \frac{2\gamma\epsilon}{1-\gamma}$$

Proof. Consider two operators, the first H^π converges to V^π and

$$(H^\pi v)(s) = r(s, \pi(s)) + \gamma E_{s' \sim p(\cdot|s,\pi(s))}[v(s')]$$

The second H^* converges to V^* and

$$(H^* v)(s) = \max_a [r(s,a) + \gamma E_{s' \sim p(\cdot|s,a)}[v(s')]]$$

Since π is greedy with respect to V we have $H^\pi V = H^* V$ (but this does not hold for other value functions V').

Then,

$$\begin{aligned}
\|V^\pi - V^*\|_\infty &= \|H^\pi V^\pi - V^*\|_\infty \\
&\leq \|H^\pi V^\pi - H^\pi V\|_\infty + \|H^\pi V - V^*\|_\infty \\
&\leq \gamma \|V^\pi - V\|_\infty + \|H^* V - H^* V^*\|_\infty \\
&\leq \gamma \|V^\pi - V\|_\infty + \gamma \|V - V^*\|_\infty \\
&\leq \gamma(\|V^\pi - V^*\|_\infty + \|V^* - V\|_\infty) + \gamma \|V - V^*\|_\infty
\end{aligned}$$

where in the second inequality we used the fact that since since π is greedy with respect to V then $H^\pi V = H^* V$.

Reorganizing the inequality and recalling that $\|V^* - V\|_\infty \leq \epsilon$, we have

$$(1-\gamma)\|V^\pi - V^*\|_\infty \leq 2\epsilon\gamma$$

□

The above theorem says that if we have a small error in L_∞ norm, the effect is bounded. However, in most cases we will not be able to guarantee such an approximation! This is since we are unable even to verify the L_∞ norm of two value function efficiently, since we need to consider *all* states.

8.2 From RL to ML

We would like to reduce our reinforcement learning problem to a supervised learning problem. This will enable us to use any of the many techniques of machine learning to address the problem. Let us consider the basic ingredients of supervised learning. The most important ingredient is having a labeled sample, which is sampled i.i.d.

Let us start by considering an idealized setting. Fix a policy π that we like to learn its value function. We like to generate a sample

$$\{(s_1, V^\pi(s_1)), \ldots, (s_m, V^\pi(s_m))\}$$

We first need to discuss how to sample the states s_i in an i.i.d. way. We can generate trajectory, but we need to be careful, since adjacent states are definitely dependent! One solution is to space the sampling from the trajectory using the mixing time of π. This will give us samples s_i which are from the stationary distribution of π and are almost independent. For the action, we clearly will use π, no problem there.

The hardest, and most confusing, ingredient is the labels $V^\pi(s_i)$. In machine leaning we actually assume that someone gives us the labels to build a classifier (or alternatively, in unsupervised learning, we have no clear ground truth). If someone is able to give us $V^\pi(s)$, it seems like assuming the problem away, since this is what we would like to learn.

Actually we need to define three things: (1) states and actions, which we will do using π, (2) a loss function, which will give the tradeoffs between different approximation errors, and (3) labels, which we are not give explicitly.

We will define a differential loss function $J(w)$. This will enable us to compute the gradient of the loss function $\nabla_w J(w) = \langle \frac{\partial}{\partial w_1} J(w), \ldots, \frac{\partial}{\partial w_d} J(w) \rangle$, and update

the weights in the negative direction of the gradient, namely, $\Delta_w = -0.5\alpha\nabla_w J(w)$, where α is the step size of the learning rate. In supervised learning we are guaranteed that the process will converge to a local minimum (or a saddle point).

Let us start by defining the loss function,

$$J(w) = \sum_s \mu(s)(V^\pi(s) - \hat{V}^\pi(s;w))^2$$

where $\mu(s)$ is the steady state distribution of π. So we can compute the gradient and update the weights. The Stochastic Gradient Descent (SGD) is simply sampling a single state s and using it, i.e.,

$$\Delta w_t = \alpha(V^\pi(s) - \hat{V}^\pi(s;w))\nabla\hat{V}^\pi(s;w)$$

We now need to build the sample, namely, how we replace $V^\pi(s)$. The basic idea is to find an unbiased estimator U_t such that $E[U_t|s_t] = V^\pi(s_t)$. We now will present a few different approaches to derive such estimators.

The most simple approach is to use Monte-Carlo sampling. Recall that in Monte-Carlo we have $R_T(s) = \sum_{t=1}^T r_t$, starting at the first visit of s. Clearly, we have $E[R_T(s)] = V^\pi(s)$, since samples are independent, so we can set $U_T(s) = R_T(s)$. The update becomes,

$$\Delta w_t = \alpha(R_t(s) - \hat{V}^\pi(s;w_t))\nabla_w\hat{V}^\pi(s;w)$$

We can try to use the same idea for $TD(0)$. The estimate of $TD(0)$ for s_t is $R_t(s_t) = r_t + \gamma\hat{V}^\pi(s_{t+1};w)$. The first problem is that this is a biased estimator since

$$V^\pi(s_t) = E[r_t + \gamma V^\pi(s_{t+1})] \neq E[r_t + \gamma\hat{V}(s_{t+1};w)]$$

We have an additional problem with the gradient, since we have w influencing also the target $R_t(s_t)$, something we do not have in MC (or generally in ML). For this reason $\nabla_w\hat{V}(s_t;w)$ is called a semi-gradient. The update becomes,

$$\Delta w(s) = \alpha[r_t + \gamma\hat{V}(s_{t+1};w) - \hat{V}(s_t)]\nabla_w\hat{V}(s_t;w)$$

Finally we get to $TD(\lambda)$. Similar to $TD(0)$ we can now define the forward update to be,

$$\Delta w = \alpha[R_t^\lambda - \hat{V}(s_t;w)]\nabla_w\hat{V}(s_t;w)$$

and the backward update,

$$\begin{aligned} e_t &= \gamma\lambda e_{t-1} + \nabla_w\hat{V}(s_t;w) \\ \Delta_t &= r_t + \gamma\hat{V}(s_{t+1};w) - \hat{V}(s_t;w) \\ \Delta w &= \alpha\Delta_t e_t \end{aligned}$$

8.3 Linear Functions

We first start with the state encoding. In general we assume that each state s is encoded by a vector $x(s) \in \mathbb{R}^d$. For notation, let $x(s) = (x_1(s), \ldots, x_d(s))$. This will be useful for any hypothesis, and specifically, linear function and neural networks.

A linear function is characterized by a vector of weights $w \in \mathbb{R}^d$. The value of the function is

$$\hat{V}(s; w) = w^\top x(s)$$

This implies that the SGD updates become

$$w_{t+1} = w_t + \alpha[U_t - \hat{V}(s_t; w_t)]x(s_t)$$

The main benefit of the linear function is when $d \ll |S|$. When $d = |S|$ we can simply encode a look-up table using w. Simply set $x_i(s) = I(s = s_i)$. Then, $\hat{V}(s; w) = \sum_{i=1}^d w_i I(s = s_i)$. This implies that $\hat{V}(s_i; w) = w_i$.

For a linear function the gradient is simply $x(s)$, i.e., $\nabla_w w^\top x(s) = x(s)$. The Monte-Carlo update will become

$$\Delta w = \alpha[R_t - \hat{V}(s_t; w_t)]x(s_t)$$

For $TD(0)$ we have

$$\Delta w = \alpha[r_t + \gamma \hat{V}(s_{t+1}; w_t) - \hat{V}(s_t; w_t)]x(s_t)$$

For $TD(\lambda)$, for the forward view we have

$$\Delta w = \alpha[R_t^\lambda - \hat{V}(s_t; w_t)]x(s_t)$$

For the backward view we have,

$$\begin{aligned} e_t &= \gamma\lambda e_{t-1} + x(s_t) \\ \Delta_t &= r_t + \gamma \hat{V}(s_{t+1}; w_t) - \hat{V}(s_t; w_t) \\ \Delta w &= \alpha \Delta_t e_t \end{aligned}$$

We would like to discuss the convergence of the different algorithms. Recall that the loss function is strongly convex, i.e.,

$$J(w) = \sum_s \mu(s)(V^\pi(s) - \hat{V}^\pi(s; w))^2$$

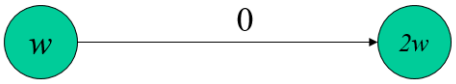

Figure 8.1: Two state snippet of an MDP

Therefore there is a unique w_{min} which minimizes $J(w)$. Monte-Carlo is simply SGD, and therefore the convergence and its rate follow from known results in optimization.

The convergence of TD is more problematic, since it uses a semi-gradient and not the true gradient. The good news is that for linear functions, when using an on-policy, there is a proof of convergence. However, the convergence is not to w_{min} but rather to w_{TD}. The difference is due to the fact that we are minimizing the expression

$$\sum_s \mu(s)(r(s,\pi(s)) + \gamma E_{s'}[\hat{V}(s';w)] - \hat{V}^{\pi}(s;w))^2$$

The difference in the losses can be bounded as follows

$$J(w_{TD}) \leq \frac{1}{1-\gamma} J(w_{min})$$

8.4 Off-policy

We would like to see what is the effect that the samples come following a different policy, namely, an off-policy setting. There is no issue for Monte-Carlo, and the same logic would still be valid. For TD, we did not have any problem in the look-up model. We would like to see what can go wrong when we have a function approximation setting.

Consider the following part of an MDP (see Figure 8.1) consists of two modes, with a transition from the first to the second, with reward 0. The main issue is that the linear approximation gives the first node a weight w and the second $2w$. Assume we start with some value $w_0 > 0$. Each time we have an update for the two states we have

$$w_{t+1} = w_t + \alpha[0 + \gamma(2w_t) - w_t] = [1 + \alpha(2\gamma - 1)]w_t$$

For $\gamma > 0.5$ we have $\alpha(2\gamma - 1) > 0$, and w_t diverges.

We are implicitly assuming that the setting is off-policy, since in an on-policy, we would continue from the second state, and eventually lower the weight.

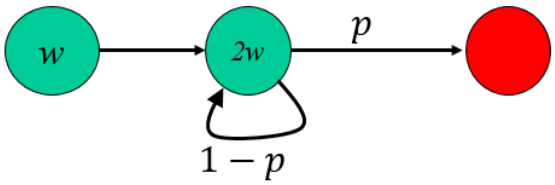

Figure 8.2: The three state MDP. All rewards are zero.

To have a "complete" example consider the three state MDP in Figure 8.2. All the rewards are zero, and the main difference is that we have a new terminating state, that we reach with probability p.

Again, assume that we start with some $w_0 > 0$ We have three types of updates, one per possible transition. When we transition from the initial state to the second state we have

$$\Delta w = \alpha[0 + \gamma(2w_t) - w_t] = \alpha(2\gamma - 1)w_t$$

The transition from the second state back to itself has an update,

$$\Delta w = \alpha[0 + \gamma(2w_t) - (2w_t)] = -2\alpha(1 - \gamma)w_t$$

The transition to the terminal state we have

$$\Delta w = \alpha[0 + \gamma 0 - (2w_t)] = -2\alpha w_t$$

When we use on-policy, we have all transitions. Assume that the second transition happens $n \geq 0$ times. Then we have

$$\frac{w_{t+1}}{w_t} = (1 + \alpha(2\gamma - 1))(1 - 2\alpha(1 - \gamma))^n(1 - 2\alpha) < 1 - \alpha$$

This implies that w_t converges to zero, as desired.

Now consider an off-policy that truncates the episodes after n transitions of the second state, where $n \ll 1/p$. This implies that in most updates we have

$$\frac{w_{t+1}}{w_t} = (1 + \alpha(2\gamma - 1))(1 - 2\alpha(1 - \gamma))^n > 1$$

and therefore, for the right setting of n we have that the weight w_t diverges.

We might hope that the divergence is due to the online nature of the TD updates. We can consider an algorithm that in each iteration minimizes the square error. Namely,

$$w_{t+1} = \arg\min_w \sum_s [\hat{V}(s_t; w) - E^\pi[r_t + \gamma \hat{V}(s_{t+1}; w_t)]]^2$$

For the example of Figure 8.2 we have that

$$w_{t+1} = \frac{6\gamma - 4p}{5} w_t$$

So for $6\gamma - 4p > 5$ we have divergence. (Recall that $\gamma \approx 1 - \epsilon$ and $p \approx \epsilon$, so this would hold in most settings we are most interested in.)

What we have seen that there is a *deadly triad*:

1. Function Approximation
2. Bootstrapping methods, such as TD.
3. Off-policy training.

When the three conditions hold, we have counter examples for the convergence.

Here is a summary of the known convergence and divergence results in the literature:

	algorithm	look-up table	linear function	non-linear
on-policy	MC	+	+	+
on-policy	$TD(0)$	+	+	-
on-policy	$TD(\lambda)$	+	+	-
off-policy	MC	+	+	+
off-policy	$TD(0)$	+	-	-
off-policy	$TD(\lambda)$	+	-	-

8.5 Applications: games

8.5.1 Backgammon

Tesauro developed the TD-gammon in the late 1980's and 1990's. The system uses a neural network approximation, using a $TD(\lambda)$ updates. The initial system was able to win against other computer programs, which where hand-designed, and later matched the performance of the best human. For the training the system used self-play, namely playing against itself.

The neural network has a single hidden layer (with 40/80/160 nodes, in the three different generations of the project). There are 198 inputs. The hidden layer has sigmoidal nodes with $\sigma(z) = \frac{1}{1+e^{-z}}$. The output y can be viewed as the probability of winning from a given position.

The 198 inputs are composed as follows. For each of the 24 slots and for each of the two colors, we have four Boolean inputs. The first indicates if there is a single chip, the seocnd indicates two or more chips, the third indicates exactly three chips and the fourth indicates four or more chips. We have two Boolean variables that indicate who turn it is (one for white and one for black). Finally, we have two aggregate inputs, per color: (1) number of chips divided by 2 and (2) number of chips removed divided by 15.

The TD-gammon uses the standard updates.

$$\Delta w_t = \alpha(y_t - y_{t-1}) \sum_{k=1}^{t} \nabla \lambda^{t-k} \nabla_w y_k$$

When the game ends we replace y_t by z the outcome of the game.

The evaluation of the position is done using a max-min tree, whose depth is 1/2/3. The max-min tree has max layer where the value of a node is the maximum of their children, and min layer, where the value of a node is the minimum of their children. Given the tree (of size at most 20/400/8000 in different generations) we evaluate the leaves using the neural network, and propagate the values in the min-max tree to get the predicted value of a position.

The network is initialized with small random weights. Note that due to the game nature, even with random policies the game terminates. The training is done using self-play, where the number of games was $300K/800K/1500K$ in different generations.

One very intriguing aspect of the TD-gammon is that it *improved the human performance*. For example, when the first role is $4-1$, expects did the 1 move in location 6, while the TD-gammon moved location 24. Following the TD-gammon, experts changed their view and started playing those positions in the same way that TD-gammon did.

8.5.2 Attari games

The system was developed by DeepMind researchers during $2013-2015$. The system learns to play from the raw video frames, a variety of 49 games. The learning uses a neural network and employs Q-learning (and a Deep-Q-Network, DQN).

The neural network works as follows. There is a preprocessing stage that maps frames of size 210×160 with 128-colors, to 84×84 and 4-colors. This mapping is fixed and shared by all games. The neural network has three convolutional layers followed by one completely connected layer. The output layer has multiple outputs. The nodes in the network use RelU gates.

The DQN innovations are: (1) Experience reply, reusing the same example many times, (2) fixed Q-target, when setting the target update, and (3) clipping the error. To better understand the innovations, we need to consider the flow of the training.

In the training, we first select an action a_t, using ϵ-greedy policy. We observe the reward and the next state (s_t, a_t, r_t, s_{t+1}), and store them in the reply memory. We sample a mini-batch from the reply memory. For each sample we use the fixed-Q-target (more latter). We use the SGD to minimize the MSE.

For the fixed-Q-target, we do the update as follows. We have two sets of weights w_t, the current weights, and w_t^- as the fixed weights. We update w_t to w_t^- every (large) number of plays, setting $w_t = w_t^-$. The update at time t becomes:

$$\Delta w_t = \alpha \Gamma_t \nabla_{w_t} Q(s_t, a_t; w_t)$$

where

$$\Gamma = r_t + \gamma \max_a Q(s_{t+1}, a; w_t^-) - Q(s_t, a_t; w_t)$$

Note that w_t^- is used only to estimate the future return from s_{t+1}. The clipping enforces $\Gamma_t \in [-1, +1]$.

The system is able to match human performance in 29 out of 49 Attari games.

8.6 Bibliography Remarks

The work of Gerald Tesauro on TD-gammon is described in [3, 4].

The DeepMind system for playing Attari games is described in [1].

Part of the outline borrows from David Silver class notes and the the book of Sutton and Barto [2].

Bibliography

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

[3] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995.

[4] Gerald Tesauro. Programming backgammon using self-teaching neural nets. *Artif. Intell.*, 134(1-2):181–199, 2002.