Reinforcement Learning

Kristian Wichmann

February 5, 2018

1 Basics of reinforcement learning

Reinforcement learning deals with an *agent* making *decisions* over time in dealing with some *environment*, to maximize some cumulative, scalar *reward*.

An example would be playing a video game. Here, the player is the agent. The control inputs are the decisions. The game and its internal logic is the environment. And the score is the reward.

Sometimes, a reward will be greater if it is postponed - it is not always advantagegous to reap any immediate reward. In other words, reinforcement learning algorithms will not benefit from being greedy.

1.1 The reward hypothesis

The basis of all reinforcement learning is the *reward hypothesis*, which states:

> All goals can be described by the maximization of some cumulative, expected reward.

1.2 Observation, action, and reward

At at given timestep t, a reinforcement learning agent gets some input; an *observation* O_t about the environment. It then takes an *action* A_t. And finally it gets a scalar *reward* R_t.

1.3 History and state

At any given time step t, the agent has a *history* H_t, which simply consist of all the observations, actions, and rewards that have happened so far:

$$H_t = O_1, A_1, R_1, O_2, A_2, R_2, \cdots, O_t, A_t, R_t, \tag{1.1}$$

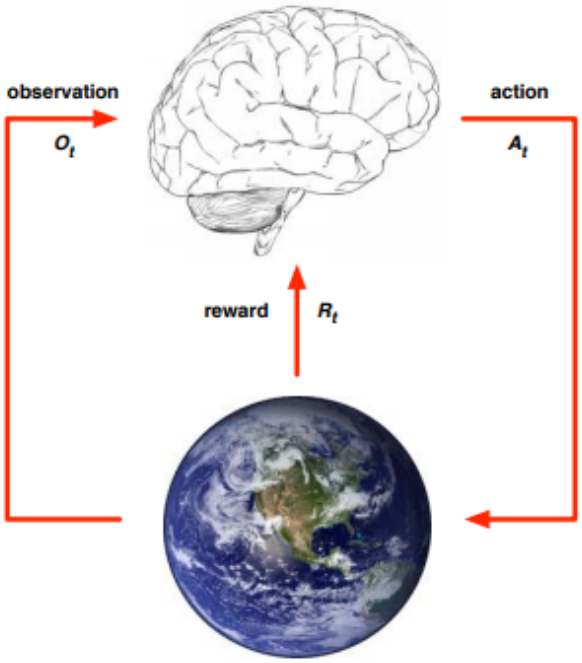

Figure 1: The interplay between agent and environment.

A *state* is a way for to parse this history into a more meaningful form. Formally, the state representation is simply a function of the history:

$$S_t = f(H_t) \tag{1.2}$$

It is very important to note, that this is generally distinct from the *environment state* H_t^e. The environment state contains complete information about the environment, including data and mechanisms that may be hidden to the agent. Hence H_t^e can depend on other things than just the historySuch information is *private* to the environment.

The *agent state* S_t^a on the other hand is the internal representation the agent uses to decide on which actions to take. Once again, it can be any function of history:

$$S_t^a = f(H_t) \tag{1.3}$$

1.4 Markov states

A state S_{t+1} is called a *Markov state* or an *information state* if it only depends on the state of the previous time step. Expressed probabilistically:

$$\mathbb{P}[S_{t+1}|S_1, S_2, \cdots S_t] = \mathbb{P}[S_{t+1}|S_t] \tag{1.4}$$

In other words, we don't need the entire history to decide on an action: Knowing the present is enough. Or put another way:

> The future is independent of the past, given the present.

Or:

> The state is a *sufficient statistic* of the future.

The environment state is always Markov, as by definition it contains all information about what can happen next. Similarly, the state consisting of the entire history is trivially Markov as well.

1.5 Full observability

This is the case where, in fact, we can observe everything about the environment and its inner workings. So that:

$$O_t = S_t^a = S_t^e \tag{1.5}$$

Sometimes this is reasonable. Sometimes not. But it will be a useful theoretical situation. This is known as a *Markov decision process* or MDP for short.

When this condition is not fulfilled, we speak about *partial observability* or a *partially observable evironment*. Here the agent only indirectly observes the invironment. This situation is known as a *partially observable Markov decision process* or POMDP for short.

1.6 State example: Bayesian beliefs

This state representation can be seen as a current best bet at what the actual environment state is. In other words, it is represented by Bayesian probabilities, which may then be updated over time using Bayes' rule:

$$S_t^a = (\mathbb{P}[S_t^e = s_1], \mathbb{P}[S_t^e = s_2], \cdots, \mathbb{P}[S_t^e = s_n]) \tag{1.6}$$

1.7 State example: Recurrent neural net

Here, the state S_t^a is a linear combination of the observation O_t and the state of the last time step S_{t-1}^a, followed by a non-linear *activation function* σ:

$$S_t^a = \sigma(W_O O_t + W_S S_{t-1}^a) \tag{1.7}$$

Here, the W's are weight matrices with sizes corresponding to the dimenstionality of observations and state vectors. Typical choices for σ are sigmoid, tanh, or rectified linear unit.

2 Components of an agent

A reinforcement learning agent may contain one or more of the following components:

- *Policy*: The agent's behaviour function. Shows how the agent gets from its state to deciding on an action.
- *Value function*: A measure of how desirable it is to be in a given state, or perform a given action.
- *Model*: The agent's representation of the environment.

We'll examine each of these in greater detail below:

2.1 Policy

A policy π is a mapping from state to action. For a *deterministic* policy, this is an ordinary function:

$$\pi(s) = a \tag{2.1}$$

So the state s is always mapped to the action a. We should ideal choose π so that the reward is maximized.

But a policy can also be *stochastic*, i.e. probabilistic. In this case π takes the form of conditional probabilities:

$$\pi(a|s) = \mathbb{P}[A = a|S = s] \tag{2.2}$$

2.2 Value function

A value function V_π is a prediction of the future reward for state s under a given policy π:

$$V_\pi(s) = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots | S_t = s] \tag{2.3}$$

Here γ is a *discounting factor*, between 0 and 1. 0 means that we only care about the immediate reward, while values approaching 1 means that we care progressively more about long run rewards.

2.3 Model

In this context, a model tries to predict the evolution of the environment. These can take two different forms:

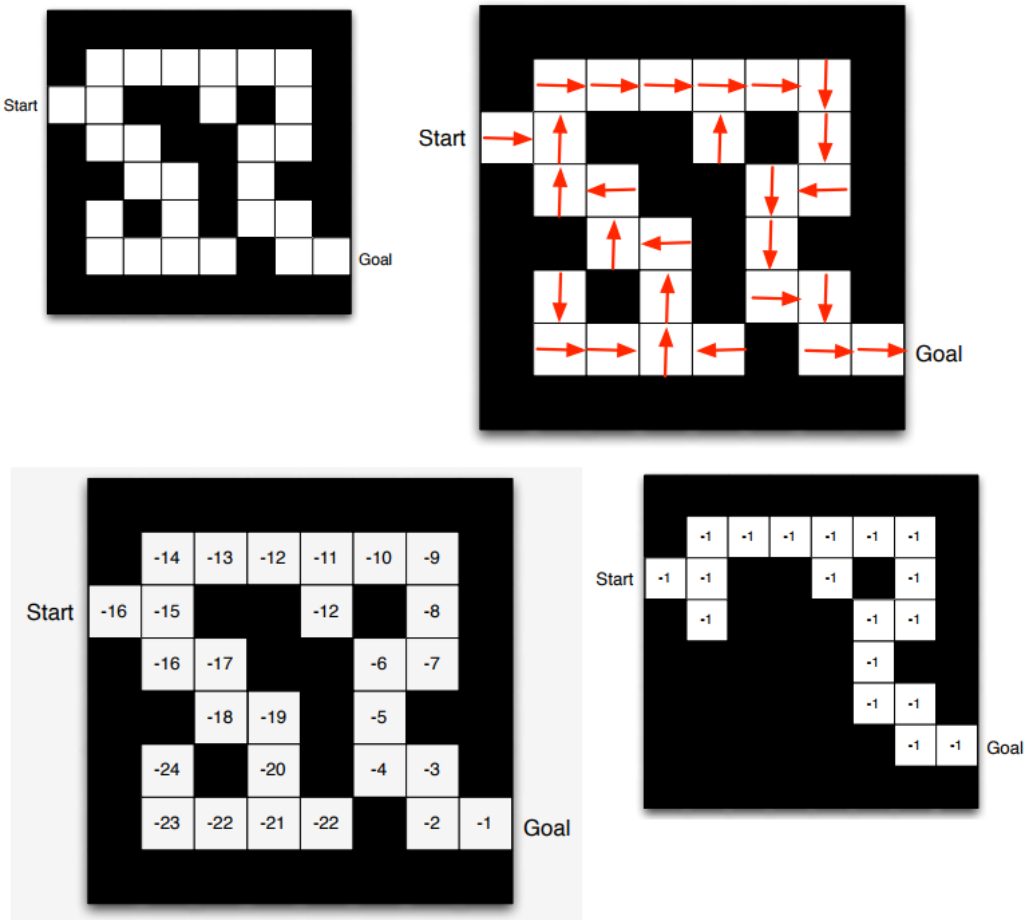

Figure 2: Upper left: Maze example. Upper right: Policy for maze. Lower left: Value function for maze. Lower right: Reward prediction for maze.

- *Transition prediction* $\mathcal{P}$ models the change of state of the environment:

$$\mathcal{P}^a_{ss'} = \mathbb{E}[S_{t+1} = s' | S_t = s, A_t = a] \tag{2.4}$$

- *Reward prediction* $\mathcal{R}$ models the change in (immediate) reward:

$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] \tag{2.5}$$

2.4 Example: Maze

As an example, consider an agent that has to find its way though the maze in the upper left of figure 2. The problem has the following qualities:

- Rewards: We want the agent to get through the maze as quickly as possible, so each time step has a reward of -1.
- Actions: The agent can move up, down, left, or right in each time step.
- State: The state is the agents current position square in the maze.

A good policy for solving the problem is shown in the upper right. Here the arrows dictates the action given the state (position).

The lower left shows the value function corresponding to the policy: The future reward is negative the number of steps needed to get out of the maze from the current state (position). An optimal action can be chosen greedily, by picking the possibility with the lowest value of the function.

The lower right shows a reward prediction model for the maze. This one technically depends on both state and action, but as we no it is -1 no matter what.

2.5 Categorization of agents

Based on the above, we can broadly categorize agents into several categories:

- *Value based agents* are based on a value function - with the policy being implicit. I.e. in each step we can greedily pick the action with the highest future reward, based directly on the value function.
- *Policy based agents* on the other hand has the policy as its key ingredient. Here, the policy mapping directly shows what action to take in each state.
- *Actor critic agents* takes both a poicy and a value function into account. Ideally the "best of both worlds".
- *Model free agents* may by policy and/or value function based, but has no model! I.e. it doesn't try to make predictions about the environment.
- *Model agents* may by policy and/or value function based, but includes modelling.

3 Reinforcement learning and planning

There's two fundamental types of problems in sequential decision making: *Reinforcement learning* and *planning*.

Reinforcement learning can be viewed as dumping the agent into an alien environment it initially knows nothing about. The agent then begins to interact with the environment, eventually improving its policy for doing so.

Planning is the situation where the agent starts with a perfect model of the environment. The agent then uses this model to make calculations without any external interaction, which then informs policy.

So in planning, we can in principle look many steps ahead, determining states and rewards (or expectation values thereof) and use this to pick optimal actions. In other words, a tree search.

3.1 Exploration versus exploitation

So reinforcement learning is very much a trial-and-error process. The agent should hopefully learn an effective policy from interacting with the environment. This requires *exploration*. Ideally without losing too much reward along the way.

Exploration finds more information about the environment.

Planning, on the other hand relies on *exploitation* of the complete knowledge of the environment.

Exploitation uses known information to maximize reward.

Usually, a combination of exploration and exploitation is necessary for achieving success. So there's a balance between the two, sometimes knows as *exploration-exploitation tradeoff*. This consideration is unique to reinforcement learning as opposed to machine learning as a whole.

3.1.1 Examples

Situation: You're going out to eat.

- Exploitation: Going to your favorite restaurant.
- Exploration: Trying a new restaurant.

Situation: Online banner advertisement.

- Exploitation: Show the most succesful advert so far.
- Exploration: Display a different advert.

Situation: Drilling for oil.

- Exploitation: Drill at the best known location.
- Exploration: Drill at a new location.

3.2 Prediction and control

Prediction is trying out a given policy in practice and evaluating the results.

Control on the other hand is optimizing the future, i.e. finding the best strategy.

Usually in reinforcement learning, we have to do prediction in order to control.

4 Markov reward processes

Recall that a Markov decision process (or MDP for short) is a case where everything about the environment is known - the agent state is the same as the environment state. This also means that it has the Markov property.

It turns out that almost all reinforcement learning problems can be restated as MDP's. For instance:

- Optimal control primarily deals with continuous MDP's.
- Partially observable problems can be conversted into MDP's.
- *Bandits* are MDP's with one state.

In this section, we will deal with a simpler case - known as Markov reward processes - and the tackle the general problem later.

4.1 Transition probability matrix

Consider for a moment an agent with a state having the Markov property, which simply takes the same (or no) action in every time step. Then given the initial state s there is a probability it will end up in state s' in the next time step - it can depend only on s and s' per the Markov property:

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s] \tag{4.1}$$

These propabilities form a matrix; the *transition probability matrix* $\mathcal{P}$:

$$\mathcal{P} = \begin{pmatrix} \mathcal{P}_{11} & \mathcal{P}_{12} & \cdots & \mathcal{P}_{1n} \\ \mathcal{P}_{21} & \mathcal{P}_{22} & \cdots & \mathcal{P}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \mathcal{P}_{n2} & \cdots & \mathcal{P}_{nn} \end{pmatrix} \tag{4.2}$$

This form assumes a finite number of states n, but in principle this can be infinite.

Because the entries are probabilites, each row of $\mathbb{P}$ must sum to 1:

$$\forall i \in \{1, 2, \ldots, n\} : \sum_{j=1}^{n} \mathcal{P}_{ij} = 1 \tag{4.3}$$

Such a system - represented by the tuple $\langle S, \mathcal{P} \rangle$ - is known as a *Markov process* or *Markov chain*.

4.2 Markov reward processes

A *Markov reward process* (or MRP) is a Markov process with a finite set of states and a *reward function* $\mathcal{R}$ as well as a discount factor $\gamma \in [0, 1]$. The reward function is defined as the expectation value of the next reward given the current state:

$$\mathcal{R} : s \mapsto \mathcal{R}_s = \mathbb{E}[R_{t+1} | S_t = s] \tag{4.4}$$

So a Markov reward process is the tuple $\langle S, \mathcal{P}, \mathcal{R}, \gamma \rangle$

4.3 Return and value function

The *return* of a given outcome of a Markov reward process[1] is:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{n=1}^{\infty} \gamma^{n-1} R_{t+n} \tag{4.5}$$

In other words, the return is the discounted future reward. This is the quantity we wish to maximize.

The value function, as we've already discussed above, is the expectation value of the return:

$$v(s) = \mathbb{E}[G_t | S_t = s] \tag{4.6}$$

[1] Or of any system with rewards and a discounting factor γ, really.

4.4 The Bellman equation for MRP's

We can now use equations 4.6, 4.5, and 4.4 to get a recursive formula for the value function:

$$
\begin{aligned}
v(s) =& \mathbb{E}[G_t|S_t = s] = && (4.7)\\
& \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots |S_t = s] = && (4.8)\\
& \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots)|S = s] = && (4.9)\\
& \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] = && (4.10)\\
& \mathbb{E}[R_{t+1}|S_t = s] + \gamma \mathbb{E}[G_{t+1}|S_t = s] = && (4.11)\\
& \mathcal{R}_s + \gamma \sum_{s'=1}^{n} \mathbb{P}[G_{t+1}|S_{t+1} = s']\mathbb{P}[S_{t+1} = s'] = && (4.12)\\
& \mathcal{R}_s + \gamma \sum_{s'=1}^{n} \mathcal{P}_{ss'} v(s') && (4.13)
\end{aligned}
$$

Here, $\mathcal{R}_s$ is the reward for leaving state s - which might seem somewhat backwards. Summing it up:

$$v(s) = \mathcal{R}_s + \gamma \sum_{s'=1}^{n} \mathcal{P}_{ss'} v(s') \tag{4.14}$$

Treating the value and reward functions as vectors v and $\mathcal{R}$, this can be rewritten in matrix form:

$$v = \mathcal{R} + \gamma \mathcal{P} v \tag{4.15}$$

This equation can be solved using the usual methods:

$$v = \mathcal{R} + \gamma \mathcal{P} v \Leftrightarrow (I_n - \gamma \mathcal{P})v = \mathcal{R} \Leftrightarrow v = (I_n - \gamma \mathcal{P})^{-1}\mathcal{R} \tag{4.16}$$

This direct solution however, is only tractable for small n, as the run time for matrix inversion is $O(n^3)$. Instead, there is a number of other, faster possibilities, including:

- Dynamic programming.
- Monte-Carlo evaluation.
- Temporal difference learning.

4.4.1 Intuition behind the Bellman equation for MRP's

What is the intuition behind equation 4.14? Imagine the agent is in state s. We wish to calculate the expectation of the future discounted reward. First we leave s which gives an immediate reward of $\mathcal{R}_s$. To get the future reward we have to make a weighted average over all the possible states we can go to from s', each having a future reward of $v(s')$. Finally this is discounted by multiplying with γ.

5 Markov decision processes

Now for the full MDP treatment. This case is very similar to the MRP case, but now there is several different actions to choose from, collectively written as $\mathcal{A}$. To each action $a \in \mathcal{A}$, corresponds a transition probability matrix $\mathcal{P}^a$. So formally, a Markov decision process is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of actions.
- $\mathcal{P}^a$ is a probability transition matrix corresponding to the action $a \in \mathcal{A}$.
- $\mathcal{R}_s^a$ is the expected reward for taking action $a \in \mathcal{A}$ when in state $s \in \mathcal{S}$.
- γ is the discounting factor.

5.1 Policy

Recall that a policy π is a way to choose an action $a \in \mathcal{A}$ given the state $s \in \mathcal{A}$:

$$\pi(a|s) = \mathcal{P}[A_t = a | S_t = s] \tag{5.1}$$

Theorem 5.1. *Given a Markov decision process $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π for it, then:*

- *The state sequence $S_1, S_2, \ldots$ is a Markov process with probability transition matrix:*

$$\mathcal{P}_{ss'}^{\pi} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^a \tag{5.2}$$

- *The combined state/reward sequence $S_1, R_1, S_2, R_2, \ldots$ is a Markov reward process with reward function:*

$$\mathcal{R}_s^{\pi} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_s^a \tag{5.3}$$

Proof. This follows from the law of total probability. □

This means that any MDP with a policy can always be reduced to a MRP, should we so wish.

5.2 State- and action-value function

The *state-value function* $v_\pi(s)$ of a MDP is the expected return starting from state s and then following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{5.4}$$

The *action-value function* $q_\pi(s, a)$ is the expected return starting in state s, taking action a and then following policy π, i.e.:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{5.5}$$

5.3 Bellman's expectation equations

It turns out that state- and action-value functions obey recursive Bellman equations as well. First the state-value function, where we explicitly write out the sum over the possibilities of the first action of the π policy:

$$\begin{aligned} v_\pi(s) =& \mathbb{E}_\pi[G_t|S_t = s] = & (5.6)\\ & \sum_{a\in\mathcal{A}} \mathbb{P}[S_t = s, A_t = a]\mathbb{E}_\pi[G_t|S_t = s, A_t = a] = & (5.7)\\ & \sum_{a\in\mathcal{A}} \pi(a|s)q_\pi(s, a) & (5.8)\end{aligned}$$

So the state-value function involved the action-value function. Let's examine the latter:

$$\begin{aligned} q_\pi(s, a) =& \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = & (5.9)\\ & \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots |S_t = s, A_t = a] = & (5.10)\\ & \mathbb{E}_\pi[R_{t+1}|S_t = s, A_t = a]+ & (5.11)\\ & \quad \gamma\mathbb{E}_\pi[R_{t+2} + \gamma R_{t+3} + \cdots |S_t = s, A_t = a] = & (5.12)\\ & \mathcal{R}^a_s + \gamma\mathbb{E}_\pi[G_{t+1}|S_t = s, A_t = a] & (5.13)\end{aligned}$$

Now, write out the sum over that state at $t + 1$:

$$\begin{aligned} q_\pi(s, a) =& \mathcal{R}^a_s + \gamma \sum_{s'\in\mathcal{S}} \mathbb{P}[S_{t+1} = s', S_t = s, A_t = a]\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] = & (5.14)\\ & \mathcal{R}^a_s + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'} v_\pi(s') & (5.15)\end{aligned}$$

This means that we have a set of coupled equations:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a), \quad q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \tag{5.16}$$

Inserting the latter into the former we get an equation for state-value function only:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right] \tag{5.17}$$

Similarly, we can get an equation for action-value function by inserting the former into the latter:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \left[\sum_{a'\in\mathcal{A}} \pi(a'|s') q_\pi(s',a') \right] \tag{5.18}$$

Do note the renaming of indices, though.

5.3.1 Intuition behind the Bellman expectation equations

What is the intuition behind the two equations 5.16?

Consider first the state-value function $v_\pi(s)$. Here, imagine the agent starting at state s. From there is can take a number of actions $a \in \mathcal{A}$, so we have to do a weighted average over the expected future rewards for each of these, the weights being $\pi(a|s)$. And the expected future rewards for being in state s and taking action a is exactly $q_\pi(s,a)$.

Consider then the action-value function $q_\pi(s,a)$. Here, the agent is in state s and takes action s. First, it gets the immediate reward $\mathcal{R}_s^a$ and then it goes to another state s', so we have to sum over these possibilities. The exptected future reward for being in state s' is $v_\pi(s')$. Finally, we must remember to discount the future reward by multiplying by γ.

5.4 Deterministic agents

A *deterministic* agent is one that always chooses a specific action $a(s)$ when in state s. Expressed in Kronecker delta form:

$$\pi(a|s) = \delta_{a,a(s)} \tag{5.19}$$

This means that the expectation equation for the state-value function simplifies:

$$v_\pi(s) = q_\pi(s, a(s)) \tag{5.20}$$

The intuition should be clear: The expected future reward in state s is simply the expected future reward for being in state s and taking action $a(s)$, since this is the only allowed action according to the policy.

We can now combine equation 5.20 with the action-value part of equation 5.16 to get:

$$v_\pi(s) = \mathcal{R}_s^{a(s)} + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^{a(s)} v_\pi(s') \tag{5.21}$$

5.5 Optimal value functions and policies

The *optimal state-value function* $v_*(s)$ is the maximum[2] when considering all possible policies:

$$v_*(s) = \max_\pi v_\pi(s) \tag{5.22}$$

Similarly, the *optimal action-value function* $q_*(s, a)$ is:

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{5.23}$$

Because both are optimal, we must have:

$$v_*(s) = \max_a q_*(s, a) \tag{5.24}$$

I.e. the best future reward for being in state s must be achieved by taking the optimal action a. This is known as *Bellman's optimality equation*.

We can define a partial ordering on policies as follows:

$$\pi \geq \pi' \Leftrightarrow \forall s \in \mathcal{S} : v_\pi(s) \geq v_{\pi'}(s) \tag{5.25}$$

Theorem 5.2. *For a Markov decision process, there exists an optimal policy π_*, i.e. on that is better than or as good as an arbitrary policy:*

$$\forall \pi : \pi_* \geq \pi \tag{5.26}$$

All such optimal policies achieves the optimal state-value and action-value function:

$$v_{\pi_*} = v_*, \quad q_{\pi_*} = q_* \tag{5.27}$$

Proof. We can construct such a policy by:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname*{argmax}_{a \in \mathcal{A}} q_*(s, a) \\ 0 & \text{otherwise} \end{cases} \tag{5.28}$$

Since we must have $v_*(s) = \max_a q_*(s|a)$ it follows that this policy achieves both optimal state-value and action-value function. □

[2] Or supremum, to be sure is exists.

This shows that not only does an optimal policy exist, but a deterministic, optimal policy exists!

5.6 Bellman's optimality equations

As noted above, Bellman's optimality equation is expressed in 5.24. We can use this along with the right hand side of equation 5.16 to get:

$$v_*(s) = \max_a \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right] \tag{5.29}$$

Or we can write the same equation, but this time for $q_*(s, a)$:

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \left[\max_{a'} q_*(s'|a') \right] \tag{5.30}$$

These non-linear equations are known as *Bellman's optimality equations*. In general these have no closed-form solution. Instead, typically iterative methods are used for solving them. Such strategies include:

- Value iteration.
- Policy iteration.
- Q-learning.
- Sarsa.

These will be examined in the following.

6 Dynamic programming

Dynamic programming is a method for solving complex problems by breaking them down into subproblems, solving these and putting them back together to solve the overall problem.

In order for dynamic programming to be applicable, the problem must have two basic properties: *optimal substructure* and *overlapping subproblems*.

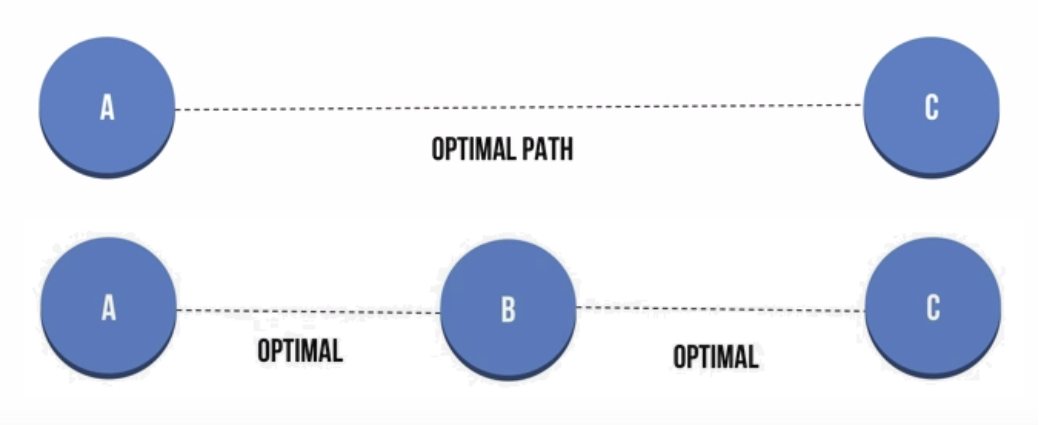

Figure 3: The principle of optimality shown for the case of finding optimal paths between points. The principle states, that we can find the optimal path from A to C (above) if we know the optimal paths from A to B and B to C (below).

6.1 Optimal substructure

This is also known as *the principle of optimality*. Assume we have an optimal solution for the case (A, C). Then that optimal solution should be obtainable from having the optimal solutions for the cases (A, B) and (B, C). Looking at figure 3 where the principle is illustrated for the problem of finding optimal paths.

Since the principle can be applied iteratively, eventually the problem can be reduced to a set of "smallest", subsequent subproblems. Because of this, mathematical induction is often used to prove a given problem has optimal substructure.

6.2 Overlapping subproblems

This means that each of the subproblems occur "many times" and that we therefore can cache the solutions from these so we don't have to redo the same problem over and over.

6.3 Example: Fibonacci numbers

As is well known, the Fibonacci numbers are defined recursively:

$$F_0 = 0, \quad F_1 = 1, \quad F_{n+1} = F_n + F_{n-1} \tag{6.1}$$

By this very definition, it is clear that if we know the all $F_i, i \leq n$ we can always find F_{n+1}, which means that the principle of optimality holds.

If we did this using only recursion, the fibonacci function would call itself a large number of times. We can prevent this by caching all F_n that is calculated. Thus the overlapping subproblem criterion is also satisfied.

7 The contraction mapping theorem

Before we continue exploring dynamc programming in the context of reinforcement learning, let's take a mathematical side trek which will be convenient for showing the convergence of certain algorithms.

Definition 7.1. *A contraction mapping T on the metric space (X, d) is a Lipschitz function $T \to T$ with Lipschitz constant $q \in [0, 1[$, i.e.:*

$$\forall x, y \in X : \ d(T(X), T(Y)) \leq q \cdot d(x, y) \tag{7.1}$$

Theorem 7.1. *(The contraction mapping theorem) Let T be a contraction mapping on the non-empty, complete metric space (X, d). Then T has a unique fixed-point x^*:*

$$\exists! \ x^* \in X : \ T(x^*) = x^* \tag{7.2}$$

This fixed-point can be found by iteration of T from any starting point $x_0 \in X$:

$$x_n = T(x_{n-1}), \quad (x_n) \underset{n\to\infty}{\to} x^* \tag{7.3}$$

Proof. Using the triangle inequality for any $x, y \in X$ we have:

$$d(x, y) \leq d(x, T(x) + d(T(x), T(y)) + d(T(y), y) \tag{7.4}$$
$$\leq d(x, T(x) + q \cdot d(x, y) + d(T(y), y) \tag{7.5}$$

Here we've used that T is a contraction. Isolate $d(x, y)$ to get:

$$d(x, y)(1 - q) \leq d(x, T(x) + d(T(y), y) \Leftrightarrow d(x, y) \leq \frac{d(x, T(x) + d(T(y), y)}{1 - q} \tag{7.6}$$

This is known as the *fundamental contraction inequality*. We can use it to prove that T has at most one fixed-point: Assume both x and y are fixed points. Then equation 7.6 says $d(x, y) \leq 0$, leaving a distance of zero as the only option, which means that $x = y$.

Now, consider the mapping $T^n : X \to X$ defined as repeated application of T n times. This mapping is also a contraction, with Lipschitz constant

q^n. We now wish to show that $(x_n) = (T^n(x_0))$ is Cauchy. We now apply the fundamental contraction inequality to x_m and x_n:

$$d(x_m, y_n) \leq \frac{d(x_m, T(x_m) + d(T(x_n), x_n)}{1-q} \tag{7.7}$$

$$= \frac{d(T^m(x_0), T(T^m(x_0)) + d(T(T^n(x_0)), T^n(x_0))}{1-q} \tag{7.8}$$

$$= \frac{d(T^m(x_0), T^m(T(x_0)) + d(T^n(T(x_0)), T^n(x_0))}{1-q} \tag{7.9}$$

$$\leq \frac{q^m \cdot d(x_0), T(x_0)) + q^n \cdot d(T(x_0), x_0)}{1-q} \tag{7.10}$$

$$= \frac{q^m + q^n}{1-q} d(x, T(x_0)) \tag{7.11}$$

Since $q < 1$, this can be made arbitrarily small by making m and n large enough. Hence (x_n) is Cauchy and therefore convergent, since X is complete. Let's call the limit x^*.

To show that x^* is a fixed-point consider the recursive definition of the series:

$$x_{n+1} = T(x_n) \tag{7.12}$$

Now apply the limit $n \to \infty$ on both sides of the equation:

$$\lim_{n\to\infty} x_{n+1} = \lim_{n\to\infty} T(x_n) \tag{7.13}$$

Since T is Lipschitz, it is also continuous, and so we may interchange limit and application of T:

$$\lim_{n\to\infty} x_{n+1} = T\left(\lim_{n\to\infty} x_n\right) \tag{7.14}$$

But we know that both these limits are equal to x^*, so:

$$x^* = T(x^*) \tag{7.15}$$

□

The theorem is also known as *Banach's fixed-point theorem.*

8 Dynamic programming for MDP's

It turns out that we can use dynamical programming for finding value functions for MDP's. Here, the recursive substructure is implied by Bellman's

optimality equations. And the value function can be cached as calculations are done. Hence, both criteria are satisfied.

This assumes perfect knowledge of the MDP, and therefore it can only be used for planning. There's essentially two types of question we can ask here:

- Prediction: Given a MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π, what is the corresponding state-value function $v_\pi(s)$?
- Control: Given a MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, what is the optimal state-value function $v_*(s)$?

We'll look at both below.

8.1 Iterative policy evaluation

Evaluation relies on prediction: Given a policy π we want to find the state-value function $v_\pi(s)$. One approach - indeed the one we will use in this section - is to obtain the value function iteratively through Bellman's expectation equation 5.17. Here, we start by a guess v_1 - or simply zeros - at the state-value function as a vector, and then insert it into the expectation equation to get v_2. Iterating this process, getting v_3, v_4, v_5 etc., v_n will eventually converge to v_π:

$$v_1 \to v_2 \to v_3 \to \cdots \underset{n\to\infty}{\to} v_\pi \tag{8.1}$$

We will prove this later.

The update is done using *synchronous backup*, which means that we store the values of v_i for all the states and use them to calculate the values for v_{i+1}. Only then are the "active" values used in the next step calculation changed.

8.1.1 Example: Starting with $v_1 = 0$

Assume out inital guess v_1 simply consists of all zeros. Then the second iteration is:

$$v_2(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_1(s') \right] = \tag{8.2}$$

$$\sum_{a\in\mathcal{A}} \pi(a|s) \mathcal{R}_s^a \tag{8.3}$$

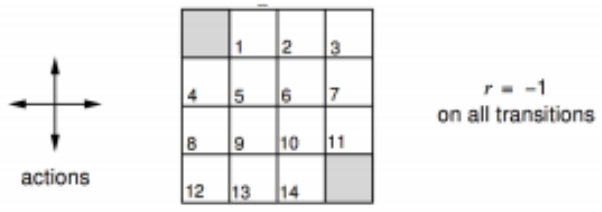

Figure 4: The gridworld environment. The grey areas make up the terminal state.

Similarly, the third iteration becomes:

$$v_3(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_2(s') \right] = \tag{8.4}$$

$$\sum_{a\in\mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a'\in\mathcal{A}} \pi(a'|s') \mathcal{R}_{s'}^{a'} \right] \tag{8.5}$$

And so on. Of course in practice this is done numerically.

8.1.2 Example: Gridworld

Consider the gridworld environment shown in figure 4. The agent's state is the square it occupies. The possible action are moving up, down, left, or right. The grey squares make up a terminal state. Trying to move across the outer border will make the agent stay where it is. In each step before it reaches the terminal area, the reward is -1.

The policy π we wish to examine is uniform random choice between the four available actions in any non-terminal state:

$$\pi(\uparrow|\cdot) = \pi(\rightarrow|\cdot) = \pi(\downarrow|\cdot) = \pi(\leftarrow|\cdot) = \frac{1}{4} \tag{8.6}$$

The left part of figure 5 shows how v_k changes as the iterative policy evaluation process is performed. Eventually it converges; at $k = \infty$ the number at each square shows (save for the sign) the average number of steps a random walker takes before getting to the terminal state.

8.2 Control: Policy iteration

Looking at the right hand side of 5, we see the *greedy policy* with respect to v_k. I.e. the policy that we choose the action for which the corresponding state

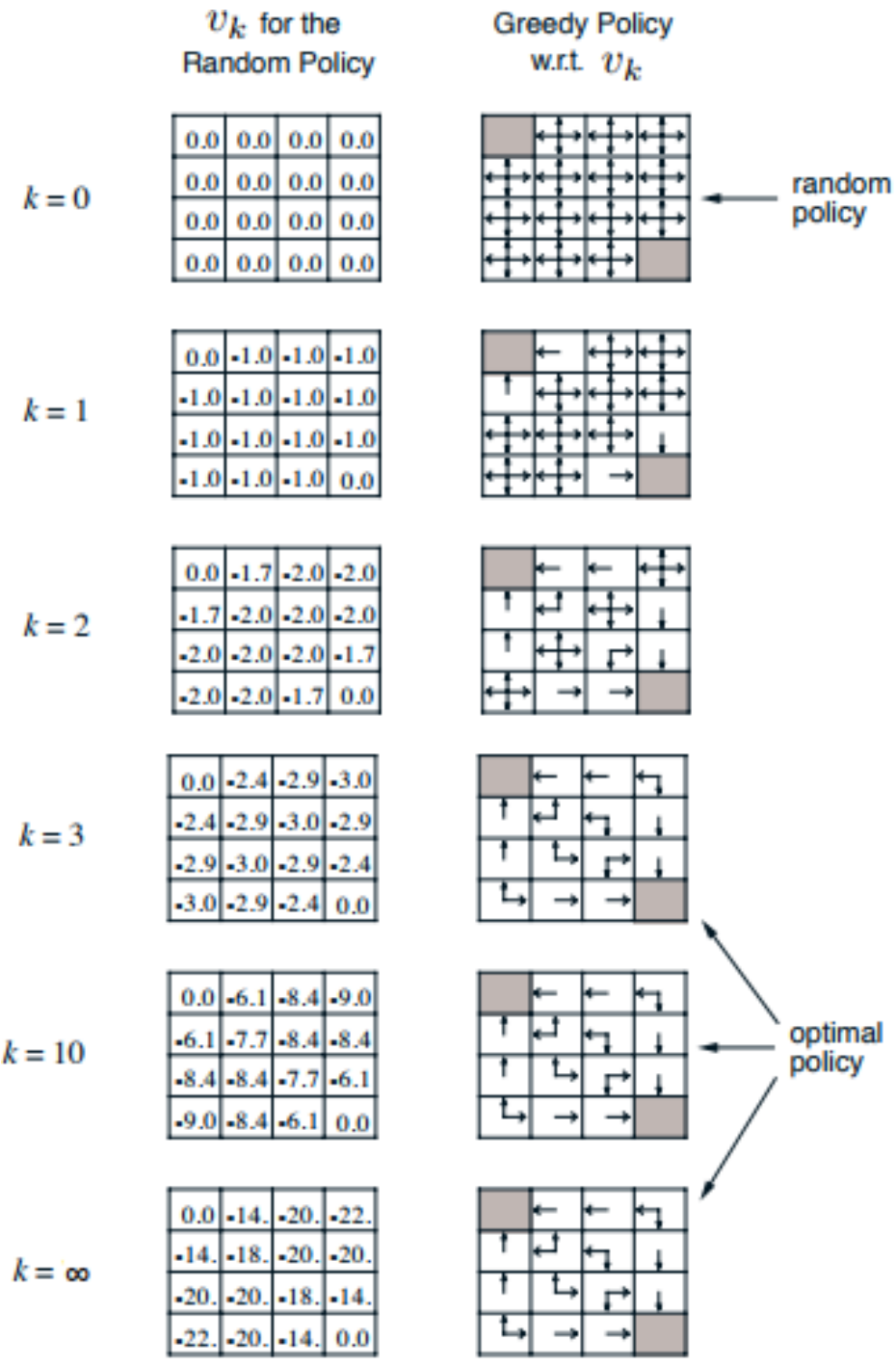

Figure 5: Left: Evolution of v_k starting at zero. Right: The greedy policy corresponding to v_k.

is maximal (or if there's a tie, randomize even between those with maximal state-value).

These policies seem to be doing rather well. In fact, after only three iterations, it has arrived at the optimal policy, even if it takes way longer for v_k to converge to v_π! This should give us an idea for how to improve our policy π:

1. Start with a policy π.
2. Find v_π using iterative policy evaluation.
3. Construct the policy $\pi' = \text{greedy}(v_\pi)$.

The new policy will always be better (or at least as good) as π:

$$\pi' \geq \pi \tag{8.7}$$

We will show this in the next section. If the process is applied iteratively, it will always converge to an optial policy π_*. This is known as *policy iteration*. Again, this will be shown in the next section.

For the gridworld example, only one such iteration was needed to reach an optimal policy. This is rarely the case for more complicated problems.

8.3 Convergence of iterative policy improvement

So why does all of this work? Let's start by considering only deterministic policies. Since greedy policies are deterministic (or can be made deterministic while retaining efficiency), this can be done without loss of generality.

So let π be a deterministic policy, i.e.:

$$\pi(s) = a(s) \tag{8.8}$$

Now make a new policy π' by being greedy with respect to v_π. Recall, that according to Bellman's expectation equation:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a) = q_\pi(s, a(s)) \tag{8.9}$$

Here we have used that π is deterministic. So being greedy corresponds to maximizing $q_\pi(a, s)$. This means:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\text{argmax}}\ q_\pi(s, a) \tag{8.10}$$

Why does this improve the policy? Let's see:

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) \geq q_\pi(s, a(s)) = v_\pi(s) \tag{8.11}$$

In the last equal sign, we used equation 8.9 again. Now we can use equation 8.11 and the definiton of q_π to show the desired result:

$$v_\pi(s) \leq q_\pi(s, \pi'(s)) = \mathbb{E}_{\pi'}[R_t + \gamma v_\pi(S_{t+1}) | S_t = s] \tag{8.12}$$

Here we've used that the expectation value of a constant is simply the constant itself, meaning that $\gamma v_\pi(S_{t+1}) = \mathbb{E}_{\pi'}[\gamma v_\pi(S_{t+1})]$. But now we may use 8.11 again:

$$v_\pi(s) \leq \mathbb{E}_{\pi'}[R_t + \gamma q_\pi(S_{t+1}, \pi'(s)) | S_t = s] \tag{8.13}$$

But now we can apply the same steps again to get:

$$v_\pi(s) \leq \mathbb{E}_{\pi'}[R_t + \gamma R_{t+1} + \gamma^2 v_\pi(S_{t+2}) | S_t = s] \tag{8.14}$$

This can be iterated indefinitely, so in the end we get:

$$v_\pi(s) \leq \mathbb{E}_{\pi'}[R_t + \gamma R_{t+1} + \gamma^2 R_{r+2} + \gamma^3 R_{r+3} + \cdots | S_t = s] = \tag{8.15}$$

$$\mathbb{E}_{\pi'}[G_{t+1} | S_t = s] = v_{\pi'}(s) \tag{8.16}$$

This shows that $\pi \leq \pi'$.

Now assume that we perform policy iteration until there is no further improvement. I.e. until:

$$v_\pi = v_{\pi'}, \quad q_\pi = q_{\pi'} \tag{8.17}$$

In this case, we can rewrite equation 8.11, but this time with an equal sign:

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) = q_\pi(s, a(s)) = v_\pi(s) \tag{8.18}$$

But this means that π satisfies the Bellman optimality equation:

$$v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a) \tag{8.19}$$

Hence π is optimal. This shows convergence of iterative policy improvement.

8.4 Modified policy iteration

As we noticed in the gridworld example, sometimes we don't really need the full convergence of v_* to get a good policy by acting greedy.

Modified policy iteration exploits this idea by only taking k steps in each policy evaluation loop.

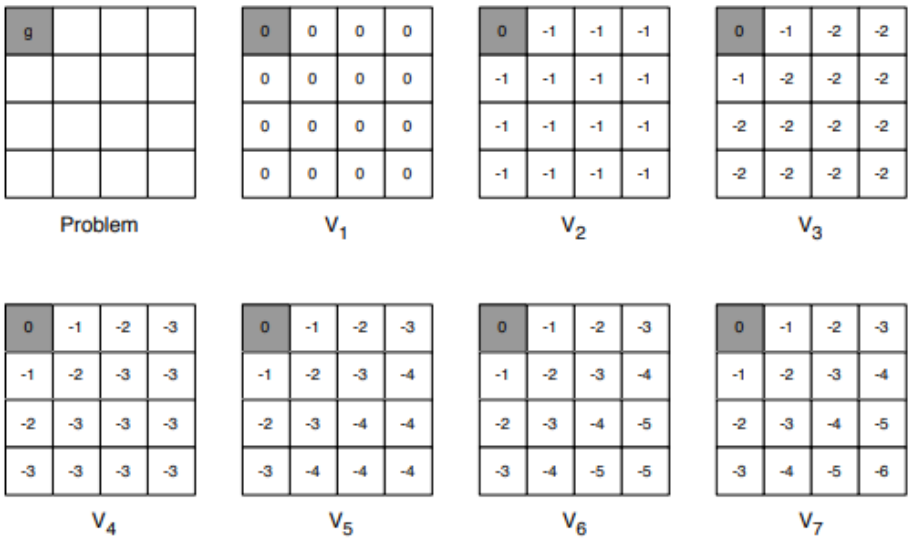

Figure 6: Left: Evolution of v_k starting at zero. Right: The greedy policy corresponding to v_k.

8.5 Value iteration

This method explicity exploits the principle of optimality: An optimal strategy for an agent in state s necesserily starts with a first optimal action a_*. If we know this, we can also choose the optimal action a'_* in any state s' the agent may end up in after taking action a_*. This is essentially what is expressed in Bellman's optimality equation:

$$v_*(s) = \max_a \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right] \tag{8.20}$$

Following the idea of iterative policy evaluation, we can start at some value of $v_*(s)$ and iterate this equation. This is known as *value iteration*.

Since we're essentially being greedy in every update step, this is equivalent to modified policy iteration with $k = 1$.

8.5.1 Example: More gridworld

As an example, consider the gridworld from earlier, except with only one terminal square. The possible actions at each non-terminal square is still the four directions. But this time, we search for a value function instead of evaluating a policy. Figure 6 shows the state-value function as value iteration

is performed starting from a uniform value of zero. We see that each step propagates the reward through the state space until convergence after 7 steps.