9. On-policy Prediction with Approximation

Notes

- Function approximation is used here to estimate the state-value function from on-policy. This means that v_π is approximated given a known policy π. The approximate value function is given by $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$, where $\mathbf{w} \in \mathbb{R}^d$ refers to a weight vector.
- The weight vector $\mathbf{w}$ can be feature weights from a neural network or a decision tree. Typically, the number of weights d is much less than the number of states: $d \ll |\mathcal{S}|$, and changing one weight changes the estimated value of many states. This also means that updating one state will consequently affect the values of many other states. This *generalisation* makes learning potentially more powerful but also more difficult to understand.
- Function approximation is useful in partially observable problems. For a parameterised function $\hat{v}$ which does not allow the estimated value to depend on certain aspects of the state, then it is the same as those aspects being unobservable.
- Value-function Approximation
  - We can refer to an individual update by the notation $s \mapsto u$, which can be interpreted as the estimated value for state s is shifted towards the update target u. For example, MC updates can be written as $S_t \mapsto G_t$, or n-step TD updates as $S_t \mapsto G_{t:t+n}$.
  - In function approximation, instead of a simple fractional shift towards u, ,more complex and sophisticated methods can be used to implement the update, so that updating s generalises into changes in other states as well.
  - It is important that learning occurs online, while there is interaction between the environment or a model of it. These learning methods should also be able to handle non-stationary target functions.
- The Prediction Objective ($\overline{\text{VE}}$):

- In function approximation, it is necessary to have an objective for prediction since unlike the tabular case, we are not guaranteed convergence to the true value function. It is also important because making one state's estimate more accurate is likely to make others' less accurate.
- The prediction objective, the *mean squared value error*, is given by

$$\overline{\mathrm{VE}}(\mathbf{w}) \doteq \sum_s \mu(s)\big[v_\pi(s) - \hat{v}(s,\mathbf{w})\big]^2$$

  where $\mu(s) \geq 0, \sum_s \mu(s) = 1$ is the state distribution, which is how much we 'care' about the error in each state.
- Under on-policy training, the state distribution is called the *on-policy distribution*. In the continuing task case, the on-policy distribution is the stationary distribution under π. In the episodic case, we can let $h(s)$ denote the probability that an episode begins in state s and $\eta(s)$ be the expected number of time steps spent in a state in a single episode:

$$\eta(s)h(s)\sum_{\bar{s}}\eta(\bar{s})\sum_a \pi(a|\bar{s})p(s|\bar{s},a), \quad \mu(s) = \frac{\eta(s)}{\sum_{s'}\eta(s')} \ (\text{undiscounted})$$

- Ideally, we would want to find a *global optimum* where we have $\overline{\mathrm{VE}}(\mathbf{w}^*) \leq \overline{\mathrm{VE}}(\mathbf{w})$. However, just like a lot of problems in machine learning, the *local optimum* is the best we can do for nonlinear function approximators, and this is often enough.

- Stochastic-gradient and Semi-gradient Methods:
  - For an example where $S_t \mapsto v_\pi(S_t)$, even though we are given the true value, it is still difficult for our function approximator to be exactly correct, since we have limited resources and resolution, ie. there is no $\mathbf{w}$ that can get all the values exactly correct.
  - *Stochastic gradient-descent methods* adjusts the weight vectors by a small amount:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla\big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\big]^2 \\ &= \mathbf{w}_t + \alpha\big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\big]\nabla\hat{v}(S_t,\mathbf{w}_t)\end{aligned}$$

Note that we do not seek to attain zero error for all states, but only an approximation that balances the errors in each state. It is also known that SGD methods converges to the local minimum only when α decreases over time in a specific manner.

- In the case where we have an approximation of the value $U_t \in \mathbb{R}$, where $S_t \mapsto U_t$, we can just replace $v_\pi(S_t)$ with U_t in the equation above. If U_t is an unbiased estimate, then $\mathbf{w}_t$ is guaranteed to converge to a local optimum.
- For a biased estimate U_t, for example in DP methods with bootstrapping targets, the target often depends on the weight vector $\mathbf{w}_t$, and the equation above would not hold. *Semi-gradient methods* is the idea of taking into account the effect of changing the weight vector $\mathbf{w}_t$ on the estimate, but ignoring its effect on the target.
- Semi-gradient methods do not converge as well, but they do converge reliably in certain important cases, such as in the linear case. They are known the enable significantly faster learning, and also for learning to be continual and online, without waiting for the end of the episode. Pseudocode for the semi-gradient TD(0) method with is given in page 203.
- *State aggregation* is the idea of grouping states together, with one estimated value (one component of the weight vector $\mathbf{w}_t$ for each group. This means that the value of a state is estimated as its group's component, and only the component is updated when the state is updated.

- Linear Methods:
  - The linear case is one of the most important cases of function approximation, where for every state there is a corresponding real-valued vector, known as a *feature vector*, $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \dots, x_d(s))^\intercal$. The linear state-value function can then be written as

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\intercal \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s).$$

    Here, $x_i(s)$ are *basis functions* as they form a linear basis for the set of approximate functions. In SGD, the gradient of the value function is

$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$, and it gives a simple SGD update equation: $\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[U_t - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(S_t)$. For the linear case, the method is guaranteed to converge to a global optimum.

- For the semi-gradient TD(0), where $U_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$, the global optimum is not guaranteed. Ignoring the effect of the weight on the target, the update equation can be given as

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\big(R_{t+1} + \gamma \mathbf{w}_t^\intercal \mathbf{x}_{t+1} - \mathbf{w}_t^\intercal \mathbf{x}_t\big)\mathbf{x}_t \\ &= \mathbf{w} + \alpha\big(R_{t+1}\mathbf{x}_t - \mathbf{x}(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\intercal\big)\end{aligned}$$

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}] = \mathbf{w}_t + \alpha(\mathbf{b} + \mathbf{A}\mathbf{w}_t) \quad \text{where} \quad \mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d, \quad \mathbf{A} \doteq \mathbb{E}\big[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\intercal\big] \in \mathbb{R}^{d \times d}.$$

  It is clear from the equation above that if the system converges, it converges to the weight vector $\mathbf{w}_{\text{TD}}$ if $\mathbf{b} + \mathbf{A}\mathbf{w}_t = 0$, which leads to $\mathbf{w}_{\text{TD}} \doteq \mathbf{A}^{-1}\mathbf{b}$, which is called the *TD fixed point*. Linear semi-gradient TD(0) converges to this point, and a proof can be found in page 206.

- AT the TD fixed point, the $\overline{\text{VE}}$ for the continuing case is within a bounded expansion of the lowest possible error attained in the limit by the MC method:

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

  Note that γ is often near 1, so the error here can be quite large. This means that TD methods can have substantial loss in its asymptotic performance. However, TD methods often have reduced variance, and thus are faster when compared to MC methods.

- Some on-policy bootstrapping methods also have the same bound as the one above. It is important that the states are updated according to the on-policy distribution.
- Pseudocode for the n-step semi-gradient TD can be found in page 209.

- Feature Construction for Linear Methods:

- Choosing appropriate features is an important way of adding prior domain knowledge to the system. Features should correspond to the aspects of the state space along with any generalisation.
- A limitation of linear methods is that it cannot take into account any interactions between features.
- Polynomials:
  - Polynomials are used to link together two features that would otherwise be independent of one another in the linear case, in the simplest manner. A polynomial for a k-dimensional state space with order-n polynomials can be written as $x_i(s) = \prod_{j=1}^{k} s_j^{c_{i,j}}$.
  - The number of features in an order-n polynomial basis grows exponential with the dimension k, where there are $(n+1)^k$ features. Thus, it is generally necessary to select a subset of the features for function approximation.
  - Not recommended for online learning.
- Fourier Basis:
  - We can use sines and cosines to represent out features. Suppose each state s is represented by a vector of length k, $\mathbf{s} = (s_1, s_2, \ldots, s_k)^\intercal,\ s_i \in [0,1]$. The ith feature n the order-n Fourier cosine basis can be written as

    $$x_i(s) = \cos(\pi \mathbf{s}^\intercal \mathbf{c}^i), \quad \mathbf{c}^i = (c_1^i, \ldots, c_k^i)^\intercal,\ c_j^i \in \{0, \ldots, n\}$$

    where $j = 1, \ldots, n$ and $i = 1, \ldots, (n+1)^k$. The inner product $\mathbf{s}^\intercal \mathbf{c}^i$ assigns an integer in $\{0, \ldots, n\}$ to each dimension in $\mathbf{s}$, where the integer determines the feature's frequency along that dimension.
  - When using Fourier cosine features with semi-gradient learning algorithms, it is preferably to use a step-size parameter of $\alpha_i = \alpha / |\mathbf{c}^i|$, where α is the basic step-size parameter. Fourier cosine features with Sarsa can produce good performance compared to other basis functions.
- Coarse Coding:

    - In a two-dimensional state space, features can correspond to circles in the state space, such that if a state is inside a circle, the corresponding feature has a value of 1 (present), other wise 0 (absent). The idea of 1-0-valued feature is called a *binary feature*, and this idea of representing a state with overlapping features is called *coarse coding*.
    - If we train one state, the weights of all the circles intersecting that state will be affected, which results in the other states within the union of the circles being affected. If the circles are samll, then there will be a narrow generalisation over a short distance. If the circles are large, there will be a broad generalisation, which can limit the learned function to a coarse approximation.
- Tile Coding:
    - Tile coding is a form of coarse coding, where the space is partition with a uniform grid, called a *tiling*. A single tiling is essentially just a case of state aggregation.
    - The strength of tile coding is when multiple tilings are used, and each offset by a fraction of a tile width. A single state is represented by the tile in which it falls in for each tiling.
    - An advantage of tile coding is that the overall number of features (tiles) that are active at one time is the same for any state. This allows the step-size parameter to be set in an easy and intuitive way. Another advantage is that the binary features make computation less expensive.
    - The way in which the offsets of each tiling can affect how the states generalise. Uniform offsets result in a strong effect along the diagonal in many patterns, thus asymmetric offsets can help avoid these effects. If w denotes the tile width and n is the number of tilings, then $\frac{w}{n}$ is a fundamental unit.
    - For a continuous space of dimension k, it is recommended to use the first odd integers $(1, 3, 5, \ldots, 2k-1)$, with n set to an integer power of 2 greater than or equal to $4k$. For example, a 3 dimensional space will have offsets of $(0, 0, 0)$, $(1, 3, 5)$, $(2, 6, 10)$ and $(3, 9, 15)$.
    - The shape of the tiles is important as well as it can determine the generalisation in a particular direction. In practice, it is often desirable

to use different shaped tiles in different tilings.

- *Hashing* is used to reduce memory requirements, where random tiles from non-contiguous, disjoint regions are connected to one another to form a single tile. This works because high resolution is often only needed in a small fraction of the state space. It frees us from the curse of dimensionality.

- Radial Basis Functions:
  - This is a natural generalisation of coarse coding to continuous valued features, where the features can have values in the interval $[0, 1]$. The equation of a RBF is given by

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

    where c_i is the features prototypical or centre state and σ_i is the feature's width.
  - The key advantage of RBFs is that they produce approximate functions that are smooth and differentiable. However, this has no practical significance. The downside of RBFs is that it is more computational inefficient.
  - An *RBF network* is a linear function approximator using RBFs for its features. Some methods also allow the changing of the centres and widths of the RNF features, which makes it into a nonlinear function approximator.

- Selecting Step-Size Parameters Manually:
  - For most cases, the step-size parameter is set manually. Theoretically, the step-size parameter should be slowly decreasing in order to guarantee convergence, but this is often too slow.
  - In the tabular case, a step size of $\alpha = 1/\tau$ means that the estimate will approach the mean of its targets after around τ experiences with the state.
  - In general function approximation, there is no clear notion of number of experiences with a state, since we are in a continuous state space, rather than discrete. A good rule of thumb for the step-size parameter of linear

SGD method, if we want to learn in around τexperiences, is $\alpha \doteq (\tau \mathbb{E}[\mathbf{x}^\intercal \mathbf{x}])^{-1}$. This works best if the feature vector $\mathbf{x}$ does not vary greatly in length, where $\mathbf{x}^\intercal \mathbf{x}$ is constant.

- Nonlinear Function Approximation: Artificial Neural Network:
  - Artificial Neural Networks (ANNs) are widely used for nonlinear function approximation.
  - In an ANN, the units are usually semi-linear units, where they compute a weighted sum of the input signals and feed it into a non-linear function called an *activation function*.
  - An ANN with a single hidden layer containing a large enough finite number of non-linear units can approximate any continuous function. The non-linearity is the key to ANNs. Deeper neural networks are often required to approximating complex functions for many tasks in AI.
  - Training the hidden layers of an ANN is a way to automatically create features appropriate for a given problem without hand crafting these features. The backpropagation algorithm is often used to/ learn the weights of these layers.
  - Overfitting is often a common problem in deep ANNs due to the large number of weights. Methods such as cross validation, regularisation and weight sharing are common methods to counteract the effects of overfitting. A popular method to reducing overfitting is using dropout, where units and its connections are randomly removed from the network during training.
  - Deep belief networks are used to train the deep layers of a deep ANN, where the deepest layer of the network is trained using an unsupervised learning algorithm, which can extract features that capture statistical regularities of the input stream. The next deepest layer is trained the same way, and so on. The final step involves backprapogation to fine-tune the network. This generally works much better than backpropagation with random initialised values.
  - Batch normalisation is a technique that makes training deep ANNs easier, where it normalises an input batch to have zero mean and unit variance.

- Deep residual learning has been shown to be effective, where the residual function is defined to be the difference between a function and the identity function. In deep ANNs, this is done by adding a shortcut which skips a block of layers, and concatenates the input of the block to the output of the block.

- Least-Squares TD:
  - Here is a method for linear function approximation that requires more computation, but not computed iteratively. This is arguably the best that can be done. Recall that the TD fixed point is given by $\mathbf{w}_{\text{TD}} = \mathbf{A}^{-1}\mathbf{b}$, where $\mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\intercal],\ \mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t]$. However, why must be compute this solution iteratively.
  - We can compute estimates of $\mathbf{A}$ and $\mathbf{b}$, and then directly computing the TD fixed point, and this is called the *Least-Squares TD (LSTD)* algorithm. The estimates can be written as

$$\widehat{\mathbf{A}}_t \doteq \sum_{k=0}^{t-1}\mathbf{x}_k(\mathbf{x}_k - \gamma\mathbf{x}_{k+1})^\intercal + \epsilon\mathbf{I} \quad \text{and} \quad \widehat{\mathbf{b}}_t \doteq \sum_{k=0}^{t-1} R_{t+1}\mathbf{x}_k,$$

    giving $\mathbf{w}_t = \widehat{\mathbf{A}}_t^{-1}\widehat{\mathbf{b}}_t$. This method is the most data efficient form of linear TD(0), but it also more expensive computationally. The two estimates can be updated incrementally with t, and this can be done in $\mathcal{O}(1)$ per timestep. However, updating $\widehat{\mathbf{A}}_t$ involves an outer product, which has computational complexity of $\mathcal{O}(d^2)$. The memory required to hold the estimate is also $\mathcal{O}(d^2)$. We can also bypass the $\mathcal{O}(d^3)$ computation of the inverse of $\widehat{\mathbf{A}}_t$ by using the *Sherman-Morrison formula*, which is $\mathcal{O}(d^2)$:

$$\begin{aligned}\widehat{\mathbf{A}}_t^{-1} &= \left(\widehat{\mathbf{A}}_{t-1} + \mathbf{x}_{t-1}(\mathbf{x}_{t-1} - \gamma\mathbf{x}_t)^\intercal\right)^{-1} \\ &= \widehat{\mathbf{A}}_{t-1}^{-1} - \frac{\widehat{\mathbf{A}}_{t-1}^{-1}\mathbf{x}_{t-1}(\mathbf{x}_{t-1} - \gamma\mathbf{x}_t)^\intercal\widehat{\mathbf{A}}_{t-1}^{-1}}{1 + (\mathbf{x}_{t-1} - \gamma\mathbf{x}_t^\intercal)\widehat{\mathbf{A}}_{t-1}^{-1}\mathbf{x}_{\mathbf{t-1}}}\end{aligned}$$

  - The semi-gradient TD(0) only requires memory and per-step computation that is $\mathcal{O}(d)$. The greater data efficiency of LSTD compared to the

computational expense depends on how large d is and how important it is to learn quickly.

    - LSTD requires no step-size parameter, but still requires ϵ, which can dictate how fast and effective learning is. LSTD lack of step-size parameter also means that it never forgets, and often mechanisms need to be put in place to induce forgetting.

- Memory-based Function Approximation:
    - A *parametric* to function approximation adjusts the parameters of a functional form intended to approximate the value function over a problem's entire state space, where a training example is used to update the value estimate.
    - Memory-based function approximation are an example of *nonparametric* methods. This method saves training examples in memory as they arrive, but do not update any parameters. When a query state's value estimate is needed, a set of training examples is retrieved and used to compute a value estimate of the query state. This method is sometimes called *lazy learning*. Note that this method is not limited to a fixed parameterised class of functions, thus being *nonparametric*.
    - We focus on *local-learning* methods which approximate a value function only locally around a current query state. Often, the distance between the query state and the training examples is used to judge the relevance of the training example. The simplest method is called the *nearest neighbour*, where the value of the nearest example is used as the estimate of the query state. The *weighted average* methods retrieve a set of examples and weight them by the distance from the query state.
    - Memory-based methods main advantage is that they are not limited to pre-specified functional forms, which allows higher accuracy as more data accumulates. Memory-based local approximation methods are able to focus on relevant states in trajectory sampling, and there would be no need to have a global approximation in many areas of the state space.
- Kernel-based Function Approximation:
    - K*ernel functions* numerically express how relevant knowledge about one state is to any other state, where it is given by $k : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$, and

denoted by $k(s, s')$. *Kernel regression* is the method that computes a kernel weighted average of the targets of all examples stored in memory. Let $\mathcal{D}$ be the set of stored example, and $g(s')$ be the target for state s', then $\hat{v}(s, \mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s') g(s')$.

- A common kernel used is the Gaussian radial basis function (RBF). If the weights and/or centres and widths are learned here, then it becomes a (non)linear function approximation problem. If the RBFs are centred of the states of the stored example, then it becomes a memory-based kernel regression method.
- In a linear parametric regression method, we can write $k(s, s') = \mathbf{x}(s)^\intercal \mathbf{x}(s')$, which will produce the same results. Using this, one can just point out that we can just instead construct kernel functions directly without referring at all to feature vectors. This means that for high dimensional spaces, we only need to work with a scalar value rather than a high dimensional vector, and this is often called the *kernel trick*.

- Interest and Emphasis:
  - We have only considered methods that treat all states encountered equally. However, we are often only interested in some states and not others. If we can target our resources on states that are more relevant, the the performance of our function approximation can be improved.
  - We introduce a non-negative scalar variable I_t called the *interest*, which indicates the degree to which we are interested in accurately valuing the state. It may depend on the trajectory up to time t or the learned parameters at time t. The distribution μ is then defined as the distribution of states encountered while following the target policy, weighted by the interest.
  - We can also introduce another non-negative scalar variable M_t called the *emphasis*. This variable multiples the learning update and thus places the right emphasis on the learning done at time t. The learning rule can then be replaced by

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha M_t [G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1}] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1}),$$
$$M_t = I_t + \gamma^n M_{t-n}$$

- Adding these two variables would also mean that there will be multiple on-policy distribution rather than just one, since the distribution now depends on how the trajectories are initiated.

Exercises

9.1 In the tabular case, we have the update algorithm of

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)].$$

Comparing this equation to equation 9.15, we want our weight vector to be our value estimate from the tabular case. This can be achieve by letting $\mathbf{w} \in \mathbb{R}^N$ with N is the total number of states, which means that each weight vector element corresponds to a single state. Given $\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\intercal \mathbf{x}(s)$, we can let the feature vector $\mathbf{x}(s)$ be unity at the nth element corresponding to the nth state, and zero elsewhere (one-hot encoding). This would also mean that $\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$. For a particular state s denoted by index k\, we can then write the update algorithm as

$$w_{k,\, t+n} = w_{k,\, t+n-1} + \alpha[G_{t:t+n} - w_{k,\, t+n-1}], \text{ where } w_k = V(s).$$

9.2 There are $n+1$ elements in the set of $\{0, 1, \dots, n\}$, which means that $n+1$ values for the power of s_j. Given that there are k dimensions, a single feature can be generally be written as $s_1^a s_2^b \dots s_k^c$, where $a, b, c \in \{0, 1, \dots, n\}$. Thus, there are k times of $n+1$ permutations of a feature, therefore there are $(n+1)^k$ features.

9.3 Here, $n = 2$ since the largest power is a square, and thus $c_{i,j} = \{0, 1, 2\}$.

9.4 In this case, we can use strip tiles along the state dimension that is less likely to have an effect. Alternatively, we can use rectangular tiles with its length along the state dimension that is less likely to have an effect. Both of these methods mean that along the direction of the dimension that has more influence, the tiles are more 'dense'.

9.5 We are given that the learning preferably takes 10 presentations with the same feature vector before learning nears its asymptote. This means that we want $\tau = 10$. For any state, there is an activation of a single tile in each of the 98 tilings. Therefore, $\alpha = (10 * 98^2)^{-1} = 1.04 \times 10^{-5}$.

9.6 If $\tau = 1$, $\alpha = (\mathbb{E}[\mathbf{x}^\intercal \mathbf{x}])^{-1}$:

$$
\begin{aligned}
\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\big[U_t - \hat{v}(S_t, \mathbf{w}_t)\big]\nabla\hat{v}(S_t, \mathbf{w}_t) \\
&= \mathbf{w}_t + \frac{1}{\mathbb{E}[\mathbf{x}^\intercal\mathbf{x}]}\big[U_t - \hat{v}(S_t, \mathbf{w}_t)\big]\nabla\hat{v}(S_t, \mathbf{w}_t) \\
&= \mathbf{w}_t + \frac{1}{\mathbb{E}[\mathbf{x}^\intercal\mathbf{x}]}\big[U_t - \mathbf{w}_t^\intercal\mathbf{x}\big]\mathbf{x} \quad \text{since } \hat{v}(S_t, \mathbf{w}_t) = \mathbf{w}_t^\intercal\mathbf{x} \\
\mathbf{w}_{t+2} &= \mathbf{w}_{t+1} + \frac{1}{\mathbf{x}^\intercal\mathbf{x}}\big[U_t - \mathbf{w}_{t+1}^\intercal\mathbf{x}\big]\mathbf{x} \\
&= \mathbf{w}_{t+1} + \frac{1}{\mathbf{x}^\intercal\mathbf{x}}\left[U_t - \left[\mathbf{w}_t + \frac{1}{\mathbf{x}^\intercal\mathbf{x}}\big[U_t - \mathbf{w}_t^\intercal\mathbf{x}\big]\mathbf{x}\right]^\intercal\mathbf{x}\right]\mathbf{x} \\
&= \mathbf{w}_{t+1} + \frac{1}{\mathbf{x}^\intercal\mathbf{x}}\left[U_t - \mathbf{w}_t^\intercal\mathbf{x} + \frac{1}{\cancel{\mathbf{x}^\intercal\mathbf{x}}}\big[U_t - \mathbf{w}_t^\intercal\mathbf{x}\big]\cancel{\mathbf{x}^\intercal\mathbf{x}}\right]\mathbf{x} \\
&= \mathbf{w}_{t+1} + 0 \\
&= \mathbf{w}_{t+1}
\end{aligned}
$$

9.7 Recall that the logistic function is given by $f(x) = 1/(1 + e^{-x})$, and the learning algorithm is

$$
\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\big[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\big]\nabla\hat{v}(S_t, \mathbf{w}_t) \\
&= \mathbf{w}_t + \alpha\left[v_\pi(x) - \frac{1}{1 + e^{-x}}\right]\frac{e^{-x}}{(1 + e^{-x})^2}
\end{aligned}
$$

9.8 The cross entropy of a distribution q relative to a distribution p is defined as $H(p, q) = -\mathbb{E}_p[\log q] = -\sum_i p_i \log q_i$. In order to get a cross entropy loss, we use the logistic function to go from our value estimates to the distribution of p and q. Let $p \in \{y, 1 - y\}$ and $q \in \{\hat{y}, 1 - \hat{y}\}$, where $y = 1/(1 + e^{-v_\pi(S_t)})$ and $\hat{y} = 1/(1 + e^{-\hat{v}(S_t, \mathbf{w}_t)})$. The cross entropy loss can then be written as $H(p, q) = -y \log \hat{y} - (1 - y)\log(1 - \hat{y})$.

Replacing the squared error in the learning algorithm with the cross entropy loss:

$$
\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla\Big[-y\log\hat{y} - (1-y)\log(1-\hat{y})\Big] \\
&= \mathbf{w}_t + \frac{1}{2}\alpha\Big[-\nabla y\log\hat{y} - y\frac{\nabla\hat{y}}{\hat{y}} + \nabla y\log(1-\hat{y}) + (1-y)\frac{\nabla\hat{y}}{1-\hat{y}}\Big] \\
&??
\end{aligned}
$$