Sequential Decision Making and Reinforcement Learning

(SDMRL)

Approximate Methods

Sarah Keren

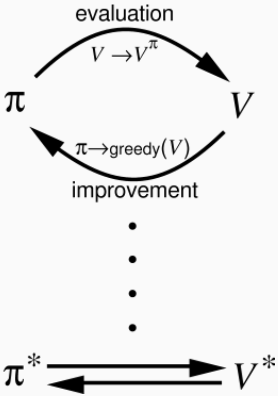
The Taub Faculty of Computer Science
Technion - Israel Institute of Technology

Acknowledgments

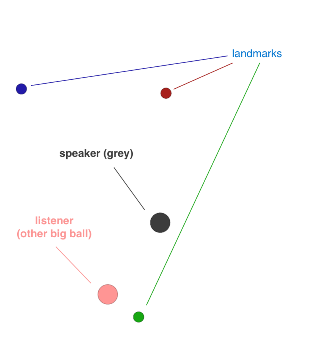
- David Sliver's course on RL: *https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silve*
- Slides by Malte Helmert, Carmel Domshlak, Erez Karpas and Alexander Shleyfman.

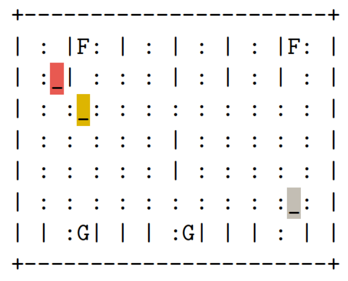
Large Spaces

Anatomy of RL Algorithms

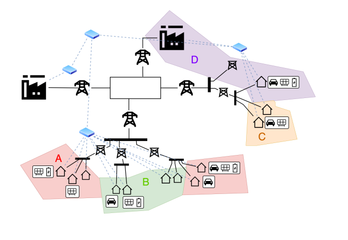


RL Examples


landmarks
speaker (grey)
listener
(other big ball)


F:
F:
:G
:G


traffic sign
recognition
surround view ON
STOP
STOP
50
Finish
3.750 km


D
C
A
B

Large Spaces

- When a problem has a large state and/or action space we can no longer represent the V or Q functions as explicit tables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??

Large Spaces

- When a problem has a large state and/or action space we can no longer represent the V or Q functions as explicit tables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...

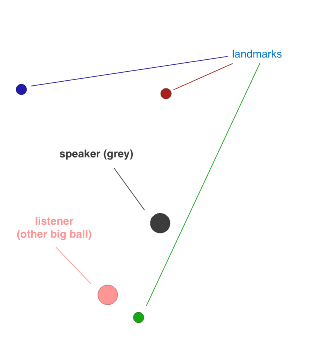
Approximation

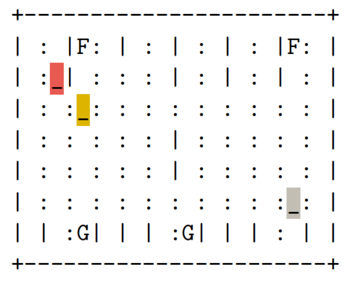
Revision:

- Methods discussed so far
  - Model-free RL
    - Monte-Carlo
    - Temporal Difference (TD) (e.g., Q-learning)
  - Model-based (e.g. Adaptive Dynamic Programming (ADP))
- 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

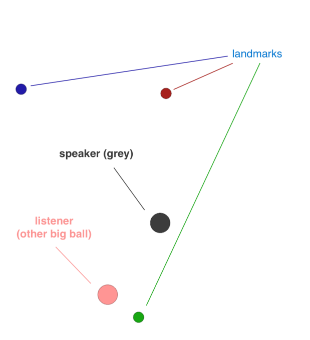
What to do?

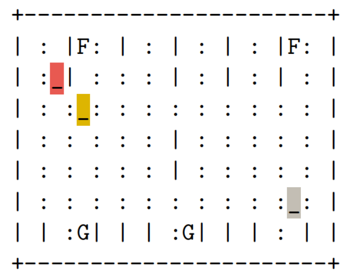
Function Approximation

- 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

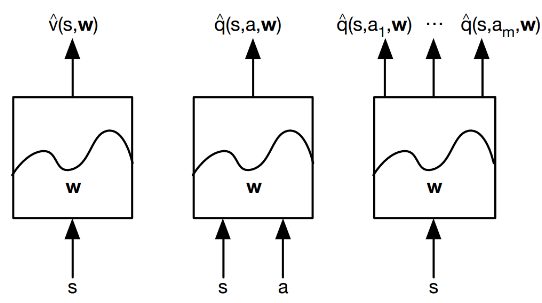


Image by David Silver

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.

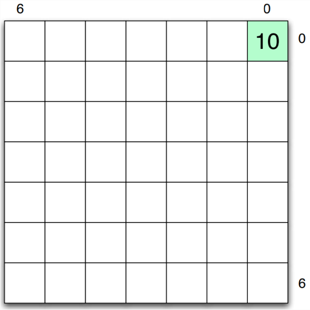
$$\tilde{\mathcal{V}}(s, \theta) = \begin{pmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{pmatrix} \cdot \begin{pmatrix} \theta_1 \\ \vdots \\ \theta_n \end{pmatrix}$$

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.

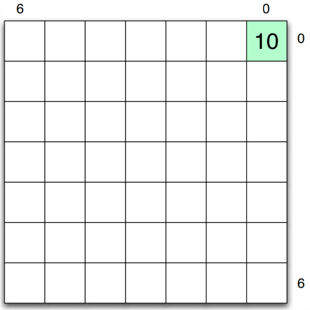
- Features for $s = (x, y)$: $f_1(s) = x$, $f_2(s) = y$
- Parameterized representation of value function ?



Example

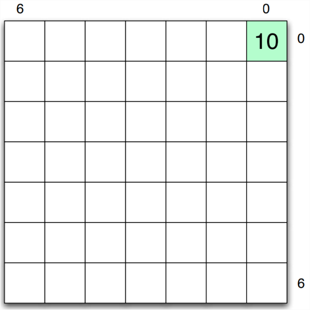
Grid with no obstacles, deterministic actions Up-Down-Left-Right, no discounting, -1 reward everywhere except +10 at goal.

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



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

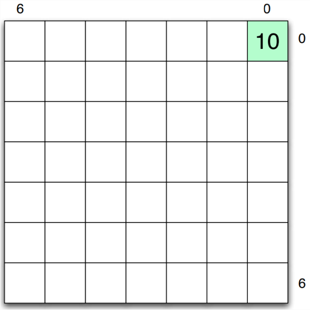
- Features for $s = (x, y)$: $f_1(s) = x$, $f_2(s) = y$
- Parameterized representation of value function ?
  - $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?



Example

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

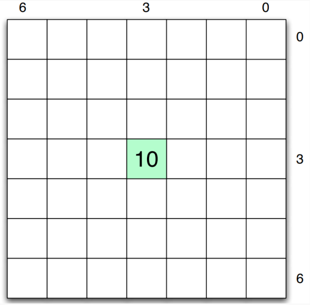
- Features for $s = (x, y)$: $f_1(s) = x$, $f_2(s) = y$
- Parameterized representation of value function ?
  - $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?
  - Yes.
  - $\theta_0 = 10$, $\theta_1 = \theta_2 = -1$
  - note: upper right is origin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



What if we change the reward function ?

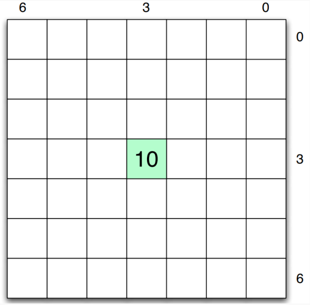
Grid with no obstacles, deterministic actions Up-Down-Left-Right, no discounting, -1 reward everywhere except +10 at goal.

- 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?



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!

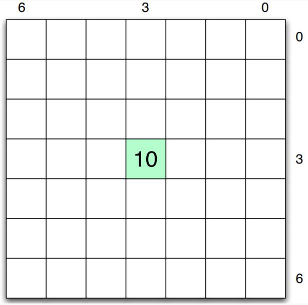


Suggestions?

But What If..

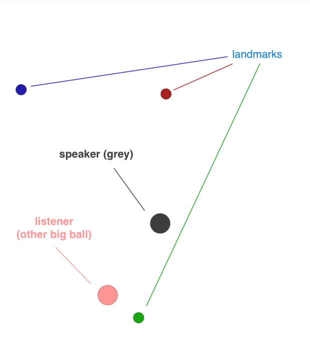
Grid with no obstacles, deterministic actions Up-Down-Left-Right, no discounting, -1 reward everywhere except +10 at goal.

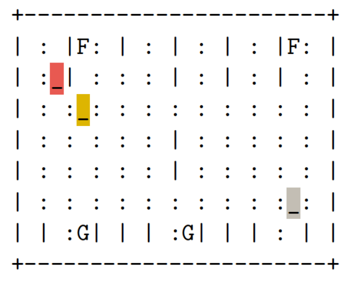
- Features for $s = (x, y)$:
  $f_1(s) = x$, $f_2(s) = y$
  $f_3(s) = |3 - x| + |3 - y|$
- $\mathcal{V}_\theta(s) = \theta_0 + \theta_1 x + \theta_2 y + \theta_3 f_3(s)$
- Is there a good linear approximation?
- Yes!
- $\theta_0 = 10, \theta_1 = \theta_2 = 0, \theta_3 = -1$



Linear Value Function Approximation

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.

Approximation for TD-based RL

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

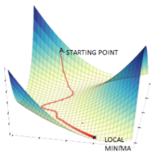
- The gradient of f at point θ, denoted $\nabla f(\theta)$ is an n-dimensional vector that points in the direction where f increases most steeply at point θ.
- Calculus tells us that $\nabla f(\theta)$ is just a vector of partial derivatives

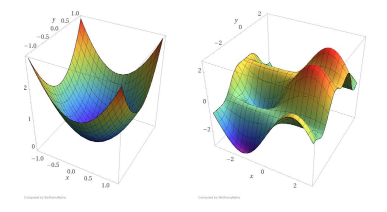
$$\nabla f(\theta) = \left[\frac{\partial f(\theta)}{\partial \theta_1}, \ldots, \frac{\partial f(\theta)}{\partial \theta_n} \right]$$

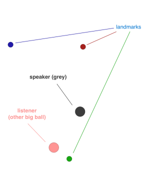
where $\frac{\partial f(\theta)}{\partial \theta_i} = \lim_{\epsilon \to 0} \frac{f(\theta_1, \ldots, \theta_{i-1}, \theta_i + \epsilon, \theta_{i+1}, \ldots, \theta_n) - f(\theta)}{\epsilon}$

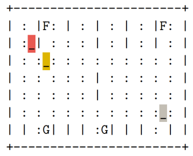
- We can decrease f by moving in negative gradient direction

Aside: Gradient Descent









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 ...

$$
\begin{aligned}
\theta_i \leftarrow \theta_i - \alpha \frac{\partial \mathbb{E}_j}{\partial \theta_i} \\
= \theta_i - \alpha \frac{\partial \mathbb{E}_j}{\partial \mathcal{V}_\theta(s_j)} \frac{\partial \mathcal{V}_\theta(s_j)}{\partial \theta_i} \\
= \theta_i - \alpha(\mathcal{V}_\theta(s_j) - v(s_j)) \frac{\partial \mathcal{V}_\theta(s_j)}{\partial \theta_i} \\
=^{linear} \theta_i - \alpha(\mathcal{V}_\theta(s_j) - v(s_j)) f_i(s_j)
\end{aligned}
$$

Aside: continued ...

$$
\begin{aligned}
\theta_i &\leftarrow \theta_i - \alpha \frac{\partial \mathbb{E}_j}{\partial \theta_i} \\
&= \theta_i - \alpha \frac{\partial \mathbb{E}_j}{\partial \mathcal{V}_\theta(s_j)} \frac{\partial \mathcal{V}_\theta(s_j)}{\partial \theta_i} \\
&= \theta_i - \alpha (\mathcal{V}_\theta(s_j) - v(s_j)) \frac{\partial \mathcal{V}_\theta(s_j)}{\partial \theta_i} \\
&=^{linear} \theta_i - \alpha (\mathcal{V}_\theta(s_j) - v(s_j)) f_i(s_j)
\end{aligned}
$$

- 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

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

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.

Approximation for Model-Based RL

What will we try to approximate ? How ?

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

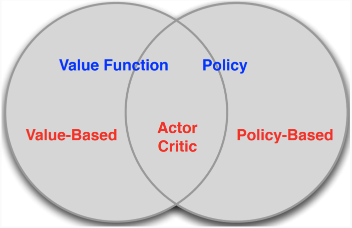


Image by David Silver

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

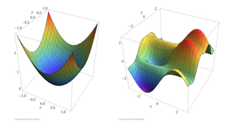
Advantages of Policy-Based RL

Advantages:

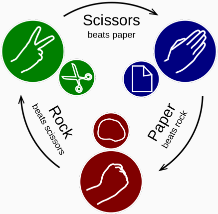
- Better convergence properties
- Effective in high-dimensional or continuous action spaces
- Can learn stochastic policies

Disadvantages:

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

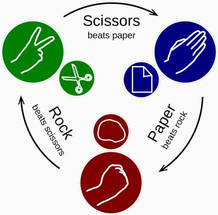


Example: Rock-Paper-Scissors



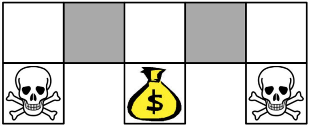
- Two-player game of rock-paper-scissors
    - Scissors beats paper
    - Rock beats scissors
    - Paper beats rock
- Consider policies for iterated rock-paper-scissors

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)

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

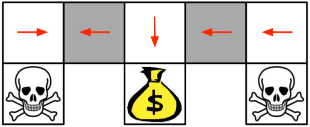
- Compare value-based RL, using an approximate value function

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

- To policy-based RL, using a parametrised policy

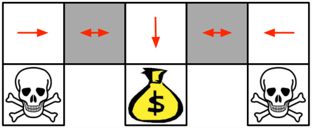
$$\pi_\theta(s, a) = g(\phi(s, a), \theta)$$

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)



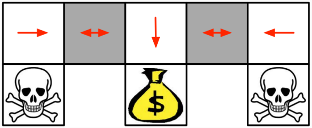
- An optimal stochastic policy will randomly move E or W in grey states

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

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

- 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 $Q^{\pi_\theta}(s_t, a_t)$

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

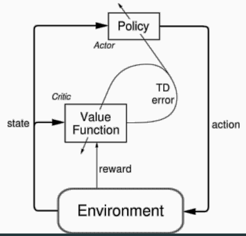
function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function

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

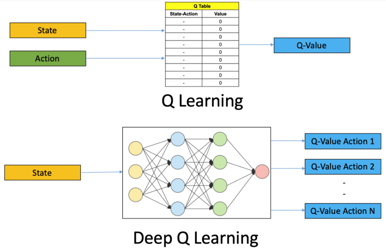
- Simple actor-critic algorithm based on action-value critic
- Using linear value function approxmators.
  $Q_w(s,a) = \phi(s,a)^T w$
  - Critic Updates w by linear $TD(0)$
  - Actor Updates θ‘ by policy gradient

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

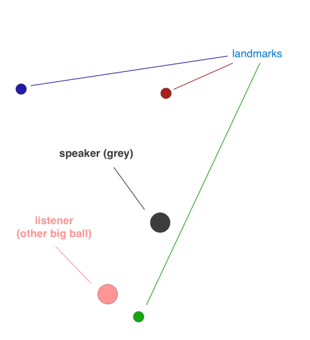
Conclusion

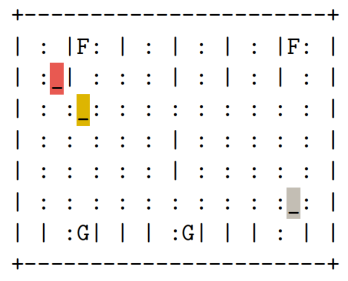
Deep Reinforcement Learning

- Combines reinforcement learning (RL) with deep learning to solve complex decision-making problems.

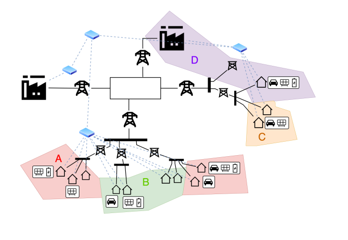


RL Examples


landmarks
speaker (grey)
listener
(other big ball)




traffic sign
recognition
surround view ON
STOP
STOP
50
Finish
3.750 km


D
A
B
C

Additional Topics

- Incremental vs. Batch methods:
  - sample efficiency
  - Reply buffer