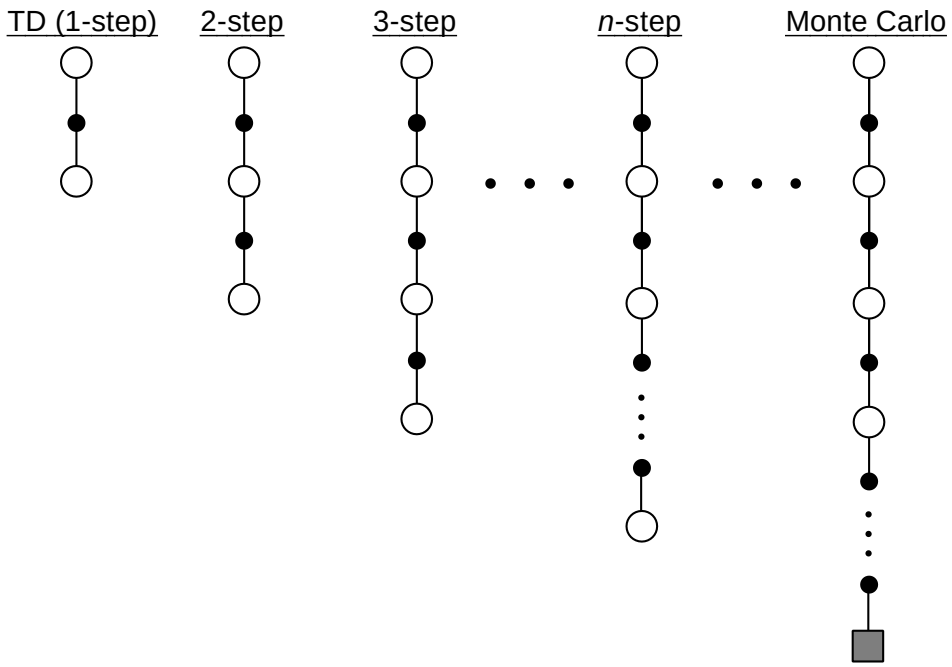
Lecture 21: Reinforcement Learning - Part III

- Recap of TD and Monte Carlo
- Eligibility traces
- Generalization and function approximation

N-step TD predictions

Main idea: look farther into the future when you do a TD backup



Mathematics of N-step TD prediction

- Monte Carlo algorithms use the full return as a target:

$$R_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-t-1} r_T$$

- TD uses the approximate value function $\hat{V}$ to estimate the return after the first step:

$$R_t^{(1)} = r_{t+1} + \gamma \hat{V}(s_{t+1})$$

- But we could truncate the return after any number of steps, and use $\hat{V}$ as an approximation for the rest. E.g. 2-step return:

$$R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 \hat{V}(s_{t+2})$$

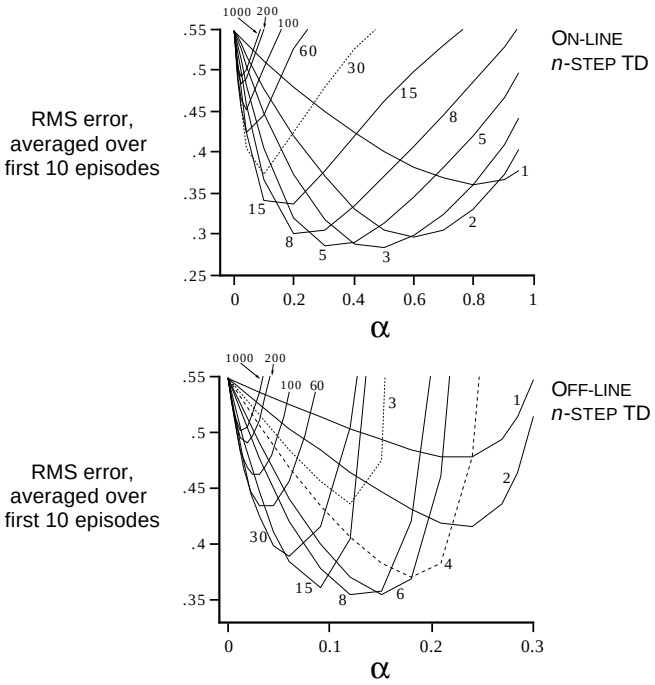
- In general, an N-*step return* is:

$$R_t^{(N)} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{N-1} r_{t+N} + \gamma^N \hat{V}(s_{t+N})$$

- *Issue: what is a good N?*

Some empirical evidence

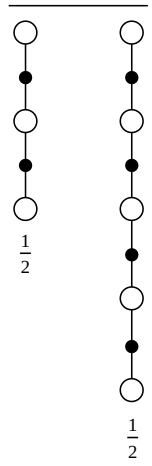
Random walk task, 19 states



Averaging N-step returns

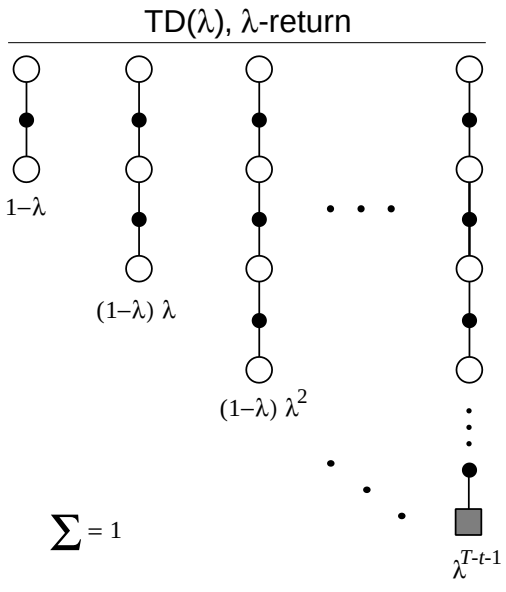
- Idea: backup an average of N-step returns

  E.g. Half of a 2-step and half of a 4-step return



- Since any N-step return reduces the error, any linear combination of them will also reduce the error
- $TD(\lambda)$ is a particular way of averaging *all* the N-step backups

Forward view of $TD(\lambda)$



λ-return:

$$R_t^\lambda = (1-\lambda)\sum_{N=1}^{\infty} \lambda^{N-1} R_t^N,$$

where λ is a parameter between 0 and 1

Backup using λ-return:

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha \left[R_t^\lambda - \hat{V}(s_t) \right]$$

This weighs each N-step backup by a weight λ^{N-1}, depending on the time since the state was visited.

Relationship to TD and Monte Carlo

Suppose we have a trial-base task, and the trial ends at T. The λ-return can be re-written as:

$$R_t^\lambda = (1-\lambda) \sum_{N=1}^{T-t-1} \lambda^{N-1} R_t^N + \lambda^{T-t-1} R_t$$

The first term shows truncated returns until termination, the last one shows the whole return for the trial

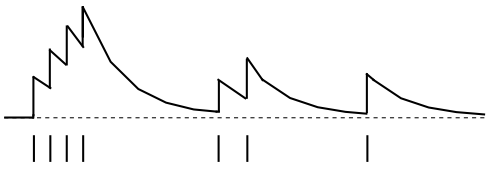
- If $\lambda = 1$, we get $R_t^\lambda = R_t$ - Monte Carlo!
- If $\lambda = 0$, we get $R_t^\lambda = R_t^{(1)}$ - Temporal difference! (we will call it TD(0) from now on)

λ provides a way of interpolating between TD(0) and Monte Carlo!

Eligibility traces

- The previous algorithm is helpful for understanding but is not convenient to implement directly
- For a convenient implementation, we use an extra variable for each state, $e_t(s)$, called the *eligibility trace*, which keeps track of how long ago the state was visited
- Update rule for the *accumulating eligibility trace*:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$



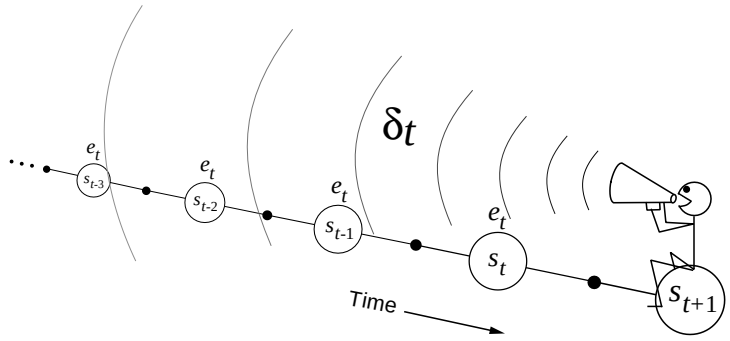
On-line tabular $TD(\lambda)$

1. Initialize $V(s)$ arbitrarily and $e(s) = 0$, for all states s
2. Pick a start state s
3. Repeat for every time step:
   (a) Choose action a based on policy π and the current state s
   (b) Take action a, observe immediate reward r and new state s'
   (c) Compute the TD error: $\delta \leftarrow r + \gamma V(s') - V(s)$
   (d) Mark the current state as visited: $e(s) \leftarrow e(s) + 1$
   (e) For all states s, update the value function and eligibility trace:

$$\begin{aligned} V(s) &\leftarrow V(s) + \alpha\delta e(s) \\ e(s) &\leftarrow \gamma\lambda e(s) \end{aligned}$$

   (f) $s \leftarrow s'$

What $TD(\lambda)$ does



- On every time step t, we compute the TD error:

$$\delta_t = r_{t+1} + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)$$

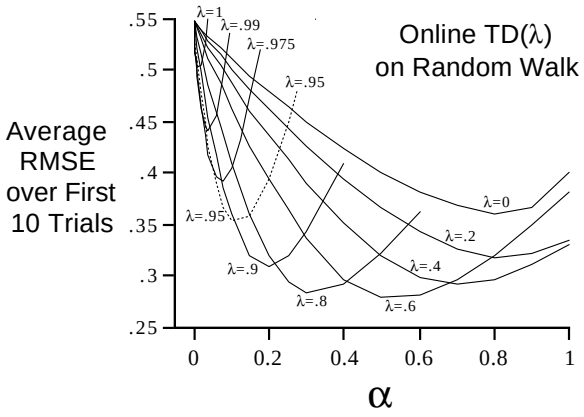
- Shout δ_t backwards to past states
- The strength of your voice decreases with temporal distance by $\gamma\lambda$

Relation of the on-line algorithm to TD(0) and Monte Carlo

- We can show (by algebraic manipulations) that the two algorithms discussed so far are equivalent
- In the on-line algorithm, setting $\lambda = 0$ gives us TD(0) (just like before)
- Setting $\lambda = 1$ gives us Monte Carlo, also as before - we call this TD(1)

  But this is a *better implementation of Monte Carlo!*
  - Straightforward to apply to continuing tasks
  - Works incrementally and on-line, instead of waiting until the end of the episode

Typical empirical results



- Typically intermediate values of λ converge the fastest
- A sharp decrease in performance happens at λ very close to 1
- The algorithm converges in the limit, with probability 1, to correct values, for any λ

Implementation issues

- A naive implementation would update all states (or state-action pairs) on every time step
- But in practice, for most values of γ and λ, the eligibility traces are very near zero for all states except those most recently visited
- A clever implementation can keep track only of the states with non-zero traces, which makes the algorithm a few times more expensive than TD(0)
- When using function approximation, extra expense is even less.

Summary of eligibility traces

- Eligibility traces provide an efficient, incremental way to combine temporal difference and Monte Carlo methods
- Like Monte Carlo methods, they are robust to lack of Markov property (see, e.g. Loch and Singh, 1998)
- But preserve the advantage of TD in terms of bootstrapping, incremental computation
- Can significantly speed up learning (many experiments indicate intermediate λ is consistently the best)
- But there is a cost in computation (now more than one state is updated on every step)

Why function approximation?

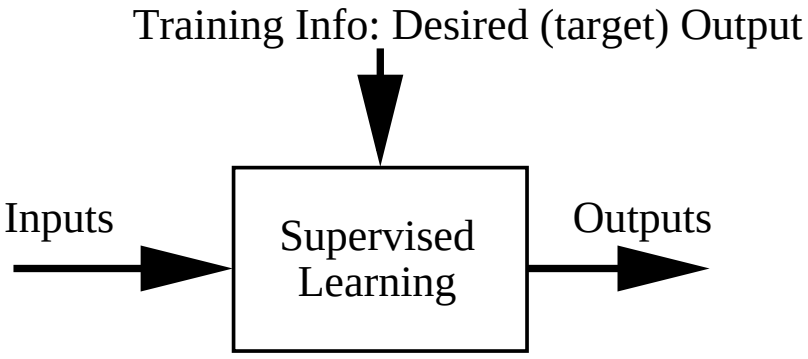
- In general, state spaces are continuous or too large to represent as a table
- If every state has a separate entry in the table, then every state has to be visited at least a few times before having a good approximation; in the limit every state should be visited infinitely often, which is not feasible

Main idea: Use a function approximator to generalize from the seen states to unseen ones

This is what supervised learning algorithms do too!

Adapt supervised learning algorithms

- A training example has an input and a target output
- The error is measured based on the difference between the actual output and the desired (target) output



Value-based methods

We will use a function approximator to represent the *value function*

- The input is a description of the state (or state-action pair)
- The output is the predicted value of the state (or state-action pair)
- The target output comes from the RL update rule
  E.g. for TD(0), the target would be $r_{t+1} + \gamma V(s_{t+1})$

What kind of function approximator can we use?

In principle anything we want

- A table where several states are mapped to the same location - *state aggregation*
- Gradient-based methods:
  - Linear approximators
  - Artificial neural networks
  - Radial Basis Functions
  - SVMs?
- Memory-based methods:
  - Nearest-neighbor
  - Locally weighted regression
- Decision trees

0.001in=0.401920.001in0.1in=0.401920.1in

Special requirements for the function approximator:

- Fast, incremental learning (so we can learn during the interaction)
- Ability to handle non-stationary target functions

Gradient Descent Methods

Consider the policy evaluation problem: learning V^π for a given policy π

The approximate value function $V(s_t) = f(\theta, \phi_t)$, where ϕ_t are the attributes (features) describing s_t, and θ is a **parameter vector**
E.g. θ could be the connection weights in a neural network

We will update θ based on the errors computed by the reinforcement learning algorithm

Performance measure

- We want to find a parameter vector θ that minimizes the mean squared error:

$$MSE(\theta) = \frac{1}{2} \sum_{s \in S} P(s) \left(V^{\pi}(s) - V(s)\right)^2$$

  What should P be?
- In our case P is the **on-policy distribution**: distribution of states created when the agent acts according to π

Gradient descent update

Works like in the supervised learning case:

$$\begin{aligned} \theta &\leftarrow \theta - \alpha \nabla_{\theta} MSE(\theta) \\ &= \theta - \alpha \nabla_{\theta} \frac{1}{2} \sum_{s \in S} P(s) \left(V^{\pi}(s) - V(s)\right)^2 \\ &= \theta + \alpha \sum_{s \in S} P(s) \left(V^{\pi}(s) - V(s)\right) \nabla_{\theta} V(s) \end{aligned}$$

To do this incrementally, we use the **sample gradient**:

$$\theta \leftarrow \theta + \alpha \left(V^{\pi}(s) - V(s)\right) \nabla_{\theta} V(s)$$

The sample gradient is an unbiased estimate of the true gradient. The rule would converge to a local minimum of the error function, if α is decreased appropriately over time

Using TD targets

Instead of V^π, we will use the targets that come from the $TD(\lambda)$ algorithm:

$$\theta \leftarrow \theta + \alpha \left(\nu_t(s) - V(s)\right) \nabla_\theta V(s)$$

If we use Monte Carlo, then $\nu_t = R_t$ is an unbiased estimate of the true value function, and the algorithm still converges to a local minimum, provided α is decreased appropriately

If $\nu_t = R_t^\lambda$ with $\lambda < 1$, ν_t is **not** an unbiased estimate, and we cannot say anything about the convergence in general
But the algorithm is well defined, and used in practice

On-line gradient descent $TD(\lambda)$

In addition to the weight vector θ, we will have an eligibility trace vector $\mathbf{e}$, with one eligibility for every weight

1. Initialize the weight vector θ arbitrarily, and $\mathbf{e}$ = 0.
2. Pick a start state s
3. Repeat for every time step t:
   (a) Choose action a based on policy π and the current state s
   (b) Take action a, observe immediate reward r and new state s'
   (c) Compute the TD error: $\delta \leftarrow r + \gamma V(s') - V(s)$
   (d) Compute the eligibility of every weight vector to be updated:

$$\mathbf{e} \leftarrow \gamma\lambda\mathbf{e} + \nabla_\theta V(s)$$

   (e) Update the weight vector: $\theta \leftarrow \theta + \alpha\delta\mathbf{e}$
   (f) $s \leftarrow s'$

Linear methods

Each state represented by feature vector $\phi(s) = (\phi_1(s) \ldots \phi_n(s))'$

The value function is a linear combination of the features:

$$V(s) = \theta \cdot \phi(s) = \sum_{i=1}^{n} \theta_i \phi_i(s)$$

So the gradient is very simple: $\nabla_\theta V(s) = \phi(s)$

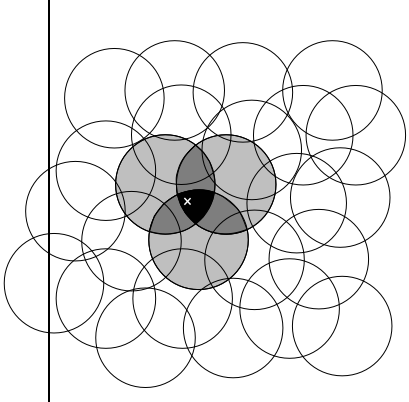
The error surface is quadratic with a single global minimum

Tsitsiklis and Van Roy: Linear gradient-descent $TD(\lambda)$ converges w.p.1 to a parameter vector θ_∞ in the "vicinity" of the best parameter vector θ^*:

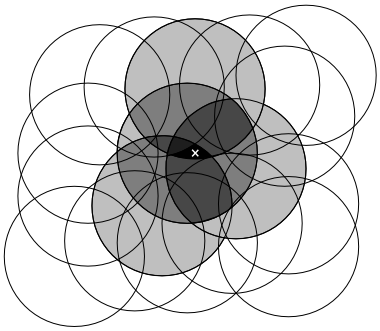
$$MSE(\theta_\infty) \leq \frac{1 - \gamma\lambda}{1 - \gamma} MSE(\theta^*)$$

Coarse coding

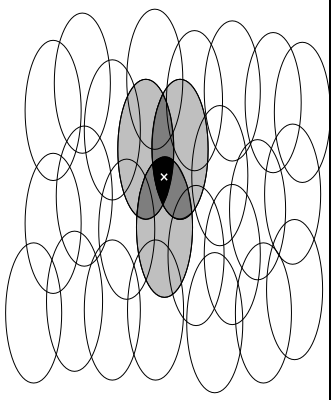
Main idea: we want linear function approximators, but with **lots of features**, so they can represent complex functions



a) Narrow generalization



b) Broad generalization



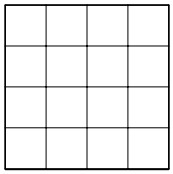
c) Asymmetric generalization

Discretizing the state space

Suppose we have a continuous state space with two continuous variable (e.g. like in the Mountain-Car task)

The simplest tile coding approximator would be just a grid discretizing the state space:

- The features are all 0 except for the cell holding the current state, which is 1 (like a 1-of-n encoding)
- All states in the same cell have the same value (given by the weight of the cell)



Pros and cons of discretizations

Pros:

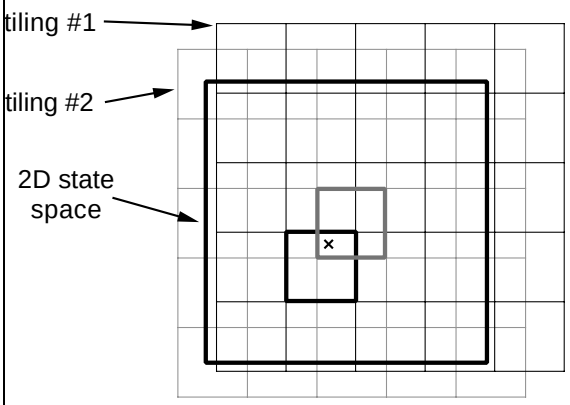
- Easy to compute the value function of a state
- Easy to update as well (more like the table lookup case).

Cons:

- To get good precision, we need a very fine grid - going back to the table lookup case?
- States in the vicinity of a separation line could have radically different values (approximation is discontinuous)

Tile coding (continued)

Main idea: Overlap several tilings!



Shape of tiles ⇒ Generalization

#Tilings ⇒ Resolution of final approximation

Characteristics of tile coding

- Each tile is a binary feature
- The number of features that are activated at any time is constant, equal to the number of tilings
- It is easy to compute the indices of the features activated, and easy to compute the weighted sum
- The overall discretization is very fine, and at the same time the discontinuities are smoothed out
- The shape of the tiles reflects prior domain knowledge

Cf. CMAC (Albus, 1971)

Summary of function approximation

- It is necessary for practical purposes!!!
- Proving convergence is much harder than in the tabular case
- Linear approximators tend to e well-behaved (and work well in practice!)
- Recent results indicate that convergence can be ensured if the policy changes slowly over time
- But this means slower learning...