CSC 311: Introduction to Machine Learning

Lecture 11 - Reinforcement Learning

Roger Grosse Chris Maddison Juhan Bae Silviu Pitis

University of Toronto, Fall 2020

Reinforcement Learning Problem

- Recall: we categorized types of ML by how much information they provide about the desired behavior.
  - Supervised learning: labels of desired behavior
  - Unsupervised learning: no labels
  - Reinforcement learning: reward signal evaluating the outcome of past actions
- In RL, we typically focus on sequential decision making: an agent chooses a sequence of actions which each affect future possibilities available to the agent.

An agent

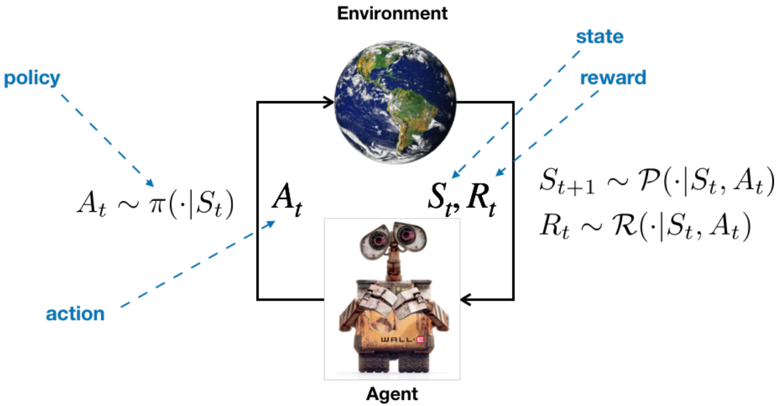
observes the world

takes an action and its states changes

with the goal of achieving long-term rewards.

Reinforcement Learning

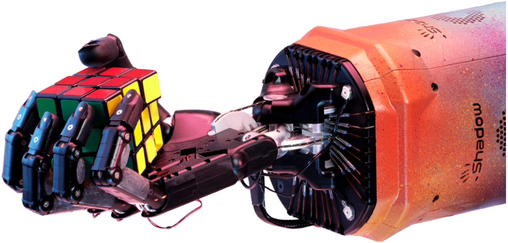
Most RL is done in a mathematical framework called a Markov Decision Process (MDP).



MDPs: States and Actions

- First let's see how to describe the dynamics of the environment.
- The state is a description of the environment in sufficient detail to determine its evolution.
  - Think of Newtonian physics.
  - Markov assumption: the state at time $t + 1$ depends directly on the state and action at time t, but not on past states and actions.
- To describe the dynamics, we need to specify the transition probabilities $\mathcal{P}(S_{t+1} \mid S_t, A_t)$.
- In this lecture, we assume the state is fully observable, a highly nontrivial assumption.

MDPs: States and Actions



- Suppose you're controlling a robot hand. What should be the set of states and actions?
  - states = sensor measurements, actions = actuator voltages?
  - states = joint positions and velocities, actions = trajectory keypoints?
- In general, the right granularity of states and actions depends on what you're trying to achieve.

MDPs: Policies

- The way the agent chooses the action in each step is called a policy.
- We'll consider two types:
  - Deterministic policy: $A_t = \pi(S_t)$ for some function $\pi : \mathcal{S} \to \mathcal{A}$
  - Stochastic policy: $A_t \sim \pi(\cdot \mid S_t)$ for some function $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$. (Here, $\mathcal{P}(\mathcal{A})$ is the set of distributions over actions.)
- With stochastic policies, the distribution over rollouts, or trajectories, factorizes:

$$p(s_1, a_1, \ldots, s_T, a_T) = p(s_1)\, \pi(a_1 \mid s_1)\, \mathcal{P}(s_2 \mid s_1, a_1)\, \pi(a_2 \mid s_2) \cdots \mathcal{P}(s_T \mid s_{T-1}, a_{T-1})\, \pi(a_T$$

- **Note:** the fact that policies need consider only the current state is a powerful consequence of the Markov assumption and full observability.
  - If the environment is partially observable, then the policy needs to depend on the history of observations.

MDPs: Rewards

- In each time step, the agent receives a reward from a distribution that depends on the current state and action

$$R_t \sim \mathcal{R}(\cdot \mid S_t, A_t)$$

- For simplicity, we'll assume rewards are deterministic, i.e.

$$R_t = r(S_t, A_t)$$

- What's an example where R_t should depend on A_t?
- The return determines how good was the outcome of an episode.
    - Undiscounted: $G = R_0 + R_1 + R_2 + \cdots$
    - Discounted: $G = R_0 + \gamma R_1 + \gamma^2 R_2$
- The goal is to maximize the expected return, $\mathbb{E}[G]$.
- γ is a hyperparameter called the discount factor which determines how much we care about rewards now vs. rewards later.
    - What is the effect of large or small γ?

MDPs: Rewards

- How might you define a reward function for an agent learning to play a video game?
  - Change in score (why not current score?)
  - Some measure of novelty (this is sufficient for most Atari games!)
- Consider two possible reward functions for the game of Go. How do you think the agent's play will differ depending on the choice?
  - **Option 1:** +1 for win, 0 for tie, -1 for loss
  - **Option 2:** Agent's territory minus opponent's territory (at end)
- Specifying a good reward function can be tricky.
  `https://www.youtube.com/watch?v=tlOIHko8ySg`

Markov Decision Processes

- Putting this together, a Markov Decision Process (MDP) is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$.
  - $\mathcal{S}$: State space. Discrete or continuous
  - $\mathcal{A}$: Action space. Here we consider finite action space, i.e., $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$.
  - $\mathcal{P}$: Transition probability
  - $\mathcal{R}$: Immediate reward distribution
  - γ: Discount factor $(0 \leq \gamma < 1)$
- Together these define the environment that the agent operates in, and the objectives it is supposed to achieve.

Finding a Policy

- Now that we've defined MDPs, let's see how to find a policy that achieves a high return.
- We can distinguish two situations:
  - Planning: given a fully specified MDP.
  - Learning: agent interacts with an environment with unknown dynamics.
    - I.e., the environment is a black box that takes in actions and outputs states and rewards.
- Which framework would be most appropriate for chess? Super Mario?

Value Functions

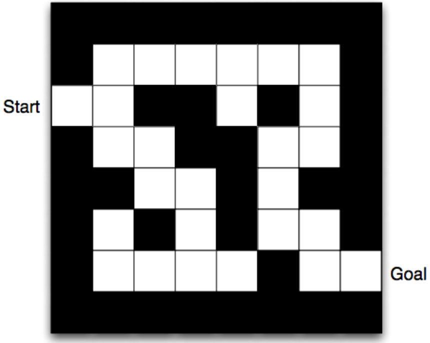
Value Function

- The value function V^π for a policy π measures the expected return if you start in state s and follow policy π.

$$V^\pi(s) \triangleq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid S_t = s\right].$$

- This measures the desirability of state s.

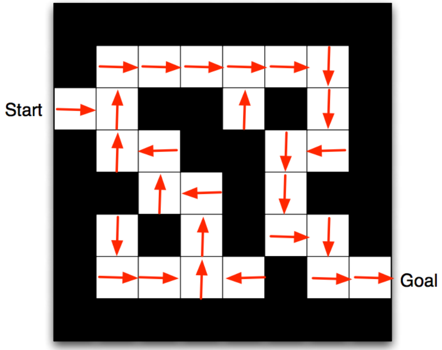
Value Function



- Rewards: -1 per time-step
- Actions: N, E, S, W
- States: Agent's location

[Slide credit: D. Silver]

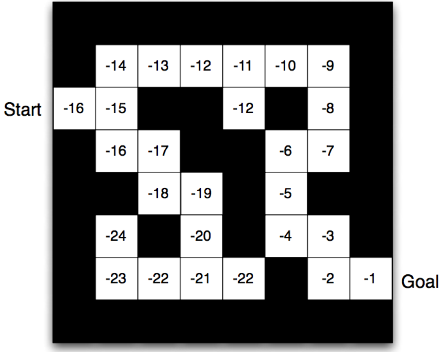
Value Function



- Arrows represent policy $\pi(s)$ for each state s

[Slide credit: D. Silver]

Value Function



- Numbers represent value $V^{\pi}(s)$ of each state s

[Slide credit: D. Silver]

Bellman equations

- The foundation of many RL algorithms is the fact that value functions satisfy a recursive relationship, called the Bellman equation:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t \,|\, S_t = s] \\ &= \mathbb{E}_\pi[R_t + \gamma G_{t+1} \,|\, S_t = s] \\ &= \sum_a \pi(a \,|\, s) \left[r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \,|\, a, s) \, \mathbb{E}_\pi[G_{t+1} \,|\, S_{t+1} = s'] \right] \\ &= \sum_a \pi(a \,|\, s) \left[r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \,|\, a, s) \, V^\pi(s') \right] \end{aligned}$$

- Viewing V^π as a vector (where entries correspond to states), define the Bellman backup operator T^π.

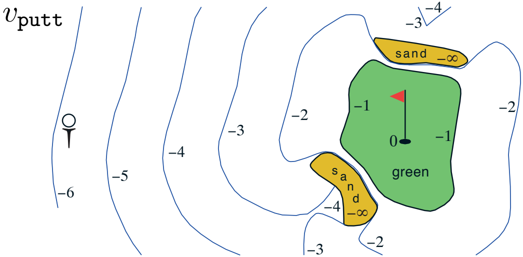
$$(T^\pi V)(s) \triangleq \sum_a \pi(a \,|\, s) \left[r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \,|\, a, s) \, V(s') \right]$$

- The Bellman equation can be seen as a fixed point of the Bellman operator:

$$T^\pi V^\pi = V^\pi.$$

Value Function

A value function for golf:



— Sutton and Barto, *Reinforcement Learning: An Introduction*

State-Action Value Function

- A closely related but usefully different function is the state-action value function, or Q-function, Q^π for policy π, defined as:

$$Q^\pi(s,a) \triangleq \mathbb{E}_\pi \left[\sum_{k \geq 0} \gamma^k R_{t+k} \mid S_t = s, A_t = a \right].$$

- If you knew Q^π, how would you obtain V^π?

$$V^\pi(s) = \sum_a \pi(a \mid s)\, Q^\pi(s,a).$$

- If you knew V^π, how would you obtain Q^π?
  - Apply a Bellman-like equation:

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \mid a, s)\, V^\pi(s')$$

  - This requires knowing the dynamics, so in general it's not easy to recover Q^π from V^π.

State-Action Value Function

- Q^π satisfies a Bellman equation very similar to V^π (proof is analogous):

$$Q^\pi(s,a) = \underbrace{r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \mid a, s) \sum_{a'} \pi(a' \mid s') Q^\pi(s', a')}_{\triangleq (T^\pi Q^\pi)(s,a)}$$

Dynamic Programming and Value Iteration

Optimal State-Action Value Function

- Suppose you're in state s. You get to pick one action a, and then follow (fixed) policy π from then on. What do you pick?

$$\arg\max_a Q^\pi(s, a)$$

- If a deterministic policy π is optimal, then it must be the case that for any state s:

$$\pi(s) = \arg\max_a Q^\pi(s, a),$$

otherwise you could improve the policy by changing $\pi(s)$. (see Sutton & Barto for a proper proof)

Optimal State-Action Value Function

- Bellman equation for optimal policy π^*:

$$\begin{aligned} Q^{\pi^*}(s,a) &= r(s,a) + \gamma \sum_{s'} \mathcal{P}(s', \mid s,a) Q^{\pi^\star}(s', \pi^\star(s')) \\ &= r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_{a'} Q^{\pi^\star}(s',a') \end{aligned}$$

- Now $Q^* = Q^{\pi^\star}$ is the optimal state-action value function, and we can rewrite the optimal Bellman equation without mentioning $\pi^\star$:

$$Q^*(s,a) = \underbrace{r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_{a'} Q^*(s',a')}_{\triangleq (T^* Q^*)(s,a)}$$

- Turns out this is *sufficient* to characterize the optimal policy. So we simply need to solve the fixed point equation $T^* Q^* = Q^*$, and then we can choose $\pi^*(s) = \arg\max_a Q^*(s,a)$.

Bellman Fixed Points

- **So far:** showed that some interesting problems could be reduced to finding fixed points of Bellman backup operators:
  - Evaluating a fixed policy π

$$T^\pi Q^\pi = Q^\pi$$

  - Finding the optimal policy

$$T^* Q^* = Q^*$$

- **Idea:** keep iterating the backup operator over and over again.

$$\begin{aligned} Q &\leftarrow T^\pi Q && \text{(policy evaluation)} \\ Q &\leftarrow T^* Q && \text{(finding the optimal policy)} \end{aligned}$$

  - We're treating Q^π or Q^* as a vector with $|\mathcal{S}| \cdot |\mathcal{A}|$ entries.
  - This type of algorithm is an instance of dynamic programming.

Bellman Fixed Points

- An operator f (mapping from vectors to vectors) is a contraction map if
$$\|f(\mathbf{x}_1) - f(\mathbf{x}_2)\| \leq \alpha \|\mathbf{x}_1 - \mathbf{x}_2\|$$
for some scalar $0 \leq \alpha < 1$ and vector norm $\|\cdot\|$.
- Let $f^{(k)}$ denote f iterated k times. A simple induction shows
$$\|f^{(k)}(\mathbf{x}_1) - f^{(k)}(\mathbf{x}_2)\| \leq \alpha^k \|\mathbf{x}_1 - \mathbf{x}_2\|.$$
- Let $\mathbf{x}^*$ be a fixed point of f. Then for any $\mathbf{x}$,
$$\|f^{(k)}(\mathbf{x}) - \mathbf{x}^*\| \leq \alpha^k \|\mathbf{x} - \mathbf{x}_*\|.$$
- Hence, iterated application of f, starting from any $\mathbf{x}$, converges exponentially to a unique fixed point.

Finding the Optimal Value Function: Value Iteration

- Let's use dynamic programming to find Q^*.
- Value Iteration: Start from an initial function Q_1. For each $k = 1, 2, \dots,$ apply

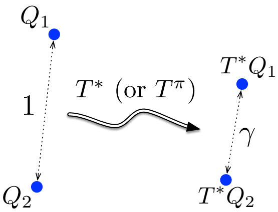
$$Q_{k+1} \leftarrow T^* Q_k$$

- Writing out the update in full,

$$Q_{k+1}(s,a) \leftarrow r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \max_{a' \in \mathcal{A}} Q_k(s',a')$$

- Observe: a fixed point of this update is exactly a solution of the optimal Bellman equation, which we saw characterizes the Q-function of an optimal policy.

Value Iteration



- **Claim:** The value iteration update is a contraction map:

$$\|T^*Q_1 - T^*Q_2\|_\infty \leq \gamma \|Q_1 - Q_2\|_\infty$$

- $\|\cdot\|_\infty$ denotes the L^∞ norm, defined as:

$$\|\mathbf{x}\|_\infty = \max_i |x_i|$$

- If this claim is correct, then value iteration converges exponentially to the unique fixed point.
- The exponential decay factor is γ (the discount factor), which means longer term planning is harder.

Bellman Operator is a Contraction

$$
\begin{aligned}
|(T^*Q_1)(s,a) - (T^*Q_2)(s,a)| &= \left| \left[r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \mid s,a) \max_{a'} Q_1(s',a') \right] - \right. \\
&\qquad \left. \left[r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \mid s,a) \max_{a'} Q_2(s',a') \right] \right| \\
&= \gamma \left| \sum_{s'} \mathcal{P}(s' \mid s,a) \left[\max_{a'} Q_1(s',a') - \max_{a'} Q_2(s',a') \right] \right| \\
&\leq \gamma \sum_{s'} \mathcal{P}(s' \mid s,a) \max_{a'} \left| Q_1(s',a') - Q_2(s',a') \right| \\
&\leq \gamma \max_{s',a'} \left| Q_1(s',a') - Q_2(s',a') \right| \sum_{s'} \mathcal{P}(s' \mid s,a) \\
&= \gamma \max_{s',a'} \left| Q_1(s',a') - Q_2(s',a') \right| \\
&= \gamma \left\| Q_1 - Q_2 \right\|_\infty
\end{aligned}
$$

- This is true for *any* (s,a), so

$$
\left\| T^*Q_1 - T^*Q_2 \right\|_\infty \leq \gamma \left\| Q_1 - Q_2 \right\|_\infty ,
$$

which is what we wanted to show.

Value Iteration Recap

- So far, we've focused on **planning**, where the dynamics are known.
- The optimal Q-function is characterized in terms of a Bellman fixed point update.
- Since the Bellman operator is a contraction map, we can just keep applying it repeatedly, and we'll converge to a unique fixed point.
- What are the limitations of value iteration?
  - assumes known dynamics
  - requires explicitly representing Q^* as a vector
    - $|\mathcal{S}|$ can be extremely large, or infinite
    - $|\mathcal{A}|$ can be infinite (e.g. continuous voltages in robotics)
- But value iteration is still a foundation for a lot of more practical RL algorithms.

Towards Learning

- Now let's focus on **reinforcement learning**, where the environment is unknown. How can we apply learning?
    1. Learn a model of the environment, and do planning in the model (i.e. model-based reinforcement learning)
        - You already know how to do this in principle, but it's very hard to get to work. Not covered in this course.
    2. Learn a value function (e.g. Q-learning, covered in this lecture)
    3. Learn a policy directly (e.g. policy gradient, not covered in this course)
- How can we deal with extremely large state spaces?
    - Function approximation: choose a parametric form for the policy and/or value function (e.g. linear in features, neural net, etc.)

Q-Learning

Monte Carlo Estimation

- Recall the optimal Bellman equation:

$$Q^*(s,a) = r(s,a) + \gamma \mathbb{E}_{\mathcal{P}(s' \mid s,a)} \left[\max_{a'} Q^*(s',a') \right]$$

- **Problem:** we need to know the dynamics to evaluate the expectation
- Monte Carlo estimation of an expectation $\mu = \mathbb{E}[X]$: repeatedly sample X and update

$$\mu \leftarrow \mu + \alpha(X - \mu)$$

- **Idea:** Apply Monte Carlo estimation to the Bellman equation by sampling $S' \sim \mathcal{P}(\cdot \mid s,a)$ and updating:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Big[\underbrace{r(s,a) + \gamma \max_{a'} Q(S',a') - Q(s,a)}_{=\text{ Bellman error}} \Big]$$

- This is an example of temporal difference learning, i.e. updating our predictions to match our later predictions (once we have more information).

Monte Carlo Estimation

- **Problem:** Every iteration of value iteration requires updating Q for every state.
  - There could be lots of states
  - We only observe transitions for states that are visited
- **Idea:** Have the agent interact with the environment, and only update Q for the states that are actually visited.
- **Problem:** We might never visit certain states if they don't look promising, so we'll never learn about them.
- **Idea:** Have the agent sometimes take random actions so that it eventually visits every state.
  - ε-greedy policy: a policy which picks $\arg\max_a Q(s, a)$ with probability $1 - \varepsilon$ and a random action with probability ε. (Typical value: $\varepsilon = 0.05$)
- Combining all three ideas gives an algorithm called Q-learning.

Q-Learning with ε-Greedy Policy

- Parameters:
  - Learning rate α
  - Exploration parameter ε
- Initialize $Q(s,a)$ for all $(s,a) \in \mathcal{S} \times \mathcal{A}$
- The agent starts at state S_0.
- For time step $t = 0, 1, ...,$
  - Choose A_t according to the ε-greedy policy, i.e.,

$$A_t \leftarrow \begin{cases} \text{argmax}_{a \in \mathcal{A}} Q(S_t, a) & \text{with probability } 1 - \varepsilon \\ \text{Uniformly random action in } \mathcal{A} & \text{with probability } \varepsilon \end{cases}$$

  - Take action A_t in the environment.
  - The state changes from S_t to $S_{t+1} \sim \mathcal{P}(\cdot | S_t, A_t)$
  - Observe S_{t+1} and R_t (could be $r(S_t, A_t)$, or could be stochastic)
  - Update the action-value function at state-action (S_t, A_t):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_t + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a') - Q(S_t, A_t) \right]$$

Exploration vs. Exploitation

- The ε-greedy is a simple mechanism for managing the exploration-exploitation tradeoff.

$$\pi_\varepsilon(S;Q) = \begin{cases} \operatorname{argmax}_{a\in\mathcal{A}} Q(S,a) & \text{with probability } 1-\varepsilon \\ \text{Uniformly random action in } \mathcal{A} & \text{with probability } \varepsilon \end{cases}$$

- The ε-greedy policy ensures that most of the time (probability $1-\varepsilon$) the agent exploits its incomplete knowledge of the world by chooses the best action (i.e., corresponding to the highest action-value), but occasionally (probability ε) it explores other actions.
- Without exploration, the agent may never find some good actions.
- The ε-greedy is one of the simplest, but widely used, methods for trading-off exploration and exploitation. Exploration-exploitation tradeoff is an important topic of research.

Examples of Exploration-Exploitation in the Real World

- Restaurant Selection
  - Exploitation: Go to your favourite restaurant
  - Exploration: Try a new restaurant
- Online Banner Advertisements
  - Exploitation: Show the most successful advert
  - Exploration: Show a different advert
- Oil Drilling
  - Exploitation: Drill at the best known location
  - Exploration: Drill at a new location
- Game Playing
  - Exploitation: Play the move you believe is best
  - Exploration: Play an experimental move

[Slide credit: D. Silver]

An Intuition on Why Q-Learning Works? (Optional)

- Consider a tuple (S, A, R, S'). The Q-learning update is

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_{a' \in \mathcal{A}} Q(S', a') - Q(S, A) \right].$$

- To understand this better, let us focus on its stochastic equilibrium, i.e., where the expected change in $Q(S, A)$ is zero. We have

$$\begin{aligned} &\mathbb{E} \left[R + \gamma \max_{a' \in \mathcal{A}} Q(S', a') - Q(S, A) | S, A \right] = 0 \\ \Rightarrow &(T^* Q)(S, A) = Q(S, A) \end{aligned}$$

- So at the stochastic equilibrium, we have $(T^* Q)(S, A) = Q(S, A)$. Because the fixed-point of the Bellman optimality operator is unique (and is Q^*), Q is the same as the optimal action-value function Q^*.

Off-Policy Learning

- Q-learning update again:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_{a' \in \mathcal{A}} Q(S', a') - Q(S, A) \right].$$

- **Notice:** this update doesn't mention the policy anywhere. The only thing the policy is used for is to determine which states are visited.
- This means we can follow whatever policy we want (e.g. ε-greedy), and it still coverges to the optimal Q-function. Algorithms like this are known as off-policy algorithms, and this is an extremely useful property.
- Policy gradient (another popular RL algorithm, not covered in this course) is an on-policy algorithm. Encouraging exploration is much harder in that case.

Function Approximation

Function Approximation

- So far, we've been assuming a tabular representation of Q: one entry for every state/action pair.
- This is impractical to store for all but the simplest problems, and doesn't share structure between related states.
- **Solution:** approximate Q using a parameterized function, e.g.
  - linear function approximation: $Q(\mathbf{s}, \mathbf{a}) = \mathbf{w}^\top \psi(\mathbf{s}, \mathbf{a})$
  - compute Q with a neural net
- Update Q using backprop:

$$t \leftarrow r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{t+1}, \mathbf{a})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(t - Q(\mathbf{s}, \mathbf{a})) \nabla_{\boldsymbol{\theta}} Q(\mathbf{s}_t, \mathbf{a}_t).$$

Function Approximation (optional)

- It's tempting to think of Q-learning with function approximation as minimizing the squared norm of the Bellman errors:

$$\mathcal{J}(\boldsymbol{\theta}) = \mathbb{E}_{S,A}\left[\left(r(S,A) + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(S',a') - Q_{\boldsymbol{\theta}}(S,A)\right)^2\right]$$

- Why isn't this interpretation correct?
  - The expectation depends on $\boldsymbol{\theta}$, so the gradient $\nabla \mathcal{J}(\boldsymbol{\theta})$ would need to account for that.
  - In addition to updating $Q_{\boldsymbol{\theta}}(S,A)$ to better match $r(s,a) + \gamma Q_{\boldsymbol{\theta}}(S',a')$, gradient descent would update $Q_{\boldsymbol{\theta}}(S',a')$ to better match $\gamma^{-1}(Q_{\boldsymbol{\theta}}(S,A) - r(S,A))$. This makes no sense, since $r(S,A) + Q_{\boldsymbol{\theta}}(S',a')$ is a better estimate of the return.
- Q-learning with function approximation is chasing a "moving target", and one can show it isn't gradient descent on any cost function. The dynamics are hard to analyze.
- Still, we use it since we don't have any good alternatives.

Function Approximation

- Approximating Q with a neural net is a decades-old idea, but DeepMind got it to work really well on Atari games in 2013 ("deep Q-learning")
- They used a very small network by today's standards

  1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
  2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
  3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
  4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
  5. update ϕ': copy ϕ every N steps

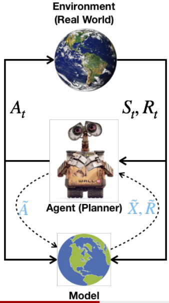
- Main technical innovation: store experience into a replay buffer, and perform Q-learning using stored experience
  - Gains sample efficiency by separating environment interaction from optimization — don't need new experience for every SGD update!

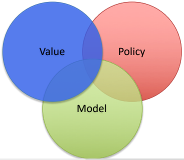
Atari

- Mnih et al., *Nature* 2015. Human-level control through deep reinforcement learning
- Network was given raw pixels as observations
- Same architecture shared between all games
- Assume fully observable environment, even though that's not the case
- After about a day of training on a particular game, often beat "human-level" performance (number of points within 5 minutes of play)
  - Did very well on reactive games, poorly on ones that require planning (e.g. Montezuma's Revenge)
- `https://www.youtube.com/watch?v=V1eYniJ0Rnk`
- `https://www.youtube.com/watch?v=4MlZncshy1Q`

Recap and Other Approaches

- All discussed approaches estimate the value function first. They are called value-based methods.
- There are methods that directly optimize the policy, i.e., policy search methods.
- Model-based RL methods estimate the true, but unknown, model of environment $\mathcal{P}$ by an estimate $\hat{\mathcal{P}}$, and use the estimate $\mathcal{P}$ in order to plan.
- There are hybrid methods.





Reinforcement Learning Resources

- Books:
  - Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, 2nd edition, 2018.
  - Csaba Szepesvari, Algorithms for Reinforcement Learning, 2010.
  - Lucian Busoniu, Robert Babuska, Bart De Schutter, and Damien Ernst, Reinforcement Learning and Dynamic Programming Using Function Approximators, 2010.
  - Dimitri P. Bertsekas and John N. Tsitsiklis, Neuro-Dynamic Programming, 1996.
- Courses:
  - Video lectures by David Silver
  - CIFAR and Vector Institute's Reinforcement Learning Summer School, 2018.
  - Deep Reinforcement Learning, CS 294-112 at UC Berkeley