Individual Learning
and Population Based Learning
– Combining Reinforcement Learning
with a Genetic Algorithm

Björn Bökelund

TRITA-NA-E03156
Individual Learning
and Population Based Learning
– Combining Reinforcement Learning
with a Genetic Algorithm

Björn Bökelund

TRITA-NA-E03156

Master’s Thesis in Computer Science (20 credits)
at the School of Electrical Engineering,
Royal Institute of Technology year 2003
Supervisor at Nada was Frank Hoffmann
Examiner was Jan-Olof Eklundh
Abstract
Individual Learning and Population Based Learning
Combining Reinforcement Learning with a Genetic Algorithm

This thesis presents a novel method to combining population based learning, using a genetic algorithm, with individual learning, using reinforcement learning. The method is based on the state-action function, which is biased by the genetic algorithm to guide the reinforcement learner.

The novel method is based on the assumption that the problem, for which the reinforcement learner has to find the best policy, possesses both stable and unstable features. Hence, to test the method, a test problem is created with the requirements stated above and with no known optimum solution. The test problem used in this thesis with those requirements is a randomly created track for which a vehicle has to find the optimum path.

It is demonstrated that the novel method is superior to reinforcement learning alone, in terms of convergence speed and performance. However, the novel method is computationally intensive during the evolution of the genome. Thus, the method is only suitable for problems that offer non-time critical training data.

Sammanfattning
Individuell inlärning och populationsbaserad inlärning
Kombinera Reinforcement Learning med en genetisk algoritm

Denna rapport presenterar en ny metod för att kombinera populationsbaserad inlärning med individuell inlärning genom att kombinera en genetisk algoritm med "reinforcement"-inlärning. Metoden är baserad på "state-action"-funktionen som är viktad av den genetiska algoritmen för att vägleda "reinforcement"-inläraren.


Acknowledgments

I would like to take this opportunity to express my gratitude to my supervisor Frank Hoffmann for guiding, helping and giving me many great ideas during the work with this Master’s project. I would also like to thank Shi Chen for introducing me to \LaTeX{} and the proofreaders Jonas Bökelund, Jan-Erik Bökelund, Berit Bökelund, Maria Eman and Torbjörn Jörnholm. Finally, I would like to thank Danica Kragic for attending at the Master’s Thesis presentation.
## Contents

1 **Introduction** 1  
   1.1 Background ............................................... 1  
   1.2 Problem and Objectives ................................. 2  

2 **Theory** 5  
   2.1 Reinforcement Learning .................................. 5  
      2.1.1 Introduction ......................................... 5  
      2.1.2 The Reinforcement Learning Model ................. 5  
      2.1.3 Models of Expected Reward .......................... 7  
      2.1.4 Exploration versus Exploitation .................... 8  
      2.1.5 Markov Decision Processes .......................... 10  
      2.1.6 Temporal Difference Learning ...................... 12  
   2.2 Genetic Algorithms ...................................... 14  
      2.2.1 Motivation of Genetic Algorithms ................. 15  
      2.2.2 The Canonical Genetic Algorithm .................. 15  
      2.2.3 The Baldwin Effect .................................. 16  

3 **Implementations** 19  
   3.1 The Simulation Model .................................... 19  
      3.1.1 Choosing the Test Problem ......................... 19  
      3.1.2 Simulation Objectives ............................... 20  
      3.1.3 The Race Track Model ............................... 20  
      3.1.4 The Vehicle Model ................................... 21  
   3.2 Learning Algorithms .................................... 23  
      3.2.1 Reinforcement Learning Using Q-learning .......... 23  
      3.2.2 Reinforcement Learning and Genetic Algorithm .... 27  
      3.2.3 The Genetic Algorithm .............................. 29  
   3.3 Development of the Simulation Tool ..................... 31  

4 **Simulations and Results** 33  
   4.1 Tracks ................................................. 33  
   4.2 Terminology ........................................... 34  
   4.3 Pure RL ............................................... 34
4.3.1 Finding the optimum $\tau$ value .................................. 35
4.3.2 Decreasing $\tau$ .................................................. 35
4.3.3 Performance on Test Tracks ................................. 37
4.4 The Combined RL and GA Algorithm ......................... 38
  4.4.1 Evolving the Genome ......................................... 38
  4.4.2 Performance on Test Tracks ................................. 38
4.5 Comparing Model Two and Decreasing-$\tau$ ..................... 39
  4.5.1 Limitations ................................................... 40
4.6 Analysis ............................................................... 41

5 Conclusions ......................................................... 45
  5.1 Future Work ....................................................... 46

References .............................................................. 48

A The $z$-test .......................................................... 49
Chapter 1

Introduction

1.1 Background

There are in principle two ways to learn; you can be told what to do in different situations or you get credit or blame for doing good respectively bad things. The former is called supervised learning and the latter is called learning with a critic, of which reinforcement learning (RL) is the most prominent representative. There are two major concepts of RL, namely states and actions. The state represents the agent’s present situation. In every state, the agent has to choose between a set of actions. When an action is taken the agent is transited to a new state and the agent receives a reward. The objective of the agent is to maximize the cumulative reward in the long run. In order to achieve this goal, the agent has to explore unknown state-action pairs and find the optimal states and actions, in respect to reward maximization. For complex environments with a high-dimensional state space the exploration needed to find the optimal states-actions will be considerable time-consuming. [1]

The idea of evolutionary learning is to mimic the learning that takes place in biological systems. The process of evolutionary learning is the following; to begin with, a population of agents is created. Every agent is assigned a random set of genes that control some feature of the agents, e.g. the behavior. After some time the agents are evaluated according to some fitness function, which is a measurement of how well the agents are behaving from some point of view. The agents with the highest fitness value are selected and are allowed to reproduce. The offspring are assigned genes from the fittest agents according to some rule, e.g. crossover of genes from two parents. Some amount of mutation during crossover may occur, which introduces new genes into the population. Learning by evolution is slow, thus is not well suited for rapidly changing environments. [2]

There are many examples of tasks being carried out in environments that have some similarities, e.g. driving a car in different cities. The driver could
use prior knowledge from previous visited cities when a new and unknown
city is about to be entered. Comparing different environments could reveal
some resemblance, which is called stable features. This means, features that
do not change across different environments. In the example of driving a car
in different cities, a stable feature could be the behavior of always keeping
to the right in a right turn or that the appearance of the road sign remains
the same across several cities. An unstable feature could be the geometric
configuration of roads across the city. Thus, such features will have to be
explored for every new city visited and thereby preventing prior knowledge
to be used.

1.2 Problem and Objectives

The objective of this Master’s project is to investigate the interaction be-
tween individual learning, using RL, and explorative learning, using genetic
algorithms. This will be done through answering the following two questions:

- RL requires substantial amount of time for exploring environments
  with large state spaces but has the ability to explore completely unknown
  environments. Evolutionary learning, on the other hand, learns by selecting
  agents that is best suited for a certain environment. How can these two
  methods be combined, gaining advantages from both methods? That is, take
  advantage of RL’s ability to explore environments without prior knowledge
  and the advantage of evolutionary learning to adapt the behavior of agents
  to fit certain environments.

- Is it possible to use prior knowledge of the state-value function, re-
garding the stable features, through evolutionary learning by encoding this
  information in chromosomes?

To investigate this interaction, an appropriate test problem has to be
chosen with the following requirements: firstly, the test problem has to
contain some dynamic, inherently unpredictable aspects but on the other
hand it must possess some stable features. Secondly, the problem should
not have any known trivial solution or known optimal solution because of
the lack of new knowledge solving such problem would give.

Finally, the simulated world will bear a certain degree of resemblance to
a real world situation, in such a way that it smoothly can be transformed
to a real world experiment using existing robots and sensors. However, this
transformation will not be carried out in this Master’s project.

To begin with, simulations will be done using only RL. This will be
followed by incorporating genetic adaptation into the simulations. This will
give the opportunity to investigate the interaction between the both types
of learning.

RL methods that will be taken in consideration are those that can be
referred to as temporal difference methods and in particular the Q-learning
method. This is motivated by the prominence of RL and the popularity of using Q-learning for several problems in various fields.
Chapter 2

Theory

2.1 Reinforcement Learning

2.1.1 Introduction

As previously mentioned, there are two ways to learn: either you are told what to do in different situations or you get credit or blame for doing good respectively bad things. The former is called supervised learning and the latter is called learning with a critic, of which reinforcement learning (RL) is the most prominent representative. The basic idea of RL is that agents learn behavior through trial-and-error, and receive rewards for behaving in such a way that a goal is fulfilled. An infant or an animal can be imagined to learn through RL, since they learn how to behave to achieve goals, e.g. finding food. Due to the fact that there is no supervisor, or teacher, in RL the agent has to make an action and then wait for a response. This interaction with the environment is a fundamental idea of nearly all theories of learning and especially in RL. [1]

There are two major concepts of RL, namely states and actions. The states represent the environment, as the agent perceives it. A state could for instance represent the physical position of the state. In every state, the agent has to choose between a set of actions. When an action is taken the agent is transited to a new state and the agent receives a reward. This decision task is complex and much research has been done in this field. In fact, RL in computer science dates back several decades. [1]

The following section discusses the difficulties of action selection.

2.1.2 The Reinforcement Learning Model

The idea of RL can be generalized into a model, in which there are two components: an agent that makes decisions and an environment in which the agent acts. The environment is defined to be everything outside the agent, and thereby the agent cannot directly control the environment. A
complete specification of an environment is called a task, which defines a RL problem.

For every time step, the agent perceives information from the environment about the current state, \(s\). The information perceived could be, for example, the positions of a physical agent, to simplify say the \(x\) and \(y\) coordinates. In every state, the agent takes an action \(a\), which transits the agent to a new state. As mentioned before, when taking that action the agent receives a reward.

Formally the model can be written as follows; for every time step \(t\) the agent is in a state \(s_t \in S\) where \(S\) is the set of all possible states, and in that state the agent can take an action \(a_t \in (A_t)\), where \((A_t)\) is the set of all possible actions in the state \(s_t\). As the agent transits to a new state \(s_{t+1}\) at time \(t+1\) it receives a numerical reward \(r_{t+1} \in \mathbb{R}\).

For an external observer, the agent’s objective is to achieve goals, whereas for the agent there is another objective, namely to maximize the cumulative reward in the long run. In order to make the agent achieve a desired goal, the reward function has to be set accordingly.

To be able to maximize the received rewards, the agent has to know what action to take in every state that gives the best long-term reward. The agent uses trial-and-error to find the best action and then maps every state with an action. This mapping is called a policy and is denoted \(\pi_t\). In other words, \(\pi_t(s,a)\) is the probability that the agent will take action \(a\) if it is in state \(s\). This can be expressed as probability of \(a_t = a\) when \(s_t = s\).

The goal of the task is defined by the reward function, which determines whether an action is good or bad. In order to find the maximum cumulative reward, the cumulative reward for each state needs to be estimated. This estimation is called the value function \(V(s)\), which is calculated as the total amount of reward the agent will accumulate staring at a state \(s\), taking actions according to a policy \(\pi\). This is denoted \(V^\pi(s)\). Roughly speaking, a policy is a rule telling the agent what actions to take in different situations. A policy that maximizes the value function is called the optimal policy, denoted \(\pi^*\). \[1\]

An informal way of explaining the RL model is by studying the following dialogue between the agent and the environment. The agent could be, for instance, a robot in a physical world \[3\]:

**Environment:** Your are in state 5. You have 4 possible actions
(move left, move right, move up and move down).

**Agent:** I will take action 1 (move left).

**Environment:** You are rewarded 10 units. You are in state 10.
You have 2 possible actions (move left and move right).

**Agent:** I will take action 2 (move right).

**Environment:** You are rewarded -2 units. You are in state 12.
You have 3 possible actions (move left, move right and move up).
2.1. REINFORCEMENT LEARNING

Agent: I will take action 3 (move up).

2.1.3 Models of Expected Reward

The objective of the agent, as previously mentioned, is to maximize the cumulative rewards in the long run. How the long run should be defined is not obvious. Several models have been proposed. Three of them have been frequently discussed [3] and they will be reviewed in the following sections.

Finite Horizon Model

The finite horizon model is perhaps the most obvious model. The expected cumulative reward is the sum of the rewards starting from time \( t = 0 \) and continuing for \( h \) steps in the future. The expected reward at time \( t \) is called the return, denoted \( R_t \). Thus the expected reward is:

\[
R_t = \sum_{t=0}^{h} r_t
\]  

(2.1)

where \( r_t \) is the reward received at time step \( t \). [1] An optimal policy could use this in two ways: the policy looks \( h \) steps in the future, which is referred to as an \( h \)-step optimal policy or the policy looks \( (h - t) \) steps in the future. This is called a non-stationary policy and the agent is called to be doing receding-control. The finite horizon model is not always applicable because of the requirement of knowing the agent’s lifetime in advance. [3]

Infinite Horizon Model

For continuing tasks, i.e. infinite or long life span agents, the previous model will not be satisfactory. The sum of rewards could be infinite which implies a problem finding the optimal policy, since the sum of rewards could be infinite for any policy. To solve this problem a discount factor is introduced. The meaning of discounting in this context is that rewards that will be received in the future are less valued than the immediate received rewards. The return is defined in this model as:

\[
R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}
\]

(2.2)

where \( 0 \leq \gamma \leq 1 \) is the discount rate. This series is bounded if the value of reward sequence \( \{r_t\} \) is bounded. A value of \( \gamma \) approaching one makes the agent consider future rewards more than an agent with a value of closer to zero. [1]
CHAPTER 2. THEORY

Average Reward Model

In some cases, it might be useful to have an agent that optimizes the average of rewards in the long run. This can be done by using equation 2.1 and divide the series with the total number of steps and letting the number of step go to infinity:

$$R_t = \lim_{h \to \infty} \frac{1}{h} \sum_{t=0}^{h} r_t$$

(2.3)

A policy with this behavior is referred to as a gain optimal policy. One problem with this model is that it is not possible to discriminate a policy that has a large initial gain from a policy which has not because of the dependency of long-run average performance. [3]

2.1.4 Exploration versus Exploitation

A major difference between RL and supervised learning is as follows; supervised learning uses a supervisor who knows what actions to take, whereas in RL the environment has to be explored in order to find out which actions to take. [3] When an agent takes an action with an unknown reward it is said to be exploring, while taking a known action, and thereby using knowledge, it is said to be exploiting [1]. Whether an agent should explore or exploit is a difficult problem which is showed in the example below.

A simple and common problem that gives an intuitive understanding for exploration and exploiting is referred to as the "$k$-armed bandit" problem. In this problem, there are $k$ "one-armed bandits" that pays off with 1 or 0 with some unknown probability. The agent has $h$ pulls which it can choose arbitrary from the $k$ machines. If the agent finds a machine that has a high probability for paying off with 1, should it continue pulling this machine or should it explore some other machine with unknown, but perhaps higher, pay off probability? This exemplifies the exploration-exploitation dilemma. [3]

There are several methods to solve the exploration-exploitation dilemma. Two of them are quite simple and are presented below.

Sample-Average Method

In the "$k$-armed bandit" problem above, the environment has only one state. Henceforth, let this be true to simplify the methods descriptions.

Consider when the agent is playing the "$k$-armed bandit" problem for the first time, it cannot determine which action that will give the best reward, i.e. the optimal action. However, if the agent plays several times, where every pull is a play, it could estimate the reward for every action. The reward of an action is called the action value. The estimated action value is denoted $Q_t(a)$ and the true action value is denoted $Q^*(a)$. The true action
value is defined as the mean of the actual received rewards as the number of rewards goes to infinity. Hence, a good estimate would be the mean of the previously received rewards. If a certain action, $a$, has been chosen $k_a$ times prior to the time $t$, its action value can be estimated:

$$Q_t(a) = \frac{r_1 + r_2 + \cdots + r_{k_a}}{k_a}$$  \hspace{1cm} (2.4)$$

where $r_1, r_2, \ldots, r_{k_a}$ are the rewards received prior to $t$.

The true value is obtained by taking an infinite number of actions:

$$Q^*(a) = \lim_{k \to \infty} Q_t(a)$$  \hspace{1cm} (2.5)$$

This method of using estimate is called the sample-average method. [1]

**Greedy Action-Selection Methods**

The greedy action of a state is simply the action that has been highest rewarded in the present time. Several proposed action selection methods are based on greedy actions.

The most simple method based on greedy actions is the greedy method that always selects the greedy action. The obvious problem of this method is that it will not give rise to any exploration. Hence, this method is likely to be unsuccessful in most cases since unexplored actions with potentially higher rewards are never discovered. [1]

The greedy method can be altered to allow some exploration to occur. This is done in the $\epsilon$-greedy methods, where $\epsilon$ denotes the probability of selecting a random action. The major advantage with this method is that for an infinite number of tries, or plays, every action will be tested and thus the true value for each action will be found according to equation 2.5 and consequently the optimal action can be found. Nevertheless, equation 2.5 does not give any information about the effectiveness of the method. The effectiveness of the method and how to set the value of $\epsilon$ depends on the task; for example whether or not it is deterministic or stationary, and it also depends on the variance of the reward. [1]

**SoftMax Action Selection**

One disadvantage with $\epsilon$-greedy methods is that the selection of the non-greedy actions is done randomly, which implies that good and bad actions are selected with equal probability. A better way to do this is to select the good actions more often than the bad ones. This is done in a method called SoftMax selection. The probability, $p(a)$, of choosing an action, $a$, is calculated using a distribution, for example the Boltzmann’s distribution:
\[ p(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{k=1}^{n} e^{Q_t(a_k)/\tau}} \]  

(2.6)

where \( \tau \) is called the temperature and \( n \) is the number of possible actions in the current state. A smaller \( \tau \) gives more discrimination between probabilities for good and bad actions and the opposite relation applies. [1]

Consider equation 2.6, as the agent explores the environment, the \( Q \)-function receives rewards and thereby changing the value of the \( Q \)-function, see section 2.1.6. This implies that the discrimination between good and bad actions will increase as the exploration goes along. Consequently, the probability of selecting greedy actions will increase with time.

### 2.1.5 Markov Decision Processes

The decision process is when an agent in some state and time takes an action and receives a reward from the environment. A decision process that satisfies the Markov property is called a Markov Decision Process (MDP). [4]

The Markov Property

An agent needs to receive information from the environment in order to decide its current state. This information could come from many sources, e.g. sensory inputs, memories of previous inputs and efferent motor signals. [4]

In RL, the agent makes decisions as a function of the current state, or state signal. For each action taken by the agent, the environment responds in some way. Consider how the environment will respond, at time \( t+1 \), on an action taken at time \( t \). The response can be divided into two cases: in the first case, the response is general causal and depends on whatever happened in the past. This implies that information about states, actions and rewards has to be known, for all events from \( t = 0 \), in order to tell the response at time \( t+1 \). This can be expressed as a probability distribution:

\[ Pr\{s_{t+1} = s', r_{t+1} = r|s_t, a_t, r_t, \ldots, s_1, a_1, r_1, s_0, a_0\} \]  

(2.7)

for all possible \( s' \) and \( r \) and past events \( s_t, a_t, r_t, \ldots s_1, a_1, r_1, a_0, a_0 \), where \( s' \) is the next state.

In the other case, the response at time \( t+1 \) depends only on the state and action taken in the previous state, \( t \). This property for the state, or state signal, is called the Markov property. It can be expressed as a probability distribution:

\[ Pr\{s_{t+1} = s', r_{t+1} = r|s_t, a_t\} \]  

(2.8)

for all \( s', r, s_t \) and \( a_t \), where \( s' \) is referring to the next state.
2.1. REINFORCEMENT LEARNING

An example of the Markov property is a moving cannon ball. All information needed to calculate the ball’s next state is the current position, direction and velocity, thus what happened in the past is irrelevant.

It can be shown that for an environment that has the Markov property, it is possible to predict future rewards based on knowledge on the current state only, with equal good result as with the complete knowledge of the total history leading up to the current state. [1]

Description of MDP

Formally, a MDP consist of:

- A set of states $S$ and actions $A$.
- Transition probabilities:

$$P_{ss'}^{a} = Pr\{s_{t+1} = s'|s_t = s, a_t = a\} \quad (2.9)$$

where $s'$ is referring to the next state, $s$ is any state and $a$ is any action so that $s \in S$ and $a \in A$. The transition probability is the probability of a next state $s'$ at time $t + 1$ given the state and action at time $t$.

- The expected value of the next reward:

$$R_{ss'}^{a} = E\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\} \quad (2.10)$$

where $s'$ is referring to the next state and $s$ is any state and $a$ is any action so that $s \in S$ and $a \in A$. [1] A formal description with transition distribution and reward distribution can be found in [4].

Value Functions and Policies

In the RL model, the objective is to find a policy that gives maximum reward in the long-run, see section 2.1.2. A policy tells which action to take in every state, a stochastic mapping from states to actions that can be expressed $\pi : S \rightarrow A$. [4] As mentioned before, the value of a state $s$ when taking action $a$ under a policy $\pi$ is denoted $V^\pi(s)$ and is the expected sum of future rewards when starting in state $s$ and following the policy $\pi$. This can expressed formally for MDP as:

$$V^\pi(s) = E_\pi\{R_t|s_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\right\} \quad (2.11)$$

where $E$ denotes the expected value and $R$ denotes the return, which in this case is the sum of all future rewards using infinite horizon model, see section 2.1.3.

In order to find an optimal policy, a definition of this has to be given. A policy $\pi$ is optimal if that policy is better or equal to any other policy $\pi'$. This is defined as:
\[ \pi \geq \pi' \text{ if and only if } V^\pi(s) \geq V^{\pi'}(s), \text{ for all } s \in S \quad (2.12) \]

The value of the state is the expected rewards following a policy \( \pi \). However, to find the optimal action, the current policy cannot be strictly followed because of the possibility of existing undiscovered policies that are better than the current policy. The value of a state \( s \) when taking an arbitrary action \( a \) and thereafter following the policy \( \pi \) is given by the state-action function, denoted: [1]

\[ Q^\pi(s, a) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\} \quad (2.13) \]

This function will be used in the Q-learning algorithm described later.

### 2.1.6 Temporal Difference Learning

There are three fundamental different types of methods for solving the RL problem, namely dynamic programming, Monte Carlo methods and Temporal Difference (TD) learning. Dynamic programming requires a complete model of the environment, whereas Monte Carlo methods and TD learning do not. TD learning combines the ideas of dynamic programming and Monte Carlo methods. [1]

**Temporal Credit Assignment Problem**

For many tasks, the reward is not known until several time steps later, for example, when the task is over, or a certain goal is reached. This is called a delayed reward. A problem that arises for those kinds of tasks is that it is difficult to tell which actions that were good and which were bad, i.e. which actions that contributed to the final reward. An example of this is when playing a game of chess; the agent has to wait until the end of the game to know whether it played good or bad. Nevertheless, at that point it is not easy to say which actions actually led to the outcome. This problem of finding out which actions to give credit is referred to as the temporal credit assignment problem. [3]

**Prediction by Temporal Difference**

Consider again a game of chess. Solving the temporal credit assignment problem can in this case be regarded as predicting the outcome of the game. For each move, a prediction can be made. In conventional predicting-learning, the prediction for each move is compared with the outcome of the game and updated accordingly. Whereas in TD-learning the prediction is updated according to temporally successive predictions, that is the prediction of one move is compared with the prediction of the next move and
updated according to this difference. The advantage of TD-learning over
conventional learning is that TD-learning is more incremental and therefore
easier to compute, converges faster and gives better predictions. [5]

The Value Prediction problem and the Control Problem
RL can be divided into two separate problems: the value prediction prob-
lem and the control problem. The problem of finding the optimal policy is
referred to as the control problem and the problem of evaluating the policy
is referred to as the value prediction problem. [1]

In the next section, the prediction problem will be reviewed and followed
by a presentation of the control problem.

The Value Prediction Problem
As mentioned earlier, there are to ways to predict the value function. The
first way is to wait until the final return is known. This is done in Monte
Carlo methods. A simple Monte Carlo method calculates the value function
according to:

\[ V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)] \] (2.14)

where \( R_t \) is the actual return after time \( t \) and \( \alpha \) is a constant step-size pa-
rameter. The drawback for this method is that the \( R_t \) can only be calculated
after the task is finished. [1]

The second way is to use prediction by temporal difference, described
above. Instead of waiting to the final return is known the immediate reward
\( r_{t+1} \) is used together with the difference of the value prediction between the
current \( V(s_t) \) state and the next state \( V(s_{t+1}) \). The simplest TD method is
\[ TD(0): \]

\[ V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \] (2.15)

where \( \alpha \) is a step-size parameter and \( \gamma \) is the discount rate. [1]

The control problem: \( Q \)-learning and SARSA
There are two types of methods in TD for the control problem: on-policy
and off-policy methods. In the former, the value function is updated using
results from the same policy used by the agent. In the latter, the value
functions can be updated using a different policy. \( Q \)-learning is an off-policy
method and SARSA is an on-policy method. [1]

The key idea behind these two methods is to follow a policy, which could
be for example \( \epsilon \)-greedy, and then estimate the \( Q \)-function for every state.
Since the SARSA method is an on-policy method, it uses the same policy
for the control problem and prediction problem it can only update the $Q$-function relatively to the next state visited. The selected policy could for instance be the $\epsilon$-greedy method. This means that a small amount of the taken actions will not be the greedy action, implying that the $Q$-function will be updated with an incorrect value in those cases. The update function is similar to that of the $TD(0)$ method but it updates the $Q$-function instead of the value function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$  \hspace{1cm} (2.16)$$

where $0 \leq \alpha \leq 1$ is the learning rate and $0 \leq \gamma \leq 1$ is the discount factor.

The $Q$-learning method is similar to the SARSA method but since it is an off-policy method, it is possible to use a different policy for the prediction problem. Instead of taking a sub-optimal action, which can be the case for $\epsilon$-greedy, the optimal action is always selected. Thus the update rule uses the greedy action, expressed mathematically with the max function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$  \hspace{1cm} (2.17)$$

where $0 \leq \alpha \leq 1$ is the learning rate and $0 \leq \gamma \leq 1$ is the discount factor. \[1\]

See algorithm 1 for the complete $Q$-learning algorithm.

**Algorithm 1: The $Q$-learning Algorithm**

Initialize $Q(s, a)$ arbitrarily
repeat
  Initialize $s$
  repeat
    Choose $a$ from $s$, that is $a \in A_t$, using policy derived from $Q$
      (e.g. $\epsilon$-greedy)
    Take action $a$, observe $r$ and $s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  until $s$ is terminal
until all episodes are done

2.2 Genetic Algorithms

The history of genetic algorithms (GA) dates back to the beginning of computer science. In the beginning, evolutionary systems were studied as an optimization tool for engineering problems. John Holland introduced GA in 1960 with the purpose of study the phenomena of natural adaptation and incorporate these ideas into computer science. \[6\]
2.2. GENETIC ALGORITHMS

2.2.1 Motivation of Genetic Algorithms

Many problems include optimization of some function for some variables, e.g. maximize profit or minimize cost. Genetic algorithms are typically concerned with nonlinear problems, where the combined effect of several variables has to be considered to find the optimal value. The search space for such problems can be large. Assume that the problem is encoded using a binary string with length $L$. The search space will then be the size of $2^L$, thus a random search algorithm will not be a successful method to solve the problem. However, a random search algorithm that is biased in respect to where it should look for solutions, would require substantially less time to find a solution. A GA is a random search with that property. [2]

There are several other search algorithms other than GA, although they are not well-suited for real world problems. For instance, calculus based methods require that derivatives exist and noisy functions may be difficult for these methods. GA differs fundamentally in several ways from conventionally search algorithms, e.g. GA have probabilistic transition rules rather than deterministic and uses objective functions instead of derivatives. [7]

2.2.2 The Canonical Genetic Algorithm

The genetic algorithm introduced by Holland is referred to as the canonical genetic algorithm. Most of the existing theory applies for this model. [2]

Terminology

Since GA is inspired by biological systems, it is relevant to look at the correspondence between them both. In biology, a chromosome is a set of genes. The chromosome forms the genetic prescription for construction and functions of organisms. The total genetic package is called genotype for biological systems.

On the other hand, in GA the chromosomes are called strings, which are usually binary coded. Furthermore, genes are referred to as features and are usually one or more bits of the strings and finally, the genotype is referred to as structures. When an organism is created from its genotype in a biological system and interacting with the environment, a phenotype is created. The equivalence in GA is called a particular set of parameters. In biological systems, the position of the genes is called locus and the value of a particular gene is called allele, whereas the former is referred to as string position and the latter is referred to as feature value in GA. [7]

Evaluation and Fitness

To begin with, the GA starts with creating an initial population. Every individual of this population is assigned, often randomly, a binary string of
length $L$. When the initial population is created, every individual, that is every string, is evaluated using an evaluation function that measures the performance of each individual with respect to a set of parameters. The fitness function transforms the evaluated value to a fitness value, which is a measurement of the individual compared to the other individuals. The fitness value of an individual is used to set the probability of that individual to reproduce. The fitness can be calculated as:

\[ f_i = \frac{e_i}{\bar{e}} \]  

where $e_i$ is the evaluation of a string $i$, and $\bar{e}$ is the average evaluation of all strings in the population. There are also other methods to calculate the fitness.

One generation, for the GA, is the process of going from the current population to the next population. Selection is made from the current population according to the fitness value, and from the selected individuals an intermediate population is created. The next population is created through recombination and mutation of the intermediate population.

Selection can be made in several ways. The main idea, however, is to select the strings with the highest fitness value more often than those with lower fitness value. A simple method is to assign a space in a roulette wheel proportionally to the fitness of that string.

As indicated above, an intermediate population is created from the selected individuals. Thereafter, recombination can occur. This is done through crossover, which is a process where two strings are selected randomly from the intermediate population and a recombination point, $x$, is selected randomly. Assume strings have length $L$. Bits from the first string are selected from bit one to bit $x$ and bits from string two are selected from bit $x + 1$ to bit $L$. This gives a new string on which mutation may be applied. Biological mutation is represented by flipping bits in GA. This occurs with some small probability $p_m$, typical $< 0.01$. After recombination and mutation has been applied to the string, the string is transferred to the next population and a new generation begins. [2]

### 2.2.3 The Baldwin Effect

The Lamarckian hypothesis argues that knowledge or traits acquired during lifetime of an organism can be transmitted to its offspring through genetic changes. Although this theory has been rejected by the majority of biologists, a theory with a similar effect do exists, namely the Baldwin effect. Baldwin argued that if an organism is able to learn a certain behavior that is crucial for its survival, that organism would have a better chance to survive, i.e. its genes would have a higher probability to survive in the population. If the requirements of what is good to learn stay the same through time, the
2.2. GENETIC ALGORITHMS

genes responsible for that learning behavior will spread in the population and in the long run, those genes will in fact be encoded into the genes. However, this encoding will not be done in a direct sense, as in the Lamarckian case, but rather in an indirect sense. [6]

Hinton and Nowlan worked out a model to test the Baldwin effect. The basic idea of this model, which will not be described in detail in this report, is that the fitness function, if seen as a fitness landscape, will have the shape of a spike. Such a fitness function requires that an agent have to behave in an exact way to be assigned a fitness value above zero. This requirement will force the agents in a population to search for substantial time before any of them will be assigned a fitness value above zero.

However, in the model the agents were equipped with a capability to learn during life. When an agent learns, it means that it moves through the fitness landscape, thus instead of remaining at the same position in the fitness landscape it will have the ability to move somewhat back and forth. This movement will actually increase the probability of getting a fitness value above zero for every agent and therefore increase the speed of learning. Another way to look at it is that the fitness function with a spike shape is smoothened out when agents are equipped with the capability of learning. [8]
Chapter 3

Implementations

In this chapter, the implementations are presented. To begin with, the selection of an appropriate test problem is discussed, followed by introducing the selected simulation model.

3.1 The Simulation Model

3.1.1 Choosing the Test Problem

Choosing a suitable simulation model is a delicate process. According to the requirements, the model needs to resemble a real world situation to some extent, yet be simple enough to simulate.

A simple test problem with both dynamic and unpredictable aspects is the grid-world with randomly placed obstacles. The grid-world is a simple rectangular world with \( N \) equally sized squares in which the agent can move. In each square the agent can take four actions, namely move to any of the adjacent four squares. The objective of the agent could be to move from one specific square, the starting square, to another specific square, the goal square. However, for such problems an optimal shortest path algorithm [9] could be applied, hence the grid-world does not fulfill the requirements.

Nevertheless, the grid-world can be slightly changed into a more complex problem. Suppose that the grid-world can take any form and that the agent is forbidden to go outside some subpart of world. Further, instead of going some arbitrary path between two points, suppose that the agent is restricted to follow some predefined track, which is unknown to the agent. Such problem lacks an optimum solution and is consequently more appropriate as a test problem compared to the basic grid-world. A real world example of such problem is a vehicle racing through a racetrack. Motivation to investigate such a model can be found, for instance, in the automobile industry as well as in the gaming industry.

Summing up the above, it shows that the racetrack model is an appropriate test problem, thus it is chosen as the test problem of this Master’s
As described above, the chosen test problem is a simulated race track with random, initially unknown curvature. The vehicle could be given a numerous set of different objectives, but to simplify the implementation process, the objectives were reduced to the following three:

- Avoid contact with track sides.
- Minimize steering actions.
- Keep close to centerline.

The vehicle is supposed to drive through the track for a fixed number of times. The first laps are for training purposes and the last lap is for the purpose of performance evaluation.

The next two sections describe how the race track is created and how the vehicle is simulated.

3.1.3 The Race Track Model

The racetrack could be created in several ways, however, the track needs to be continuous and smooth enough for the vehicle to drive through, yet randomly shaped so that it possess unstable features. The racetrack can be created with the properties described above, by using a mathematical curve, for instance the curve given by the cubic Bézier formula [10]:

\[ r(t) = (1-t)^3p_1 + 3t(1-t)^2b + 3t^2(1-t)c + t^3p_2 \]  

(3.1)

where \( p_1 \) and \( p_2 \) are called points and \( b \) and \( c \) are called control points. The control points can be expressed using the slopes \( k_1 \) and \( k_2 \) at the points \( p_1 \) and \( p_2 \) as:

\[ b = p_1 + \alpha k_1 \]
\[ c = p_2 + \beta k_2 \]  

(3.2)

where \( \alpha \) and \( \beta \) are positive constants. In order to create more interesting curves, the Bézier formula is applied iterative over several points where the first and the last point is the same in order to create a closed curve. In the simulations, four pair of points are used. Consequently, this gives a curve with three connected sub-curves, each with a fixed number of uniformed spread parameter values. The points and slopes are selected arbitrary from an interval.

The steering capabilities of the vehicle is limited, thus the minimum radius curvature, \( R \), of the track must always exceed the vehicle’s minimum turning circle in order to guarantee that the vehicle is able to pass all the way around the track. The radius of curvature is given by [11]:

\[ R = \frac{1}{\kappa} \]  

(3.3)
3.1. THE SIMULATION MODEL

\[ R = \left( \frac{dx}{dt} \right)^2 + \left( \frac{dy}{dt} \right)^2 \right)^{3/2} \]

For simplicity, the width of the track is fixed. The coordinates of the track sides are given by the normal to the track points, calculated by the gradient.

Figure 3.1 depicts a typical racetrack with the midpoints dotted and the track sides lined.

3.1.4 The Vehicle Model

To guide its way through the track, the vehicle is equipped with a simple type of vision that enables it to measure the positions of nearby objects in front of the vehicle as well as on the side of the vehicle, e.g. track sides. This vision implies that the distance to track sides can be calculated and thereby the deviation distance from the track center and the curvature changes ahead of the vehicle.

The track is divided into segments, which are composed by the area between two track points obtained from the Bézier formula described above. The track points are written as \( p_0, ..., p_3 \) in figure 3.2. For each track point, the angles relative to the following track points are calculated. The first track point, \( p_1 \), is always in line with \( p_0 \) so this angle is disregarded. However, the angles to the following two points, \( p_2 \) and \( p_3 \), are a measurement of the curvature ahead of the vehicle. These angles are referred to as \( \phi_2 \) and \( \phi_3 \) and they are passed to the vehicle. For simplicity, the angles are calculated once for every segment, which is an obvious approximation. Nevertheless, the segments can be made arbitrary small, which makes the approximation error arbitrary small.
Furthermore, the deviation distance \( \delta \), that is the distance between the vehicle and the centerline, is passed to the agent, as well as the deviation angle \( \alpha \), that is the vehicle’s direction relative the the centerline. The described vehicle model is shown in figure 3.2.

![Figure 3.2: The vehicle model.](image)

Summarizing the above, the state function is given by the position relative the track center, the deviation angle relative the centerline and curvature changes given as relative angles to track points ahead of the vehicle. This completes the description of the sensors.

In order to make the description of the vehicle model complete, the dynamics of the vehicle have to be described. The vehicle has a constant velocity \( v \) and a direction \( \alpha \) at time \( t \). The position and direction of the vehicle is updated on regular time intervals \( \Delta t \). Suppose that during this time interval the vehicle changes its direction with the amount \( \Delta \alpha \). The position of the vehicle relative to the previous time \( t \) is then:

\[
\begin{align*}
    x(t + \Delta t) &= x(t) + v \Delta t \sin(\alpha(t) + \Delta \alpha) \\
    y(t + \Delta t) &= y(t) + v \Delta t \cos(\alpha(t) + \Delta \alpha)
\end{align*}
\]

(3.4)

Formula 3.4 is written with respect to the axis in figure 3.2.

The direction changes, \( \Delta \alpha \), are limited to a maximum value in order to prevent the vehicle to make complete turn thus preventing the vehicle to go in the opposite direction from which it is supposed to.
3.2 Learning Algorithms

In this chapter, the implementations of the simulation model are described. First reinforcement learning is described. Next, the combined method of reinforcement and a genetic algorithm is described.

3.2.1 Reinforcement Learning Using Q-learning

In reinforcement learning, the states and actions is a central concept. The value function can be used to learn the optimal policy, though this would require perfect knowledge of the transition function $\delta(s, a)$ and the immediate reward $r$ [12]. This is not the case for the chosen test problem. Instead of using the value function, the state-action function $Q$ can be used. Two proposed learning algorithms using the state-action function are the $Q$-learning algorithm and the SARSA algorithm previously described. Which one of them performs better is not clear and is out of scope of this Master’s project to investigate. However, $Q$-learning is chosen under the assumption that an off-policy learning method might perform better.

The Q-function

As previously mentioned, the vehicle is equipped with a couple of sensors to mediate information about the environment to the agent. The sensors in the vehicle are as follows; the deviation distance $\delta$, the angles $\phi_2$ and $\phi_3$ of two track points ahead of the vehicle and finally the deviation angle, $\alpha$, relative to the center line.

In each state, the agent has to choose which action to take. Each action taken in a state may lead to an unique next state and thereby gaining a particular reward. Hence, to find the optimum policy, $Q$-learning stores both the state representation and the action representation. This is done in the state-action function, referred to as the $Q$-function. Summing up the above, the state-action function can be written as:

$$Q(\delta, \alpha, \phi_2, \phi_3, a)$$

where $a$ is the action taken.

There are several ways to implement the state-action function. The main idea is to map the inputs, i.e. the given information about the environment, to some sort of representation of the state. For the test problem in question, the inputs to the state-action function are nearly continues and hence representing this function would require a huge $Q$-function. Consequently, to avoid excessive computation, continuous $Q$-functions should not be used.

A possible solution is to make a discrete mapping. Several approaches are proposed, for example with a neural network [13]. Which method would be optimal for the test problem in question could be investigated, however, that...
is out of scope of this Master’s project and consequently a simple method is used.

The mapping is done using a simple look-up table in the following manner; for each sensor’s variable, a maximum and minimum value is established by running the simulation several times, see figure 3.3. The interval is divided into sub-intervals that are assigned discrete values. These values represent the discrete environment variables.

This method could indeed be improved by using a priori knowledge about how the policy behaves, for example the deviation angle relative to the center line is expected to be close to zero rather than to be close to $\pi/2$ since the former variable values result in higher rewards compared to the latter. Knowing this, the mapping could be made coarse for $\alpha$ values near $\pi/2$ and less coarse for $\alpha$ values close to zero. Observing figure 3.3, it is motivated to use priori knowledge for the deviation angle and deviation distance.

![Figure 3.3: Typical sensors input distribution (training track one). The y-axis is the number of times each input value has occurred. The x-axis is the sensor values, respectively.](image)

Each sensor variable is mapped into three discrete values. Figures 3.4 (a) and 3.4 (b) show this mapping for the deviation angle and deviation distance. The mappings of $\phi_2$ and $\phi_3$ are done with equal sized intervals.

A similar mapping is also used for the action values. For each state, the agent can take five different actions with the corresponding $\Delta \alpha$ values:

$$\Delta \alpha = \frac{1}{50} [-2\pi, -\pi, 0, \pi, 2\pi] \quad (3.6)$$

Summing up, the $Q$-function uses four environment variables. Each vari-
able can take three different values. That gives a state space of size $3^4 = 81$. In each state, five different actions can be taken. Altogether, this gives a $Q$-function with $3^4 \cdot 5 = 405$ entries. The $Q$-function is implemented using a matrix of size $3 \times 3 \times 3 \times 3 \times 5$.

**Rewards**

Besides states and actions, there is another important part of reinforcement learning, namely the rewards. If no reward is given, there is no way for the vehicle to know which states-actions pairs are good and which are not. In the racetrack model, three objectives were presented, namely to avoid contact with track sides, minimize steering actions and keep close to the centerline. These objectives are accomplished by giving rewards accordingly, that is if the vehicle is far from the center line or it changes its direction by a big amount it should get a small reward compared to if it is close to the center line and makes only small changes of its direction. This could be implemented in a linearly manner so that rewards are linearly proportional of the deviation distance and so forth. However, considering the objectives this might not be the optimal solution.

Consider an equally small change in deviation distance when the vehicle is near the centerline and when it is near the track side. With linear rewards, a small change would give equivalent changes in reward despite the different outcome of those actions. One of them could be devastating while the other would be completely harmless. Accordingly, the reward function should have a greater slope for variable values opposing the objectives compared to variable values in line with the objectives. Such behavior can be achieved by using a Gaussian function, shown in figure 3.5, given by:

$$f(x) = e^{-x^2}$$

**Figure 3.4: Mapping of input variables.**
gives reward values close to one for small deviation distances while distances close to the track border gives a value close to zero. This requires that the $\delta$ variable is rescaled to fit the curve.

\[ r = e^{(-C_1 \delta / \text{TrackWidth})^2} + e^{(-C_2 \alpha / \text{MaxAngle})^2} \]  \hspace{1cm} (3.8)

where $C_1$ and $C_2$ are constants which both were set to 10 after experiments.

Some actions are fatal to make, both in reality and in the test problem. Fatal actions in the test problem are those who put the vehicle outside the track boundaries. Those actions are rewarded with a value less than zero:

\[ r = -D \]  \hspace{1cm} (3.9)

where $D$ is set to 5 after experimentation.

**Action Selection**

Three action selection methods were discussed in section 2.1.4. It is there argued that SoftMax action selection has some advantages over $\epsilon$-greedy action selection. Due to this fact, and because of SoftMax is more suitable to use in the combined GA and RL algorithm, see section 3.2.2 for motivation. SoftMax action selection is used in the simulations.
3.2. LEARNING ALGORITHMS

Decreasing $\tau$

The idea with SoftMax action selection is the following: from the beginning, when the state space is unexplored, every action in a state is chosen with equal probability. As the state space is being explored, the probability of choosing good actions compared to bad actions increases, see section 2.1.4.

The $\tau$ parameter has to be chosen so that all actions are explored. Yet in the last episodes, the learning algorithm has to converge. For some $\tau$ values the learning algorithm converges, but to a suboptimal value, see section 4.3.3. To avoid this, the $\tau$ parameter can be decreased for every episode in the following manner:

$$\tau = C(1 - \frac{Episode}{TotalEpisodes})$$  \hspace{1cm} (3.10)

where $C$ is a constant. The optimum $C$ value is found to be 2.75, see section 4.3.2.

3.2.2 Reinforcement Learning and Genetic Algorithm

The main objective of this Master’s project is to study the interaction between individual learning and evolutionary learning using reinforcement learning and a genetic algorithm respectively. The idea is to combine RL, which starts with no prior knowledge for every new track, with a GA that adapts to the environment, thus improving the properties of the RL in terms of convergence rate.

How to combine RL with a GA is not obvious. One way to incorporate GA functionality into the RL algorithm is by letting the GA tell the RL algorithm, somehow, which actions should be tried first, loosely speaking. These actions can be called the preferred actions. As the RL explores and exploits the environment in which it acts, it stores knowledge in the $Q$-function on how to obtain a high return. Hence, a natural way to mediate information about the preferred actions is by having the GA altering the $Q$-function in such a way that the preferred actions are given special treatment, i.e. by adding a bias to the $Q$-function for those state-actions pairs that are preferred.

The bias can be added in several ways. Two different models of adding bias were tried out in this Master’s project. Those two models are described in the following section.

Two different models

In the first model, model one, one of the actions of each state is given a bias. The genome determines which action that is given this bias. This means that for initial $Q$-values:
\[ Q(s_j, a_k) = 0 \]  \hspace{1cm} (3.11)

where \( j \) and \( k \) are integers so that \( s_j \in S \) and \( a_k \in (A_t) \), \( (A_t \) and \( S \) are defined in 2.1.2), except for an action \( a_i \), determined by the genome. For which:

\[ Q(s_j, a_i) = bias_1 \]  \hspace{1cm} (3.12)

The value of the \( bias_1 \) is set to the mean of the standard deviation of the \( Q \)-function, i.e. calculating the standard deviation of each state and then taking the mean over all states. This value is calculated to approximately 0.35 after a trial run on training track one, and is used as the bias in the simulations.

The second model, model two, is similar to the first. The difference is that a state specific bias is introduced. This means that the entire state can have a bias. This functionality is added on top of that of model one which means that there could be two biases, one for the entire state and one for one specific state-action pair in each state. For initial \( Q \)-values this implies that:

\[ Q(s_j, a_k) = \{ 0, bias_1 \} \]  \hspace{1cm} (3.13)

except for one action \( a_i \), determined by the genome for which

\[ Q(s_j, a_i) = bias_2 \]  \hspace{1cm} (3.14)

where \( bias_2 = 2 \cdot bias_1 \).

The state-action bias in model two, \( bias_2 \), is twice the state bias, \( bias_1 \), in order to make it possible for the learning algorithm to separate them.

Each gene is coded with three bits. A motivation for this is given in section 3.2.3. Three bits genes limit the number of combinations to eight. The eight combinations were divided into two groups: in the first group, the state bias is set to zero and in the other group, the state bias is set to a non-zero value, i.e. \( bias_1 \). Five combinations are assigned to the first group and three to the last. In the fist group, one of the five state-action pairs is assigned a bias, whereas in group two only three state-action pairs can be assigned a value. The state-action pairs that result in the smallest change of direction are chosen to be eligible. This is motivated by that small changes of the direction are likely to occur more often than big changes. See table 3.1 and table 3.2 for details.

The purpose of biasing the \( Q \)-function is to guide the exploration of the state space. The guidance is not always reliable because of the random features of the tracks.

One way to announce to the action selection method, which genes to trust and which not to trust, is by setting the amount of exploration to be

\[ Q(s_j, a_k) = 0 \]  \hspace{1cm} (3.11)
done for each state. A handy way to achieve this is by setting the parameter \( \tau \), equivalent to a temperature in a Boltzmann distribution, in the SoftMax function in equation 2.6. Small values of \( \tau \) correspond to more exploitative behavior, larger values support more explorative actions. Using the \( \epsilon \) in \( \epsilon \)-greedy selection to achieve this is not immediately applicable because \( \epsilon \) needs to be decreased for every episode whereas the temperature in SoftMax function is constant. This is one of the reasons why SoftMax selection is chosen as the action selection policy in the Q-learning algorithm, see section 3.2.1.

Summing up, for every state there are two genes: one for the bias and one for the temperature. The temperatures, denoted \( \tau \), are coded with three bits and are given in the following interval:

\[
\tau = 0.2 + 0.075i, \quad i = 1..8
\]  

(3.15)

The state specific \( \tau \)-values are used both in model one and in model two.

### 3.2.3 The Genetic Algorithm

#### Analysis of GA

In order to establish some implementation details, such as how to encode the genome and how to set the probability of mutation, the behavior of the GA needs to be analyzed. To perform the analysis, a deterministic test problem is created. A random test genome is created and a population with random genomes is created. Fitness is calculated as the number of matching genes between the test genome and the currently evolved genome.

As mentioned before, the RL algorithm uses a state space of size 81. For every state, the GA stores the two genes: one for the bias and one for the state specific \( \tau \)-value. This means that the numbers of genes required are \( 81 \cdot 2 = 162 \). Each gene is coded as a binary number. More bits could possibly give a better value. Nevertheless, more bits give a larger search space for evolution to explore and thus increases the convergence time. Through empirical studies, it is revealed that convergence time increases drastically
as the number of bits increases. This is due to the fact that the number of combinations to explore increases exponentially as the number of bits increases.

The number of generations is set to 25, which is an acceptable value both in terms of computation time, see section 4.4.1, and in terms of gained fitness.

It is discovered that when using three bits genes, the algorithm converges to approximately 61% of maximum fitness after 25 generations, see figure 3.6 (a). However, if the genes are coded with 5 bits the algorithm converges to approximately 55% of maximum return after 25 generations, see figure 3.6 (b).

Reducing the number of bits to use from five to three increases the final fitness, although the fitness is still not sufficiently large. To increase the fitness even further, without increasing the number of generations used, the GA is applied to a subset of states only. That is, a subset of states is treated as they are one single state and the genes are evolved for that subset.

The two most important input variables are the deviation angle, $\alpha$, and the deviation distance, $\delta$, from the centerline. Consequently, the subset is made up of the states depending on $\alpha$ and $\delta$ and independent of curvature angles, $\phi_1$ and $\phi_2$. That is, every state with identical $\alpha$ and $\delta$ but with different $\phi_2$ and $\phi_3$ are treated as a single state by the GA. This reduces the state space to $3 \cdot 3 = 9$. Applying the GA on this subset on the deterministic test problem gives a fitness value of approximately 82% of the maximum return after 25 generations, see figure 3.7. This is far a better value than what is achieved without the use of the subset.
3.3 Development of the Simulation Tool

The simulation tool and learning algorithms are all implemented in MATLAB. Other languages were considered such as C++. However, MATLAB was chosen because of its superior mathematical capabilities.

As mentioned before, the simulations are computing intensive. Consequently, several methods are used to reduce the simulation time. MATLAB has the capability to compile the functions into an executable application.

Mutation

Mutation is indeed important. Without mutation, the genetic algorithm would not be able to explore the entire search space. The probability of mutation is important to consider; a low mutation probability implies a small probability to explore and a big probability to exploit. Exploring is a good thing to do for early generations, for which the genome is not well fitted, whereas exploiting is a good thing to do when the genome is well fitted. Thus, the probability of mutation should decrease as the fitness of the population increases. This can be achieved by multiplying the mutation probability by a mutation factor < 1. The mutation probability is set to 5% for generation one. Figure 3.8 shows resulting return using different multiplication factors.

As depicted in figure 3.8 a), a mutation factor of 0.950 and 0.960 gives equal final fitness. While, a mutation factor of 0.960 performs slightly better in terms of mean fitness, see figure 3.8 b). Consequently, the mutation factor is set to 0.960 in the simulations throughout this report.

Figure 3.7: Analysis of the GA applied to a subset of states of the deterministic test problem. Population size: 20

3.3 Development of the Simulation Tool
Figure 3.8: The return using three different multiplication factors. Three bit genes. Population size: 20.

This reduces the simulation time dramatically, approximately by 50% or more.

To minimize the simulation time needed, precalculations are made and look-up tables are used whenever it is possible.

Further, all learning algorithms and the GA are implemented with no use of existing libraries etc. This is motivated by that they are quite simple to implement and also that it provides further understanding of the algorithms.
Chapter 4

Simulations and Results

The purpose of the simulations is to investigate whether the combination of RL and GA, as described in model one and model two, is successful. A successful algorithm is here defined as achieving better than pure RL in terms of final fitness and convergence time.

The idea of combining RL with a GA is that the latter is able to slowly adapt to the environment and thus is able to guide the exploration directed by the RL. To enable this feature, the GA needs to be trained on randomly created tracks, gaining some knowledge of the stable features of the tracks. Consequently, both training tracks and test tracks are needed to be created. This is described in the following section.

4.1 Tracks

In total, six tracks are created, three for training purposes and three for testing purposes. The training tracks are shown in figure 4.1 and the test tracks are shown in figure 4.2.

(a) Training track one.  (b) Training track two.  (c) Training track three.

Figure 4.1: The three training tracks.
4.2 Terminology

First, some terminology will be defined. Throughout this report, a run is referring to a certain number of completed episodes, usually 200 episodes as in figure 4.3. An episode is here defined as being the actual simulation: the vehicle always starts from the starting position of the current track and is simulated for a fixed time period, the simulation time. Hence, the actual updates done by the learning algorithm depends on the velocity. The simulation time is set so that the vehicle has the ability to approximately complete a whole turn round the track.

Further, the maximum return is referring to the run with maximum final return, that is the return on the last episode.

The mean return is calculated according to the following formula:

$$\text{MeanReturn} = \frac{\overline{R}_1 + \overline{R}_2 + \ldots + \overline{R}_n}{n}$$ (4.1)

where $\overline{R}_1 + \ldots + \overline{R}_n$ are vectors containing the return for each episodes for the runs 1..n where n is the number of runs.

4.3 Pure RL

To be able to make a fair comparison between the combined RL and GA algorithm and pure RL, the optimum parameters for the RL need to be established in advance. Otherwise, the combined RL and GA algorithm might perform better than RL merely because of ill-fitted parameters. First of all, the number of episodes to use needs to be investigated. For practical reasons, i.e. the computation time, the number of episodes is set to 200. Ideally, the number of episodes should be set to infinity to fulfill the condition of the convergence theorem [14] of Q-learning. Nevertheless, 200 episodes is a reasonable approximation for infinity in this case.

Figure 4.3 shows the return versus episodes on training track one. The difference between maximum return and minimum return indicates that the algorithm converges to a suboptimal value. A way to avoid this is discussed in 4.3.2.

Figure 4.2: The three test tracks.
4.3. PURE RL

4.3.1 Finding the optimum $\tau$ value

What differs between the pure RL algorithm and model one and two is that the two models are able to alter the temperature, $\tau$. Hence, the range, in which $\tau$ can vary, and is therefore needed to be selected properly.

The purpose of the selected $\tau$ range is to maximize the return. To achieve this, the range of $\tau$ is chosen to cover suboptimal values below and above the optimum value and thus including the optimum value. The selected range given by equation 3.15 meets that demand.

The parameter sweep of $\tau$ is carried out on all three training tracks and the return is calculated as the mean of the three tracks. To reduce the random properties of the RL algorithm, each parameter value is tested 30 times for each track and the mean is calculated. The result of the parameter sweep is presented in the figure 4.4.

According to the figure 4.4, the optimal value of $\tau$ is 0.425, which is the value used as $\tau$ for the pure RL algorithm.

4.3.2 Decreasing $\tau$

In order to prevent the RL algorithm to converge to a suboptimal value, as described in section 3.2.1, the RL algorithm is equipped with a decreasing $\tau$, in accordance with equation 3.10. The optimal value of the constant $C$ is found by a parameter sweep, see figure 4.5. Each $C$ value is tested 30 runs for each track and the mean is calculated.

Figure 4.5 implies that the optimal value of $C$ is 2.75 and $C$ is set to this value for all simulations concerning the decreasing-$\tau$ algorithm.
CHAPTER 4. SIMULATIONS AND RESULTS

Figure 4.4: Parameter sweep for the $\tau$ parameter.

Figure 4.5: Mean return using RL with decreasing-$\tau$ with different $C$ values.
4.3. PURE RL

4.3.3 Performance on Test Tracks

As previously described, the idea with using a decreasing $\tau$ is to avoid the algorithm to converge to a suboptimal value. If the suggested method is successful, the performance of the decreasing $\tau$ method should be better than pure RL on the test tracks.

Each of the two algorithms are tested for 30 runs on each test track and the mean return is calculated. The performance on test track one can be studied in figure 4.6.

According to figure 4.6, the decreasing-$\tau$ method performs slightly better than pure RL. Comparing the two algorithms, it shows that the decreasing-$\tau$ algorithm performs poorly for more than half of the episodes, whereas the pure RL algorithm converges rather fast but to a lower value. This implies that the decreasing-$\tau$ algorithm is successful in avoiding to converge to a suboptimal value. A comparison of all the test tracks are shown in table 4.1. The table shows that the decreasing-$\tau$ algorithm performs better in terms of mean return on all test tracks. However, considering the maximum return, the difference is not that obvious. Considering the maximum return on test track two, pure RL performs best.

In order to scientifically determine which of the two algorithms that performed better, the $z$-test is applied. See appendix A for a description of the $z$-test. The results of the $z$-test is presented in table 4.2. The decreasing-$\tau$ algorithm performs significantly better on test track two and three, whereas on test track one, there is no significant difference. Nevertheless, the decreasing-$\tau$ algorithm can be regarded as the better algorithm.
4.4 The Combined RL and GA Algorithm

4.4.1 Evolving the Genome

After determining the optimum mutation curve, see section 3.2.3, and the optimal number of bits to use, see section 3.2.3, the genome is evolved. As mentioned before, three tracks were used as training material. In order to smooth out the random properties of the RL algorithm, every individual is tested for 30 runs on each track and the mean is used as the fitness value. The population size is set to 20 individuals which is the same size used in the deterministic problem, see section 3.2.3.

In order to reduce the time needed to do the evolution, the fitness is calculated after only 50 episodes.

First, the genome for model one is evolved. The fitness versus the generation is shown in figure 4.7 (a).

Because of the differences between model one and model two, see section 3.2.2, it is motivated to evolve the genome separately for model two. The result is presented in figure 4.7 (b).

4.4.2 Performance on Test Tracks

As shown in figure 4.4.2, model two performs better on test track one. The convergence rate is quite similar for the two methods but model one converges to a suboptimal value. The figure also reveals that model one is quite stable once it has converged, whereas model two is more unstable. This could be an explanation why it is more successful in avoiding to converge to a suboptimal value.
4.5. COMPARING MODEL TWO AND DECREASING-$\tau$

In the previous sections, it is determined with the $z$-test that the decreasing-$\tau$ algorithm performs better than pure RL. It is also concluded that model two performs better than model one. That leaves us with one candidate
40  

CHAPTER 4. SIMULATIONS AND RESULTS

<table>
<thead>
<tr>
<th>Table 4.3: Return of model one versus model two.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Track One</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Max return</td>
</tr>
<tr>
<td>Model one</td>
</tr>
<tr>
<td>Model two</td>
</tr>
<tr>
<td>Mean return</td>
</tr>
<tr>
<td>Model one</td>
</tr>
<tr>
<td>Model two</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4.4: The result of z-test for model one and model two.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Track One</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Model one, mean</td>
</tr>
<tr>
<td>Model two, mean</td>
</tr>
<tr>
<td>z-value</td>
</tr>
<tr>
<td>α</td>
</tr>
</tbody>
</table>

based on pure RL and another candidate based on a combined RL and GA algorithm.

One question remains; which of these two candidates performs best? To determine this, the z-test is applied as before. The result is shown in table 4.5. Apparently, model two is superior to the decreasing-τ algorithm, because the former is significantly better than the latter on all three test tracks.

4.5.1 Limitations

Although the test tracks are created randomly and the combined RL and GA algorithm are successful on all test tracks, see section 4.5, it is not clear under which conditions the combined algorithm is successful. In order to investigate this and to determine the limitations of the combined RL and GA algorithm, the conditions in the test problem were altered somewhat.

The test problem could of course be altered in several ways: for example changing the curvature of the tracks or the track width. However, for simplicity, only the velocity of the vehicle is changed.

<table>
<thead>
<tr>
<th>Table 4.5: The result of z-test for model two and decreasing-τ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Track One</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Model two</td>
</tr>
<tr>
<td>Decreasing τ</td>
</tr>
<tr>
<td>z-value</td>
</tr>
<tr>
<td>α</td>
</tr>
</tbody>
</table>
4.6. ANALYSIS

Two changes were tried out; increasing the velocity by 50% and decreasing the velocity by 50%. The result is shown in Table 4.6 and Table 4.7.

In the first experiment, when the velocity is increased with 50%, model two performed better on all three test tracks. In the second experiment, model two still performs better on track one and track two. While, on track three, the model two performed worse than decreasing $\tau$. This behavior is analyzed in the following section.

### Table 4.6: Mean return. Velocity increased with 50%.

<table>
<thead>
<tr>
<th>Model Two</th>
<th>Test Track One</th>
<th>Test Track Two</th>
<th>Test Track Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increasing $\tau$</td>
<td>68.0</td>
<td>81.1</td>
<td>77.3</td>
</tr>
<tr>
<td>$z$-value</td>
<td>3.17</td>
<td>5.82</td>
<td>2.61</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.9975</td>
<td>0.995</td>
<td>0.99</td>
</tr>
</tbody>
</table>

### Table 4.7: Mean return. Velocity decreased with 50%.

<table>
<thead>
<tr>
<th>Model Two</th>
<th>Test Track One</th>
<th>Test Track Two</th>
<th>Test Track Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decreasing $\tau$</td>
<td>88.3</td>
<td>100.8</td>
<td>86.6</td>
</tr>
<tr>
<td>$z$-value</td>
<td>2.04</td>
<td>1.30</td>
<td>-1.44</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.95</td>
<td>0.75</td>
<td>-0.90</td>
</tr>
</tbody>
</table>

### Analysis

In the previous section, it is concluded that model two performed worse than decreasing $\tau$ on test track three when the velocity is decreased with 50%. This could give an indication under which conditions model two is successful.

A possible explanation why model two is unsuccessful on merely one track of several tested, is that the states needed to be explored were not explored during the training, i.e. during the evolution of the genes. In order to verify this hypothesis, the following test is committed: firstly, model two is run on all three training tracks and the visited states are noted. As usual, each track is tested for 30 runs in order to get a good sample. Secondly, model two is tested on the test tracks with increased and decreased velocity, and again the visited states are noted. The states visited for the training tracks are summed and taken as a reference, denoted $V$.

The visited states for the test tracks are noted in two ways: Firstly, all states visited during the 30 runs are noted. Secondly, the maximum run is selected and the states visited for this optimum run are noted.

Further, the visited states for the training tracks and the test tracks are compared. Table 4.8 shows the relation between the matching states of the
track and the visited states of the training tracks. A visited state, \( v_i \), of the
test track is a matching state if and only if \( v_i \in V \), where \( i \) denotes the state
number. The relation in table 4.8 is calculated as follows:

\[
f = \frac{\# \text{ matching states}}{\# \text{ visited states}}
\]  

(4.2)

Considering table 4.8, there is a correlation between the \( f \)-value given
by equation 4.2, and the return given in table 4.6 and 4.7. For example,
the return for model two, when the velocity is decreased with 50\%, is lower
than the return given by decreasing \( \tau \) under the same conditions, see table
4.7. The \( f \)-value for the corresponding track and velocity is the lowest,
considering the maximum run, compared to the other tracks.

A probable explanation for this behavior is that when the states needed
to find the optimum policy are not previously explored, the algorithm is
deluded to explore suboptimal states, thus may converge to a suboptimal
policy.

As mentioned in section 3.2.2, the \( \tau \) parameter determines the balance
between exploitative and explorative behavior. Table 4.9 shows the \( \tau \)-values
evolved for model two, which uses a subset of states previously described.

Although the evolved \( \tau \)-values are quite homogenous, there are two ex-
ceptions. First of all, for the state corresponding to \( \alpha = 2 \) and \( \delta = 1 \), the
\( \tau \)-value is the lowest possible. Second of all, for state corresponding to \( \alpha = 3 \)
and \( \delta = 3 \), the \( \tau \)-value is somewhat higher than for the other states. This
indicates that the need of exploration and exploitation differs between those
states.

To investigate this difference, the following test is performed; for each
state, the preferred action determined by the genome is compared to the
actual action taken by the maximum run. This test is performed for all
states, not only those in the subspace. The percentage of matching actions
of the bias and the best policy is shown in table 4.10. It shows that the state
with lowest \( \tau \)-value indeed has a high rate of matching actions, and the state
with highest \( \tau \)-value has a lower rate of matching actions. Hence, there is a
correlation between \( \tau \)-values and the rate of matching actions, however, the

| Table 4.8: The \( f \)-values in %.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Test Track One</td>
<td>Test Track Two</td>
</tr>
<tr>
<td>50% increased vel.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>84</td>
<td>100</td>
</tr>
<tr>
<td>Maximum</td>
<td>78</td>
<td>95</td>
</tr>
<tr>
<td>50% decreased vel.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>82</td>
<td>100</td>
</tr>
<tr>
<td>Maximum</td>
<td>75</td>
<td>91</td>
</tr>
</tbody>
</table>
correlation is not perfect.

To understand the mechanism behind the need of different explorative behavior, the stable features of the state can be investigated through inspecting the use of the state in different situations, e.g. in which segment and in what track the state is visited. This is performed in the following test; for each segment of each track, the number of visits to the particular state is compared to the sum of visits to the other states in that segment.

The relative visits to the particular state are calculated as follows:

$$\text{RelativeVisits} = \frac{\text{Visitations}(s_i)}{\sum_{k=1}^{N}\text{Visitations}(s_k)}$$

(4.3)

where $s$ is the state, $i$ is the state being evaluated and $N$ is the total number of states in current segment and subspace.

Figures 4.9 (a) and (b) show the relative visits, according to equation 4.3, for the two states with highest and lowest $\tau$-values, on the three test tracks respectively. Figure (a), with the lowest possible $\tau$-value, shows several similarities between the three test tracks in terms of total visits and in terms of periodical patterns. Whereas in figure (b), the similarities between the tracks are less obvious. Track three is aberrant to the other tracks with a peak value close to segment 60. These differences supports the idea of states with more or less stable features; the state shown in figure (b), is used under specific circumstances, thus could be considered more instable than the state in (a), which is visited in more similar situations. Instable features require a more explorative behavior than stable features, thus the former needs a higher $\tau$-value than the latter which agrees with actual $\tau$-values evolved, see table 4.9.

Comparing figure 4.9 (a) with (c), which depicts the relative curvature of the test tracks, shows to some extent that where (c) has its peak (a) has non-peaks. This indicates that this state is more often visited for segments with small relative curvature compared to segments with high relative curvature. In other words, segments which correspond to straight sections of the tracks.

This is not what could be expected. A state with these properties should be the center state, i.e. $\alpha = 2$ and $\delta = 2$, depicted in figure 4.9 (d), since this state corresponds to a vehicle position closest to the center line. For the center state, there should be only minor differences between the tracks and hence require a low $\tau$-value which is the case for the state corresponding to $\alpha = 2$ and $\delta = 2$. An explanation to this displacement could be that all the tracks are curved in similar shapes. Hence, the state of $\alpha = 2$ and $\delta = 1$ could in fact have the best correspondence to the mean curvature of tracks, thus be the most stable state and require the lowest $\tau$-value.
Table 4.9: Evolved $\tau$ values for different states.

<table>
<thead>
<tr>
<th></th>
<th>$\alpha = 1$</th>
<th>$\alpha = 2$</th>
<th>$\alpha = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta = 1$</td>
<td>0.425</td>
<td>0.275</td>
<td>0.425</td>
</tr>
<tr>
<td>$\delta = 2$</td>
<td>0.425</td>
<td>0.425</td>
<td>0.425</td>
</tr>
<tr>
<td>$\delta = 3$</td>
<td>0.425</td>
<td>0.425</td>
<td>0.500</td>
</tr>
</tbody>
</table>

Table 4.10: Matching actions of bias and those taken in the maximum run (%).

<table>
<thead>
<tr>
<th></th>
<th>$\alpha = 1$</th>
<th>$\alpha = 2$</th>
<th>$\alpha = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta = 1$</td>
<td>78</td>
<td>93</td>
<td>89</td>
</tr>
<tr>
<td>$\delta = 2$</td>
<td>78</td>
<td>93</td>
<td>85</td>
</tr>
<tr>
<td>$\delta = 3$</td>
<td>85</td>
<td>81</td>
<td>85</td>
</tr>
</tbody>
</table>

Figure 4.9: The number of visitations of the particular state relative to the other states.
Chapter 5
Conclusions

The main issue studied in this thesis was how to combine individual based learning with population based learning. The purpose is to increase the learning rate of the individual learner by using prior knowledge discovered by the population based learner. The integrated learning scheme is studied under the assumption that the environment possesses stable as well as unstable features of which the stable ones are learned by the population based learner to guide and facilitate the task of the individual learner.

A novel method was proposed, inspired by natural evolution and learning in animals. The method combines an individual learning agent, namely reinforcement learning, with a population based learning scheme, namely a genetic algorithm. The common link between both learning schemes is the state-action function, which captures the knowledge accumulated by the RL.

It was demonstrated that the GA can improve the convergence speed and the performance of the RL learner by encoding priori knowledge and indications of stable and unstable features into the chromosomes.

To the author’s knowledge, a GA has not previously been applied to evolve guidance to a RL by biasing the state-action function and by controlling the balance between explorative and exploitative behavior. By utilizing this guidance, the performance of the RL can be increased in terms of convergence speed, which was demonstrated to be effective in avoiding convergence to a suboptimal policy and thereby increasing the overall performance of the RL learner.

Furthermore, the presented novel method shows promising result. The novel method referred to as model two in the report, is significantly better than the best RL method, referred to as decreasing $\tau$. The $\alpha$-value is at least 0.900 for all three test tracks. Even though the properties of the vehicle model is changed by altering the velocity of the vehicle, model two performs significantly better compared the decreasing $\tau$ method on four of six test tracks, with an $\alpha$-value of at least 0.95.

Although the presented novel method was demonstrated superior in per-
formance in most cases compared to RL alone, the method is not applicable to all situations for which RL can be applied. The novel method is computationally expensive during the evolution of the genome. Hence, the novel method is suitable only for problems which can be trained offline or for which training data is readily available.

Nevertheless, there are learning problems for which the novel method is a valid approach, namely those that are a part of a more general set of learning problems that have common properties. The chosen test problem of this thesis was to control a vehicle driving through a race track. This can be compared to the realistic problem of manoeuvering a Formula One car, for which the approach of the novel method is at least plausible. The alternative would be, using pure RL, to reduce the state space by replacing the features that are invariant in the $Q$-function with a static value for these features. However, the general problem is that the nature of the features are not known in advance, that is if they are static or dynamic. In the novel method, the features are assigned values in accordance with their degree of dynamic or static nature. This makes the approach conceptually interesting, for example in the context of learning in robotics.

Finally, some further efforts are worth spending on this method.

5.1 Future Work

An interesting fact that was revealed in this thesis was that the novel method was successful in avoiding the RL to converge to a suboptimal policy. This fact is worth further investigations. For example, is this property preserved if the number of episodes are increased?

Further, the extensive use of computing required in the evolution of the genome is not satisfactory. Could this be improved in some way? There might be a more effective way of finding the optimal genome. Instead of doing a full simulation, the GA could be trained on subparts of the track only, and later join using some statistical method.

Finally, the issue of adaptation is not at all studied in this thesis. A real problem for which there exists stable and unstable features, will most likely be time-dependant. Thus, in such cases there is a need of a adaptive behavior. To incorporate adaptation into the novel method, a feed-back functionality from the actual simulation to the genome has to be included.
References


Appendix A

The z-test

According to [15] a method of comparing to means is the z-test. Let $X$ and $Y$ be normal distributed random variables with the independent samples $X_1, X_2, \ldots$ and $Y_1, Y_2, \ldots$ respectively, then the z-test can decide if the two variables are significantly discriminated. The z-test is applied as follows; two hypotheses are stated, namely the null hypothesis that say that the two variables are equal and the alternative hypothesis that say that there is a significant difference between the two variables. The z-value is given by:

$$z = \frac{\bar{X} - \bar{Y}}{\sigma_{X-Y}} \quad (A.1)$$

and

$$\sigma_{X-Y} \approx \sqrt{\frac{S_x^2}{N_x} + \frac{S_y^2}{N_y}} \quad (A.2)$$

where $S_x$ and $S_y$ are the standard deviation for $X$ and $Y$ respectively and $N_x, N_y$ are the number of samples in $X$ and $Y$ respectively.

If the z-value is above the critical region the null hypothesis is rejected, that is $z > z_\alpha$, where $\alpha$ is the level of significance. The value of $z_\alpha$ for different levels of significance can be found in various tables, for example in [?].