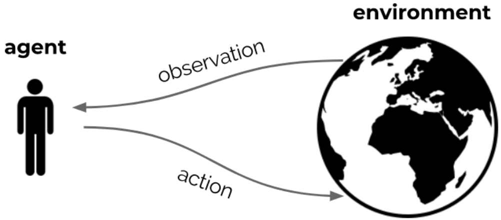
Lecture 5: Function Approximation and Deep Reinforcement Learning

Hado van Hasselt

UCL, 2018

Recap



- Reinforcement learning is the science of learning to make decisions
- Agents can learn a policy, value function and/or a model
- The general problem involves taking into account time and consequences
- Decisions affect the reward, the agent state, and environment state

Function approximation and deep reinforcement learning

- The policy, value function and model are all functions
- We want to learn (one of) these from experience
- If there are too many states, we need to approximate
- In general, this is called RL with function approximation
- When using deep neural nets, this is often called deep reinforcement learning
- The term is fairly new — the combination is decades old

Function approximation and deep reinforcement learning

This lecture

- We consider learning value functions

Next lecture

- Learn explicit policies

Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve large problems, e.g.

- Backgammon: 10^{20} states
- Go: 10^{170} states
- Helicopter: continuous state space
- Robots: informal state space (physical universe)

How can we scale up our methods for prediction and control?

Value Function Approximation

- So far we mostly considered lookup tables
  - Every state s has an entry $v(s)$
  - Or every state-action pair s, a has an entry $q(s, a)$
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
  - Individual states are often not fully observable

Value Function Approximation

- Solution for large MDPs:
  - Estimate value function with function approximation

$$v_\theta(s) \approx v_\pi(s) \qquad \text{(or } v_*(s)\text{)}$$
$$q_\theta(s, a) \approx q_\pi(s, a) \qquad \text{(or } q_*(s, a)\text{)}$$

  - Generalise from seen states to unseen states
  - Update parameter θ using MC or TD learning
- If the environment state is not fully observable:
  - Use the agent state
  - Consider learning a state update function $S_{t+1} = u(S_t, O_{t+1})$
  - Henceforth, S_t denotes the agent state

mostly in POMDP
this means i.e. Robots

Which Function Approximator?

There are many function approximators, e.g.

- Artificial neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- Coarse coding

In principle, any function approximator can be used, but RL has specific properties:

- Experience is not i.i.d. — successive time-steps are correlated
- Agent's policy affects the data it receives
- Value functions $v_\pi(s)$ can be non-stationary → Bootstrapping in TD because we update the func. we want to learn
- Feedback is delayed, not instantaneous

Classes of Function Approximation

- Tabular: a table with an entry for each MDP state
- State aggregation: Partition environment states
- Linear function approximation: fixed features (or fixed kernel)
- Differentiable (nonlinear) function approximation: neural nets

What should you choose? Depends on your goals.

- Top: good theory but weak performance
- $\vdots$
- Bottom: excellent performance but weak theory
- (Deep) neural nets often perform best (although not always)

Gradient Descent

- Let $J(\theta)$ be a differentiable function of parameter vector θ
- Define the gradient of $J(\theta)$ to be

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$$

- Goal: To find a (local) minimum of $J(\theta)$
- Method: move θ in the direction of negative gradient

$$\Delta\theta = -\frac{1}{2}\alpha \nabla_\theta J(\theta)$$

where α is a step-size parameter

Approximate Values By Stochastic Gradient Descent

- Goal: find θ that minimise the difference between $v_\theta(s)$ and $v_\pi(s)$

$$J(\theta) = \mathbb{E}_\pi \left[(v_\pi(S) - v_\theta(S))^2 \right]$$

we don't have access to this

Note: The expectation if over the state distribution — e.g., induced by the policy

- Gradient descent:

$$\Delta\theta = -\frac{1}{2}\alpha\nabla_\theta J(\theta) = \alpha\mathbb{E}_\pi \left[(v_\pi(S) - v_\theta(S))\nabla_\theta v_\theta(S) \right]$$

- Stochastic gradient descent:

$$\Delta\theta_t = \alpha(v_\pi(S_t) - v_\theta(S_t))\nabla_\theta v_\theta(S_t)$$

Feature Vectors

- Represent state by a feature vector

$$\phi(s) = \begin{pmatrix} \phi_1(s) \\ \vdots \\ \phi_n(s) \end{pmatrix}$$

- $\phi : \mathcal{S} \to \mathbb{R}^n$ is a fixed mapping from state (e.g., observation) to features
- Short-hand: $\phi_t = \phi(S_t)$
- For example:
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece and pawn configurations in chess

Linear Value Function Approximation

- Approximate value function by a linear combination of features

$$v_\theta(s) = \theta^\top \phi(s) = \sum_{j=1}^{n} \phi_j(s)\theta_j$$

- Objective function (‘loss‘) is quadratic in θ

$$J(\theta) = \mathbb{E}_\pi \left[(v_\pi(S) - \theta^\top \phi(S))^2 \right]$$

- Stochastic gradient descent converges on global optimum
- Update rule is simple

$$\nabla_\theta v_\theta(S_t) = \phi(S_t) = \phi_t \qquad \Longrightarrow \qquad \Delta\theta = \alpha(v_\pi(S_t) - v_\theta(S_t))\phi_t$$

Update $=$ step-size $\times$ prediction error $\times$ feature vector

Table Lookup Features

- Table lookup can be implemented as a special case of linear value function approximation
- Let the n states be given by $\mathcal{S} = \{s^{(1)}, \ldots, s^{(n)}\}$.
- Using table lookup features

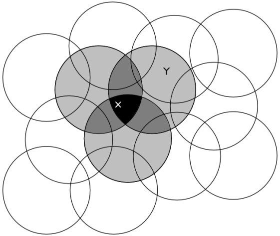
$$\phi^{table}(s) = \begin{pmatrix} \mathbf{1}(s = s^{(1)}) \\ \vdots \\ \mathbf{1}(s = s^{(n)}) \end{pmatrix}$$

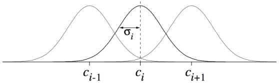
- Parameter vector θ gives value of each individual state

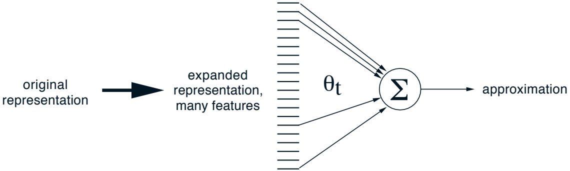
$$V(s) = \begin{pmatrix} \mathbf{1}(s = s^{(1)}) \\ \vdots \\ \mathbf{1}(s = s^{(n)}) \end{pmatrix} \cdot \begin{pmatrix} \theta_1 \\ \vdots \\ \theta_n \end{pmatrix}$$

Example: Coarse Coding

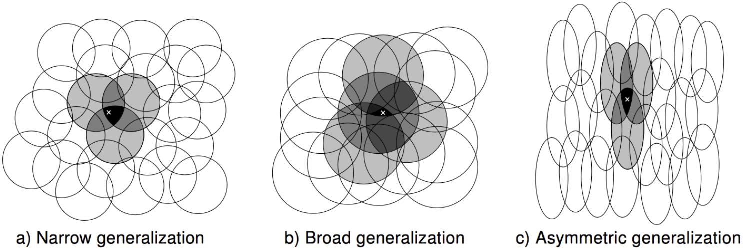
- Coarse coding provides large feature vector $\phi(s)$
- Parameter vector θ gives a value to each feature







Generalization in Coarse Coding



- Note that we will aggregate multiple states
- This means the resulting features are non-Markovian
- This is the common case when using function approximation
- Consider whether good solutions exist for given feature + function approximation

Incremental Prediction Algorithms

- The true value function $v_\pi(s)$ is typically not available
- In practice, we substitute a target for $v_\pi(s)$
  - For MC, the target is the return G_t

$$\Delta\theta_t = \alpha(G_t - v_\theta(s))\nabla_\theta v_\theta(s)$$

  - For TD, the target is the TD target $R_{t+1} + \gamma v_\theta(S_{t+1})$

$$\Delta\theta_t = \alpha(R_{t+1} + \gamma v_\theta(S_{t+1}) - v_\theta(S_t))\nabla_\theta v_\theta(S_t)$$

Monte-Carlo with Value Function Approximation

- The return G_t is an unbiased, noisy sample of $v_\pi(s)$
- Can therefore apply supervised learning to (online) "training data":

$$\{(S_0, G_0), \ldots, (S_t, G_t)\}$$

- For example, using linear Monte-Carlo policy evaluation

$$\begin{aligned}\Delta\theta_t &= \alpha(G_t - v_\theta(S_t))\nabla_\theta v_\theta(S_t)\\ &= \alpha(G_t - v_\theta(S_t))\phi_t\end{aligned}$$

- Monte-Carlo evaluation converges to a local optimum
- Even when using non-linear value function approximation
- For linear functions, it finds the global optimum

TD Learning with Value Function Approximation

- The TD-target $R_{t+1} + \gamma v_\theta(S_{t+1})$ is a biased sample of true value $v_\pi(S_t)$
- Can still apply supervised learning to "training data":

$$\{(S_0, R_1 + \gamma v_\theta(S_1)), \ldots (S_t, R_{t+1} + \gamma v_\theta(S_{t+1}))\}$$

- For example, using linear TD

$$\begin{aligned}\Delta\theta_t &= \alpha \underbrace{(R_{t+1} + \gamma v_\theta(S_{t+1}) - v_\theta(S_t))}_{= \delta_t, \text{ `TD error'}} \nabla_\theta v_\theta(S_t) \\ &= \alpha \delta_t \phi_t\end{aligned}$$

Convergence of MC

- With linear functions, MC converges to

$$\min_\theta \mathbb{E}\left[(G_t - v_\theta(S_t))^2\right] = \mathbb{E}\left[\phi_t\phi_t^\top\right]^{-1}\mathbb{E}\left[v_\pi(S_t)\phi_t\right]$$

- Proof:

$$\begin{aligned}
\nabla_\theta \mathbb{E}\left[(G_t - v_\theta(S_t))^2\right] = \mathbb{E}\left[(G_t - v_\theta(S_t))\phi_t\right] &= 0 \\
\mathbb{E}\left[(G_t - \phi_t^\top\theta)\phi_t\right] &= 0 \\
\mathbb{E}\left[G_t\phi_t - \phi_t\phi_t^\top\theta\right] &= 0 \\
\mathbb{E}\left[\phi_t\phi_t^\top\right]\theta &= \mathbb{E}\left[G_t\phi_t\right] \\
\theta &= \mathbb{E}\left[\phi_t\phi_t^\top\right]^{-1}\mathbb{E}\left[v_\pi(S_t)\phi_t\right]
\end{aligned}$$

Convergence of TD

- With linear functions, TD converges to

$$\min_\theta \mathbb{E}\left[(R_{t+1} + \gamma v_\theta(S_{t+1}) - v_\theta(S_t))^2\right] = \mathbb{E}\left[\phi_t(\phi_t - \gamma\phi_{t+1})^\top\right]^{-1} \mathbb{E}\left[R_{t+1}\phi_t\right]$$

  (in continuing problems with fixed γ)
- This is a different solution from MC
- Typically, the asymptotic MC solution is preferred
- But TD methods may converge faster, and may still be better

Residual Bellman updates

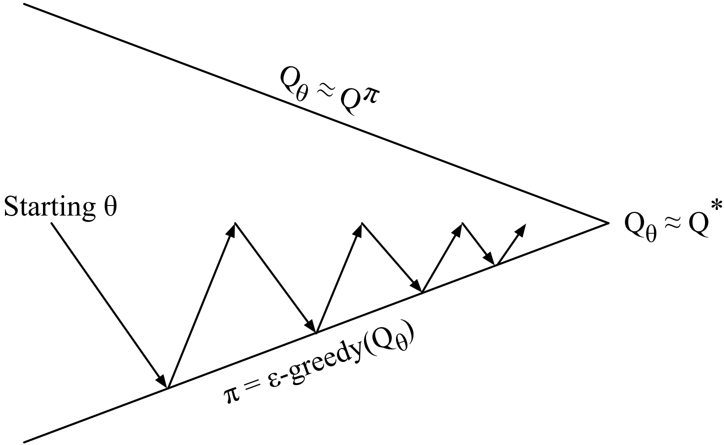
TD: $\Delta\theta_t = \alpha\delta\nabla_\theta v_\theta(S_t)$ where $\delta_t = R_{t+1} + \gamma v_\theta(S_{t+1}) - v_\theta(S_t)$

- This update ignores dependence of $v_\theta(S_{t+1})$ on θ
- Alternative: Bellman residual gradient update

loss: $\mathbb{E}\left[\delta_t^2\right]$ update: $\Delta\theta_t = \alpha\delta_t\nabla_\theta(v_\theta(S_t) - \gamma v_\theta(S_{t+1}))$

- This tends to work worse in practice
- So, in, e.g., Tensorflow, we use: $[\![R_{t+1} + \gamma v_\theta(S_{t+1})]\!] - v_\theta(S_t)$ to do TD where $[\![\cdot]\!]$ treats the argument as constant, as in `tf.stop_gradient(.)`

Control with Value Function Approximation



Policy evaluation Approximate policy evaluation, $q_\theta \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

Action-Value Function Approximation

- Approximate the action-value function

$$q_\theta(s, a) \approx q_\pi(s, a)$$

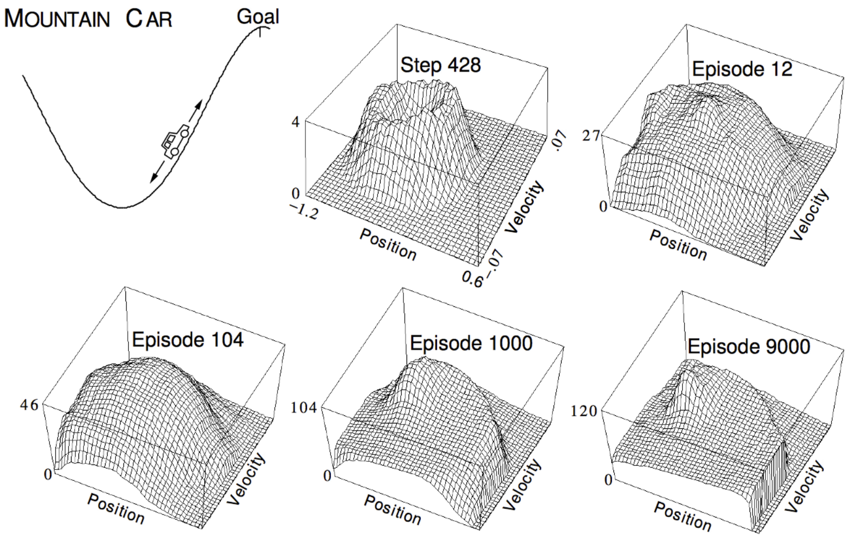
- For instance, with linear function approximation

$$q_\theta(s, a) = \phi(s, a)^\top \theta = \sum_{j=1}^{n} \phi_j(s, a)\theta_j$$

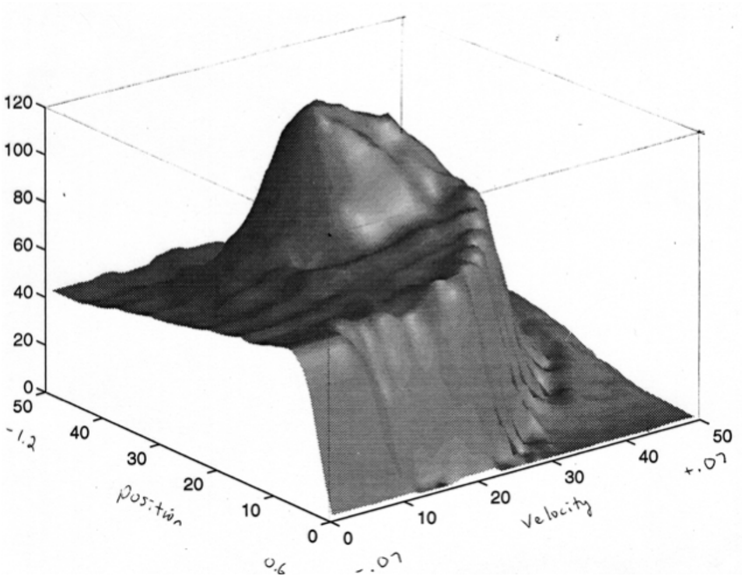
- Stochastic gradient descent update

$$\begin{aligned} \Delta\theta &= \alpha(q_\pi(s, a) - q_\theta(s, a))\nabla_\theta q_\theta(s, a) \\ &= \alpha(q_\pi(s, a) - q_\theta(s, a))\phi(s, a) \end{aligned}$$

Linear Sarsa with Coarse Coding in Mountain Car



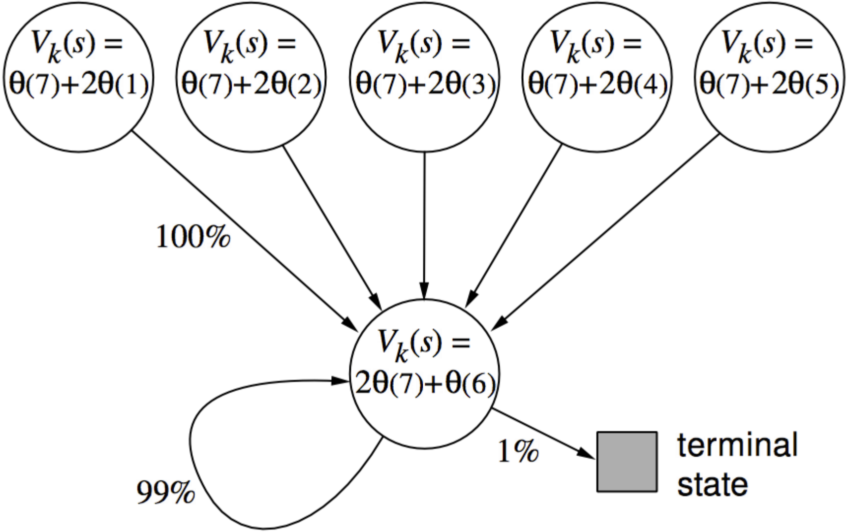
Linear Sarsa with Radial Basis Functions in Mountain Car



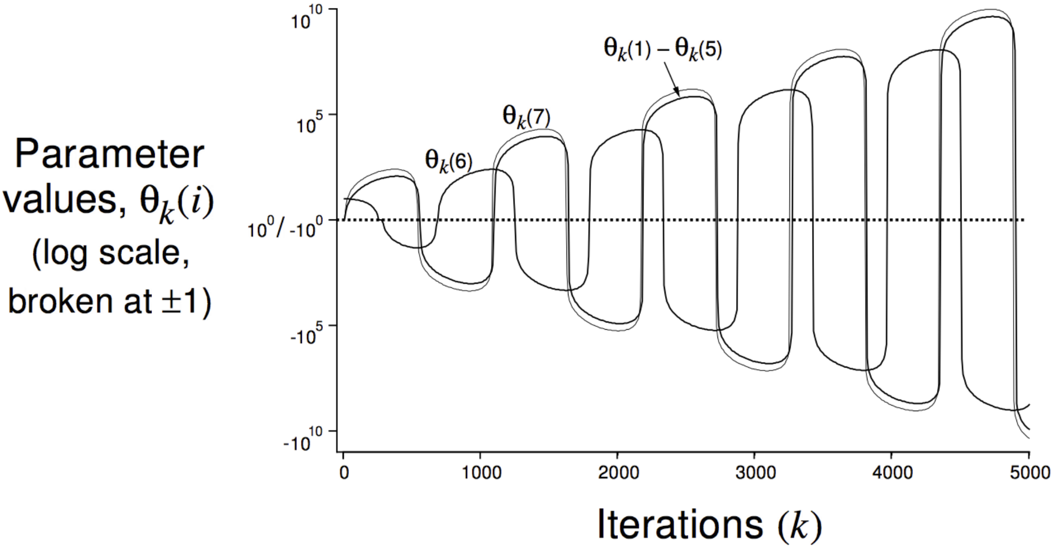
Convergence Questions

- When do incremental prediction algorithms converge?
  - When using bootstrapping (i.e. TD)?
  - When using (e.g., linear) value function approximation?
  - When using off-policy learning?
- Ideally, we would like algorithms that converge in all cases

Baird's Counterexample



Parameter Divergence in Baird's Counterexample



Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗

Convergence of Control Algorithms

- Tabular control learning algorithms (e.g., Q-learning) can be extended to FA (e.g., Deep Q Network — DQN)
- The theory of control with function approximation is not fully developed
- Tracking is often preferred to convergence
  (I.e., continually adapting the policy instead of converging to a fixed policy)

Batch Reinforcement Learning

- Gradient descent is simple and appealing
- But it is not sample efficient
- Batch methods seek to find the best fitting value function for a given a set of past experience ("training data")

Least Squares Prediction

- Given value function approximation $v_\theta(s) \approx v_\pi(s)$
- And experience $\mathcal{D}$ consisting of $\langle state, estimated\ value \rangle$ pairs

$$\mathcal{D} = \{\langle S_1, \hat{v}_1^\pi \rangle, \langle S_2, \hat{v}_2^\pi \rangle, ..., \langle S_T, \hat{v}_T^\pi \rangle\}$$

- E.g., $\hat{V}_1^\pi = R_{t+1} + \gamma v_\theta(S_{t+1})$
- Which parameters θ give the best fitting value function $v_\theta(s)$?

Stochastic Gradient Descent with Experience Replay

Given experience consisting of $\langle state, value \rangle$ pairs

Dataset

$$\mathcal{D} = \underbrace{\{\langle S_1, \hat{v}_1^\pi \rangle, \langle S_2, \hat{v}_2^\pi \rangle, ..., \langle S_T, \hat{v}_T^\pi \rangle\}}_{\text{Replay Buffer}}$$

Repeat:

1. Sample state, value from experience

$$\langle s, \hat{v}^\pi \rangle \sim \mathcal{D}$$

2. Apply stochastic gradient descent update

$$\Delta\theta = \alpha(\hat{v}^\pi - v_\theta(s))\nabla_\theta v_\theta(s)$$

Converges to least squares solution

$$\theta^\pi = \underset{\theta}{\text{argmin}}\ \text{LS}(\theta) = \underset{\theta}{\text{argmin}}\ \mathbb{E}_\mathcal{D}\left[(\hat{v}_i^\pi - v_\theta(S_i))^2\right]$$

Linear Least Squares Prediction

- Experience replay finds least squares solution
- But it may take many iterations
- Using linear value function approximation $v_\theta(s) = \phi(s)^\top \theta$
  we can solve the least squares solution directly

Linear Least Squares Prediction (2)

- At minimum of $LS(\theta)$, the expected update must be zero

$$
\begin{aligned}
\mathbb{E}_{\mathcal{D}}\left[\Delta\theta\right] &= 0 \\
\alpha \sum_{t=1}^{T} \phi_t (\hat{v}_t^{\pi} - \phi_t^{\top}\theta) &= 0 \\
\sum_{t=1}^{T} \phi_t \hat{v}_t^{\pi} &= \sum_{t=1}^{T} \phi_t \phi_t^{\top} \theta \\
\theta_t &= \left(\sum_{t=1}^{T} \phi_t \phi_t^{\top} \right)^{-1} \sum_{t=1}^{T} \phi_t \hat{v}_t^{\pi}
\end{aligned}
$$

- For N features, direct solution time is $O(N^3)$
- Incremental solution time is $O(N^2)$ using Shermann-Morrison

Linear Least Squares Prediction Algorithms

- We do not know true values v_π (have estimates $\hat{v}_t$)
- In practice, our "training data" must use noisy or biased samples of v_π

LSMC Least Squares Monte-Carlo uses return

$$v_\pi \approx G_t$$

LSTD Least Squares Temporal-Difference uses TD target

$$v_\pi \approx R_{t+1} + \gamma v_\theta(S_{t+1})$$

- In each case we can solve directly for the fixed point

Convergence of Linear Least Squares Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✓	✗
	LSTD	✓	✓	-
Off-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✗	✗
	LSTD	✓	✓	-

Deep reinforcement learning

- Many ideas immediately transfer when using deep neural networks:
  - TD and MC
  - Double learning (e.g., double Q-learning)
  - Experience replay
  - ...
- Some ideas do not easily transfer
  - UCB → MCTS? Too much generalisation?
  - Least squares TD/MC

Example: neural Q-learning

- Online neural Q-learning may include:
  - A network q_θ: $O_t \implies (q[1], \ldots, q[m])$ (m actions)
  - An ϵ-greedy exploration policy: $q_t \implies \pi_t \implies A_t$
  - A Q-learning loss function on θ

$$I(\theta) = \frac{1}{2}\left(R_{t+1} + \gamma \left[\!\left[\max_a q_\theta(S_{t+1}, a)\right]\!\right] - q_\theta(S_t, A_t)\right)^2$$

    where $[\![\cdot]\!]$ denotes stopping the gradient, so that the semi-gradient is

$$\nabla_\theta I(\theta) = \left(R_{t+1} + \gamma \max_a q_\theta(S_{t+1}, a) - q_\theta(S_t, A_t)\right) \nabla_\theta q_\theta(S_t, A_t)$$

  - An optimizer to minimize the loss (e.g., SGD, RMSProp, Adam)

Example: TF pseudo-code for Q-learning

```
# Compute Q values Q(S_t, .)
q = q_net(obs)

# Get action A_t
action = epsilon_greedy(q)

# Compute Q(S_t, A_t)
qa = q[action]

# Step in environment
reward, discount, next_obs = env.step(action)

# Get max of values at next state
max_q_next = tf.reduce_max(q_net(next_obs))

# Compute TD-error, do not to propagate into next state value
delta = reward + discount * tf.stop_gradient(max_q_next) - qa

# Define loss
q_loss = tf.square(delta)/2
```

Example: DQN → Two additional components

- DQN (Mnih et al. 2013, 2015) includes:
  - A network q_θ: $O_t \mapsto (q[1], \ldots, q[m])$ (m actions)
  - An ϵ-greedy exploration policy: $q_t \mapsto \pi_t \implies A_t$
  - A replay buffer to store and sample past transitions
  - Target network parameters θ^-
  - A Q-learning loss function on θ (uses replay and target network)

$$I(\theta) = \frac{1}{2}\left(R_{i+1} + \gamma [\![\max_a q_{\theta^-}(S_{i+1}, a)]\!] - q_\theta(S_i, A_i)\right)^2$$

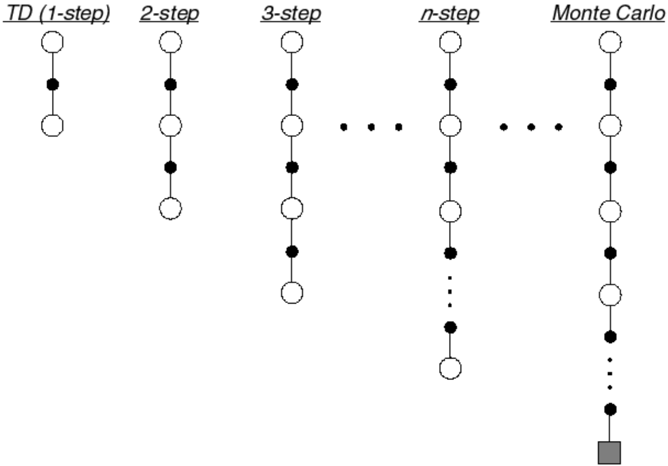
  - An optimizer to minimize the loss (e.g., SGD, RMSprop, or Adam)
  - Update $\theta_t^- \leftarrow \theta_t$ occasionally
    (e.g., every 10000 steps — on all other steps $\theta_t^- = \theta_{t-1}^-$)
- Replay and target networks make RL look more like supervised learning
- It is unclear whether they are vital, but they helped for DQN
  ↳ because target is more stationary
- "DL-aware RL"

Multi-step updates

- When we bootstrap, updates use old estimates
- Information can propagate back quite slowly
- In MC information propagates faster, but the updates are noisier
- We can go in between TD and MC

n-Step Prediction

- Let TD target look n steps into the future



n-Step Return

- Consider the following n-step returns for $n = 1, 2, \infty$:

$$\begin{array}{lll} n = 1 & (TD) & G_t^{(1)} = R_{t+1} + \gamma v(S_{t+1}) \\ n = 2 & & G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 v(S_{t+2}) \\ \vdots & & \vdots \\ n = \infty & (MC) & G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-t-1} R_T \end{array}$$

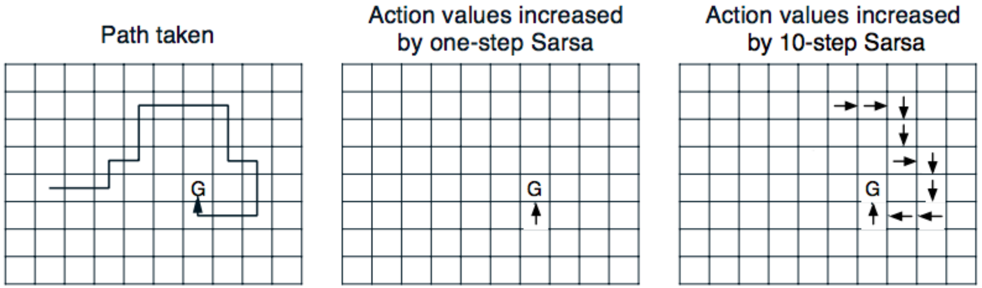
- Define the n-step return

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n v(S_{t+n})$$

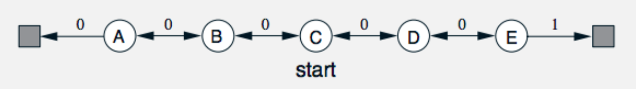
- n-step temporal-difference learning

$$v(S_t) \leftarrow v(S_t) + \alpha \left(G_t^{(n)} - v(S_t) \right)$$

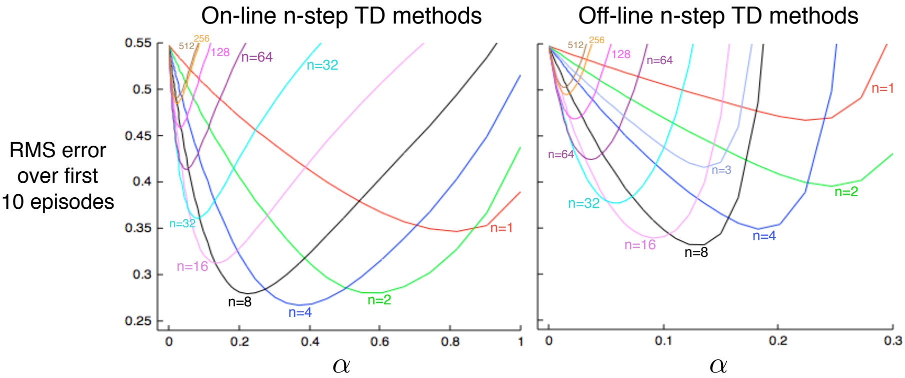
Multi-step Return



Large Random Walk Example



(but with 19 states, rather than 5)

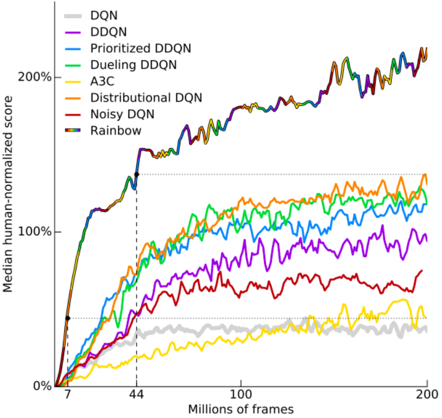


Benefits of multi-step returns

- Multi-step returns have benefits from both TD and MC
- Typically, intermediate values of n are good
- When going off-policy, can be combined with importance sampling corrections

Deep reinforcement learning research

- Deep RL is a rich and fertile research area
- Many improvements have been proposed, performance keeps improving
- Still many open questions, e.g.,
  - How best to construct agent state (including memory)?
  - How best to construct losses?
  - How best to improve data efficiency?
  - Can we understand learning dynamics better?
  - Can we learn and use models?
  - ...



Questions?