Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Ondřej Pacovský

On-line learning in real-time environments

Evolutionary and Adaptive Systems Group, University of Sussex

Vedoucí diplomové práce: Dr. Emmet Spier

Studijní program: Teoretická Informatika,
Neprocedurální programování a umělá inteligence

Acknowledgements

Many thanks to all the people who helped me with this project, to name some of them:

Emmet Spier, for encouraging me to pursue my crazy idea of combining a random network, reinforcement and Hebbian principles and for helpful comments and advice.

Neil Robinson, for improving the text by zillions of the's, a's and other nice words.

Stefan Jacobs, for advice and for releasing his ReadyBots under GPL.

The millions of people around the Linux and other open soft, who steal their own man-hours and resources from their rich employers and give them to the poor.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Táboře dne 10. prosince 2005 Ondřej Pacovský

Table of contents

Table of contents 5

Abstract 9

Prologue 11

1 Introduction 13

1.1 Aim 13
1.2 Background 13
  1.2.1 Reinforcement learning 13
  1.2.2 Artificial intelligence in game development 14
  1.2.3 Game development in Artificial intelligence 15
1.3 Structure 15

2 General Methods 17

2.1 Environment 17
  2.1.1 Virtual world 17
  2.1.2 Flow in experiments 18
  2.1.3 Learner controller 18
2.2 Controller input and output 18
  2.2.1 Input 19
  2.2.2 Output 19
  2.2.3 Feedback 19
2.3 Performance measure 19
  2.3.1 Internal immediate feedback 20
  2.3.2 Global fitness measure 20
2.4 Evaluation 21
  2.4.1 Baseline controllers 21

3 Reinforcement Learning 23

3.1 Reinforcement learning techniques 23
  3.1.1 Reinforcement Learning Problem 23
  3.1.2 Q-learning 23
3.2 Application to our domain 24
  3.2.1 Input discretization 24
    Position 24
    Time 25
    Resulting states 25
  3.2.2 Output and parameters 25
3.3 Experiments 26
  3.3.1 Direct aim 26
  3.3.2 Random y-distortion 27

3.3.3 Random y-distortion $-z$ . . . 28
3.3.4 Summary . . . 29

4 Stimuli → Action ↔ Reward Network . . . 31

4.1 Basic principle . . . 31
4.2 Detailed algorithm description . . . 32
4.2.1 Definitions . . . 32
4.2.2 Action selection . . . 32
4.2.3 Feedback and adaptation . . . 33
Relation to Hebbian learning . . . 33
4.2.4 Whole algorithm . . . 34
4.2.5 Winner strength . . . 34
4.3 Stimuli . . . 35
4.3.1 Stimuli in SARN . . . 35
Single-stimulus case . . . 35
2-D example . . . 35
4.3.2 Towards automatic stimulant . . . 36
4.3.3 Echo state network . . . 36
Weight initialisation . . . 37
4.3.4 ESN stimulant - SARN on steroids . . . 37
4.4 SARN application . . . 38
4.4.1 The handcrafted transformer . . . 38
4.4.2 Parameters . . . 38
4.5 Results for basic experiments . . . 39
4.5.1 Direct aim . . . 39
4.5.2 Random y-distortion . . . 40
4.5.3 Random y-distortion $-z$ . . . 41
4.5.4 Concluding remarks . . . 41

5 Further experiments with SARN . . . 43

5.1 Number of stimuli . . . 43
5.2 Over-learning . . . 45
5.3 Environment changes . . . 47
5.3.1 Improving re-adaptability . . . 49
Aging . . . 49
5.3.2 Results . . . 49
5.4 Conclusion . . . 51

6 Case-study of SARN-ESN . . . 53

6.1 Observed behaviour . . . 53
6.2 Perceived stimuli . . . 54
6.3 Action weight development . . . 54
6.3.1 Important action weights . . . 55
6.3.2 Development of identified weights . . . 57
The “idle” action . . . 57
The “right” action . . . 58
The “left” action . . . 58
6.3.3 Summary . . . 59
6.4 Interpretation of the stimuli . . . 60

6.4.1 Direct interpretation 60
6.4.2 Linear regression 61
6.4.3 Input weights 62
The inhibited stimulus 22 63
Directional asymmetry - stimulus 38 64
6.5 Putting the mosaic together 64
6.5.1 Adaptation dynamics and action selection 65
6.6 Lessons 65

7 Discussion and future work 67

7.1 Discussion 67
7.1.1 Teacher involvement 67
7.1.2 Applicability 68
7.2 Future work 68
7.2.1 Combining with higher-level algorithms 68
7.2.2 Algorithm improvements 68
7.2.3 Stimulant 69
7.2.4 Analysis 69
7.2.5 Human opponent 69
7.2.6 Outputs 70
7.3 Conclusion 70

Bibliography 71

Appendix A Software architecture 73

A.1 Socket interface 73
A.1.1 Unreal → controller 73
A.1.2 Controller → unreal 73
A.2 Unreal side (UScript) 74
A.2.1 Comments on using Unreal Engine 74
Running the server 74
Experiments faster than real-time 75
A.2.2 Alternative engines 75
A.3 Client controller side (C++) 75

Abstract

Název práce: On-line learning in real-time environments
Autor: Ondřej Pacovský
Katedra (ústav): Evolutionary and Adaptive Systems Group, University of Sussex, UK
Vedoucí diplomové práce: Dr. Emmet Spier
e-mail vedoucího: emmet@sussex.ac.uk

Abstrakt: Práce se zabývá vývojem nového algoritmu pro zpětnovazební učení (reinforcement learning), nazvaného Stimulus → Action ↔ Reward Network (krátce SARN). Cílem je vyvinout algoritmus pro nasazení v reálném prostředí. To klade na použité techniky dvě hlavní omezení: řídící algoritmus musí pracovat se vstupy v oboru reálných čísel a učení musí probíhat za pochodu, bez předchozích trénovacích běhů. Dalším cílem je minimalizovat zásahy učitele (člověka) nutné pro úspěšné nazazení algoritmu pro daný problém. Architektura SARN kombinuje konekcionistickou síť a skalární zpětnou vazbu použitím hebbovských principů. Postupnou změnou vah v síti se tvoří vazby mezi relevantními vstupy (stimuly) a akcemi, které vedou k pozitivní zpětné vazbě. Protože použitý algoritmus je schopen rychle vybrat důležité vstupy, je možné použít vstupní prostor poměrně velké dimenze. To vede k myšlence použití náhodné rekurentní sítě pro před-zpracování vstupu. Prototyp byl testován ve virtuálním prostředí Unreal 2004. V porovnání s Q-learning vykazuje SARN v časové skále desítek sekund až jednotek minut typicky lepší výsledky. Zejména po spojení s Echo State Network vyžaduje SARN narozdíl od většiny srovnatelných algoritmů velmi málo zásahů od učitele nad rámec skalární zpětné vazby. Díky těmto vlastnostem je algoritmus použitelný například pro řízení autonomních robotů nebo protivníků v počítačových hrách.

Klíčová slova: real-time adaptation, on-line learning, reinforcement learning, Hebbian network, Echo State Network

Title: On-line learning in real-time environments
Author: Ondřej Pacovský
Department: Evolutionary and Adaptive Systems Group, University of Sussex, UK
Supervisor: Dr. Emmet Spier
Supervisor's e-mail address: emmet@sussex.ac.uk

Abstract: In this work, a novel reinforcement learning algorithm, Stimulus → Action ↔ Reward Network (SARN), is developed. It is targeted for application in real-time domains where the inputs are usually continuous and adaptation must proceed on-line, without separate training periods. Another objective is to minimise the amount of problem-specific teacher (human) input needed for successful application of the algorithm. The SARN architecture combines a connectionist network and scalar reinforcement feedback by employing Hebbian principles. By adapting the network weights, connections are established between stimuli and actions that lead to positive feedback. Since the links between the input stimuli and the actions are formed quite rapidly, it is possible to use a large number of stimuli. This leads to the idea of using recurrent random network (Echo State Network) as a pre-processing layer. Prototype implementation is tested in Unreal 2004 game environment. The comparison with Q-learning shows that on the time scale of tens of seconds to minutes, SARN typically achieves better performance. When coupled with an Echo State Network, SARN requires a uniquely low amount of problem-specific information supplied by the teacher. These features make SARN useful for domains such as autonomous robot control and game AI.

Keywords: real-time adaptation, on-line learning, reinforcement learning, Hebbian network, Echo State Network

Prologue

The question is not whether a human brain can understand human brain. It is whether the human brain can understand women.

-- op

Chapter 1
Introduction

1.1 Aim

The main goal of this is to an develop algorithm usable for on-line reinforcement learning in real-time conditions. The focus is given on reducing the teacher-supplied problem-specific information. A pilot implementation is done in the Unreal game engine environment. The intent is to show what benefits can the game industry gain from artificial life research and also how existing game software tools can be effectively used for research.

By on-line learning in this context we mean that there are no ‘‘dry run” trials for the agent before he is exposed to the final environment. The agent has to learn ‘‘on the run” and exploration will potentially be costly for him. During the run, the environment cannot undergo perturbations and thus the learning process never ends. This is quite different from the more traditional ‘‘train $\rightarrow$ test $\rightarrow$ deploy”approach. In on-line learning, the speed and progress of learning has importance on the agent performance. This work focuses on time scales of minutes. This represents the period within which an adaptive controller should be able to significantly improve its performance in given environment conditions.

Real-time usage of the learning algorithm usually has two consequences. One is that the algorithm must be fast enough to perform learning and control given resources available in each time period. The other is that the controller inputs are usually real-valued and may carry significant amount of noise.

1.2 Background

In real-time learning, there is usually no teacher that can provide exact input-output examples. That is why this work deals with with reinforcement learning. The related work in this field is outlined in the following subsection, 1.2.1. Afterwards, the relation between academic artificial intelligence and AI used in computer games is discussed.

1.2.1 Reinforcement learning

Application of common reinforcement learning techniques such as Q-learning[21] in real-time environments has two major problems: the need of discrete state space and the convergence time.

A widely used approach to overcome state discretization problems is to approximate Q function by a neural network. This approximation, together with storing representative examples was successfully used in [5] to control an under-water robot in real-time. Another example is [13], where the Kohonen network was used to discretize the input space. Some approaches such as stochastic reinforcement learning (SRV) [6] are also able to cope with real-valued output (action) space.

To reduce the size of the learning problem and thus enable faster convergence times, various techniques of problem subdivision were exist. The HASSLE algorithm[3] restricts the large state-spaces by automatically dividing policies into hierarchies before applying Q-learning. The advantage is that after the division, the individual problems (which have considerable smaller action-state spaces) can be solved independently. As the division is not coupled with the learning itself, there is a risk that the resulting policies will have inferior performance to the one that can be found for the original problem. Another approach to automatic discovery of policy hierarchy is Concurrent HEX-Q[16]. There, the task hierarchy is not fixed during the learning process and division can be changed in spite of new evidence received.

1.2.2 Artificial intelligence in game development

Current computer games use some artificial intelligence techniques. This is natural, because games create virtual environments that to some extend mimic the real world. And these environments are often inhabited by virtual creatures, animals, monsters and humanoids. In order for these creatures to match the player expectations, some level of autonomy is given to them. Most often these are carefully pre-scripted sequences that are followed without much awareness of the actual state of the environment. Even in this case, all the code related to the artificial living creatures is called artificial intelligence (AI). Because the term artificial intelligence has another meaning in academic world, we will use **game AI** for the former meaning and **academic AI** to refer to the academic field of artificial intelligence.

The computer games environment would seem as an ideal domain for applying academic AI. However game development has its specifics and current state of academic AI does not always provide appropriate solutions. Most notably:

deployment time. Application of an AI algorithm to a concrete problem usually requires lot of additional work. Analysing input and output data, pre-processing, post-processing, tuning of various parameters of the algorithm - these all require many man-hours of a skilled AI researcher.

high resource usage. Games are run on mainstream personal computers and dedicated consoles. The behaviour of creatures present in the game can take just a fraction of the computing power available - the rest is reserved for graphics, physics and game logic. However, as computing power increases and dedicated processors are used for some of these tasks, more resources can be allocated to game AI.

real-time. Most of the algorithms developed in academic AI are tested and used offline. At least until quite recently when AI began to be employed in real-world environments such as autonomous robot control

Because game industry nowadays is about marketing and profit, the additional resources spent on state-of-the-art AI techniques must be paid off by the added value. What are the expected advantages of employing novel AI techniques in computer games?

better user experience. Undoubtedly, playing against opponents that are able to surprise by novel and adaptive strategy is more enjoyable than against the ones that follow pre-defined scripts each time.

novel game types. Higher level of intelligent behaviour can enable completely new experiences that are currently only possible against a human opponent. Also, advanced AI may allow player to participate in the virtual environment on a completely new level. In [17] for example, the game consists of training a squad of agents.

less pre-scripting. The hypothesis is that self-aware agents need less supervision and thus less pre-defined scripted behaviour from the game designer. If agents are able to manage their needs such as survival by themselves, their role in the game could be specified in more general terms. This is subject to controversy, because in the current state of AI, autonomous controllers quite often cause more trouble than they solve. They sometimes get stuck in some way or find a solution which is "too different from what the designer expected".

1.2.3 Game development in Artificial intelligence

The other side of the coin is that game environments can provide a good test-bed for a wide variety of artificial intelligence techniques. Of all game types, two received most attention: multi-user First-person shooters (FPS) and real-time strategies (RTS).

As FPS games specialise on the fidelity of 3D environment, artificial opponents must be able to respond intelligently on a wide range of activities. From mid-level (movement, shooting, dodging) to high-level (team cooperation). A variety of AI techniques are being applied to this domain, including neural networks[22], logic[8] and multi-agent systems[9]. Whole systems were developed for interfacing with various FPS games, such as Gamebots [2][11].

The real-time strategies operate at different form of abstraction. A successful opponent must be able to prepare and coordinate both quick reactive plans and overall strategy. This involves planning, opponent modelling and other challenges[4][19].

1.3 Structure

In this work, we develop a novel reinforcement learning method suitable for real-time application and test it in game environment. The rest of the report is organised in 6 chapters.

In chapter 2 we describe the environment and performance measures used for experiments. In chapter 3, we present results for Q-learning applied to the three basic experiments.

After identifying the main problems of conventional reinforcement learning techniques applied to real-time problem, a novel reinforcement learning algorithm, SARN, is introduced in chapter 4. The initial results are interesting and therefore we try to identify the limits of SARN on further experiments in chapter 5. Since the inner dynamics of SARN, especially when coupled with Echo State Network are quite rich and hard to understand, we analyse the learning dynamics in chapter 6.

In chapter 7, we conclude and give outline of possible future directions. Appendices contain the implementation details, developer documentation and source code.

Chapter 2
General Methods

In this chapter, we describe the environment in which algorithms operate, the experimental setting and the performance measures used for evaluation.

2.1 Environment

The system for experiments was developed in Unreal 2004 game engine. Unreal was chosen because it frequent usage in recent game projects, it supports Linux and is easy to extend and modify. The software architecture and implementation is described in appendix A.

2.1.1 Virtual world

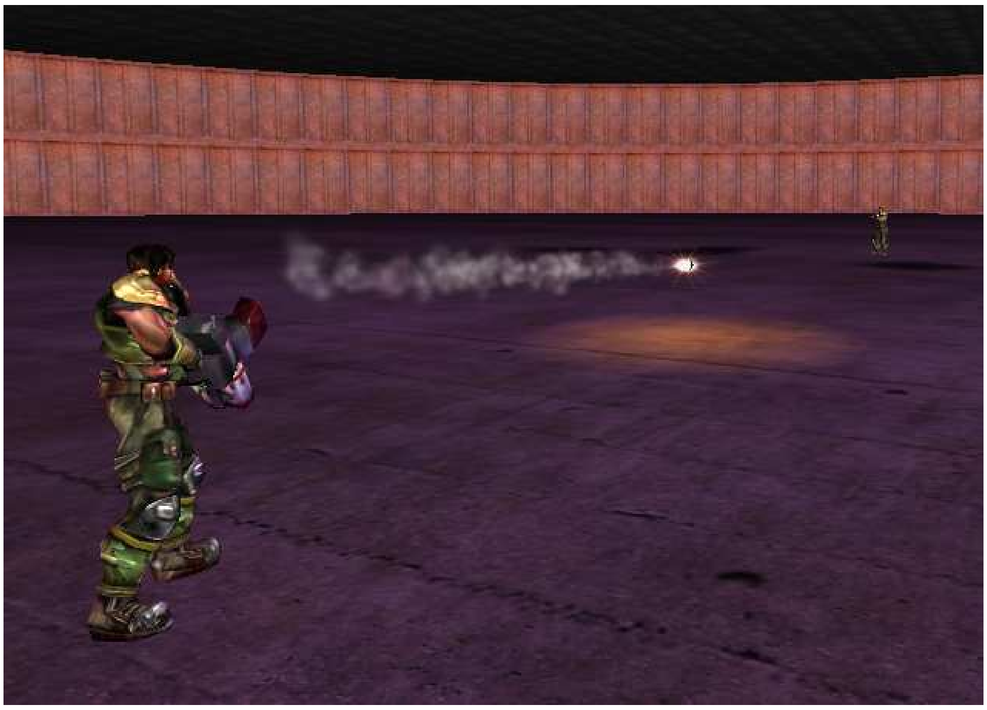

Figure 2.1. Screen-shot from the experiment in Unreal environment. (Movie available at http://op.matfyz.cz/sarn)

The environment created for experiments is an arena with a flat surface, large enough that the walls do not affect the experiments. There are two simulated avatars: the learner and the attacker.

The Attacker is equipped with a rocket launcher and can fire rockets. A rocket is simulated as a relatively slow projectile which explodes as soon as it hits an avatar, ground or wall. The wall is sufficiently far away that those explosions do not affect the learner avatar. For the purpose of the experiments the learner is given enough health so that he never dies during the experiment. Still, he perceives the impacts and the damage taken is used for feedback. The learner always faces the attacker.

2.1.2 Flow in experiments

Experiments progress as follows: The attacker and learner are placed at their initial places. The distance between them is set so that the attacker's rockets approach the learner in about one second, giving enough time for learner's avoidance maneuver.

Repeatedly during the length of the experiment the attacker fires four rockets with a five second interval. After each burst, the avatars are placed back to their initial positions. This is done because if the Learner is hit by a rocket, is is pushed back by the impact and distance from the attacker would thus constantly increase.

The attacker aims at the learner with pre-defined distortion. The distortion is selected as a 2D point relative to the learner's chest in the imaginary plane formed by legs-head and spread arms. The direction defined by arms is y (left is negative) and the one defined by a line between legs and head is z (head is positive, legs negative). We call this plane a **collision plane** later in the text.

The distortion is computed for every shot according to the given experiment. The simplest case is no distortion at all, where the rocket always hits the learner in the chest unless he moves. Another technique used in the experiments is to draw a point from a random distribution in some area around the learner.

All the statistics were collected using automated batches. The developed system also allows a human player to join the game and shoot at the learner himself. This is very useful for understanding effects of the learning algorithms. The graphics client also allows to oversee a running automated experiment.

2.1.3 Learner controller

The Learner avatar is controlled by a fixed-time step control loop. At each step, the controller receives current input, chooses action and this action is performed for the duration of the time step. The tick duration T between 0.1 and 0.25 was used, according to the controller used. For simplicity, only one action can be performed at a given time step.

2.2 Controller input and output

Most important for the controlling algorithm are the inputs from the environment, including feedback and the output describing actions to take. These are detailed in the following subsections.

2.2.1 Input

Because the only goal in the experiments described in later chapters is to avoid rockets, the controller inputs inform the learner about approaching projectiles. In the automatic experiments, there is always at most one projectile relevant to the Learner. There might be other projectiles in the arena, for example those which missed the target and are flying towards the wall, but those are not relevant to the learner. For this reason, the input is limited to only one projectile, the one fired most recently. For this projectile, the intersection with the learner avatar's collision plane (defined in 2.1.2) is computed and following values are sent:

- p - boolean value specifying there is a projectile (if there is not, other values are zero)
- t - time to impact the collision plane
- y, z - target-centric coordinates of impact in the collision plane

2.2.2 Output

The actions the learner avatar can take in order to evade the rockets are strafe left and strafe right. The strafing is done in circle around the attacker, thus learner always faces attacker. In addition the learner may choose not to perform any action at a given time tick, thus remaining idle. The three basic actions are summarised in the following table:

identifier	action
0	idle
1	strafe right
2	strafe left

There is one more rationale for introducing idle action. When the avoidance component would be used together with other modules, it would be inactive most of the time and only overtake control for evasion maneuvers.

2.2.3 Feedback

The controller is informed about the health status of avatar each time tick. This gives internal feedback about avoidance performance. When learner avatar is hit by a rocket, the health is decreased by a up to 100 points, depending on distance of the avatar from the explosion. Direct hits have maximum penalty.

2.3 Performance measure

In this section, we define internal and external performance measure. The former is used directly in the controller for learning, while the latter is used for comparison with other controllers, as is further described in the next section, 2.4.

For time interval $I = [t_1, ..., t_n]$, let us define $d(I)$ as the health difference for this period. Usually, $d(I) \leqslant 0$, because the learner takes damage (or keeps the same health at best). Additionally, we define $a(I)$ as the number of non-idle actions performed in interval I.

2.3.1 Internal immediate feedback

Because missiles are fired in 5 second intervals and the maximum damage that a single missile can cause is 100 health points, the worst average damage is $\frac{100}{5} = 20$ health points per second. If we want to give the projectile avoidance twice as much priority than the requirement to move as little as possible, we choose the scaling constant for a to be $10T$. So if the agent were to move all the time, he would receive an average penalty of 10 per second.

For the actual value used for reinforcement, we use linear combination of a and d:

$$r(I) = -d(I) - 10Ta(I)$$

The interval I in this case is one time step. As can be seen from the definition, $r \leqslant 0$ in all situations.

This is an **internal** fitness measure because the agent himself perceives the damage taken and of course has a knowledge of his own actions. In some cases it might have been helpful to reinforce him for avoiding missiles, that is, for not taking damage. This would make the learning easier, because the link between evasion action would be clearer. Without positive feedback, the only link is being punished when evasion action is not taken.

2.3.2 Global fitness measure

For off-line analysis after an experiment was performed, we define global **fitness measure**. It is designed to compare different controllers, taking into account maximal possible damage. This gives a more objective measure with known theoretical optimum.

Let us introduce:

$d_{\max}(I)$ maximum damage that can be caused in the experiment

$a_{\max}(I)$ total number of actions performed

Now we can define the damage fitness component as

$$D(I) = \frac{d(I) - d_{\max}(I)}{d_{\max}(I)}$$

and the action component as

$$A(I) = \frac{a(I) - a_{\max}(I)}{a_{\max}(I)}$$

Both components are in the interval [0; 1]. Value of 0 means worst and 1 means best result of given criteria (taking least damage or not moving). Using these, we define the **fitness** as:

$$p(I) = D(I)A(I)$$

By taking the multiplication of damage and action component, the controllers that successfully fulfil **both** requirements will receive highest fitness. Satisfying only one condition will not help much. For example, if the controller were to avoid all missiles ($D \cong 1$) at the expense of extensive movement ($A \to 0$), the result will be still close to 0.

The interval I used in association with fitness is either the whole experiment, giving overall performance or some part of it. This enables tracking of progress of fitness development during the experiment.

Note that the same value of $d_{\max}$ is used for all experiments. It specifies the maximal theoretically achievable damage when every rocket would directly hit the learner avatar. In some experiments such as direct-aim (Section 3.3.1), controllers can really approach this maximum. However in most other experiments, the $d_{\max}$ is not achievable. Due to this, we cannot directly compare d (and in turn, p) values for different experimental settings. What we can do is to compare the performances relative to some baseline controller, as explained in the next section.

2.4 Evaluation

Having defined performance measures and the overall experimental environment, we can now describe how controllers were compared. Different controllers are evaluated during a period of automatic test, usually 2 minutes. Because each missile not avoided has quite high impact on the overall fitness, the fitness of individual runs is quite noisy, requiring multiple runs for reliable results.

The measure used for comparison is fitness, p as defined in 2.3.2. Two quantities are computed for a different interval

- **overall**, where the measured interval I is the whole run. This is the most important measure as it captures overall controller performance - faster learning algorithms with least time spent on penalised exploration will have best overall fitness
- **final**, where the measured I is the last 20 seconds of the experiment. This gives the performance after the whole learning period, no matter how costly the learning was.

2.4.1 Baseline controllers

Because for most of the performed experiments, a simple controller performs quite well, the measured controller is always compared to two baseline non-adaptive solutions: **idle** and **random**.

The idle controller simply performs no actions. It is thus likely to receive all targeted shots. On the other hand, it receives no discounts for moving.

The random controller randomly chooses an action to perform. This is quite a good strategy for avoiding hits, but results in large penalties for moving. When only one of three actions is idle, random controller will always perform two thirds of non-idle actions, obtaining the action fitness component of $A \cong \frac{1}{3}$.

Chapter 3

Reinforcement Learning

In this chapter we apply Q-learning on the described problem of missile avoidance. While some aspects of Q-learning are not suited for our problem, it will give us some insight into the problem difficulty. We will also try to identify weak spots and use this information in the design of the new algorithm in chapter 4.

The first section gives overview of reinforcement learning methodology and quick description of the Q-learning algorithm. Reader familiar with Q-learning can skip to section 3.2, where discretization and parameters for our domain are discussed.

3.1 Reinforcement learning techniques

The reinforcement learning is a specific type of learning where the teacher only supplies a reward or punishment for actions performed by the agent. This is as opposed to supervised learning, where the teacher also supplies the correct solution for example cases.

3.1.1 Reinforcement Learning Problem

The vast majority of work in the Reinforcement Learning field is done within the Reinforcement Learning Problem methodology[18]. We will use a simplified deterministic version of the model for description. The agent-environment interaction proceeds in discrete time steps. At each time step, the agent perceives the environment state $s_t \in S$, reward for previous action $r_t \in \mathbb{R}$ and selects action to perform, $a_t \in A$. Environment state space S and action space are finite sets. The action to be performed by agent is selected by his policy. Policy $\pi(a,\ s)$ specifies a probability of performing action a at state s. The agent's goal is to find the policy which maximises reward accumulated over a long period.

3.1.2 Q-learning

The Q-learning algorithm[21] is one of most commonly used methods to solve Reinforcement Learning Problem. The learning process consists of estimating the action-value function Q. For a given state s and action a, the value of $Q(s, a)$ is the expected utility value for taking action a in state s. The utility of state s for policy π is defined in terms of discounted future rewards:

$$V^\pi(s) = E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s\} = \Sigma_a \pi(s,a)(R_s^a + \gamma V^\pi(S(s,a)))$$

where R_s^a is the immediate reward for a in state s, $S(s, a)$ is the state of the environment after performing a in state s and $\gamma \in [0; 1]$ is the discount factor.

The resulting algorithm is as follows:

```
Q-learning
   1. initialise $Q$, set $s$ to initial state
   2. repeat until terminal state is reached
   3.     choose action $a$ from state $s$, derived from current policy $Q$
   4.     perform $a$, observe $s'$ and receive reward $r$
   5.     $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
   6.     $s \leftarrow s'$
```

Figure 3.1. Q-learning algorithm

The action selection on line 3 is performed via a trade off between exploitation (selecting the best action estimated by current Q, that is $\pi(s) = \text{argmax}_a Q(s, a)$) and exploration. In our case, the ε-greedy action selection was used: perform random action with probability ε, otherwise follow the optimal policy given current estimation.

As it can be seen, Q-learning can be applied even when the optimal policy is not currently applied (off-policy). This is important for our experimental setting where there is no separate training phase and controller must learn on the run.

3.2 Application to our domain

3.2.1 Input discretization

Q-learning as well as most other reinforcement learning algorithms requires the input state space to be discrete. Because inputs in our problem come as continuous values, we need to discretize them. As described in Section 2.2.1, the input from the environment contains information about a single projectile: time to impact and coordinates of expected hit position relative to the learner.

Only the y (left-right) component of the hit position is used in the discretization, because it is most relevant to the avoidance task. The height component z has less impact on the outcome in most of the experiments.

That leaves us with state-space containing two continuous variables: time to impact (t) and position (y). One state (N) is reserved for situation where there is no projectile in the environment and the rest of the state-space is formed of the Cartesian product of time discretization and y position discretization. Each component will be now described separately.

Position

The value of y quite directly corresponds to the hit/no-hit result. The first attempts were to divide it into some number (5, for instance) of intervals to provide the controller with smoother information about the projectile position. This would be somewhat general, because it would be able to operate in different settings, such as different projectile sizes and avatar body sizes. However the adaptation speed of Q-learning was low when y states do not map exactly to hit/no-hit conditions, due to conflicting rewards.

The solution was to discretize y as close as possible to the actual experiment. In this case, the projectile hits the avatar if the y component is below 27 (in both left and right directions). This leads to the following 3-state discretization:

state	caption	condition
y_0	in hit zone	$\lvert y \rvert < 27$
y_1	safe - left	$y < -27$
y_2	safe - right	$y > 27$

Time

Time was discretized into four states. The Meaning of the first three, $t_0 - t_2$, is straightforward. As shown in the following table, they divide the time space into three intervals:

state	description	condition
t_0	projectile too far	$t > 0.8$
t_1	projectile closing	$0.8 \geqslant t > 0.4$
t_2	just before impact	$t < 0.4$
t_3	projectile lost	2 ticks after projectile disappears (see explanation in the text below)

What is the strange t_3 state? Time discretization originally contained only the former three states. However this lead to very bad performance and controllers were not able to improve even in long periods. It was discovered that the learning agent usually receives punishment for a hit in state N, when the info about the causing projectile has already disappeared. This made state N very confusing for the learning algorithm: Not moving in N means that I will take damage. Moving in N means I will be punished for performing far too many non-idle actions. Recall that a missile is fired once every five seconds and hits target in about a second. Therefore 80% of the time, bot is in state N.

To solve this problem, the additional "projectile lost" (t_3) time state was created. This state is set to be active two time ticks after info about current projectile is lost. The important thing is that y component of the information is saved so that it can be carried on to the t_3 state. This means that in most cases, the hit penalty is received in the t_3, y_0 state, while all the surrounding states (t_3, y_1, t_3, y_2, and N) are safe. The policy should then avoid state t_3, y_0.

Resulting states

In total, there are 13 states. This is 4×3 for the combined time and position states with the projectile and a single additional state for no projectile.

3.2.2 Output and parameters

The controller chooses among three **actions** (idle, right and left). The action to perform for each time tick is selected using ε-greedy policy. This means that with probability ε, the action is selected at random, instead of the one that has the currently maximal Q value.

The internal immediate feedback, r (health difference minus penalty for non-idle actions, defined in section 2.3.1), is used as the **reward**.

The exploration factor for ε-greedy action selection was 0.01. The learning parameter α was relatively high, 0.7, encouraging quick learning at the cost of possible sub-optimality. The discount factor γ was set to 0.3.

Time tick duration for the Q-learning controller was set to $T = 0.25$s. This was chosen as a tradeoff between fast reaction speed and lowest possible number of states responsible for the latest reward.

3.3 Experiments

The Q-learning controller performance was measured in three basic experiment types. Duration of these experiments is two minutes and each one presents a slightly different avoidance problem via a changed shooting pattern. Those experiments will be used to compare the Q-learning to other controller types in the following chapters.

3.3.1 Direct aim

In the **direct** experiment setting, the attacker always aims at the learner. In terms of the aiming distortions, where d_a^t denotes distortion in collision plane[3.1] axis a selected in time tick t:

$$d_y^t = 0$$

$$d_z^t = 0$$

This means that unless the learner moves, he will be hit. The most efficient strategy to avoid projectiles in this setting is to make exactly one or two steps to left or right each time the projectile approaches. The idle controller receives all the shots. The random controller performs quite well, avoiding most hits at the expense of extensive movement.

On figure 3.2 we can see a fitness plot, which will be frequently used in other chapters as well. It visualises the fitness development during experiments. Each point is the global fitness measure (f, defined in section 2.3.2) taken for 20 second interval, giving 6 points for the 2 minute period. The plot for each controller is averaged from 40 independent runs.

Mean fitness of the two baseline controllers, idle and random, is shown by dotted lines. The main result, combined fitness measure is plotted in full line. The two components of combined fitness measure are plotted as well, so that we can see how the behaviour developed. These are dashed line for damage-only fitness and dash-dot for action-only fitness. Note that the individual components are not comparable to the combined or summary plots - the overall fitness is a product of them and because they both are in $[0; 1]$ interval, the product will always be lower or equal than each of the components.

3.1. defined in section 2.1.2

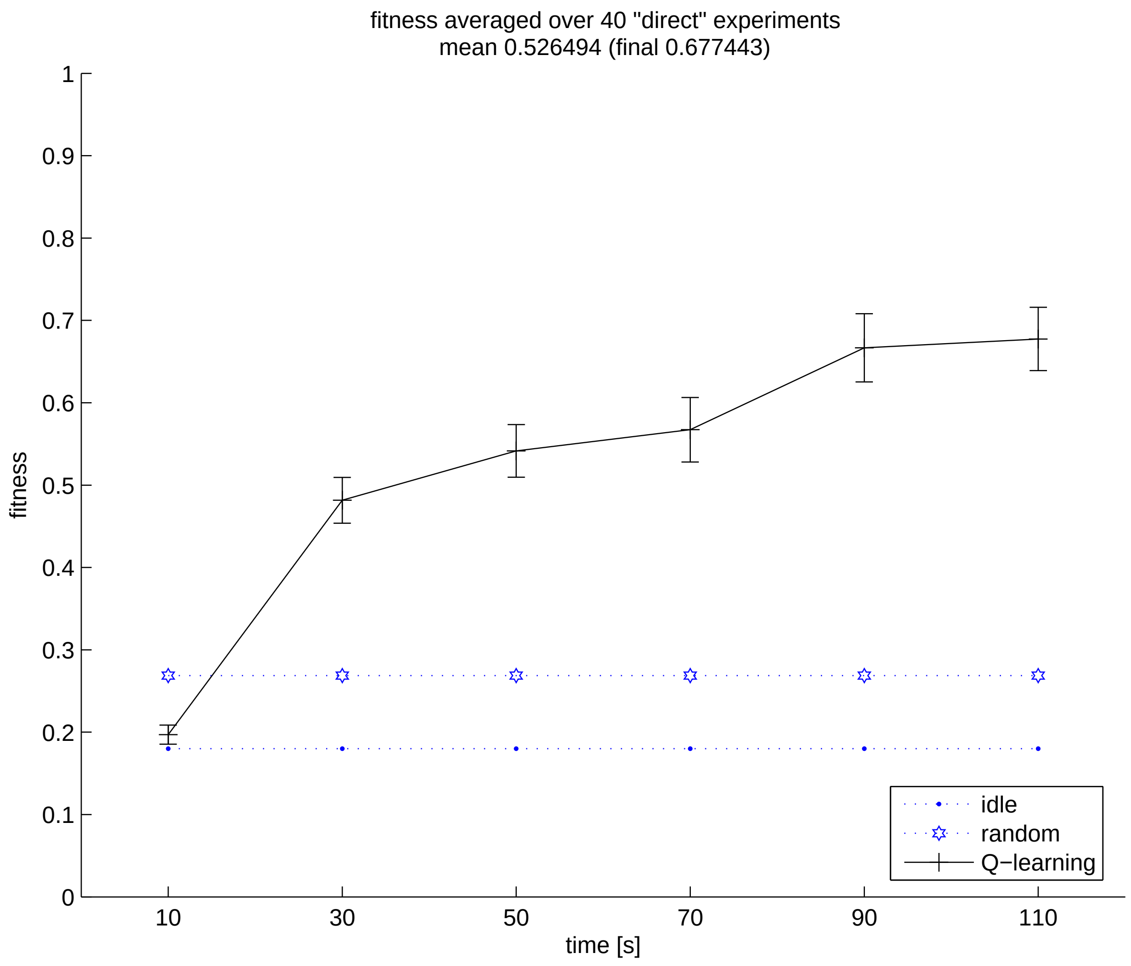

Figure 3.2. Fitness development of Q-learning controller in the **direct** experiment. The performance progressively increases up to almost 0.7.

As we can see on figure 3.2, Q-learning performance increases from initial 0.2 (worse than random) to 0.5 in the first 30 seconds and then continually improves, reaching almost 0.7 in the 2 minutes. This is a good result and given more time, performance could probably further improve.

3.3.2 Random y-distortion

The next experiment, abbreviated **yd**, introduces random distortion to the actual missile target. The missile target is moved away from the learner avatar by y_d^t, drawn from uniform random distribution over interval $[-50; 50]$. Roughly for $|y_d| \in 25$ the missile would hit the avatar if he stands on the same spot as when the missile was fired. The z distortion component is zero as it was in the **direct** experiment - the aimed height stays the same. In short,

$d_y^t = U([-50; 50])$

$d_z^t = 0$

Let us see how the Q-learning algorithm copes with this:

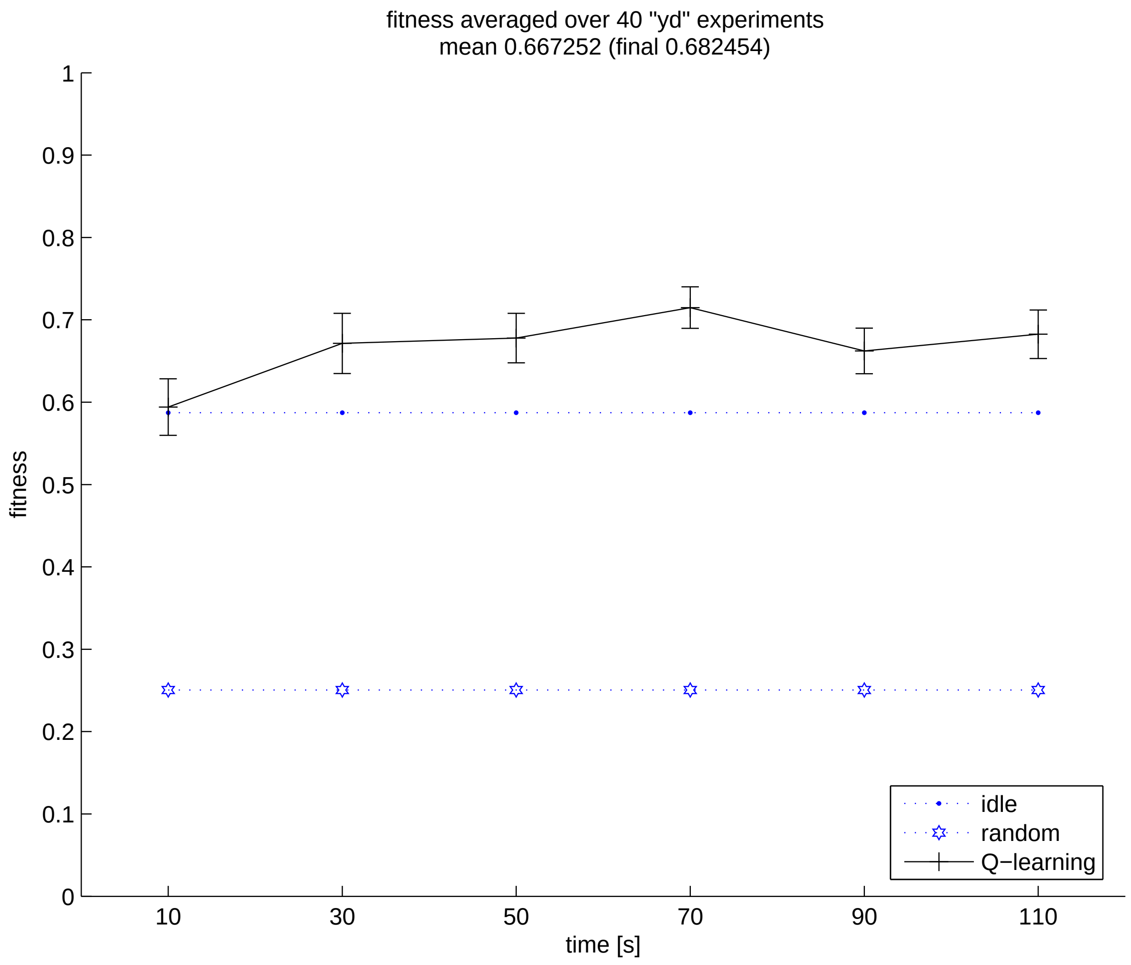

Figure 3.3. Fitness development of Q-learning controller in semi-random aiming experiment. Q-learning is able to slightly improve from the initial performance, which matches the idle baseline controller.

As we can see in figure 3.3, Q-learning controller starts off with the same performance as idle controller. Because every action is punished, it can learn to stay idle in a few iterations. However because some of the rockets hit the avatar, the controller will be forced to try to perform avoidance. We can see that during the course of the experiment, the algorithm is able to raise fitness up to 0.7. The boolean nature of the punishment (rocket either directly hits avatar, causing maximal damage or misses completely) might be difficult to interpret for the learning algorithm. To test this hypothesis, the next experiment will try to make the relation between avoided-y and hit damage smoother. This might help the learner to learn which states are “safe”, hopefully leading to better utilisation of idle actions.

3.3.3 Random y-distortion $- z$

To replace the boolean (hit/no-hit) rewards by a smoother measure, a quite simple modification can be made. The missiles in Unreal environment explode on impact, damaging objects in the surrounding sphere. When they explode directly on the target body, they cause maximum damage, but when the hit is not direct, the damage is proportional to the distance of avatar from the explosion. Thus if the attacker aims to the ground below learner avatar, the received negative feedback will be roughly proportional to the distance the learner travelled away from his original position. This might make the learning easier, because the agent is rewarded for slight progress by lower penalties. The described aiming distortions are thus:

$d_y^t = U([-50;50])$ (same as in yd)

$d_z^t = -120$

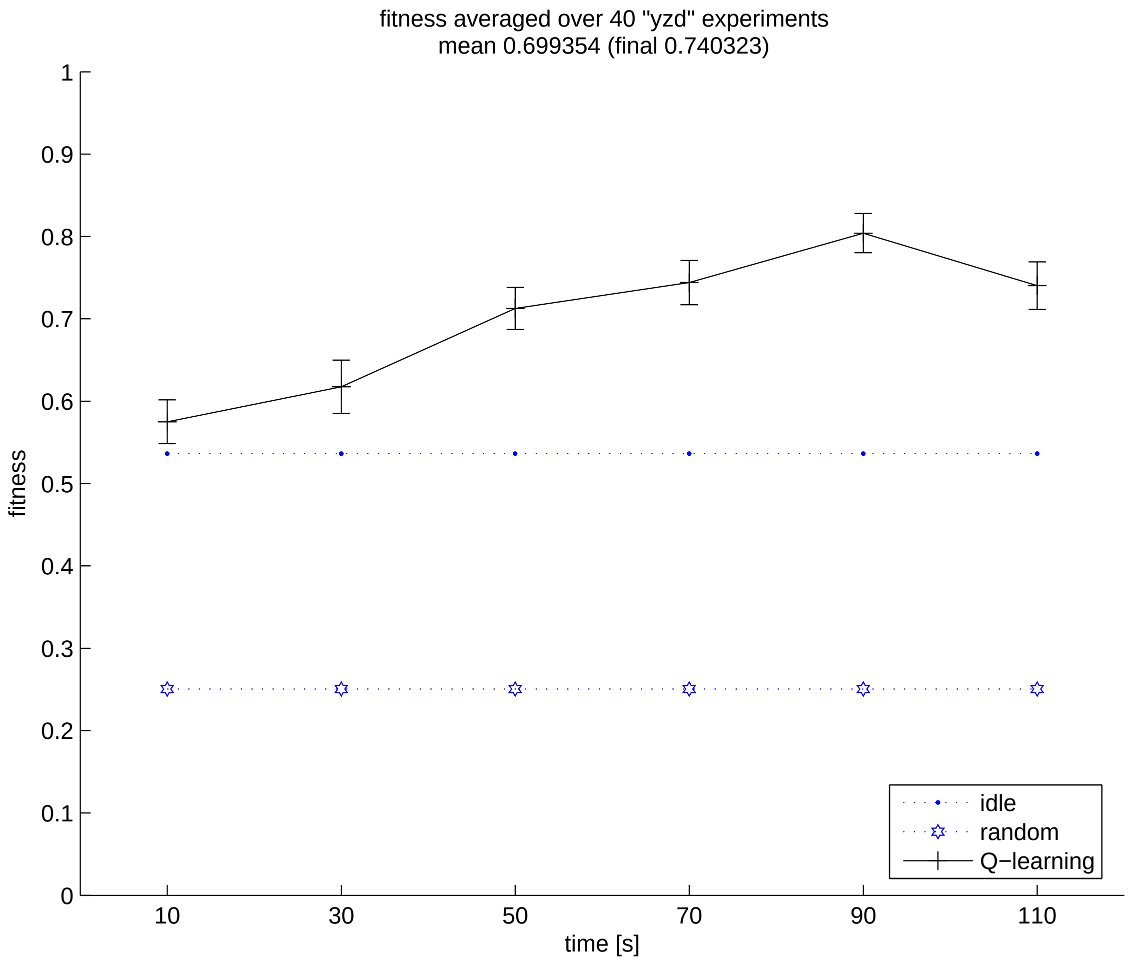

Figure 3.4. Fitness development of Q-learning controller in experiment where rockets are shot semi-randomly to the ground below learner, so that smoother damage feedback is received (yzd). Clearly this makes learning easier and the controller outperforms idle solution by larger margin than in the **yd** experiment

The results are shown in figure 3.4 above. Having the reward proportional to the amount of evasion actions taken helps the development of action effectiveness. The learning progresses further away from the baseline solutions. In 90 seconds, the fitness is improved by about 0.2 (Cr. less than 0.1 in the **yd** experiment). Interestingly, performance slightly drops in the last 20 seconds. This phenomena was not closely analysed here. Related over-learning issues will be discussed in section 5.2.

3.3.4 Summary

The Q-learning results are summarised in table 3.1.

experiment	idle	random	Q-learning	improvement
direct	0.180	0.269	0.526 (0.677)	1.960 (2.521)
yd	0.587	0.250	0.667 (0.682)	1.137 (1.162)
yzd	0.536	0.251	0.699 (0.740)	1.304 (1.381)

Table 3.1. Summary of Q-learning results. The three columns in the middle denote mean fitness of respective controllers. The improvement is measured as mean fitness of Q-learning controller divided by mean fitness of the better of the two trivial controllers (random, idle). Fitness is shown in the form `overall (final)` for Q-learning and for improvement.

The most important column is the improvement. It allows us to compare performance between experiments, because it is measured against baseline controllers in that experiment. The better of the two baseline controllers is always used for this comparison. We can see that the improvement of overall fitness in **yzd** experiment is over twice as large as the one in **yd** experiment. The smoother feedback indeed positively affected learning. The best improvement was accomplished in the easiest, most deterministic experiment where missiles are always targeted directly to the learner.

The behaviour after first two minutes was not explored here. Initial experiments indicate that fitness can further improve during a longer adaptation period.

We can conclude that Q-learning can be successfully applied to this task and significantly tune performance during the 2 minutes period. This involves only 24 fired shots, which can be compared to the number of trials in conventional Q-learning terminology. This is reasonably rapid learning and a good result, but we must be aware of the amount of teacher-supplied information contained in the state discretization. The achieved solution is highly specific to the experiments performed. We will try to address this issue in the next chapter.

Chapter 4
Stimuli → Action ↔ Reward Network

The application of Q-learning to our domain required great deal of preprocessing done by the teacher. In this chapter, we introduce a novel algorithm, SARN, which combines Hebbian and reinforcement learning principles. It moves down on the level of abstraction towards lower level, hopefully making it better suited for real-time tasks. The new algorithm uses a set of real-valued stimuli rather than discrete states. This is a key difference from standard reinforcement learning methods and enables use of automated preprocessing technique.

After introducing basic principles of SARN in section 4.1, we move onto a full description in section 4.2. The topic of obtaining stimuli for SARN is discussed in section 4.3. Both human-made and automatic approach based on Echo State Network is presented. This chapter is concluded by applying SARN to the missile avoidance task and testing it on the same experiments as Q-learning in the previous chapter.

4.1 Basic principle

Instead of working within the discrete states of the Reinforcement Learning Problem, SARN utilises feedback information to perform Hebbian learning between real-valued **stimuli** and the actions to perform. The flow is depicted on figure 4.1.

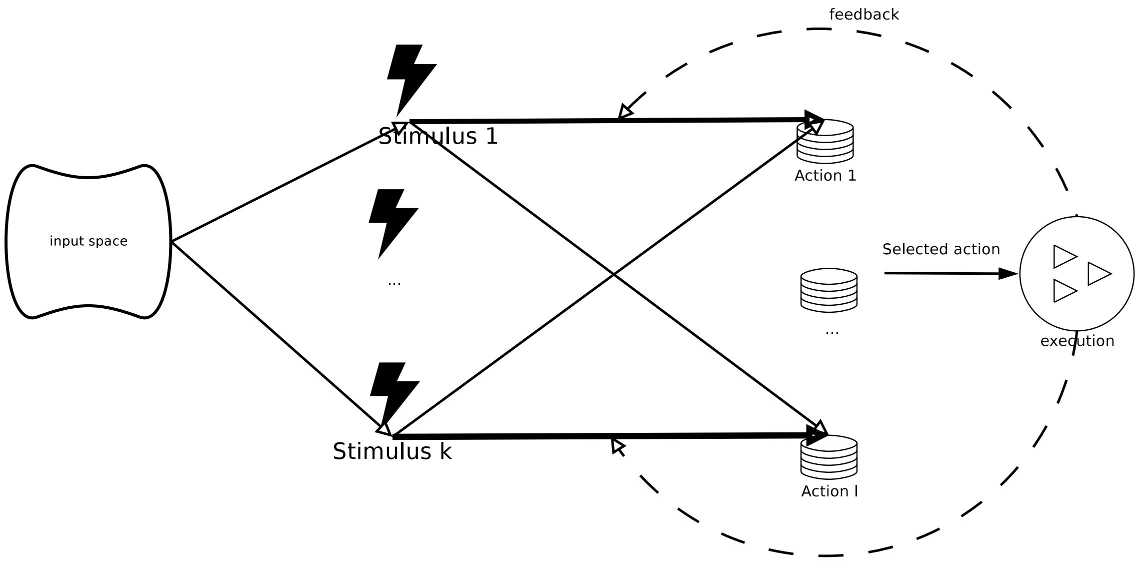

Figure 4.1. SARN overview. Input state is transformed into a set of real-valued stimuli. Using weights between stimuli and actions, the most active action node is selected to perform action. Feedback from action execution is used to modify weights in Hebbian manner. Further description in the text.

The input space is transformed to a given number of real-valued stimuli. The stimuli are connected to actions by links of a given weight (representing strength if the link). Given the current activity of the stimuli and weights, the activity of action nodes is computed. The action corresponding to the most active node is performed in the given time step. When feedback is received, the weights corresponding to the performed action and the stimuli that were most active at the time the action was selected are updated in Hebbian manner. If the feedback was positive, the strength is increased, otherwise it is decreased.

The next section contains a detailed algorithm description. The meaning of the stimuli and how they are obtained from the input state is then discussed in section 4.3.

4.2 Detailed algorithm description

We begin with basic definitions, and move on to action selection. Afterwards, weight adaptation is described and then whole algorithm is put together.

4.2.1 Definitions

Let there be n real-valued inputs, k stimuli, l actions and h specifying the size of tracked history. We will use the following terms:

input state. vector i in input state space, $i \in T \subset \mathbb{R}^n$

stimuli. represented as vector, s, in a k-dimensional space where each dimension is an interval, $s \in S = [-1; 1]^k$

stimulant. mapping $x: T \rightarrow S$ from input space to the stimuli space. In more general case, it can take previous input states into account as well: x: $T^m \rightarrow S$ for some $m \in N$.

action weight matrix. matrix w of size $k \times l$, with elements $w_{\text{sa}} \in [0; 1]$ (s denotes stimulus, a action)

history. array of size h, consisting of <stimuli, action> pairs

feedback. value in $[-1; 1]$ interval

learning parameters $\boldsymbol{\alpha}, \boldsymbol{\psi}$. α - traditional learning parameter, governing overall speed (strength) of adaptation, γ - reward discount factor. Both $\alpha, \gamma \in [0, 1]$

4.2.2 Action selection

The connection weights between stimuli space and actions (**action weights**) are stored in the action weight matrix. They are initialised to 0.5. Given the weights and current stimuli, action is chosen as follows:

```
int chooseAction(vector stimuli) {
 a = w * stimuli
 return maxarg(a)
}
```

As shown in the pseudo-algorithm, the action nodes' activity is computed by matrix multiplication of action weights and current stimuli vector. The most active node is selected as the next action.

4.2.3 Feedback and adaptation

SARN requires feedback to be in $[-1; 1]$ interval. Positive feedback is interpreted as a reward, negative as punishment.

The assignment of feedback to the actions that were performed is inspired by eligibility traces[18]. The basic principle is that the amount of responsibility for an action performed t steps back for current reward is given by ψ^t. The parameter $\gamma \in [0; 1]$ is called the discount factor. To be able to apply eligibility trace-based weights update, the performed actions, together with the current stimuli values are stored in the history array.

Each time step after the feedback has been received, the weights corresponding to the last h actions are updated. The amount of modification depends on learning parameter α and decays with a power of parameter γ as we go deeper into the history. This is accomplished in the `applyFeedback` function:

```
void applyFeeback(float feedback) {
 for i=0 ... h-1
  adapt(history[i].stimuli, history[i].performedAction, $\alpha\gamma^{i+1}$, feedback)
}
```

The function makes use of `adapt` function, which actually performs the weight modifications. `adapt` updates the weights corresponding to a given action, given a stimuli feedback:

```
void adapt(vector stimuli, int performedAction, float adaptationStrength,
float feedback) {
 w(performedAction,:) += adaptationStrength * feedback * stimuli'
}
```

Where `w(a,:)` denotes a-th row of matrix w [4.1]. The weights corresponding to performed action are modified to reinforce or weaken links to stimuli that were active, depending on whether `feedback` is positive or negative.

Relation to Hebbian learning

The `adapt` function in fact implements an (anti-)Hebbian learning rule[7].

Let us consider the case when feedback is positive and look what happens for different activations of stimuli.

- If a given stimulus s is positive, then $\triangle w_{\mathrm{sa}} = \mathrm{cs} > 0$ (where $c > 0$), strengthening the link between stimulus s and action a.
- When the stimulus is negative, meaning that it is not active at the given time, $\triangle w_{\mathrm{sa}} = \mathrm{cs} < 0$. In this case, the link between stimulus s and action a is weakened.

4.1. MATLAB notation

This corresponds to a Hebbian learning rule where $\triangle w_{\text{ij}} = \alpha\text{ij}$ when we take $i = s$ and $j = 1$, because we only adapt when action is performed.

With negative feedback, the argument is similar, leading to $\triangle w_{\text{ij}} = -\alpha\text{ij}$, which is the anti-Hebbian rule, weakening the active stimuli-action links and strengthening inactive stimuli-action links.

4.2.4 Whole algorithm

Putting together the described functions, the full control and adaptation loop of the SARN algorithm proceeds as follows:

```
SARN() {
 initialise w to 0.5
 while(running) {
  applyFeedback(getFeedback())      // perform adaptation for latest
feedback

  stimuli = stimulant(getInput())   // stimulant invocation
  action = chooseAction(stimuli)    // select action to perform

  history[0...h-1] = history[1...h]   // update history
  history[h].action = action
  history[h].stimuli = stimuli

  performAction(action)            // send next action to output
 }
}
```

We suppose that `performAction` blocks the SARN process until the next time ticks elapses.

4.2.5 Winner strength

The adaptation as presented so far works, but only on a short time scale. Once actions which lead to positive feedback are identified, they are reinforced each time they are performed and the corresponding action weights reach the limit, which is 1. This problem is somewhat similar to the weights over-growing in pulsed neural networks[1].

To prevent growing of the weights in situations where the relevant stimuli are already chosen and action weights already pick the preferred action by large enough margin, **winner strength** term is added to the adaptation strength calculation in the `applyFeedback` function.

```
void applyFeeback(float feedback) {
 for i=0 ... h-1
  adapt(history[i].stimuli, history[i].performedAction,
     $(1-$ history[i].winnerStrength$)\alpha\gamma^{i+1}$, feedback)
}
```

The winner strength parameter is computed after an action has been selected and is then added to the history array. Let us denote the winner strength as ω. The value of ω means how much the selected action was preferred against other actions. If $\omega = 1$, there was a big margin and the action is expected to be selected under the same circumstances (stimuli activations). If $\omega = 0$, it was a draw, meaning that even the slightest difference in stimuli may shift the odds towards another action. Let a be the vector of action node activities and m the index of selected action ($m = \text{argmax}(a)$). Now we define $b = a_m - a$.

If $\forall i\colon b_i > \omega_a$, the winner is said to be absolute and $\omega = 1$

Otherwise, ω is computed as $\omega = \omega_m \sqrt{\Sigma_{i=1}^{l} b_i}$

The key term in the formula is vector b that measures selected action activation against other actions. The remaining details of how winner strength is calculated are not very important. They were chosen so that the reinforcement of actions would drop rapidly as stimuli-action associations become settled.

4.3 Stimuli

What remains to be described before applying SARN is what the stimuli represent and how do we design the stimulant.

The stimulant is a very important part of SARN, because its action selection only has the power of a single layer Perceptron. Most tasks require higher processing power and if SARN is to be applied for them, the higher-level processing must be contained in the stimulant. In this section we describe the stimulant in more detail and progress towards an automatic stimulant, based on a random recurrent network.

4.3.1 Stimuli in SARN

The current values of stimuli are derived from the current input state, but input state and stimuli have certain differences. The first difference is that stimulus has pre-defined maximal and minimal bounds so that the degree of stimulus activity can be measured. The second difference is a crucial property of stimuli: For a given problem, the stimuli set must contain stimuli relevant to the actions. Let us first consider the case with a single stimulus in the stimuli set.

Single-stimulus case

Suppose that the input state space is split into two subspaces T_1 and T_2 and the maximum reward for the agent is achieved if he performs action a_1 when $t \in T_1$ and a_2 otherwise. In order for SARN to be able to find the optimal policy in this scenario, we need stimulant x such that $x(t) \geqslant c \Leftrightarrow t \in T_1$.

Of course, we usually cannot guarantee that given stimulant has those properties. If we could find such a stimulant, we would have the optimal controller at hand anyway. So what we try to do is to design (by hand or automatically) a stimulant which hopefully contains enough relevant stimuli the SARN can act upon. The advantage of SARN is that will automatically select the relevant stimuli out of the (possibly large) stimuli set.

2-D example

Let us illustrate the difference between state and stimulus on a simple example: the state consists of 2D coordinates of a ball in an arena (x and y) and we want the agent to execute specific action when the ball is in certain positions of the input space. Let $x, y \in [-1; 1]$. With a state transformer of identity, we obtain two stimuli. One is most active when the input is on the right and the other when the input is on the top. With the SARN action selector, we can only respond with actions on such as the ball is on the top, the ball is on the right and (by requiring both stimuli to be active) the ball is on the top right.

On the other hand, with the use of more sophisticated stimulant with more stimuli, our agent can trigger actions at virtually any region of the 2D input space and be localised as needed (one point, radial are with exponentially decreasing magnitude etc.)

4.3.2 Towards automatic stimulant

How do we obtain sufficiently rich stimuli set, for the particular problem?

The simplest way to obtain a stimulant is to design by hand. Using expert knowledge about the problem, we can identify parts of the state-space that seem important for action selection. This is quite similar to discretization in Q-learning. While it is good for analysis and preserves some tractability, handcrafting the stimulant has obvious disadvantages. Handcrafting is highly task-specific and will make it harder for the learner to operate in unpredicted situations and environment changes. Most importantly, it requires far too much human (teacher) intervention.

The task of transforming one continuous space to another is well suited for artificial neural networks. An imaginary network could take the input state as input and the activation of network nodes could be taken as individual stimuli. Using sigmoidal transfer functions and multiple layers, high computational abilities can be achieved.

However how do we find such network? How do we pick the topology and the weights? Standard supervised learning techniques for ANN are of no use, because the input-output examples are not available. Some unsupervised clustering methods could be of some use. Some thoughts in this direction are given in section 7.2.3.

It turns out that there is a type of ANN that might satisfy many of the requirements on stimulant. It is a random recurrent neural network with fixed weights and topology. Therefore it is in a way independent from the problem concerned and the richness of stimuli it provides is given by the size of the network. This idea was developed simultaneously by Maass et. al. (Liquid State Machine[12]) and independently by Jaeger (Echo State Network[15]). This work is based on the latter.

4.3.3 Echo state network

Echo state network is quite a recent novel recurrent network architecture. It utilises sufficiently large number of hidden nodes with fixed connections to provide a high-dimensional reservoir of activities reflecting the network inputs. The original Echo State Network also has read-out output nodes and training of the ESN consists of computing the output weights. Because output nodes are in fact a non-recurrent single layer of Perceptrons, the supervised learning can be done by computing pseudo-inverse. In this work, the output weights and nodes are not used and only activities of hidden nodes are important.

The ESN network comprises of n inputs and k nodes. The nodes are connected to inputs (W^{in}, $k \times n$) and recurrently to themselves (W, $k \times k$) via **recurrent weights**. Additionally, each node has a bias, denoted by vector θ.

The network operates in discrete time steps like a conventional recurrent neural network. The activation function used is tanh. Transition from one step to the next can be described by following equation:

$$s^{t+1} = \tanh\left(W^{\text{in}} i^t + W s^t + \theta\right)$$

i denotes input vector, s the hidden nodes' activity and tanh applies the hyperbolic tangent function to every item of the matrix. Superscripts denote time step.

Weight initialisation

The input weights and thresholds are selected randomly. Typically, 20% of them were set to non-zero, in $[-1;1]$ interval.

The recurrent weights are also initialised randomly, but not arbitrarily. Completely random recurrent connections could result in chaotic behaviour of the network. We need the network to respond the same way to the same sequence of inputs. After a number time ticks, the network should "forget" the previous history. This way, the activity of network reflects inputs for a number of time steps and then settles down, like echo.

This requirement on recurrent weight matrix is formally defined as Echo State Property[15]. For practical usage, the network with Echo State Property is generated as follows:

1. Create initial sparse matrix W' with p_r non-zero elements
2. Compute $\lambda_{\max}$, the eigen value of maximal absolute value
3. Re-scale W' so that $W = \frac{\psi}{\lambda_{\max}} W'$

The parameter ψ controls the amount of recurrence in W. It can be set in interval $(0, 1]$ and matrix W will still have Echo State Property. Higher value will cause network to have richer recurrent activity, thus having longer echo responses. Lower value enforces simpler dynamic. For this project $p_r = 0.05$ and $\psi = 0.7$ was used.

4.3.4 ESN stimulant - SARN on steroids

Echo state network as stimulant for SARN works as follows. Each time tick, the inputs from the environment are fed directly to the ESN (normalised if possible). One ESN iteration is performed, leading to a new activity vector. This vector is taken directly as the SARN stimuli set. Therefore each ESN node is taken as one stimulus. The tanh activation function's output is already in the $[-1; 1]$ interval, thus no scaling is necessary.

Note that the ESN stimulant does not generally perform mapping $T \rightarrow S$, because of its dependency on input history. The performed mapping is therefore $T^m \rightarrow S$ and the history size m can be controlled by the ψ parameter in the ESN weight matrix initialisation. Note that the action weights are the only weights modified during learning of SARN. The ESN stays static, as the handcrafted controller does.

The common size of the ESN core for experiments was chosen to be 40. From experiments with ESN[14] it appears that the cores begin to have ‘‘interesting”properties from size of about 30. Adding more nodes is usually beneficial for the network’s computing power. For example if the ESN is trained to approximate a time series, higher number of nodes will generally lead to better approximations. Important thing to note is that the weight initialisation technique is not well developed yet. Especially when using lower number of nodes such as 40, the ESN performance will vary from one network from another. For our purposes, we chose among twenty random networks depending on their performance on a simple arithmetics task. Such network is not particularly biased for our problem, but is in some way better than others. Developing better initialisation techniques is a subject of intensive research (see for example [20]).

4.4 SARN application

To apply SARN in our environment, we need to define inputs, feedback and outputs. The outputs are straightforward as they are the same as Q-learning used. These are the idle, strafe left and strafe right action.

As for the inputs, the four values from the environment are used:

- p - boolean value specifying there is a projectile
- t - time to impact collision plane
- y, z - coordinates of impact in the collision plane

For the ESN stimulant, these four values are just normalised and passed as inputs to ESN. The handcrafted stimulant is described in a separate subsection.

The feedback uses the same reward as used for Q-learning, but scales it to $[-1; 1]$ interval so that performing idle action while learner does not get hit is slightly beneficial (positive).

4.4.1 The handcrafted transformer

As in Q-learning, z component is not used. There are 4 stimuli related to p, t, y inputs:

stimulus	description	mapping
s_1	Is there a projectile?	p
s_2	Is it urgent?	$1 - t$
s_3	Projectile aims left	$\frac{y}{100}$
s_4	Projectile aims right	$-\frac{y}{100}$

Note that stimuli s_3 and s_4 are symmetrical, $s_3 = -s_4$. Both are supplied because SARN does not have the capability to negate the stimulus. Apart from that, the mapping is pretty straightforward. Not much effort was spent on improving the handcrafted stimulant to get better SARN performance. (Cf. discretization for Q-learning in 3.2.1 where additional memory state had to be created)

4.4.2 Parameters

Values of $\alpha = 0.1$ and $\gamma = 0.7$ were used in all experiments presented in this work. Time tick duration used $T = 0.1$s. If not stated otherwise, all SARN experiments employ the winner strength term with $\omega_a = 0.02$ and $\omega_m = 10$.

As in Q-learning application, there are no training and testing phases. The adaptation occurs during the whole life-time of the agent controlled by SARN.

4.5 Results for basic experiments

In the next sections, the performance of SARN will be compared to Q-learning results from section 3.3. Both the handcrafted stimuli and ESN stimuli version will be used. The SARN with handcrafted stimuli is referred to as **SARN-hand**. The SARN with automatic Echo State Network stimuli is denoted **SARN-*n*** where n denotes the size of the ESN (and therefore number of SARN stimuli).

4.5.1 Direct aim

The plots used for comparison are basically the same as described in Section 3.3.1. Together with baseline random and idle controller results, the fitness of Q-learning and two SARN controllers is plotted: the SARN-hand (with handcrafted stimulant) and SARN-40 (with ESN stimulant of 40 nodes).

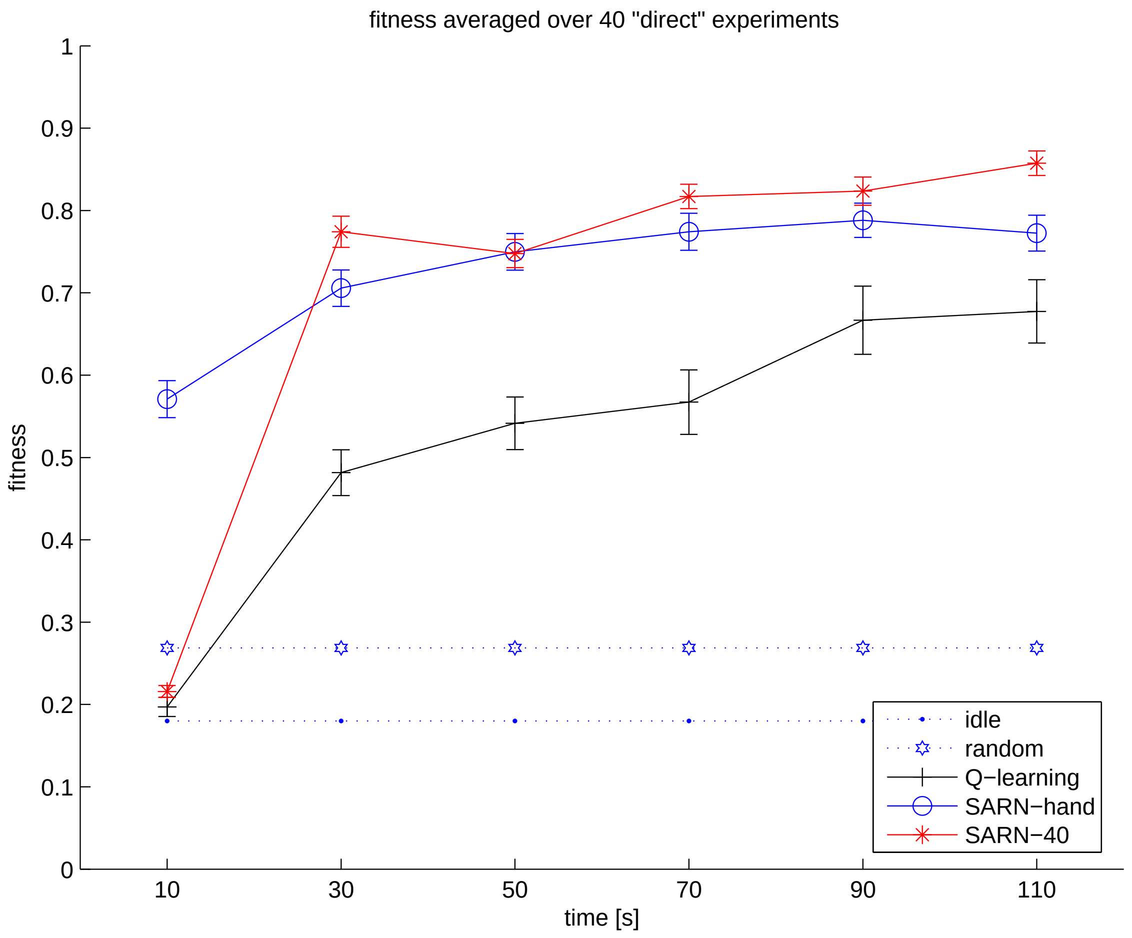

Figure 4.2. Fitness development in the **direct** experiment: SARN controllers compared to Q-learning. Both SARN controllers adapt quicker than Q-learning and reach higher fitness in the scope of 2 minutes. Note how SARN-hand adapts quicker, having high fitness from the start, while SARN-40 catches up later.

The first comparison shown in figure 4.2 is encouraging. SARN seems to be able to reach good fitness quicker than Q-learning. In the scope of 2 minutes, it also achieves better final fitness.

Another interesting result is that the SARN with automatic ESN stimuli reaches a better final fitness than the handcrafted-stimuli. The ESN-hand adopts very quickly from the start a good fitness of almost 0.6, whereas it takes 30 seconds for ESN-40 to reach that level. However after it has been reached, automatic stimuli performance is superior and further improves. It seems that the ESN reservoir can provide some stimuli that are more useful than the pre-designed ones.

4.5.2 Random *y*-distortion

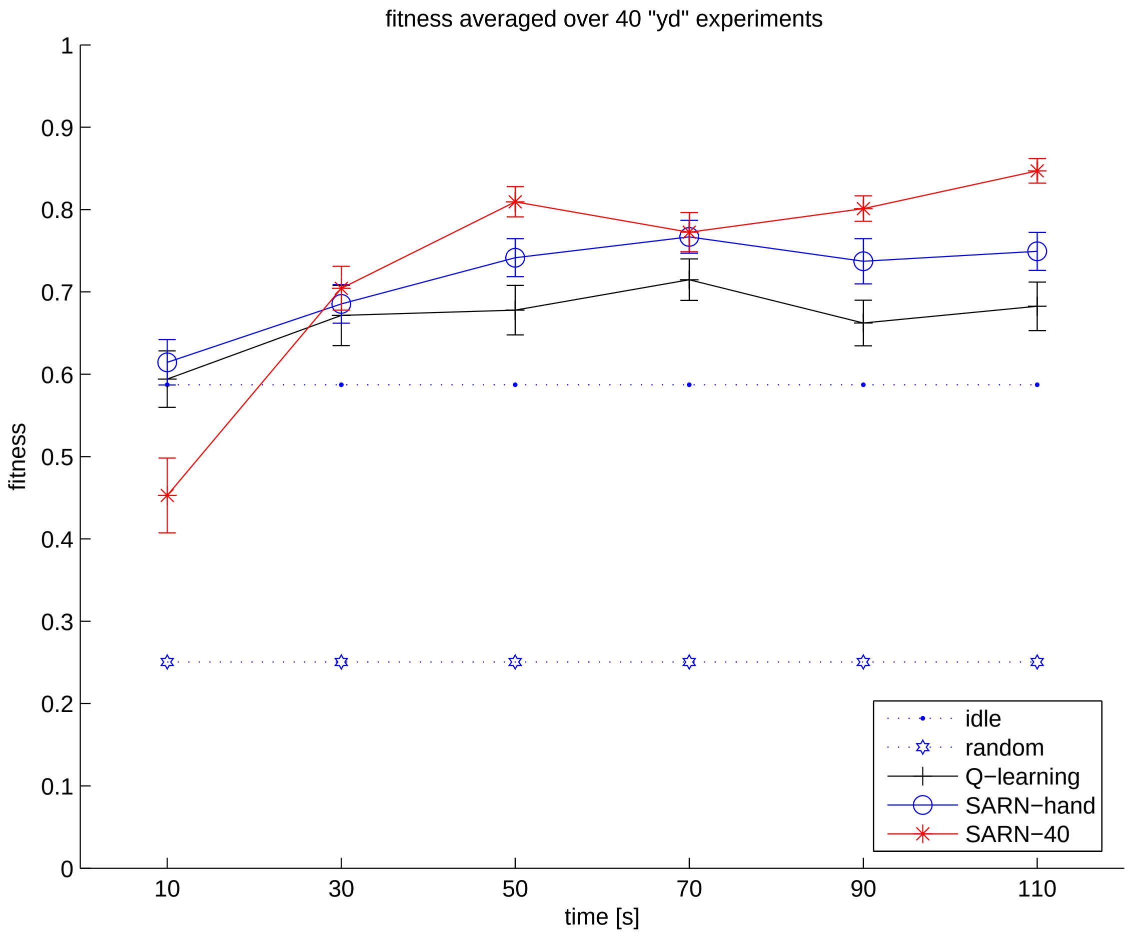

Figure 4.3. Fitness development in the **yd** experiment: SARN controllers compared to Q-learning. The trends from figure 4.2 still hold, both SARN are quicker to adapt than Q-learning in the scope of 2 minutes.

As can be seen on figure 4.3, the trends from previous experiments hold on the harder task of avoiding rockets with a distorted target. Let us see if the smoother feed-back of **yzd** experiment will enable SARN to further improve fitness as it did for Q-learning.

4.5.3 Random y-distortion $- z$

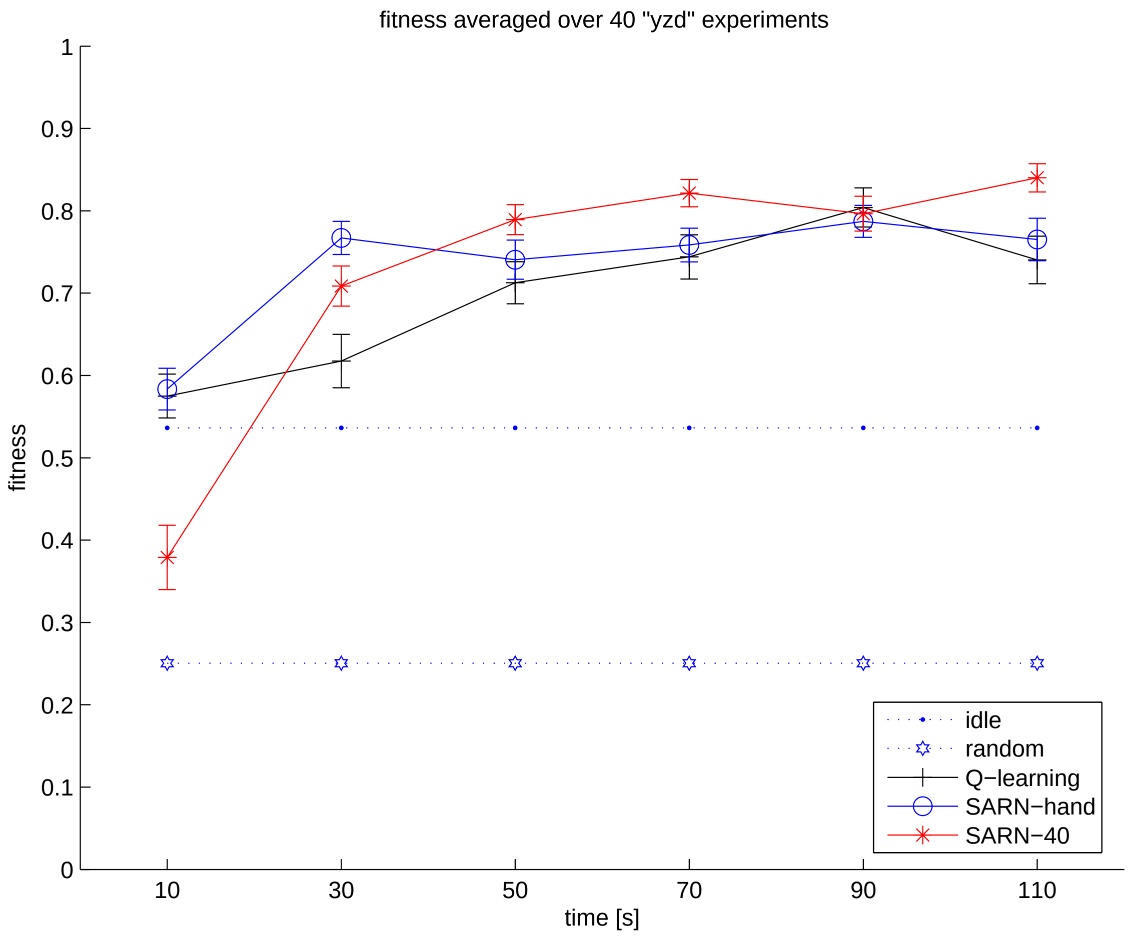

Figure 4.4. Fitness development in the **yzd** experiment: SARN controllers compared to Q-learning. No apparent improvement for SARN can be seen in this case.

As shown in figure 4.4, SARN results in **yzd** experiment are quite similar to these in **yd**. It seems that smoother reward assignment does not significantly affect SARN learning outcomes in this experiment. This makes Q-learning performance in this experiment comparable to SARN. The explanation why smooth reward feedback did not further improve SARN performance is as follows. SARN already works with continuous values when assigning feedback to stimuli. For example, when avoidance is based on "Projectile aims left" (s_3) stimulus in the SARN-hand case, the amount of punishment for a hit in this particular connection will be proportional to the activation of the s_3 stimulus. Due to the nature of the stimuli, SARN is biased towards proportional relations between stimuli and feedback.

4.5.4 Concluding remarks

experiment	idle	random	Q-learning	SARN-hand	SARN-40
direct	0.180	0.269	0.526 (0.677)	0.728 (0.772)	0.688 (0.857)
yd	0.587	0.250	0.667 (0.682)	0.718 (0.749)	0.722 (0.847)
yzd	0.536	0.251	0.699 (0.740)	0.736 (0.765)	0.712 (0.840)

Table 4.1. Summary of SARN results compared to Q-learning. First three columns are the same as in table 3.1. The additional two column show mean fitness (mean final fitness) for the two SARN controllers.

The results are summarised in Table 4.1. Both SARN controllers have a good overall and final performance, comparable or better than Q-learning in the scope of two minutes.

From the table we can see that mean fitness SARN-echo is superior to SARN-hand only in the **yd** experiment, despite that it reaches better fitness in the final periods of the run. This is because the ESN stimuli version needs a longer time to reach the "first acceptable strategy". When the total fitness is measured as mean over the whole period, it is worse for ESN stimuli. The speed of learning and different stimuli versions will be more closely analysed in Section 5.1.

The most important result is that the SARN with automatic stimuli performs well. When we compare the effort needed to develop handcrafted stimulant or even discretization in Q-learning, SARN-ESN shines. Of course, the avoidance task is not a particularly difficult one, especially when the inputs already provide information about projectiles in learner-centred coordinates. However even for these inputs at hand, developing discretization that enables Q-learning to find a good solution is quite tricky.

From the user point of view, the bot controlled by SARN is quite responsive to conditioning. Its behaviour is quite natural, even though the evasion strategies do not always match the ones we would expect. The controller exhibits some avoidance behaviour after the first shots and after one minute it is quite reliable. It is interesting to try to find weak spots of the evasion strategy and exploit them. Because the controller is continuously learning, it will try to further improve the strategy, based on these valuable examples. On the other hand, the algorithm has its flaws. For example when the bot receives several successive shots, the learned strategy is completely forgotten and bot enters a period of erratic movement until it cools down due to negative action feedback. Another problem is that the bot needs to be shot from time to time. Otherwise he forgets that being slightly punished for moving is better than standing on the spot and taking a full hit. One might argue that even intelligent creatures are lazy and need a whipping once in a while...

Results of basic experiments with SARN are encouraging, we will try to uncover more weak and strong features in additional experiments, commenced in the next chapter.

Chapter 5

Further experiments with SARN

In this chapter we will explore the SARN controller properties in additional experiments, uncovering some of its limitations.

In the first section, we try to identify the relation between number of stimuli and learning speed. The second section explores stability over longer time periods. Finally in the third section, the controllers undergo an environment change during the experiment.

5.1 Number of stimuli

The previous experiments indicated that the number of stimuli have an impact on the speed of learning. This is hardly surprising - more stimuli mean potentially more conflicting action triggers and choosing among those is likely to take more time. On the other hand, more stimuli may give more information and better chances to optimise the strategy. We analyse this phenomena closer by running the **yzd** experiment again. with different stimuli sets: handcrafted (4 stimuli) and ESN-based, with 40, 80 and 120 stimuli.

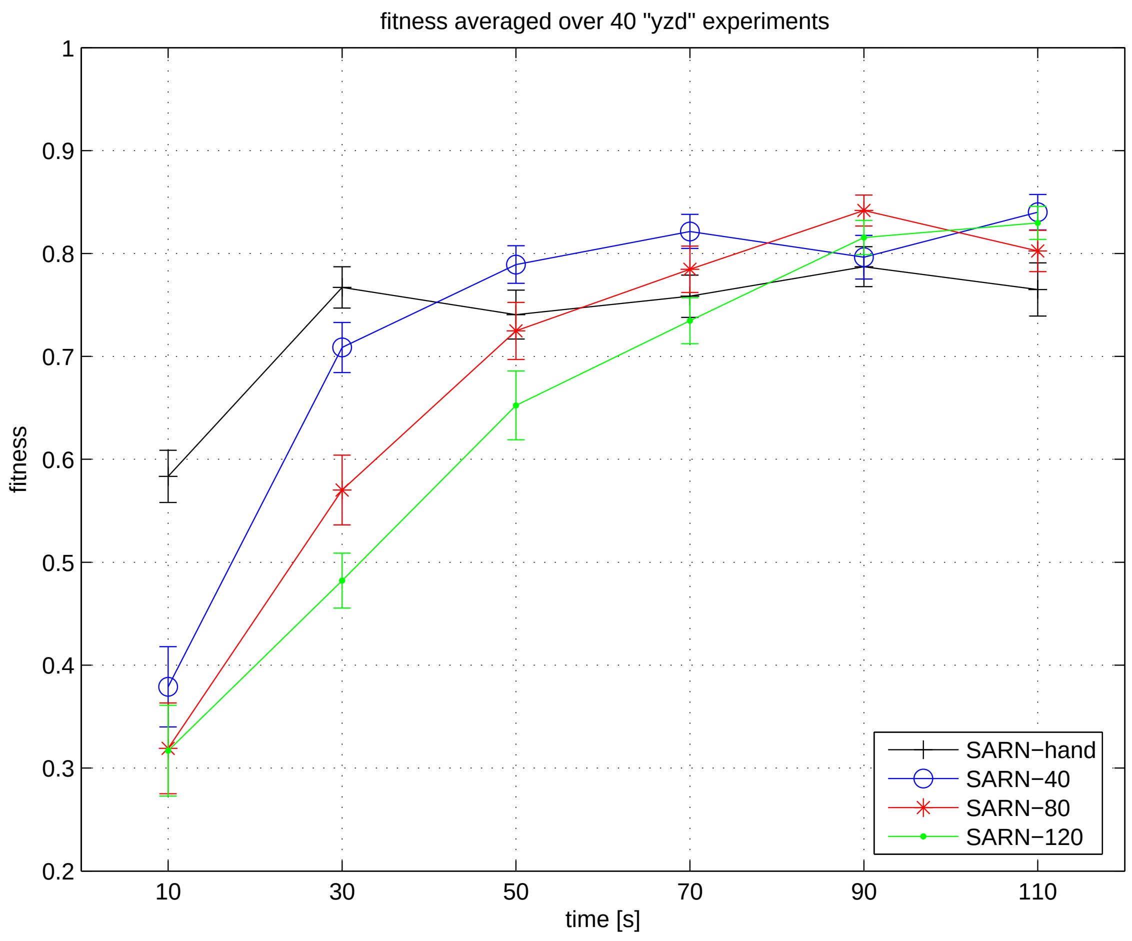

Figure 5.1. Test of different SARN stimuli set sizes on **yzd** experiment. The more stimuli, the longer does it take to reach certain performance level.

As we can see on plot 5.1, more stimuli require more time to enter the second adaptation period, where the change of fitness is not so dramatic anymore. Let $t(c, f)$ be the time needed for controller c to reach average fitness of f. If we look at $t(c, f)$ for $f = 0.7$ on the figure 5.1, we can identify roughly linear dependence of $t(c, 0.7)$ on the number of stimuli of controller c. Linear dependence would be an interesting property, but more experiments in other environments need to be performed to test that holds for other problems. Because the number of alone stimuli says nothing about information content in general, not much can be proven for the stimuli count alone. Some estimates could be obtained for generic stimuli sets. However the relation of these results to the real-world problems would be questionable. It is the fact that ESN has complex internal dynamics that makes it usable for hard problems.

Even the linear relation to number of stimuli does not necessarily guarantee good scaling. If harder problems would generally involve exponential growth of the number of stimuli towards the input dimension, the algorithm would soon reach its limits as Q-learning does.

A clearly positive result is that all three ESN-based automatic stimuli transformers outperformed the handcrafted transformer in the final fitness. This means that unbiased random network can prepare a richer reservoir of stimuli than those that were designed by a human teacher specifically for the problem. On the other hand, it does not seem that a bigger ESN would allow reaching of a higher final fitness than ESN of size 40, at least within the 2-minute time period used in the experiments. The experiments performed are relatively simple and have only four inputs, of which only three really seem to be relevant. Analysis on a wider range of harder and more diverse experiments would uncover more about the advantages or disadvantages of bigger ESN reservoirs.

5.2 Over-learning

The experiments performed so far were performed in the short time scale of 2 minutes, because short time adaptations is a crucial requisite of learning in real-time. However this does not mean that the performance in longer time scopes is not important. For example, some of the initial SARN versions suffered performance drops in later periods of adaptation. The results presented here also give justification for the winner strength parameter, which appeared almost from the void in the SARN description in chapter 4.

To study the properties of algorithm in a longer time scope, the **yzd** experiment (see sections 3.3.3 and 4.5.3) was performed again for a longer period of six minutes.

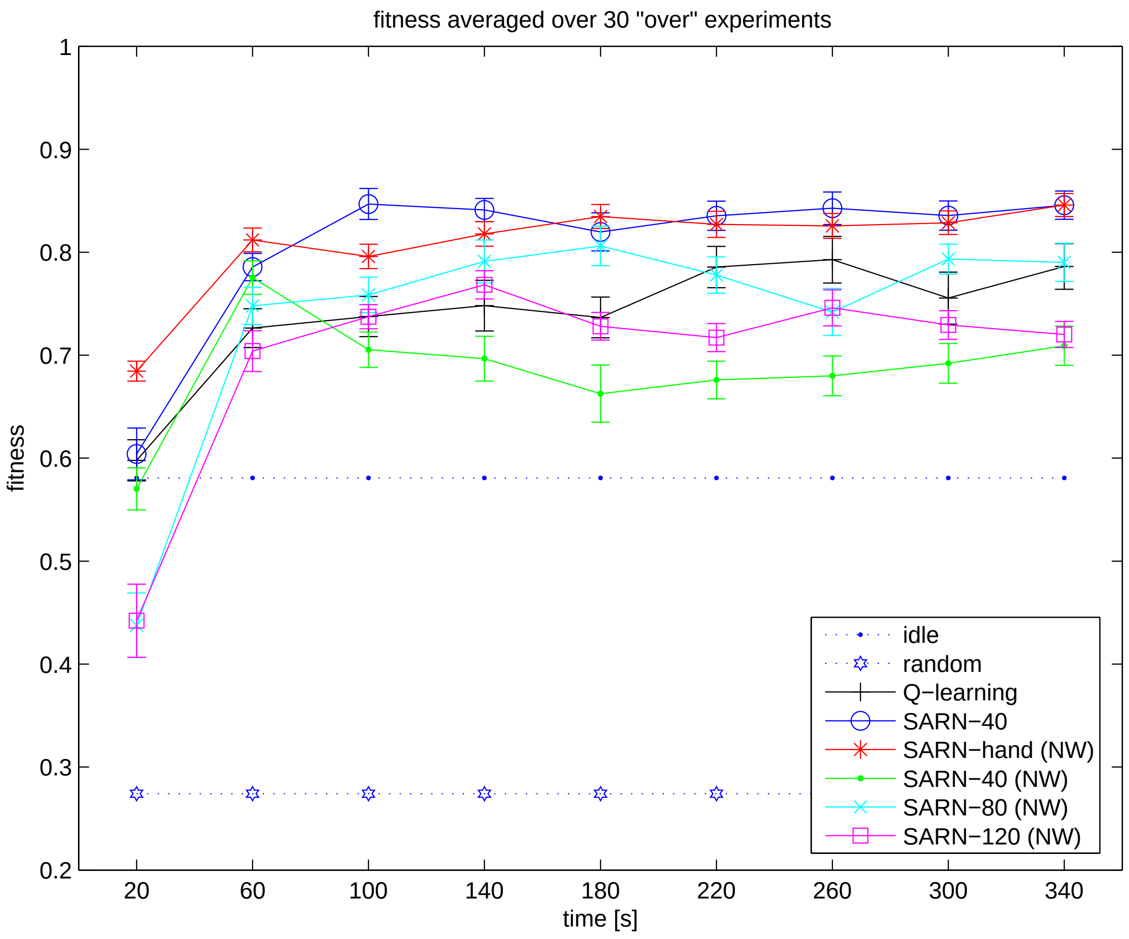

Figure 5.2. The original **yzd** experiment run for six minutes. The NW denotes SARN without winnerStrength parameter included in adaptation. After the initial period of one minute, some of the NW variants' performance degrades. This is particularly apparent for SARN-40 NW.

The result is shown on figure 5.1. The SARN without the `winnerStrength` parameter (which was introduced in section 4.2.5) is abbreviated as NW. The standard SARN variants hold the same fitness for the whole testing period (only SARN-40 shown here) and Q-learning even improves during the course. However some of the NW SARN variants' performance degrades after the first 80 seconds. The relation to the number of stimuli is not straightforward. The fact that SARN-hand (NW) does seem to suffer performance drop can be attributed by easier structure in 4-stimuli case, where interference from the other action weights is not that apparent.

To see better what happens in the dramatic period around one minute, where some NW variants seem to enter "decadence phase", a closeup for 40 and 120-stimuli SARN is shown at figure 5.3. It reveals that the initial learning dynamics are quite similar for both pairs of controllers. Around time 70s, they suddenly split and the NW variants begin to degrade.

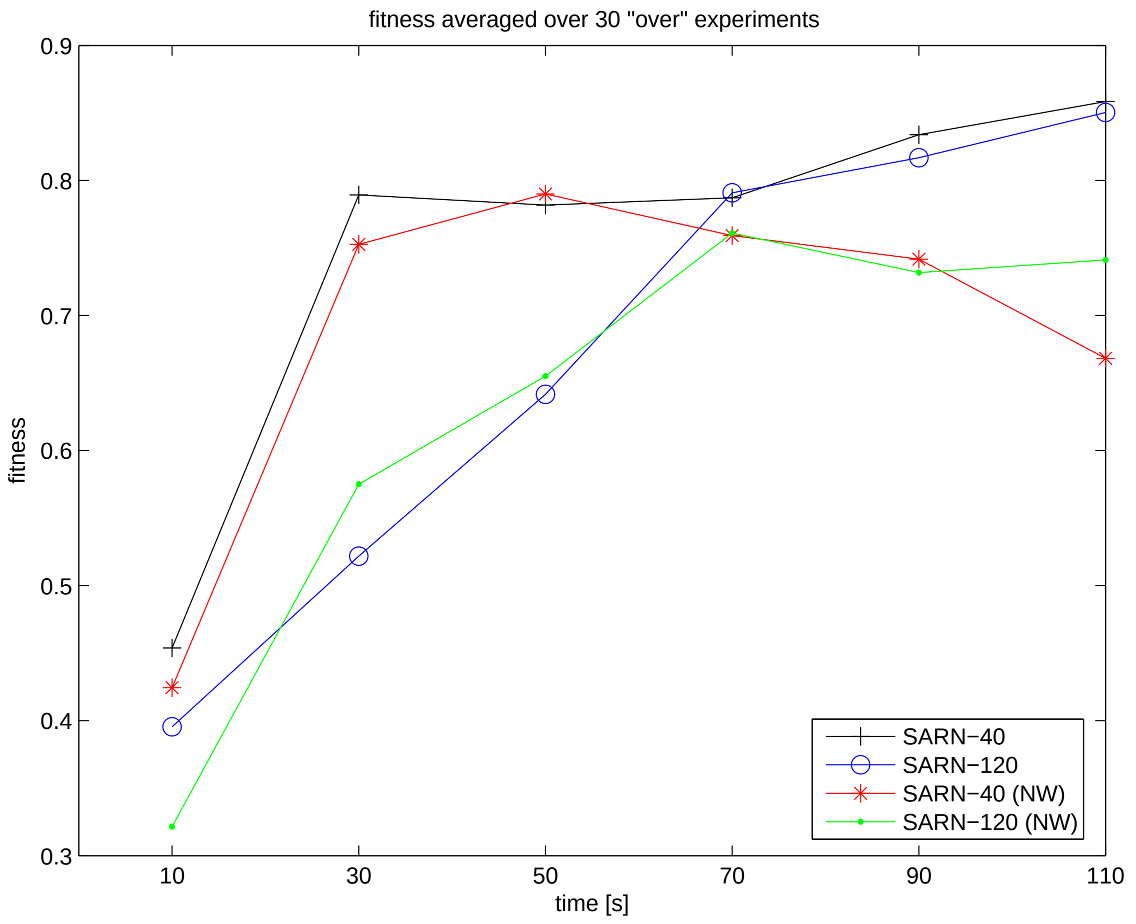

Figure 5.3. Detailed look on the initial period for 40 and 120 stimuli SARN in standard and NW variant. Progress is similar for each pair with the same number of stimuli until 70 seconds. Then, the NW variants start to degrade.

Closer analysis of this phenomena revealed that at the breaking point, the strongest action weights of NW SARN reach the limit if 1. The hard limit evidently causes problems with action selection because conflicting stimuli get reinforced for other actions and eventually overtake as the former dominant action weights have nowhere to grow. The `winnerStrength` parameter in standard SARN prevents growing of the weights when the sum gets larger.

However the winner strength parameter as currently used is not without flaws either. Theoretically, when an action in given stimuli circumstances reaches the absolute threshold ω_a, it is not punished by negative feedback. In practice this does not happen so drastically, because other stimuli links will eventually get reinforced. This makes the currently winning action weaker and the winner strength is no longer 1, thus the action's weights start receiving feedback again. It seems that it would be more reasonable to invert the winner strength for negative feedback. Rationale: if the action has a firm belief that performing it in given stimuli conditions is beneficial, the more we should punish it when the feedback suddenly shows the opposite result. However this has the effect that even well developed associations may be lost in a very quick period. Due to history propagation and other randomisation factors, even the correct stimuli-action association will receive incorrect punishment feedback occasionally.

The adaptation rule clearly needs to be studied further. Some possible directions are discussed in the concluding chapter.

5.3 Environment changes

In all the experiments performed so far, the environment did not change during the whole run. The controllers were put into each experiment's conditions without previous knowledge, but they could rely on the regularities of the environment in order to optimise their performance during the run. What we would want from an adaptive controller is the ability to cope with changes in the environment.

A simple yet quite radical change in the environment is to change the way the projectiles affect the avatar. In a standard (virtual) world, the rockets harm those who are close enough to the explosion. In the "crazy" world, the projectiles heal the victim by the amount they would otherwise harm him. This drastically changes the optimal strategy and the achievable fitness. The experiment is run for three minutes and after the first minute, the world goes crazy and the effect of projectiles is changed. Note that the non-idle action penalty is still unchanged. Apart from the inverse damage effect, the experiment is the same as yzd experiment, that is, aiming with random y-distortion to the floor below learner.

Figure 5.4 depicts the averaged runs of four SARN controllers compared to Q-learning and baseline solutions.

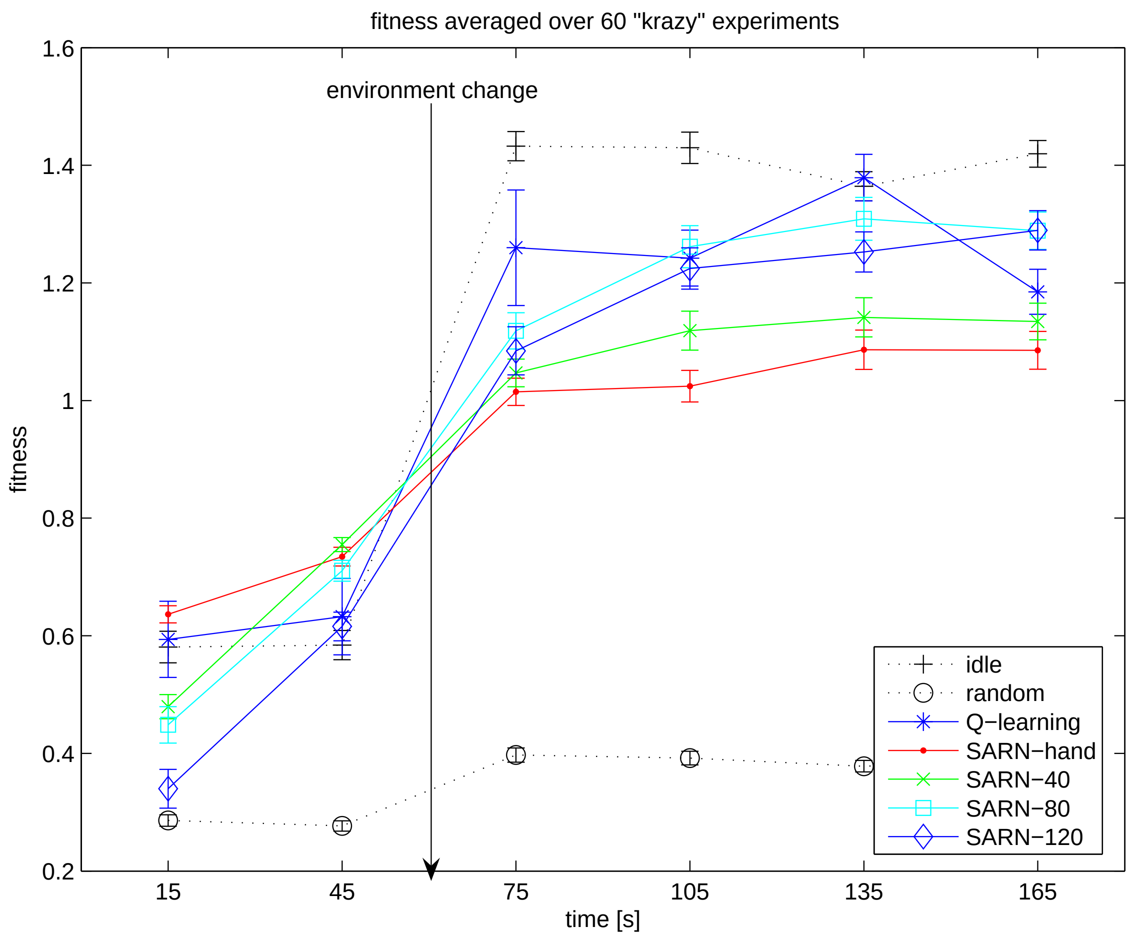

Figure 5.4. Performance of four SARN controllers with Q-learning and baseline solutions in experiment where the environment change happens after the first minute (denoted by arrow).

As can be seen, the fitness grows rapidly after the environment change for all controllers. This is most evident for the idle controller - it still receives random shots but suddenly they become beneficial, yielding the best fitness of all the controllers, at least until SARN-80 reaches its peak. All the adaptive controllers are able to improve their performance in the new conditions, but are not generally able to come up with a strategy which would outperform the idle controller. A better strategy would have to effectively catch the rockets. Catching a rocket which is fired at a random target seems harder than avoiding it as it requires better timing and more precise positioning.

Generally the SARN controllers with more stimuli outperform the ones with less stimuli. Q-learning performs very well, reaching the performance of idle controller around time 135s, but its performance than drops and stabilises around 1.2.

The SARN-hand performs worst of all SARN controllers. This might be caused by the fact that more stimuli provide better chance to find a new stimuli relevant to the actions in the new environment. However the more likely case is that controllers with larger number of stimuli haven't yet settled their weights when the environment change occurs. We already know that the initial learning period takes longer when the number of stimuli is larger. The next subsection will show that the latter is the case.

5.3.1 Improving re-adaptability

In attempt to improve SARN re-adaptability, modifications of the SARN algorithm were explored. It seems that there is not enough “willingness” to improve fitness of a SARN controller, that already works well. After all, the only penalty that a controller receives in the changed environment is the one for moving.

The first attempt was to introduce ε-greedy action selection to encourage exploration, as it is used in the Q-learning controller. However, this did not improve performance - it was only hurt by performing more random (and thus mostly non-idle) actions. The cause is probably that one action alone does not have much chance to cause significant positive feedback and thus be strengthened in the adaptation. Recalling that the only way to get significant positive feedback in this experiment would be to catch the rocket. An interesting extension of ε-greedy action selection would be to perform a whole sequence of random actions instead - the length could naturally be the length of eligibility trace history, h. While this could enable the learner to discover beneficial associations that are not achievable by one-step exploration, it would be even more costly. This idea was not further explored in this project.

The first successful modification was to disable the winner strength parameter. While it prevents unconstrained growth of the weights, it also seems to slow down unlearning of bad habits. Or at least of the habits that became less effective than others due to changed conditions. Results are shown in figure 5.5 later. Before that however, we will describe another SARN modification.

Aging

This technique targeted to improve re-adaptation employs spontaneous forgetting. The idea is that associations that are not currently used (and thus reinforced) shall be slowly forgotten. Therefore, only the links that are being reinforced once in a while will remain learned, the other will diminish. This should allow easier acquisition of new skills. This idea is implemented by subtracting a constant from all action weights every time tick:

```
SARN() {
...
 while(running) {
  w -= wAgingDelta     //substract a constant from all action weights
  w = max(0,w)         //do not let whe weights fall below 0

  // the rest is unchanged ...
  receiveFeedback(getFeedback())
  ...
 }
}
```

5.3.2 Results

The results for the two SARN variants modifications, no winner strength term (NW) and weight aging with wAgingDelta = 0.001 (aging) are shown in figure 5.5.

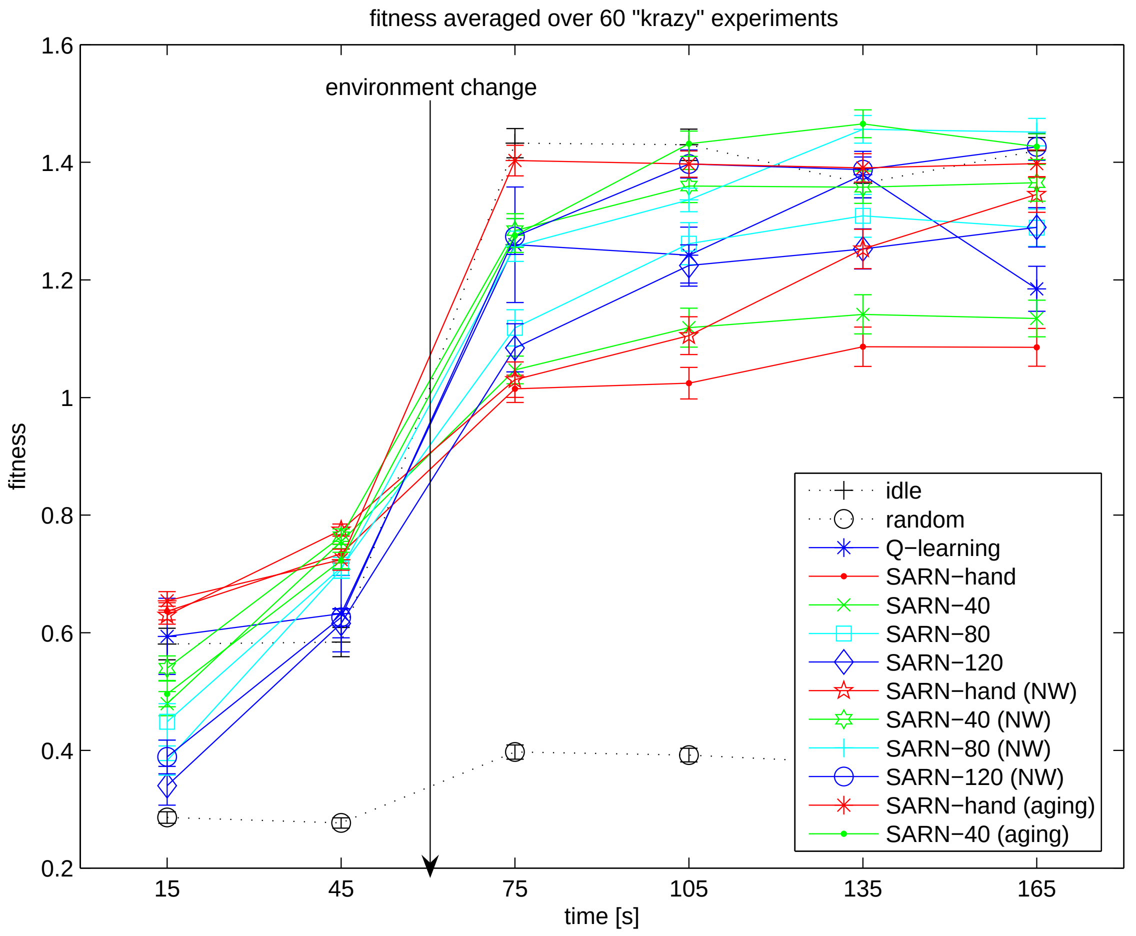

Figure 5.5. Performance of modified SARN controllers in with the original SARN-40 in changing environment. NW - no winner strength parameter, aging - added weight aging term. Aging and NW with certain stimuli sizes are able to perform better than idle controller.
The environment change happens after the first minute (denoted by arrow).

Both NW and aging modifications are significantly better than regular SARN and match the idle performance. There are some signs of outperforming the idle controller's fitness (SARN-40,80 NW and SARN-40 aging), but reliable projectile catching is not achieved. Mean fitness of the best results of each method are compared in terms of mean overall and (final) fitness in the following table:

mean fitness	idle	Q-learning	SARN-80	SARN-80 (NW)	SARN-hand (aging)
overall	1.135	1.052	0.995	1.053	**1.148**
final	1.419	1.185	1.289	**1.451**	1.398

While these SARN variants work better for this specific problem, they have problems with long-term stability and should rather be seen as indications of current SARN adaptation algorithm weaknesses. The fact that the standard SARN with winner strength parameter performs worse shows that the details of SARN adaptation need to be further tested and tuned. While the winner strength parameter is quite advantageous for learning stabilisation, it is not yet well tuned for a wider range of problems. In its current form it successfully prevents over-learning of good associations, but seems also to slow down unlearning of these associations when it is necessary.

The aging parameter indicates a possible extension of SARN, but in this raw form has some problems. Most notably, the appropriate value `wAgingDelta` parameter is problem specific. For this specific experiment, a value of 0.001 worked well, other values did not. Another problem is that by flat forgetting, the rarely used but important associations will be lost and have to be learned again.

Another important aspect uncovered by this experiment is the importance of boundary between positive and negative feedback. Current SARN implementation is not able to judge whether the current strategy is good enough. SARN is eventually able to adapt to the new environment properties and the aging variants do it faster. However neither is able to develop a strategy that would reliably catch the projectiles. A possible cause is that the "catch the rocket" task, compared to "avoid the rocket", lacks punishment. It is only punished for moving. However not for not catching the missile. In order to be able to decide what is good or what is good enough, the controller would ideally need to know maximal and minimal achievable fitness. Obviously, this is the kind of information it will rarely have. However it could possibly keep track of the achieved rewards, moving the boundary of positive/negative during learning. There is a risk of introduced instability, but this topic is worth pursuing in future work.

In comparison, the Q-learning performs relatively well in the changed conditions. It is mainly due to the fact that it does not care whether reward is good or bad - it only estimates utilities of the state-action pairs. The comparison of the utilities makes one action better than another. So when the environment changes, the new utilities will be propagated after some time. Yet still, there is not sufficient exploration drive that would lead to further improvement of the strategy in in the course of the experiment.

5.4 Conclusion

The experiments helped us to better understand SARN features and current limitations related to stimuli size, environment changes and longer time scales.

The relation to Echo State Network still remains somewhat mysterious. In the performed experiments, fitness achieved with ESN stimuli was comparable to the one obtained when handcrafted stimulant was used. However what do the ESN stimuli mean in the input space? How many of them are involved in the action selection? We will try to shed some light on this topic in the next chapter.

Chapter 6
Case-study of SARN-ESN

So far, the SARN algorithm has been mainly analysed from the outside, looking at the results. We have seen that it works well for most of the performed experiments, but not much was uncovered about the actual development process, especially when the ESN stimuli ware used. In this section, the stimuli and internal development of the weights will be studied. We will attempt gain some insight into the learning process during one specific run. The sequence chosen for analysis is three minutes of **yd** experiment with SARN-40 controller, that is SARN with stimuli from an ESN of size 40.

This chapter requires a good understanding of how inputs, the ESN and SARN are connected and how the SARN weights adaption works. These topics were covered in section 4.

While the properties of Echo State Network were not thoroughly studied yet, it is not a goal of this work to study the isolated ESN on its own. The primary interest is how the ESN works as a stimulant for SARN. Since SARN weights change in time, the ESN and SARN can be seen as two coupled dynamic systems. Despite there being direct link from actions to the ESN in the architecture itself, the chosen actions affect the avatar's behaviour. Thus the environment is affected and the changes are reflected in different inputs that are fed back to the ESN.

We start by describing the observed behaviour in the first section. Then we will have a quick look at the perceived stimuli in section 6.2. Because the interpretation of most of the stimuli is unclear, we will turn to the developed action weights (section 6.3) to identify the most important stimuli during the development. Having identified the important stimuli and analysed development of their links for each action, we get back to the stimuli interpretation in section 6.4. Using regression and analysis of inputs weights, we will be able to link some of the stimuli to the input states. In section 6.5, we put all the information together and attempt to explain the observed behaviour. The final section 6.6 summarises what have we learned about SARN and its interaction with the Echo State Network.

6.1 Observed behaviour

The learner initially learns stands still. After receiving the first hit, he starts moving erratically. Within the first two rounds, he learns the first avoidance behaviour. This consists of making three steps right and one left whenever the projectile is closing, no matter where it aims. After one more round, the number of "fuzzy" unnecessary moves is reduced and the maneuver is to take two steps right. After a while, it is hit by a projectile whilst within the evasion maneuver, because the projectile was aimed to the right.

This triggers another change in learner behaviour and after few exploration rounds it settles down in the state where it avoids projectiles by stepping twice to the left when the projectile is aiming right from the avatar. When the attacker aims left from the avatar, the evasion maneuver is to step right once and then back left twice. Interestingly, this strategy works almost perfectly, because the avatar side-steps the projectile just in time.

6.2 Perceived stimuli

This section serves mostly as a motivation for the next sections. The straightforward approach is to look at the stimuli, identify what they mean and then relate them to the adaptation and finally to the observed behaviour. As we will show, matters are not that simple when the ESN is involved.

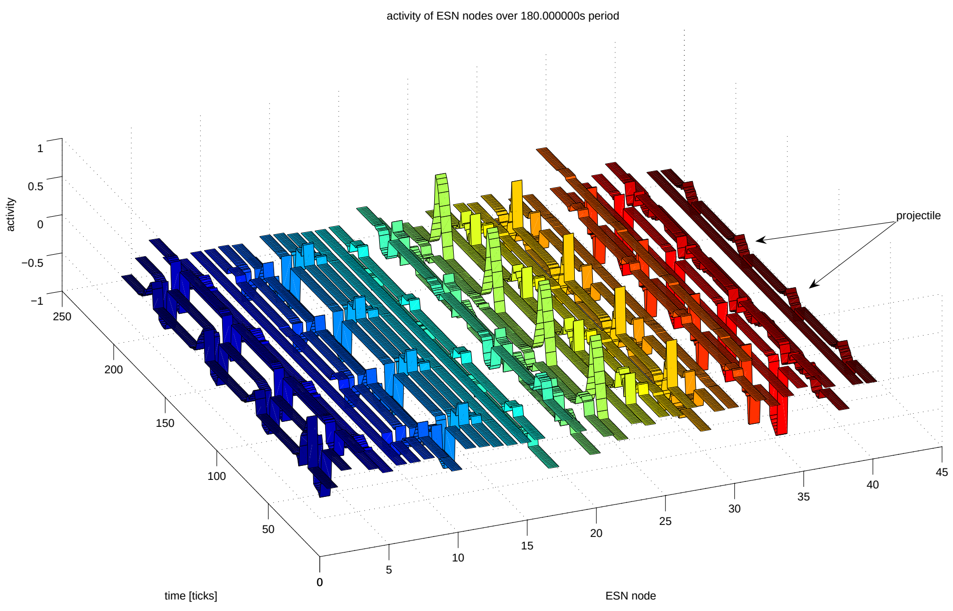

Figure 6.1. ESN activity over 20 seconds. Each ESN node (y) is plotted as separate ribbon over time (y). Height represents level of activity. We can identify rhythmic behaviour governed by the four projectiles fired in given period (two of the projectiles are marked by arrow).

Figure 6.1 shows how the nodes of random network respond to inputs received from the environment. The high-level view tells us that the nodes show similar patter every time a projectile is fired. The detailed picture behind activations is not clear. Most of the nodes seem to do something potentially interesting. Which of them are relevant? Fortunately, SARN identified them by reinforcing or weakening the links of stimuli relevant to particular actions. That is why we now turn to the action weight development. If we can identify the most relevant stimuli for SARN, we can later go back to specific stimuli and try to uncover their relation to the inputs.

6.3 Action weight development

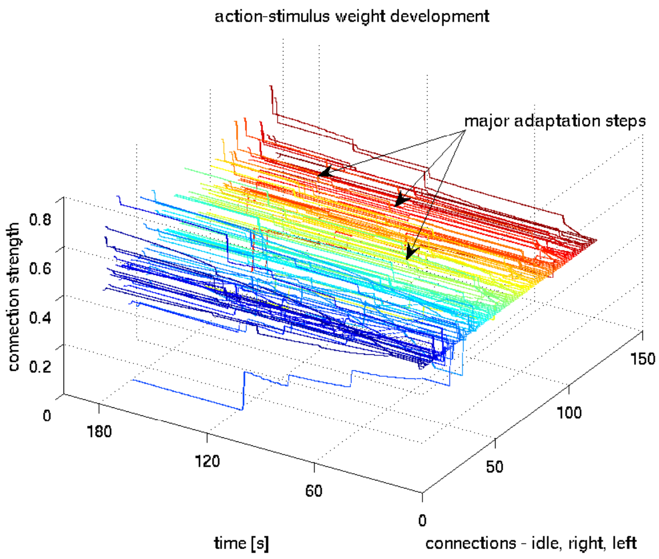

Figure 6.2. Action weight development during three minutes of experiment (y axis). The connections plotted are 40 for each action: idle, right and left respectively (x axis). The z axis corresponds to weight value.

Figure 6.2 captures the whole process of the weights development. It contains a huge amount of data - 40 connections for each of the three actions for every time tick in the three minute run period. Still, some interesting properties can be identified. Over half of the weights are not affected by the learning all, staying at the initial value of 0.5. This means that for these weights, the corresponding stimulus is not relevant to the given action. This can happen for two reasons. The more likely cause is that this stimuli was neither strongly active (close to 1) nor strongly inactive (close to -1) and thus the action weight was not modified at all. The second cause might be that this specific action weight received both negative and positive feedback and those two evened out. This would mean that this particular stimulus is of no use for deciding whether to perform given action or not.

We can see that the big adaptation steps occur quite rarely and on a large number of stimuli together. A closer look reveals that the affected area is usually one third of the action weights. Those correspond to the rocket hits, affecting the action performed just before the hit the most. Some of them are marked by arrow in figure 6.2.

We will now identify important weights and analyse them more closely.

6.3.1 Important action weights

The behaviour of the controller at the end of the run is most affected by the stimuli that are relevant. Relevant in SARN means that they have a strong connection weight. To be more precise, stronger than other weights. These are thus clearly to be counted as important.

On the other hand, we are also interested in weights which are minimal: if the weight was inhibited by a large amount, it means that associating given stimulus with that action lead to negative feedback. The rest of weights will be close to initial strength, that is, 0.5. If they are not zero, why are they considered irrelevant? The fact is that when action weight w_{sa} between stimulus s_n and action a has not been changed during learning, the same is usually true for the other two weights for that stimulus, (leading to other actions). If this is the case, that is $\forall a_1, a_2$: $w_{s_n a_1} = w_{s_n a_2}$ for some stimulus s_n, then this stimulus takes no part in action selection at all. This can be easily proven from the action selection criteria:

$$\text{maxarg}_a wb = \text{maxarg}_a(\Sigma_{s \neq s_n} w_{\text{as}} b_s + w_{\text{as}_n} b_{s_n})^{w_{s_n a_i} = w_{s_n a_j}} \underset{=}{\ } \text{maxarg}_a(\Sigma_{s \neq s_n} w_{\text{as}} b_s)$$

Based on these observations, we define the **weight importance measure** for stimulus s and action a as follows:

$$d_{\text{sa}} = |w_{\text{sa}}^t - w_{\text{sa}}^0|,$$

where w_{sa}^i denotes action weight between stimuli s and action a at time i and t is the duration of the run.

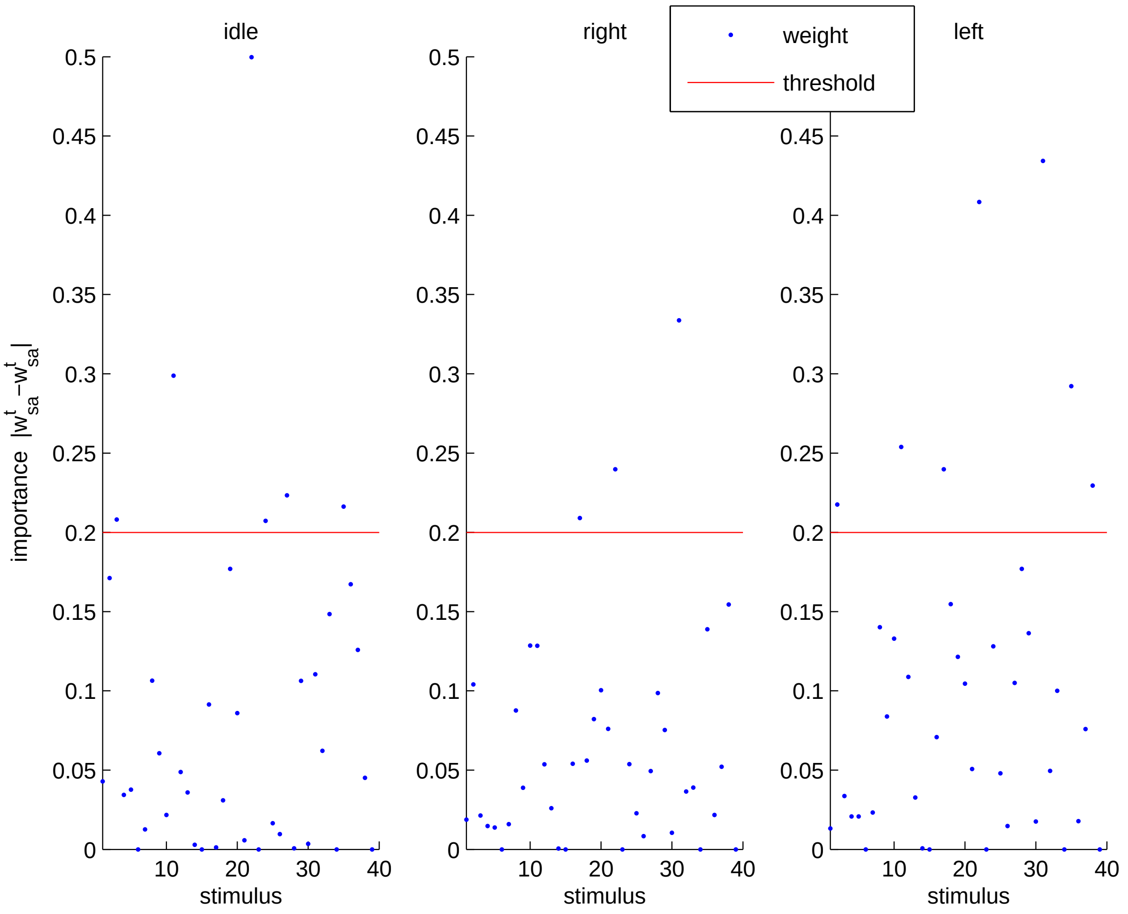

Figure 6.3. Selection of most important weights by weight importance measure d_{sa}. Importance measure weights for three actions is plotted, together with the boundary $d_{\text{sa}} > 0.2$. Above the threshold, there are six identified stimuli for idle action, three for right and seven for left.

Figure 6.3 shows how measure d_{sa} applied to all connection weights highlights the most important stimuli. Each action is plotted separately, but all actions relate to the same set of 40 stimuli. For further analysis, only the connections with d_{sa} greater than 0.2 are considered important. This leaves us with six connections for "idle" action, three for "right" action and seven for "left" action. In total, this involves only 10 stimuli, because some stimuli are important for more than one action.

6.3.2 Development of identified weights

Having identified the interesting stimuli for each action, we can go back to figure 6.2, which showed the action weight development. However this time, we will only look at the identified important weights. The next three paragraphs contain separate plots for each action. The stimuli, plotted on the x-axis are denoted by their index in the ESN. The y-axis represents time and height on z-axis is the weight value at a given time. Note that there are two important aspects. Firstly, we look at the strongest and weakest weights at the end of the adaptation period. The strongest can be said to drive the action selection. The weakened ones are also important, because the learner was punished for performing given action when the corresponding stimuli were active. Therefore they represent some negative association, an environment state unsuitable for a given action. Secondly, we look at the dynamics of the development, whether it is smooth or stepped and whether it proceeds in one direction for each weight or has some more complicated history.

The "idle" action

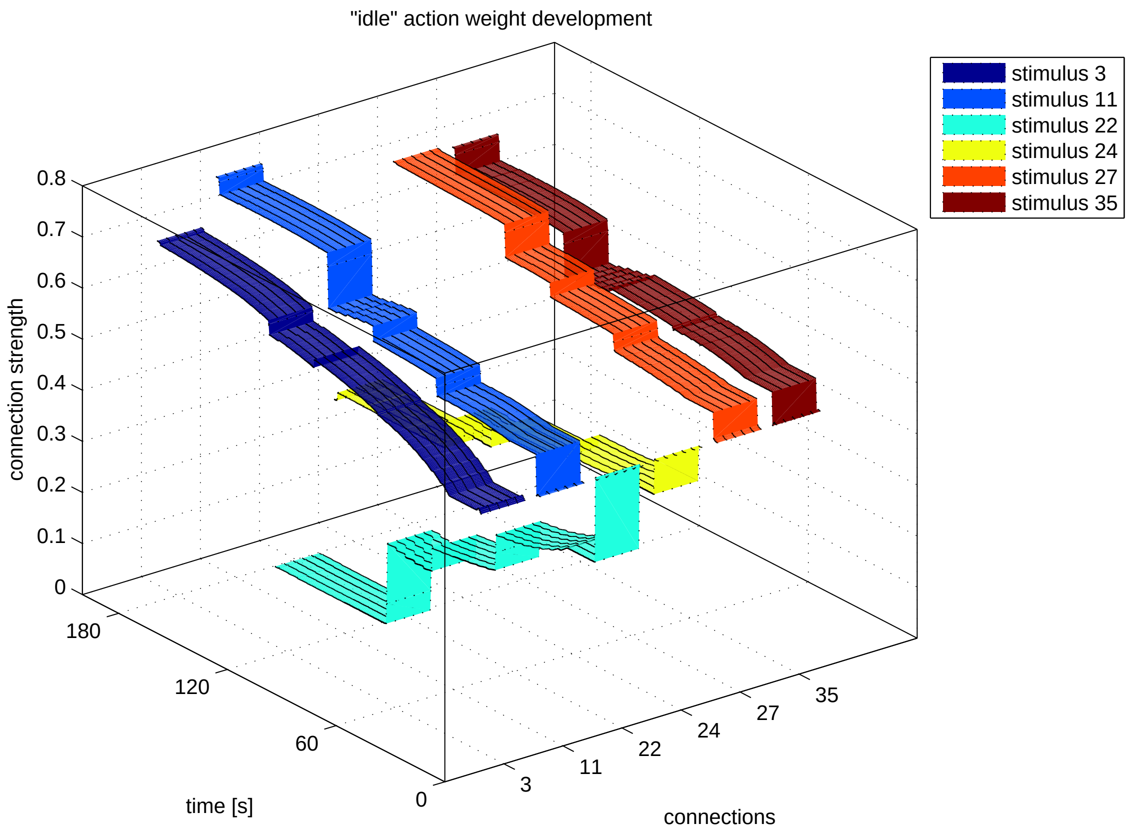

Figure 6.4. Development of important weights identified for the "idle" action. The x axis denotes stimulus, y time and z the weight value. Two of the stimuli are inhibited, while four are reinforced and those have highest impact on action selection.

Weight development for “idle” action is shown in figure 6.4. The activation is mostly driven by four stimuli 3, 11, 27, and 35. Another interesting case is stimulus 22 - is it inhibited, reaching zero activity.

The “right” action

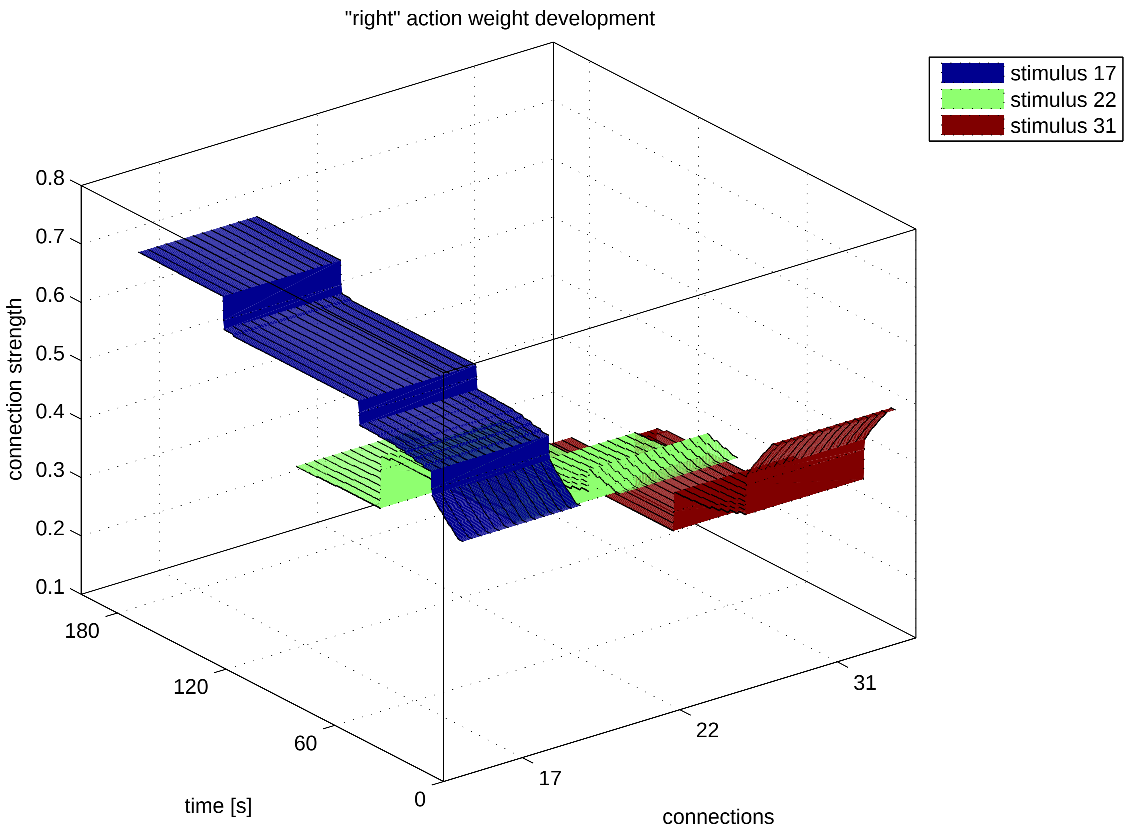

Figure 6.5. Development of important weights identified for the “right” action. The x axis denotes stimulus, y time and z the weight value. Two of the stimuli are inhibited. Only one is reinforced and therefore has marginal impact on selection of this action.

The weights for the “right” action, shown in figure 6.5, exhibit the simplest dynamics of the three. The action seems to have a single dominant stimuli, 17. We can hypothesise that this stimulus corresponds to a closing rocket. Note that stimulus 22 is severely weakened, as in the “idle” case. What can this stimulus mean in the input space, if it is inhibited by actions that are usually mutually exclusive? Let us see whether the same stimulus is also inhibited for the “left” action.

The “left” action

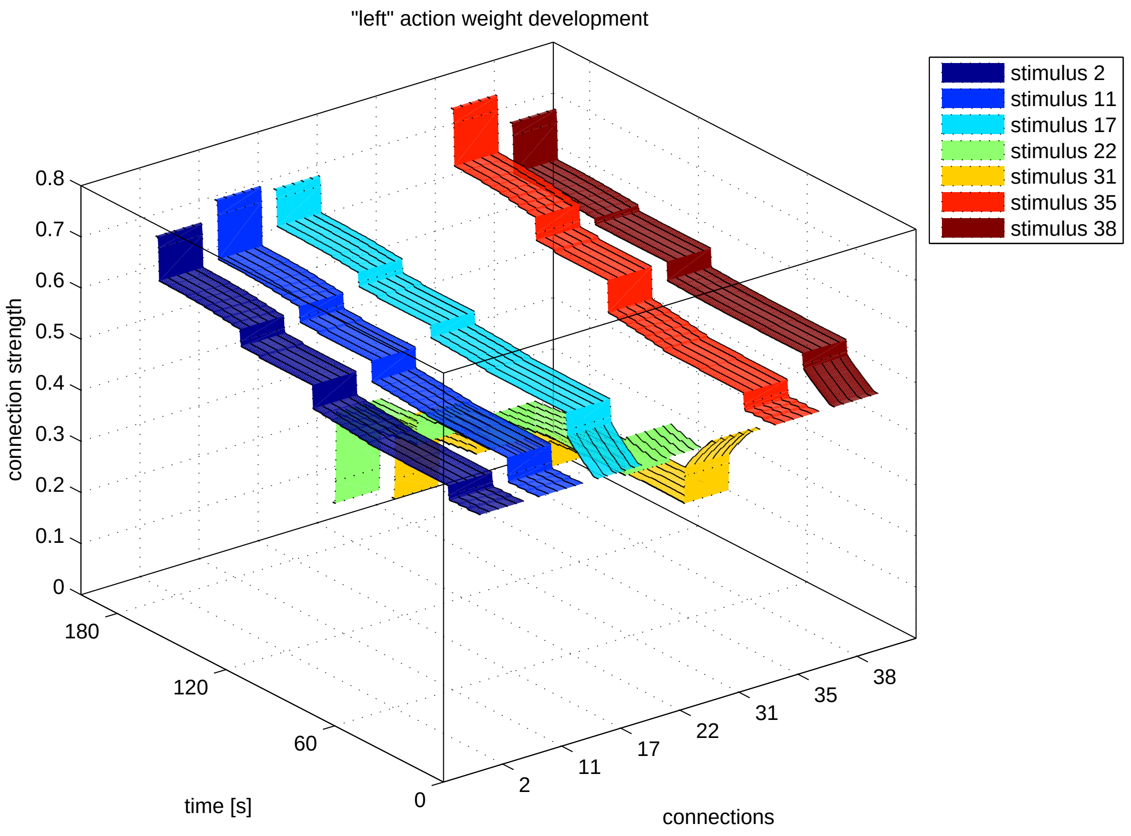

Figure 6.6. Development of important weights identified for the "left" action. The x axis denotes stimulus, y time and z the weight value. Out of the total 7 identified stimuli, two are inhibited (17 and 22, the same as in "right" action). Other five stimuli are reinforced.

The "left" action, depicted in figure 6.6, seems to have the most weights involved in the development of all three actions. As for the other actions, stimulus 22 is inhibited. It would seem that stimulus 22 is not important for action selection after all, because it is inhibited for all actions. However the amount of inhibition differs (the final action weights are 0.004, 0.26 and 0.091 for idle, right and left action respectively), so it is not completely irrelevant to the action selection and it also probably played some role in certain stages of strategy development.

The "left" action has again stronger connection to action 17 and weaker to 31 - as the "right" action has. This means that actions "left" and "right" are driven by the same stimuli, but "left" has some extra stimuli connections (2, 11, 35, 38) which make it dominant over "right" in certain situations. The stronger links could also make it subordinate if the stimuli were negative. However from the observed behaviour it seems that the left action is preferred, therefore the stimuli are likely to be positive in situations where decision between left and right is being made.

6.3.3 Summary

We have learned which stimuli are most important for which actions and which of them drive the action selection. An interesting fact is that actions, even antagonistic ones, can inhibit the same stimulus. Another remark that corresponds to the behaviour observed in this and other runs is that right and left action share some stimuli and are therefore quite close in the stimuli space. That means there is a sharp distinction between idle and non-idle actions, but the boundary between left and right is much lower.

The dynamics of development of all three actions is quite similar. The changes of each weight monotonically adjust it to one direction. The modifications occur usually in quick large jumps, but slower smooth modification is also present. The jumps usually occurred for more stimuli at one time together, quite often for all the important stimuli for a given action.

This gives some insight into how action weights correspond to the behaviour observed in the experiment. Yet it is still not known what the identified stimuli mean in the input space - for which situations are they active or inactive. This is the aim of the next section.

6.4 Interpretation of the stimuli

Now let us look how those identified stimuli react to the inputs to the ESN network.

After a quick look from a plane in the following subsection 4.2, we will get back to the ground and apply regression technique (subsection 6.4.2). Then we will analyse the ESN input weights in subsection 6.4.3. None of these gives complete understanding, but combined together, we can outline some explanation of the agent behaviour and learning process.

6.4.1 Direct interpretation

The task is as follows. Now we have identified which of the 40 stimuli are most important for the adaptation and action selection. Those are 10 stimuli, corresponding to activities of the ESN network nodes. Those activities are generated by four inputs, that are fed into the network:

- p - boolean value specifying there is a projectile
- t - time to impact collision plane
- y, z - coordinates of impact in the collision plane

We want to identify how activity of particular stimulus relates to the state of the input space. More precisely to a sequence of input states, because ESN is recurrent and preserves some information from the recent previous states. We will start by looking at those quantities plotted together over a time period.

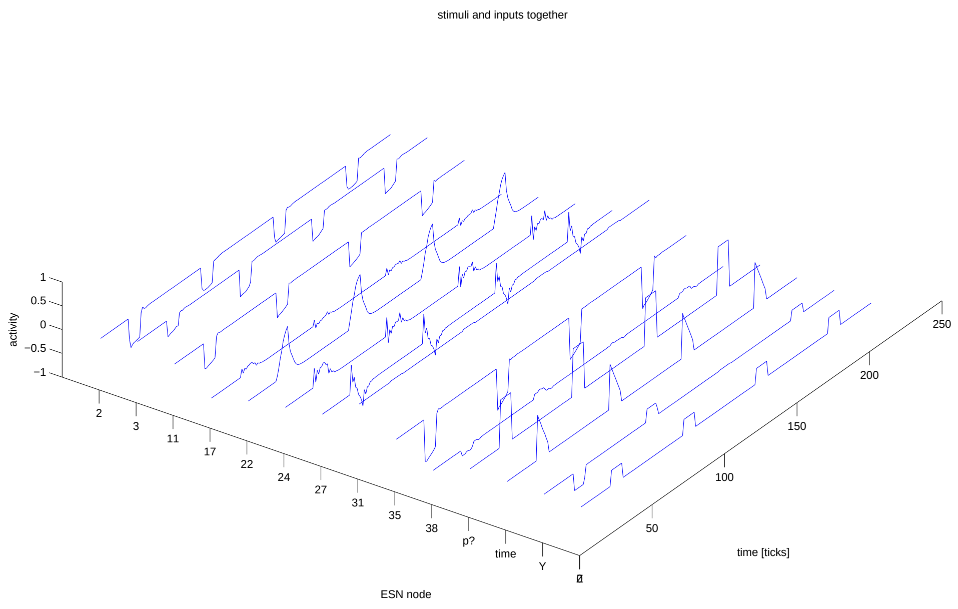

Figure 6.7. This figure shows 20 seconds of stimuli activity, together with four received inputs that drive theirs activity.

On the y-axis, the stimuli and inputs are plotted: stimuli 2-38 and four inputs - projectile present p?, time to impact t, collision plane coordinates y and z.

x-axis denotes time and z-axis the input/stimulus activity.

Figure 6.7 shows activity of the stimuli together with the four inputs. There is a periodical pattern, corresponding to activity of the boolean input "is projectile present?". This is not a surprise, all the other inputs are zero when p is zero. The fourth input, z is de-facto a copy of the first in this component, because the attacker always aims at the same height. So if $p = 1$ (projectile is present), z is some constant.

This means that 75% of the network inputs have very similar patterns throughout the experiment. The network echoes this by the waves of activity. The difficult question is how the subtle differences, mainly in the y input are reflected in the stimuli. This input must indeed be captured by some stimuli, because the SARN agent responds to the direction of the rocket in the environment.

As is usual with the ESN, direct observation of the activity does not really explain how the sequences inputs are mapped to the outputs. Still, the situation is not as incomprehensive as as it is for higher values of the ψ constant controlling recurrence. More insight about the details of stimuli action will be gained by regression in the next subsection.

6.4.2 Linear regression

The simplest way to find dependencies between input vectors and output values is to perform linear regression. For matrix of inputs i and vector of outputs s, the linear regression finds vector b such that mean squared error of residue vector ε is lowest:

$$s = bi + \varepsilon$$

The matrix i in our case represents inputs during the time period and s represents stimulus activities over that period.

The regression was done on only three input components (p, t, y), the z was not included for reasons given in previous subsection. The resulting components of b for each stimulus are shown table 6.1.

stimulus	**p**	**t**	**y**	**error**
2	−0.35	−0.57	0.10	0.18
3	−0.75	−0.22	−0.29	0.01
11	−0.80	−0.20	0.04	0.01
17	0.62	−0.15	−0.53	0.48
22	1.40	−0.54	0.04	0.15
24	1.08	−0.67	−0.42	0.52
27	−1.69	1.63	−0.04	0.55
31	1.18	−0.39	0.23	0.25
35	−0.21	−0.67	0.07	0.24
38	0.15	−0.04	0.78	0.39

Table 6.1. Linear regression coefficients for the given stimuli together with residual error. The coefficients are rounded to two decimal digits.

Where the error is small enough (2, 3, 11, 22, 31, 35), we can look at the absolute value of each coefficient. If one or two input components dominates, we can assume that the particular stimuli is driven by that input(s). This leads to the following summary

Stimuli 2 and 22 are driven by both p and t.

Stimuli $3, 11$ and 35 are bound to t.

Stimulus 31 has dominant coefficient associated with the p variable.

There are also stimuli where the coefficient for y dominates (38) or is combined with other inputs (24, 17), but in these cases the error is too high.

The results of linear regression can be taken as a guidance, but should not be overestimated. Linear regression is only capable of capturing linear dependencies. The recurrent ESN can combine the inputs in non-linear fashion and due to recurrence, the input history is also relevant. We will gain some more clarification by analysing the ESN input weights in the next subsection.

6.4.3 Input weights

Analysing recurrent ESN connections is out of the scope of this project. However we can try to have a look at the input connections. Since the recurrence constant ψ for the used ESN is not very high, the activity of at least some nodes should be significantly governed by its inputs. If a node has strong connection from a particular input, the node's activity will be somehow affected by that input.

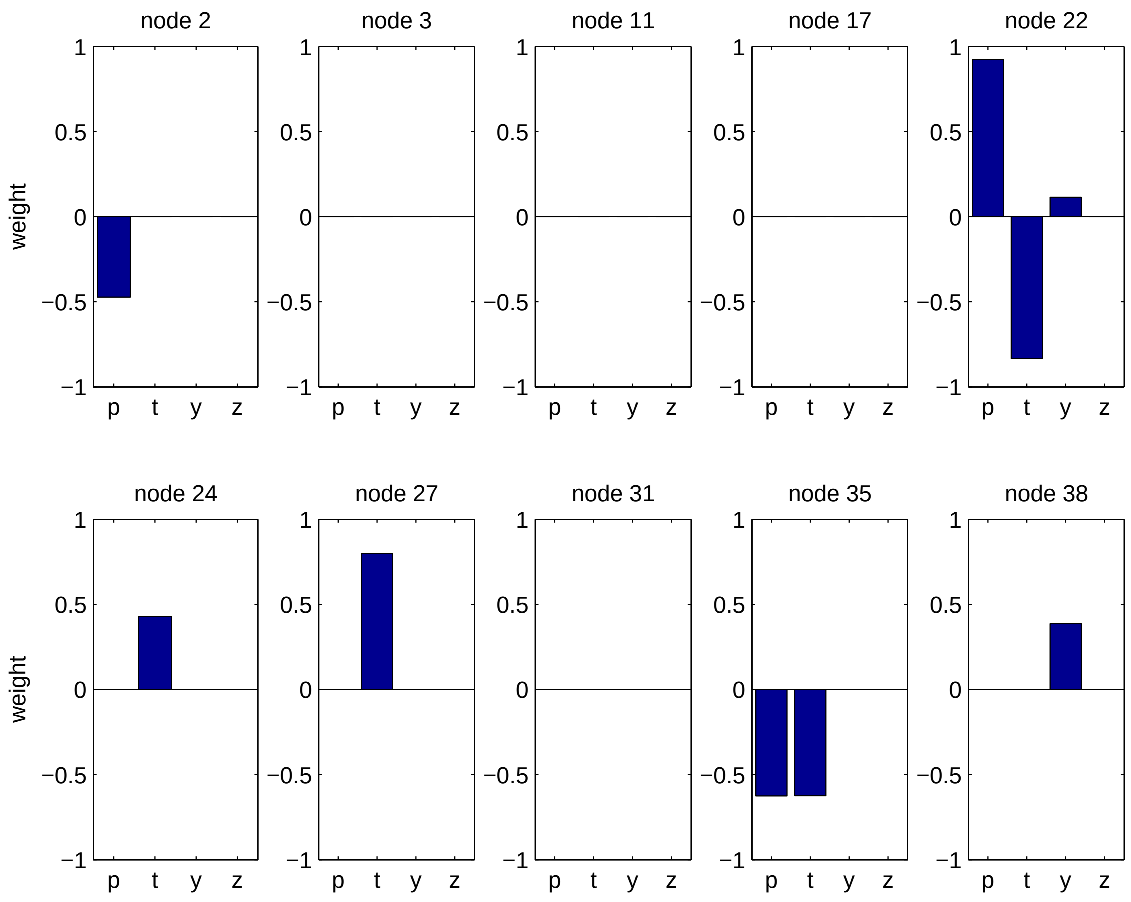

Figure 6.8. Input connection weights of the selected ESN stimuli. Input weights are plotted for each selected node (stimulus) separately. The $x-$ axis denotes input (projectile present, time to impact, y and z impact coordinate), y-axis the connection weight to given input.

Stimuli 2, 22, 24, 27, 35 and 38 have strong weights for some inputs, giving us some insight on their activation. On contrary, no information about the remaining stimuli 3, 11, 17 and 31 can be obtained from this plot.

Figure 6.8 shows input weights for all the ten identified stimuli. Some of the nodes have all input connections close to zero (3, 11, 17 and 31). This gives us no information except that there is something more complicated going on. Fortunately, the regression analysis has shed some light on stimuli 3, 11 and 31, so we at least have some indices even though it was not confirmed directly from the network topology. The weights give us information about two important stimuli: 22 and 38

The inhibited stimulus 22

Connection weights of stimulus 22 give a hint as to why it is one of the most inhibited stimuli for all actions. Since it has positive connection to p and negative to t, the activity of the node is roughly $y = p - y$. This means that it will grow from about zero towards one, reaching the maximum just when the projectile hits. Undoubtedly, this makes the stimulus very relevant to the most important situations in the environment - so either positively or negatively, all the actions must "state their positions" regarding this stimulus. The major weight changes are in ticks close to the projectile impact, which is when this stimulus is most active and thus it will receive more weight modifications than others. On the other hand, when does the avatar need to act, to evade the missile? Not when hit. Before it. This explains why the connections from all actions are weakened and, most importantly, why the idle action has almost zero connection to this stimulus. Sitting on the spot when projectile is about to hit is the worst we can do. Of course, it is safe to sit on the spot **when** we have already performed an evasion maneuver (as the y input would indicate). Recall that SARN has only linear capabilities - this means it cannot combine stimuli in such a way.

Directional asymmetry - stimulus 38

The fact that stimulus 38 is the only one of the selected governed solely by y input indeed confirms the hypothesis about the difference between left and right action weights: Left action has strong connection to 38 whereas "right" does not. Using this stimulus thus probably enables selection of avoidance direction depending on the projectile target.

6.5 Putting the mosaic together

Combining results from weight development and input-stimuli relationship, we can now try to answer some of the questions that have risen rose about particular stimuli. We will interpret the observed behaviour in terms of stimuli and action weights.

We have have only limited understanding of correspondence between inputs and the ESN stimuli activities. For example activations of stimuli 3, 11 and 17 remain a mystery, even though they seem to play some role in action selection. We studied each stimuli separately, but for action selection, these stimuli are combined together. However general hypothesis about the action activation can be proposed.

The idle action has two strong stimuli which are not shared with other actions: 3 and 27. The interpretation of 27 is clear: it is active when the projectile is far away . Results from linear regression (not verified in the ESN topology) suggest strong negative association of stimulus 3 with input p. This stimulus might thus add the second important part to idle action invocation: do nothing when no projectile is in the environment. Recall that action-stimuli weights have only positive values. So for staying idle when $p = 0$, we need stimulus that is positive when $p = 0$ and negative (or low) otherwise. So the two stimuli uniquely relevant to idle action are likely to be responsible for its dominance when projectile is not present or is still far away enough.

As for the non-idle actions, the situation is less clear. This is mostly because we know a few about stimulus 17, which is the only identified reinforced stimulus for "right"action and is also shared with left action. Detailed look at its activity (figure 6.9) indeed ranks it as the most strange of the interesting stimuli.

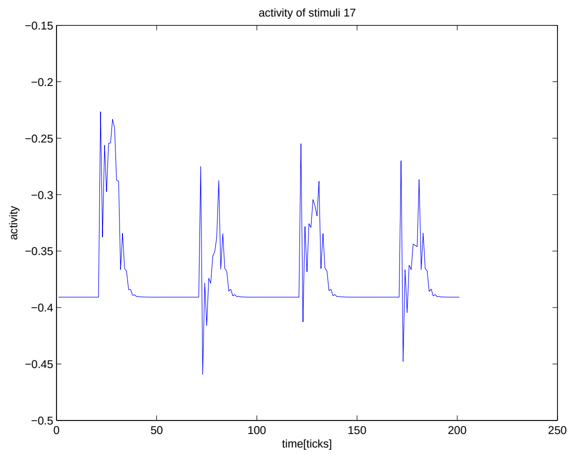

Figure 6.9. Detailed look at activity of important but mysterious stimulus 17. The patterns of activation vary.

The question is how the “right”action realises timing. Either it is via the stimulus 17. Both direct plot and regression analysis show that it appears to be related to projectile closing, but it is not solely driven by t alone. Alternatively the “right” action does not perform timing and relies on higher activation of the idle action. Either way, “right” seems to be activated whenever it has higher activation than “left” due to current projectile target, y.

The timing of “left” action is more clear as it makes use of stimuli 2, 3 and 35. All three have similar activation pattern. Zero when there is no projectile and negative when projectile is far away. The activity grows larger as projectile gets closer. The exactly tuned weights to these stimuli probably play the crucial role in switching between waiting until the projectile gets closer (idle) and stepping (left) when it finally approaches. We can also see why more than one step to the left is performed, even though a single one would be sufficient for the avoidance: all the three stimuli grow until the projectile disappears from the environment. This includes a period when it is already too late for avoidance maneuver. So by adjusting the time of first step so that the evasion starts early enough, the “left” action suffers the by-product of moving when it is no longer necessary. Perhaps a higher penalty for moving could help to combine the timing information with some stimulus that would inhibit “left” activity after the relevant avoidance time.

The role of stimulus 38 seems to be to trigger action “right” when the projectile aims left from the learner. It makes “right” win over “left”, because it gives negative contribution when $y < 0$. However as soon as the step is taken, y is no longer negative thus “left” becomes dominant again. Another step back right does not occur, because of other stimuli exciting “left” (recall growing “left” activation due $2, 3$ and 35 as projectile gets close).

6.5.1 Adaptation dynamics and action selection

The dynamics of learning, development of the action-stimuli weights can be described as monotonous process of adjusting weights of stimuli relevant to a given action. It is interesting how this is reflected in the actual behaviour performed by the agent. The observed strategies went through various stages. Due to the winner-takes-all action selection, even a slight change in action weights can cause one action to outperform the current winner and cause quite radical change in behaviour. The action weight changes would have different impact on behaviour when other techniques such as soft-max action selection are employed.

6.6 Lessons

The explanation of SARN-ESN controller behaviour is not easy, even on a simple task with three actions.

By detailed analysis, we were able to outline how the controller produces the observed behaviour. The most difficult part is to relate the ESN stimuli to environment states and to uncover how a number of stimuli combined together affect the activation of an action. We could go further and test the hypotheses about the concrete stimuli-action mappings in the environment. However exact understanding of one SARN instance in one run at one particular time is not the aim.

The crucial result of this analysis are the observed techniques utilised by SARN to tune action selection performance. We have seen that SARN is able to identify relevant stimuli, inhibit relevant but harming stimuli and make use of both positive and negative stimuli activation. Some of the consequences of the fact that SARN does not have the power to make use of stimuli beyond linear combination were identified. In fact, because the action weights can only be positive, SARN can not make use of negated stimulus. Thus it can operate with positive linear combination of the stimuli vector s, that is $a = b's$ where $b \in [0,1]^l$.

The identified properties will help in future extensions of SARN as well as in improving the basic adaptation algorithm.

Chapter 7
Discussion and future work

The concluding chapter consists of discussion, suggestions for future work and conclusion.

7.1 Discussion

The main achievements of SARN are the significantly reduced teacher involvement before and during the learning process and its applicability to real-time domains with continuous inputs, compared to standard reinforcement learning techniques. We will discuss these two in separate subsections.

7.1.1 Teacher involvement

For a problem where inputs and outputs (actions) are specified, The role of the teacher in SARN consists of choosing the stimulant and providing feedback.

The feedback needs to tell the agent whether his recent actions were good or bad. This is generally a higher requirement than just supplying reward, because the teacher has to specify the threshold for what is good (enough). On the other hand, for some tasks, especially on the lower level of the control hierarchy, this might be more convenient than specifying rewards for every state. As shown in the **yd** and **yzd** experiments (section 4.5), the feedback need not necessarily be smooth to guide the learning progress.

The role of the stimulant is a more complicated one. When it is designed by hand, the task is similar to designing state discretizer. Designing a stimulant will probably have similar difficulties as discretization, because it is up to the teacher to identify ‘‘interesting” parts of the input state and create stimuli that capture them. For stimulant design it is not crucial to minimise the number of stimuli as it is to minimise the state space in discretization. This is because SARN is able to automatically identify the relevant ones. In the example application, design of the discretizer was harder than the handcrafted stimulant, because extra artificial time states had to be created to enable successful learning. Nothing comparable is present in the handcrafted stimuli and SARN is still able to perform better than Q-learning in the experiments.

The most important feature of SARN is that it is able to utilise automatic stimulant from Echo State Network. We cannot say that every ESN will work out-of-the box for every problem, as some input normalisation will probably have to take place. Additionally, because the weight initialisation techniques are not yet well developed, the properties of different echo state networks created with the same parameters will vary. This is especially apparent for smaller random core sizes below 50. There was not enough time to perform comparative tests, but the expectation is that the performance of SARN will vary when using different ESN cores. The problem can be overcome by more biased ESN choice for the specific problem or by using larger number of nodes (80 or more, but this depends on the number of inputs and amount of recurrence needed). Despite this, SARN-ESN is an algorithm with uniquely low teacher involvement in the whole process of applying the technique to a particular problem.

The experiments performed provide enough evidence to make SARN an interesting technique to apply in various domains. However they are far to simple to claim that SARN will scale well to more difficult environment settings. Most notably, the information supplied in the missile avoidance task makes it very easy to develop a good avoidance strategy, without the need to further process the inputs. Also, the number of actions is low. Some initial experiments allowing the learner to jump and duck were successful, but these are subject to ongoing work.

7.1.2 Applicability

What can be said about the range of problems SARN is expected to be good at?

For SARN itself, when stimulant is already given, the adaptation algorithm is able to identify stimuli that lead to positive feedback and combine them in a positive linear fashion to select actions. This is done quite rapidly and thus is suitable for real-time deployment. The solutions found by SARN are by no means guaranteed to be optimal and unless negative feedback is received, the algorithm will not search for better solutions. Compared to common reinforcement learning techniques, no convergence theorems exist for SARN.

The ESN stimuli substitute for the low computing power of the SARN action selector. This is similar to what the original ESN contributed to recurrent neural network learning: the random network provides enough interesting percepts, so that for the output layer only a single layer of neurons suffices, which is much easier to learn.

Therefore, the applicability of SARN-ESN is largely derived from the properties of ESN. The ESN is a recurrent non-linear network with limited memory. SARN-ESN is thus expected to be good for real-time continuous inputs in tasks where fast adaptation is required. Neither SARN nor the ESN possess long-term memory except for the learned stimuli-action associations. Therefore tasks which are supposed to require such ability are not well suited for SARN.

These properties position make SARN best for the lower level tasks where relatively direct links between inputs and output exist and (possibly delayed) feedback is available.

7.2 Future work

In this section, possible future directions and algorithm improvements are briefly outlined.

7.2.1 Combining with higher-level algorithms

The most important goal is to embed SARN within a full-featured controller. In the game environment, it could be quite easily integrated with hand-coded scripts dealing with higher level tasks such as team strategy and path-planning.

Other interesting applications would be inside plan- or logic- based robot controller. SARN should be well suited to low-level behaviours.

7.2.2 Algorithm improvements

The experiments in sections 5 and 6 uncovered many possible algorithm improvements. Here is a brief list of them:

- tuning of the weight update rule (especially the winner strength parameter)

- implementing aging terms
- auto-normalising feedback given past performance
- making SARN capable of negating a stimulus

7.2.3 Stimulant

Comprehensive statistics need to be collected about variance of performance of SARN with different random ESN networks with the same topology.

Alternative ways for automatic stimulant exist. For some problems, a non-recurrent, possibly random transformation of the input space could be sufficient. The stimulant could also make use of automatic clustering techniques like Kohonen Network[10] to identify interesting parts of the input.

The results of automatic state discretization techniques, developed for other reinforcement learning techniques, could be used. There would be one stimulus for each state of the discretized space and its activation would be computed by a distance metric on the input space.

7.2.4 Analysis

Methods to analyse the weight development and stimuli could be extended. The identification of the important weights might include a term for the rate of change of a given weight during the course of algorithm. This would enable better understanding of the learning dynamics.

For smaller inputs states as in our case, sampling techniques could help to identify the relation of input space to the ESN stimuli. The problem is that the number of points that need to be sampled would be large because of the fact that the ESN is affected by input history. Having at least roughly identified the areas of input state that correspond to highest and lowest activation of a given stimuli would greatly help understanding of the SARN behaviour on specific problems.

7.2.5 Human opponent

It is interesting that SARN-ESN is capable of learning full avoidance with the attacker controlled by a human[7.1] player. The automatic attacker does not focus directly on the shortcomings of learner's current strategy. However when we, for example, purposely teach the bot to be aware of the direction of the projectile, he will learn to step away from it. Since the projectile direction-unaware avoidance strategy has an obvious flaw, aiming slightly left (or right) hits the evading learner and gives him a valuable negative example. Repeated counter-examples enable the bot to learn a better strategy. When the learner acquires the new behaviour, he actually becomes quite hard to hit. We no longer fully "see into his brain" and therefore he can surprise us by novel evasion maneuvers.

This kind of "external" human teacher involvement should not be considered negative as in the case of pre-processing. It does not require the teacher to be expert on the algorithm. The teacher guides learner by providing conscious counter-examples. This is much closer to human learning.

7.1. In fact the author of this work is not exactly a human, but for the sake of argument, we can assume so without loss of generality.

7.2.6 Outputs

The fact that we use discrete actions is important for the current SARN adaptation algorithm. We use it to identify the stimuli-action associations to be rewarded. A possible improvement towards more relaxed actions would be to allow parallel actions to be executed simultaneously. We would reinforce/weaken all the stimuli and actions that were active at the time. Of course, this enables some actions to take "free ride" with the important ones and get reinforcement.

An additional extension would be to enable actions to express the degree of activity. The Hebbian reinforcement would still be applicable. This would be a step towards continuous outputs of the controller.

7.3 Conclusion

The main goal of developing real-time learning which is able to operate on-line given reinforcement feedback was reached. The SARN algorithm performs well in range of experiments and requires much less information from the teacher than comparable algorithms. However both SARN and the Echo State Network that is used for pre-processing are young and need formal analysis and experiments to be performed in real-world scenarios to test the robustness and scalability.

The developed methodology seems to have a good potential for real-time environments like robotics and game development. The Unreal Mod created for experiments allows the human player to play against a bot controlled by the SARN algorithm. While it cannot be used as a full-featured opponent in the current form, it demonstrates the potential of artificial intelligence techniques in computer games.

Movies from the environment and the electronic version of this document can be found at http://op.matfyz.cz/sarn.

Bibliography

[1] L. F. Abbott and Sacha B. Nelson. Synaptic plasticity: taming the beast. *Nature Neuroscience*, 3:1178 -- 1183, November 2000.

[2] Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, and Sheila Tejada. Gamebots: A 3d virtual world test-bed for multi-agent research.

[3] Bram Bakker and Jurgen Schmidhuber. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. WWW, accessed 17 Feb 2005, 2004. http://citeseer.ist.psu.edu/679066.html.

[4] Michael Buro. Real-time strategy games: A new ai research challenge. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 1534--1535. Morgan Kaufmann, 2003.

[5] M. Carreras, P. Ridao, and A. El-Fakdi. Semi-online neural-q- learning for real-time robot learning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Las Vegas, USA, October 2003.

[6] V Gullapalli. A stochastic reinforcement learning algorithm for learning realvalued functions. *Neural Networks*, 3:671--692, 1990.

[7] Donald O. Hebb. *The Organization of Behavior*. John Wiley, New York, 1949.

[8] S. Jacobs, A. Ferrein, and G. Lakemeyer. Unreal golog bots. In *IJCAI'05 WS on Reasoning, Representation, and Learning in Computer Games*, 2005.

[9] Gal A. Kaminka, Manuela M. Veloso, Steve Schaffer, Chris Sollitto, Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, and Sheila Tejada. Gamebots: a flexible test bed for multiagent team research. *Commun. ACM*, 45(1):43--45, 2002.

[10] T. Kohonen, M. R. Schroeder, and T. S. Huang, editors. *Self-Organizing Maps*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[11] John Laird. It knows what you're going to do: Adding anticipation to a quakebot. In *Proceedings of the 5th International Conference on Autonomous Agents*. ACM, ACM Press, 2001.

[12] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531--2560, 2002.

[13] I.J. Nagrath, Laxmidhar Behera, K. Madhava Krishna, , and K. Deepak Rajasekar. Real-time navigation of a mobile robot using kohonen's topology conserving neural network. In *Proceedings of ICAR '97*, 1997.

[14] Ondrej Pacovsky. Behaviour selection using echo state network (artificial life course project). http://op.matfyz.cz/papers/behSelectESN.pdf, January 2005.

[15] Paul-Gerhard Plöger, Adriana Arghir, Tobias Günther, and Ramin Hosseiny. Echo state networks for mobile robot modeling and control. In *RoboCup*, pages 157--168, 2003.

[16] Duncan Potts and Bernhard Hengst. Discovering multiple levels of a task hierarchy concurrently. *Robotics and Autonomous Systems*, 49(1-2):43--55, November 2004.

[17] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Evolving neural network agents in the nero video game. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIGâ05)*, 2005.

[18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. A Bradford Book.

[19] Penelope Sweetser and Simon Dennis. Facilitating learning in a real time strategy computer game. In Ryohei Nakatsu and Jun'ichi Hoshino, editors, *IWEC*, volume 240 of *IFIP Conference Proceedings*, pages 49--56. Kluwer, 2002.

[20] Tijn van der Zant, Vlatko Becanovic, Kazuo Ishii, Hans-Ulrich Kobialka, and Paul G. Ploger. Finding good echo state networks to control an underwater robot using evolutionary computations. In *IAV 2004 - The 5th Symposium on Intelligent Autonomous Vehicles,*, 2004. http://www.ais.fraunhofer.de/ vlatko/Publications/Finding$_{g}$ood$_{e}$sn$_{F}$INAL$_{P}$hotoCopyReady.pdf, acessed 20 Jul 2005.

[21] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279--292, 1992.

[22] Stephano Zanetti and Abdennour El Rhalibi. Machine learning techniques for fps in q3. In *ACE '04: Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 239--244, New York, NY, USA, 2004. ACM Press.

Appendix A
Software architecture

The environment code and automatic runs were implemented inside Unreal in UScript Language. All the control and learning algorithms were implemented in a C++ client interfacing with Unreal. Controlling bots from C++ via socket is not a real necessity - the learning algorithms could be implemented in UScript as well. I am not sure how fast would that implementation be (UScript is about 10 times slower than C++), ESN might be a problem.

A.1 Socket interface

The bot controlling interface is realised as a control loop where each time tick, the Unreal bot sends inputs to the sockets and reads the action to perform from the socket afterwards.

A.1.1 Unreal $\rightarrow$ controller

The data are sent in XML. Each time tick, the following data are sent:

- Bot state
    - health
    - vertical speed
    - state (crouching, jumping)
- Projectile info (if a projectile is present in the environment)
    - time to impact collision plane (t)
    - coordinates of the expected impact (y, z)

A.1.2 Controller $\rightarrow$ unreal

The controller issues one command each time tick. The allowed actions are

- no action
- step forward/back
- strafe left/right
- jump
- duck
- fire

A.2 Unreal side (UScript)

The Unreal side runs the experiment and provides server-side implementation of the TCP socket interface (described in A.1) for controlling the learner bot.

The diaagram of main classes is depicted on figure A.1.

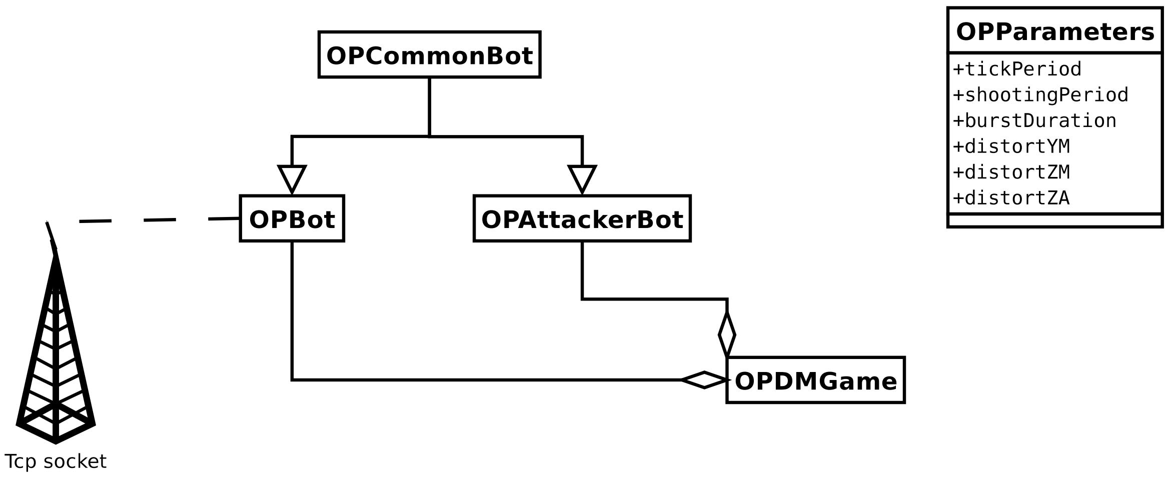

Figure A.1. UML diagram of the core Unreal classes.

The `OPParameters` object stores all information needed to alter experiments. It can be configured from a config file. The parameters include experiment timing and shooting distortion.

The `OPDMGame` class is derived from unreal deathmatch game type. It prepares (spawns) the learner bot and attacker bot (or enables human player to log in as attacker). After spawning the bots, the game waits until attacker bot has a connection to the contolling sockets, sets up experiment and runs it for the specified duration.

The `OPCommonBot` provides common bot features needed for experiments. The main part consists of disabling some of the standard Unreal bot behaviours inerhited from `xBot`.

The `OPAttackerBot` class implements the attacker's behaviour - shooting at the target (learner bot) with distortion and timing given in `OPParameters`.

The majority of Unreal code is contained in the `OPBot` class. It implements the server side of socket interface. That involves gathering the current status and projectile information and sending it to the socket and receiving commands and executing them. The `OPBot` runs in fixed time steps. The information about approaching projectiles is accumulated using `ReceiveProjectileWarning` event.

A.2.1 Comments on using Unreal Engine

Running the server

It is possible to download a command-line dedicated server and run experiments on it. This is useful for entertaining clusters with large experiment batches.

The convenient thing is that Unreal graphic client can be used to log into the experiment and spectate.

Experiments faster than real-time

It is possible to run experimment in speeds faster than real-time, but the maximum possible speedup is roughly $80T$, where T denotes time tick. To use his feature, the tick rate and speedup must be set in UT2004.ini as follows below and the server must be tun with `-lanplay` and `mutator=UnrealGame.MutGameSpeed` arguments.

```
[IpDrv.TcpNetDriver]
LanServerMaxTickRate=120

[UnrealGame.MutGameSpeed]
NewGameSpeed=SPEEDUP
```

The game will adjust game tick duration according to some formula which is not based on CPU load (the load is still slow when Unreal ticks start to take longer time). Without access to the source code, it is probably to run UT in fixed time tick duration and/or maximum possible CPU utilisation, but maybe I missed some parameter.

The fact that fixed frame rate cannot be said is unfortunate, because especially when experiment is run at accelerated speed, other processes in the system may (and will) cause unreal server to miss some sheduler time and the result will be prolonged duration of game tick.

A.2.2 Alternative engines

At the time when this project was started, Unreal 2004 (using Unreal Engine 2.5) was considered the best choice, because it was one of the lastest engines with the bot developer community was already estabilished (unlike Doom 3 and Half-life 2). Furthermore, Stefan Jacobs's ReadyBot PROLOG/GOLOG implementation[8] contained a good start for designing socket control interface.

Unreal also has a relatively good Linux support and relatively good accessibility to modding. Whole UScript source is provided, which means that every high-level feature can be modified. This includes almost everything needed for bot development, but for example path-planning is in native C++ code and thus there is no way to modify is a without licence.

The recent release of Quake 3 under GPL makes it an interesting alternative, because even though the engine is older than Unreal 2.5, the full source is priceless. Id software engines, as always, have a full Linux support.

A.3 Client controller side (C++)

The C++ client implements all the control and learning algorithms.

The socket control interface abstraction is provided by class Bot. It allows the descendants to access information about current state of the avatar and approaching projectiles, as sent by the server. The output of the controller, actions to perform, are also abstracted by Bot.

The descendant of Bot, ActionStateBot class, is an abstract class that translates interface provided by Bot to a problem-independent language of actions and states. That is done by discretising actions and states to integer values by pluggable ActionDiscretisers and StateDiscretisers. Note that descrendants may choose not to use StateDiscretiser. For that purpose, the Info structure from Bot, representing input space, is acessible. The inerhitance hierarchy is depicted in figure A.2.

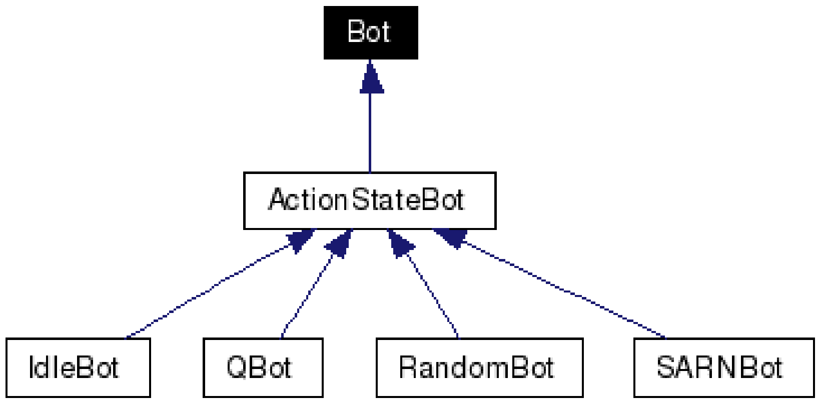

Figure A.2. Bot class hierarchy.

Apart from baseline controllers (IdleBot and RandomBot), the descendants of ActionStateBot are the QBot and SARNBot.

QBot is an implementation of standard Q-learning algorithm.

SARNBot contains the implementation of SARN. For stimuli, it uses the pluggable Stimulant interface. As is shown in figure A.3, two basic implementations exist. The HandStimulant contatins problem-dependent stimuli designed by hand. ESNStimulant is a problem-independent stimulant, where stimuli are Echo State Machine Nodes. The ESN is fed by vectorised inputs each time tick.

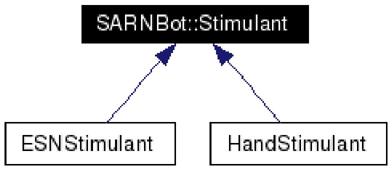

Figure A.3. Stimulant interface implementations.

The SARN algorithm implementation reflects the description in section 4.2. The main parts are contained in the `receiveFeedback` function, which iteratively calls `adapt` for each action in the history with discounted strength, depending on the position (time step) in the history array. The `adapt` function realises the actual weight updating for given action and context.