About Approximate Methods

- Extend tabular methods to problems with arbitrary large state spaces.
- In this case, we cannot put every state in a table and just record the associated reward. All our state space won't fit in memory, and even if it did, we won't have time to fill the table,
- we will use a function approximator that will take the state input and output the value of the state.
- Generalization Issue: we hope that our function approximator will generalize the state space, that is, if we get information about one state, it can be useful for similar states too, so we can actually learn something for states we don't see.

9.0 Chapter Intro

- Approximating v_π from experience generated using a known policy π.
- The approximate value function is represented not as a table but as a parametrized function with weight vector $\vec{w} \in \mathbb{R}^d$. We hope that $\hat{v}(s,w) \approx v_\pi(s)$., where:
  - $\hat{v}(s,w)$ is our approximate value function
  - $v_\pi(s)$ is the true value function
  - w is the weight vector (parameter we learn).

Note:

- $\hat{v}$ might be a linear combination of state features or a neural network, or the function computed by a decision tree, etc.
- Typically the # of weights is much smaller than the # of states: $d << |S|$.
- Approximation makes RL applicable to partially observable problems. If the parameterized function does not allow the estimated value to depend on certain aspects of the state, then it is just as if those aspects are unobservable.

9.1 : Value-function Approximation

All the prediction methods covered so far involve an update to an estimated value function that shifts its value towards a backed-up value (update target).

Let's denote an individual update by $s \longmapsto u$. For example:

• Monte Carlo (MC):	$S_t \rightarrow G_t$
• TD(0):	$S_t \rightarrow R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t)$
• n-step TD:	$S_t \rightarrow G_{t:t+n}$
• Dynamic Programming (DP):	$s \rightarrow E_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) \mid S_t = s]$

we can interpret each update as an example of the desired input-output behavior of the value function.

- up to now the update was trivial: table entry for state s's estimated value is shifted a fraction of the way towards the update while other states' estimates are unchanged.
- Now updating s generalizes so that the estimated value of many other states are changed as well.

[supervised learning] Function approximation methods expect to receive examples of the desired input-output behavior of the function they are trying to approximate.

- we view each update as a conventional training example,
- it's important to be able to choose a function approximation method suitable for online learning.

9.2: The Prediction Objective $\overline{VE}$

Assumptions we made in the tabular setting:

- No explicit objective needed; we could make each state's estimate exactly equal the true value.
- Updating one state does not affect others.

With function approximation, both assumptions break down;

- we cannot perfectly represent every state (limited parameters)
- updating parameters affect all states.

Mean Squared Error Objective:

Error in a state = Mean Squared Error (MSE) btwn approximate value $\hat{v}(s, w)$ and true value $v_\pi(s)$:

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

// Mean Squared Error
// Error in a state.

where:

- $\mu(s) \geq 0$ is a distribution representing how much we care about accuracy in state s.
  - The objective weighs the squared error in each state by $\mu(s)$.
  - Often $\mu(s)$ is chosen to be the fraction of time spent in state s under policy π. This is called the on-policy distribution. This is our focus.

Continuous vs episodic tasks

In continuing tasks, the on-policy distribution is the stationary distribution under π.

In episodic tasks, it depends on how the initial states of episodes are chosen:

- let $h(s)$ denote the probability that an episode begins in each state s.
- let $\eta(s)$ denote the # of time steps spent, on average, in state s in a single episode.
- time is spent in a state s if:
  - episodes start in s, OR:
  - transitions are made into s from a preceding state $\bar{s}$ in which the time is spent.

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

- This system of equations can be solved for the expected # of visits $\eta(s)$. The on-policy distribution is then the fraction of time spent in each state normalized to sum to 1.

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

Performance objective: is it the right metric?

- $\overline{VE}$ is a good starting point but it is not clear that it is the right performance objective.
- Ultimate goal (reason why we are learning a value function): find a better policy.
- If we stick to $\overline{VE}$, the goal is to find a global optimum, a weight vector w^* for which $\overline{VE}(w^*) \leq \overline{VE}(w) \quad \forall w$.
- Complex approximators may aim for a local optimum.

9.3: Stochastic-gradient and semi-gradient methods

9.3.1: Stochastic gradient

In gradient descent methods,

- the weight vector is a column vector w/ a fixed # of real valued components:
  $w \doteq (w_1, w_2, \ldots w_d)^T$
- The approximate value function $\hat{v}(s, w)$ is a differentiable function of w for all s,
- we update w at each step; w_t is the weight vector at step t.
- At each step we observe a new example $S_t \mapsto v_\pi(S_t)$, assuming for now that we have the true value $v_\pi(S_t)$,
- we want to find the best possible w, by minimizing error on the observed samples:

$$w_{t+1} \doteq w_t - \frac{1}{2}\alpha \nabla \left[v_\pi(S_t) - \hat{v}(S_t, w_t) \right]^2$$

$$= w_t - \alpha \left[v_\pi(S_t) - \hat{v}(S_t, w_t) \right] \nabla \hat{v}(S_t, w_t),$$

where:

α = learning rate,

$\nabla f(w)$ = column vector of partial derivatives of the expression wrt components of the vector;

$$\doteq \left(\frac{\partial f(w)}{\partial w_1}, \frac{\partial f(w)}{\partial w_2}, \ldots, \frac{\partial f(w)}{\partial w_d} \right)^T$$

- The overall step in w_t is proportional to the negative gradient of the example's squared error.

- SGD bc the update is done on a single randomly selected training example at each time step. (Traditional GD uses all training example to compute gradient).
- over many examples we make many small steps and the overall effect is to minimalize an average performance (here $\overline{VE}$).
- Convergence to a local optimum is guaranteed depending on α.

9.3.2: True value estimates

- Actually, we don't know the true value $v_\pi(S_t)$ but only have an approximation U_t! Just plug it in place of $v_\pi(S_t)$ in the update:

$$w_{t+1} \doteq w_t - \alpha\left[U_t - \hat{v}(S_t, w_t)\right]\nabla\hat{v}(S_t, w_t)$$

MC Estimate: $U_t = \sum_{i=t}^{T} \gamma^{i-t-1} R_i$

- unbiased; on average it $= v_\pi(S_t)$.
- fixed when w changes
- converges near global optimum.
- Need to wait till episode ends.

Bootstrap Estimate: $U_t = R_t + \gamma\hat{v}(S_{t+1}, w)$

- Biased; using current imperfect weight vector.
- Function of $w \rightarrow$ changes with w
- Gradient calc treated U_t as constant.
- converges to TD fixed point (local optimum)

For TD fixed point w_{TD}: $\overline{VE}(w_{TD}) \leq \frac{1}{1-\gamma}\min_w\left[\overline{VE}(w)\right]$

when the approximator is linear in features / basis, ie: $\hat{v}(S_t, w_t) = w_t \cdot x(S)$, then:

$$w_{t+1} \leftarrow w_t + \alpha \left[G_t - \hat{v}(S_t, w_t) \right] x(S_t)$$ since

$$\frac{d\hat{v}(s)}{dw_i} = x_i(s)$$

assigns "credit" to most activated features for success and failure.

9.3.2.1: Monte Carlo

- If U_t is an unbiased estimate (like MC), then we have convergence guarantees under certain conditions for α.
- $U_t = G_t$ (actual return after completing an episode).

Algo: on-policy prediction.

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, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G_t - \hat{v}(S_t, \mathbf{w}) \right] \nabla \hat{v}(S_t, \mathbf{w})$

9.3.2.2: Bootstrapping (n-step or DP targets)

- $U_t = G_{t:t+n}$ (n-step target) — the target depends on w_t
- $U_t = \sum_{a,s',r} \pi(a|S_t)\, p(s', r \mid s, a)\left[r + \gamma \hat{v}(s', w_t)\right]$ (DP target)
- Both are **biased estimates** bc they use the current value of the weight vector w_t.
- **Semi-gradient methods**, they do not converge as robustly as gradient methods but do converge reliably in important cases like linear cases (discussed later).
  - called semi-gradient bc the target itself depends on the weights being updated, but here is assumed to be static.
- Enable fast learning w/ online setting.

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'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$ (treated as constant)
    $S \leftarrow S'$
  until S is terminal

// take into account the effects of changing weight vector on estimate ($v(S_t|w_t)$), but ignore the effects on the target ($v(S_{t+1}|w_t)$)

9.3.2.3: State Aggregation

Simple form of generalizing function approximation in which states are grouped together with one estimated value (one component of w) per group.

9.4: Linear Methods

One of the simplest cases for function approximation is when the approximation function $\hat{v}(\cdot, w)$ is a linear combination of the weight vector w. In this setting, every state is represented as a vector $x(s)$, and the approximated state-value is just the inner product btwn w and $x(s)$:

$$\hat{v}(s, w) \doteq w^T \cdot x(s) \doteq \sum_{i=1}^{d} w_i x_i(s) \quad , \text{ where :}$$

- w is the weight vector
- $x(s)$ is a feature vector representing state s

- For linear methods, features are basis functions because they form a linear basis of the set of approximate functions.
- Using SGD with linear function approximation is easy bc the gradient of the approximate value function wrt w is:

$$\nabla \hat{v}(s, w) = x(s).$$

- Thus the linear case SGD update is:

$$w_{t+1} = w_t + \alpha \left[u_t - \hat{v}(s_t, w_t) \right] \cdot x(s_t)$$

- Easy form that has only one optimum so many methods are guaranteed to converge to a global optima.

9.4.1: Semi-gradient convergence (linear method)

- Semi-gradient TD(0) also converges to a point near the local optimum under linear function approximation.

Starting with the general update and substituting linear function approximation, the update at time t is:

$$w_{t+1} = w_t + \alpha \Big[\underbrace{\big(R_{t+1} + \gamma w_t^T x_{t+1}\big)}_{\text{TD(0) target} = R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t)} - \underbrace{w_t^T x_t}_{\hat{v}(S_t, w_t)} \Big] \underbrace{x_t}_{\nabla \hat{v}(S_t, w_t)}$$

$$\Rightarrow \boxed{w_{t+1} = w_t + \alpha \left[R_{t+1} x_t - x_t \left(x_t - \gamma x_{t+1} \right)^T w_t \right]}$$

(Notation shorthand: $x_t = x(S_t)$)

Once the system has reached steady state, for any given w_t the expected update becomes:

$$E\left[w_{t+1} \mid w_t \right] = w_t + \alpha \left(b - A w_t \right), \text{ where:}$$

- $b \doteq E\left[R_{t+1} x_t \right] \in \mathbb{R}^d$
- $A \doteq E\left[x_t \left(x_t - \gamma x_{t+1} \right)^T \right] \in \mathbb{R}^{d \times d}$

From $E\left[w_{t+1} \mid w_t \right] = w_t + \alpha \left(b - A w_t \right)$ it is clear that, if the system converges, it must converge to the weight vector w_{TD} at which:

$$b - A w_{TD} = 0 \quad \text{// this is needed to satisfy } E\left[w_{t+1} \mid w_t \right] = w_t.$$

$$\Rightarrow b = A w_{TD}$$

$$\Rightarrow \boxed{w_{TD} \doteq A^{-1} b.} \quad \text{// TD fixed point.}$$

TD fixed point = the state of equilibrium. The value estimates of a policy no longer changes.

- $\mathbf{A}$ needs to be positive definite to ensure that $\mathbf{A}^{-1}$ exists and to ensure stability (shrinking of w).

Expansion Bound: There is a bound to the error. At the TD fixed point, it has been proven that $\overline{VE}$ is within a bounded expansion of the lowest possible error (attained by the MC method):

$$\overline{VE}(w_{TD}) \leq \frac{1}{1-\gamma} \min_{w} \overline{VE}(w)$$

// continuing case,

- γ is often near 1 so the expansion factor can be large.
- But TD method have a much lower variance than MC and are faster.
- An analogous bound can be found for other on-policy bootstrapping methods (for example, DP):
  - one-step action-value methods such as semi-gradient SARSA(0) converge to an analogous fixed point and bound.
  - For episodic tasks there is also a bound.

The n-step semi-gradient TD extends TD(0) to n-steps:

Pseudocode:

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

Input: the policy π to be evaluated // purpose = policy evaluation!
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 // n = # of steps to look ahead.
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 $t = 0, 1, 2, \ldots$: // main loop that steps thru time.
  | If $t < T$, then: // if episode is not yet over.
  | 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: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$ // bootstrap. $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\,[G - \hat{v}(S_\tau,\mathbf{w})]\,\nabla\hat{v}(S_\tau,\mathbf{w})$ // weight update.
  Until $\tau = T - 1$

The key equation of this algorithm is:

$$w_{t+n} \doteq w_{t+n-1} + \alpha\left[G_{t:t+n} - \hat{v}\left(S_t, w_{t+n-1}\right)\right]\nabla\hat{v}\left(S_t, w_{t+n-1}\right)$$

where the n-step return is generalized to:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}\left(S_{t+n}, w_{t+n-1}\right),$$
$$0 \leq t \leq T - n.$$

9.5: Feature Construction for Linear Methods

One part of function approximation is that states are represented by feature vectors, there are several ways to create feature representations.

9.5.1: Polynomials:

For example, a state has 2 numerical dimensions, s_1 and s_2. $x(s) = (s_1, s_2)^T$ is not able to represent any interaction btwn the 2 dimensions, so we can represent s by: $x(s) = (1, s_1, s_2, s_1 s_2)^T$. The initial 1 feature allows for the representation of affine functions in the original state numbers.

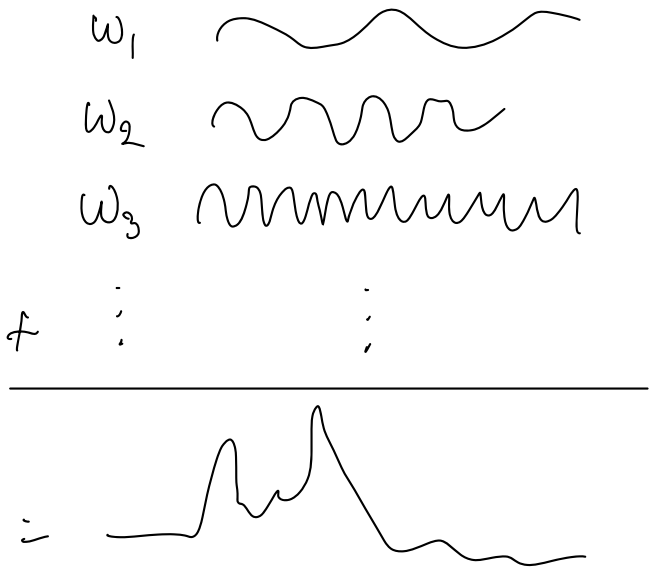
It is generally necessary to select a subset of the feature for function approximation. This can be done using prior beliefs about the nature of the functions to be approximated.

Suppose each state s corresponds to k numbers, $s_1, s_2, \ldots, s_k$, with each $s_i \in \mathbb{R}$. For this k-dimensional state space, each order-n polynomial-basis feature x_i can be written as

$$x_i(s) = \Pi_{j=1}^{k} s_j^{c_{i,j}}, \tag{9.17}$$

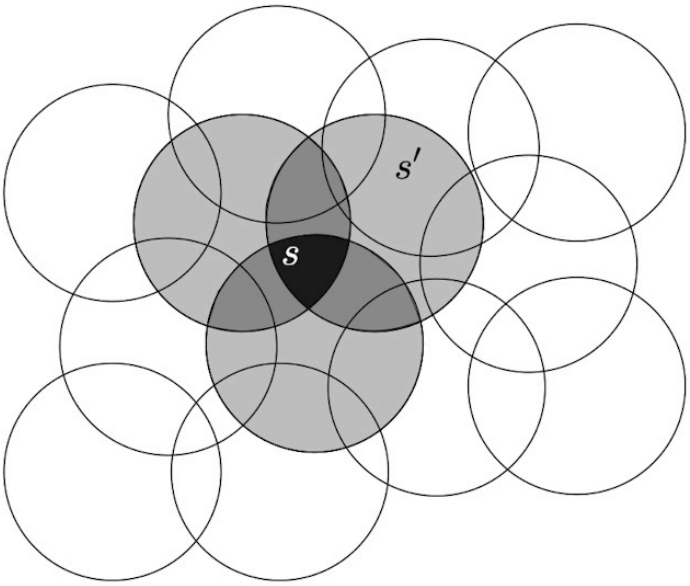
where each $c_{i,j}$ is an integer in the set $\{0, 1, \ldots, n\}$ for an integer $n \geq 0$. These features make up the order-n polynomial basis for dimension k, which contains $(n+1)^k$ different features.

9.5.2: Fourier:



Param: order? interaction terms?

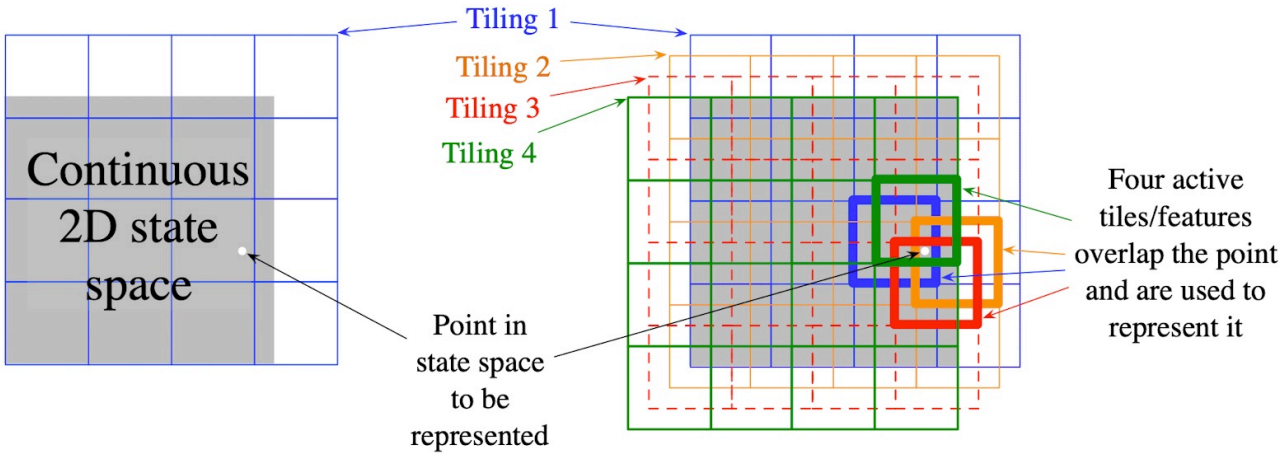
9.5.3: Coarse Coding: uses overlapping circles as features.



- Feature = circle in the 2D space,
- Each state is various union of circles (features),
- Assuming linear gradient descent function approximation, each circle has a corresponding weight (an element of w) that is affected by learning,
- If we update one state, the weights of all circles containing that state will be adjusted,
- More overlap among states = more generalization,

9.5.4: Tile Coding: a structured version of coarse coding.

- Coarse coding for multi-dimensional continous spaces.
- Receptive fields of the features are grouped into partitions of the state space (each partition is called a tiling and each element of the partition is a tile).
- Allows for better uniform generalization across state space compared to random circle placements (coarse coding).



- Tiles need not be grids. Can be irregular, diagonal stripes, etc!

9.5.4: Radial basis function (RBF)

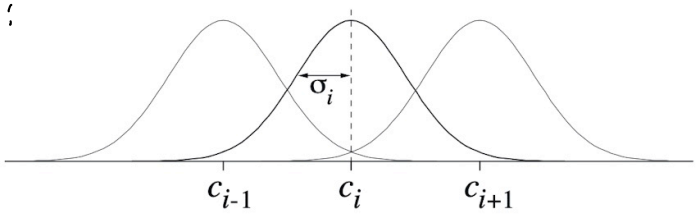
- Generalization of coarse coding to continous-valued features (instead of a feature being 0 or 1, it can be anything in the interval).
- A typical RBF feature x_i has a Gaussian response dependent only on the distance btwn the state s and the feature's center c_i, and relative to the feature's width σ_i:

$$x_i(s) \doteq \exp\left[-\frac{|s - c_i|^2}{2\sigma_i^2}\right] \in [0, 1]$$

(s: current state; c_i: center of the i^{th} RBF; σ_i: width of the i^{th} RBF.)

// function has value 1 if state s is at center c_i, decays towards 0 if moving away from c_i.

- The norm or distance metric can be chosen in the most appropriate way. One dimensional example w/ Euclidean distance metric:



1D RBF

$$\hat{v}(s, w) \approx \sum_{i=1}^{n} w_i x_i(s)$$ // n = # of RBF,

- Advantage: they produce approximate functions that vary smoothly and are differentiable,
- Doubts on pratical significance,
- An RBF network is a linear function approximator using RBFs for its features.
- Some learning methods for RBF networks change the center and widths of the features as well, making them nonlinear approximators.
- Nonlinear methods may be able to fit target functions much more precisely [greater computational complexity].

9.6: Selecting Step-size parameters manually

A good rule of thumb for setting the step-size parameter of linear SGD methods is:

$$\alpha \doteq \left(\mathbb{E}\left[X^\top X\right]\right)^{-1}, \quad \text{where:}$$

X = random feature vector

9.7: Non-linear function approximation : Neural Networks

In RL, NN can use TD errors to estimate value functions, or they can aim to maximize expected reward as in gradient bandit or a policy-gradient algorithm (as we shall see in Ch 13).

9.8: Least Squares TD

→ Direct computation of the TD fixed point instead of the iterative method.

Recall that TD(0) with linear function approximation converges asymptotically to the TD fixed point:

$$w_{TD} \doteq A^{-1} b, \text{ where}$$

- $b \doteq \mathbb{E}[R_{t+1} x_t]$
- $A \doteq \mathbb{E}[x_t (x_t - \gamma x_{t+1})^T]$

The least-square TD Algorithm (LSTD) computes directly estimates for A and b, rather than iteratively:

- $\hat{b}_t \doteq \sum_{k=0}^{t-1} R_{t+1} x_k$
- $\hat{A}_t \doteq \sum_{k=0}^{t-1} x_k (x_k - \gamma x_{k+1})^T + \varepsilon I$ (I: Identity matrix) where:
  - I is the identity matrix
  - εI ensures that $\hat{A}$ is always invertible.

It seems that these estimates should be divided t and they should, but we don't care because when we use them we effectively divide one by other.

LSTD estimates the fixed point as:

$$w_t \doteq \hat{A}_t^{-1} \hat{b}_t \quad \text{// TD fixed point}$$

complexity

- The computation involves the inverse of A, so complexity is $O(d^3)$.
- Fortunately our matrix has a special form of the sum of outer products, so the inverse can be computed incrementally with only $O(d^2)$. This is using the the Sherman-Morrison formula.
- Still less efficient than the $O(d)$ of incremental version, but can be handy depending on hows large d is.

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'; $\mathbf{x}' \leftarrow \mathbf{x}(S')$
    $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^\top (\mathbf{x} - \gamma \mathbf{x}')$
    $\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - (\widehat{\mathbf{A}^{-1}}\mathbf{x})\mathbf{v}^\top / (1 + \mathbf{v}^\top \mathbf{x})$
    $\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}$
    $\mathbf{w} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$
    $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$
  until S' is terminal

9.9: Memory-Based Function Approximation

So far, we discussed the parametric approach to function approximation. The training examples are seen, used by the algorithm to update the parameters, then discarded.

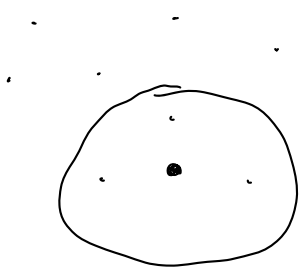
Memory-based function approximation saves the training examples as they arrive (or at least a subset of them) without updating the parameters. Then, whenever a value estimate is needed, a set of examples is recovered from memory and used to produce a value estimate for the queried state.

There are many different memory-based methods, here we focus on local learning:

- approximate a function only in the neighborhood of the current query state.
- retrieve states in memory based on some distance measure.
- the local approximation is discarded after use.

Some examples:

- Nearest neighbor method: local example where we retrieve from memory only the closest state and return that state's value as an approximation.



$$\hat{V}(s_q) = \frac{1}{k} \sum_{i=1}^{K} G(s_i)$$

(s_q: query state)

Ex: k = 3 here.

- weighted average: retrieve a set of closest states and weight their values according to distance.
- locally weighted regression: similar but fits a surface to the values of the nearest states with a parametric approximation method.

Properties of non-parametric memory methods:

- No limit of functional form: can represent any shape.
- More data = more accuracy.
- Local effect: learning affects neighboring states most directly.
- Experience effect more immediate to neighboring states.
- Retrieving neighboring states can be long if there are many data in memory (use of k-d trees may improve speed).

9.9: Kernel-Based Function Approximation

Memory-based methods depend on assigning weights to examples $s' \longmapsto g$ depending on the distance btwn s' and the query states s. The function that assign weights is called a kernel function or simply a kernel. $k(s, s')$ can be viewed as a measure of the strength of the generalization btwn s and s'.

Kernel Regression is the memory-based method that computes a kernel weighted average of all examples stored in memory:

$$\hat{v}(s, \mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s')\, g(s'), \quad \text{where:}$$

- $\hat{v}$ is the predicted value for query state s
- $\mathcal{D}$ is the set of stored examples.
- $g(s')$ denotes the target value for a stored state s'

RBF:

A common kernel is the Gaussian radial basis function (RBF) used in RBF function approximation. RBFs are features whose centers can be fixed at the beginning or adjusted during learning. For the fixed version, it's a linear parametric method whose parameters are the weights of each RBF, typically learned w/ SGD. Approximation is a form of linear combination of the RBFs.

Kernel regression with RBF differs a bit:

- Memory-based: the RBFs are centered on the states of stored examples.
- Non-parametric: no parameters to learn.

In practice, the kernel is often the inner product:

$$k(s, s') = x(s)^T x(s').$$

Complexity of kernel methods grows with # of data points, not # of features.

9.11 : Looking deeper at on-policy learning : Interest and Emphasis

- In this chapter we treated all states as if they were equally important, but we often have more interest in some states or some actions.

The on-policy distribution is defined as the distribution of states encountered in the MDP while following the target policy. Generalization: rather than having one policy for the MDP, we will have many. All of them are a distribution of states encountered in trajectories using the target policy, but they vary in how the trajectories are initiated.

Interest :

- New random variable I_t, called interest, indicating the degree to which we are interested in accurately valuing a state (or state-action pair) at time t.

The μ in the prediction objective is now defined as the distribution of states encountered while following the target policy, weighted by interest:

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

Emphasis:

- Another new random variable M_t, called **emphasis**. It is a non-negative scalar which multiplies the learning update and thus emphasizes or de-emphasizes the learning at time t:

$$w_{t+n} \doteq w_{t+n-1} + \alpha M_t \left[G_{t:t+n} - \hat{v}(S_t, w_{t+n-1}) \right] \nabla \hat{v}(S_t, w_{t+n-1})$$

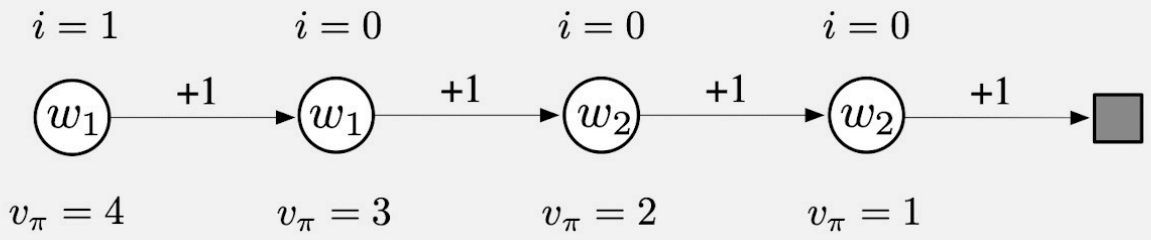
Emphasis is determined recursively from the interest:

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

- $0 \leq t < T$
- $M_t \doteq 0 \quad \forall \quad t < 0.$

Example 9.4: Interest and Emphasis

To see the potential benefits of using interest and emphasis, consider the four-state Markov reward process shown below:



Episodes start in the leftmost state, then transition one state to the right, with a reward of +1, on each step until the terminal state is reached. The true value of the first state is thus 4, of the second state 3, and so on as shown below each state. These are the true values; the estimated values can only approximate these because they are constrained by the parameterization. There are two components to the parameter vector $\mathbf{w} = (w_1, w_2)^\top$, and the parameterization is as written inside each state. The estimated values of the first two states are given by w_1 alone and thus must be the same even though their true values are different. Similarly, the estimated values of the third and fourth states are given by w_2 alone and must be the same even though their true values are different. Suppose that we are interested in accurately valuing only the leftmost state; we assign it an interest of 1 while all the other states are assigned an interest of 0, as indicated above the states.

First consider applying gradient Monte Carlo algorithms to this problem. The algorithms presented earlier in this chapter that do not take into account interest and emphasis (in (9.7) and the box on page 202) will converge (for decreasing step sizes) to the parameter vector $\mathbf{w}_\infty = (3.5, 1.5)$, which gives the first state—the only one we are interested in—a value of 3.5 (i.e., intermediate between the true values of the first and second states). The methods presented in this section that do use interest and emphasis, on the other hand, will learn the value of the first state exactly correctly; w_1 will converge to 4 while w_2 will never be updated because the emphasis is zero in all states save the leftmost.

Now consider applying two-step semi-gradient TD methods. The methods from earlier in this chapter without interest and emphasis (in (9.15) and (9.16) and the box on page 209) will again converge to $\mathbf{w}_\infty = (3.5, 1.5)$, while the methods with interest and emphasis converge to $\mathbf{w}_\infty = (4, 2)$. The latter produces the exactly correct values for the first state and for the third state (which the first state bootstraps from) while never making any updates corresponding to the second or fourth states.

9.12 : Summary

- we need generalization and supervised learning function approximation can do it, by treating each update as a training example.
- Crucial use of a weight vector w as the parameters in parameterized function approximation.
- The $\overline{VE}$ measures gives an error to rank the different approximations.
- use SGD to find a good weight vector, also semi-gradient methods like TD methods where bootstrapping makes the weight vector appear in the update target.
- Good results for semi-gradient methods in the linear case which is the most well-understood theoretically.
- LSTD to find a solution analytically when the # of weights is reasonable.
- Nonlinear methods $\longrightarrow$ deep reinforcement learning!