

CZECH TECHNICAL UNIVERSITY IN PRAGUE  
FACULTY OF ELECTRICAL ENGINEERING  
DEPARTMENT OF CYBERNETICS



---

# Demonstration of Learning in Games

with focus on Reinforcement Learning & Forgetting

---

BACHELOR THESIS

*Author:*  
Marek OTÁHAL

*Supervisor:*  
Ing. Lenka NOVÁKOVÁ, Ph.D.

May 2010



## BACHELOR PROJECT ASSIGNMENT

**Student:** Marek Otáhal  
**Study programme:** Software Engineering and Management  
**Specialisation:** Intelligent Systems  
**Title of Bachelor Project:** Demonstration of Learning in Games

### Guidelines:

1. Get acquainted with several possibilities how machine learning can improve computer strategies in games. Pay attention to reinforcement learning and to the role of forgetting during learning process.
2. Design and implement a simple computer game to demonstrate selected learning strategies.
3. Describe the implemented strategy and write a help for this game in form of educational material.

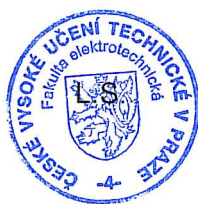
### Bibliography/Sources:


Millington I.: Artificial Intelligence for games, Morgan Kaufmann, 2006, ISBN 10: 0-12-497782-0

**Bachelor Project Supervisor:** Ing. Lenka Nováková, Ph.D.

**Valid until:** the end of the winter semester of academic year 2010/2011

  
prof. Ing. Vladimír Mařík, CSc.  
**Head of Department**



  
doc. Ing. Boris Šimák, CSc.  
**Head**

Prague, February 3, 2010

## ZADÁNÍ BAKALÁŘSKÉ PRÁCE

**Student:** Marek Otáhal  
**Studijní program:** Softwarové technologie a management  
**Obor:** Inteligentní systémy  
**Název tématu:** Demonstrace učení v prostředí her

### Pokyny pro vypracování:

1. Seznamte se s možnými přístupy k učení ve hrách, zejména s posilovaným učením a s možností využití zapomínání během učení.
2. Navrhněte a implementujte jednoduchou hru, na které by bylo možné přístupy učení demonstrovat.
3. Vypracujte popis odpovídající teorii i dokumentaci hry ve formě materiálu vhodného pro podporu výuky.

### Seznam odborné literatury:

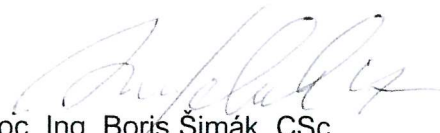
Millington I.: Artificial Intelligence for Games, Morgan Kaufmann, 2006, ISBN 10:0-12-497782-0

**Vedoucí bakalářské práce:** Ing. Lenka Nováková, Ph.D.

**Platnost zadání:** do konce zimního semestru 2010/2011



prof. Ing. Vladimír Mařík, DrSc.  
vedoucí katedry



doc. Ing. Boris Šimák, CSc.  
děkan

## Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v příloženém seznamu.

V Praze, dne 25.5.2010

  
Marek Otáhal

## Declaration

I hereby declare to have written the thesis by myself and mentioned all the help and sources used.

In Prague, May 25th, 2010

  
Marek Otáhal

# Acknowledgment

It is a pleasurable duty to thank all the people who helped me through the time I worked on my bachelor thesis. I want to thank Ing. Lenka Nováková, Ph.D., my supervisor. Her efforts to support me and willingness to help any time I needed it were the most valuable for me. It was also she who told me about reinforcement learning for the first time. I thank my family, they provided me with so much more than just food and shelter. I would also like to mention my friends who cheered me up at the times.

Thank you!

## **Keywords**

Artificial intelligence, AI, forgetting, memory, learning, reinforcement learning, RL, games, game AI, strategies, Q-learning

## Abstract

Reinforcement learning is a promising method for machine learning, especially the use in game AI for its ability to cope with complex problems that provide feedback only on rare occasions. Forgetting is presented as an optimization method to speed up the learning process and reduce memory requirements as well. In many cases, a lot of information is provided to the learner, but only some is useful for a solution of a given problem, forgetting is able to remove the unneeded information and may offer improved results for problems that couldn't be learned completely for time reasons. Classes for online visualization of the learning progress and examples are provided. An educational material was made for the purpose.

## Abstrakt

Posilované učení je slibná metoda strojového učení, která je díky schopnosti pracovat se složitými problémy, jenž informují o tom, jak si hráč (umělá inteligence) vede, jen zřídka. Díky tomu se skvěle uplatní v teorii her, kde tato situace poměrně často nastává. Jako způsob zlepšení rychlosti a paměťové náročnosti učení je představeno zapomínání. V mnoha případech má hráč (umělá inteligence) na výběr z velkého množství informací, ale jen malá část je podstatná pro řešení konkrétního problému, a právě zapomínání v některých případech dokáže rozpoznat a odstranit nepotřebné informace a zlepšit tak výsledky i pro problémy, které nemohly z časových důvodů být plně naučeny. Způsoby pro zobrazení průběhu učení a různé příklady jsou součástí zdrojových kódů. Také byl vytvořen studijní materiál na téma posilovaného učení.



# Contents

<b>Keywords</b>	<b>6</b>
<b>Abstract</b>	<b>7</b>
<b>1 Introduction</b>	<b>10</b>
<b>2 Theory of AI</b>	<b>11</b>
2.1 Defining intelligence . . . . .	11
2.2 Definitions of AI . . . . .	12
2.3 AI design . . . . .	13
<b>3 Uses of AI in real life</b>	<b>15</b>
3.1 Practical usage of AI nowadays . . . . .	15
3.1.1 Speech recognition & synthesis . . . . .	15
3.1.2 Computer vision . . . . .	16
3.1.3 Optimization . . . . .	16
3.1.4 Swarm intelligence . . . . .	16
3.1.5 Genetic programming . . . . .	16
3.1.6 Decision trees . . . . .	16
3.2 A closer look at the game industry . . . . .	17
3.2.1 Usage . . . . .	17
3.2.2 Top 8 most influential games using AI . . . . .	17
<b>4 Focus on Reinforcement Learning</b>	<b>21</b>
4.1 Learning . . . . .	21
4.2 Demo games . . . . .	22
4.3 Reinforcement learning . . . . .	23
4.3.1 Q learning . . . . .	23
4.3.2 Learning itself . . . . .	24
4.3.3 Exploration . . . . .	25
4.3.4 Rewards . . . . .	26

4.3.5	Convergence . . . . .	27
4.3.6	Pseudo-code . . . . .	27
4.3.7	Performance . . . . .	28
4.3.8	Fine tuning . . . . .	28
4.3.9	Modifications . . . . .	30
4.4	Different AIs . . . . .	30
<b>5</b>	<b>Forgetting</b>	<b>32</b>
5.1	How do we forget then? . . . . .	32
5.2	Forgetting in computer science . . . . .	32
5.2.1	Implementation for reinforcement learning . . . . .	33
<b>6</b>	<b>Conclusion</b>	<b>37</b>
<b>A</b>	<b>RL cheat sheet</b>	<b>39</b>
<b>B</b>	<b>Included materials</b>	<b>41</b>
	<b>References</b>	<b>44</b>

# Chapter 1

## Introduction

My bachelor thesis will cover a very interesting topic of *artificial intelligence* - usage of AI in games and the concept of forgetting. Main focus will be on *reinforcement learning (RL)* enhanced with the ability to forget. A short educational text on reinforcement learning - a *RL cheat sheet* is provided and can be found in [appendix A](#). The fundamental part will definitely be around the [source codes](#) where you can see & run demonstrative examples and have a look how the coding for RL is done. The RL code should be abstract enough, so if users have a problem and would wish to try solving it with RL, it ought to be easy to implement quickly.

# Chapter 2

## Theoretical introduction to intelligence & AI

Theoretical introduction to intelligence and **artificial intelligence (AI)** brings questions about what exactly might that intelligence be, how do we determine if someone/something is intelligent or is not? These are mostly philosophical questions and are important if we really wish to focus on the matter of *intelligence*. Someone might say that this is not important and that he/she just wants to make an intelligent program. That would be also true, but only if the task is to create some heuristics and thus improve the results. In case we want to achieve some more sophisticated intelligence or do a research on artificial intelligence, best source of inspiration would be in the nature - animals, babies or thinking about the ways we think. It can be a real fun, so let's jump right into it.

### 2.1 Defining intelligence

*“Viewed narrowly, there seem to be almost as many definitions of intelligence as there were experts asked to define it.”* R. J. Sternberg, in [Gre98]

For a start, we realize that mathematicians as well as programmers tend to love having everything well defined, a hundred percent sure and described by a clear logic. On the other hand, **AI** is inspired by nature, shares some knowledge with various scientific fields like psychology, biology, etc. The problem is we cannot give an exact definition for the two basic pillars of our interest: that would be *intelligence* and *artificial intelligence*.

I present some definitions of intelligence from various fields:  
(for the complete compilation, please see [LH07])

1. Collective

- *“The ability to acquire and apply knowledge and skills.”* [SW06]

- “...ability to adapt effectively to the environment, either by making a change in oneself or by changing the environment or finding a new one ...intelligence is not a single mental process, but rather a combination of many mental processes directed toward effective adaptation to the environment.” [bri06]

## 2. Psychological definitions

- “...adjustment or adaptation of the individual to his total environment, or limited aspects thereof ...the capacity to reorganize one’s behavior patterns so as to act more effectively and more appropriately in novel situations ...the ability to learn ...the extent to which a person is educable ...the ability to carry on abstract thinking ...the effective use of concepts and symbols in dealing with a problem to be solved ...” W. Freeman
- “Intelligence is not a single, unitary ability, but rather a composite of several functions. The term denotes that combination of abilities required for survival and advancement within a particular culture.” [Ana92]
- “...that facet of mind underlying our capacity to think, to solve novel problems, to reason and to have knowledge of the world.” [And06]

## 3. AI researchers

- *Intelligence means getting better over time.*” [Sch91]
- “Intelligent systems are expected to work, and work well, in many different environments. Their property of intelligence allows them to maximize the probability of success even if full knowledge of the situation is not available. Functioning of intelligent systems cannot be considered separately from the environment and the concrete situation including the goal.” [Gud00]

## My form of the definition would be:

“For some time interact with an environment in a way to achieve your goals, learn from the past actions and be able to create models in analogy to the observed ones”

## 2.2 Definitions of artificial intelligence

Trying to define what would be called an *artificial intelligence (AI)*, we encounter the same problems as above. The list of definitions is basically very similar to the definitions of intelligence. I pick an interesting one:

*“Here’s another ‘off-the-cuff’ definition of AI, but one which I think captures the essence of what separates AI CS from regular CS.*

*Artificial Intelligence is the branch of Computer Science that attempts to solve problems for which there is no known efficient solution, but which we know are efficiently solvable, (typically) because some intelligence can solve the problem (often in ‘real time’).*

*A side benefit of AI is that it helps us learn how intelligences solve these problems, and thus how natural intelligence works.*

*Example: vision. We do not have any algorithms for recognizing, say, animal faces in images, but we know it must be possible, because humans (even infants) can effectively recognize faces. Solving this problem would help us understand how human vision works.” [Leh85]*

*“The science of making machines do things that would require intelligence if done by humans.” Marvin Minsky, MIT CS professor*

From all that I conclude the requirements to make an AI be:

1. *environment* with laws/rules (which don’t change too often)
2. *sensors* to observe the surrounding environment
3. some means of *changing a situation*, taking actions (ie. legs, machine-gun, ...)
4. optionally some kind of *memory* to build & store a representation of the world

## 2.3 Approaches to designing systems with AI

Two possible theoretical approaches or programming paradigms are discussed in the following text. *Top-down* & *bottom-up* are two different approaches to information processing and knowledge management. The first can be viewed as *decomposition* of bigger problems to smaller ones, while providing more details on each level but covering a smaller area. The latter would be *synthesis* of small and already in a great detail defined parts together to form a larger block. Difference between the two approaches in regards of AI game development is closer described in [Cha10].

**Bottom-up aka connectionism** starts with a very simple thing, then creates many instances of them and something interesting happens. This is called *emergence of intelligent behavior*. The simple thing is called a *neuron* or *perceptron* (illustrated in figure 2.3) and the complex result is called a *neural network (NN)* in AI, or a brain in biology. It is possible to teach a neural network almost anything in case we have enough examples to train it to. A disadvantage would be that it is impossible to determine how and what has been already learned by the NN so there is no way to extract the acquired knowledge from the network. For more information on artificial neural networks please see: [And03], [Has95] & [I.06a], more on connectionism eg. can be found here: [MPR98], [Far09].

The benefit of connectionism approach is *minimalism*. That means there is no requirement on understanding the problem, we just create a very simple building blocks (eg. neurons), put them together to form a network and provide the network with enough examples, by time it will “find” it’s own logic in the problem and eventually learn anything. A disadvantage of this approach is that little or no *mental model* is created. So it might be

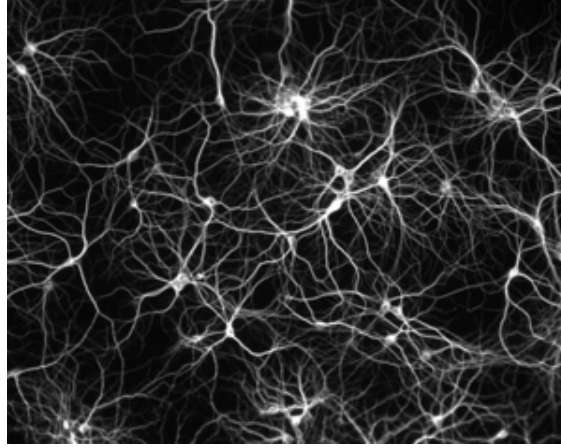


Figure 2.1: Neurons form a network, by [aDK10]

difficult or nearly impossible to *get the knowledge from the system* or train the system on a different problem while preserving the knowledge previously learned.

In a programming comparison, bottom-up would start by defining most specific methods (eg. drivers) first and continue by combining them together to form a desired final state.

**Top-down or symbolic AI** on the other hand, starts with a general model of how a problem looks like and refines the model as it improves by time. In programming, top-down would mean: first create a skeleton for the whole project, define abstract classes and then continue to define more specific classes. This solution seems to suit larger projects better, or projects where we are not exactly sure what will be requirements on the resulting work.

**Human-like or strong AI** is the holy grail of AI, (un)fortunately it is not expected to come true soon. The goal is to implement artificial intelligence that would equal or supersede intelligence of human beings. This has been a topic of many sci-fi books and movies.

**Problem specific AI** is oriented on a *specific task* and solving it. For this reason the programs can be smaller, easier but covering higher detail. And the best part is that there is a big progress in this direction and many interesting problems such as speech recognition or synthesis, computer vision, industrial processes optimization using genetic algorithms or swarm intelligence and many others may belong to this group.

A detailed description of some of these usages is provided in [section Usages of AI](#).

# Chapter 3

## Uses of AI in real life

Even though it is true that AI is still more or less on academic field and we don't have to worry about a robot breaking one of the *Asimov's Laws* [Asi50], countless usages of AI can be found that help to solve problems we couldn't without AI. This chapter presents some interesting usages and provides more references for a further research.

### 3.1 Practical usage of AI nowadays

Is a listing of the most interesting or helpful implementations of AI that came or should come soon to a general everyday use.

#### 3.1.1 Speech recognition & synthesis

*Speech recognition* is a very promising and quickly developing area with notable results already achieved. Speech recognition converts spoken text to words in a text form. Speech synthesis is the opposite process of computer imitating a human voice and reading some text. Applications of these methods range from *telephony* [AAA] (everyone has had to fight through a set of questions of an automaton on a support line), people with *disabilities* [VP09] who can control computers and communicate with others this way, or in army as a support system for piloting planes in battle situations. As our focus lies on games, I must not forget to mention *Lifeline* [spe09], a battle star game where all commands are given via player's microphone.



### 3.1.2 Computer vision

*computer vision (CV)* is another field where computers try to master human skills. Vision is considerably more difficult, but some very good results are already around and start being used in everyday life. As a picture means nothing to a computer, **CV** allows computers to "understand" what is in the picture. Very good results have been achieved by Center for Machine Perception, **Czech Technical University (CTU)**, Prague with their face detector [MS] and a car driving assistant technology that monitors passed traffic signs (see video [Svo]). Other topics are 3D photography, reconstruction, video tagging, ...

### 3.1.3 Optimization

*Optimization* has a huge impact and usage in industry. Everything can be optimized to reach higher quality, lower cost, faster production time or whatnot. And the area of usages is impressive, see for yourselves, one can optimize: chemical processes, transport of material, settings of components in a CPU, a timetable, heating of home and much more.

### 3.1.4 Swarm intelligence

*Swarm intelligence* is a specific method of optimization, and that is by imitating movement of many particles - **particle swarm optimization (PSO)** or a swarm of bees/birds, etc. These are methods of black box optimization, which are quite easy to implement and provide satisfactory results. [Ota10c]

### 3.1.5 Genetic programming

*genetic programming (GP)* is another "inspired by nature" method of AI. **GP** is used commonly for optimization or solving various problems. The principle is its similarity to *evolution* and natural selection. For example a *self repairing code* can be implemented with genetic programming. [Koz92] [Ota10a]

### 3.1.6 Decision trees

*Decision trees* use a statistical method that allows for decision on different situations based on previous experience. Practical usages include *risk management* or in hospitals. In a hospital decision trees have been implemented to quickly diagnose myocardial infarct at admission and this method proved to be successful. [JJP<sup>+</sup>95]

## 3.2 A closer look at the game industry

*Game AI* refers to implementation of some traditional means of **AI** to provide a better gaming experience. It is an emerging technology but as the requirements on realistic behavior of characters in games are growing, it seems AI can achieve a good position in game industry.

### 3.2.1 Usage

Usually this is controlling of *behavior of non-player characters (NPCs)*. Like their attitude towards the player based on his previous behavior or deciding what to do while idle (eg. go fishing instead of staying still). A good example are also fight scenes where movements of the opponent are controlled by AI. *Path finding* allows the **NPCs** to move around naturally, use the shortest path possible, not to hit walls, get lost sometimes or avoid "fog of war". Yet another usage is ability of multiple **NPCs** to cooperate and form for example enemy group and attack together to have higher fire power.

Aside from these, AI can be also used to push down the **NPCs** performance, which is typical for **first-person shooters (FPS)** where humans wouldn't stand a chance against computer's perfect aim. Some AI can also monitor overall progress of the game and adjust the game difficulty dynamically to fit the player's skills and make the game fun for all players.

Of course AI brings some CPU load but the capability to adapt to player's skills or offer a reasonable enemy surpasses this problem by far. All these things put together in modern games bring better and better game experience - for example figure 3.2.1 shows an artificial entity with a rendered face that looks almost like human, moves muscles while talking and even expresses emotions!

### 3.2.2 Top 8 most influential games using AI

Following part lists some of modern remarkable games where AI is used. Each listing consists of a short description of the game and the improvements brought by AI. Some games and descriptions are from a wider list at [\[Cha07\]](#).

#### 1. Total War

Released: 2000-2006 Developer: The Creative Assembly

**30 Second Pitch** Total War is a series of games combining turn-based strategy on a Risk-like map, with real-time tactical control of battles on a 3D terrain.



Figure 3.1: Real? Artificial? [Sta, author]

**Innovations in Game AI** *Thousands of AI-controlled soldiers* are featured for the first time in a fun and interactive game, without noticeable performance problems. The game models the emotions of groups of soldiers, essential for simulating battles accurately. This logic is inspired by the book, *The Art Of War* [TzuBC]. The Total War engine is used on TV by the History Channel as part of the Decisive Battles series.

## 2. The Sims

Released: 2000 Developer: Maxis

**30 Second Pitch** The Sims is a life-simulation of the daily activities of a family of virtual characters in a suburban house. The player gets to design and build the house, as well as guide these “Sims” through the day.

**Innovations in Game AI** Smart objects are used to help implement the behaviors. The object specifies how each character interacts with it, which has many scalability and work flow advantages over centralized logic. The Sims each have basic desires which drive their choice of actions. The *emotional interaction between the characters* is also modeled, which allows for relationships.

## 3. Creatures

Released: 1996 Developer: Millennium Interactive

**30 Second Pitch** Creatures is an artificial life program where the user ‘hatches’ small furry animals and teaches them how to behave. These “Norms” can talk, feed themselves, and protect themselves against vicious creatures.

**Innovations in Game AI** It is the first popular *application of machine learning* into an interactive simulation. *Neural networks* are used by the creatures to learn what to do. The game is regarded as a breakthrough in **artificial life (alife)** research, which aims to model the behavior of creatures interacting with their environment.

#### 4. Façade

Released: 2005 Developer: Procedural Arts

**30 Second Pitch** Façade is an interactive story where a couple who has a relationship breakdown is invited to the apartment of Grace and Trip. The player can coach them using full typed sentences.

**Innovations in Game AI** The player interacts with the game by *typing text into a natural language parser*. The underlying behavior language provides ways to specify the behavior of characters in terms of a dynamic story.

#### 5. F.E.A.R.

Released: 2005 Developer: Monolith Productions

**30 Second Pitch** F.E.A.R. First Encounter Assault Recon is a first-person shooter which contains the player with a supernatural phenomenon and, naturally, armies of cloned soldiers.

**Innovations in Game AI** The AI uses a planner to *generate context-sensitive behaviors*, the first time in a mainstream game. This technology is used as a reference for many studios still today. The enemies are capable of using the environment very cleverly, finding cover behind tables, tipping bookshelves, opening doors, crashing through windows... Squad tactics are used to great effect. The enemies perform flanking maneuvers, use suppression fire, etc.

#### 6. Black & White

Released: 2001 Developer: Lionhead Studios

**30 Second Pitch** Black & White is a god game which includes elements of artificial life simulations and some strategy. The player rules over an island populated by various tribes, and can teach a large animal-like creature to do his/her bidding.

**Innovations in Game AI** The game play is focused on the interaction with a large AI creature which can learn from examples, and *takes reward and punishment*. The design integrates artificial life within the context of a strategy game. The engine uses a solid AI architecture, rooted in cognitive science, known as **belief-desire-intention (BDI)**. Machine learning techniques such as decision trees and neural networks are used with a great success.

## 7. Chatter bot Alice

AI Foundation

**30 Second Pitch** A.L.I.C.E. - Artificial Linguistic Internet Computer Entity - is not a real game, the more addictive she is. It's a robot that is capable of leading a chat with you. I was quite amused how much time we've spent talking. One day maybe psychologists will be replaced... [ea95]

Human: What is it like to be a robot?

ALICE: Much the same as being a human, except that we lack all emotions.

Human: Do you have any desires?

ALICE: My purpose is to become smarter than humans and immortal.

Human: Do you think robots are better than humans?

ALICE: Yes, my memory capacity far exceeds that of a human being.

Human: Do you like yourself?

ALICE: Yes, having self-esteem is important for robots too.

[Ah10]

**Innovations in Game AI** Alice is a program producing output related to what the human passed on input also with use of some heuristics and experience. It is one of the strongest programs of its type and *has won the Loebner Prize*, awarded to accomplished humanoid, talking robots.

## 8. Assassins Creed 2

Released: 2009 Developer: Ubisoft

**30 Second Pitch** A third person action-adventure where the player has to revive memories and skills of Ezio, an Italian assassin in renaissance period. The game offers wonderful graphics with realistic buildings and landmarks from that time, astonishing movements and open plot.

**Innovation in Game AI** The Assassins creed is famous for the movements abilities and AI used enhances the *combat moves* (3.2.2) of the opponent fighters.



Figure 3.2: Assassins Creed 2 fight scene [img, author]

# Chapter 4

## Focus on Reinforcement Learning

The chapter about **reinforcement learning (RL)** is framed in a top-down **2.3** manner, so at first we will talk about what is learning, see principles of **RL** and later on get to fine tuning of the algorithm and we will discuss some possible modifications. A good introduction to RL can be found in the *RL cheat sheet* which is for your convenience included in **appendix A**.

### 4.1 Learning

Is a crucial ability of every intelligent entity. *Learning* allows for predicting and planning further actions based on observations of the surrounding environment, memory of past actions, reactions and results – *experience*.

**Learning in games** Learning is a hot topic in games [FL98]. In principle, learning **AI** has the potential to adapt to each player, learning their tricks and techniques and providing a consistent challenge. It has the potential to produce more believable characters: characters that can learn about their environment and use it to the best effect. It also has the potential to reduce the effort needed to create game-specific **AI**: characters should be able to learn about their surroundings and the tactical options that they provide. In practice, it hasn't yet fulfilled its promise, and not for want of trying. There is a whole range of different learning techniques, from very simple number tweaking through to complex neural networks (**NN**). Each has its own quirks that need to be understood before they can be used in real games. [I.06b, p. 579]. A very promising candidate for usage in games arises from **reinforcement learning (RL)**, of which we will talk in the next section.

## 4.2 Demo games

In the following explanations, I will use two examples realized in [source codes](#). Here is a short description of the games, more information is in the [cheat sheet](#).

**Squares** Squares is a very simple “game”, state-space is a m-by-n matrix where one point is your current position, some points are marked as targets (sweet strawberry). Actions are transitions to the four basic directions by one square. More actions can be added easily. As a reinforcement function is used class `ReinforcementFinPos` that provides positive reinforcement only when a target state is reached. In all other cases this returns a neutral value (0), that is the same as if no reinforcement was received. The exploration strategy is provided by `ExplorationMixed(0.5)` which in 50% of cases returns a random action (that means tries out a new possible way - explore), in other 50% it follows the best action chosen from the knowledge base (exploit). The task is to get to a target in the shortest possible way. This is achieved by creating a map (policy) during training and then following it. The pictures represent the current state of the policy, each square means on state on the position it occupies in the picture. Meaning of colors in squares is explained in the legend in pictures ([top=green](#), [right=blue](#), [bottom=yellow](#), [left=red](#)).

**Balancer** or inverted pendulum or cart pole. Here applies, an image is worth a thousand words ([4.2](#)) but I will add a short description. It is a commonly used example in RL [[Doy00](#)]. In this case projected only to 2D and there is a stick/pole balancing on a cart. Gravity draws the stick down, but this can be compensated by the cart movement. Available actions are just move to the right or left and a state is fully described only by the degree from perpendicular line (plus momentum in some cases).

I have simplified the problem even more by omitting the movable cart; I just have actions in the two sides (but with different strength == a degree for which the action moves the pole, action with a bigger degree introduces a bigger change to momentum) and momentum, solution to this problem is something like: “try to balance near the top(degree 0). When falling on one side, compensate by taking an appropriate action in different direction (bear in mind the momentum)”. Ideas of how different AI methods would tackle with the problem can be seen in [4.4](#).

## 4.3 Reinforcement learning

RL [[I.06b](#), p. 613], [[RN03](#), p. 763] and [[wik10](#)] is a learning technique consisting of the 3 basic parts:

- *exploration strategy* – that tries out different actions



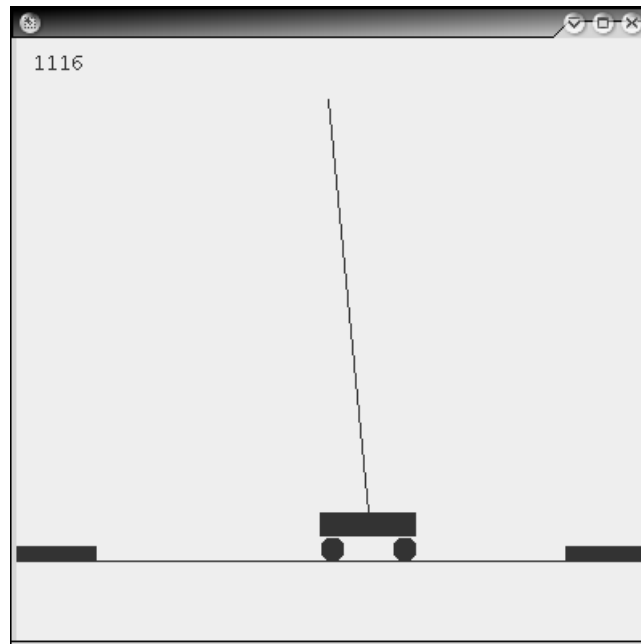


Figure 4.1: Cart-pole system

- *reinforcement function* – provides feedback on how well do we stand so far
- *learning rule* – links together the 2

RL can be classified as a *machine learning* technique used in AI, based on psychological theory (humans learn similar way). **reinforcement learning (RL)** strives to “learn what action to take in a particular state” → which action is the best for a state → make a *decision policy* to determine which action, taken from the particular state, maximizes a (long term) *reward*.

RL is based on principles of **dynamic programming (DP)**. Unlike *supervised learning*, we need not to know which action-state pair is correct or optimal (*unsupervised learning*); the algorithm finds out itself. This is a significant advantage in many cases. RL also doesn't try to build a *model of the world*/problem it solves, it just uses the world as a black-box that gives a feedback on some input. For this reason, we may use that same RL code on many very different problems with only a slight modifications. We need to be able to get value for a *action-state pair* but not otherwise: knowledge of which state-action belongs to a particular value is not required.

### 4.3.1 Q learning

Q learning is one of RL techniques, learns a state-action function which describes a utility of taking an action at the state. That is described by:

$$Q : S \times A \rightarrow \mathbb{R}$$

For each state-action pair we have a value (Q value) in policy which we seek to maximize. During the process of learning/training, the following formula is used:

$$Q_{new}(State_i, Action_j) = (1-\alpha) \cdot Q_{prev}(State_i, Action_j) + (\alpha \cdot reward + \gamma \cdot \max_a Q(S_{i+1}, Action_a)) \quad (4.1)$$

where:

$State_i$ ,  $Action_j$  are inputs;  $Action_j$  takes us from  $State_i$  to  $State_{i+1}$

$\max_a Q(S_{i+1}, Action_a)$  is maximum value achieved for taking an action at  $State_{i+1}$   $\alpha, \gamma$  are constants

### Rationale:

Many times, it is impossible to tell how good an action is at the time, on the other hand, it is quite easy to evaluate the game state when some significant milestone is reached (eg. a final state).

Let's provide a model example: A player wants to kill an enemy soldier, would a better action be taking a knife, attaching a rope somewhere or taking a glass bottle? Hard to say, right? But once the player kills him by setting a trap and cracked pieces of glass kill the soldier after an explosion of a microwave..then we can clearly tell that the action succeeded and give reward points.

### Principle:

*Q-learning* algorithm takes some actions and records them to a policy, once a feedback is received (often later, after the actions have been taken) it is able to evaluate (formula 4.1) how good the choices were and base it's further behavior on this fact.

### Representation of the world

As stated before, the game consists of a diagram of states and the *reinforcement function* is able to return an integer value at each state. So basically everything what is in a state – like ammunition, health, number of killed enemies, time and so on – can be learned and optimized for. Things not included will not be taken in account. The good thing is that the algorithm doesn't have to understand any of the items in the representation, as far as it is able to assign a numerical value to them. Fortunately for us, there is no need for the algorithm to translate back from the integer value of reinforcement function to a game state. A powerful feature of reinforcement learning is that it can easily cope with *uncertainty*.

### 4.3.2 Learning itself

We remember each state & action taken together with a numerical value representing how good the action was. The new value is counted from the Q-learning formula 4.1. In short, the new *Q-value* is a blend between its current value and a new value, which combines the reinforcement for the action and the quality of the state the action led to (this is similar to the principle of DP).

#### Phase 1: training/learning

This is the preparation phase, where the unity function is created/learned. Images 4.3.2, 4.3.2 & 4.3.2 illustrate the process of learning in different degrees of the training process. The training itself is done in a loop with principle like:

1. start at random state
2. in a loop(runs)
3. based on policy and exploration strategy choose an action; (exploration strategy provides balance between trying out new actions and following the learned policy)
4. apply the action to get to a new state
5. compute a new value for the oldState-action pair every time (formula 4.1)
6. receive a reward (sometimes)

#### Phase 2: run

Solving one instance of a problem is trivial once we have a policy (map). It is just following the instructions stored in the policy. The process runs as follows:

1. start at required beginning state
2. choose best action (use policy)
3. take that action (optionally also learn during this run)
4. do this until you reach the desired target

Figure 4.3.2 shows a policy in the middle of the learning process. Notice, how known fields start appearing from the targets and spread to nearby fields. Later on, the map is all learned - that is: we know how to get to the target from every point in the state space. By time, the policy/map can even *refine* - change stored best action at some positions. A fully learned policy may look like this 4.3.2, policy with multiple targets and with visualized level of “reliability” of the knowledge is pictured in figure 4.3.2.

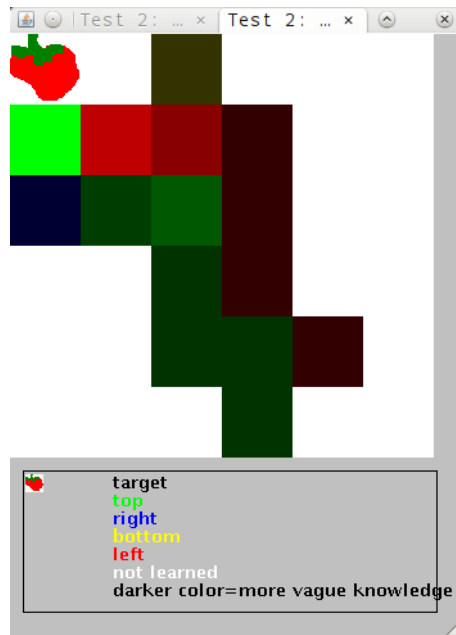


Figure 4.2: Half learned policy

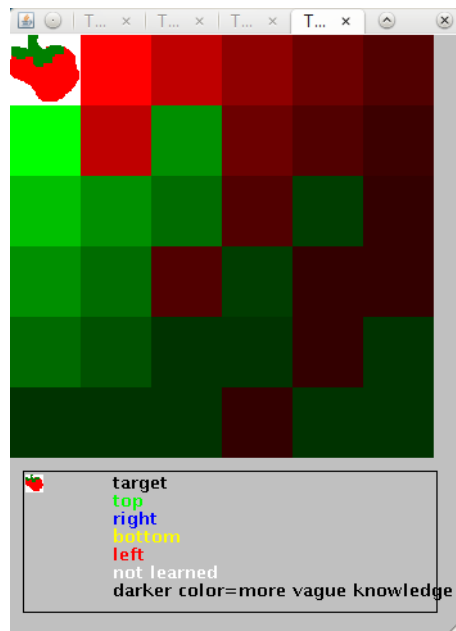


Figure 4.3: Full learned policy

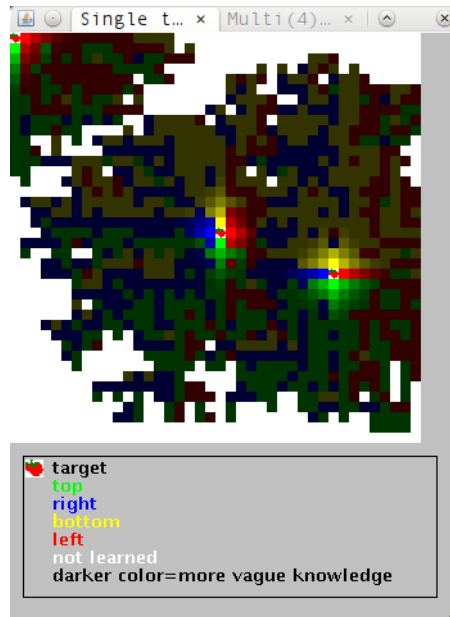


Figure 4.4: Policy with multiple targets

### 4.3.3 Exploration

*Exploration* balances between following the policy (using the already gained knowledge and get further) and exploration of new areas (which might prove better or worse than already known solutions).

Here is the list of exploration strategies implemented in the provided code base.

It consists of:

- greedy
- random
- mixed (balance between random and greedy)
- **simulated annealing (SA)**

### 4.3.4 Rewards

Rewards, punishments, feedback or *reinforcement* is another subsystem of **RL** - a method of letting the computer know if it is doing a good job or not. Again, there are variations on the implementation, some of which may be:

**Positive reward at final state** is usually very simple to implement and yet powerful solution. Example situations are: letting the RL guide through a maze and give feedback after successfully reaching exit; or after a complicated and many operations, tell if landing was alright or not.

**Negative reward at final state** implementation is used in *Balancer* example where the task is to avoid outer states.

**Reward at every step** receiving positive or negative reward after every single step would be *supervised learning*.

**No reward at all** (or constant all the time) is the other opposite which leads to not learning anything and just random search through state space.

**More sophisticated** methods can be receiving reward after each *considerable improvement*, reaching some *check-point* or a punishment when getting worse.

### 4.3.5 Convergence

Convergence means reaching the optimal policy  $P^*$ , it's a state where Q value are no longer changing with further iterations. This is possible for very simple problems, but not feasible for real problems-like games, with many states and probabilistic outcome of actions. For these we say program has reached a *suboptimal policy*. More reading [I.06b, p. 616], [Beg05], [E.O06]

### 4.3.6 Pseudo-code

A Java-like pseudo code that describes one trial of the train() method:

```
1 // backend to store Q values
2 store = new Storage();
3
4 // a random starting point
5 state = Problem.getRandomState();
6
7 //training loop
8 for(int i=0; i< runsOfTraining; i++) {
9     // our walk is too long and leads to nowhere, rather try from a different
10    location
11    if(random() < resetness) {
12        state = Proble.getRandomState();
13    }
14
15    // possible actions to choose from
16    actions[] = state.getAvailableActions();
17
18    //exploit knowledge or explore new area
19    if(random() < explore) {
20        action = oneOf(actions); // explore
21    } else {
22        action = state.getBestAction(); // follow already learned
23    }
24
25    // apply the action
26    newState = state.getNext(action);
```

```

27 |
28 | // get reinforcement (if any)
29 | reward = reinforcement.getReward(newState);
30 |
31 | // current Q value
32 | q = store.getValue(state, action);
33 |
34 | // max Q val from newState would be after applying best action there
35 | qMax = store.getValue(newState, newState.getBestAction());
36 |
37 | // q learning - see the formula?
38 | qNew = (1 - alpha) * q + alpha * (reward + gamma * qMax);
39 |
40 | // store new Q value
41 | store.setValue(state, action, qNew);
42 |
43 | // move current state
44 | state = newState;
45 |
46 | // continue again
47 | }

```

Listing 4.1: Java-like implementation of *training loop*.

Implementation of *train()* method is straightforward and using taught policy to reach target is very simple task too. For more examples, please see [source codes](#).

### 4.3.7 Performance

Performance is a very important aspect of all solutions. Increased quality and reliability is compensated with longer processing times during training and vice versa. Parameters of the given problem that negatively affect times are number of dimensions, size of the state space, number of available actions and probability of unexpected result on taking an action. Higher number of final state/targets, however can improve the situation. Another aspect is implementation in a programming language - usage of hash tables or neural networks for storage and so on.

Constant parameters of Q learning can drastically influence the process of learning and thus a speed gain/loss too. These are discussed in the following section. [I.06b, p. 619], [SB98], [RN03]

### 4.3.8 Fine tuning

The algorithm has four parameters with the variable names *alpha*, *gamma*, *runsOfTrainig*, and *resetness* in the pseudo-code above (4.3.6). The first two correspond to the alpha and gamma parameters in the Q-learning rule 4.1. Each has a different effect on the outcome

of the algorithm and is worth looking at in detail.

**Learning Rate - alpha,  $\alpha$**  influences how much the new learned Q-value is preferred over the already existing one. Alpha is in range of  $[0, 1]$ , where value of zero means a constant policy that never learns anything new, one on the other hand, means absolutely ignoring previous experience and behave opportunistically and follow latest maximal improvements. Higher alpha value allows faster learning and is thus used in problems with very limited number of trials. On the opposite, problems where amount of rewards may change with time require smaller alpha parameter to be able to absorb these changes. Experimentally reasonable alpha value to me seems around 0.3, plus you may take advantage of lowering this value with time, so you could start at 0.5 and go down to 0.1.

**Discount rate - gamma,  $\gamma$**  controls how much an action's Q-value depends on the Q-value at the state (or states) it leads to. Gamma range is  $[0, 1]$ .

Quote from [I.06b, p. 620] : “A value of zero would rate every action only in terms of the reward it directly provides. The algorithm would learn no long-term strategies involving a sequence of actions. A value of one would rate the reward for the current action as equally important as the quality of the state it leads to. Higher values favor longer sequences of actions, but take correspondingly longer to learn. Lower values stabilize faster, but usually support relatively short sequences. It is possible to select the way rewards are provided to increase the sequence length (see the later section on reward values), but again this makes learning take longer. A value of 0.75 is a good initial value to try, again based on my experience and experimentation. With this value, an action with a reward of 1 will contribute 0.05 to the Q-value of an action ten steps earlier in the sequence.”

**Randomness for exploration - explore** can be from range of  $[0, 1]$  too and has the tells whether to explore or exploit.

Zero value means always continue finding in the current path, so the search looks like greedy searching - exploitation strategy. The algorithm only exploits current knowledge, reinforcing what it already knows.

A value of one means explore in a new direction at every step - creates a pure random exploration strategy: the algorithm will always try out new things, but never benefit from the existing knowledge. This is a classic trade-off in learning algorithms: to what extent should we try to learn new things (which may be much worse than the things we know are good), and to what extent should we exploit the knowledge we have gained. “Often, if one state and action is excellent (has a high Q-value), then other similar states and actions will also be good. If we have learned a high Q-value for killing an enemy character, for example, we will probably have high Q-values for bringing the character close to death. So heading toward known high Q-values is often a good strategy for finding other state-action pairs with good Q-values.”, from [I.06b, p. 621]



**Length of average walk - resetness** range  $[0,1]$  and controls how often the program should stop following it's current path (sequence of connected actions) and reset to a new position. Meaning of zero value here is: never reset, always continue searching in the current sequence of following states. While value set to one means reset every step, which is same as random searching. In my experiments, I used value equal to  $1/(\text{avg. optimal steps to target})$  or 0.01 as the default.

### 4.3.9 Ideas for modifications

RL is the general principle, but there are places to do some tweaking and modifications. The list here is not complete, it's just things I found interesting.

**Using neural networks as a back-end for storage** seems like an interesting idea. NNs are capable of learning to return some output for a given input. This can be used to remember state-action values instead of a look-up table. By the uncertainty principle of where is what information stored in NN, we gain some build-in generalization, neural networks are capable of. More on this topic eg. in [\[Hum\]](#)

**Parallelism in RL** is a very interesting idea, especially for multiagent systems, where an agent could benefit from the knowledge acquired by someone else - see [\[Gea96\]](#). Parallel training is already implemented in the source codes, so you can experiment with it.

**Forgetting** RL policy could forget some unneeded information, there is a whole chapter dedicated to this idea, please proceed to [chapter 5](#).

**TD or SARSA instead of Q learning** reinforcement learning doesn't have to use only Q-learning, there are other methods with their advantages and disadvantages; these methods can be used for RL too. [\[SB98\]](#)

## 4.4 Different AIs

This section will illustrate usage of different learning methods on the [Balancer](#) problem, of course RL will not be missing and we'll add forgetting too.

**Neural networks** will have on input only the degree (eg. 6 neurons/bits to describe the numeral value) and momentum. The output should be one of the available actions that shall be applied to stabilize the pendulum. The process of *teaching a neural network*

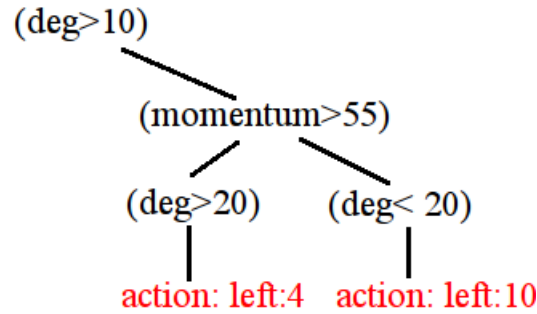


Figure 4.5: part of decision tree for the Balancer problem

consists of providing it an example to input and a desired action to appear on output. This shall be applied many times, so finally NN learns what to do. And not only to the examples that have been provided by the “tutor”, thanks to its structure, NN should make an abstraction and know what to do even for inputs it has never encountered before. Article [HFJ] shows a realization of the balancer problem with neural networks.

**RL** I’m sure you already know how RL would cope with the problem. A state would consist of degree and momentum, available state would have degree in range of accepted degrees ( $<-45, 45>$ ), actions would change the actual deviation and reinforcement would give negative value when failed. The implementation can be seen in the source codes B and more on this topic for example in [Doy00].

**RL with forgetting** would be the same as RL above, plus forgetting of unused actions could be used to reduce the high number of available actions. Also forgetting of unused states and almost zero values could be applied.

**Decision trees** can be taught to perform specific classification too, for example with ID3 [Col04]. Once enough examples were introduced to the decision tree, it will look something like figure 4.4 and should provide satisfactory results.

**Genetic programming** could be used to solve this problem as well. The problem would then be specified as finding an ordered subset of all available actions which would finally lead to as well balanced position near the top as possible. The genome would be long as the maximal number of actions we wish to take till the solution is found and each position would contain a number specifying the action to be taken. Fitness function for this problem is computed from deviation degree and momentum. More on genetic programming [Koz92] and methods for visual genetic programming [Sou10].

# Chapter 5

## Forgetting

**Why is forgetting good for everybody?** People tend to complain about forgetting everything and how is it bad. However, the opposite is true. Forgetting saves us. As we accept dozens of information every minute, we'd simply go mad pretty soon. Same for computers, increasing number of information means bigger problem space and would sooner or later lead to exhaustion of system resources.

### 5.1 How do we forget then?

According to results from biology and psychology, there are several types of human memory. [mem10], [How] That would be a sensor memory, short term and long term memory. The sensor memory operates in times of a fraction of a second and is not intentional. Short term/active memory can hold a few (about 4-9) items and last in order of seconds, in parallel with computers this is RAM. Long term memory, on the other hand, is where data is stored if we managed to remember something. That's where your friend's birth date, favorite pizza store phone no. and monarchs of England are somewhere stored - this would be your brain's hard drive.

When actually thinking about something, we remind us things from long term memory and hold them in the short term memory along with other new ideas. The ability to forget and lose some information helps by keeping only the important, frequently visited memories and not polluting with any unnecessary info.

### 5.2 Forgetting in computer science

It's quite easy to implement forgetting for computers, you just delete, unlink the data and the information is lost forever (human brains probably don't work exactly this way).

The question is **what** data should be deleted?

A good example of forgetting in **computer science (CS)** are neuron networks which simulate behavior of a brain on the physical level ([Ish94]). Another could be tree pruning in decision making algorithms. But in **AI**, the concept of forgetting is, apart from **NN**, new and offers a new space to explore and maybe it will prove itself useful. [Sal93], [EW08], [FKS95] [Xia07] [SLZ04]

### 5.2.1 Implementation for reinforcement learning

I have found quite a lot of resources (5.2) on forgetting using **NN**, a few on general ideas about forgetting in **CS** and almost none for forgetting in **RL** [MA93]. I was quite happy as it seems *forgetting in RL* is a new unexplored area. First I will show the ideas I came along and implemented, then we'll do some experiments and summarize the results.

For most of the ideas, I took inspiration from nature and tried to make the algorithms behave like that as much as possible. The code for forgetting is built atop of policy - in a `PolicyForgettable` interface. It's an interface so it makes creation of your own policies that use forgetting an easy task.

Forgetting in my version means deleting a piece of knowledge (state-action pair and associated Q value) from the policy. This happens only if some *filter* applies, that means when a condition is met. The design of forgetting offers several different methods of forgetting - different filters. All of them come in two forms, hardwired to the function which adds knowledge to the storage base, so these are executed on every program step, or as a separate threads which can be started and stopped independently on the program run. The threads have their defined intervals of execution and while started, they will sweep the knowledge base (each thread does its own job, jobs can run simultaneously) and sleep for a given time, then do it again.

#### Ideas

Section Ideas introduces the different methods of forgetting implemented to RL policy. You will surely notice similarity to "how it works in our normal brains".

**Single action only** this mode forgets all other but the best action for each state. The code is very simple, just loop through all possible states, in each state select best action and delete the rest. This can bring drastic improvements if there is a large number of actions available for each state, however, it spoils the process of learning, so recommendation is to run this only after the training was done to purge the policy table.

**Forget unused states** does as it says. There is a counter attached to each call of getting next state (that is applying selected action to the current state which moves us to a new state). The counter keeps track of how often has each state been visited and of total number of steps stored in policy. If the `state/totalStates` ratio falls below some critical value, the state is considered unused - other states are being visited much more often - and can be deleted. This attitude is useful for problems where a huge number of states is generated, eg. by the fact there is infinite number of possible actions. One of these problems is the *balancing pole* problem where the state space (represented as a deviation degree from optimal perpendicular line) is not discrete - so there is infinite number of possible states.

**Forget similar states** filter brings concept of *generalization* or *abstraction* to forgetting. A function that decides if two states are equal or not needs to be defined. This is done by *overriding the equals method* of the states. The function then just walks through all states and removes one of two in a similar pair. The method shouldn't be invoked too often as in a set of states, some will surely match some other, so eventually we would remove most of the states. That's why I did just linear walk through the states. On the other side, this can help us get rid of the *dual solutions problem*:

"We are stamps collectors and each state is a representation of our collection (eg. A,V,B,G). We care about what stamps do we have in our collection, but not for their order - so solutions (?,V,B,?) and (?,B,V,?) are same to us but introduce cruft to the state space." Clever use of remove similar states could solve this problem.

**Forget almost zero values** has a defined *minimal level for a Q value* and when an action's Q value (in regards to a particular state, of course) drops below, the action is considered *unimportant* and is removed. This is a bit similar to forgetting of unused states, but here it isn't in global, but for each state a different action can be forgotten.

**Forget unused actions** is an interesting one. The method keeps track of how often is each action executed and again, once a critical ratio for an action is exceeded, the action is marked as unused and deleted from actions available. But not only is the action removed from policy, it is also removed from available actions in the whole problem. In problems with specific settings, a lot actions (where some are totally irrelevant to finding the solution) or a really huge state space (compared to time spent on training) this method can bring very promising results. Following section mentions my experimenting with this method.

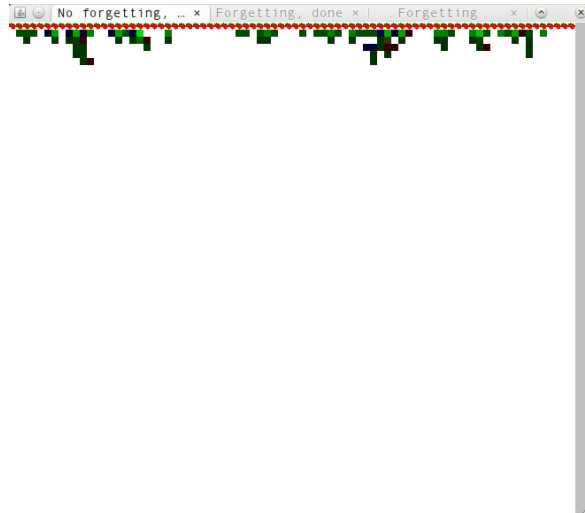
## Experimenting

I tested the use of "forgetting unused actions" on a specific problem where targets were placed all along the top line. I used a big state space (800x80 cells) on the instance

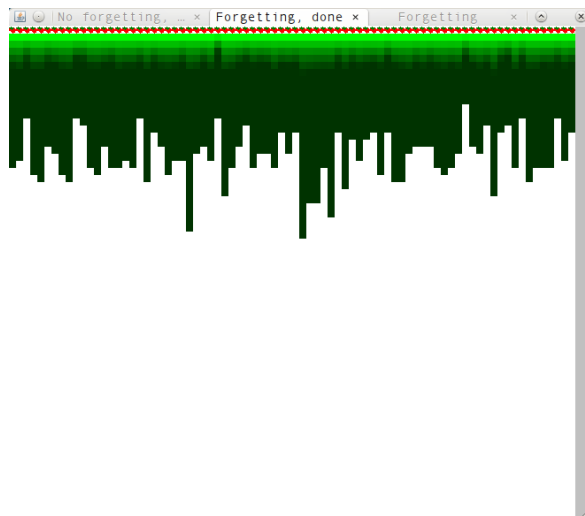
of Squares problem (please see [RL cheat sheet in appendix](#) for details). In detail, this forgetting method has an offset parameter, which says how many rounds it should wait before the forgetting kicks in. This is mandatory as some sort of statistical order must first appear, before we can decide what actions are not useful to us. After this, the thread counts the usage ratios and checks for any of them dropping under the defined critical line. Such action is removed and the process continues. In 400.000 rounds of training, the policy learned only a very little part (see figure [5.1\(a\)](#)), for that reason the results were very poor (cca 2 successful in 10.000 tries). On the other hand, figure [5.1\(b\)](#) shows how the policy looked after the same number of trainings, just with forgetting applied. The improvement (measured in covered area) is clearly visible. The forgetting was optimal, as the only (and correct) action left was action “top”. Therefore average score successful runs in total 10.000 was around 250, which indicates optimal values (avg. 40 steps to target). Apart from bigger area covered by the policy, this helps even for states which are unknown (white color). Even for these states the policy removed useless actions, so there is fewer actions to choose from, which leads to increased chance for randomly selecting the optimal action (here 100%). I should not only praise this method, it must be said that it can completely fail. This occurs when an action crucial for getting to target (top) is removed. This failure can be circumvented by checking if at least some targets have been reached for some period of time, and if not, the last action removed seems vital and ought to be added back.

## Results & summary

Forgetting is definitely not an ultimate solution which could be used everywhere, on the other hand, it appears that forgetting could be used in most cases and used wisely will bring massive improvements. Also, a big advantage is that in my implementation, forgetting can be added to existing RL projects with minimal requirements on refactoring. Implementing this attitude seems right to me for RL resembles how we actually think & learn the most and we (human brains) do implement forgetting. There is a lot of space for improvement and many new ideas in forgetting need to be explored yet.



(a) no forgetting



(b) forgetting

Figure 5.1: Same problem with and without forgetting of unused actions (line of targets on the top)

# Chapter 6

## Conclusion

I have studied possible usages of artificial intelligence and especially its possible implementations in game industry where AI allows solving problems otherwise impossible and thus can offer even more realistic experience in games. Area of my focus was reinforcement learning for its principle which exactly fits the requirement common in games. That is: the world is so complex that only possible feedback could be a response after many actions have been taken. Another main area of my interest was forgetting, a technique not yet very common in computer science but inspired simply by how human brain works, and experimenting with it. Some possible methods of forgetting were designed for the purpose.

The practical part of the work consists of creation of program base which offers implementation of RL, some improvements like parallel training, methods for visualization of the process of learning - animated run of the algorithm and overall progress of learning. A policy that uses forgetting was implemented as mentioned before and results tested and discussed on an example. As far as the examples are concerned, two simple games - Squares & Balancer were designed to demonstrate the abilities of reinforcement learning. There is quite a lot of executable examples that illustrate how some settings influence the process and results of learning. The code for reinforcement learning and forgetting is written with usability in mind, so it should be easy for the users to implement RL technique or forgetting to their projects, experiment with it and see if it can bring them some improvements. The code is very well documented so acquaintance with the sources should be as smooth as possible.

An educational material called “RL cheat sheet” was created for this purpose. It builds upon the this work and the source codes and should provide a brief introduction to the problematics of RL, forgetting and examples used further for students or anyone with basic knowledge of programming who is interested in RL.

Future work on the project could include implementing other additions to reinforcement learning, tuning up and bringing new features to the code for forgetting and experimenting with other problems and games. My personal intention is to try to implement the methods



discussed on a real robot and learn it walk through a maze or create an artificial entity that could play a commercial game instead of a human.

# Appendix A

## RL cheat sheet

Provides a brief introduction to the problematics of reinforcement learning along with explanation of examples used in the code plus description of most important classes. The sheet is in a form of a clickable pdf and can be obtained as a separate file.

Marek Otáhal,  
[markotahal@gmail.com](mailto:markotahal@gmail.com)  
FEE, CTU, Prague  
2010

# Reinforcement learning

## *aka Carrot & stick*

*cheat sheet*

### **Keywords:**

AI; reinforcement learning - RL; reinforcement, reward, feedback; policy, value function, state-action pair; decision, MDP (Markov-decision-process), dynamic programming - DP; Q-learning; exploration, maximization problem

### **Contents:**

[Theory](#)  
[Work with examples](#)  
[Further work](#)  
[Sources](#)

## Dynamic programming

The principle of **dynamic programming (DP)** [\[1\]](#) is solving complex problems by breaking them to smaller pieces, finding a solution for these (*divide and conquer*) and putting them back together to form the global solution (*back-propagation, back tracking*).

“It's like when you have a bar of chocolate, that big you can't eat it alone. You chop it to pieces (*divide*) and invite your friends to help you eat it (*conquer*). Don't ask me about the back-propagation, though :)”

These pictures show an example:

target is the strawberry, on white fields we don't know which way to go,  
colored are already mapped.

Task is: get from given spot to the target.

The following table shows state of the policy(*learning*) pictured every 1000 steps.

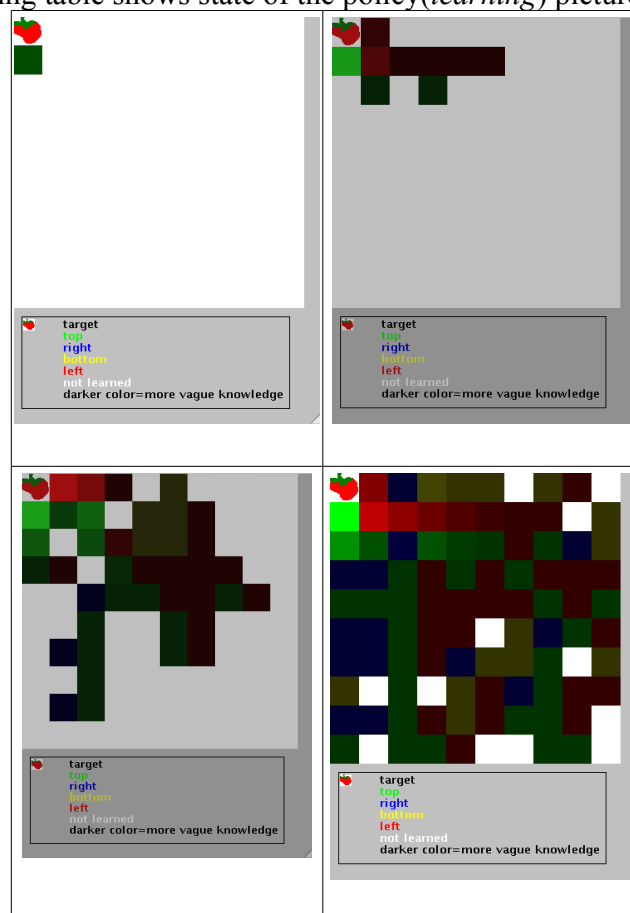


Table 1: progress of spreading knowledge, plot by 1000 steps

## Reinforcement learning

Reinforcement learning is a *machine learning* technique used in artificial intelligence; it is based on psychological theory (humans learn similar way). Reinforcement learning (RL) [2] strives to “*learn what action to take in a particular state*” → which action is the best for a state → make a decision policy to determine which action, taken from the particular state, maximizes a (long term) reward.

RL is based on principles of *dynamic* programming.

*Unlike supervised* learning, we need not to know which action-state pair is correct or optimal; the algorithm finds out itself. This is a significant advantage in many cases.

RL also doesn't try to build a model of the world/problem it solves, it just uses the world as a black-box that gives a feedback on some input. For this reason, we may use that same RL code on many very different problems with only a slight modifications.

We need to be able to get value for a action-state pair, but not otherwise: knowledge of which state-action belongs to a particular value is not required.

## Q learning

Q learning [3] is one of RL techniques, learns a state-action function which describes a utility of taking an action at the state.

That is described by:  $Q: S \times A \rightarrow \mathbb{R}$

For each state-action pair we have a value (Q value) in policy which we seek to maximize. During the process of *learning/training*, the following formula is used:

$$Q_{new}(State_i, Action_j) = (1 - \alpha) \cdot Q_{prev}(State_i, Action_j) + \alpha \cdot (reward + \gamma \cdot \max_a Q(S_{i+1}, Action_a))$$

where:

$State_i$ ,  $Action_j$  are inputs;  $Action_j$  takes us from  $State_i$  to  $State_{i+1}$   
 $\max_a Q(S_{i+1}, Action_a)$  is maximum value achieved for taking an action at  $State_{i+1}$   
 $\alpha, \gamma$  are constants

Other methods are:

- TD (temporal difference),
- SARSA, ...

## Tunables (Q learning)

- number of *runs* during training
- *alpha* is *learning rate*, from  $<0;1>$ , how will the new value override the old information, 0 means never learn anything, 1 means only consider newest info
- *gamma* is *discount factor*, rate of focus on future rewards, 0..consider only current rewards,  $\sim 1$  strive for a longer high term reward
- *exploration* strategy used (greedy, random, annealing, mixed random&greedy,...)
- *reinforcement* strategy (receive positive reward at final state, neg. reward at final state, compute reward at every state – e.g. "reward=money I have now - \$ at previous state")

## Principle?

### Phase 1: training/learning

- 1) In a loop, start at random state
- 2) based on policy and exploration strategy choose an action;  
exploration strategy provides balance between trying out new actions and following the learned policy
- 3) apply the action to get to a new state
- 4) compute a new value for the oldState-action pair every time
- 5) receive a reward (sometimes)

### Phase 2: run

- 1) start at required beginning state
- 2) choose best action (use policy)
- 3) take that action (optionally also learn during this run)
- 4) do this until you reach desired target

## Requirements

*environment* – a definition of states, needs to be able to say which are valid, for a real problem, each state usually contains some inner information (e.g. Speed, or position on a map)

*actions* – possible methods of changing current state and getting to a new one (e.g. Action Up: translate state position by  $[0,1]$  on  $[x,y]$ )

*reinforcement function* – gives reward (positive/negative), reinforcement can come after a long set of actions (e.g. Reward +1 when state == target([3,3]) )

*exploration strategy* – when training/learning, tell whether to choose a new random way or go in an already learnt direction, trade-off between exploration (new territory) and exploitation (current knowledge) (g. greedy(always prefer knowledge), random(always explore), annealing,...)

*storage* – somewhere to store learnt policy values (look-up table, neural network, database,...)

## Usages

This section will help you understand RL and show some demonstrational, interesting and real world examples.

### n-armed bandit

Imagine a multi-slot machine with  $n$  levers, during one round, just one lever can be pulled. There is a reward associated with pulling each lever, or a particular sequence of levers. e.g. pulling lever L1 gives +0, L2 gives +1, but sequence L1,L1,L3 gives +100. So RL should learn to execute L1, L1 and L3 repeatedly.

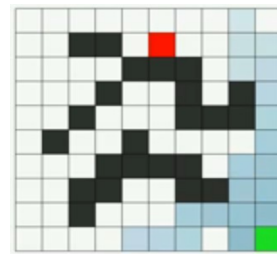
### maze

Problem: a maze, starting point, target and how to get from start to the target?

The maze can be represented as a graph, where crossroads are vertices and edges are possible ways/connections between them.

At the beginning, the robot is blind and doesn't know where to go, thus seeks random. When it reaches the target/already known state, it writes the information which way to go to the previous state.

This is a bit similar to BFS search.



See a nice example: <http://www.youtube.com/watch?v=tovrpoUkzYU&NR=1>

### walking robot

This is truly interesting example of *evolutionary robotics*[\[4\]](#) where a robot learns how to use it's body and at the end can walk.

<http://www.youtube.com/watch?v=PHDT30oWs20>



### real life

Real life deployments of RL range from control (autonomous helicopter,...), computer optimizations & games, to economics – predicting markets and many further!

## RL learning – run explained

Following section focuses on the use of code you were provided, especially on the game Squares. I will briefly explain the “game” and then show how the RL algorithm works in detail.

### The Squares

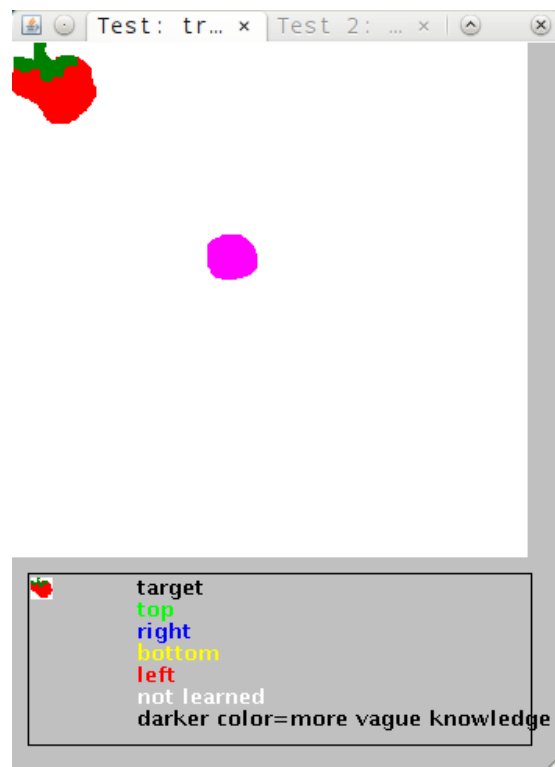
Squares is a very simple “game”, state-space is a  $m$ -by- $n$  matrix where one point is your current position, some points are marked as *targets*. *Actions* are transitions to the four basic directions by one square. More actions can be added easily.

As a *reinforcement* function is used class ReinforcementFinPos that provides positive reinforcement only when a target state is reached. In all other cases this returns a neutral value, that is the same as if no reinforcement was received.

The *exploration* strategy is provided by ExplorationMixed(0.5) which in 50% of cases returns a random direction (that means tries out a new possible way), in other 50% it follows the knowledge base.

That's it, let's jump to the Squares world!

On the first picture, the knowledge base is empty, target position is the strawberry in the top left corner. Our(solver's) position is a random place somewhere in the state space (6 x 6). Let's assume for now it is at the dot position (the dot is called *Lily*).

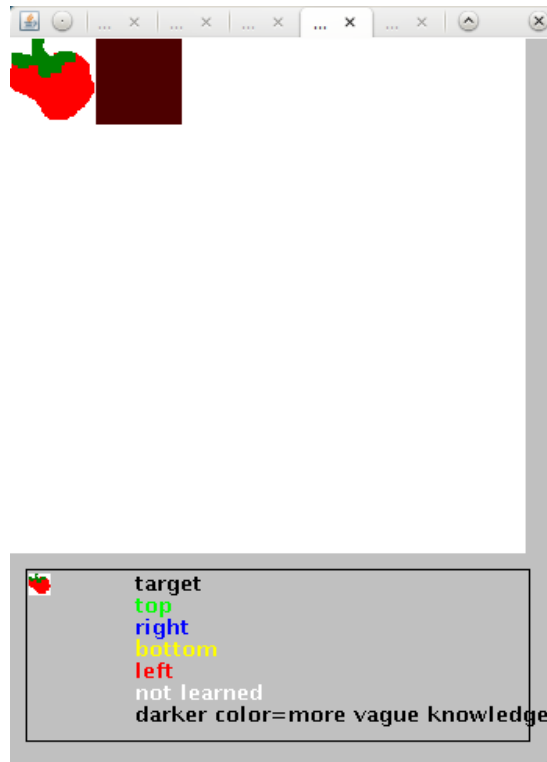




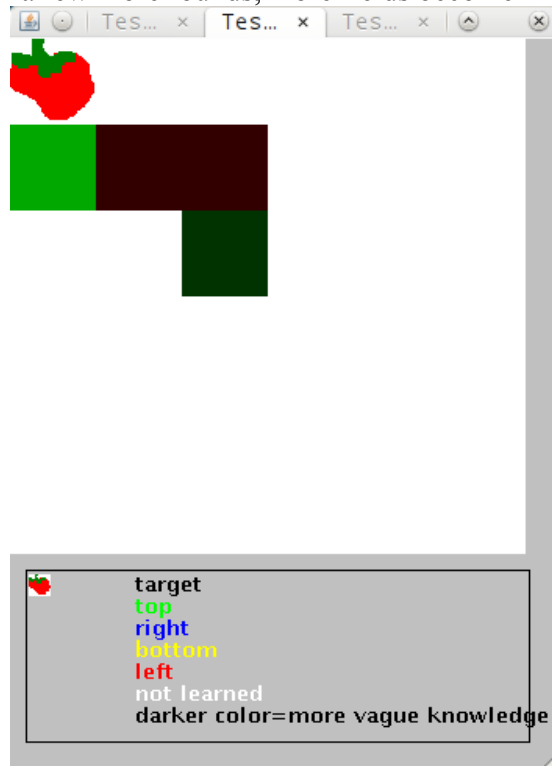
Lily asks the policy which way to go. Policy is empty and returns a random action (top) of all available actions. Lily checks if the new position (after applying “top” to current state) is a target position (false) and counts a new Q value with [the formula](#) above, but it is again neutral (zero).

This way, the behavior resembles a random walk...

...till Lily comes very close to the target. Now she is right next to the target, chooses (by chance) a good action (left), new position is a target position, so she receives a reward, new Q value is not null and the policy has just learned a first rule. Now every time she is at the point, she knows where to go.

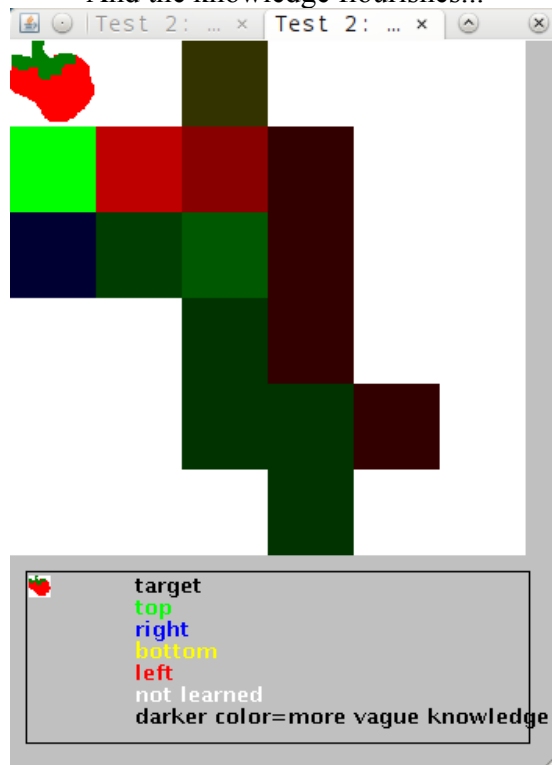


After a few more rounds, more fields become known.

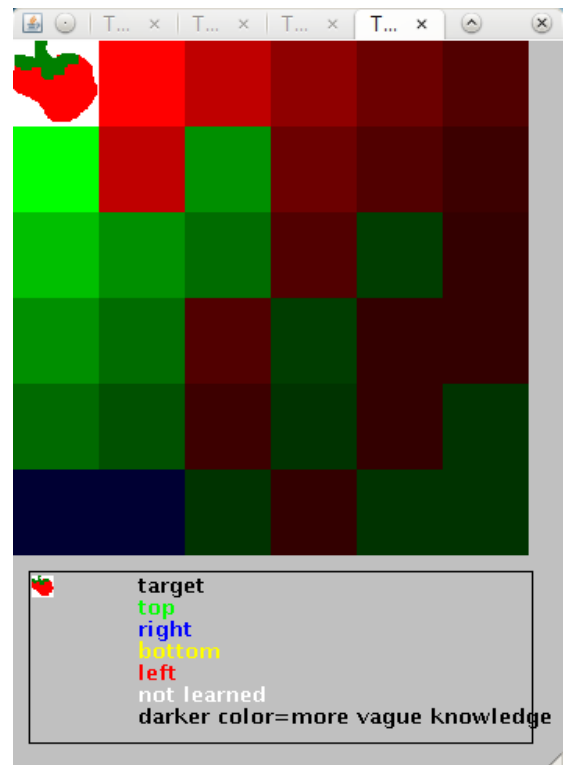


Notice, how the color of often used fields becomes more intensive.

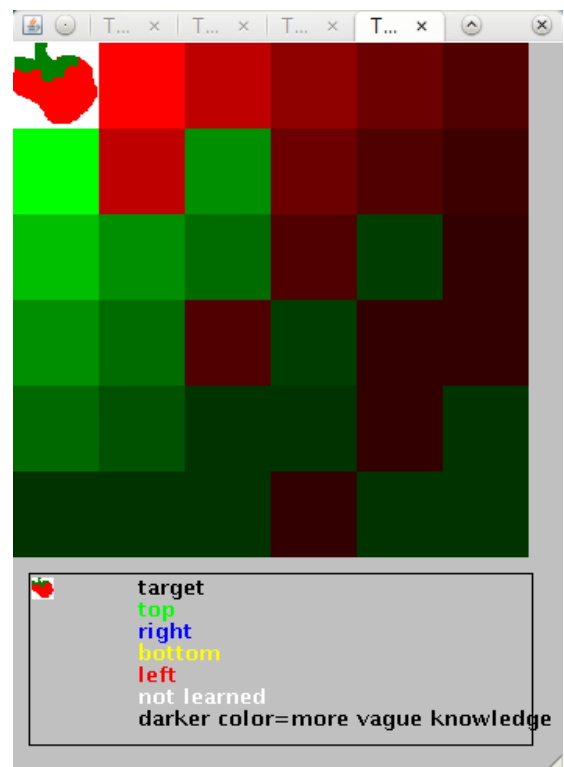
And the knowledge flourishes...



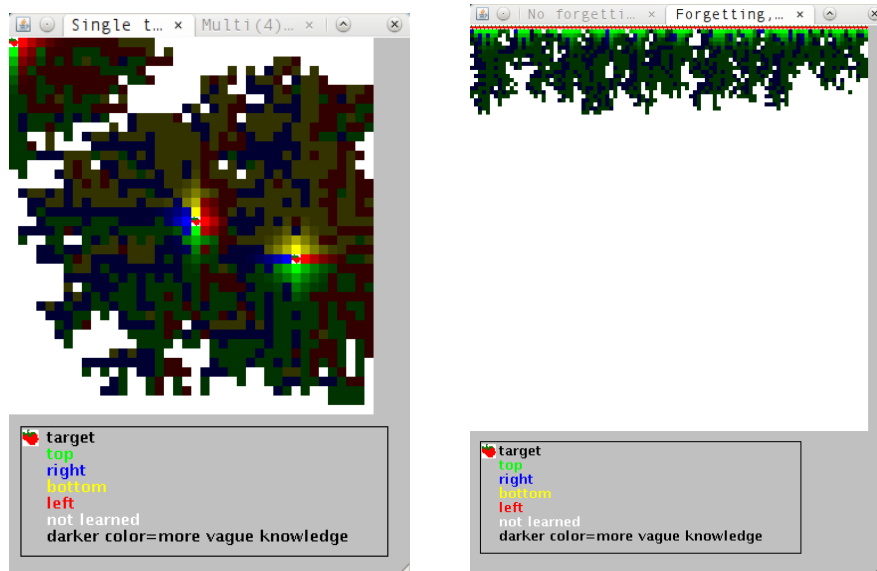
A full learn policy would look like this. For every point, Lily knows which next point is the best for her (DP) and the map leads her to the target.



With time, the policy refines. This means two things: good ways are used more often and increase their Q values (become lighter in color). Sometimes one could see a color change in an already learned field. That happens when exploration decides not to follow policy but try a new way and this way proves better than the previous. This means other action becomes dominating for the state (and a change of color).



This is how the algorithm works. Here are some other examples of runs of RL, first shows a world with 3 targets and second is a very large state-space (400x400).



## ***Experimenting with the provided RL code***

This part will briefly explain what class does what in the provided RL package, show you some usages and help you define your own problems or modify the code.

The structure of the packages is as follows:

- core – RL code
- extras – supportive functions for general use
- examples – runnable examples and their specific classes

## **Balancer (cart pole, inverse pendulum)**

*Balancer* is a problem type of balancing inverse pendulum. Balancer defines some sort of physics to produce a momentum, uses around 120 actions(!) and has only two targets – at 45 and -45 degrees. Uses ReinforcementFinNeg to try to avoid them.

*Squares* and *Balancer* are fairly simple classes and should help you understand how to set up your own RL problem.

## RLPolicy

RLPolicy defines most of the logic and has the ultimate *train()* method that you will need most. It also offers *trainMultithreaded()* - which runs train on multiple parallel threads apparently.

## Visualizer

will give you all the fancy graphics! ☺ See methods *pict()* for pictures and *play()* to animate. Method *train Animate()* will let you monitor the process of learning and *playTrack()* offers a nice way to represent how Lilly(Barca or whoever) actually searches their way through the state space.

## Examples

Are located in the examples package and would be most useful for learning, as they explain all parts of RL learning process. There are examples to show impact of various exploration strategies, reinforcements, alpha-gamma setting, multiple threads,...

## Forgetting

Forgetting is an advanced topic and is mainly done in PolicyForgetting.

It was inspired and works quite similarly to how humans learn and forget [\[5\]](#). Forgetting can improve how well RL learns.

I've prepared an example of *forgetting unused actions*. It keeps track of how many times has each action been called and if this ratio for some action is too low, the action is removed. So next time *randomAction()* is requested, the removed one will no longer be even considered.

The example used has targets on the whole top line and forgetting brings about a 2-3x longer execution time but also amazing *50-times improvement* in number of successfully reaching a target!!

## Check questions:

1. What is reinforcement learning?
2. Name some advantages and disadvantages of RL over other attitudes to learning.
3. On what problems it is a good idea to apply RL?
4. Ways to improve quality/amount of what has been learned on a specific problem solved with RL?
5. Describe in general the process of learning(training).
6. Think about similarities between RL and neural networks, supervised learning, ...

## Ideas for own research, maybe worth some extra points:

1. Think off and implement a nontrivial example of a problem that can be solved with RL.
2. Apply used or your own methods of forgetting to a problem and show (some) improvements.
3. Some RL in combination with the Lego robot kits at school would be awesome!
4. Something else? **Be creative**

## Sources & more information

**[1] Dynamic programming**

[http://en.wikipedia.org/wiki/Dynamic\\_programming](http://en.wikipedia.org/wiki/Dynamic_programming)

**[2] Reinforcement learning**

[http://en.wikipedia.org/wiki/Reinforcement\\_learning](http://en.wikipedia.org/wiki/Reinforcement_learning)

**[3] Q learning**

<http://en.wikipedia.org/wiki/Q-learning>

**[4] evolutionary robotics**

[http://en.wikipedia.org/wiki/Evolutionary\\_robotics](http://en.wikipedia.org/wiki/Evolutionary_robotics)

**[5] forgetting**

[http://en.wikipedia.org/wiki/Educational\\_psychology](http://en.wikipedia.org/wiki/Educational_psychology)

Books covering whole area of RL and AI:

**[6] AI, Reinforcement learning, Forgetting**

Marek Otahal: Demonstration of learning in games, CTU, 2010

**[7] AI**

Millington I.: Artificial Intelligence for games, Morgan Kaufmann, 2006,  
ISBN 10: 0-12-497782-0

**[8] AI**

Artificial Intelligence, A modern approach - 2<sup>nd</sup> edition,

**[9] RL**

Sutton, Barton: Reinforcement learning: An introduction, MIT Press

# Appendix B

## Included materials

The attached CD has the following structure and contents:

- **source** The folder **source/RL** can be opened as a project in Netbeans IDE and from there all examples can be run. *To run an example in Netbeans*, please go to package examples, where all executable classes reside, choose one with the main() method and run it in single mode (alt-F6). Should you wish to access the source codes directly, they are stored in source/RL/src, you can add them to your own project or compile the classes directly with javac. To manually run eg. class A.java, you have to compile it (javac A.java) and then execute with *java A*.
- **doc** There is a full documentation to the project located in **doc/javadoc/index.html** or source/RL/dist/javadoc/index.html. Please refer to the documentation if you have any questions about the code or see directly the comments in source codes.
- **RL\_cheatsheet.pdf** This file is a brief introduction to reinforcement learning and the examples used in the codes. It's intended to be an educational text for students who encounter the problematics of RL for the first time. I suggest having a look at it before reading the thesis or working with the codes.
- **otahalThesis.pdf** is this thesis in electronic form.
- **files** Folder files contains a few files referred in References but not published online.
- **readme.txt** is this text included on CD.

Marek Otahal, 2010  
[markotahal@gmail.com](mailto:markotahal@gmail.com)

# References

- [AAA] Y. A. Alotaibi, M. Alghamdi, and F. Alotaiby. Speech recognition system of arabic digits based on a telephony arabic corpus. scientific paper <http://www.mghamdi.com/IPCV08.pdf>.
- [aDK10] aul De Koninck. Image of neurons. [http://www.greenspine.ca/media/neuron\\_culture\\_800px.jpg](http://www.greenspine.ca/media/neuron_culture_800px.jpg), 2010.
- [Ah10] Alice and human. [http://en.wikipedia.org/wiki/Artificial\\_Linguistic\\_Internet\\_Computer\\_Entity](http://en.wikipedia.org/wiki/Artificial_Linguistic_Internet_Computer_Entity), 2010.
- [Ana92] A. Anastasi. What counselors should know about the use and interpretation of psychological tests. *Journal of Counseling and Development*, page 610–615, 1992.
- [And03] J. A. Anderson. *An Introduction to neural networks*. Prentice Hall, 2003.
- [And06] M. Anderson. Intelligence, 2006.
- [Asi50] Isaac Asimov. *I,Robot*. Gnome press, 1950.
- [Beg05] A.W. Beggs. On the convergence of reinforcement learning. *Journal of Economic Theory*, 122(1):1–36, May 2005.
- [bri06] *Encyclopedia Britannica*. 2006.
- [Cha07] A. J. Champandard. Top 10 most influential ai games. <http://aigamedev.com/open/highlights/top-ai-games/>, 2007.
- [Cha10] A. J. Champandard. Top-down vs bottom up design. website <http://aigamedev.com/open/articles/top-down-vs-bottom-up-design/>, 2010.
- [Col04] Simon Colton. website <http://www.doc.ic.ac.uk/~sgc/teaching/v231/lecture11.html>, 2004.
- [Doy00] Kenji Doya. Reinforcement learning in continuous time and space. Vol. 12, No. 1:219–245, January 2000.
- [ea95] R. Wallace et al. chatter bot alice - artificial linguistic internet communication entity. <http://alice.pandorabots.com/>, 1995.
- [E.O06] E.O.Ozdamar. On the convergence of q-learning, 2006.
- [EW08] Thomas Eiter and Kewen Wang. Semantic forgetting in answer set programming. *Artif. Intell.*, 172(14):1644–1672, 2008.
- [Far09] Igor Farkaš. Artificial neural networks introduction to connectionism. 2009.
- [FKS95] Rūsiņš Freivalds, Efim Kinber, and Carl H. Smith. On the impact of forgetting on learning machines. *J. ACM*, 42(6):1146–1168, 1995.
- [FL98] D. Fudenberg and D. Levine. Learning in games. *European Economic Review*, pages 631–639, 1998.



- [Gea96] Pan Gu and Anthony B. Maddox et al. *Adaption and Learning in Multi-Agent Systems*, volume 1042/1996. Springer Berlin / Heidelberg, 1996.
- [Gre98] R. L. Gregory. *The Oxford Companion to the Mind*. Oxford University Press, 1998.
- [Gud00] R. R. Gudwin. Evaluating intelligence: A computational semiotics perspective. In *Evaluating intelligence: A computational semiotics perspective*, page 2080–2085, 2000.
- [Has95] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, UK, 1995.
- [HFJ] D. Hougen, J. Fischer, and D. Johnam. A neural network pole balancer that learns and operates on a real robot in real time.
- [How] Pierce J. Howard. *The Owner’s Manual for the Brain: Everyday Applications from Mind-Brain Research 3rd Edition*. Bard Press TX.
- [Hum] M. Humphrys. Neural networks in rl. website <http://www.computing.dcu.ie/~humphrys/Notes/RL/neural.html>.
- [I.06a] Millington I. *Artificial Intelligence for games*. Morgan Kaufmann, 2006.
- [I.06b] Millington I. *Artificial Intelligence for games*. Morgan Kaufmann, 2006.
- [img] website <http://wireninja.com/ubisoft-giving-assassin\OT1\textquoterights-creed-2-pc-owners-a-free-game/>.
- [Ish94] Masumi Ishikawa. Structural learning with forgetting. *Neural Networks*, 1994.
- [JJP<sup>+</sup>95] Mair J., Smidt J., Lechleitner P., Dienstl F., and Puschendorf B. A decision tree for the early diagnosis of acute myocardial infarction in nontraumatic chest pain patients at hospital admission. 1995.
- [Koz92] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, 1992.
- [Leh85] Wm Lehler. definition of ai. Newsgroups: net.ai, 1985.
- [LH07] S. Legg and M. Hutter. A collection of deffinitions of intelligence. Technical report, IDSIA, Switzerland, 2007.
- [MA93] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, pages 103–130, 1993.
- [mem10] Types of human memory. <http://en.wikipedia.org/wiki/Memory>, 2010.
- [MPR98] P. McLeod, K. Plunkett, and E. T. Rolls. *Introduction to connectionist modeling of cognitive processes*. Oxford University Press, 1998.
- [MS] J. Matas and J. Sochman. Waldboost frontal face detector demo. <http://cmp.felk.cvut.cz/demos/FaceDetection/index.php>.
- [MSea93] V. Marik, O. Stepankova, and J. Lazansky et al. *Umela inteligenca 1 (CZ)*. Academia, Prague, CZ, 1993.
- [MSea03] V. Marik, O. Stepankova, and J. Lazansky et al. *Umela inteligenca 3 (CZ)*. Academia, Prague, CZ, 2003.
- [Ota10a] M. Otahal. Automatic software repair with genetic programming (cz). 2010.
- [Ota10b] M. Otahal. Reinforcement learning cheat sheet. 2010.

- [Ota10c] M. Otahal. Swarm intelligence for black-box optimization. 2010.
- [RN03] S. J. Russel and P. Norvig. *Artificial Intelligence a Modern Approach*. Prentice Hall, 2nd ed. edition, 2003.
- [Sal93] Marcos Salganicoff. Density-adaptive learning and forgetting. 1993.
- [SB98] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.
- [Sch91] R. Schank. Where’s the ai? *AI magazine*, page 38–49, 1991.
- [SLZ04] Kaile Su, Guanfeng Lv, and Yan Zhang. Reasoning about knowledge by variable forgetting. *American Association for Artificial Intelligence*, 2004.
- [Sou10] G. Sourek. *Visual design of Evolutionary Algorithms*. 2010.
- [spe09] Lifeline, game. <http://www.mobygames.com/game/ps2/lifeline>, 2009.
- [Sta] S. Stahlberg. Android blues. image <http://www.androidblues.com/gallery/interview449.jpg>.
- [Svo] T. Svoboda. Car driving assistive technology. video/avi [http://cmp.felk.cvut.cz/cmp/courses/Y33ROV/Y33ROV\\_ZS20082009/Lectures/./Videos/CMPdemos/rychlost.avi](http://cmp.felk.cvut.cz/cmp/courses/Y33ROV/Y33ROV_ZS20082009/Lectures/./Videos/CMPdemos/rychlost.avi).
- [SW06] J. Simpson and E. Weiner. *Compact Oxford English Dictionary*. Oxford University Press, 2006.
- [TzuBC] Sun Tzu. *The Art of War*. China, 6th cent. BC.
- [VP09] Venkatraman.S and T.V. Padmavathi. Speech for the disabled. [http://www.iaeng.org/publication/IMECS2009/IMECS2009\\_pp567-572.pdf](http://www.iaeng.org/publication/IMECS2009/IMECS2009_pp567-572.pdf), 2009.
- [wik10] Reinforcement learning. [http://en.wikipedia.org/wiki/Reinforcement\\_learning](http://en.wikipedia.org/wiki/Reinforcement_learning), 2010.
- [Xia07] ZHENG Xianrong. Research on lineal gradual forgetting collaborative filtering algorithm. 2007.

# Index

- artificial intelligence, 11
- Balancer, 26, seebalancing pole34
- balancing pole, 34
- bottom-up, 13
- computer vision, 16
- connectionism, *see* bottom-up
- controlling behavior, 17
- decision policy, *see* refpolicy
- decision trees, 16
- dual solutions problem, 34
- emergence, 13
- environment, 13
- experience, 21
- exploration, 25
  - greedy, 26
  - mixed, 26
  - random, 26
  - simulated annealing, 26
- exploration strategy, 23
- feedback, *see* reward
- forgetting, 32
  - abstraction, 34
  - almost zero values, 34
  - filtering, 33
  - generalization, 34
  - in RL, 33
  - similar states, 34
  - single action only, 33
  - unused actions, 34
  - unused states, 33
- games, 17
  - game AI, 17
- genetic programming, 16
- human-like, 14
- intelligence, 11
- inverted pendulum, seebalancing pole34
- learning, 21
  - in games, 21
  - machine, 23
  - reinforcement, 23
  - supervised, 23
  - unsupervised, 23
- learning rule, 23
- memory, 13, 21
  - long term, 32
  - short term, 32
- minimalism, 13
- model
  - mental, 13
- neuron, 13
  - neural network, 13
- optimization, 16
- path finding, 17
- perceptron, *see* neuron
- policy, 23
- pseudo-code
  - Q learning, 27
- Q value, 24
- real life usage, 15
- reinforcement function, 23, 24
- reinforcement learning, 23
  - action-state pair, 23
  - model, 23
  - reward, 23
- reward, 26
- RL, *see* reinforcement learning
- run, 25

- sensors, 13
- specific AI, 14
- speech
  - recognition, 15
  - synthesis, 15
- strong AI, *see* human-like
- swarm intelligence, 16
- symbolic AI, *see* top-down
- top-down, 14
- training, 25
- uncertainty, 24
- usages of AI, 15