1 On-policy Prediction with Approximation

In this section we consider the applications of function approximation techniques in reinforcement learning, to learn mappings from states to values. Typically we will consider parametric functional forms, in which case we can achieve a reduction in dimensionality of the problem (number of parameters smaller than state space). In this way, the function generalises between states, as the update of one state impacts the value of another.

Function approximation techniques are applicable to partially observable problems, in which the full state space is not available to the agent. A function approximation scheme which is ignores certain aspects of the space behaves just as if those aspects are unobservable.

1.1 Value-function Approximation

Many techniques from supervised learning are applicable to learning value functions from experience, but not all are equipped to deal with the non-stationarity that often occurs in RL. In RL it is also important to be able to learn online.

1.2 The Prediction Objective ($\overline{\mathrm{VE}}$)

Define a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$ that represents how much we care about each state s. Given an estimator $\hat{v}(s, \mathbf{w})$ of $v_\pi(s)$, parameterised by $\mathbf{w}$, we define our objective function as the *Mean Squared Value Error*

$$\overline{\mathrm{VE}} \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 . \tag{1}$$

Often we choose $\mu(s)$ to be the fraction of time spent in s. Under on-policy training this is referred to as the *on-policy distribution*. In continuing tasks, this is the stationary distribution under π.

At this stage it is not clear that we have chosen the correct (or even a good) objective function, since the ultimate goal is a good policy for the task. For now, will continue with $\overline{\mathrm{VE}}$ nonetheless.

The on-policy distribution in episodic tasks

In an episodic task the on-policy distribution depends on how the initial states of the episode are chosen. Let $h(s)$ be the probability that an episode begins in state s and $\eta(s)$ be the expected time spent in s per episode. Note that you can either start in s or transition there from $\bar{s}$, so

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s|\bar{s}, a) \quad \forall s \in \mathcal{S}.$$

One can solve this system for η, then take the on-policy distribution as

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')} \quad \forall s \in \mathcal{S}.$$

This is the natural choice without discounting. With discounting we consider it a form of termination and include a factor of γ in the second term of the recurrence relation above.

1.3 Stochastic-gradient and Semi-gradient Methods

(Stochastic) Gradient Descent

We assume that states appear in examples with the same distribution $\mu(s)$, in which case a good strategy is to minimise our loss function on observed examples. *Stochastic gradient-descent* moves the weights in the direction of decreasing $\overline{\text{VE}}$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}\left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w})\right]^2 \tag{2}$$

$$= \mathbf{w}_t + \alpha\left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w})\right]\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}). \tag{3}$$

Of course, we might not know the true value function exactly, we will likely only have access to some approximation of it U_t, possibly corrupted by noise or got from bootstrapping with our latest estimate. In these cases we cannot perform the above computation, but we can still make the general SGD update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\left[U_t - \hat{v}(S_t, \mathbf{w})\right]\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}) \tag{4}$$

If U_t is an unbiased estimate of the state value for each t, then the sequence $\mathbf{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions for decreasing α.

The Monte Carlo target $U_t = G_t$ is an unbiased estimator, so locally optimal convergence is guaranteed in this case. Algorithm is given below.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

Semi-Gradient Descent

We don't get the same convergence guarantees if we use bootstrapping estimates of the value function in our update target, for instance if we had used the TD(0) update $U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w})$. This is because the target now depends on the parameters $\mathbf{w}$, so the gradient is not exactly the gradient of our loss function – it only takes into account the change on our estimate with respect to $\mathbf{w}$. For this reason we call updates such as this *semi-gradient methods*.

Semi-gradient methods are often preferable to pure gradient methods since they can offer much faster learning, in spite of not giving the same convergence guarantees. A prototypical choice is the TD(0) update, an algorithm for which is given in the box below.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

State Aggregation

State aggregation is a simple form of generalising in which we group together states and fix them to have the same estimated value.

1.4 Linear Methods

As always, linear methods of function approximation are an important special case

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s) \tag{5}$$

where $\mathbf{x}(s)$ are feature vectors, vectors of functions (features) $x_i : \mathcal{S} \to \mathbb{R}$. The SGD update for the linear model is

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\,[U_t - \hat{v}(S_t, \mathbf{w})]\,\mathbf{x}(s). \tag{6}$$

Naturally, the linear case is the most studied and the majority of convergence results for learning systems are for this case (or simpler). In particular, there is the benefit that there is a unique global optimum for our loss function (in the non-degenerate case).

Convergence of Linear TD(0)

The semi-gradient TD(0) algorithm is known to converge under linear function approximation. The point converged to is not the global optimum, but a point near the local optimum. We consider this case in more detail. First write $\mathbf{x}_t = \mathbf{x}(S_t)$ then rearrange the update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\left(R_{t+1} + \gamma\mathbf{w}_t^\top\mathbf{x}_{t+1} - \mathbf{w}_t^\top\mathbf{x}_t\right)\mathbf{x}_t \tag{7}$$

$$= \mathbf{w}_t + \alpha\left(R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\mathbf{w}_t\right). \tag{8}$$

Now note that we can write

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathrm{A}\mathbf{w}_t)$$

where $\mathbf{b} = \mathbb{E}[R_{t+1}\mathbf{x}_t]$ and $\mathrm{A} = \mathbb{E}\left[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\right]$. It's clear now that in a steady state we must have (can be shown that A positive definite and so invertible)

$$\mathbf{w}_{\mathsf{TD}} = \mathrm{A}^{-1}\mathbf{b}.$$

We call this point the *TD fixed point*, linear semi-gradient TD(0) converges to this point. (In the notes there is a box with some details.)

At the TD fixed point (in the continuing case) it has been proven that $\overline{\text{VE}}$ is within a bounded expansion of the lowest possible error

$$\overline{\text{VE}}(\mathbf{w}_{\text{TD}}) \leq \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}). \tag{9}$$

It is often the case that γ is close to 1, so this region can be quite large. The TD method jas substantial loss in asymptotic performance. Regardless of this, it still has much lower variance than MC methods and can thus be faster. The desired update method will depend on the task at hand.

Other Linear Updates

Linear semi-gradient DP $U_t = \sum_a \pi(a|S_t) \sum_{s',r} p(s', r|S_t, a)[r + \gamma \hat{v}(s', \mathbf{w}_t)]$ with updates according to the on-policy distribution also converged to the TD fixed point. There are convergence results for other step methods we have considered too. Critical to all of these is that updates are taken according to the on-policy distribution. For other update distributions, bootstrapping methods can diverge to infinity. n-step semi-gradient TD is given in the box below.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, a positive integer n
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{v}(S_\tau, \mathbf{w})\right] \nabla \hat{v}(S_\tau, \mathbf{w})$
  Until $\tau = T - 1$

1.5 Feature Construction for Linear Methods

Discussed in this section

- Polynomial Basis
- Fourier Basis

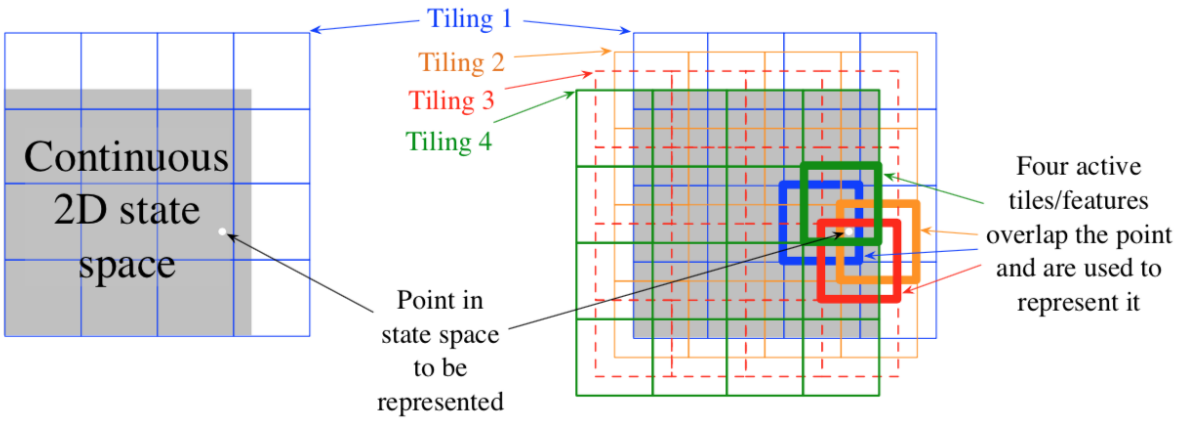
- One could use other orthogonal function bases but they are yet to see application in RL.
- Radial Basis Functions. (Offer little advantage over coarse coding with circles, but greatly increases computational complexity)

1.5.3 Coarse Coding

One way of promoting generalisation between states is to cover the state-space in overlapping regions, with each region representing a feature. If the state is being considered, then all regions that contain this state will be activated. The amount of overlap of the receptive fields (the states which can activate a feature) dictates the breadth of generalisation.

1.5.4 Tile Coding

A *tiling* of a continuous state space is form of coarse-coding that creates a partition of the state space (all of the state space is covered but elements of the tiling do not overlap). We call a sub-region of a tiling a *tile*. One might introduce multiple overlapping tilings to incorporate generalisation.



An advantage of tilings is that, because each tiling forms a partition, the total number of features active at any one time is just the total number of tilings used. So $\alpha = \frac{1}{kn}$, where n is the number of tilings, results in k-trial learning. That is, on average the learning asymptotes after k presentations of each state (assuming all updates use the same, constant target).

Tile coding is computationally efficient and may be the most practical feature representation for modern sequential digital computers.

A useful trick for reducing memory requirements is *hashing*. One can essentially hash the state space, then tile the hashed values. This means that each tile in the hashed space will represent (multiple) pseudo-randomly distributed tiles in the original space. Since only a small proportion of the state space needs to have high resolution value estimates, this can be a good way to reduce memory with little loss in performance.

1.6 Selecting Step-Size Parameters Manually

In the tabular case, taking $\alpha = \frac{1}{\tau}$ will mean that the estimate for a state will approach the mean of its targets, with the most recent targets having the greatest effect, in about τ experiences.

With function approximation, there is not a clear notion of the number of visits to a state because of continuous degrees of generalisation. However, a sensible consideration for learning from τ

presentations is

$$\alpha = \frac{1}{\tau \mathbb{E}[\mathbf{x}^\top \mathbf{x}]}. \tag{10}$$

1.7 Nonlinear Function Approximation: Artificial Neural Networks

These of course see a lot of application in RL, especially with deep learning. There are some good review articles on the web.

1.8 Least-Squares TD

We saw earlier that TD(0) with linear function approximation converges to the TD fixed point

$$\mathbf{w}_{\text{TD}} = \mathrm{A}^{-1}\mathbf{b},$$

where $\mathbf{b} = \mathbb{E}[R_{t+1}\mathbf{x}_t]$ and $\mathrm{A} = \mathbb{E}\left[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top\right]$. Previously we computed the solution iteratively, but this is a waste of data! We could compute the MLE of A and $\mathbf{b}$ and then use those. This is the *Least-Squares TD Algorithm*, it uses the estimators

$$\hat{\mathrm{A}}_t = \sum_{k=0}^{t-1} x_k(x_k - \gamma x_{k+1})^\top + \epsilon \mathrm{I} \quad \text{and} \quad \hat{\mathbf{b}}_t = \sum_{k=0}^{t-1} R_{t+1} x_k \tag{11}$$

where we introduce $\epsilon > 0$ to ensure that the sequence of $\hat{\mathrm{A}}_t$ are each invertible. (These are estimates of $t\mathrm{A}$ and $t\mathbf{b}$ but the t cancel out.)

This is the most data efficient form of TD(0), but it is also more computationally intensive. Implementing incrementally and with tricks to do the matrix inverse (because of the particular form of A as sum of outer products), one can do this in $O(d^2)$ computations, where d is the number of parameters/features (note that this is independent of t). (For comparison, the semi-gradient TD(0) method needs $O(d)$ computations.) The formula for A is

$$\hat{\mathrm{A}}_t = \left(\hat{\mathrm{A}}_{t-1} + x_t(x_t - \gamma x_{t+1})^\top\right)^{-1} \tag{12}$$

$$= \hat{\mathrm{A}}_{t-1}^{-1} - \frac{\hat{\mathrm{A}}_{t-1}^{-1} x_t (x_t - \gamma x_{t+1})^\top \hat{\mathrm{A}}_{t-1}^{-1}}{1 + x_t(x_t - \gamma x_{t+1})^\top \hat{\mathrm{A}}_{t-1}^{-1} x_t} \tag{13}$$

To store $\hat{A}_{t-1}$ LSTD also needs $O(d^2)$ memory. LSTD has no step-size parameter, which means that it never forgets – this can be a blessing or a curse depending on the application. The choice between LSTD and semi-gradient TD will depend on the application, for instance the computation available and the importance of learning quickly. Pseudocode for LSTD is given below.

LSTD for estimating $\hat{v} = \mathbf{w}^\top \mathbf{x}(\cdot) \approx v_\pi$ ($O(d^2)$ version)

Input: feature representation $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal) = \mathbf{0}$
Algorithm parameter: small $\varepsilon > 0$

$\widehat{\mathbf{A}^{-1}} \leftarrow \varepsilon^{-1}\mathbf{I}$ — A $d \times d$ matrix
$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$ — A d-dimensional vector
Loop for each episode:
  Initialize S; $\mathbf{x} \leftarrow \mathbf{x}(S)$
  Loop for each step of episode:
    Choose and take action $A \sim \pi(\cdot|S)$, observe R, S'; $\mathbf{x}' \leftarrow \mathbf{x}(S')$
    $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^\top (\mathbf{x} - \gamma \mathbf{x}')$
    $\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - \big(\widehat{\mathbf{A}^{-1}}\mathbf{x}\big)\mathbf{v}^\top / \big(1 + \mathbf{v}^\top \mathbf{x}\big)$
    $\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}$
    $\mathbf{w} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$
    $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$
  until S' is terminal

1.9 Memory-based Function Approximation

As an alternative to the parametric approaches discussed above, we might instead store all the training examples and execute an algorithm on the whole dataset when required, such as LOESS or nearest neighbour averaging. This approach is sometimes called *lazy learning*. The methods that go with this are non-parametric function approximation schemes. One can often evaluate the function approximation locally in the neighbourhood of the current state, which helps with the curse of dimensionality.

1.10 Kernel-based Function Approximation

Using kernels to define similarities between states for generalisation, e.g. kernel regression for state values.

1.11 Looking Deeper at On-policy Learning: Interest and Emphasis

Sometimes we are not equally interested in each state, so limited resources can be better spent than to treat every state equally. For instance, in discounted episodic problems we might be more interested in starting states because later rewards are discounted.

Introduce the scalar random variable $I_t \geq 0$ called $interest$, the degree of interest we have in accurately valuing the state at time t. If we don't cared al all about the state then $I_t = 0$, if we fully care then it might be 1 (but it is formally allowed to take any non-negative value). The interest can be set in any causal way. The distribution in our loss function $\overline{\text{VE}}$ is then defined as the distribution of states encountered when following the target policy, weighted by the interest.

We also introduce the scalar random variable $M_t \geq 0$, called the *emphasis*. The emphasis multiplies the learning update at each time-step. For general n-step learning

$$\mathbf{w}_{t+n} = \mathbf{w}_{t+n-1} + \alpha M_t [G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})] \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}_{t+n-1}) \quad 0 \leq t < T, \tag{14}$$

with the emphasis defined recursively as

$$M_t = I_t + \gamma^n M_{t-n} \tag{15}$$

with $M_t = 0 \ \forall t < 0$.