Where we are

- We started by explaining how to solve RL problems: using value iteration and policy iteration.
- So is RL done? Class over?
- No because there are two problems:
  — we don't know the MDP
  — the state space is often huge
- We've discussed how to solve the first of these using a method that learns from experience: Q-learning (which was based on TD learning).
- But both Q-learning is still a *tabular* methods: it needs to maintain a table which has as many entries as the number of state-action pairs.
- Now we talk about the second problem: how do we modify Q-learning to avoid this?

- The basic idea is to approximate the Q-value (or value function) rather than compute it exactly.
- Let's first discuss this idea in the context of temporal difference learning.
- Recall TD is the a method which, at every step, updates the estimate of a single step s as

  $$J_{t+1}(s) = J_t(s) + \alpha_t \left(r_{t+1} + \gamma J_t(s') - J_t(s) \right)$$

  where s' is the next state after s; and for all other states, we keep the same estimate from time t to $t + 1$.
- Clearly, need to maintain a table to implement this, with as many entries in the table as the number of steps.
- What if, instead, we have some kind of method that approximates $J_t(s) \in \mathbb{R}^n$ from a smaller number of parameters $\theta \in \mathbb{R}^d$?

  Then maybe we could update the parameters θ from step to step.

This is a screenshot from AirSim, a software for self-driving cars.
The state space is obviously huge.
But what if we model value function as depending ONLY on a few parameters.
Let's think about what such parameters might be.
(1) v_o^r: car velocity orthogonal to the road if going right (zero if going left)
(2) v_o^l: car velocity orthogonal to the road going left (zero if going right)
(3) whether you're over any of the lines [0 or 1]

So we imagine that J(s) is a function of a vector x(s) in R^3
Presumably, we can train some method to extract these five things from the image

Next: model $J(s) = \theta_1 x_1(s) + \theta_2 x_2(s) + \theta_3 x_3(s)$
Start with theta = [1,1,1]
Every time a transition occurs...update theta.
This updates all the state estimates simultaneously

What will this learn?

It might learn the relative good state is one where:

— you are not over the lines (if theta 3 is a big negative)

— your orthogonal speeds are both close to zero (if theta_1, theta_2 are negative)

Well, this isn't perfect, but it's not a bad first approximation to the truth

We could make it better by increasing the number of parameters to more than 3

- So to summarize:

  — given a state s, we will extract certain features $x(s)$ from it

  [the problem of extracting features is non-trivial and often requires some other ML methods]

  We have $s \in \mathbb{R}^n, x(s) \in \mathbb{R}^d$ and $n >> d$.

  — we will model $J_\pi(s) \approx \theta^T x(s)$

  — every time the system transitions from a state s to a state s' and obtains a reward of r, we will update θ
- The best choice of θ is unknown
- On the other hand, the procedure for going from s to $x(s)$ is assumed to be known for now.

  ...think about the previous example. Can you think of more features to add?

- Let’s discuss one example of this: state aggregation.
- Consider a random walk on nodes {1,2,…,1000}.
  — If you visit the far-left, you terminate with a reward of -1. If you visit the far-right, you terminate with a reward of 1.
  — Discount factor is 1, all other rewards are zero.
  — Your policy makes you transition left and right with equal probability
- Goal: evaluate the value vector of this policy.
  (Can actually show: it grows linearly from left to right)
- OK, you can do this with TD(0).
- But let’s use this as a toy example to discuss something more complicated.
- Suppose we aggregate states 1-100, 101-200, 201-300, etc, to reduce this to a problem with 10 states.

- How we will do this: we map states to features in the following way:

  if $s \in \{1,\ldots,100\}$, then $x(s) = \mathbf{e}_1$

  if $s \in \{101,\ldots,200\}$, then $x(s) = \mathbf{e}_2$

  if $s \in \{201,\ldots,300\}$, then $x(s) = \mathbf{e}_3$

  etc
- So for $s \in \{1,\ldots,100\}$, we have $\hat{J}_\pi(s) = \theta^T \mathbf{e}_1 = \theta_1$

  So for $s \in \{101,\ldots,200\}$, we have $\hat{J}_\pi(s) = \theta^T \mathbf{e}_2 = \theta_2$

  etc
- In other words, with these features, linear approximation just means that all the states in $\{1,\ldots,100\}$ get the same value, and likewise all the states in $\{101,\ldots,200\}$
- We can then train the method to try to find the best $\theta_1, \ldots, \theta_{10}$
- OK, we haven't said anything about how to do that, but the following slide shows the final result.

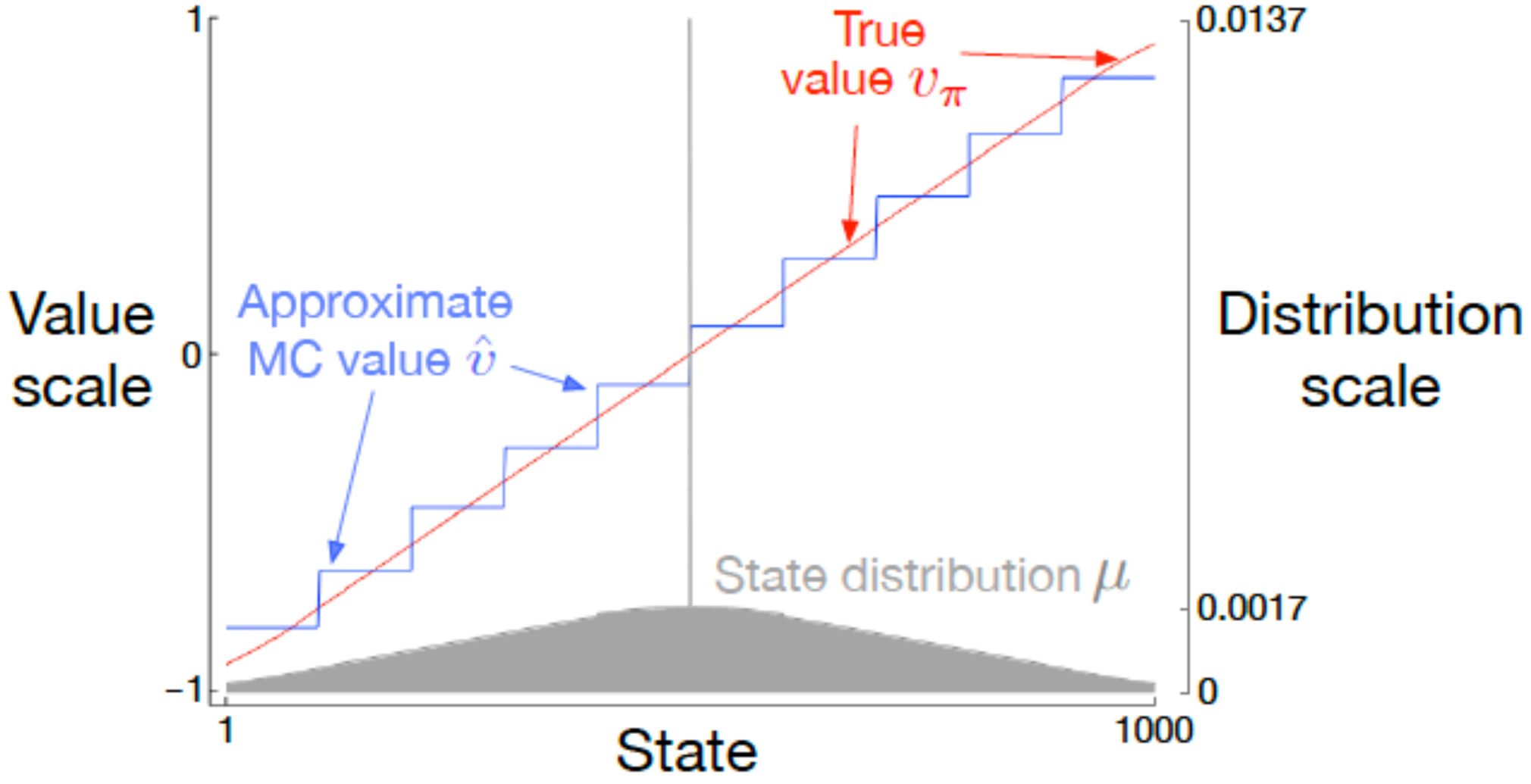


Figure 9.1 from Sutton & Barto. We haven't discussed how to compute the approximation, but this is what it would look like.

- OK, so we discussed what it means to extract features $x(s)$ from a state...and what kind of approximation we can hope to get in the context of policy evaluation.
- Now let's talk about how to generalize TD learning to the case when we are updating the linear approximation $\theta^T x(s)$.
- This will be an iterative method. We will start with θ_0, arbitrarily chosen.
- At step t, suppose we have the vector θ_t and are in state s. The policy takes action a, obtains reward r_{t+1}, and we transition to a state s'.
- In TD learning, we updated $J_{t+1}(s) = J_t(s) + \alpha_t(r_t + \gamma J_t(s') - J_t(s))$ ...but we can't do this anymore before we don't have a tabular method.
- But let's recall the interpretation of this: we obtain a new "piece of information" which is $r_{t+1} + \gamma J_t(s')$ about what the true value of state s should be, and we nudge ourselves in that direction.

- Let's think back to $J_{t+1}(s) = J_t(s) + \alpha_t \left(r_t + \gamma J_t(s') - J_t(s) \right)$
- Observe: every time $r_t + \gamma J_t(s') > J_t(s)$, then we **increase** from $J_t(s)$ to $J_{t+1}(s)$.
- Every time $r_t + \gamma J_t(s') < J_t(s)$, then we **decrease** from $J_t(s)$ to $J_{t+1}(s)$.
- OK, but now we are not going to update J_t directly. Instead, we think of $J_t(s)$ as a function of θ, and we update θ_t to θ_{t+1}

- So the way we want to think about it is as follows.
- We have our estimate of the value of node s: it is $\theta_t^T x(s)$.
- But now we get a "new" piece of information about what the value of this state should be: it is $V_{\text{new}} = r_t + \gamma \theta_t^T x(s')$.
- So we update θ_t as:

  $$\theta_{t+1} = \theta_t + \alpha_t(r_t + \gamma \theta_t^T x(s') - \theta^T x(s))$$
- This does not make sense. Why?
- Better:

  $$\theta_{t+1} = \theta_t + \alpha_t(r_t + \gamma \theta_t^T x(s') - \theta^T x(s))x(s)$$
- This is called **temporal difference learning with linear function approximation.**

- Consider temporal difference with linear function approximation:

  $$\theta_{t+1} = \theta_t + \alpha_t \left[r_t + \gamma \theta_t^T x(s') - \theta_t^T x(s) \right] x(s)$$

- Suppose $r_t + \gamma \theta_t^T x(s') > \theta_t^T x(s)$. Then the quantity in brackets is positive.
- We can think of $J_t(s) = \theta_t^T x(s)$ as "our belief about the value of state s at time t."
-
- As a consequence,

  $$\begin{aligned} J_{t+1}(s) &= \theta_{t+1}^T x(s) \\ &= (\theta_t + \text{(something positive)}\, x(s))^T x(s) \\ &\geq \theta_t^T x(s) = J_t(s) \end{aligned}$$

- ...Sounds good!

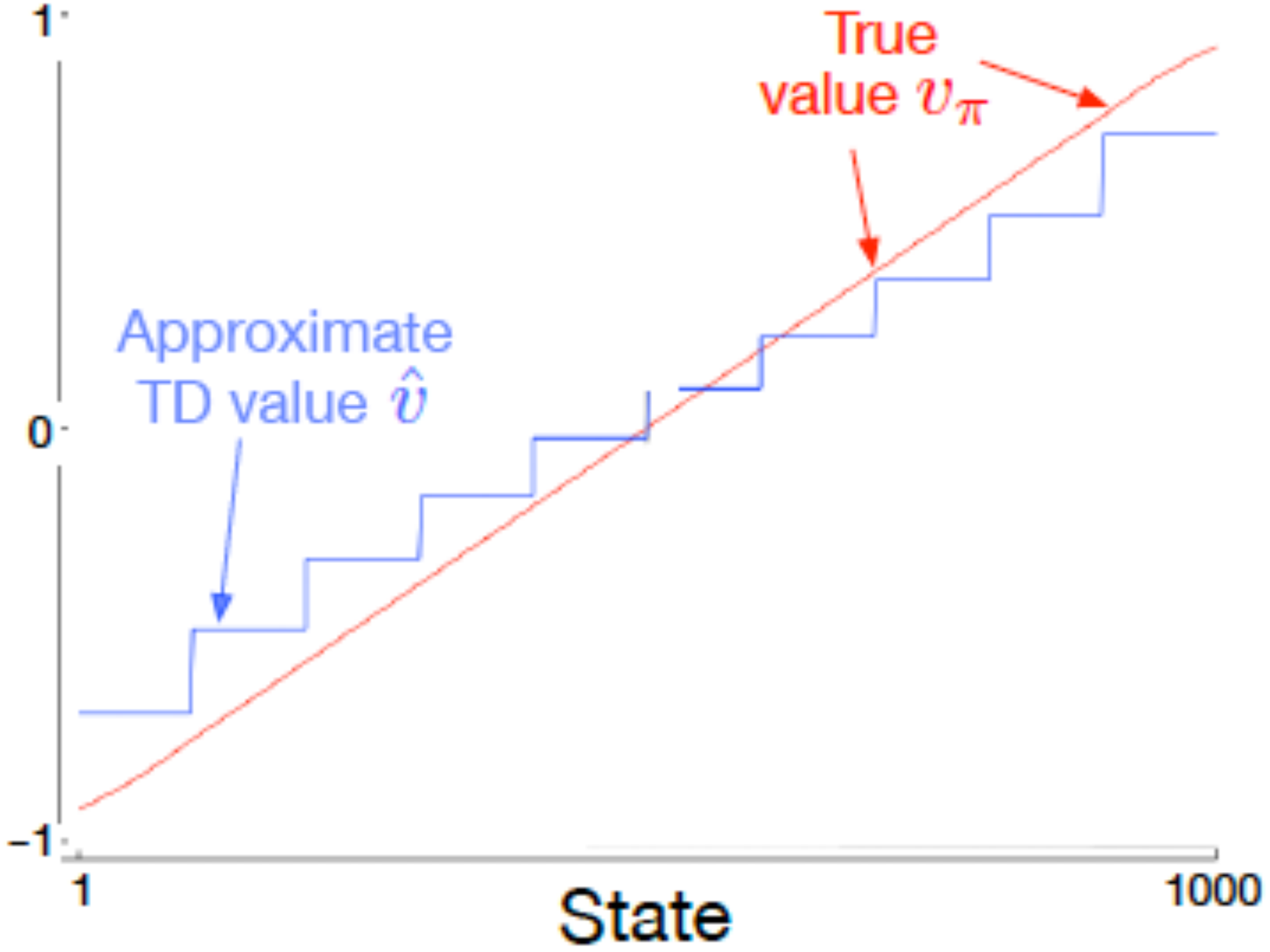
- Consider temporal difference with linear function approximation:

  $$\theta_{t+1} = \theta_t + \alpha_t \left[r_t + \gamma \theta_t^T x(s') - \theta_t^T x(s) \right] x(s)$$

- Likewise, suppose $r_t + \gamma \theta_t^T x(s') < \theta_t^T x(s)$. Then the quantity in brackets is negative.
- As a consequence,

  $$\begin{aligned} J_{t+1}(s) &= \theta_{t+1}^T x(s) \\ &= (\theta_t + \text{(something negative)}\, x(s))^T x(s) \\ &\leq \theta_t^T x(s) = J_t(s) \end{aligned}$$

- ...again, this is good.
- Keep in mind: when you update from θ_t to θ_{t+1}, you could be changing **every** $J_t(s)$ to a different $J_{t+1}(s)$



TD learning on the example from earlier
Long story short: it's OK

- Similarly, you can estimate the Q-values under a policy π.
- You know take a state-action pair (s, a) and extract features $x(s, a)$ from it.
- You then use the linear approximation

  $Q(s, a) \approx \theta^T x(s, a)$. You try to learn the vector θ from experience.
- At step t, you are at state s, take action a recommended by the policy, obtain a reward r_{t+1}transition to s', there take action a' recommended by the policy, and update as

  $\theta_{t+1} = \theta_t + \alpha_t(r_t + \gamma\theta_t^T x(s', a') - \theta_t^T x(s, a))x(s, a)$
- Can also do Q-learning: $\theta_{t+1} = \theta_t + \alpha_t(r_t + \gamma \max_{a'} \theta_t^T x(s', a') - \theta_t^T x(s, a))x(s, a)$

  ....again actions can be taken to maximize current Q_t, with random actions taken an ϵ fraction of the time.

- Let's talk about the big picture.
- Policy evaluation: fix a policy π, for all states s do,

  $$V_{t+1}(s) = \sum_a \pi(a \mid s) r(s,a) + \gamma \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid a, s) V_t(s)$$

- This works, meaning $V_t(s) \to J_\pi(s)$ for all states s.
- Temporal difference learning:

  $$V_{t+1}(s_t) = V_t(s_t) + \alpha_t \left(\left[r_t + \gamma V_t(s_t') \right] - V_t(s_t) \right)$$

  $V_{t+1}(s) = V_t(s)$ for all $s \neq s_t$.
- This works provided you perform the update for every state infinitely often.
- Temporal difference learning with linear approximation: given that you have a way to extract certain features $x(s)$ from state s:

  $$\theta_{t+1} = \theta_t + \alpha_t \left(\left[r_t + \gamma \theta_t^T x(s_{t+1}) \right] - \theta_t^T x(s_t) \right) x(s_t)$$

- Can prove: this works provided you generate a single infinite path.

- Let's talk about the big picture.
- Value iteration evaluation: for all states s, do

$$V_{t+1}(s) = \max_a \left(r(s,a) + \gamma \sum_{s'} P(s' \mid a, s) V_t(s) \right)$$

- This works, meaning $V_t(s) \to J^*(s)$ for all states s.
- Value iteration for Q-values: for all state-action pairs s, a do:

$$Q_{t+1}(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s, a) \max_{a'} Q_t(s', a')$$

- Logic: the Q-values under the optimal policy, i.e., $Q^*(s,a)$ are a fixed point of this.

  This works meaning $Q_{t+1}(s,a) \to Q^*(s,a)$ for all state-action pairs s, a.

- From

  $$Q_{t+1}(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) \max_{a'} Q_t(s',a')$$

  we went to

  $$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha_t \left(\left[r_t + \gamma \max_{a'} Q_t(s'_t,a') \right] - Q_t(s_t,a_t) \right)$$

  $Q_{t+1}(s\,.\,a) = Q_t(s,a)$ for all other state-action pairs.
- That was **Q-learning**. It works provided every state-action pair is selected infinitely often.
- Q-learning with linear function approximation: we assume the ability to extract a collection of features $x(s_t,a_t)$ from a state-action pair:

  $$\theta_{t+1} = \theta_t + \alpha_t \left(\left[r_t + \gamma \max_{a'} \theta_t^T x(s_{t+1},a') \right] - \theta_t^T x(s_t,a_t) \right) x(s_t,a_t)$$

- Does not always work!

This is a screenshot from TORCS, a racing car simulator
Used to be fairly popular, now AirSim and CARLA are used more commonly for driving simulations
It's a bit simplistic, but its also relatively lightweight: easy to run multiple runs
More modern simulators are based on the Unreal Engine, which creates photorealistic quality video, but is slow
You can see the video itself at https://www.youtube.com/watch?v=kAtSlJRGJhA

Learning Overtaking and Blocking Skills in Simulated Car Racing

Han-Hsien Huang, Tsaipei Wang
Department of Computer Science, National Chiao Tung University
Hsinchu City, Taiwan

***Abstract*—In this paper we describe the analysis of using Q-learning to acquire overtaking and blocking skills in simulated car racing games. Overtaking and blocking are more complicated racing skills compared to driving alone, and past work on this topic has only touched overtaking in very limited scenarios. Our work demonstrates that a driving AI agent can learn overtaking and blocking skills via machine learning, and the acquired skills are applicable when facing different opponent types and track characteristics, even on actual built-in tracks in TORCS.**

This paper focuses on the learning of overtaking and blocking skills. While these are important skills in car racing, they pose unique challenges because the driving agent needs to respond and adapt to the behaviors of other cars, at times needing to deviate from the optimal or safe route, therefore incurring extra danger of crashing. As a result, the planning and execution of these skills is significantly more complicated than attempting to drive well by oneself.

II. TORCS AND OVERTAKING/BLOCKING BEHAVIORS

This paper from 2015 asked a simple question: can we use Q-learning to teach an AI to overtake and block another car?

- The car to be overtaken is one of the standard algorithms that comes from the simulation.
- They use four variables:
  — d_y, the difference in positions between “our” car and the “other” car along the track
  — d_x, the difference in positions between the same two cars across the track
  — d_v, the speed difference
  — p_x, position of “our” car across the track [i.e., 0 is center of track, negative numbers are left of center, positive numbers are right of center]
- This is continuous. In order to get a problem with finite number of state-action pairs, they discretize this.

TABLE II. INTERVALS OF VARIABLES IN Q-LEARNING

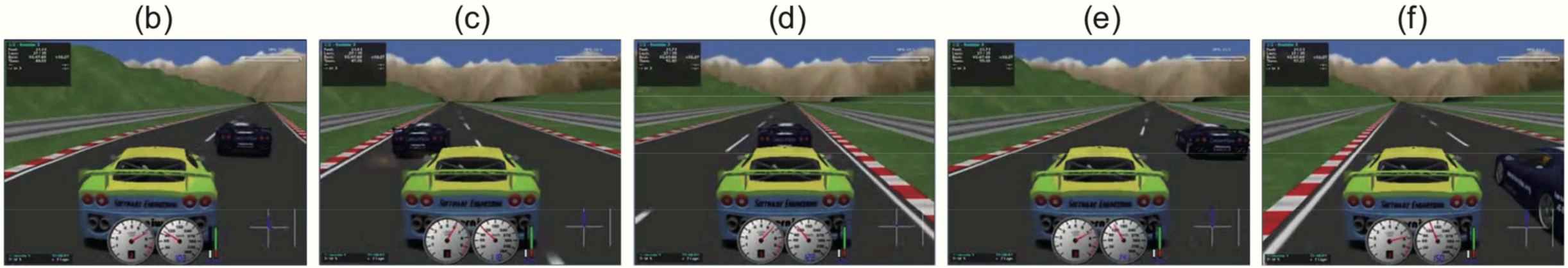
Variable	Intervals
d_y (m)	[0 5), [5 10), [10 15), [15 20), [20 25), [25 30)
d_x (m)	(-∞ -10), [-10 -5), [-5 -3), [-3 -1), [-1 0), [0 1), [1 3), [3 5), [5 10), [10 ∞)
d_v (km/h)	[0 10), [10 20), [20 30), [30 50), [50 70), [70 100), [100 150), [150 200), [200 ∞)
p_x (m)	(-∞ -3), [-3 -2), [-2 -1), [-1 0), [0 1), [1 2), [2 3), [3 ∞)

So there are 5x10x9x8 states

Actions are also discretized as: move one meter left, move one meter right, stay on current path (at same velocity)

This is it!

The algorithm is a q-learning with constant alpha., discount factor = 0.8



They report success with this method, both for overtaking and then blocking afterwards.
Compared to the things we'll discuss later in the class, this is not a super-impressive result.
Still, it's a nice illustration that RL can learn from experience.
These results, though, always require tuning:
— How are you discretizing your state space?
— Alternatively, what do you code as your features?
— What step-size do you use?
— How long should you train for?
— Will the results work on a different track? Might need to train across several tracks.