Reinforcement Learning

Patricia J Riddle

Computer Science 767

Agents

Building a learning robot (or agent)

Sensors observe the state of the world - camera and sonar

A **set of actions** can be performed to alter the state - move forward, turn left

Its task is to learn a **control policy** for choosing actions that achieve goals –

docking onto a battery charger whenever its battery is low

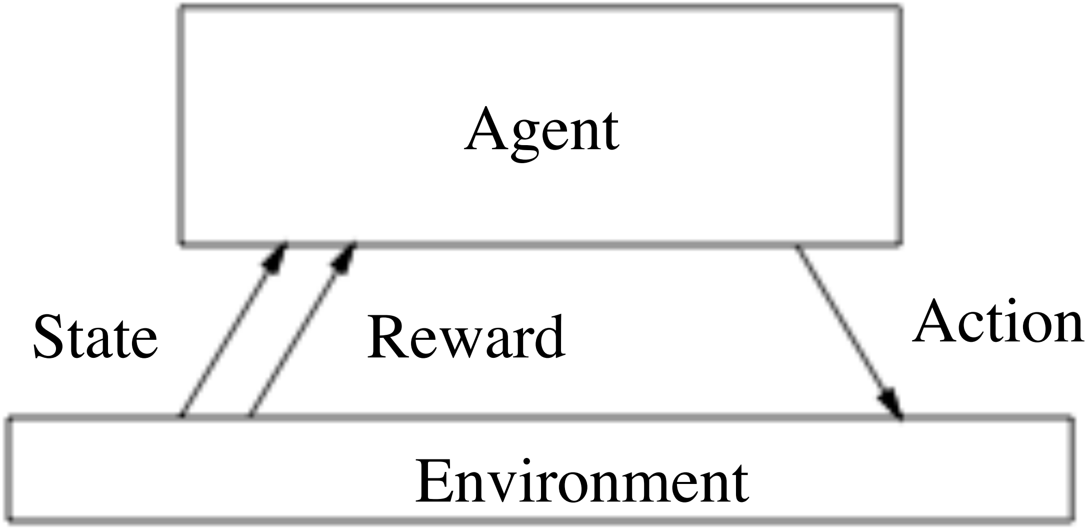
Agent Goals

We assume the goals of the agent can be defined by a **reward function** that assigns a numerical value - an **immediate payoff** - to each distinct action from each distinct state

100 for state-action transitions that immediately result in a connection to the charger and

0 for all other state-action transitions

An Agent



Control Policy

The **reward function** can be built into the robot or known only to an external teacher

The task of the robot is to **perform** sequences of actions, **observe** their consequences, and **learn** a control policy

The desired control policy is one that from any initial state chooses actions that **maximise the reward** accumulated over time by the agent

General Problem

Learning to control sequential processes

manufacturing optimization problems where reward is goods-produced minus costs involved

Sequential scheduling

choosing which taxis to send for passengers in a big city where reward is a function of the wait time of passengers and the total fuel costs of the taxi fleet

Specific settings:

actions are deterministic or nondeterministic,

agent **does or does not** have **prior knowledge** of the effects of its actions on the environment

Dynamic Programming

Field of mathematics

Traditionally used to solve problems of optimization and control

Limited in the size and complexity of problems

Supervised Learning

Training a parameterized **function approximator** (neural network) to represent functions

Sample input/output pairs

Set of questions with the right answers

Flight control system

Set of all sensor readings at a given time

How the flight control surfaces should move during the next millisecond

If we don't know how to build a controller in the first place, simple supervised learning won't help

Combination

Reinforcement learning combines dynamic programing and supervised learning

The computer is simply given a goal to achieve

Learns how to achieve the goal by trial-and-error interactions with its environment

Reinforcement Learning Model

Agent interacts with its **environment**

Agent **sensing** the environment, based on this sensory input choosing an action to perform in the environment

The **action changes the environment** in some manner and this change is communicated to the agent through a scalar **reinforcement** signal.

Three Parts

Three fundamental parts of a reinforcement learning problem:

The environment

The reinforcement function and

The value function

Environment

RL system learns mapping from situations to actions by trial-and-error interactions with a dynamic environment

Partially observable

Observations – sensor readings, symbolic descriptions, mental situations

Actions – low level, high level, or mental

Reinforcement Function

Exact **function** of future reinforcement the agent seeks to **maximise**

Mapping from **state/action pairs** to **reinforcements**

After performing an action, the RL agent will receive some reinforcement (scalar value)

The Function

RL agent **learns to perform actions** that will **maximise the sum of reinforcements** received when starting from some initial state and proceeding to a terminal state

System Designer

RL system designer to define a reinforcement function that properly defines the goals of the RL agent

3 noteworthy classes

Pure Delayed Reward

The reinforcements are all zero except at the terminal state

Sign of the scalar reinforcement indicates whether the terminal state is a goal state (a reward) or a state that should be avoided (a penalty)

Playing backgammon

Cart-pole (inverted pendulum)

Minimum Time to Goal

The reinforcement function **is -1 for ALL state transitions** except the transition to the **goal state**, in which case a **zero reinforcement** is returned.

Because the agent wishes to maximize reinforcement, it learns to choose actions that **minimize the time it takes to reach the goal state**, and in so doing learns the optimal strategy

Maximize or Minimize

The learning agent could just as easily learn to minimize the reinforcement function.

This might be the case when the reinforcement is a function of **limited resources** and the agent must learn to **conserve these resources** while achieving a goal (e.g., an airplane executing a maneuver while conserving as much fuel as possible).

Games

An alternative reinforcement function would be used in the context of a game environment, when there are two or more players with opposing goals.

In a game scenario, the RL system can learn to generate optimal behavior for the players involved by finding the maximun, minimax, or saddlepoint of the reinforcement function.

The Value Function

However, the issue of **how the agent learns** to choose “good” actions, or even how we might measure the utility of an action is not explained.

First, two terms are defined: **policy** and **value**

The Policy

A ***policy*** determines which action should be performed in each state;

a policy is a **mapping from states to actions**.

The Value

The ***value*** of a state is defined as the **sum of the reinforcements** received when **starting in that state** and **following some fixed policy** to a terminal state.

Optimal Policy

The optimal policy would therefore be the **mapping from states to actions** that **maximizes the sum of the reinforcements** when starting in an arbitrary state and performing actions until a terminal state is reached.

Under this definition the value of a state is dependent upon the policy.

Value Function

The *value function* is a **mapping from states to state values** and can be **approximated using any type of function approximator** (e.g., multi- layered perceptron, memory based system, radial basis functions, look-up table, etc.).

Value Function Example

An example of a value function can be seen using a simple Markov decision process with 16 states.

Markov Decision Process

- a Markov Decision Process is a discrete time stochastic control process.

- At each time step, the process is in some state *s*, and the decision maker may choose any action a that is available in state *s*.

- The process responds at the next time step by randomly moving into a new state s', and giving the decision maker a corresponding reward $R_a(s,s')$

Markov Decision Process

An MDP consists of a **set of states** X;

a **set of start states** S that is a subset of X;

a **set of actions** A;

a **reinforcement function** R where $R(x,a)$ is the expected immediate reinforcement for taking action a in state x;

and an **action model** P where $P(x'|x,a)$ gives the probability that executing action a in state x will lead to state x'.

Important

It is a requirement that the choice of action be **dependent solely** on the current state **observation $\boldsymbol{x}$**.

If knowledge of prior actions or states affects the current choice of action then the decision **process is not Markov**.

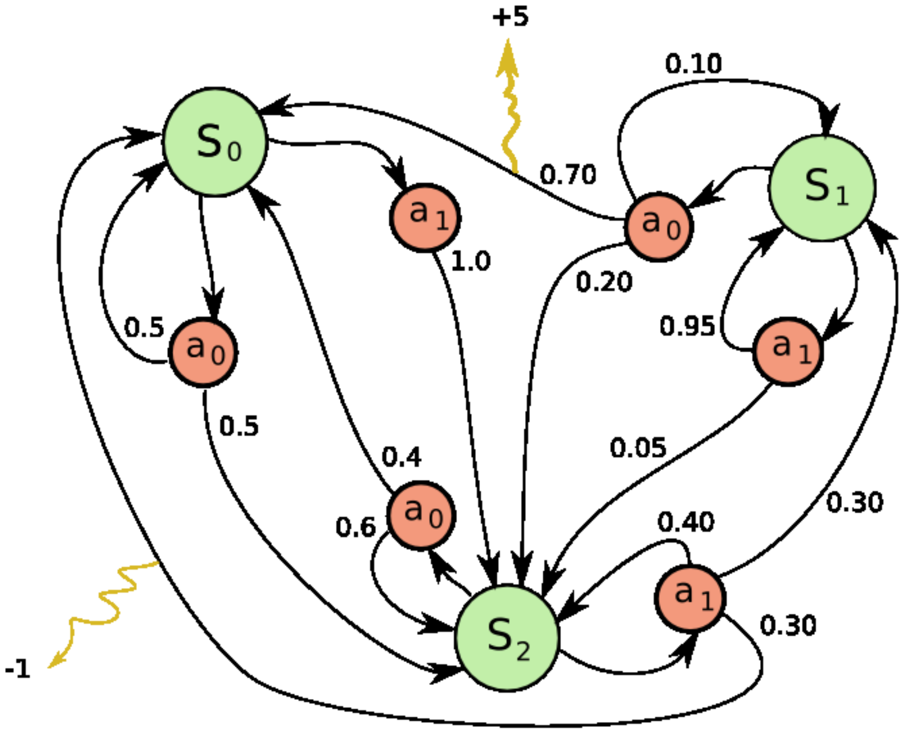
MDP vs MC

- Markov decision processes are an extension of Markov chains; the difference is the addition of actions (allowing choice) and rewards (giving motivation).

- Conversely, if only one action exists for each state and all rewards are zero, a Markov decision process reduces to a Markov chain.

Markov Decision Processes

http://en.wikipedia.org/wiki/Markov_decision_process



"Markov Decision Process example" by MistWiz –
Own work. Licensed under Public domain via Wikimedia Commons –
http://commons.wikimedia.org/wiki/File:Markov_Decision_Process_example.png#mediaviewer/File:Markov_Decision_Process_example.png

Value Function

The state space can be visualized using a 4x4 grid.

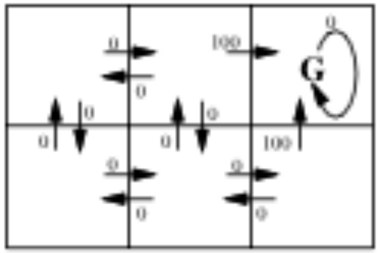
Each square represents a state.

The reinforcement function is -1 everywhere (i.e., the agent receives a reinforcement of -1 on each transition).

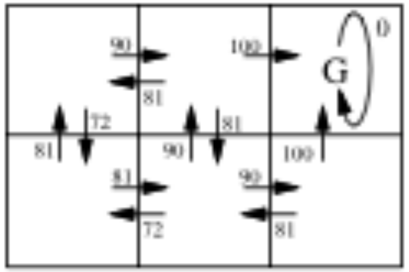
There are 4 actions possible in each state: north, south, east, west.

The goal states are the upper left corner and the lower right corner.

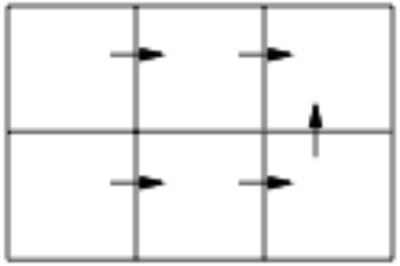
Grid-world



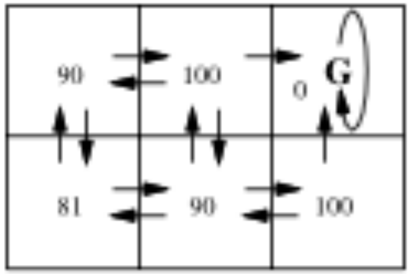
r(s,a) values
(immediate rewards)



Q(s,a) values



One optimal policy



V*(s) values

Random Value Function

The value function for the **random policy**

For each state the random policy **randomly chooses one of the four possible actions**.

The numbers in the states represent the **expected values** of the states.

For example, when starting in the lower left corner and following a random policy, on average there will be 22 transitions to other states before the terminal state is reached.

Random Policy Value Function

0	-14	-20	-22
-14	-18	-22	-20
-20	-22	-18	-14
-22	-20	-14	0

Optimal Value Function

The optimal value function is shown

Again, starting in the lower left corner, calculating the sum of the reinforcements when performing the optimal policy (the policy that will maximize the sum of the reinforcements), the value of that state is -3 because it takes only 3 transitions to reach a terminal state.

Optimal Policy Value Function

0	-1	-2	-3
-1	-2	-3	-2
-2	-3	-2	-1
-3	-2	-1	0

Optimal Value Function

Given the optimal value function, then it becomes a **trivial task** to **extract the optimal policy**.

For example, one can start in any state and simply choose the action that maximizes the immediate reinforcement received.

In other words, one can perform a **one level deep breadth-first search** over actions to find the action that will **maximize the immediate reward**

Optimal Policy

	←	←	←↓
↑	←↑	←↑→↓	↓
↑	←↑→↓	↓→	↓
↑→	→	→	

Fundamental Question

This leads us to the fundamental question of almost all of reinforcement learning research:

How do we **devise an algorithm** that will **efficiently find the optimal value function**?

Approximating the Value Function

- Reinforcement learning is a difficult problem because the learning system may perform an action and **not be told whether that action was good or bad**.
- Think of a game of chess

Temporal Credit Assignment Problem

- Caused by delayed reward
- Assigning blame to individual actions is the problem that makes reinforcement learning difficult.
- Surprisingly, there is a solution to this problem.
- It is based on a field of mathematics called ***dynamic programming***, and it involves just **two basic principles**

Principle 1

If an action causes **something bad to happen immediately**, such as crashing the plane, then the system **learns not to do that action** in that situation again.

So whatever action the system performed one millisecond before the crash, it will avoid doing in the future.

But that principle **doesn’t help for all the earlier actions** which didn’t lead to immediate disaster.

Principle 2

- If **all the actions** in a certain situation leads to bad results, then that **situation should be avoided.**

- So if the system has experienced a certain combination of altitude and airspeed many different times, whereby trying a different action each time, and all actions led to something bad, then it will learn that **the situation itself is bad**.

Power

This is a **powerful principle**, because the learning system can now **learn without crashing**.

In the future, any time it **chooses an action that leads to this particular situation**, it will **immediately learn that particular action is bad**, without having to wait for the crash.

Two principles

- By using these **two principles**, a learning system **can learn** to fly a plane, control a robot, or do any number of tasks.
- It can first **learn on a simulator**, then fine tune on the actual system.
- This technique is generally referred to as **dynamic programming**,

Essence of Dynamic Programing

- Initially, the **approximation** of the optimal value function **is poor**.
- The **mapping** from states to state values is **not valid**.
- The primary objective of learning is to **find the correct mapping**.
- Once this is completed, the **optimal policy** can **easily be extracted**.

Notation

$V^*(\mathbf{x}_t)$ is the optimal value function where $\mathbf{x}_t$ is the state vector;

$V(\mathbf{x}_t)$ is the approximation of the value function;

γ is a discount factor in the range [0,1] that causes immediate reinforcement to have more importance (weighted more heavily) than future reinforcement.

Approximation of the value function

$V(\mathbf{x}_t)$ will be initialized to random values and will contain no information about the optimal value function $V^*(\mathbf{x}_t)$.

The approximation of the optimal value function in a given state is equal to the true value of that state $V^*(\mathbf{x}_t)$ plus some error in the approximation,

$$V(\mathbf{x}_t) = e(\mathbf{x}_t) + V^*(\mathbf{x}_t)$$

where $e(\mathbf{x}_t)$ is the error in the approximation of the value of the state occupied at time t.

Next Step

The approximation of the value of the state reached after performing some action at time t is the true value of the state occupied at time t+1 plus some error in the approximation

$$V(\mathbf{x}_{t+1}) = e(\mathbf{x}_{t+1}) + V^*(\mathbf{x}_{t+1})$$

The value of state $\mathbf{x}_t$ for the optimal policy is the **sum of the reinforcements** when starting from state $\mathbf{x}_t$ and performing optimal actions **until a terminal state is reached**.

Simple Relationship

A simple relationship exists between the values of successive states, $\mathbf{x}_t$ and $\mathbf{x}_{t+1}$.

This relationship is defined by the Bellman equation and is expressed in equation.

The discount factor γ is used to exponentially decrease the weight of reinforcements received in the future

Bellman Equation

$$V^*(\mathbf{x}_t) = r(\mathbf{x}_t) + \gamma V^*(\mathbf{x}_{t+1})$$

The approximation $V(\mathbf{x}_\mathrm{t})$ also has the same relationship

$$V(\mathbf{x}_t) = r(\mathbf{x}_t) + \gamma V(\mathbf{x}_{t+1})$$

Error Relationship

The relationship in the errors of successive states.

$e(\mathbf{x}_t)=\gamma e(\mathbf{x}_{t+1})$

the process of learning is the process of finding a solution for the value function for all states $\mathbf{x}_\mathrm{t}$

Several learning algorithms have been developed for precisely this task.

Value Iteration Algorithm

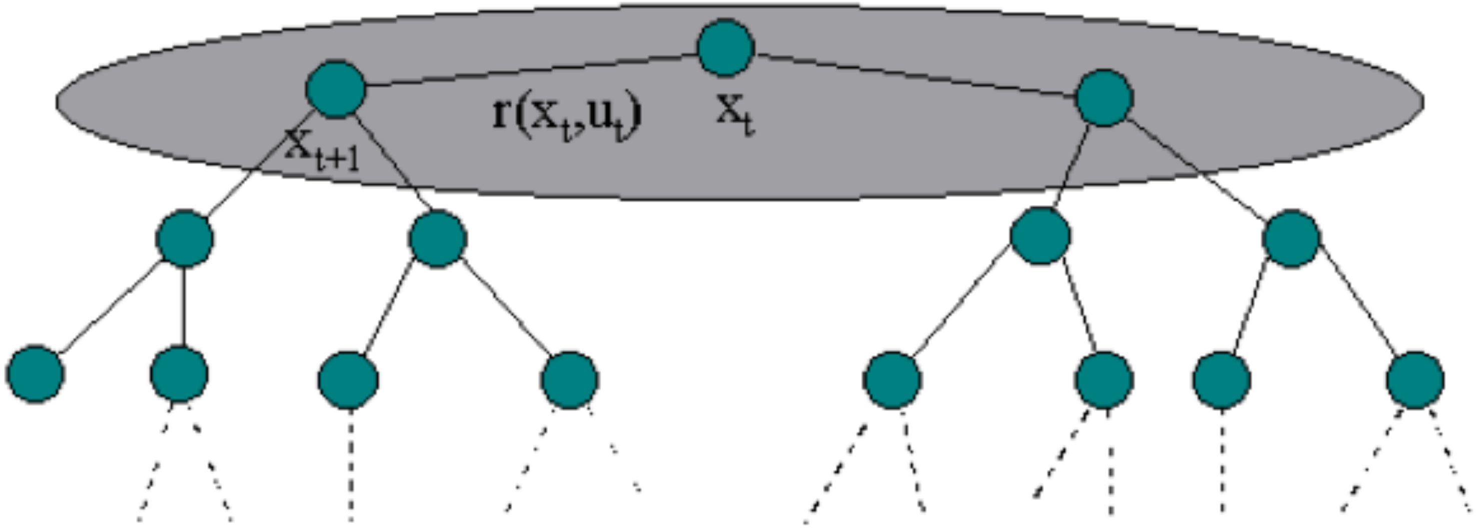
Assume the function approximator used to represent *V** is a **lookup table** (each state has a corresponding element in the table whose entry is the approximated state value)

One can find the optimal value function by performing sweeps through state space, **updating the value of each state** according to the equation until a sweep through state space is performed in which there are **no changes to state values** (the state values have converged).

$$\Delta\mathbf{w}_t = \max(r(\mathbf{x}_t,\mathbf{u}) + \gamma V(\mathbf{x}_{t+1})) - V(\mathbf{x}_t)$$

u is the action performed in state $\mathbf{x}_\mathrm{t}$ and causes a transition to state $\mathbf{x}_{\mathrm{t}+1}$, and $r(\mathbf{x}_\mathrm{t},\mathbf{u})$ is the reinforcement received when performing action **u** in state $\mathbf{x}_\mathrm{t}$.

Update Illustration



Single Update

- There are **two actions possible** in state $\mathbf{x}_t$, and each of these actions leads to a different successor state $\mathbf{x}_{t+1}$.

- In a value iteration update, one must first **find the action that returns the maximum value**.

- The only way to accomplish this is to **actually perform an action** and calculate the **sum of the reinforcement received** and the (possibly discounted) **approximated value** of the successor state $V(\mathbf{x}_{t+1})$.

Model Based

This must be done for all actions **u** in a given state $\mathbf{x}_t$, and is **not possible without a model** of the dynamics of the system.

In the case of a robot deciding to choose between paths to follow**, it is not possible to choose one path, observe the successor state, and then return to the starting state** to explore the results of the next available action.

Instead, the robot **must in simulation perform these actions and observe the results**.

Then, **based on the simulation** results, the **robot may choose the action** that results in the maximum value.

Bellman Equation

$\max(r(\mathbf{x}_t,\mathbf{u})+\gamma V(\mathbf{x}_{t+1}))-V(\mathbf{x}_t)$

Is simply the difference in the two sides of the Bellman equation, with the exception that we have generalized the equation to allow for

Markov decision processes (multiple actions possible in a given state) rather than

Markov chains (single action possible in every state).

Bellman Residual

This expression is the Bellman residual, and is formally defined as

$$e(\mathbf{x}_t) = \max(r(\mathbf{x}_t,\mathbf{u})+\gamma V(\mathbf{x}_{t+1}))-V(\mathbf{x}_t)$$

is the error function defined by the Bellman residual over all of state space.

Each update reduces the value of $E(\mathbf{x}_\mathrm{t})$, and in the limit as the number of updates goes to infinity $E(\mathbf{x}_\mathrm{t})$=0.

When $E(\mathbf{x}_\mathrm{t})$=0 is satisfied and $V(\mathbf{x}_\mathrm{t})=V^*(\mathbf{x}_\mathrm{t})$.

Learning is accomplished.

Residual Gradient Algorithms

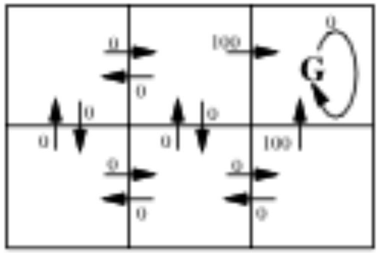
Thus far it has been assumed our function approximator is a **lookup table**.

However, this assumption severely **limits the size and complexity** of the problems solvable.

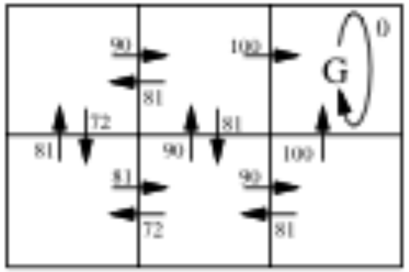
Many real-world problems have **extremely large or even continuous state spaces**.

Hence, an extension to classical value iteration is to use a function approximator that can **generalize and interpolate values of states never before seen**.

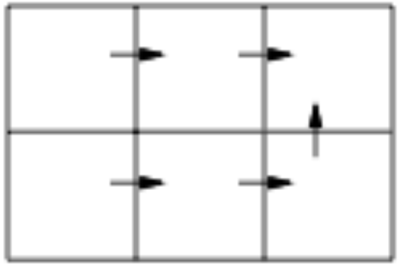
Grid-world



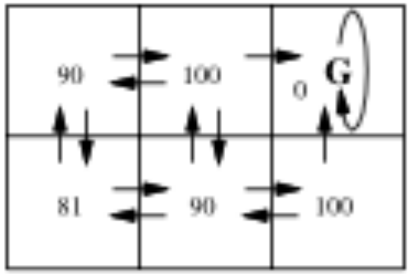
r(s,a) values
(immediate rewards)



Q(s,a) values



One optimal policy



V*(s) values

Neural Network

For example, one might use a neural network for the approximation $V(\mathbf{x}_t,\mathbf{w}_t)$ of $V^*(\mathbf{x})$, where $\mathbf{w}_t$ is the parameter vector.

The resulting network parameter update is

$$\Delta\mathbf{w_t}=-\alpha[\max_u(r(\mathbf{x_t},\mathbf{u})+\gamma V(\mathbf{x_{t+1}},\mathbf{w_t}))-V(\mathbf{x_t},\mathbf{w_t})](\partial V(\mathbf{x}_t,\mathbf{w}_t)/\partial\mathbf{w}_t)$$

where α is the learning rate,

$\max_u(r(\mathbf{x}_t,u)+\gamma V(\mathbf{x}_{t+1},\mathbf{w}_t))$is the desired output of the network,

$V(x_t,w_t)$ is the actual output of the network, and

$(\partial V(\mathbf{x}_t,\mathbf{w}_t)/\partial\mathbf{w}_t)$is the gradient of the output of the network with respect to the parameter vector.

Generalizing from Examples

Previous algorithms make **no attempt to estimate the V value for unseen state-action pairs**, unrealistic in large or infinite spaces or when the cost of executing actions is high

Substituted ANN for the table lookup and use each V update as a training example – state as input and Q^V as output

Multiple ANN

A more successful alternative is to train a separate ANN for each action using state as input and V as output

Another common alternative is to train one network with state as input and with one V output for each action

The convergence theorems no longer hold!!

Minimize the Bellman Residual

It “appears” that we are performing updates that will minimize the Bellman residual, but this is not necessarily the case.

The “target” value $\max(r(\mathbf{x}_t,\mathbf{u})+\gamma V(\mathbf{x}_{t+1},\mathbf{w}_t))$ is a function of the parameter vector **w** at time t.

Once the update to **w** is performed, the target has changed because it is now a function of a different parameter vector (the vector at time t+1).

Actually Increases

It is possible that the Bellman residual has actually been increased rather than decreased.

The error function on which gradient descent is being performed changes with every update to the parameter vector.

This can result in the values of the network parameter vector oscillating or even growing to infinity.

Mean Squared

One solution to this problem is to perform gradient descent on the mean squared Bellman residual.

Because this defines an unchanging error function, convergence to a local minimum is guaranteed.

This means that we can get the benefit of the generality of neural networks while still guaranteeing convergence.

Residual Gradient Algorithm

$\Delta\mathbf{w_t}=\alpha[r(\mathbf{x_t})+\gamma V(\mathbf{x_{t+1}},\mathbf{w_t}) - V(\mathbf{x_t},\mathbf{w_t})][(\gamma\partial V(\mathbf{x}_{t+1},\mathbf{w}_t)/\partial\mathbf{w_t})-(\gamma\partial V(\mathbf{x}_t,\mathbf{w}_t)/\partial\mathbf{w_t})]$

•The resulting method is referred to as a *residual gradient* algorithm because gradient descent is performed on the mean squared Bellman residual.

•It is important to note that if the MDP is non-deterministic then it becomes necessary to generate independent successor states to guarantee convergence to the correct answer.

Nondeterministic Markov decision processes

A ***deterministic*** Markov decision process is one in which the state transitions are deterministic (an action performed in state $\mathbf{x}_t$ always transitions to the same successor state $\mathbf{x}_{t+1}$).

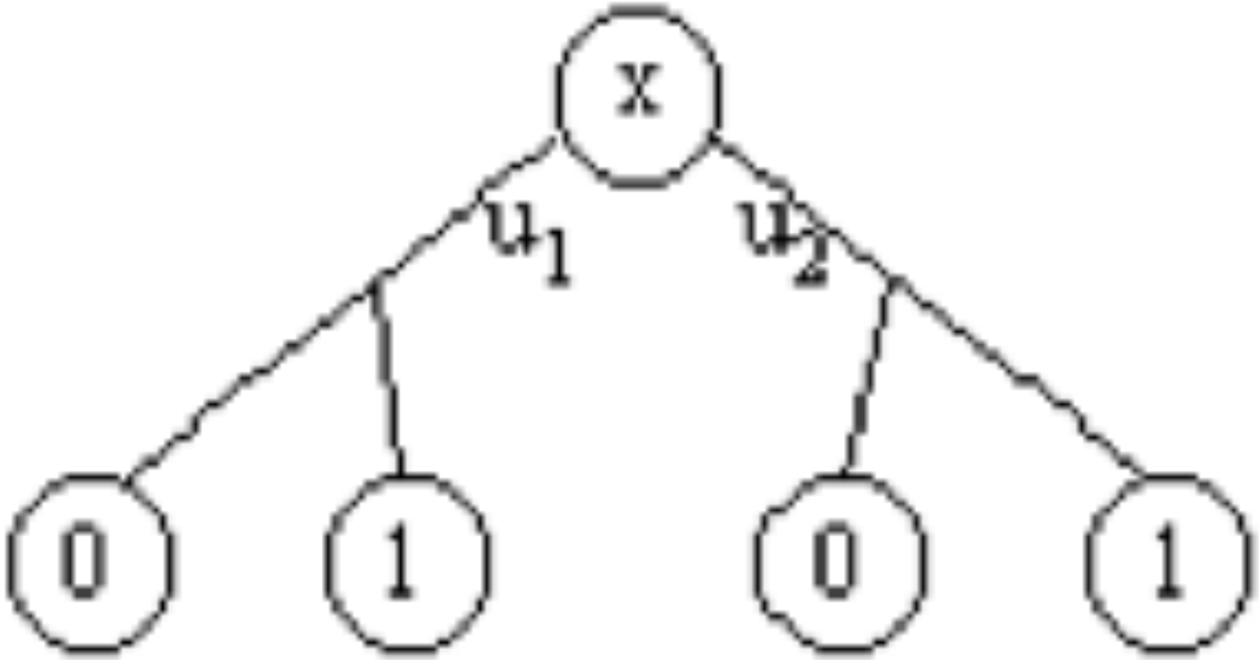
A ***non-deterministic*** Markov decision process, a probability distribution function defines a set of potential successor states for a given action in a given state.

Expected Value

If the MDP is non-deterministic, then value iteration requires that we find the action that returns the maximum ***expected* value** (the sum of the reinforcement and the integral **over all possible successor states** for the given action).

For example, to find the expected value of the successor state associated with a given action, one must **perform that action an infinite number of times, taking the integral over the values of all possible successor states for that action**.

Two possible actions



Which action?

There are two possible actions in state $\mathbf{x}$.

Each action returns a reinforcement of 0.

Action $\mathbf{u}_1$ causes a transition to one of two possible successor states with equal probability.

The same is true for action $\mathbf{u}_2$.

The values of the successor states are 0 and 1 for both actions.

Calculating Expected Value

Value iteration requires that the value of state **x** be equal to the maximum over actions of the sum of reinforcement and the expected value of the successor state.

By taking an infinite number of samples of successor states for action **u**1, one would be able to calculate that the actual expected value is 0.5.

The same is true for action u2. Therefore, the value of state **x** is 0.5

If perform value iteration on this MDP by taking a single sample of the successor state associated with each action instead of the integral, then **x** would converge to a value of 0.75.

Clearly the wrong answer.

Q-learning

Theoretically, value iteration is possible in the context of non-deterministic MDPs.

In practice it is **computationally impossible to calculate the necessary integrals** without added knowledge or some degree of modification.

Q-learning solves the problem of having to take the max over a set of integrals.

Rather than finding a mapping from states to state values (as in value iteration), *Q*-learning finds a **mapping from state/action pairs to values** (called *Q*-values).

Q function

Instead of having an associated value function, *Q*- learning makes use of the *Q*-function.

In each state, there is a *Q*-value associated with each action.

The definition of a *Q*-value is the **sum of the (possibly discounted) reinforcements** received when performing the associated action and then following the given policy thereafter.

Likewise, the definition of an optimal *Q*-value is the sum of the reinforcements received when performing the associated action and then following the optimal policy thereafter.

Q Function - Cheat

Optimal action is the one that maximizes the sum r(s,a) and V^* to the immediate successor state discounted by γ

$\pi^*(s)=\text{argmax}_a[r(s,a)+\gamma V^*(\delta(s,a))]$

But must have perfect knowledge of reward function r and the state transition function δ!!!

So create the Q function $Q(s,a)\equiv r(s,a)+\gamma V^*(\delta(s,a))$

Q Learning

Now $\pi^*(s)=\text{argmax}_a Q(s,a)$

Now we can select optimal actions even when we have no knowledge of r or δ

Q value for each state-action transition equals the r value for this transition plus the V^* value for the resulting state discounted by γ

Q Learning Properties

Still need V^* - iterative approximation or recursive definition

$V^*(s)=\max_{a'} Q(s,a')$, so

$Q(s,a)=r(s,a)+\gamma \max_{a'} Q(\delta(s,a),a')$

$\hat{Q}(s,a)$, the learner's estimate of Q, is stored in a big table which is initially filled with random values or zero

Table Update

The agent starts in some state, s, and chooses some action, a, and observes the result reward, r(s,a), and the new state, δ(s,a)

It then updates the table, $\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$

Doesn't need to know functions δ or r just executes the action and observes s´ and r so just **sampling these functions** at the current values of s and a

Bellman Equation for Q-learning

In the context of *Q*-learning, the value of a state is defined to be the maximum *Q*-value in the given state.

Given this definition it is easy to derive the equivalent of the Bellman equation for *Q*-learning.

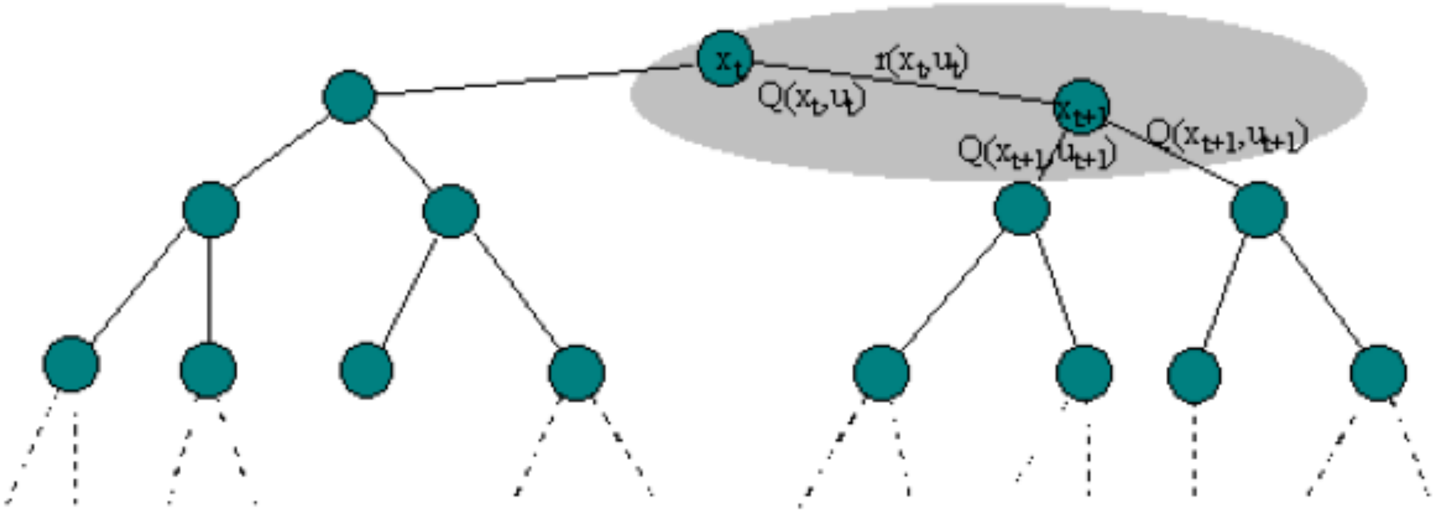
$Q(\mathbf{x}_t,\mathbf{u}_t) = r(\mathbf{x}_t,\mathbf{u}_t)+\gamma \ \mathrm{max}_{\mathrm{u(t+1)}}Q(\mathbf{x}_{t+1},\mathbf{u}_{t+1})$

Monte Carlo

Q-learning differs from value iteration in that it **doesn' t require** that in a given state **each action be performed** and the expected values of the successor states be calculated.

While value iteration performs an update that is analogous to a one level breadth-first search, *Q*-learning takes a **single-step sample of a Monte-Carlo roll-out**.

Single Step Sample



Update Equation

The update equation is valid when using a lookup table for the *Q*-function.

The *Q*-value is a **prediction of the sum of the reinforcements** received when performing the associated action and then following the policy.

To update that prediction $Q(\mathbf{x}_t,\mathbf{u}_t)$ one must perform the associated action $\mathbf{u}_t$, causing a transition to the next state $\mathbf{x}_{t+1}$ and returning a scalar reinforcement $r(\mathbf{x}_t,\mathbf{u}_t)$.

Then find the maximum *Q*-value in the new state to have all the necessary information for revising the prediction (*Q*-value) associated with the action just performed..

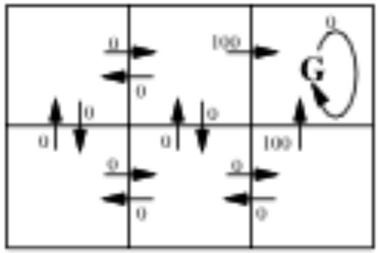
Unbiased Estimate

Q-learning **does not require one to calculate the integral** over all possible successor states in the case that the state transitions are non-deterministic.

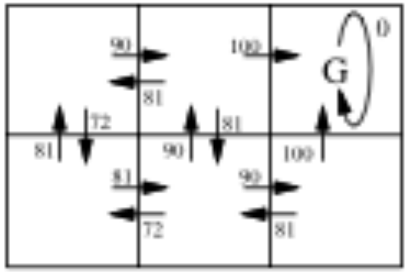
The reason is that a single sample of a successor state for a given action is an ***unbiased estimate*** of the expected value of the successor state.

After many updates the *Q*-value associated with a particular action will **converge to the expected sum** of all reinforcements received when performing that action and following the optimal policy thereafter.

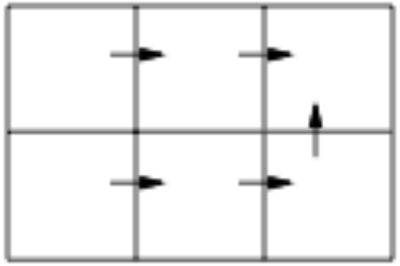
Grid-world



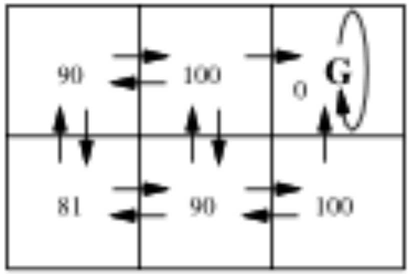
r(s,a) values
(immediate rewards)



Q(s,a) values



One optimal policy



V*(s) values

Model-free

Q learning needs **NO knowledge** of the reward function or the actions transition probabilities

Hence it is called **model-free**

However, they must be run in simulators of systems, which can be easily constructed from the distributions of the governing random variables.

It is well-known that simulating a complex system is considerably easier than generating a model of the system with all the transition probabilities

This is also why RL is said to **avoid the curse of modeling**.

Residual Gradient and Direct Q-learning

As it is possible to represent the value function with a **neural network** in the context of value iteration, so it is possible to represent the *Q*-function with a neural network in the context of *Q*-learning.

The information presented in the discussion of value iteration concerning convergence to a stable value function is also applicable to guaranteeing **convergence to a stable *Q*-function**.

Q function with neural network

Direct Q-learning

$\Delta\mathbf{w}t$ =$-\alpha[(r(\mathbf{x}_t,\mathbf{u}_t)+\gamma\ \mathrm{max}_{\mathrm{u(t+1)}}Q(\mathbf{x}_{t+1},\mathbf{u}_{t+1},\mathbf{w}_t))-$
$Q(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)]\ \partial Q(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)\ /\partial\mathbf{w}_\mathbf{t}$

Residual Graidient Q-learning

$\Delta\mathbf{w}t$ =$-\alpha[(r(\mathbf{x}_t,\mathbf{u}_t)+\gamma\ \mathrm{max}_{\mathrm{u(t+1)}}Q(\mathbf{x}_{t+1},\mathbf{u}_{t+1},\mathbf{w}_t))-$
$Q(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)]\ [(\partial\gamma Q(\mathbf{x}_{t+1},\mathbf{u}_{t+1})\ /\partial\mathbf{w}_\mathbf{t})-$
$(\partial Q(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)\ /\partial\mathbf{w}_\mathbf{t})]$

Updating Sequence

Q learning **need not train on optimal action sequences to converge to the optimal policy**

After the first full episode only one entry in the table will be updated.

If the agent follows the same sequence of actions the second table entry will be updated.

Updating Improvements

So perform updates in reverse chronological order!

Will converge in fewer iterations, although the agent has to use more memory to store the entire episode.

Retraining

Another strategy - store past state-action transitions and immediate rewards and retrain on them periodically

This can be a real win depending on relative costs (robot is very slow in comparison to replaying)

Many more efficient techniques when the system knows the δ and r functions – dynamic programming

Convergence

Qˆ values never decrease during training

$(\forall s,a,n)\hat{Q}_{n+1}(s,a) \geq \hat{Q}_{n(s,a)}$ (only if r is deterministic)

Qˆ will remain in the interval between 0 and Q

$(\forall s,a,n) 0 \leq \hat{Q}_n(s,a) \leq Q(s,a)$

Will converge if

1. Deterministic MDP,
2. Immediate rewards are bounded - $|r(s,a)|<c$
3. The agent selects actions such that it visits every state-action pair infinitely often - must execute a from s with nonzero frequency as the length of its action sequence approaches infinity

Nondeterministic Rewards and Actions

The functions δ(s,a) and r(s,a) can be viewed as first producing a probability distribution over outcomes based on s and a and then drawing an outcome at random according to this distribution - **nondeterministic markov decision process**

$Q(s,a) = E[r(s,a)]+\gamma\Sigma_{s'}P(s'|s,a)\max_{a'}Q(s',a')$, but is not guaranteed to converge

Decaying Weighted Average

Decaying weighted average of the current Qˆ and the revised estimate (important when r is not deterministic)

$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a)+\alpha_n[r+\max_{a'}\hat{Q}_{n-1}(s',a')]$, where

$$\alpha_n = \frac{1}{1+visits_n(s,a)}$$

Convergence long = 1.5 million games in Tesauro's backgammon program

Stop here

Advantage Learning

Although *Q*-learning is a significant improvement over value iteration, it is still **limited in scope** in at least one important way.

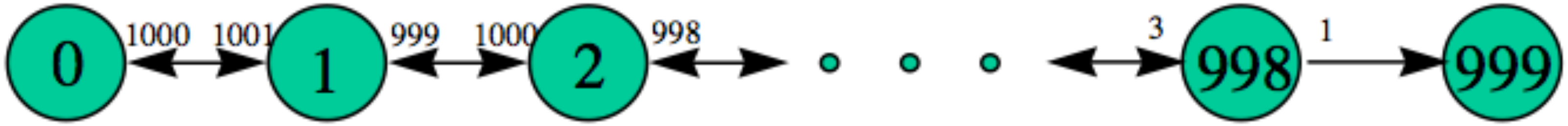
The **number of training iterations** necessary to sufficiently represent the optimal *Q*- function when using function approximators that generalize **scales poorly** with the size of the time interval between states.

The greater the number of actions per unit time (the smaller the increment in time between actions) the greater the number of training iterations required to adequately represent the optimal *Q*- function.

The explanation for this is demonstrated with a simple example.

Markov Decision Process

with 1000 states



Example

State 0 is the initial state and has a single action available, transition to state 1.

State 999 is an absorbing state.

In states 1..998 there are two actions available, transition to either the state immediately to the right or immediately to the left.

For example, in state 1, the action of going left will transition to state 0, and the action of going right will transition to state 2.

Each transition incurs a cost (reinforcement) of 1.

More Example

The objective is to **minimize the total cost accumulated** in transitioning from state to state until the absorbing states is reached.

The optimal *Q*-value for each action is represented by the numbers next to each state.

For example, in state 2 the optimal *Q*-value for the action of going left is 1000, and the optimal *Q*-value for the action of going right is 998.

The optimal policy can easily be found in each state by **choosing to perform the action with the minimum *Q*-value.**

Practical Limitations

When using a function approximator that generalizes over state/action pairs (any function approximator other than a lookup table or equivalent), it is possible to encounter **practical limitations in the number of training iterations required** to accurately approximate the optimal *Q*-function.

As the **time interval between states decreases** in size, the required **precision in the approximation** of the optimal *Q*-function **increases exponentially**.

Back to Example

The optimal *Q*-function associated with the MDP is linear and can be represented by a **simple linear function approximator**.

However, it requires an **unreasonably large number of training iterations** to achieve the level of precision necessary to generate the optimal policy.

The reason for the large number of training iterations is simple

Precision

The **difference in the *Q*-values** in a given state is **small relative** to the difference in the *Q*-values across states (a ratio of approximately 1:1000).

For example, the difference in the *Q*-values in state 1 is 2 (1001-999=2). The difference in the minimum *Q*-values in states 1 and 998 is 998 (999-1=998).

The approximation of the optimal *Q*-function **must achieve a degree of precision** such that the **tiny differences in *Q*-values in a single state are represented**.

Because the **differences in *Q*-values across states have a greater impact** on the mean squared error, during training the network learns **to represent these differences first.**

Infinite Precision

The differences in the *Q*-values in a given state have only a **tiny effect on the mean squared error** and therefore **get lost in the noise.**

To represent the differences in *Q*-values in a given state requires **much greater precision** than to represent the *Q*-values across states.

As the ratio of the time interval to the number of states decreases it becomes necessary to approximate the optimal *Q*-function with increasing precision.

In the limit, **infinite precision is necessary**.

Advantage Learning

Advantage learning does not share the scaling problem of *Q*-learning.

Similar to *Q*-learning, advantage learning **learns a function of state/ action** pairs.

However, in advantage learning the **value associated with each action is called an *advantage***.

Therefore, advantage learning **finds an advantage function** rather than a *Q*-function or value function.

The value of a state is defined to be the **value of the maximum advantage** in that state.

Degree of Suboptimality

For the state/action pair (**x**,**u**) an advantage is defined as the **sum of the value of the state and the utility (advantage) of performing action u** rather than the action currently considered best.

For optimal actions this utility is zero, meaning the value of the action is also the value of the state; for **sub-optimal actions** the **utility is negative**, representing the **degree of sub-optimality relative to the optimal action**.

Bellman Equation for Advantage Learning

The equivalent of the Bellman equation for advantage learning

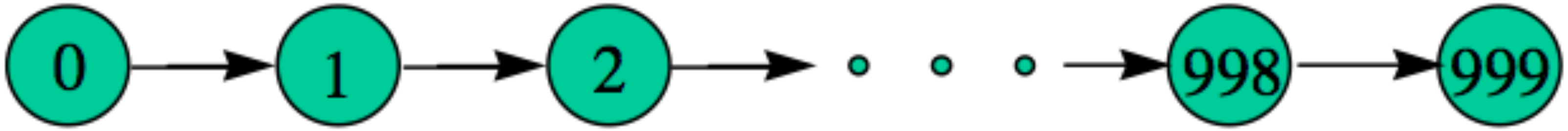
$A(\mathbf{x}_t,\mathbf{u}_t) = \max_{\mathrm{u(t)}} A(\mathbf{x}_t,\mathbf{u}_t) + ([<r(\mathbf{x}_t,\mathbf{u}_t)+\gamma \max_{\mathrm{u(t+1)}} A(\mathbf{x}_{t+1},\mathbf{u}_{t+1})>$
$-\max_{\mathrm{u(t)}} A(\mathbf{x}_t,\mathbf{u}_t)]/\Delta tK)$

where γ is the discount factor per time step,

K is a time unit scaling factor, and

<> represents the expected value over all possible results of performing action **u** in state $\mathbf{x}_\mathrm{t}$ to receive immediate reinforcement *r* and to transition to a new state $\mathbf{x}_{\mathrm{t+1}}$.

Markov Chain



Example

The initial state is 0 and the terminal state is 999.

Each state transition returns a cost (reinforcement) of 1 and the value of state 999 is defined to be 0.

Because this is a Markov chain it is not sensible to suggest that the RL system learn to minimize or maximize reinforcement.

Instead, we are concerned exclusively with predicting the total reinforcement received when starting from state n where n is a state in the range [1..998].

TD(λ)

Value iteration, *Q*-learning, and advantage learning can all solve this problem.

However, TD(λ) can solve it faster.

In the **context of Markov chains**, TD(λ) is identical to value iteration with the exception that TD(λ) updates the value of the current state based on a **weighted combination of the values of future states,** as opposed to using only the value of the immediate successor state.

Zero information

Recall that in value iteration the “target” value of the current state is the sum of the reinforcement and the value of the successor state, in other words, the right side of the Bellman equation.

$$V(\mathbf{x}_t,\mathbf{w}_t) = r(\mathbf{x}_t) + \gamma V(\mathbf{x}_{t+1},\mathbf{w}_t)$$

Notice that the “target” is also based on an estimate $V(\mathbf{x}_{\mathrm{t+1}},\mathbf{w}_{\mathrm{t}})$, and this estimate can be **based on zero information**.

Indeed, this is the case much of the time and can be demonstrated.

Arbitrary Updates

Assume that the value function for this Markov chain is represented using a lookup table.

In this case, our lookup table has 1000 elements, each corresponding to a state, and the entry in each element is the value of the corresponding state.

Before learning begins entries are **initialized to random values**.

The process of learning starts by **updating the value of state 0** to be the sum of the reinforcement received on transition from state 0 to state 1 and the value of state 1.

Remember, at this point **the value of state 1 is arbitrary**.

Inefficient

This is true for all states except the terminal state (999) which, by definition, has a value of 0.

Because **the initial values of states are arbitrary** (with the exception of the terminal state), the entire first sweep through the Markov chain (epoch) of training results in the **improvement of the approximation of the value function only in state 998**.

In the first epoch, only in state 998 is the update to the approximation based on something other than an arbitrary value.

This is **terribly inefficient**.

One Step

In fact, not until **999 epochs of training** have been performed **will the approximation of the value of state 0 contain any degree of "truth"** (the approximation is based on something other than an arbitrary value).

In epoch 2 of training, the approximation of the value of state 997 is updated based on an approximation of the value of state 998 that has as its basis the true value of state 999, rather than an arbitrary value.

In epoch 3, the approximation of the value of state 996 will be updated based on "truth" rather than an arbitrary value.

Each epoch moves "truth" back one step in the chain.

Weight Average of future

The approximation of the value of state $\mathbf{x}_t$ is updated based on the approximation of the value of the state one step into the future, $\mathbf{x}_{t+1}$.

If the value of a state were based on a **weighted average of the values of future states**, then “truth” would be propagated “back in time” much more efficiently.

SpeedUp

Instead of updating the value of a state based exclusively on the value of the immediate successor state one used the **next 2 successor states** as the basis of the update, then the number of epochs performed before the value of state 0 is no longer based on an arbitrary value is reduced from 1000 to 500.

If the value approximation of state 0 is based on a weighted combination of values of the **succeeding 500 states**, then only 2 epochs are required before the value approximation of state 0 is based on something other than an arbitrary value.

TD(λ)

This is precisely the function of TD(λ) (Sutton, 1988) for $0<\lambda<1$.

Instead of updating a value approximation based solely on the approximated value of the immediate successor state, TD(λ) basis the update on an exponential weighting of values of future states.

λ is the weighting factor.

TD(0), the case of $\lambda=0$, is identical to value iteration for the example problem stated above.

TD(1) updates the value approximation of state n based solely on the value of the terminal state.

Equations

The parameter update for TD(λ) is:

$$\Delta\mathbf{w}_t=\alpha(r(\mathbf{x}_t)+V(\mathbf{x}_{t+1},\mathbf{w}_t)-V(\mathbf{x}_t,\mathbf{w}_t))\sum_{\mathrm{k=1to\ t}}\lambda^{t-k}\nabla_{\mathrm{w}}V(\mathbf{x}_{\mathbf{k}},\mathbf{w}_{\mathbf{t}})$$

An incremental form of this equation can be derived as follows.

Given that $\mathbf{g}_{\mathrm{t}}$ is the value of the sum above for t, we can compute $\mathbf{g}_{\mathrm{t+1}}$, using only current information, as

$$\mathbf{g}_{t+1}=\nabla_{\mathbf{w}}V(\mathbf{x}_{k+1},\mathbf{w}_t)+\lambda\mathbf{g}_t.$$

Extending to MDP

Notice that the equations do not have a max or min term (over operators).

This suggests that TD(λ) is used exclusively in the context of prediction (Markov chains).

One way to extend the use of TD(λ) to the domain of Markov decision processes is to perform updates according to the regular equation while calculating the sum according to iterative equation when following the current policy.

Exploration

When a **step of exploration** is performed (choosing an action that is not currently considered “best”), the **sum of past gradients g in the iterative equation should be set to 0.**

The intuition behind this method follows.

The value of a state $\mathbf{x}_t$ is defined as the sum of the reinforcements received when starting in $\mathbf{x}_t$ and following the current policy until a terminal state is reached.

Intuition

During training, the **current policy is the best approximation to the optimal** policy generated thus far.

On occasion one must perform actions that don't agree with the current policy so that better approximations to the optimal policy can be realized.

However, one might not want the value of the resulting state propagated through the chain of past states.

This would **corrupt the value approximations** for these states by introducing information that is not consistent with the definition of a state value.

No Convergence

TD(λ) for λ=0 is equivalent to value iteration.

Likewise, the discussion of residual gradient algorithms is applicable to TD(λ) when λ=0.

However, this is not the case for $0<\lambda<1$.

No algorithms exist that guarantee convergence for TD(λ) for $0<\lambda<1$ when using a general function approximator.

Defined over the space

The fundamental question in reinforcement learning research is: **How do we devise an algorithm that will efficiently find the optimal value function?**

It was shown that the optimal value function is a solution to the set of equations defined by the Bellman equation.

The process of learning was subsequently described as the process of improving an approximation of the optimal value function by incrementally finding a solution to this set of equations.

One should notice that the Bellman equation is **defined over all of state space.**

Exploration

The optimal value function satisfies this equation for ALL $\mathbf{x}_t$ in state space.

This requirement introduces the **need for *exploration***.

Exploration is defined as intentionally **choosing to perform an action that is not considered "best"** for the express purpose of acquiring knowledge of unseen (or little seen) states.

In order to identify a (sub-)optimal approximation, **state space must be sufficiently explored**.

When to Explore?

For example, a robot facing an unknown environment **has to spend some time acquiring knowledge** of its environment.

Alternatively, **experience acquired during exploration must also be considered during action selection to minimize the costs** (negative reinforcements) associated with learning.

Although the robot must explore its environment, it should avoid collisions with obstacles.

However, the robot does not know which actions will result in collision until all of state space has been explored.

Trade-off

On the other hand, it is possible that a **policy that is "sufficiently" good will be recognized without having to explore all of state space**.

There is a **fundamental trade-off between exploration and exploitation** (using previously acquired knowledge to direct the choice of action).

Therefore, it is important to use exploration techniques that will **maximize the knowledge gained during learning while minimizing the costs** of exploration and learning time.

For a good introduction to the issues of efficient exploration see Thrun (1992).

Exploratory Policy

In the simulator, one may choose each action with the same probability.

This usually ensures that all samples are gathered properly and that all state-action pairs are updated infinitely often; the latter is a requirement for convergence to the correct values of the Q-values.

In practice, a so-called exploratory policy, with a bias towards the greedy action, is often used.

Experimentation Strategies

Common to use probabilistic approach to selecting actions

Actions with higher Qˆ are assigned higher probabilities, but every action has a non-zero probability

Probability of Selecting Action

$$P(a_i \mid s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

$P(a_i|s)$ is the probability of selecting action a_i, given

the agent is in state s,

where k>0 is the constant that determines how strongly the selection favors actions with high Q^ values

Shifting from Exploration to Exploitation

Sometimes k is varied with the number of iterations

the agent favors exploration during the early stages of learning,

then gradually shifts toward a strategy of exploitation

Limiting Greedy Exploration

With an exploratory strategy, in the kth iteration, one selects the greedy action $\text{argmax}_{u\varepsilon A(i)}$ Q(i,u) with a probability p^k and any one of the remaining actions with probability $(1-p^k)/(|A(i)|-1)$

A possible rule for the probability p^k is: $p^k = 1-(B/k)$; where B for instance could equal 0.5.

With such a rule, the **probability of selecting non-greedy actions is automatically decayed** to 0 with increasing k.

This is also called **limiting-greedy exploration.**

Types of Exploration

Random, undirected exploration, discussed above, can cause the algorithm to take **time exponential in the number of states** to converge (Whitehead, 1991).

Directed exploration strategies: counter-based (Sato et al.,1990), error-and-counter based (Thrun and Moller, 1992), and recency-based (Sutton, 1990) can overcome this drawback.

A number of exploration strategies have been discussed and experimented with in the literature: the Boltzmann strategy (Luce, 1959), the E3 strategy (Kearns and Singh, 2002), and the external-source strategy (Smart and Kaelbling, 2000).

SARSA

SARSA: SARSA (Rummery and Niranjan, 1994; Sutton, 1996; Sutton and Barto, 1998) is a well-known algorithm based on an “on-policy" control.

In on-policy control, a unique policy is evaluated for some time during the learning.

This is unlike Q-Learning, which does “off-policy" control, in which the policy being evaluated can change in every iteration.

SARSA uses the concept of learning in episodes, in which there is a “terminal" state and the episode terminates when the terminal state is reached.

SARSA(λ)

SARSA is a TD(0) algorithm.

TD(λ) can also be used in SARSA (see SARSA(λ) of Sutton (1996)) especially when the learning is episodic.

An important notion of eligibility traces, discussed in Singh and Sutton (1996), can be used to increase the power of TD(¸) methods by attaching variable weights to the reinforcements in the updating strategy.

When function approximation can be performed more easily with on-policy updates, an on-policy algorithm like SARSA becomes more effective than Q-Learning.

Discounted vs non-discounted

The discount factor γ is a number in the range of [0..1] and is used to weight near term reinforcement more heavily than distant future reinforcement.

The closer γ is to 1 the **greater the weight of future reinforcements.**

The weighting of future reinforcements has a half-life of $\sigma = \log 0.5 / \log \gamma$.

For $\gamma=0$, the value of a state is based exclusively on the immediate reinforcement received for performing the associated action.

Finite Horizon MDP

For finite horizon Markov decision processes (an MDP that terminates) it is not strictly necessary to use a discount factor.

In this case (γ=1), the value of state $\mathbf{x}_\mathrm{t}$ is based on the total reinforcement received when starting in state $\mathbf{x}_\mathrm{t}$ and following the given policy.

$$\Delta\mathbf{w}_t = \max_\mathrm{u}(r(\mathbf{x}_t,\mathbf{u})+\gamma V(\mathbf{x}_{t+1},\mathbf{w}_t))-V(\mathbf{x}_t,\mathbf{w}_t)$$

Infinite Horizon MDP

In the case of infinite horizon Markov decision processes (an MDP that never terminates), a discount factor is required.

Without the use of a discount factor, the sum of the reinforcements received would be infinite for every state.

The use of a discount factor limits the maximum value of a state to be on the order of $R/1-\gamma$.

Reward Functions

Discounted cumulative reward

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

Where r_{t+i} is generated by beginning at state s_t and repeatedly using policy π to select actions

$0 \leq \gamma < 1$ is a constant that determines the relative value of **delayed versus immediate** rewards - if $\gamma=0$ only immediate reward is considered, as γ moves closer to 1 future rewards are given more emphasis

Other Reward Functions

Finite horizon reward

$$\sum_{i=0}^{h} r_{t+i}$$

Average reward

$$\lim_{h \to \infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+1}$$

We will only focus on discounted cumulative reward!

Start again

Reinforcement Learning Problems

Delayed Reward

Exploration versus Exploitation

Partially Observable States

Life-long Learning

Delayed Reward

$\pi: S \rightarrow A$ that outputs an appropriate action, a, from the set A, given the current state s from the set S.

Delayed Reward: no training example in $<s,\pi(s)>$ form, the trainer provides only a **sequence of immediate reward values** as the agent executes its sequence of actions. The agent faces the problem of **temporal credit assignment**

Exploration versus Exploitation

The agent influences the distribution of training examples by the action sequence it chooses. Which experimentation strategy produces most effective learning?

The learner faces tradeoffs in choosing **exploration** of unknown states or **exploitation** of known states that it has already learned will yield high rewards

Partially Observable States

In many practical situations sensors only provide **partial information**.

An agent may have to consider its previous observations together with its current sensor data.

The best policy may be one which chooses specifically to **improve the observability** of the environment.

Life-long Learning

Agents often require that the robot **learn several related tasks** within the same environment.

A robot might need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pickup output from laser printers.

This raises the possibility of **using previously obtained experience** or knowledge to **reduce sample complexity** when learning new tasks.

Relationship to Dynamic Programming

Agent possesses perfect knowledge of the functions δ(s,a) and r(s,a)

Focused on how to compute the optimal policy with the least computational effort, assuming the environment can be simulated

Q learning has **NO knowledge** of the functions δ(s,a) and r(s,a)

Focus of Reinforcement Learning

Focused on the number of real-world actions the agent must perform to converge to an acceptable policy

In many practical domains, such as manufacturing problems, the costs in dollars and time of performing actions in the external world dominate computational costs

Summary

Reinforcement learning - **learning control strategies** for autonomous agents. Training information is real-valued reward for each state-action transition. Learn action policy that maximizes total reward from any starting state.

Reinforcement learning algorithms fit **Markov decision processes** where the outcome of applying an action to a state depends only on this action and state (not preceding actions or states). MDPs cover a wide range of problems - robot control, factory automation, and scheduling problems.

Summary II

Q learning is one form of reinforcement learning where the function Q(s,a) is defined as the **maximum expected, discounted, cumulative reward** the agent can achieve by applying action a to state s. In Q learning no knowledge of how the actions effect the environment is required.

Q learning is **proven to converge** under certain assumptions when the hypothesis Qˆ(s,a) is represented by a lookup table. Will converge deterministic and nondeterministic MDPs, but requires thousands of training iterations to converge in even modest problems.

Summary III

Q learning is a member of the class of **temporal difference algorithms**. These algorithms learn by iteratively reducing discrepancies between estimates produced by the agent at different times.

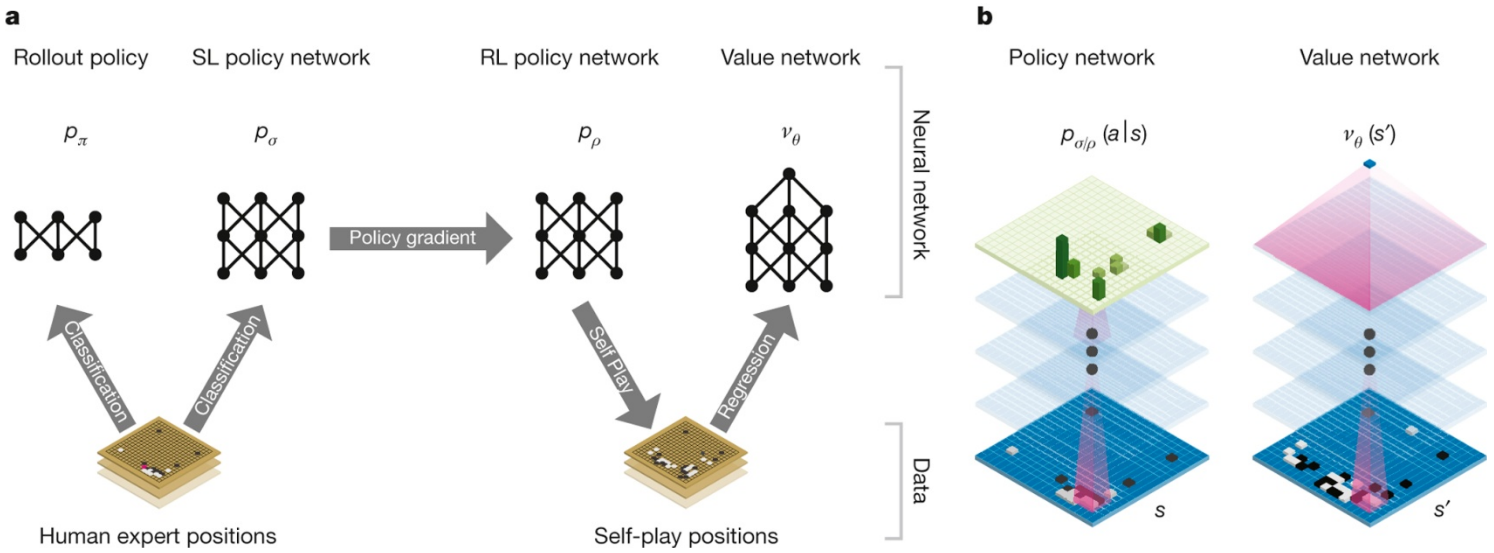
Reinforcement learning is closely related to **dynamic programming**. The key difference is that dynamic programming assumes the agent possesses knowledge of the functions δ(s,a) and r(s,a) while Q learning assumes the learner lacks this knowledge.

References

- Reinforcement Learning: A Tutorial, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.2480
- Reinforcement Learning: A Tutorial Survey and Recent Advances, http://web.mst.edu/~gosavia/joc.pdf
- Machine Learning, Tom Mitchell, (chapter on reinforcement learning)

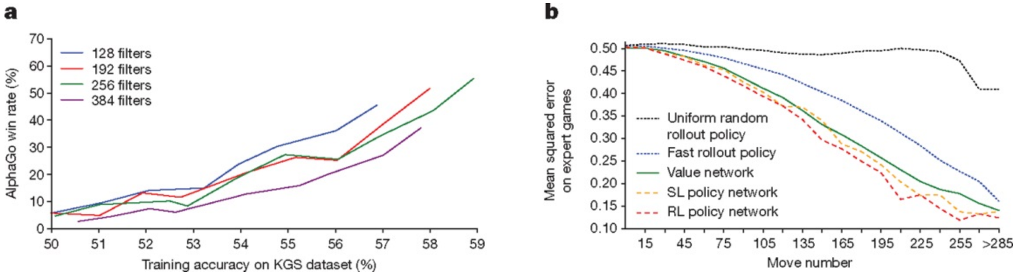
Alpha-Go

Neural network training pipeline and architecture



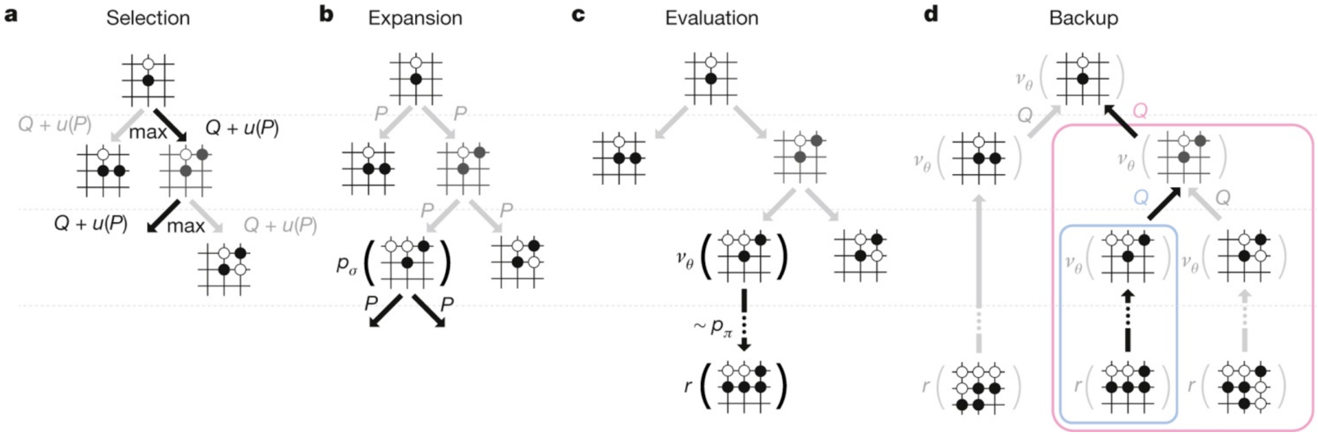
D Silver *et al. Nature* **529,** 484–489 (2016) doi:10.1038/nature16961

Strength and accuracy of policy and value networks



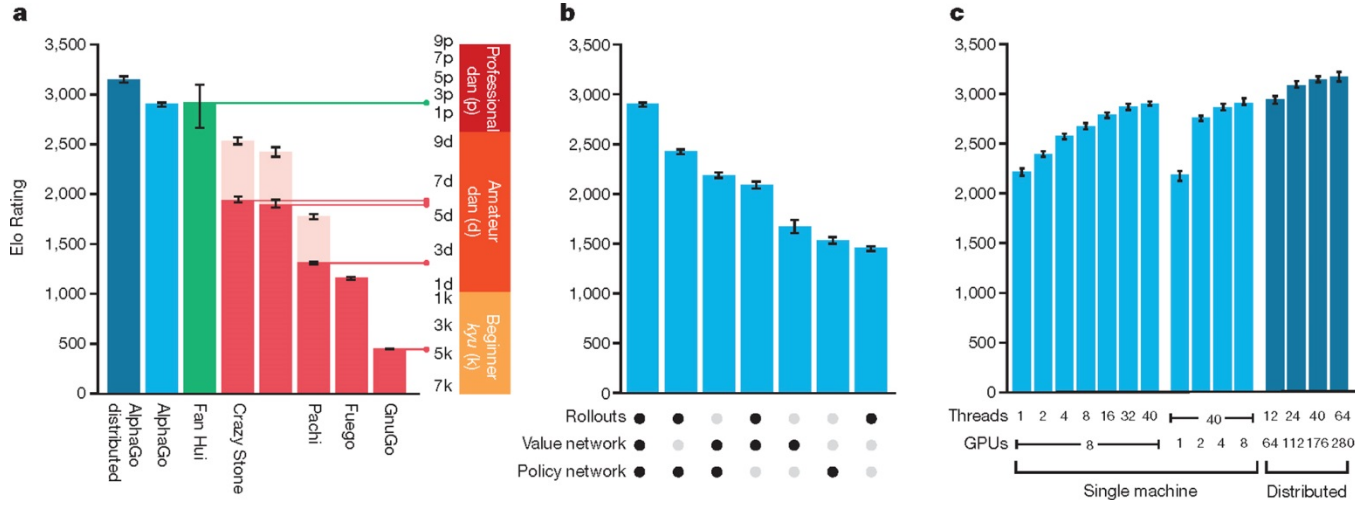
D Silver *et al.* *Nature* **529,** 484–489 (2016) doi:10.1038/nature16961

Monte Carlo tree search in AlphaGo



D Silver *et al. Nature* **529,** 484–489 (2016) doi:10.1038/nature16961

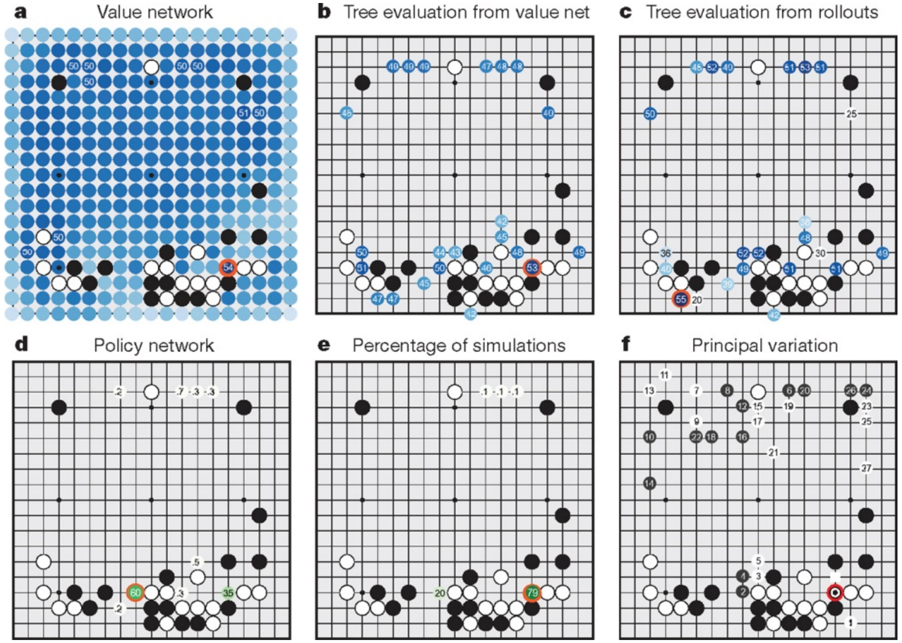
Tournament evaluation of AlphaGo



D Silver *et al.* *Nature* **529,** 484–489 (2016) doi:10.1038/nature16961

nature

How AlphaGo (black, to play) selected its move in an informal game against Fan Hui


a
Value network
b
Tree evaluation from value net
c
Tree evaluation from rollouts
d
Policy network
e
Percentage of simulations
f
Principal variation

D Silver *et al. Nature* **529,** 484–489 (2016) doi:10.1038/nature16961

Questions you should be able to answer

- How is reinforcement learning different than dynamic programing?
- What are the 3 main components of a reinforcement learning function?
- What are the major differences between Q learning and value iteration?