Sequential Decision Making and Reinforcement Learning

(SDMRL)

Model-Based RL

Sarah Keren

The Taub Faculty of Computer Science
Technion - Israel Institute of Technology

Acknowledgments

- David Sliver's course on RL: *https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silve*
- Slides by Malte Helmert, Carmel Domshlak, Erez Karpas and Alexander Shleyfman.

Approximation

Acknowledgements

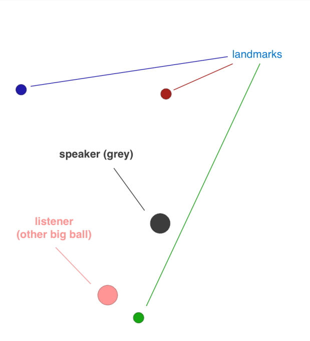
- David Sliver's course on RL: *https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silve*
- Slides by Malte Helmert, Carmel Domshlak, Erez Karpas and Alexander Shleyfman.

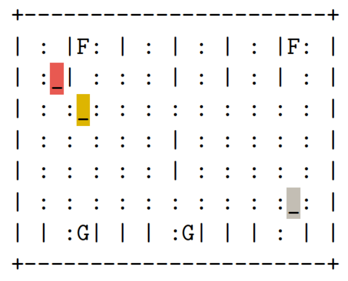
Revision:

- Methods discussed so far
  - Model-free RL
    - Monte-Carlo
    - Temporal Difference (TD) (e.g., Q-learning)
  - Model-based (e.g. Adaptive Dynamic Programming (ADP)) - soon
- All converge to optimal policy assuming a GLIE exploration strategy.
- All methods (implicitly) assume
  - the world is not too dangerous (no cliffs to fall off during exploration)
  - small state spaces

How to deal with complex tasks & high-dimensional state spaces ?

Examples





Limitations of methods seen so far ? Ideas ?

Large State Spaces

- So far we have represented value function by a lookup table: Every state s has an entry $\mathcal{V}(s)$ or every state-action pair s, a has an entry $Q(s, a)$.
- When a problem has a large state space we can no longer represent $\mathcal{V}$ and Q (or the transition and reward functions) as explicit tables.
- Even if we had enough memory
  - never enough training data
  - learning takes too long

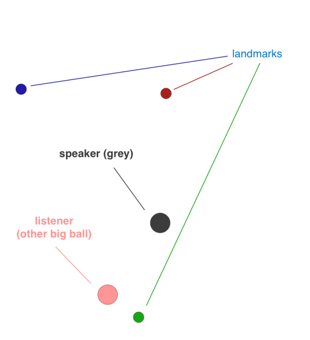
What to do?

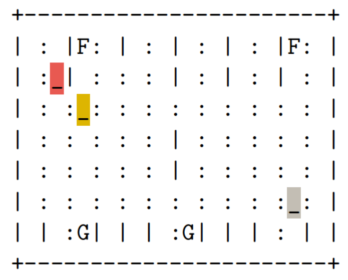
Function Approximation

- Never enough training data!
  - Must **generalize** what is learned from one situation to other "similar" new situations
- **Idea:** Instead of using large tables to represent $\mathcal{V}$ and Q, use a **parameterized function**
  - The number of parameters should be small compared to number of states (generally exponentially fewer parameters)
- Learn parameters from experience
- When we update the parameters based on observations in one state, then our $\mathcal{V}$ and Q estimates will also change for other similar states

Parameterization facilitates generalization of experience!

Back to the Examples





Parameterization ?

Value Function Parameterization

- Estimate value function with function approximation

$$\tilde{\mathcal{V}}(s, \theta) \approx \mathcal{V}_\pi(s)$$

  or

$$\tilde{Q}(s, a, \theta) \approx Q_\pi(s, a)$$

- Generalise from seen states to unseen states
- Update parameter θ using MC or TD learning.

Types of Value Function Approximation

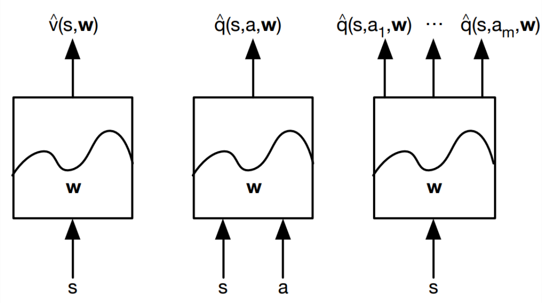


Image by David Silver

Which Function Approximator?

Which Function Approximator?

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Which Function Approximator?

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

We consider differentiable function approximators
Furthermore, we require a training method that is suitable for non-stationary, non-iid data

Which Function Approximator?

- Linear combinations of features*
- Neural network*
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

We consider differentiable function approximators
Furthermore, we require a training method that is suitable for non-stationary, non-iid data

Linear Function Approximation

- A common approximation is to represent $\mathcal{V}(s)$ as a weighted sum of the features (linear approximation)

$$\mathcal{V}_\theta(s) = \theta_0 + \theta_1 f_1(s) + \cdots + \theta_n f_n(s)$$

- The approximation accuracy is fundamentally limited by the information provided by the features

Feature Vectors

- Define a set of state features $f_1(s), \ldots, f_n(s)$
- State represented by a *feature vector*

$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

- For example:
    - Distance of robot from landmarks
    - Trends in the stock market
    - Piece and pawn configurations in chess

Can we always define features that allow for a perfect value function approximation?

Feature Vectors

- Define a set of state features $f_1(s), \ldots, f_n(s)$
- State represented by a *feature vector*

$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

- For example:
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece and pawn configurations in chess

Can we always define features that allow for a perfect value function approximation?

Yes. but...

Table Lookup Features

- Assign each state an indicator feature.
- Using table lookup features

$$x(s) = \begin{pmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{pmatrix}$$

- Parameter vector θ gives value of each individual state.

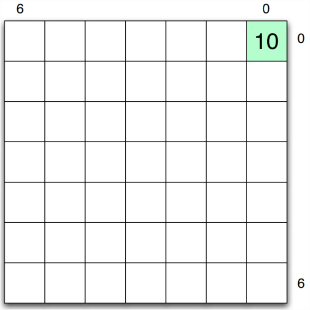
$$\tilde{\mathcal{V}}(s, \theta) = \begin{pmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{pmatrix} \cdot \begin{pmatrix} \theta_1 \\ \vdots \\ \theta_n \end{pmatrix}$$

limitations? This requires far to many features and gives no generalization.

Example

Grid with no obstacles, deterministic actions Up-Down-Left-Right, no discounting, -1 reward everywhere except +10 at goal.

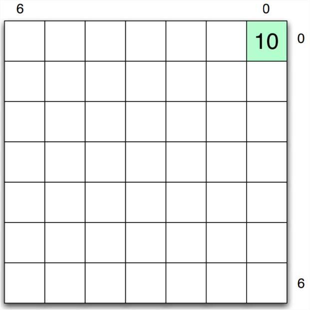
- Features for $s = (x, y)$: $f_1(s) = x$, $f_2(s) = y$
- Parameterized representation of value function ?



Example

Grid with no obstacles, deterministic actions Up-Down-Left-Right, no discounting, -1 reward everywhere except +10 at goal.

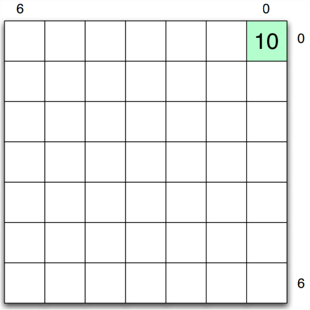
- Features for $s = (x, y)$: $f_1(s) = x$, $f_2(s) = y$
- Parameterized representation of value function ?
  - $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 x + \theta_2 y$



Example

Grid with no obstacles, deterministic actions Up-Down-Left-Right, no discounting, -1 reward everywhere except +10 at goal.

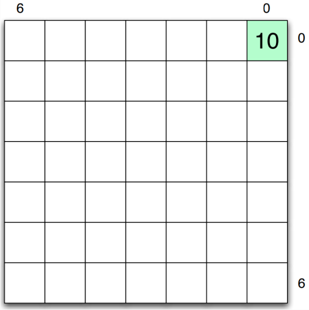
- Features for $s = (x, y)$: $f_1(s) = x$, $f_2(s) = y$
- Parameterized representation of value function ?
  - $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?



Example

Grid with no obstacles, deterministic actions Up-Down-Left-Right, no discounting, -1 reward everywhere except +10 at goal.

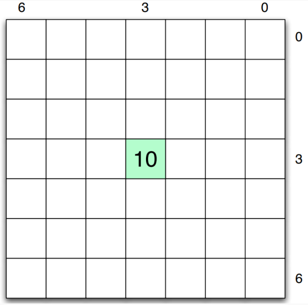
- Features for $s = (x, y)$:
  $f_1(s) = x$, $f_2(s) = y$
- Parameterized representation of value function ?
  - $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?
  - Yes.
  - $\theta_0 = 10$, $\theta_1 = \theta_2 = -1$
  - note: upper right is origin
  - $\mathcal{V}_\theta(s) = 10 - x - y$ (subtracts Manhattan distance from goal reward)



What if we change the reward function ?

Grid with no obstacles, deterministic actions Up-Down-Left-Right, no discounting, -1 reward everywhere except +10 at goal.

- Features for $s = (x, y)$: $f_1(s) = x$, $f_2(s) = y$
- $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?

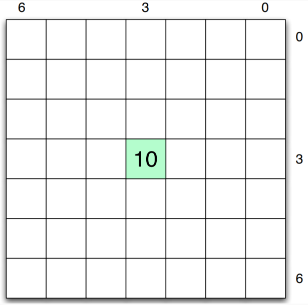


What if we change the reward function ?

Grid with no obstacles, deterministic actions Up-Down-Left-Right, no discounting, -1 reward everywhere except +10 at goal.

- Features for $s = (x, y)$: $f_1(s) = x$, $f_2(s) = y$
- $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?
- No!

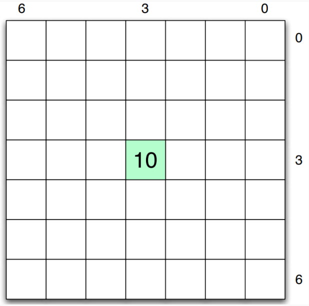
Suggestions?



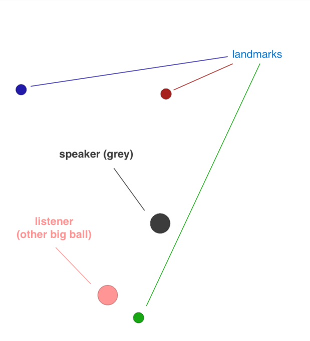
But What If..

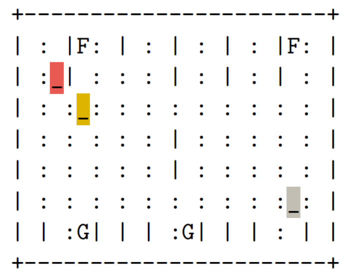
Grid with no obstacles, deterministic actions Up-Down-Left-Right, no discounting, -1 reward everywhere except +10 at goal.

- Features for $s = (x, y)$:
  $f_1(s) = x$, $f_2(s) = y$
  $f_3(s) = |3 - x| + |3 - y|$
- $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 x + \theta_2 y + \theta_3 f_3(s)$
- Is there a good linear approximation?
- Yes!
- $\theta_0 = 10, \theta_1 = \theta_2 = 0, \theta_3 = -1$



What about our domains ?





Linear Value Function Approximation

- Define a set of state features $f_1(s), \ldots, f_n(s)$
  - The features are used as our representation of states
  - States with similar feature values will be considered to be similar
  - More complex functions require more complex features $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 f_1 + \theta_2 f_2 + \cdots + \theta_n f_n(s)$
- Our goal is to learn good parameter values (i.e. feature weights) that approximate the value function well
  - How can we do this?

Linear Value Function Approximation

- Define a set of state features $f_1(s), \ldots, f_n(s)$
  - The features are used as our representation of states
  - States with similar feature values will be considered to be similar
  - More complex functions require more complex features $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 f_1 + \theta_2 f_2 + \cdots + \theta_n f_n(s)$
- Our goal is to learn good parameter values (i.e. feature weights) that approximate the value function well
  - How can we do this?
  - Let's try using TD-based RL and somehow update parameters based on each experience.

TD-based RL for Linear Approximators

TD-based RL for Linear Approximators

1. Start with initial parameter values
2. Execute action from explore/exploit policy
3. Update estimated model (if model is not available)
4. Perform TD update for each parameter: $\theta_i := ?$
5. Goto 2

What is a “TD update” for a parameter?

Aside: Gradient Descent

Given a function $f(\theta_1, ..., \theta_n)$ of n real values $\theta = (\theta_1, \ldots, \theta_n)$, suppose we want to minimize f with respect to θ

Gradient Descent

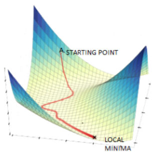
- The gradient of f at point θ, denoted $\nabla f(\theta)$ is an n-dimensional vector that points in the direction where f increases most steeply at point θ.
- Calculus tells us that $\nabla f(\theta)$ is just a vector of partial derivatives

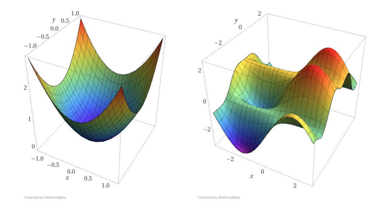
$$\nabla f(\theta) = \left[\frac{\partial f(\theta)}{\partial \theta_1}, \ldots, \frac{\partial f(\theta)}{\partial \theta_n} \right]$$

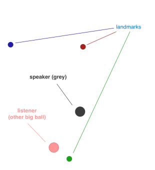
where $\frac{\partial f(\theta)}{\partial \theta_i} = \lim_{\epsilon \to 0} \frac{f(\theta_1, \ldots, \theta_{i-1}, \theta_i + \epsilon, \theta_{i+1}, \ldots, \theta_n) - f(\theta)}{\epsilon}$

- We can decrease f by moving in negative gradient direction

Aside: Gradient Descent








```
+-----------------------+
| : |F: | : | : | : |F: |
| :_| : : : | : | : | : |
| : :_: : : : : : : : : |
| : : : : : | : : : : : |
| : : : : : | : : : : : |
| : : : : : : : : : :_: |
| | :G| | | :G| | | : | |
+-----------------------+
```

Relevance to our domains ?

Aside: Gradient Descent for Squared Error

- Suppose that we have a sequence of states and target values for each state $\langle s_1, \mathcal{V}(s_1)\rangle, \langle s_2, \mathcal{V}(s_2)\rangle, \ldots$
  - for instance, produced by TD-based RL loop
- Our goal is to ?

Aside: Gradient Descent for Squared Error

- Suppose that we have a sequence of states and target values for each state $\langle s_1, \mathcal{V}(s_1)\rangle, \langle s_2, \mathcal{V}(s_2)\rangle, \ldots$
  - for instance, produced by TD-based RL loop
- Our goal is to -> minimize the sum of squared errors between our estimated function and each target value:

$$\mathbb{E}_j = \frac{1}{2}(\mathcal{V}_\theta(s_j) - v(s_j))^2$$

  where
  - $\mathbb{E}_j$ is the squared error of example j
  - $\mathcal{V}_\theta(s_j)$ is our estimated value for s_j
  - $v(s_j)$ is the target of s_j
- After seeing s_j ?

Aside: Gradient Descent for Squared Error

- Suppose that we have a sequence of states and target values for each state $\langle s_1, \mathcal{V}(s_1)\rangle, \langle s_2, \mathcal{V}(s_2)\rangle, \ldots$
  - for instance, produced by TD-based RL loop
- Our goal is to -> minimize the sum of squared errors between our estimated function and each target value:

$$\mathbb{E}_j = \frac{1}{2}(\mathcal{V}_\theta(s_j) - v(s_j))^2$$

where

  - $\mathbb{E}_j$ is the squared error of example j
  - $\mathcal{V}_\theta(s_j)$ is our estimated value for s_j
  - $v(s_j)$ is the target of s_j
- After seeing s_j -> the gradient descent rule tells us that we can decrease error by updating parameters by:

$$\theta_i := \theta_i - \alpha \frac{\partial \mathbb{E}_j}{\partial \theta_1}$$

Aside: continued ...

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial \mathbb{E}_j}{\partial \theta_i}$$

$$= \theta_i - \alpha \frac{\partial \mathbb{E}_j}{\partial \mathcal{V}_\theta(s_j)} \frac{\partial \mathcal{V}_\theta(s_j)}{\partial \theta_i}$$

$$= \theta_i - \alpha (\mathcal{V}_\theta(s_j) - v(s_j)) \frac{\partial \mathcal{V}_\theta(s_j)}{\partial \theta_i}$$

$$=^{linear} \theta_i - \alpha (\mathcal{V}_\theta(s_j) - v(s_j)) f_i(s_j)$$

Aside: continued ...

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial \mathbb{E}_j}{\partial \theta_i}$$

$$= \theta_i - \alpha \frac{\partial \mathbb{E}_j}{\partial \mathcal{V}_\theta(s_j)} \frac{\partial \mathcal{V}_\theta(s_j)}{\partial \theta_i}$$

$$= \theta_i - \alpha (\mathcal{V}_\theta(s_j) - v(s_j)) \frac{\partial \mathcal{V}_\theta(s_j)}{\partial \theta_i}$$

$$=^{linear} \theta_i - \alpha (\mathcal{V}_\theta(s_j) - v(s_j)) f_i(s_j)$$

- Thus the update becomes: $\theta_i := \theta_i + \alpha(v(s_j) - \mathcal{V}_\theta(s_j)) f_i(s_j)$
- (For linear functions) this update is guaranteed to converge to best approximation for suitable learning rate schedule

TD-based RL for Linear Approximators

1. Start with initial parameter values
2. Execute action from explore/exploit policy
3. Update estimated model (if model is not available)
4. Perform TD update for each parameter:

$$\theta_i := \theta_i + \alpha(v(s_j) - \mathcal{V}_\theta(s_i))f_i(s_j)$$

5. Goto 2

What should we use for "target value" $v(s)$?

Use the TD prediction based on the next state $s^{'}$
$v(s) = R(s) + \gamma\mathcal{V}_\theta(s^{'})$ the same as previous TD methods, only with approximation.

TD-based RL for Linear Approximators

TD-based RL for Linear Approximators

1. Start with initial parameter values
2. Execute action from explore/exploit policy
3. Update estimated model (if model is not available)
4. Perform TD update for each parameter:

$$\theta_i := \theta_i + \alpha(R(s) + \gamma \mathcal{V}_\theta(s') - \mathcal{V}_\theta(s_j)) f_i(s_j)$$

5. Goto 2

In what way do we depend on a model

TD-based RL for Linear Approximators

1. Start with initial parameter values
2. Execute action from explore/exploit policy
3. Update estimated model (if model is not available)
4. Perform TD update for each parameter:

$$\theta_i := \theta_i + \alpha(R(s) + \gamma \mathcal{V}_\theta(s') - \mathcal{V}_\theta(s_j))f_i(s_j)$$

5. Goto 2

In what way do we depend on a model

- Step 2 requires a model to select greedy action
    - For applications such as Backgammon it is easy to get a simulation-based model
    - For others it is difficult to get a good model

Q-Learning for Linear Approximators

Features are function of states and actions:

$$Q_\theta = \theta_0 + \theta_1 f_1(s,a) + \cdots + \theta_n f_n(s,a)$$

Q-Learning for Linear Approximatorss

1. Start with initial parameter values
2. Execute action from explore/exploit policy giving s' (should converge to greedy policy, i.e., GLIE)
3. Perform TD update for each parameter:

$$\theta_i := \theta_i + \alpha \left(R(s) + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s,a) \right) f_i(s,a)$$

4. Goto 2

Converges under some conditions.

What will we try to approximate ? How ?

Model-Based with Function Approximators

What will we try to approximate ? How ?

- reward and transition function.

Reminder: Value-Based and Policy-Based RL

- Value Based
  - Learnt Value Function
  - Implicit policy (e.g. ϵ-greedy)
- Policy Based
  - No Value Function
  - Learnt Policy
- Actor-Critic
  - Learnt Value Function
  - Learnt Policy

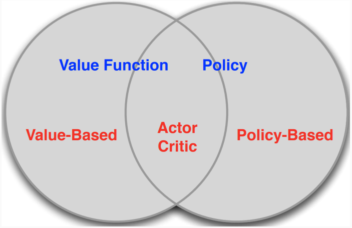


Image by David Silver

Policy-Based Reinforcement Learning

- So far, we approximated the value or action-value function using parameters θ

$$\mathcal{V}_\theta(s) \approx \mathcal{V}^\pi(s)$$

$$Q_\theta(s) \approx Q^\pi(s)$$

- A policy was generated directly from the value function
  - e.g. using ϵ-greedy
- An alternative is to directly parametrise the policy $\pi_\theta(s,a) = \mathcal{P}[a|s,\theta]$
- This is (again) model-free reinforcement learning

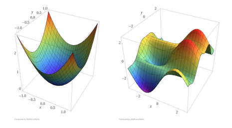
Advantages of Policy-Based RL

Advantages:

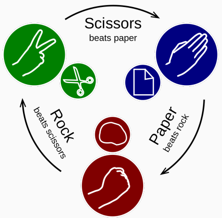
- Better convergence properties
- Effective in high-dimensional or continuous action spaces
- Can learn stochastic policies

Disadvantages:

- Typically converge to a local rather than global optimum
- Evaluating a policy is typically inefficient and high variance

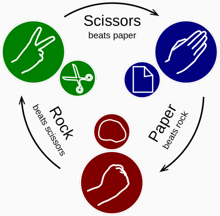


Example: Rock-Paper-Scissors



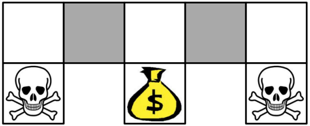
- Two-player game of rock-paper-scissors
  - Scissors beats paper
  - Rock beats scissors
  - Paper beats rock
- Consider policies for iterated rock-paper-scissors

Example: Rock-Paper-Scissors



- Two-player game of rock-paper-scissors
  - Scissors beats paper
  - Rock beats scissors
  - Paper beats rock
- Consider policies for iterated rock-paper-scissors
  - A deterministic policy is easily exploited
  - A uniform random policy is optimal (i.e. Nash equilibrium)

Example: Aliased Gridworld



- The agent cannot differentiate the grey states
- Consider features of the following form (for all N, E, S, W)

$$\phi(s,a) = 1(walltoN, a = moveE)$$

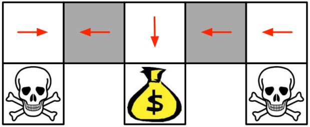
- Compare value-based RL, using an approximate value function

$$Q_\theta(s,a) = f(\phi(s,a),\theta)$$

- To policy-based RL, using a parametrised policy

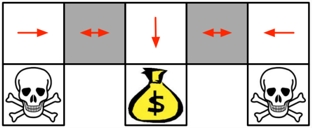
$$\pi_\theta(s,a) = g(\phi(s,a),\theta)$$

Example: Aliased Gridworld (continued 1)



- Under aliasing, an optimal deterministic policy will either
  - move W in both grey states (shown by red arrows)
  - move E in both grey states
- Either way, it can get stuck and never reach the money
- Value-based RL learns a near-deterministic policy
  - e.g. greedy or ϵ-greedy
- So it will traverse the corridor for a long time

Example: Aliased Gridworld (continued 2)



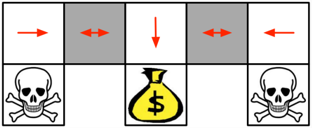
- An optimal stochastic policy will randomly move E or W in grey states

$$\pi_\theta(walltoNandS, moveE) = 0.5$$

$$\pi_\theta(walltoNandS, moveW) = 0.5$$

- It will reach the goal state in a few steps with high probability
- Policy-based RL can learn an optimal stochastic policy

Example: Aliased Gridworld (continued 2)



- An optimal stochastic policy will randomly move E or W in grey states

$$\pi_\theta(walltoNandS, moveE) = 0.5$$

$$\pi_\theta(walltoNandS, moveW) = 0.5$$

- It will reach the goal state in a few steps with high probability
- Policy-based RL can learn an optimal stochastic policy

Policy Objective Functions

- Goal: given policy $\pi_\theta(s, a)$ with parameters θ, find best θ
- But how do we measure the quality of a policy π_θ?

Policy Objective Functions

- Goal: given policy $\pi_\theta(s, a)$ with parameters θ, find best θ
- But how do we measure the quality of a policy π_θ?
    - In episodic environments we can use the start value
    - In continuing environments we can use the average value or average reward per time-step

Policy Optimisation

- Policy based reinforcement learning is an optimisation problem
- Find θ that maximises $J(\theta)$
- Some approaches do not use gradient
  - Hill climbing
  - Simplex / amoeba / Nelder Mead
  - Genetic algorithms
- Greater efficiency often possible using gradient
  - Gradient descent
  - Conjugate gradient
  - Quasi-newton
- We focus on gradient descent, many extensions possible
- And on methods that exploit sequential structure

Computing Gradients By Finite Differences

- To evaluate policy gradient of $\pi_\theta(s, a)$
- For each dimension $k \in [1, n]$
  - Estimate kth partial derivative of objective function w.r.t. θ
  - By perturbing θ by small amount ϵ in kth dimension

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$$

  where u_k is unit vector with 1 in kth component, 0 elsewhere
- Uses n evaluations to compute policy gradient in n dimensions
- Simple, noisy, inefficient - but sometimes effective
- Works for arbitrary policies, even if policy is not differentiable

Policy Gradient and REINFORCE

- Update parameters by stochastic gradient ascent / descent
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\nabla(\theta_t) = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

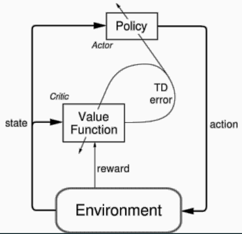
function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function

Actor Critic

- Monte-Carlo policy gradient still has high variance
- Reduce variance by adding a **critic** to estimate the action-value function

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$$

- Actor-critic algorithms maintain two sets of parameters
  - Critic- Updates action-value function parameters w
  - Actor- Updates policy parameters θ, in direction suggested by critic

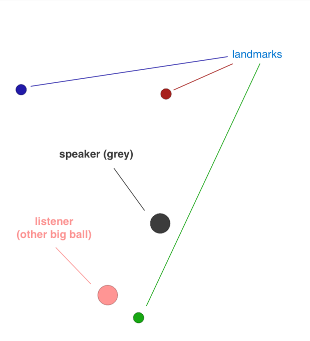


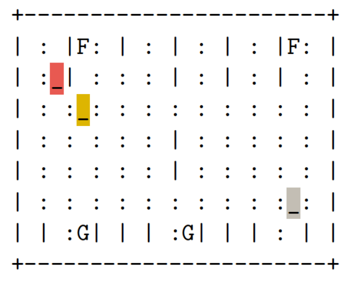
Action-Value Actor-Critic

- Simple actor-critic algorithm based on action-value critic
- Using linear value function approxmators.
  $Q_w(s, a) = \phi(s, a)^T w$
  - Critic Updates w by linear $TD(0)$
  - Actor Updates θ‘ by policy gradient

function QAC
  Initialise s, θ
  Sample $a \sim \pi_\theta$
  for each step **do**
    Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
    Sample action $a' \sim \pi_\theta(s', a')$
    $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
    $w \leftarrow w + \beta \delta \phi(s, a)$
    $a \leftarrow a', s \leftarrow s'$
  end for
end function

Our Running Examples





Parameterization ?

Additional Topics

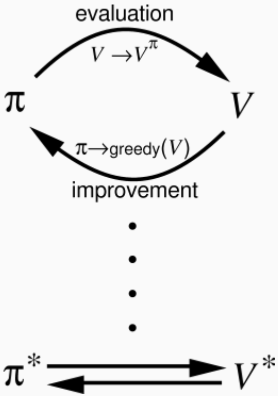
- **Incremental vs. Batch methods**:
  - sample efficiency
  - Reply buffer
- **Deep Reinforcement Learning**

What next ?

- Complex tasks
- Multi-agent settings

Recap

Anatomy of RL Algorithms



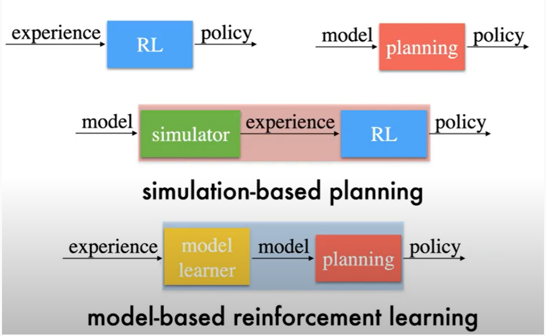


Figure 1: By Michael Littman

https://www.youtube.com/watch?v=45FKxa3qPHo

Model-Free vs. Model-Based RL

Model-Based RL

Model-Based RL

- We have so far explored two **model-free** approaches:
  - Monte-Carlo methods:

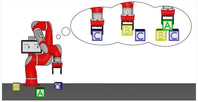
$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha(G_t - V(S_t))$$

  - Temporal Difference methods:

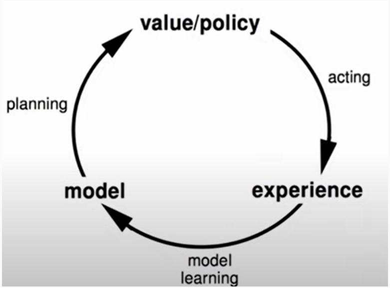
$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha(R(S_t) + \gamma V_\pi(S_{t+1}) - V_\pi(S))$$

- In **model-based RL** the agent uses a transition and reward model of the environment to make decisions about how to act.
  - The model may be initially known (e.g., chess) or unknown (e.g., robot manipulator).

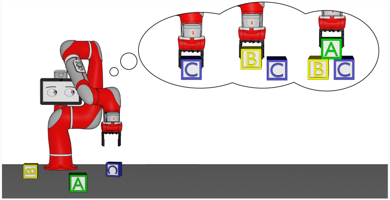
- Reinforcement learning systems can make decisions in one of two ways.
  - In the model-based approach, a system uses a predictive model of the world to ask questions of the form **“what will happen if I do x?”** to choose the best action.
  - In the alternative model-free approach, the modeling step is bypassed altogether in favor of **learning a control policy directly**. **“how much reward will I get if I do x?”**
- Although in practice the line between these two techniques can become blurred, as a coarse guide it is useful for dividing up the space of algorithmic possibilities.



BAIR blog by Michael Janner
https://bair.berkeley.edu/blog/2019/12/12/mbpo/



By Emma Brunskill
https://www.youtube.com/watch?v=vDF1BYWhqL8



Ideas for model-based evaluation ?

Ideas for model-based control ?

Model-based Estimation: Adaptive Dynamic Programming (ADP)

Adaptive Dynamic Programming (ADP)

- Follow the policy for a while
- Estimate transition model based on observations
- Learn reward function
- Use estimated model to compute utility of policy

$$V_\pi(s) = R(s) + \gamma \sum_{s'} \mathcal{P}(s'|s, \pi(s)) \cdot V_\pi(s')$$

How can we estimate transition model $\mathcal{P}$ and R?

Model-based Estimation: Adaptive Dynamic Programming (ADP)

Adaptive Dynamic Programming (ADP)

- Follow the policy for a while
- Estimate transition model based on observations
- Learn reward function
- Use estimated model to compute utility of policy

$$V_\pi(s) = R(s) + \gamma \sum_{s'} \mathcal{P}(s'|s, \pi(s) \cdot V_\pi(s')$$

How can we estimate transition model $\mathcal{P}$ and R?

Compute the fraction of times we see s' after taking a in state s (similarly for the reward function).

(*) Can bound error with Chernoff bound - that bounds the total amount of probability of some random variable Y that is in the “tail”, i.e. far from the mean.

Naïve Approach To Model-Based Control

Model-based approach to RL

1. Act randomly for a (long) time
2. Learn transition function and reward function
3. Use value iteration, policy iteration, LAO*,
4. Follow resulting policy thereafter.

Will this work ?

Naïve Approach To Model-Based Control

Model-based approach to RL

1. Act randomly for a (long) time
2. Learn transition function and reward function
3. Use value iteration, policy iteration, LAO*,
4. Follow resulting policy thereafter.

Will this work ?

Yes, if we do step 1 long enough and there are no "dead-ends".

Any problems?

Naïve Approach To Model-Based Control

Model-based approach to RL

1. Act randomly for a (long) time
2. Learn transition function and reward function
3. Use value iteration, policy iteration, LAO*,
4. Follow resulting policy thereafter.

Will this work ?

Yes, if we do step 1 long enough and there are no "dead-ends".

Any problems?

We will act randomly for a long time before exploiting what we know

Revision of Naïve Approach

Model-based approach to RL

1. Start with initial (uninformed) model
2. Solve for optimal policy given current model (using value or policy iteration)
3. Execute action suggested by policy in current state
4. Update estimated model based on observed transition
5. Goto 2

Revision of Naïve Approach

Model-based approach to RL

1. Start with initial (uninformed) model
2. Solve for optimal policy given current model (using value or policy iteration)
3. Execute action suggested by policy in current state
4. Update estimated model based on observed transition
5. Goto 2

This is just ADP but we follow the greedy policy suggested by current value estimate

Will this work ?

Model-based approach to RL

1. Start with initial (uninformed) model
2. Solve for optimal policy given current model (using value or policy iteration)
3. Execute action suggested by policy in current state
4. Update estimated model based on observed transition
5. Goto 2

This is just ADP but we follow the greedy policy suggested by current value estimate

Will this work ? No. Can get stuck in local optimum

What can be done ?

Reminder: Exploration versus Exploitation

- Two reasons to take an action in RL
  - **Exploitation:** To try to get reward. We exploit our current knowledge to get a payoff.
  - **Exploration:** Get more information about the world. How do we know if there is not a pot of gold around the corner?
- To explore we typically need to take actions that do not seem best according to our current model.
- Managing the trade-off between exploration and exploitation is a critical issue in RL
- Basic **intuition** behind most approaches:

Reminder: Exploration versus Exploitation

- Two reasons to take an action in RL
  - **Exploitation:** To try to get reward. We exploit our current knowledge to get a payoff.
  - **Exploration:** Get more information about the world. How do we know if there is not a pot of gold around the corner?
- To explore we typically need to take actions that do not seem best according to our current model.
- Managing the trade-off between exploration and exploitation is a critical issue in RL
- Basic **intuition** behind most approaches:
  - Explore more when knowledge is weak
  - Exploit more as we gain knowledge

ADP-based RL

1. Start with initial model
2. Solve for optimal policy given current model (using value or policy iteration)
3. Execute action suggested by an explore/exploit policy (explores more early on and gradually uses policy from 2)
4. Update estimated model based on observed transition
5. Goto 2

This is just ADP but we follow the explore/exploit policy

Will this work?

ADP-based RL

ADP-based RL

1. Start with initial model
2. Solve for optimal policy given current model (using value or policy iteration)
3. Execute action suggested by an explore/exploit policy (explores more early on and gradually uses policy from 2)
4. Update estimated model based on observed transition
5. Goto 2

This is just ADP but we follow the explore/exploit policy

Will this work? Depends on the explore/exploit policy

Any ideas?

Explore/Exploit Policies

- Greedy action is action maximizing estimated Q-value

$$Q(s,a) = R(s) + \gamma \sum_{s'} \mathcal{P}(s'|s,a) \cdot V_{\pi}(s')$$

  - where V is current optimal value function estimate (based on current model), and $\mathcal{R}$ and $\mathcal{P}$ are current estimates of the model
  - $Q(s,a)$ is the expected value of taking action a in state s and then getting the estimated value $\mathcal{V}(s')$ of the next state s'.
- We want an exploration policy that is GLIE (greedy in the limit of infinite exploration).

Explore/Exploit Policies

- GLIE policy 1:
  - On time step t select random action with probability $p(t)$ (i.e., ϵ) and greedy action with probability $1 - p(t)$
  - $p(t) = \frac{1}{t}$ (will lead to convergence, but is slow.)
  - Greedy action is the one that maximizes the Q value.
- GLIE policy 2: Boltzmann Exploration [1]
  - Select action a with probability $\mathcal{P}(a|s) = \frac{\exp(Q(s,a)/T)}{\sum_{a' \in \mathcal{A}} \exp(Q(s,a')/T)}$
  - T is the **"temperature"**: Large T means that each action has about the same probability. Small T leads to more greedy behavior.
  - Typically: start with large T and decrease with time.

[1] according to wikipedia - a Boltzmann distribution (also called Gibbs distribution) is a probability distribution or probability measure that gives the probability that a system will be in a certain state as a function of that state's energy and the temperature of the system.

Impact of Temperature

$$\mathcal{P}(a|s) = \frac{\exp(Q(s,a)/T)}{\sum_{a' \in \mathcal{A}} \exp(Q(s,a')/T)}$$

Suppose we have just two actions and that $Q(s, a_1) = 1$, and $Q(s, a_2) = 2$.

1. $T = 10$ gives $\mathcal{P}(a_1|s) = 0.48$, $\mathcal{P}(a_1|s) = 0.52$
   Almost equal probability, and so explore
2. $T = 1$ gives $\mathcal{P}(a_1|s) = 0.27$, $\mathcal{P}(a_1|s) = 0.73$
   Probabilities more skewed, so explore a_1 less
3. $T = .25$ gives $\mathcal{P}(a_1|s) = 0.02$, $\mathcal{P}(a_1|s) = 0.98$
   Almost always exploit a_2

Alternative Approach: Optimistic Exploration

Model-based approach to RL

1. Start with initial model
2. Solve for optimistic policy given current model using optimistic value iteration - that inflates value of actions leading to unexplored regions.
3. Execute greedy action suggested by the optimistic policy (explores more early on and gradually uses policy from 2)
4. Update estimated model based on observed transition
5. Goto 2

Basically act as if all “unexplored” state-action pairs are maximally rewarding

- Recall that value iteration iteratively performs the following update at all states:

$$V(s) := \mathcal{R}(s) + \gamma \max_a \sum_{s'} \mathcal{P}(s'|s,a) \cdot V_\pi(s')$$

  - Optimistic variant -

What do we mean by “explored enough”?

Optimistic Exploration

- Recall that value iteration iteratively performs the following update at all states:

$$V(s) := \mathcal{R}(s) + \gamma \max_a \sum_{s'} \mathcal{P}(s'|s,a) \cdot V_\pi(s')$$

  - Optimistic variant - adjusts update to make actions that lead to unexplored regions look promising
- Implement variant of VI that assigns the highest possible value V^{max} to any state-action pair that has not been explored enough
  - Maximum value is when we get maximum reward forever

$$V^{max} = \sum_{t=0}^{\infty} \gamma^t \cdot R^{max} = \frac{R^{max}}{1-\gamma}$$

What do we mean by “explored enough”?

Optimistic Exploration

- What do we mean by "explored enough"?
  - $N(s,a) > N_e$, where $N(s,a)$ is number of times action a has been tried in state s and N_e is a user selected parameter.
  - While the standard update rule is:

$$V^{max} := \sum_{t=0}^{\infty} \gamma^t \cdot R^{max} = \frac{R^{max}}{1-\gamma}$$

  - Optimistic value iteration computes an optimistic value function V^+ using updates:

$$V^+ := \mathcal{R}(s) + \gamma \max_a \begin{cases} V^{max}, & N(s,a) < N_e \\ \sum_{s'} \mathcal{P}(s'|s,a) \cdot V_\pi(s'), & \text{otherwise} \end{cases}$$

- The agent will behave initially as if there were wonderful rewards scattered all over around (-> optimistic)
- But after actions are tried enough times, we will perform standard "non-optimistic" value updates

For more details: R-max – A General Polynomial Time Algorithm for Near-Optimal https://www.natolambert.com/writing/debugging-mbrl

https://bair.berkeley.edu/blog/2019/12/12/mbpo/

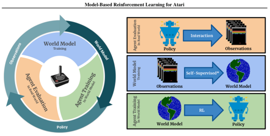
Approaches to Models-Based RL

- **Analytic gradient computation:**
  - Based on assumptions about the form of the dynamics and cost function (.e.g., Gaussian processes)
  - Can yield locally optimal control (e.g.Linear–quadratic regulator).
  - Even when these assumptions are not valid, can account for small errors introduced by approximated dynamics.
  - Linear (simplified) models, can also be used to provide guiding samples for training more complex nonlinear policies.
- **Sampling-based planning:**
  - Typically, for nonlinear dynamics models we resort to sampling action sequences (e.g., via random shooting). More sophisticated variants iteratively adjust the sampling distribution.
  - In discrete-action settings, however we can search over tree structures than to iteratively refine a single trajectory of waypoints.
    - Monte-Carlo Tree Search (MCTS)- has underpinned recent impressive results in games playing, and iterated width search.
  - In both continuous and discrete domains, can be combined with structured physics-based, object-centric priors.
- **Model-based data generation**
  - Many machine learning success stories rely on artificially increasing the size of a training set.
  - It is difficult to define a manual data augmentation procedure for policy optimization, but we can view a predictive model analogously as a learned

Model-Based RL: State-of-the-Art

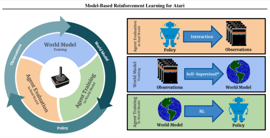
"In this paper, we explore how video prediction models can similarly enable agents to solve Atari games with orders of magnitude fewer interactions than model-free methods. We describe Simulated Policy Learning (SimPLe), a complete model-based deep RL algorithm based on video prediction models and present a comparison of several model architectures, including a novel architecture that yields the best results in our setting." (arXiv)

Paper by Kaiser et al 2020 *https://arxiv.org/abs/1903.00374*
https://medium.com/syncedreview/google-brain-simple-complete-model-based-reinforcement-learning-for-atari-b350a960

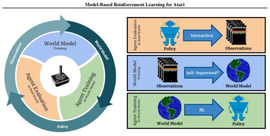


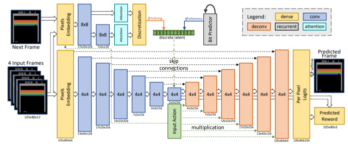
Model-Based RL: Example

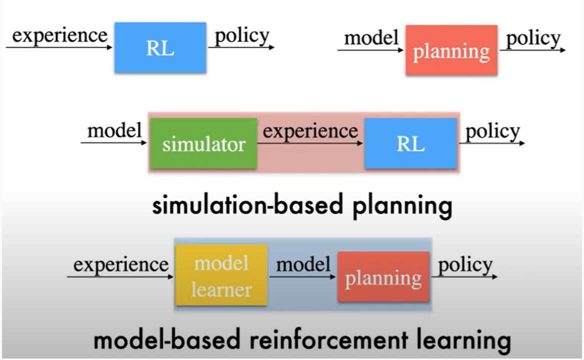
- SimPLe is a complete model-based deep RL algorithm that utilizes video prediction techniques and can train a policy to play a game within the learned model.
- SimPLe outperforms model-free algorithms in terms of learning speed on nearly all of the games, and in the case of a few games, does so by over an order of magnitude.
- The best model-free reinforcement learning algorithms require tens or hundreds of millions of time steps — equivalent to several weeks. SimPLe has obtained competitive results with only 100K interactions between the agent and the environment on Atari games, which corresponds to about two hours of real-time play.



- Agent starts interacting with the real environment following the latest policy (initialized to random).
- Collected observations used to train (update) current world model.
- Agent updates the policy by acting inside the world model. The new policy will be evaluated to measure performance and collect more data (back to 1).







Talk by Michael Littman (the funniest AI researcher in the world!)

https://youtu.be/45FKxa3qPHo?t=265

https://bair.berkeley.edu/blog/2019/12/12/mbpo/

Benchmarking Model-Based Reinforcement Learning by Wang et al. 2019 *https://arxiv.org/abs/1907.02057* When to Trust Your Model: Model-Based Policy Optimization Janner et al. 2021 *https://arxiv.org/abs/1906.08253* https://www.natolambert.com/writing/debugging-mbrl

https://bair.berkeley.edu/blog/2019/12/12/mbpo/

Model-Free vs. Model-Based RL

- Adaptive Dynamic Programming (model based)

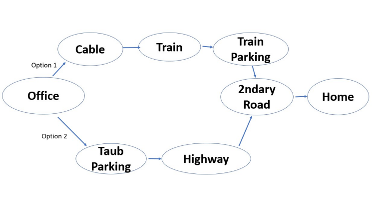
- Monte-Carlo Direct Estimation (model free)

- Temporal Difference Learning (model free)

Model-Free vs. Model-Based RL

- Adaptive Dynamic Programming (model based)
  - Harder to implement
  - Each update is a full policy evaluation (expensive)
  - Fully exploits Bellman constraints
  - Fast convergence (in terms of updates)
- Monte-Carlo Direct Estimation (model free)
  - Simple to implement
  - Each update is fast
  - Does not exploit Bellman constraints
  - Converges slowly
- Temporal Difference Learning (model free)
  - Update speed and implementation similiar to direct estimation
  - Partially exploits Bellman constraints—adjusts state to "agree" with observed successor (not all possible successors)
  - Convergence in between direct estimation and ADP

Model-Based RL for the Going Home Example ?



What Next

Large State Spaces

- When a problem has a large state space we can not longer represent the V or Q functions as explicit tables
- Even if we had enough memory
  - Never enough training data!
  - Learning takes too long

What to do??

What Next ?

Large State Spaces

- When a problem has a large state space we can not longer represent the V or Q functions as explicit tables
- Even if we had enough memory
  - Never enough training data!
  - Learning takes too long

What to do??

- Value function and policy approximators.
- Policy gradient and actor-critic.
- Monte-Carlo Tree Search
- and then...