Intelligent game agents

Author:	Mikael Heinze
Educational institute:	Aalborg University Esbjerg
Group:	CIS3-A2
Date:	28th May 2004
Supervisor:	Henrik Legind Larsen
Assisting supervisor:	Daniel Ortiz Arroyo

_____

Mikael Heinze

Preface

This paper is part of a dissertation on the subject of intelligent game agents on the 9th and 10th semesters of Master of Science – Software Engineering.

This first part of the dissertation is produced on the 9th semester at Aalborg University Esbjerg, in the period 2nd February 2004 to 28th May 2004, and is to work as a major milestone and lay the foundation for continued work on the 10th semester.

The paper has been composed by Mikael Heinze. Supervisor was Henrik Legind Larsen, assisting supervisor was Daniel Ortiz Arroyo.

The content of the enclosed CD is described in the appendix section 10.1

Abstract

The topic of getting machines to play classical board games at the same level as top ranking human players, is a research area of artificial intelligence that has received a considerably amount of attention from researchers during the years. But although the area to this point has received quite a lot of attention, and there exist examples of computers that has defeated top ranking human players, it is still an area that intrigues researches, and where much research still can be done.

This paper presents a new way on how computers can be programmed to learn to play classical board games against humans. The model is identified as goal directed learning. The model has been implemented and tested in the domain of tic-tac-toe where it has shown to be highly successful. Suggestions on how to further improve the model such that it can be applied to more complex game domains is also presented. During the next and final 10th semester, work will continue and hopefully prove that the model can be successful integrated in a more complex game domain.

An important aspect to the dissertation is existing research in the areas of artificial intelligence, game theory and previously works done in relation to getting computers to play classical board games. The first half of the paper will therefore focus on these areas. This research has been done, as investigation of these areas was considered to be of high importance before work on a personal approach could begin.

Structure of paper

Besides some general introduction presented on this and previous pages, the structure of the paper is based on eight main chapters not including references and appendix.

Chapter one is an introduction chapter defining the project. Chapter two through four will cover existing research and theories related to artificial intelligence and games. Chapter five through seven will cover the development of a solution model useful for learning computer to play the classical board game tic-tac-toe. Chapter eight will conclude on the dissertation.

In chapter nine, a list of references can be found. Throughout the paper references to one of these references will be done by the syntax [x] where x has been replaced with a number identifying the exact reference.

At the end of the paper, chapter ten will hold the appendix of the paper. Besides a section describing content of the enclosed CD, rules of chess, checkers and backgammon can be found. This is all games that are mentioned throughout the paper.

Content

1. Problem definition

The object of this project is to investigate ways to model artificial intelligence (AI). The selected problem domain is to be a gaming environment where the problem is to develop a computer opponent.

Computer games make an excellent environment for AI research. Games offer a combination of simulation and reality. The environment in which a computer player interacts is virtual, but the environment is not a simulation of a problem domain, it is the problem domain. As a result issues such as noisy sensor data, hardware dependencies and other hardware related problems can be ignored, while still addressing realistic problems.

Many computer opponents found in games take a very non human like approach when playing. The follow some rules designed end implemented by the programmer and are at best on the same level as average skilled players. Further more they are unable to learn and adapt to new strategies. So is it possible, and how is it possible to create computer agent with the ability to learn and adapt strategies?

1.1 Limitations

Two perspectives can be taken when considering playing games on computers. First perspective is to make a computer play classical games that has been played for generations before the invention of the computer. This includes board and card games. The other perspective is the true computer games that have become a major industry over the last years. Some examples are first person shooters and real time strategic games.

Throughout this dissertation the perspective will be on classical games. This is because these games offer the best environment for focusing on AI alone. For successfully playing true computer games, it is often necessary to possess a range of other skills rather than intelligence alone. This includes coordination, precision, reflexes and speed. However if intelligence is designed correctly, there should not be any reason for, why intelligent strategies working within the domain of classical games, should not have the ability to be applied to the true computer games and other domains as well.

Throughout this paper classical games will in most cases be referred to as just games. While the term true computer games will be used to define these games if necessary.

1.2 Purpose and goals

To have a reference point throughout the dissertation, purpose and goals for the dissertation are defined.

Purpose

- Demonstrate an understanding of current research in the field of artificial intelligence.
- Analyze and define requirements necessary to develop an intelligent agent that plays classical computer games against humans.
- Contribute to the area of artificial intelligence and games with new thought, ideas and methods.

Goals

- Implementation of an intelligent agent in a game domain.
- This paper documenting the above purposes and goal as well as phases, aspects and considerations of the project.

A more in depth overview of the exact objectives of the dissertation are given in the next section.

1.3 The process

This dissertation is to be separated into two parts, one for each of the two semesters 9th and 10th of Master of Science – Software Engineering. The end of the 9th semester (this semester) can be considered a major milestone with a middle exam of the dissertation.

The 9th semester are to take a more theoretically approach focusing a great deal on existing research, while the 10th semester are to take a more constructional approach focusing more an a personal solution model identified at the end of the 9th semester.

Object of the 9th semester involves the study of artificial intelligence, game theory and related work. This is to lay a foundation and helping when defining a solution model for an agent capable of playing the simple game of tic-tac-toe. The identified solution is then to be implemented as a prototype agent and evaluated.

Experience obtained from the 9th semester is then to help on the 10th semester when a detailed system design are to be done, and the identified solution model is to be tested in a more complex game domain (probably backgammon). Extensions capable of improving the basic model identified on the 9th semester are also to be identified along with solution models for the identified extensions. Experiments and final evaluation are to be done at the end of the 10th semester.

Objects of 9th semester:

- Study artificial intelligence
- Study game theory

- Study related work
- Define requirements
- Identify a basic solution model
- Design and implementation of prototype agent capable of playing tic-tac-toe
- Prototype evaluation

Objects of 10th semester (preliminary):

- Identify extensions and improvements to solution model
- Detailed design
- Agent implementation
- Agent integration in game domain considered to be of high complexity
- Experiment and testing
- Evaluation

2. Artificial intelligence

This chapter is to give an overview of the area of artificial intelligence. Its intention is to give the author of this report insight into this field of artificial intelligence, first and foremost to get an understanding of what artificial intelligence is, but also to gather inspiration from the field. For the reader of this report it is to work as an introduction to the field of artificial intelligence.

This chapter consists of the following main sections:

- Defining artificial intelligence
- Searching
- Fuzzy logic
- Fuzzy controllers
- Genetic algorithms
- Artificial neural nets
- Bayesian networks
- Conclusion on artificial intelligence

2.1 Defining artificial intelligence

Most people have a general idea about what artificial intelligence is. Asking someone and the answer will almost certainly be something like: “it is making machines or computers think”. According to Dictionary.com this answer should be rather good as the two words are defined as:

Artificial	Intelligence
1. Made by humans; produced rather than natural. 2. Brought about or caused by sociopolitical or other human-generated forces or influences: *set up artificial barriers against women and minorities; an artificial economic boom.*	1. The capacity to acquire and apply knowledge. 2. The faculty of thought and reason. 3. Superior powers of mind. *See Synonyms at mind*.

So the study of artificial intelligence is about producing some human made device, with the ability to acquire knowledge, and using this knowledge to reason rational. But instead of just accepting this rather loose definition, lets take a more in depth look at some of the definitions of artificial intelligence, in the hope that a more exact definition can be establish.

Artificial intelligence goes back to 1950 where Allan Turing wrote a paper [5] in which he stated the question “can machines think?” In this paper he proposes a game that can test if a machine is intelligent. Today this test is known as the Turing test.

The proposed game in its original form is known as the imitation game. In this game there are three people, one male, one female and one interrogator. The interrogator is isolated from the male and female and all communication happens in written form. The goal for the interrogator is then to distinguish the male from the female simply by asking questions, while those being questioned should try to fool the interrogator. The Turing test is based on replacing one of participants with a computer. The goal for the interrogator is then to distinguish the human from the computer. If the computer can fool the interrogator, then it is said to be intelligent. For the Turing test to be successful the computer would need to poses the following capabilities:

- Natural language processing
- Knowledge representation
- Reasoning abilities
- Machine learning

Some has argued that the Turing test is a good test because to pass it, a wide range of knowledge about the world is needed. Further more a great deal of flexibility in dealing with new situations is needed as well. However it has also been criticized for only taking a behavioral rather then psychological view of intelligence. The computer does in other words not need to be intelligent; it just has to give an impression of having it.

The reason why people still debate the validity of the Turing test is most likely because a broader definition of what exactly artificial intelligence is, still have not been found. In [4] the author has collected eight definitions made from other authors, and made an interesting observation. That is, the definition of artificial intelligence can be organized into four categories. The four categories are:

Systems that think like humans	Systems that think rationally
Systems that act like humans	Systems that act rationally

Table 1: Four main categories for the definition of AI

First comparing the two categories at the top against the two categories at the bottom the difference is that in the topmost, the definition of artificial intelligence deals with thoughts and reasoning. Artificial intelligence should actually be able to think, whereas the goal in the two categories at the bottom is to make a system that behaves as if it is intelligent, although it might have no intelligence at all.

Now comparing the left column against the right one, success in the left column is measured in human performance. While in the right column success is measured against an ideal concept of intelligence, which the author has called rationality. This deviation comes from the fact that human sometimes do irrational actions. The meaning of irrational action in this context is simply the fact that humans sometimes are making mistakes unintentionally, or (having the power to) acting against all logic in a given situation.

An exact definition of artificial intelligence does unfortunately not exist. Although the term is widely used, it does not attempt to give a single and universally acceptable

definition. Almost every book on the subject of artificial intelligence, seem to have its own definition. The problem is then to find the definition that appeal to one. The following definition is made by Amit Konar in [1].

> Artificial intelligence can be defined as the simulation of human intelligence on a machine, so as to make the machine efficient to identify and use the right piece of “knowledge”.

Which in my opinion is in the same category as the Turing test, and that is systems that act like humans. Whether one agree with this definition or not, it should be intuitively clear, that artificial intelligence is about getting machines to find solutions to problems in ways somewhat equal to that of humans, and this involves borrowing characteristics from human intelligence, and applying them to machines.

2.2 Searching

Search approaches are described in [1]. One way of looking at artificial intelligence, is as search. One has a set of examples and a set of rules. The learning process is simply to apply the set of rules to the examples and start looking for a solution. In many cases the task of AI can simply be reduced to the problem of performing a search within a given search space. The search space is the collection of all possible and reachable solutions, and the search for a solution is then the problem solving.

2.2.1 Generate and test

The most straightforward approach is simply to test every possible solution to a problem and then picking the best one. However as all but the simplest problem either has an infinite, or finite but very large search space, this approach is impractical. It will simply be impossible to search every solution as it will take to long or even forever. One common use of this approach is in game playing programs. The computer will try to look as far ahead in the game as possible, and the consequently selecting the best move. This form of search is known as game trees, and is explained in section 3.2.

For searching large or infinite search spaces, the simplest alternative is to do a random search. Random solutions are generated until a satisfying solution is found or a sufficient large search space has been tested. There is no guaranty that this approach will find a useful solution unless allowed to run in an infinite time. If the search space contains very few non bad solutions, random search are unlikely to be of any use.

2.2.2 Hill climbing

This approach starts by generating a random solution which is to act as the starting state. But in contrast to the random search this solution is used to generate one or more close variants. If a better variant is found this is instead used to guide the further search and so on. If after several iterations no improvement is found, the search terminates. However in the case that all neighborhood states are equal or worse than the current best solution, and the best solution is not a satisfying goal, then the search

algorithm is said to be trapped at a hillock or local extrema. One way to overcome this is by randomly selecting a new starting state, and then continuing the search process.

2.2.3 Heuristic search

Heuristic means rule of thumb. The general approach in heuristic search is to use one or more heuristic functions to determine the best candidate among a collection of legal states that can be generated from the current state. The difficult task in heuristic search, is selecting a satisfying heuristic function, as the heuristic function has to be intuitively selected or created.

2.2.4 Means and ends analysis

This search method attempts to reduce the gab between the current state and the goal state. One common way of doing this is by measuring the distance between the current state and the goal state, and then applying some operator that will reduce the gap. In mathematical theorem-proving, means and ends analysis is often used.

2.3 Fuzzy logic

Fuzzy logic [1], [16], [19] and [20], deals with the area of artificial intelligence known as uncertainty handling. The primary point in fuzzy logic is to define a way to deal with imprecision. The point is that in real life problems the answer to a question is only rarely of the form true or false. Fuzzy logic is an extension to traditional two valued logic.

Traditional two valued logic uses the binary pair {0, 1} to represent the two truth values false and true. It is however argued by Lotfi Zadeh, the father of fuzzy logic, that many real world sets of today are defined by non sharp boundaries. Example of such a set could be height. This is because height of an object often simply cannot be answered as either being high (true) or not high (false). Instead an object can be high to a degree. Fuzzy logic uses the whole interval between [0, 1]. This gives the ability to model a continuous change from false to truth and thereby giving someone ability to be somewhat high instead of either just high or not high.

2.3.1 Fuzzy sets

A set is a collection or class of objects that shares a property. A set is specified by its members (items or elements). An example could be the set of integers greater than 0 and smaller then 5 specified by A = {1, 2, 3, 4}.

In a fuzzy set each member has a membership degree between 0 and 1. The higher the number is, the higher the membership is. The elements of a fuzzy set are taken from a universe. The universe holds all the elements that come into consideration when determine the fuzzy sets members. Every element in the universe is member of the fuzzy set to some degree. This membership degree could be zero. The set of elements that have a non-zero membership is called the support. The function that ties each element from a universe to a fuzzy set is called a membership function and is often denoted by the letter μ. Figure 1 is an illustration of a membership function that defines to what degree someone is tall.

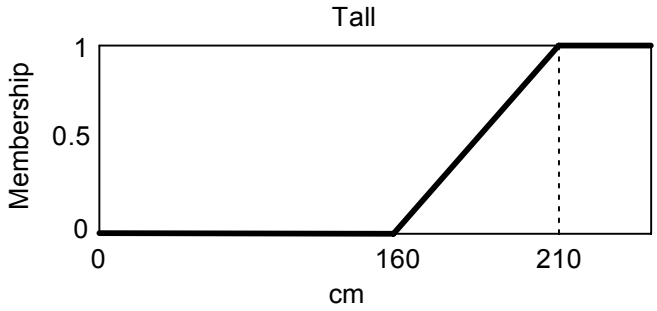


Figure 1: Illustration showing to what degree a person is tall

From Figure 1 it can be seen that if a person is below 160 cm that person is not tall at all, if the person instead is above 210 cm, the corresponding person is fully tall. The interesting part is between the intervals 160 and 210 cm where the degree to which someone is tall is a continuous change from 0 (false) to 1 (true). A membership function can also be written mathematically as below:

$$\mu_{Tall}(x) = \begin{cases} 0 \quad for \quad x \leq 160 \\ 1 \quad for \quad x \geq 210 \\ \frac{1}{50} \cdot x - 3.2 \quad otherwise \end{cases}$$

Let X be a nonempty universe. The fuzzy subset A in X is characterized by its membership function:

$$\mu_A: X \rightarrow [0, 1]$$

and $\mu_A(x)$ is the element x's degree of membership in A. This is illustrated in Figure 2.

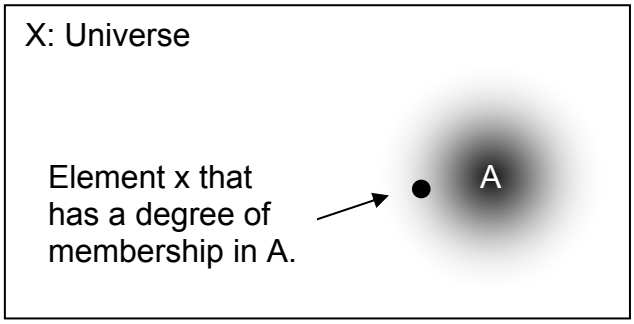


Figure 2: Element x is member of the fuzzy subset A in universe X

A fuzzy set can simply be viewed as a collection of ordered pairs:

$$A = \{(x, \mu(x))\}$$

Item x is a member of A and μ(x) is its membership. If A is a finite fuzzy set then an often used notation is:

$$A = \mu_1/x_1 + ... + \mu_n/x_n$$

to illustrate each member and its corresponding membership grade. A single pair (x, μ(x)) is called a singleton.

2.3.2 Operations on fuzzy sets

In traditional Boolean logic there are the three classical operators union (or), intersection (and), and the not operator. These operators exists in fuzzy logic too, but are defined differently.

- A or B = Max($\mu_A(x)$, $\mu_B(x)$)
- A and B = Min($\mu_A(x)$, $\mu_B(x)$)
- not A = 1 - $\mu_A(x)$

Fuzzy set operations creates a new fuzzy set from one or several existing fuzzy sets. Besides these classical operators a whole range of other operators exists. Fuzzy set operations can be divided into three groups. That is t-norms, averaging operators and t-conorms. Figure 3 illustrate how the three groups of operators are related.

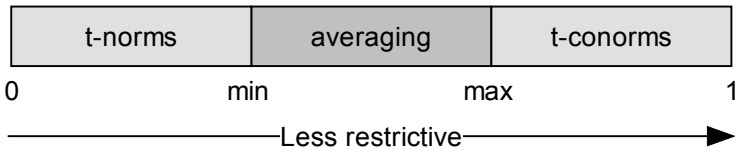


Figure 3: How the three operator types are related

The boundary between t-norms and averaging operators is defined by the classical *and* operator, also known as the *min* operator, or the intersection. The boundary between averaging operators and t-conorms is defined by the classical *or* operator, also known as the *max* operator, or the union. Figure 4 is an illustration of the three primitive set operations.

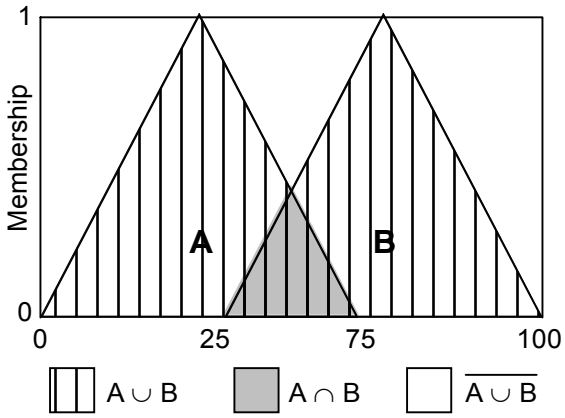


Figure 4: Illustration of the three primitive set operations

2.3.3 Fuzzy relations

A fuzzy relation is a way to describe relationship among objects. Graphically it can be illustrated as shown in Figure 5. From this illustration it can be seen that B resembles A to a degree of 0.8. Then if an operation involving B is requested, but no B exists, then doing the operation involving A might be a better solution than not do doing the operation at all.

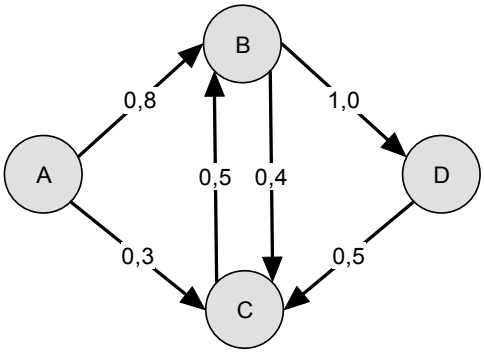


Figure 5: Graphical illustration of a fuzzy relation

To explain this model in relation to the human mind, and example could be a person asking another if he has a pick-up truck he can borrow, instead of just answering with a “no”, he could say “no but I have a trailer”. Here trailer is seen to be a pick-up truck to some degree and having some relation to pick-up truck in the given context, and using a trailer might be a reasonable solution.

Another way to illustrate a fuzzy relation is in matrix form as illustrated in Table 2.

↗	A	B	C	D
A	1	0,8	0,3	0
B	0	1	0,4	1
C	0	0,5	1	0
D	0	0	0,5	1

Table 2: Matrix representation a fuzzy relation

Although no explicit connection might exist between two objects there still might be a connection through other objects. In the above example it can be seen that D has a connection to B through C. One way to find these connections, and the strength of them, is by use of the max product closure.

2.3.4 Linguistic variables

To represent the measurements of Fuzzy logic, linguistic variables are used. A linguistic variable takes word or sentences as values. Each value is called a term set. Each value in a term set is a fuzzy variable that defines a fuzzy set. A fuzzy variable has been defined over a base variable. In short: linguistic variables → fuzzy variables → base variable.

Example of a linguistic variable could be speed. Values of this linguistic variable, which is a term set or fuzzy variable, could be slow, fast, very fast, quite slow or moderate. Each fuzzy variable is defined over a base variable that in this example could scale from 0 to 250 km/hour. Another example can be seen in Table 3.

Type	Example
Linguistic variable	Age
Term set / Fuzzy variable	young, very old, not so old
Base variable	0 to 100 years

Table 3: A linguistic variable and its fuzzy and base variable

A linguistic modifier is an operation that changes the meaning of a fuzzy variable. For example in the sentence “much larger than 0”, the word *much* modifies “larger than 0”. Although the exact effect of a modifier can be hard to define, they can often be approximated by operations as illustrated below. Where *a* is the membership function of a fuzzy variable.

$$\text{Very } a = a^2$$
$$\text{Extremely } a = a^3$$
$$\text{Slightly } a = a^{1/3}$$

2.3.5 Approximate reasoning

Approximate reasoning is the process to reach a fuzzy conclusion given some fuzzy input. In approximate reasoning we deal with what is called propositions; that is assigning graded truth values in the range from 0 (absolutely false) to 1 (completely true). A proposition can have the following form:

X is P

- X: The subject of the proposition
- P: The predicate

Example of a proposition could be:

Leaves are green

It can be seen the predicate is a fuzzy variable, as it can be argued that something can be green to a degree. Also the subject of a proposition can be a proposition. An example can be seen below:

(Leaves are green) is true

Propositions can be modified by use of linguistic modifiers. As a result the above proposition can be modified to something like:

All (leaves are green is not true)

Here *All* is called a quantifier and *not* is called a predicate modifier.

One of the most useful ways to do approximate reasoning is called the generalized modus ponens. This can be represented as:

$$\frac{\begin{array}{c}\text{If x is A then y is B} \\ \text{x is A}\end{array}}{\text{y is B}}$$

Which means that if it is given that *If x is A then y is B* and *x is A to some degree* then we can conclude that *y is B to some degree*. A more concrete example can be seen below:

If wind is blowing then window is closed
Wind is somewhat blowing

Window is somewhat closed

What exactly *somewhat* is defined as is problem dependent, but it could translate to something like the window should be open one third of what it is capable of.

2.3.6 Fuzzy possibility theory

There might be cases where the exact value for an element is not known, and as a consequence the elements exact membership degree to a fuzzy set is not known. Consider the membership function previously presented in Figure 1 and re-presented in Figure 6 for clarity.

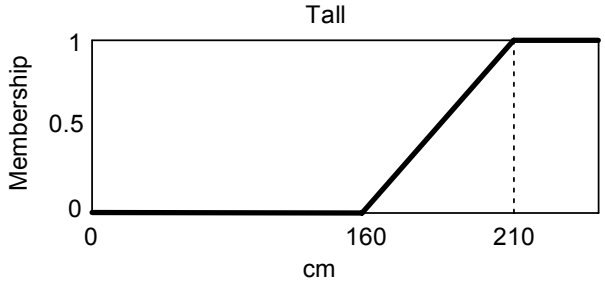


Figure 6: Illustration of the membership function to what degree someone is tall

And remember the corresponding membership function defined as:

$$\mu_{Tall}(x) = \begin{cases} 0 & for \quad x \le 160 \\ 1 & for \quad x \ge 210 \\ \frac{1}{50} \cdot x - 3.2 & otherwise \end{cases}$$

Consider a case where we only know that someone's height is between 190 and 200 cm (Lets call the person x). The possibility distribution can then be illustrated as seen in Figure 7.

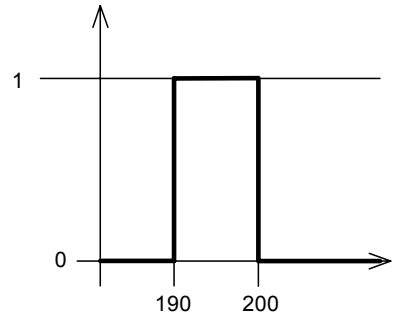


Figure 7: Illustration of possibility distribution for x's height

The answer to the question: "To what degree is the statement 'x is tall' true?" can then be answered by possibility and necessity defined as:

$$\Pi(A \mid B) = \sup_{x \in X} \min(\mu_A(x), \pi_B(x)) \qquad \text{(Possibility)}$$

$$N(A \mid B) = \inf_{x \in X} \max(\mu_A(x), 1 - \pi_B(x)) \qquad \text{(Necessity)}$$

- μ_A : Membership function
- π_B : possibility distribution

Possibility and necessity can in x's case be illustrated as in Figure 8:

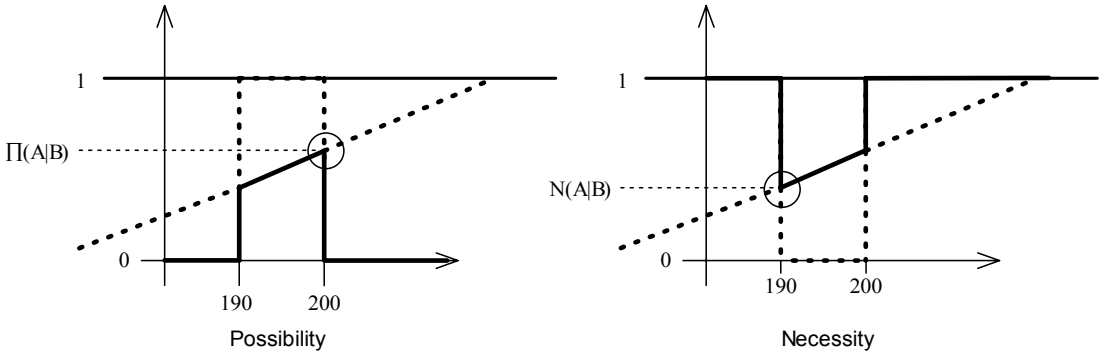


Figure 8: Possibility and necessity for x is tall

This can also be calculated as below:

$$N(Tall \mid [190,200]) = \frac{190}{50} - 3.2 = 0.6$$

$$\Pi(Høj \mid [190,200]) = \frac{200}{50} - 3.2 = 0.8$$

The result is that x is possibly tall to the degree 0.8 but necessarily tall to the degree 0.6.

2.4 Fuzzy controllers

Fuzzy controllers are described in [18]. Although fuzzy controllers often are associated with hardware control there is no reason why the principles cannot be applied to other domains. Fuzzy controllers are also sometimes called fuzzy expert systems. The term fuzzy controller is generally used when the fuzzy expert system are to control some hardware or electrical products such as washing machines, video cameras and rice cookers, or industrial equipment such as trains and industrial robots. While fuzzy expert system, is a more general term referring to a computer programs, that by use of fuzzy logic, is capable of solving problems by use of a knowledge base generally crafted by human experts.

The main idea of fuzzy controllers is to build a model of a human control expert who is able of controlling something on behalf of rules specified in form of linguistic variables. A fuzzy controller can include empirical rules. The collection of rules is called a rule base. The rules are in if-then format. The if-side is called the condition and the then-side is called the conclusion. Examples of typical fuzzy controller rules are given below.

1. If error is Neg and change in error is Neg then output is NB
2. If error is Neg and change in error is Zero then output is NM
3. …

Neg, *NB* and *NM* are linguistic terms and stands for Negative, Negative Big and Negative Medium. The controller is then able to execute the rules and compute a control signal depending of the inputs error and change in error. In a rule based controller, the control strategy is stored in a somewhat natural language. This makes the controller easier to intuitively understand and maintain for humans. In the next parts an overview of the structure of fuzzy controller is first given, finally a concrete example of how a fuzzy controller works is given.

2.4.1 Fuzzy controller structure

Figure 9 illustrates the different blocks in a typical fuzzy controller. The following section will explain the different blocks in the diagram.

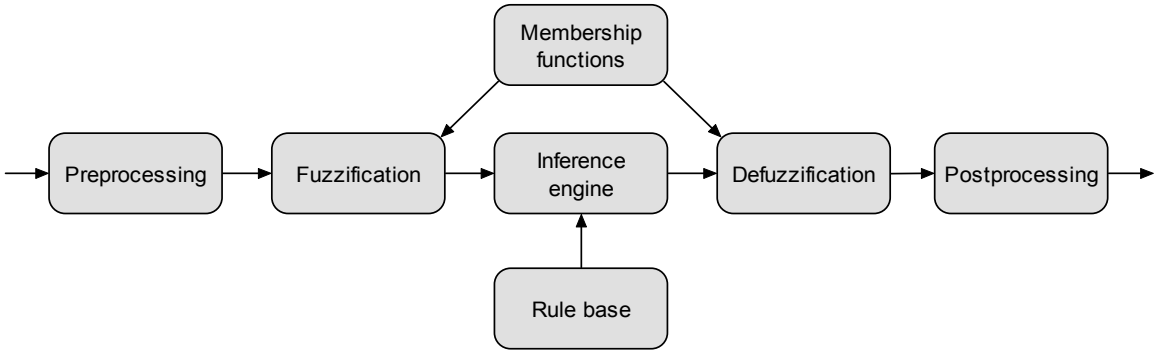


Figure 9: The different blocks of a fuzzy controller

Preprocessing

Inputs to a fuzzy controller are often hard or crisp data from measuring devices. A preprocessor makes some initial adoption of the data before it enters the actual fuzzy controller. Some examples of preprocessing could be:

- Rounding from floating points to integers.
- Normalization or scaling into a standard range.
- Filtering to remove noise.
- Averaging to obtain a tendency.
- Combining of measurements to obtain key indicators.

Membership functions

All input values to a fuzzy controller have associated linguistic terms. All these linguistic terms needs to have an associated membership function. These membership functions are used when calculating to what degree an input value fulfils the requirement for a linguistic term. This block holds all linguistic terms and their associated membership function.

Fuzzification

As the fuzzy controller takes input from the real world, this data has to be converted to fuzzy numbers. This process is called fuzzification. This is done by assigning a membership degree to each linguistic term for that input.

Rule Base

The rule base is where all rules for the fuzzy controller are stored. Basically rules are the fuzzy controller's linguistic terms which are connected in an if-then fashion, but thy can be presented in different ways. Rules are typically connected with the words *and*, *or*, *if-then*, *if and only if* and the modifier *not*.

Inference Engine

The inference engine is responsible for calculating the resulting fuzzy set. The fuzzy set is the possible output values and their calculated membership degree to the input. First each rules grade of fulfillment is calculated. This is done by aggregating each value in the condition. After each rules grade of fulfillment has been calculated, the fulfillment of each possible output can be found.

Defuzzification

The resulting fuzzy set must be converted to some kind of control signal that can be sent to the object there is to be controlled. This final control signal can be calculated in a number of different ways. Depending of the problem domain a suitable method can be selected.

Postprocessing

If necessary some final adoption of the result can be done in the postprocessing part.

2.4.2 Fuzzy controller example

Now consider a fuzzy controller who takes two input values distance and load and puts out one value speed. It could be some kind of device that was transporting gods between load and unloading points. The fuzzy controllers input and output variables along with their defined linguistic terms and membership functions can be seen in Figure 10.

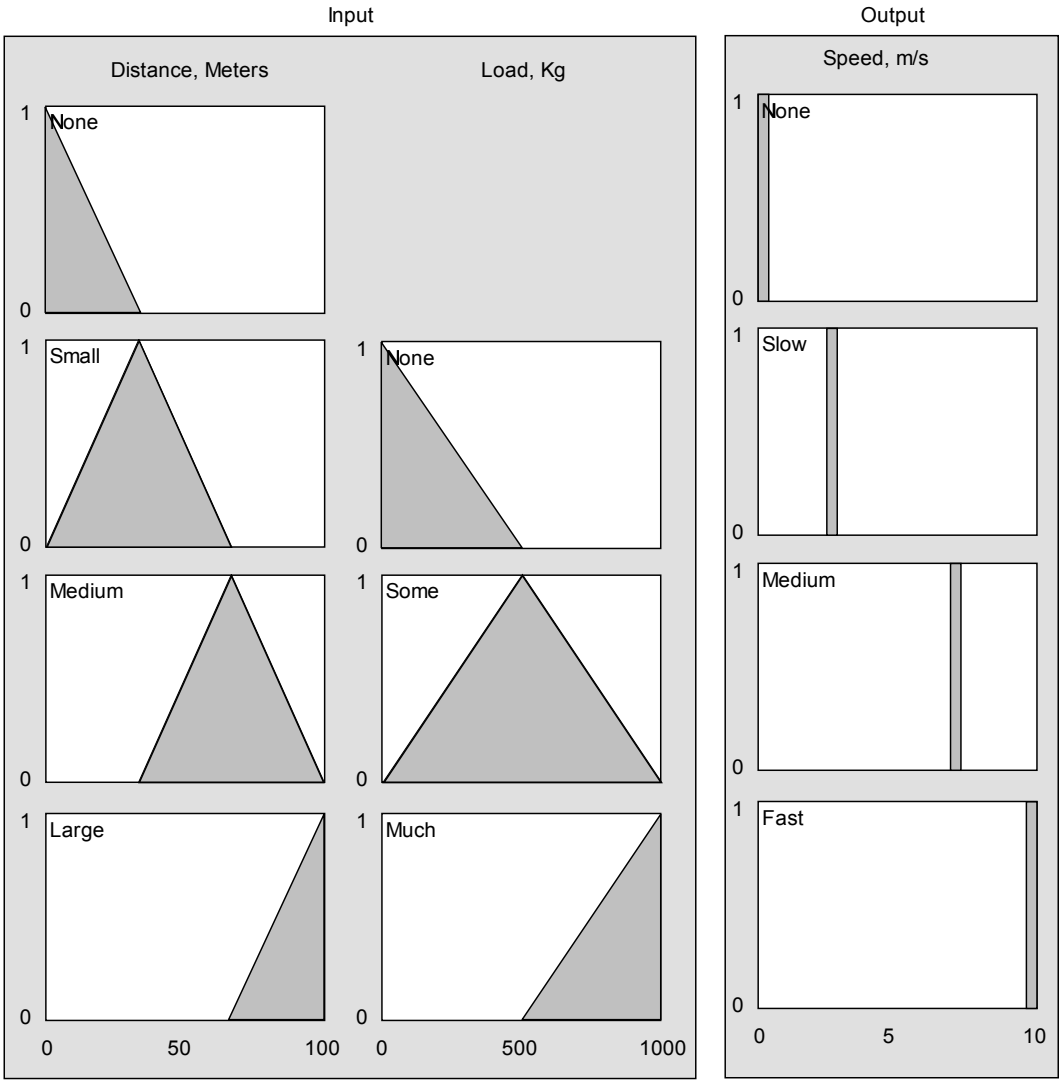


Figure 10: Membership functions for input variables Distance, Load and output variable Speed

The defined rule base could look something like:

1. If Distance is None and Load is None or Some or Much then output None
2. If Distance is Small and Load is None or Some or Much then output Slow
3. If Distance is Medium and Load is None or Some output Medium
4. If Distance is Medium and Load is Much output Slow
5. If Distance is Large and Load is None or Some output Fast
6. If Distance is Large and Load is Much output Medium

For a better overview and to check for completeness of rules such the controller will not get into undefined states, a matrix representation of the rule base can be seen in Table 4.

		Distance			
		None	Small	Medium	Large
Load	None	*None*	*Slow*	*Medium*	*Fast*
	Some	*None*	*Slow*	*Medium*	*Fast*
	Much	*None*	*Slow*	*Slow*	*Medium*

Table 4: Fuzzy controller rule base in matrix form

Now consider an example where this controller receives the following input.

Input	
Distance	50 meters
Load	250 kg

Table 5: Fuzzy controller input

The goal is then to decide the control signal, which is the speed of the device this controller is responsible for.

Preprocessing

First the input values enter the preprocesser which makes some initial filtering or adoption of the data. In this example it could simply be rounding from floats to integers or a change of the input value distance to 100 meters in the event that it was above that value. In this example we will however not make any preprocessing of the input.

Fuzzification

In the fuzzification block the inputs membership degrees to each of the linguistic terms has to be calculated. This is done by a lookup in the membership functions that can be seen in Figure 10. The membership to each linguistic term can be seen in Table 6.

Distance	
Linguistic term	Membership
None	*0*
Small	*0.5*
Medium	*0.5*
Large	*0*

Load	
Linguistic term	Membership
None	*0.5*
Some	*0.5*
Much	*0*

Table 6: The inputs membership to each associated linguistic term

Inference engine

In the inference engine the fulfillment for each rule in the rule base is calculated. Each rules grade of fulfillment can be seen in Table 7.

Rule	Fulfillment
If Distance is None and Load is None or Some or Much then output None	0
If Distance is Small and Load is None or Some or Much then output Slow	0.5
If Distance is Medium and Load is None or Some output Medium	0.5
If Distance is Medium and Load is Much output Slow	0
If Distance is Large and Load is None or Some output Fast	0
If Distance is Large and Load is Much output Medium	0

Table 7: Each rules grade of fulfillment

As *and* operator the fuzzy *min* aggregation is used and as *or* operator the fuzzy *max* aggregation is used. The resulting fuzzy set can be seen below:

Speed = 0/None + 0.5/Slow + 0.5/Medium + 0/Fast

Defuzzification

Then the resulting fuzzy set has to be defuzzified into a single valued control signal. There is no defined way to do this, but a popular method is centre of gravity defined as:

$$u = \frac{\sum \mu(x_i)\ x_i}{\sum \mu(x_i)}$$

Here x is the actual value of the linguistic terms in the output set. These values are illustrated in Figure 10 and summarized in Table 8 for clarity.

Linguistic term	Value
None	0 m/s
Slow	3 m/s
Medium	7 m/s
Fast	10 m/s

Table 8: Actual values of output linguistic terms

The calculation of the control signal is then as follows:

$$\frac{(0\cdot 0)+(0.5\cdot 3)+(0.5\cdot 7)+(0\cdot 10)}{0+0.5+0.5+0} = 5$$

The control signal would be the value 5. This would be the calculated speed the object to control is to be assigned.

Postprocessing

If necessary a final adoption of the output could be done. This is however ignored in this example.

2.5 Genetic algorithms

Genetic algorithms [22] and [23], deals with the area of artificial intelligence known as machine learning. In short genetic algorithms use the principles of Charles Darwin's theory of evolution to search for solutions to a problem. A problem is encoded such that it can be represented as a vector. This vector is called a chromosome. To start with a population of random chromosomes is created and tested up against the given problem. Depending on how well a given chromosome solves this problem, a fitness grade is assigned to the chromosome. Then a new population of chromosomes is generated by "mating" the chromosomes in the current population with each other. The process of mating is not completely random as the fittest chromosomes have a greater change of mating then the not so fit ones. If everything goes well the next generation of solutions should become fitter. This evolutionary process then continues over many generations until a satisfying solution is bred.

2.5.1 Steps in genetic algorithms

Before a genetic algorithm can be used, a way to encode the problem must be found. This encoding could for example be a vector of real numbers or integers. It could also be a string, or a more typical case, a binary string. The way to encode the problem at hand is probably the second hardest part of the genetic algorithm. The encoding of a problem is called a chromosome and may in the binary case for example look like:

10001001110010010

To start with a population of random chromosomes is created. Each of these chromosomes represents a solution to the problem. Then the following steps are repeated until a satisfying solution is found:

1. Calculate the fitness score for each chromosome in the current population.
2. Chose two parent chromosomes from the current population using fitness proportional selection.
3. Depending on crossover rate, crossover the parent chromosomes at a random point to generate their offspring.
4. Depending on the mutation rate, mutate each bit in the offspring chromosome.
5. Go to step 2 until population is complete.
6. Replace current population with the new population.
7. Go to step 1 until a satisfying solution is found.

The fitness score

The fitness score is a score assigned to each chromosome that indicates how well this chromosome is at solving the problem at hand. Deciding on an effective fitness algorithm is probably the hardest part when implementing a genetic algorithm. An example of a straightforward fitness algorithm, if the problem was the traveling sales person, is to calculate the distance. The lower the distance, the more fit the solution is.

Chromosome selection

The fitness proportional selection is a way to select members from the population in a way such that chromosomes with the best fitness value have a better chance of being selected. It does not guarantee that the fittest members are selected, just that their chances of being selected are rather good. Imagine a population of five chromosomes where the fitness for each chromosome, is a percentage of the total fitness in the population. This could be the population shown in Table 9. Then each chromosome changes of selection are the same as its fitness value.

Chromosome	Fitness
1	8.8%
2	30.8%
3	28.4%
4	5.6%
5	26.4%

Table 9: Population of five chromosomes and their fitness

This gives chromosome 2, 3 and 5 a rather good change for selection. While the not so fit chromosome 1 and 4 have a much smaller change of evolving to the next generations.

Crossover

Next step after parent chromosomes have been selected is to produce their offspring. The most widely used method for this is called single point crossover. This is done by swapping the two parent chromosomes bit at a random chosen location. This is probably best explained by looking at Figure 11.

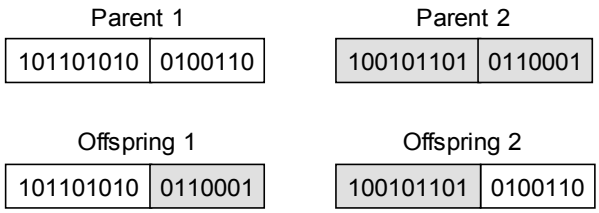


Figure 11: Crossover of two chromosomes and their offspring

Sometimes crossover does not occur. In this case the parent nodes are copied directly to the new population. The chance for a crossover to occur is usually around 70%.

Mutation

After chromosome selection and crossover, a new population of chromosomes exists. Some of them are exact copy of chromosomes from the previous population and some of them are produced by crossover. To ensure that not all chromosomes are the same, there is a small chance that a bit will mutate. Depending on how the chromosomes are encoded mutation can either happen by changing a value, or by flipping a bit. The chance for mutation is typically 1 tenths of a percent. Mutation is a fairly simple process of simply stepping through each chromosome; each bit in a chromosome then has a small chance of changing value.

2.6 Artificial neural nets

Artificial neural nets (ANN) [1], [21], [24] and [26] is a way of trying to simulate a biological brain electronically in software or hardware. Artificial neural nets are also often just called neural nets, although artificial neural nets are a more correct term, as a neural net is a biological term refereeing to the network within brains. Artificial neural nets are mainly designed to work with pattern classification. The network can take a vector of numbers and then classify that vector and thereby give a desired output.

A brain is made up of billions of tiny cells called neurons. Each neuron is connected to other neurons and communicates with them through connections made up of what is called synapses and dendrites. A neuron continuously receives signals from other neurons through these inputs. It then performs some operation on the input to find a value called the activation. If this activation value is higher than a given threshold the neuron outputs a signal. This is typically called a step function and an example can be seen in Figure 12.

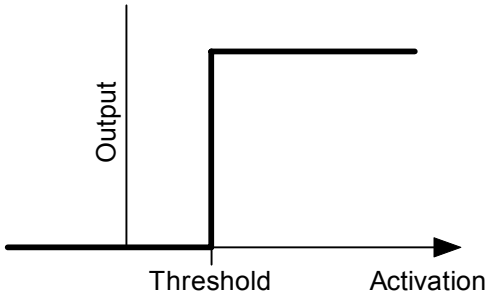


Figure 12: A step function that outputs a signal when the activation is above a given threshold

This output is then forwarded to other neurons that will receive this output signal as their input.

2.6.1 The artificial neuron

Artificial neural nets are made up of a collection of artificial neurons. There is no specific rule for the amount of neurons used in a neural net. Depending on the task, it can differ between as few as three up to millions. Figure 13 is an illustration of an artificial neuron.

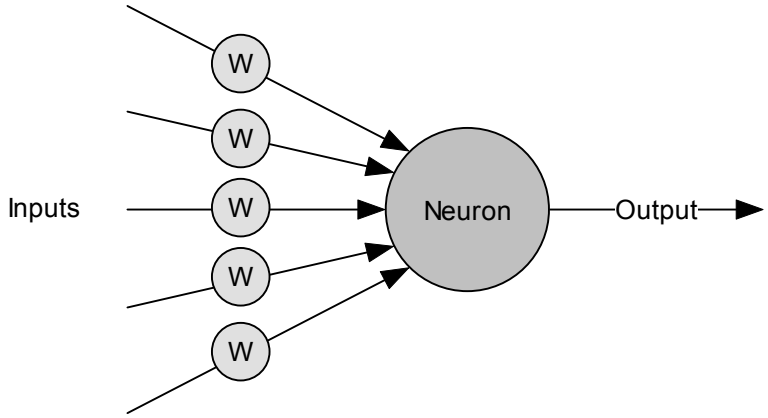


Figure 13: Artificial neuron with five inputs. Each input has an associated weight W

An artificial neuron can have any amount of inputs. Each input has an associated weight which is simply a floating point value. In most cases this weight can be both positive and negative. It is these weights that are adjusted, as a neural network learns. The inputs may be represented as x_1, x_2, x_3, …, x_n and the related weights for each input as w_1, w_2, w_3, …, w_n. The activation value for a neuron can then be found by:

$$activation = \sum_{i=0}^{i=n} w_i x_i$$

However in an artificial neuron the activation value is not put through a step function. This would limit the output to a binary output. Instead of using a simple step function, a function able to soften the output is used. One of the most popular is the sigmoid function defined as:

$$output = \frac{1}{1 + e^{-a/\rho}}$$

- Where *e* is a constant approximated to 2.71183.
- a is the activation into the neuron.
- And *p* is a value controlling the shape of the function typically set to 1.0.

This function will produce and output of the form seen in Figure 14.

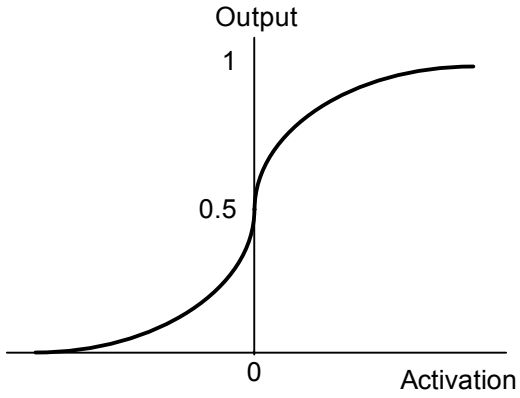


Figure 14: Output of a sigmoid function

Therefore to get a graded output greater then 0 and smaller then 1 from an artificial neuron, the activation value just has to be run through this sigmoid function.

2.6.2 Artificial neural net architecture

To create an artificial neural net, a collection of neurons, also called nodes or units, has to be connected somehow. There are different types of networks; they do all however have a standard network architecture consisting of several layers, an input layer, hidden layers, and an output layer. The input layer takes the input and sends to the hidden layers. The hidden layers then do the necessary computation before sent to the output layer, which outputs the data to the user.

The most popular architecture is called a feed-forward network. An example of a feed-forward network can be seen in Figure 15. This network gets its name because each neuron feed its output forward to the next layer until a final output is obtained.

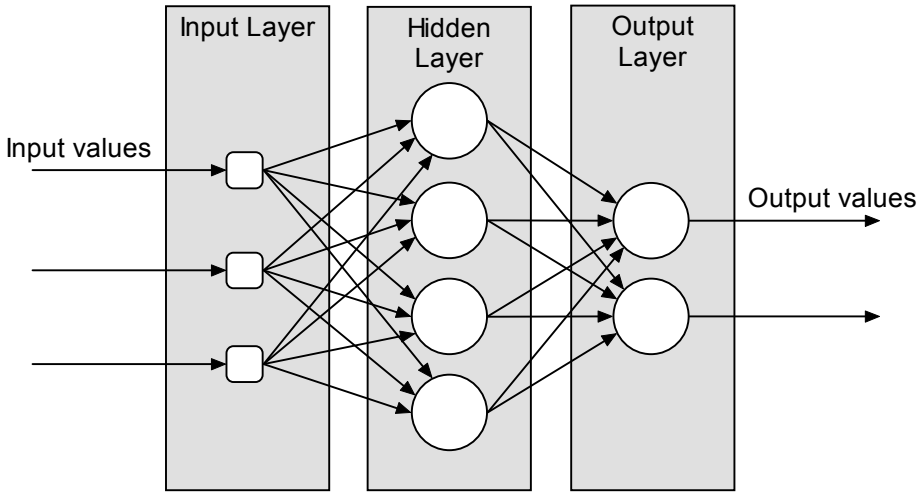


Figure 15: Example of a feed-forward network

Each input is sent to every neuron in the hidden layer and each hidden layer's neuron is connected to every neuron in the next layer. There can be any number of inputs to, and outputs from a feed-forward neural net. Also the number of neurons in each layer, as well as the number of hidden layers can be adjusted as needed, although one hidden layer typically suffices for most problems.

2.6.3 Training and memory of artificial neural nets

At its initial state a neural network is not very interesting as it has not been trained to solve any specific problem. As a result if you feed an artificial neural net with some input to start with, the output will in the beginning not be satisfying. That is because the weights associated with each neurons input are initially in a random state. All the weights associated with each connection to a neuron, therefore needs to be adjusted, such that the final net should be able to solve a specific problem. There are basically four ways an artificial neural net can get into a state where it starts to perform as desired.

- Supervised training
- Unsupervised training
- Reinforced learning
- Evolution

Supervised training is the most popular methods for training neural networks. In supervised training an input and its associated desired output is provided. The network then processes the input, and compares its output against the provided and desired output. The connection weights within the network are then adjusted accordingly. The entire set of training data is shown to the network many times and the weights are adjusted each time, until the artificial neural net behaves as desired.

One of the most popular algorithms for training neural networks are known as the back-propagation algorithm and explained in section 2.6.4.

In unsupervised training the network is provided with input but no desired output. The system itself must then decide on some features it will use to group the input data. In the unsupervised learning process the network will attempt to generate a unique set of output for one particular type of input patterns.

Reinforced learning uses success, failure, reward, and punishment as indicators of the result. In reinforced learning the network does some action on the environment and gets some feedback from the environment. The learning system is then responsible for grading the networks action as either good or bad. Based on this response the network adjusts its weights.

In the evolutionary approach genetic algorithms explained in section 2.5 is used to evolve the weights within the network. It is a requirement that the neural networks live in an environment with other networks. A fitness score is then calculated for each network depending on how well it is at solving the problem at hand. This fitness score is then used in the genetic algorithm when all the neural networks evolve to the next generation.

2.6.4 Back-propagation training algorithm

The probably most popular methods for training neural networks is the back-propagation algorithm, also known as the generalized delta-rule. It is a supervised training algorithm and therefore requires that both the input and its desired output are provided. The most significant strength of this algorithm is its ability to train multilayered feed-forward networks. Before the invention of this algorithm training was only possible for single layered feed-forward neural nets.

The adjustment of the input weights for each neuron is as follow:

$$\Delta w_i = \eta \cdot input_i \cdot \delta$$

- i : Index of neuron input
- w_i : Array of input weighs to neuron
- η : Learning rate (constant)
- $input_i$: Array of input values to neuron
- δ : Error term for neuron (see below)

If neuron is in output layer, its error term is calculated by:

$$\delta = output\ (1 - output)\ (target - output)$$

- output : Output from the neuron
- target : Desired output from the neuron

If neuron is in hidden layer, its error term is calculated by:

$$\delta = \text{output}\ (1 - \text{output})\ (\Sigma\ \delta_k \cdot w_{ik})$$

- k : Designates a successor neuron
- δ_k : Error term for successor neurons
- w_{ik} : Array of input weights from current neuron to successor neurons

For each set of training data do:

1. Feed the network with input data to obtain its output.
2. Compute δ for each of the neurons in the output layer.
3. Compute δ for each of the neurons in the hidden layer(s) using δ from the next layer.
4. Calculate Δw for each neuron input and adjust weights accordingly.
5. Repeat step 1 until error at output layer is within desired margin.

2.7 Bayesian networks

A Bayesian network (BN) [4] and [6] is a graphical model that can be used for probabilistic reasoning that in relation to AI can aid the process of decision making. BN encodes probabilistic relationships among variables and are also sometimes called belief networks. More formally a Bayesian network is a directed acyclic graph with the following characteristics:

- A set of random variables makes up the nodes of the network.
- A set of directed links represented by arrows, representing normally causal relationships, connecting the nodes.
- Each node has a conditional probability table that quantifies its effect.

Consider an example where we have a burglar alarm. This alarm can be triggered either, by a burglary or an earthquake. If the alarm is triggered, your neighbor John has promised to call you. This can be illustrated as in Figure 16.

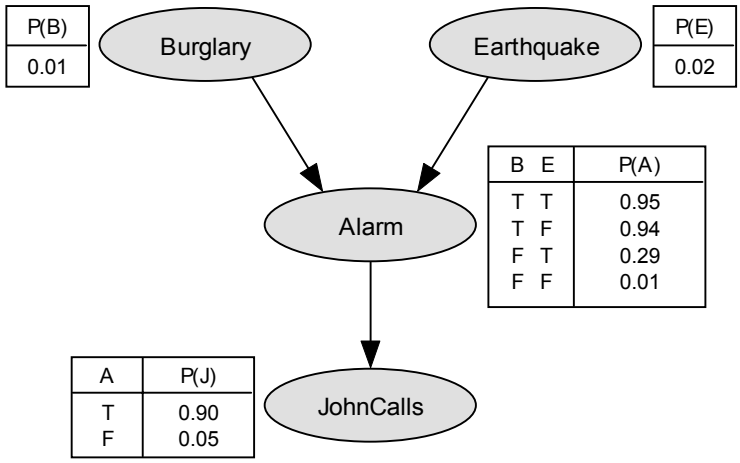


Figure 16: Illustration of a Bayesian network

Arrows are drawn from influencing nodes to influenced nodes. Influencing nodes are called parents of the influenced nodes. Influenced nodes are called children of the influencing nodes. The tables next to the nodes are called the conditional probability tables and represent the probability for the event represented by that node to happen. If the node has no parents it is called a root node and the conditional probability table simply represents the probability for that event to happen on its own. If the node is children to some nodes, the conditional probability table holds the probability for that event to happen for each possible combination of outcome of its parents.

In the above example it can for example be seen that the probability for a burglary is estimated to 1%. If there is a burglary but no earthquake the probability for the alarm to get activated is 94%, while the chance for the alarm to get activated on its own is 1%. Should the alarm be activated there is 90% chance John calls. Although not explicitly illustrated, it can also be seen that the probability for John not to call, even though the alarm is sounding, is 10%.

2.7.1 Bayes theorem

Before continuing on Bayesian networks, let's look at some basic probability theory involving Bayes theorem. This is a rather simple but powerful equation for probabilistic calculations. Bayes theorem is as follow:

$$P(H \mid E) = \frac{P(E \mid H)\ P(H)}{P(E)}$$

- *P(H | E):* Given E, probability for H to happen
- *P(E | H):* Given H, probability for E to happen
- *P(H):* probability of the H to happen
- *P(E):* probability of the E to happen

P(E) can be calculated by the formula:

$$P(E) = P(E \mid H)\ P(H) + P(E \mid \neg H)\ P(1 - P(H))$$

- *P(E | ¬H)* is the probability for E to happen given H has not happened (¬H is called the false alarm rate)

Consider an example where someone is considering investing some money into a new company. The person knows that only 20% of all start-up companies succeed. The uncertainty can be reduced by asking for an expert opinion. A known fact about the expert is that of the companies that eventually succeeds, 40% is judged good prospects, 40% is judged moderate, and 20% is judged poor. It is also known that of all start-up companies that fails, 10% is judged to be good prospects, 30% to be moderate prospects, and 60% to be poor prospects.

Let's say we have the evidence that the expert judges the company to be good. We then wish to find the probability for the company to succeed. First the probability for the evidence good (*P(E)* in Bayes theorem) needs to be calculated, this is done as:

$$P(good = T) = 0.4 \cdot 0.2 + 0.1 \cdot (1 - 0.2) = 0.16$$

This can also be considered the probability that the expert will judge any new random company good.

We can now find the probability for the company to succeed using Bayes theorem as:

$$P(success = T \mid good = T) = \frac{0.4 \cdot 0.2}{0.16} = 0.5$$

So it is given that 20% of all start up companies succeeds. Our expert judge the company as good, 40% of all companies judged good succeed, and 10% fails. The probability for the company to succeed is then 50%.

The above example can be illustrated by the Bayesian network seen in Figure 17.

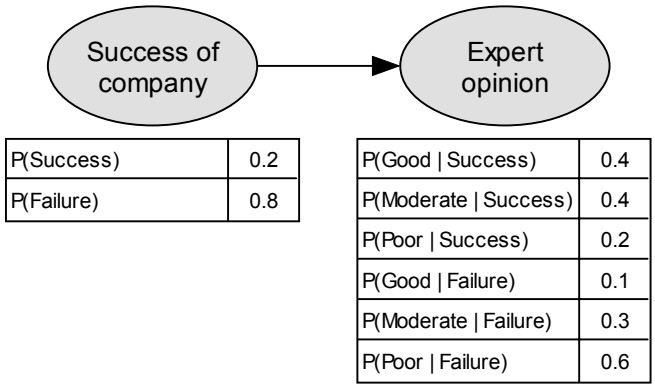


Figure 17: A simple Bayesian network

2.7.2 Joint probability distribution

The joint probability distribution (JPD) is the probability for a combination of specific assignments to each variable in the Bayesian network and it is defined as:

$$P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x \mid Parents(X_i))$$

Returning to Figure 16 it is for example possible to find the probability that John will call because the alarm has sounded, but neither a burglary nor an earthquake has occurred. The calculation can be seen below:

P(¬Burglary, ¬Earthquake, Alarm, JohnCalls)
= P(¬Burglary)P(¬Earthquake)P(Alarm | ¬Burglary, ¬Earthquake)P(JohnCalls | Alarm)
= 0.99 * 0.98 * 0.01 * 0.90 = 0.0087318

The Joint probability distribution for all combinations in the Bayesian net illustrated in Figure 16 can be seen in Table 10.

Row	Burglary	Earthquake	Alarm	JohnCalls	JPD
1	True	True	True	True	0.000171
2	True	True	False	True	0.0000005
3	True	False	True	True	0.0082908
4	True	False	False	True	0.0000294
5	False	True	True	True	0.0051678
6	False	True	False	True	0.0007029
7	False	False	True	True	0.0087318
8	False	False	False	True	0.0480249
9	True	True	True	False	0.000019
10	True	True	False	False	0.0000095
11	True	False	True	False	0.0009212
12	True	False	False	False	0.0005586
13	False	True	True	False	0.0005742
14	False	True	False	False	0.0133551
15	False	False	True	False	0.00009702
16	False	False	False	False	0.92076831

Table 10: Joint probability distributions for all combination in Figure 16

2.7.3 Inference

Inference is the computational method for deriving answers to queries in a Bayesian network. An example could be we simply want to know the probability that John will call *P(JohnCall)*. This query can be answered by taking the sum of all the rows where JohnCalls is true. We sum row 1-8 and obtain:

$$0.000171 + ... + 0.0480249 = 0.071119$$

This can be illustrated by the following equation (Only the first letter of a variable name is used. As an example does the letter J detonates the variable JohnCalls):

$$P(J = True) = \sum_{Burglary, Alarm, Earthquake} P(B) \cdot P(E) \cdot P(A \mid B, E) \cdot P(J = True \mid A)$$

Two other common types of inference are:

- **Casual inference** from causes to effects. Given a Burglary what is the probability that John calls *P(JohnCalls* | *Burglary)*.

- **Diagnostic inference** from effects to causes. Given John Calls what is the probability for a burglary *P(Burglary* | *JohnCalls)*.

Casual inference example

Using the joint probability distribution in Table 10, it is possible to answer the query *P(JohnCalls* | *Burglary)* which is the probability that John will actually call when a burglary is being made. The computation can be written as:

$$P(J = True \mid B = True) = \frac{\sum_{Alarm,\, Earthquake} P(B = True) \cdot P(E) \cdot P(A \mid B = True, E) \cdot P(J = True \mid A)}{\sum_{JohnCalls,\, Alarm,\, Earthquake} P(B = True) \cdot P(E) \cdot P(A \mid B = True, E) \cdot P(J \mid A)}$$

That is the sum of the rows in Table 10 where Burglary and JohnCalls is true divided by the rows where Burglary is true. This is the sum of row 1-4 divided by the sum of row 1-4 and 9-12. The result is:

$$\frac{0.000171 + ... + 0.0000294}{0.000171 + ... + 0.0000294 + 0.000019 + ... + 0.0005586} = 0.84917$$

So there is around 85% chances that john will actually call when a burglary is being committed.

Diagnostic inference example

Diagnostic inference can be calculated in the same way as above. It is possible to answer the query *P(Burglary | JohnCalls)* which is the probability for a burglary is being done given that John calls. The computation can be written as:

$$P(B = True \mid J = True) = \frac{\sum_{Alarm,Earthquake} P(B = True) \cdot P(E) \cdot P(A \mid B = True, E) \cdot P(J = True \mid A)}{\sum_{Burglary,Alarm,Earthquake} P(B) \cdot P(E) \cdot P(A \mid B, E) \cdot P(J = True \mid A)}$$

Which is the sum of the cells where Burglary and JohnCalls is true, divided by the cells where JohnCalls is true. This is the sum of row 1-4 divided by the sum of row 1-8. The result is:

$$\frac{0.000171 + ... + 0.0000294}{0.000171 + ... + 0.0480249} = 0.119401$$

So receiving a phone call from john, the probability for a burglary is just slightly below 12%.

2.7.4 Inference by set-factoring

As it can be seen in the above examples the calculation of a joint probability table quickly becomes a cumbersome task as the size of a Bayesian network increases. If another node with two possible values for example were added to the network, the size of the JPD table would increase to twice its size and contain 32 values. The added node could be another neighbor that also has promised to call if the alarm was sounding. Unfortunately the kind of inference presented above has been proved to be NP-hard and as a consequence other methods for calculation are needed if inference is to be done on Bayesian networks of reasonable size. A whole range of exact and approximate inference algorithms exists in the literature. Here a heuristic algorithm that performs exact inference known as set-factoring [6] will be presented.

Set factoring works by taking pairs of nodes in the Bayesian network and combining them. It begins with an initial set of node combinations, selects a pair to combine, removes the pair from the set and places the result back in the set. The process

continues until there is only one node left that holds the answer to the query. The set to combine in each step is the one with the smallest result table. If there is a tie the pair to combine is the one with the fewest parameters represented in the query and other candidate pairs.

Let us consider the example where we want to know the probability for John to call.

In the first step we have four nodes, which give six candidate pairs. Three of them share at least one parameter. The three pairs that share parameters are:

1. P(Earthquake) * P(Alarm | Burglary, Earthquake) => P'(Alarm | Burglary)
2. P(Burglary) * P(Alarm | Burglary, Earthquake) => P'(Alarm | Earthquake)
3. P(Alarm | Burglary, Earthquake) * P(JohnCalls | Alarm) => P'(JohnCalls | Burglary, Earthquake)

Which one of the candidate pairs 1 or 2 is combined, does not matter. Below we combine candidate pair 1 to obtain the new conditional probability table for *P'(Alarm | Burglary)*, calculations are included for clarity:

Burglary	**P(Alarm)**
True	Earthquake = True: (0.02 * 0.95) + Earthquake = False: (0.98 * 0.94) = 0.9402
False	Earthquake = True: (0.02 * 0.29) + Earthquake = False: (0.98 * 0.01) = 0.0156

Table 11: Calculation of conditional probability table for alarm when eliminating earthquake

The Earthquake node has been eliminated, and we instead have an alarm node, where in the case of a burglary, the chance that the alarm will be activated is 94.02%, and a 1.56% chance that the alarm will be activated when there is no burglary.

There are three nodes left in the Bayesian network, which give three candidate pairs. Two of them share at lest one parameter. The two are:

1. P(Burglary) * P'(Alarm | Burglary) => P'(Alarm)
2. P'(Alarm | Burglary) * P(JohnCalls | Alarm) => P'(JohnCalls | Burglary)

Again pair 1 is combined to obtain *P'(Alarm)*:

P(Alarm)
Burglary = True: (0.01 * 0.9402) + Burglary = False: (0.99 * 0.0156) = 0.024846

Table 12: Calculation of conditional probability table for alarm when eliminating burglary

The burglary node has been eliminated and a new alarm node where the probability for the alarm to be activated is 24.846%, is replacing the old alarm node.

There are now two nodes left in the network which give the pair to combine as seen below:

1. P'(Alarm) * P(JohnCalls | Alarm) => P'(JohnCalls)

The probability that John will call can be obtained by combining the last two nodes to obtain the conditional probability table for simply for john to call:

P(JohnCalls)				
Alarm = True		Alarm = False		
(0.024846 * 0.9)	+	(0.975154 * 0.05)	=	0.0711191

Table 13: Final calculation of probability for John to call

This result matches the result obtained in section 2.7.3 for the exact same query but the amount of calculations is far fewer as the calculation of a conditional probability table is skipped. The whole process is illustrated in Figure 18.

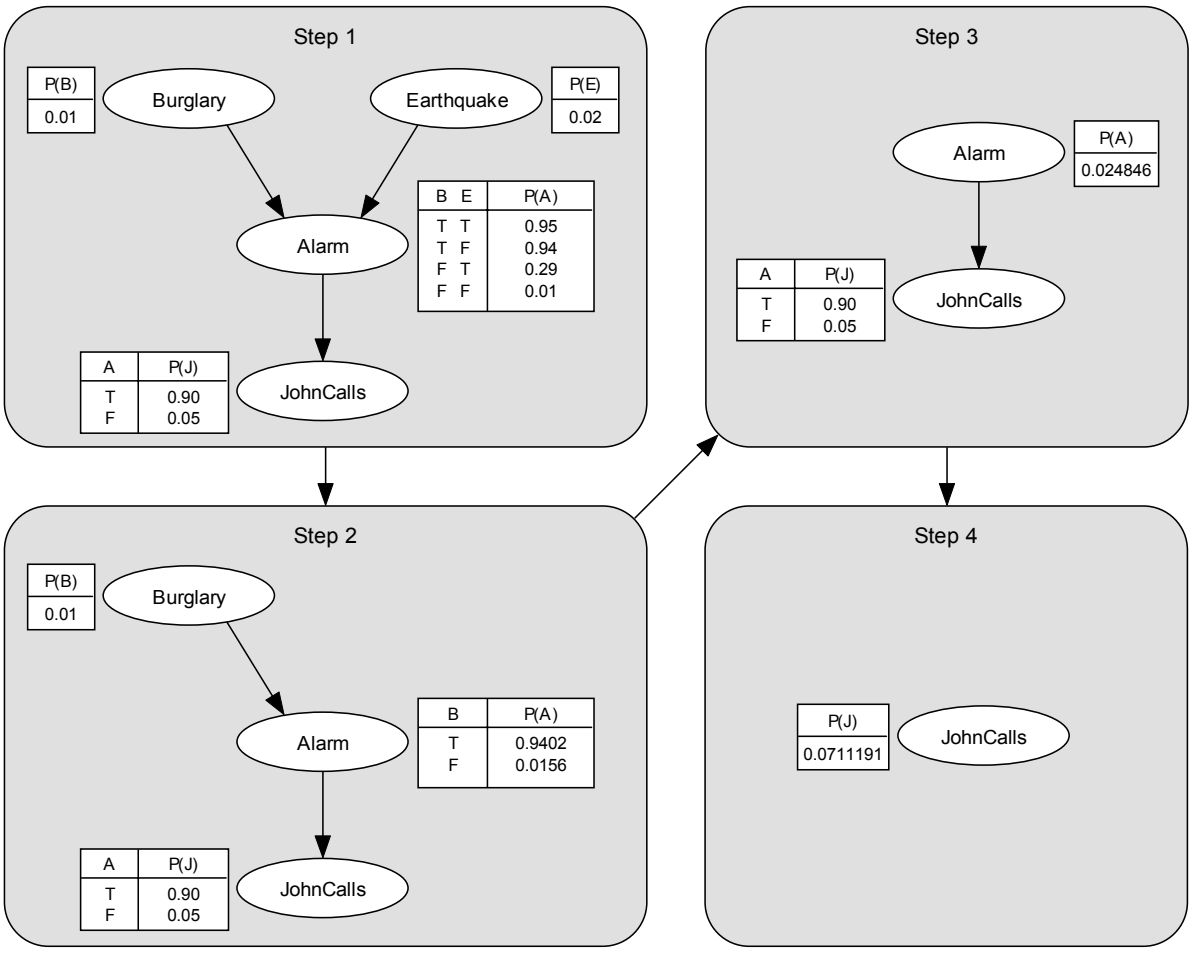


Figure 18: Illustrated example of set-factoring

2.8 Conclusion on artificial intelligence

Artificial intelligence is not about inventing the brain. It does not look like the creation of a human mind in software is a serious research area. That could be because people still see it as impossible and accept there is something about the human mind that with our understanding currently cannot be duplicated, something that perhaps can be called a soul.

Artificial intelligence can be viewed as a way to solve problems that do not have a straight forward solution. AI can also in many ways be viewed as a way that seeks to code human knowledge into software. The success seems to depend a great deal on the programmer obtaining (or helping the program to obtain) expert information, and making it correctly available to the program.

The methods used in AI spans over a range of different techniques, but they basically seem to be in at least one of two main categories. That is the category of learning or the category of supporting decision making in either a more human like or rational way.

3. Game theory

This chapter will give an overview of basic game theory. Much theory of games concerns itself with what is known as the Nash equilibrium defined as a situation in which each player makes its best response (maximizes its score), given the actions of rival players. The section will also try to define what games actually are and how games theoretically can be solved. It should also give an overview of why most games cannot be solved in practice.

The following sections are contained in this chapter:

- Classification of games
- Game trees
- Agents
- Conclusion on game theory

3.1 Classification of games

Before classifying games, it is necessary to define what a game is. In [25] the following definition is found:

> A game is defined as a decision problem with two or more decision makers – players – where the outcome for each player may depend on the decisions made by all players. Each player evaluates possible outcomes in terms of his own utility function, and works to maximize his own expected utility only.

At a high level, games can be divided into three main categories as illustrated in Figure 19. This is however only the main categories as hybrid games also exist. Football can be considered a game requiring some of all elements.

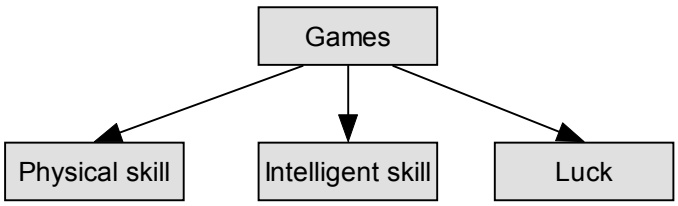


Figure 19: Main classification of games

Games interesting for artificial intelligence can mostly be categorized in the intelligent games category. These types of games are normally subdivided along two dimensions of stochasticity and information [25].

Stochastic games are games involving some form of random variable, chance or probability. Games involving a dice and many card games are in this category. The opposite is called deterministic games, and they do not contain any random effect.

The other dimension classifies games as either perfect-information or imperfect-information games. In a perfect information game all players have the same

knowledge about the current game state including its history. In perfect information games there is one best way play for each player. This will not necessarily guarantee a win, but will minimize the defeat. The trouble with many perfect-information games is however that the number of strategies is so large, that it is not possible to determine the best one. In imperfect-information games, players cannot know the result of their actions as the whole state of the game is not known. Often this occurs because players play simultaneously.

Table 14 is summarizing the classification of games along with some example games.

	Deterministic	**Stochastic**
Perfect-information	Chess	Backgammon
Imperfect-information	Paper-rock-scissor	Poker

Table 14: Classification of example games

There exists further classification of games known as Zero-Sum and Non-Zero-Sum games. Section 3.1.1 and 3.1.2 will explain these classifications.

3.1.1 Zero-Sum games

In a zero-sum game [17] no wealth or points is created or destroyed. That means that in a two player zero-sum game, what one looses the other win. The players do therefore not share any common interest. In a zero-sum game an optimal solution for a player always exists. In zero-sum games it is sometimes possible to reach a state where there exist an optimal strategy as explained below.

Dominant Strategy

A dominant strategy is a strategy that will always be the best, no matter what the other opponents do.

Consider a simple two player game played with nickel and quarters. In each round they both select one coin to play. If at least one player plays a nickel, player *A* win both coins, otherwise player *B* win both. Using the payoff matrix illustrated in Table 15, it is possible to find the dominant strategy for player *A*.

		Player B	
		Nickel	Quarter
Player A	Nickel	5	25
	Quarter	5	-25

Table 15: Payoff matrix for player A

It can be seen that that the largest payoff possible for player *A* if playing a quarter is 5. This is equal to the smallest possible payoff if playing a Nickel. In other words, playing a nickel is always at least as good as playing a quarter. So playing a nickel is called the dominant strategy.

Saddle point

Saddle point is a strategy, such that a player has no motivation for changing it because a change by any player would lead the player to possibly earn less than if she remained with the current strategy.

Changing the rules from the previous game such that if player *A* plays a nickel, Player *B* gives player *A* a nickel. If player *B* plays a nickel and player *A*, plays a quarter, player *A* gets a quarter. If both play quarters, player *B* gets a quarter from player *A*. The payoff matrix for player *A* can be seen in Table 16.

		Player B	
		Nickel	Quarter
Player A	Nickel	5	5
	Quarter	25	-25

Table 16: Player A payoff matrix

This game does not have a dominant strategy for player *A*, so another approach is needed to find the best strategy to play. To find the strategy that guaranties the largest payoff for player *A*, the minimum value is to be found in each row. In the example the lowest value in the first row is 5 while the lowest value in the second row is -25. The best strategy to play is then taking the row with the maximum of the minimum values. So the best strategy for player *A*, is always playing a nickel. This is called the saddle point.

3.1.2 Non-Zero-Sum games

Non-zero-sum games [17] are games where one person gain necessarily not comes at expense of someone else's. In zero-sum games, one person must lose for another to win. In non-zero-sum games the supply of a wealth or points is not fixed or limited. It is a case of building a resource rather than dividing it. In some non-zero-sum situations, a person can only benefit from a situation when others benefit as well. A classical example illustrating a non-zero-sum game is the prisoner's dilemma proposed by Merrill Flood in 1951.

Prisoner's dilemma

Two suspects are arrested by the police. The police do not have enough evidence for a conviction. The suspected has been separated and each of them is offered the same deal: If you confess and your partner does not, he gets a 15 year sentence and you go free. If he confesses and you do not, you get a 15 year sentence and he goes free. If none of you confess we can only give both of you 1 year for a minor charge. If you both confess, you both get 5 years. The dilemma is illustrated in Table 17.

	He confess	**He denies**
You confess	Both get 5 years	He gets 15 years, you go free
You deny	He goes free, You get 15 years	Both gets 1 year

Table 17: The classical prisoner's dilemma

What should each prisoner do? If we assume there is no honor among criminals each prisoner will typically reason that it is best to confess, as he does not no what the

other will do. But although each of them has done what seemed to be best for them, they could have gained a better result by remain silent.

3.2 Game trees

Game trees are described in [15] and [27]. One straight forward way to make a computer solve a given problem is simply to let it search for the best solution. This method is known as brute force, and can in games normally be represented as an n-ary tree where each node represents a status of the game. Such a tree is normally called a game tree. The root node represents the current state of the game. The root nodes children are all the possible moves from the current position. The children to these nodes are then the possible next moves for these game states. This is continued all the way down to the leaves of the game tree. Each leave represents a terminal position where the outcome of the game is decided.

3.2.1 Min-Max search trees

A Min-max search tree can be used to analyze the best possible next move for a player. Each leave must have a score. For example could 1 be associated with a win, 0 with a draw and -1 with a loss. Min-max works simply by examining the entire tree, and picking up the best path that can be forced.

To explain a Min-max game tree an example using the game of NIM will be given. The rules of NIM are as follow: The game board consists of piles of sticks. The players take turns removing any number of sticks from a single pile. The person removing the last stick looses. Figure 20 illustrates a game of NIM by use of matches. This game has three piles of sticks. In the left most one stick is present. In the middle and right most pile, there are two sticks. This can also be represented as (1, 2, 2).

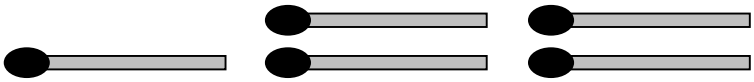


Figure 20: A game of NIM consisting of one pile with one stick and two piles with two sticks

Using a Min-max search tree it is possible to analyze every move. The Min-max search tree for the game of NIM with the (1, 2, 2) configuration, and two players, can be seen in Figure 21. At the leaves a victory for the current player, is illustrated by the value 1, while a loss is illustrated by a 0. The two emphasized paths in the central branch are the ones that can force a victory for the player about to make a move.

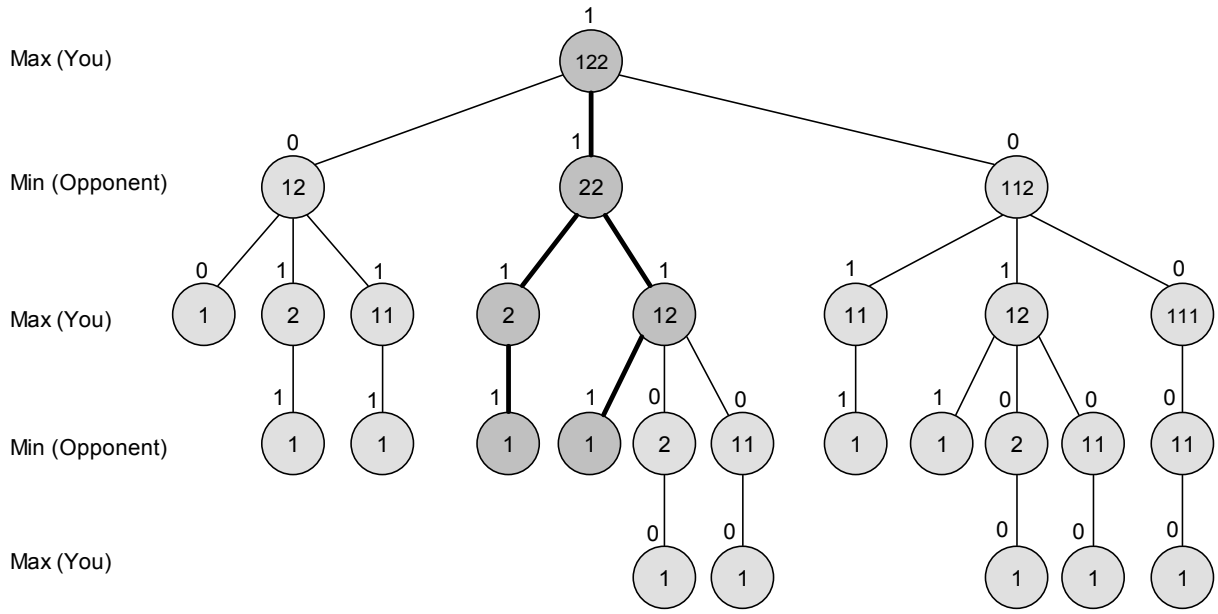


Figure 21: Game tree for a (1, 2, 2) game of NIM

The algorithm works by recursively traverse downwards the tree. In a two player game, all nodes positioned at an odd depth is called *Max* nodes, nodes at an even depth are called *Min* nodes. If the traversed node is a *Max* node its value becomes the value of the largest of its children. If the traversed node is a *Min* node its value becomes the smallest value of its children. The goal is then in each turn to take the branch with the highest value. Pseudo code for the algorithm can be seen below.

```
MinMax( Node node )
{
   if ( node is a leaf  )
      return score of node;
   generate legal next moves;
   if ( node is a min )
      for ( all children of node: c1, .. cn )
         return Min{ MinMax( c1 ), .. , MinMax( cn ) };
   else if ( node is a max )
      for ( all children of node: c1, .. cn )
         return Max{ MinMax( c1 ), .. , MinMax( cn ) };
}
```

3.2.2 Bounded look ahead

The problem with the Min-max approach and other brute force methods in general, is that their time complexity often grows exponentially.

For example in chess it has been estimated that each position in average has 35 legal moves. So if min-max is used to search one move ahead, 35 positions are to be examined. To look two moves ahead 35^2 positions are to be examined. As it can be seen in Table 18 a six move look ahead, requires almost two billion positions to be examined. It is in other words impossible to search all nodes in a large search tree.

Moves ahead to look	Number of boards to examine
1	35
2	1,225
3	42,875
4	1,500,625
5	52,521,875
6	1,838,265,625
7	64,339,296,875
8	2,251,875,390,625
9	78,815,638,671,875
10	2,758,547,353,515,625

Table 18: Estimated amount of boards to examine in a game of chess

One solution to this problem is simply to trim the tree at a given level. Depending on the time available a three or four move look ahead could be possible in a game of chess. Each of the nodes at that level is then evaluated by some heuristic function.

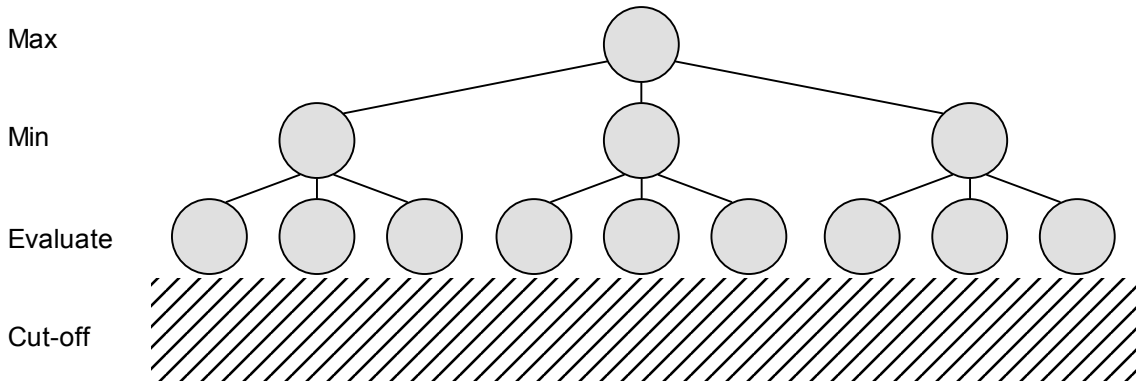


Figure 22: Bounded look ahead in a game tree

The evaluation function is constructed by an expert user and based on insight and experience. The evaluated nodes are then to function as the leaves. For all the nodes above this level the standard Min-max approach still applies. The problem with bounded look ahead, is that the evaluation function is not exact; it is just a heuristic function making an educated guess. Pseudo code for the algorithm can be seen below.

```
BoundedMinMax( Node node, Int depth )
{
   if ( depth is zero or node is a leaf  )
      return evaluated score of node;
   generate legal next moves;
   if ( node is a min )
      for ( all children of node: c1, .. cn )
         return Min{BoundedMinMax( c1, depth - 1 ), .., BoundedMinMax( cn, depth - 1 )};
   else if ( node is a max )
      for ( all children of node: c1, .. cn )
         return Max{BoundedMinMax( c1, depth - 1 ), .., BoundedMinMax( cn, depth - 1 )};
}
```

3.2.3 Alpha-Beta search

If the goal is to create the strongest possible chess player, it is important to search as deep as possible. To search deeper the branching factor must somehow be reduced. This can be done by use of an Alpha-beta search. The Alpha-beta search relies on the idea that if a choice is known to be bad for the player, then other choices are examined instead. Just like in the Min-max algorithm the tree is recursively traversed downwards. But in the Alpha-beta search, two scores are passed around in the search. That is the alpha value and the beta value.

The alpha value is the score that by any means can be forced. Anything that has a value equal to or less than this can be ignored. That is because there already is a known strategy that can produce the same or better result.

The beta value is the worst situation for the opponent. If the search finds something that produces a score equal to or better than beta for the opponent, it is too good. So the object for the player is not to make a move in that direction, and thereby deny the opponent the chance of using this strategy.

So if a move results in a score greater than alpha but less than beta, this move is to be searched further. During the search the values of alpha and beta are continually updated. If none of the legal moves produces a result that fulfill these requirements, it is avoided by making a different choice somewhere above in the search tree. In Figure 23 an Alpha-beta search is illustrated.

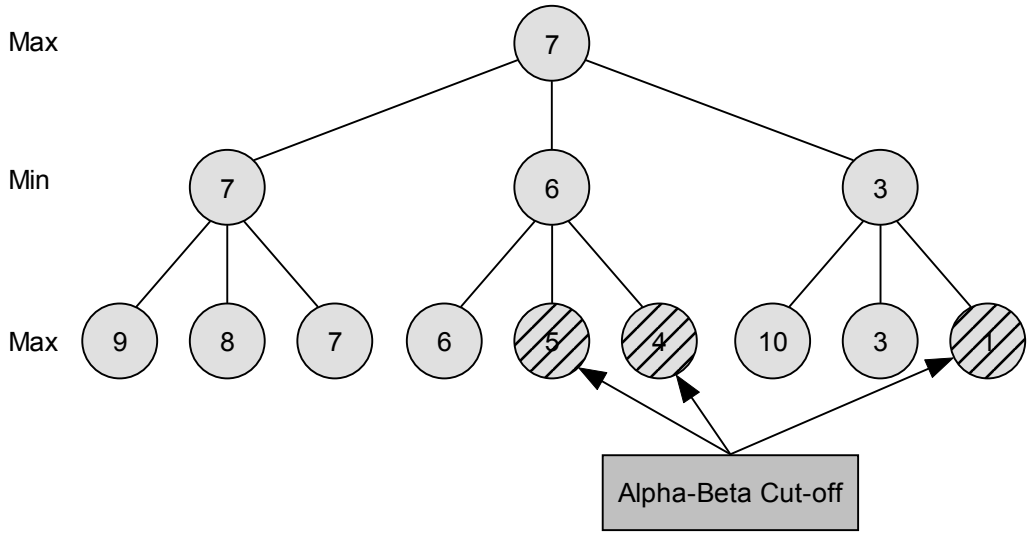


Figure 23: Alpha-beta search in a game tree

It can be seen that the nodes containing the values 5, 4 and 1 can be ignored. This is because already by examining the nodes containing the values 6 and 3, it is clear that the left-most branch will be a better move. To put it another way: if we follow the middle or right most branches, we know our opponent can force us to end the game with either 6 or 3 points. However taking the left most branches, the opponent is only capable of forcing a score of 7 points for us.

The problem with the Alpha-beta algorithm is that its performance depends on the order in witch moves are searched. If it always is the worst move that is examined first, a cut-off will never be achieved, and the algorithm works exactly in the same way as the Min-max algorithm.

However in the most favorable circumstances where the best move is always picked first, the expected branching factor is approximately reduced by the square root. This means that in the most favorable case the branching factor of chess can be reduced from 35 to 6. This will allow the algorithm to search almost twice as deep as the Min-max algorithm. Pseudo code for the algorithm can be seen below.

```
AlphaBeta( Node node, Int beta )
{
   if ( node is a leaf  )
      return score of node;
   generate legal next moves;
   if ( node is a min )
   {
      alpha = +infinity;
      for ( all children c of node )
      {
         value = AlphaBeta( c, beta );
         alpha = Min{ value, beta }
         if ( alpha <= beta )
            exit loop;
      }
      return alpha;
   }
   else if ( node is a max )
   {
      alpha = -infinity;
      for ( all children c of node )
      {
         value = AlphaBeta( c, beta );
         alpha = Max{ value, alpha }
         if ( alpha >= beta )
            exit loop;
      }
      return alpha;
   }
}
```

3.3 Agents

Before defining what an agent is in game theory a definition on the more general term computer agent, should be presented first. In [13] it is however argued that the term agent is in danger of becoming a noisy term that is subject for both confusion and misuse without an exact definition. They have however tried to come up with a weak definition of the term agent.

A general way to define the term agent is computer system that has the following properties [13]:

- **Autonomy**: Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.
- **Social ability**: Agents interact with other agents (and possibly humans) via some kind of agent-communication language.
- **Reactivity**: Agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it.
- **Pro-activeness**: Agents do not simply act in response to their environment; they are able to exhibit goal-directed behavior by taking the initiative.

It is however further argued that in the artificial intelligence community the following stronger definition also applies:

“An agent is to be a computer system that, in addition to having the properties identified above, is either conceptualized or implemented using concepts that are more usually applied to humans. For example, it is quite common in AI to characterize an agent using mentalistic notions, such as knowledge, belief, intention, and obligation. Some AI researchers have gone further, and considered emotional agents.” [13]

3.3.1 Game agent

So in games an agent can be defined as an entity with goals, plans of action and preferences that can interact with other agents. An agent can be conceptualized as a person, animal, firm, country, etc. An agent has the ability to decide on a sequence of actions that hopefully are more correct then actions made by other agents. Agents involved in games are referred to as players and assumed to be rational when selection actions necessary to reach its goal.

3.4 Conclusion on game theory

Game theory can in many ways be viewed as the theory of social situations. Some situations can be solved by simply finding the solution, while other problems are unique and requires overview and political skills to be dealt with in the best way. As in the real world, depending on the situation different skill is needed. Just like different games require different skill.

The ability to predict situations, as game trees can be used for, is a reasonable ability in AI related problems. Human being also poses the ability to predict situations. However where humans have an ability to select some limited but interesting situations for a more deep analysis, the computer instead has to select a much larger amount for analysis. In either way it is limited how many steps ahead it is possible to look for both humans and computers. As such brute force and thereby the ability to predict, can be viewed simply as yet and ability for intelligence. However relying on it alone will make it impossible to solve many real life situations.

As also illustrated in the chapter, not all problems are of a type where prediction is the key to success. Other factors such as randomness, probability and luck as well as decisions made by other, also has a key role in many situations. As such ability to learn and reason is to be important properties in many cases, if the goal is to develop an intelligent game agent.

4. Related work

This chapter will summarize some previous work done in the field of making computers play games successfully. This chapter will focus on some few but highly successful examples.

This chapter consists of the following sections:

- Deep Blue
- Chinook
- Evolutionary Checkers
- TD-Gammon
- Conclusion on related work

4.1 Deep Blue

Deep Blue [12] and [14] is a chess (reefer to appendix section 10.2.1 for the rules) computer developed by IBM researchers that in 1996 and 1997 played against the world champion of chess Garry Kasparov. While in 1996 it was defeated, it chocked the chess world by winning in 1997. Although IBM themselves said that the power behind Deep Blue is its ability to calculate 200 million positions in a second its foundation for success shall not be found in its computing power alone. This can be seen by the fact that in the year between the two matches Deep Blues computing power did only increase by a factor of two. Noticeable advances were instead obtained in Deep Blue's chess knowledge. This knowledge is gained from two databases. One called the opening book and one called the extended book.

The Databases

The opening book contains information about the opening moves that has been estimated to be best. An opening book is a set of positions along with recommended best moves. An opening book is often used to guide the early stages of a chess game.

The extended book is a database of positions that has arisen in around 700,000 real grandmaster games. For each of the positions contained in this database it is able to compute a value for moves. A move that seems good is assigned a higher value then moves that seems less good. Different factors contribute to the final score of a move.

- The number of times a move has been played.
- Relative number of times a move has been played. If move A has been played more then Move B, move A is expected to be better.
- Strength of the player that play the move.
- Recentness of a move. Recently played moves are likely to be better as possible flaws are yet to be discovered.
- Result of the move.
- Commentary on the move. Good moves are often annotated with "!" while weak moves are annotated with "?".

- Game moves vs. commentary moves. Annotators of chess games sometimes suggest alternate moves. Game moves are considered more reliable.

Making a move

Before making a move Deep Blue first checks if a move is available in the opening book. If it finds a move it is played immediately to save time for use later in the game. If a move is not found it consults the extended book. If the position is found there, it starts to award bonuses to the available moves. If a move with a high enough score is found, the move is carried out immediately, again to save time for later use in the game. If no specific move with a special high score is found, Deep Blue carries out a narrow search biased on the best scoring moves in the extended book. This is because there is the possibility that a stronger move has been overlooked. Deep Blue then has the chance of discovering it. The final move is then found by combining search results, and the results calculated from the extended book. In the event that a move can not be identified in any of the two databases, Deep Blue relies on its search algorithm.

4.2 Chinook

Chinook [10] is a checkers (reefer to appendix section 10.2.2 for the rules) program developed at the University of Alberta, Canada. Although it might not be as well known as Deep Blue, its performance is by no mean less remarkable. Currently Chinook holds the world championship title of Checkers, however if Chinook really is the best checkers player of our time, remains unknown. Chinook was set to play for the world title August 15, 1994. The match was against the seemingly unbeatable World Checker Champion Marion Tinsley. Tinsley was considered to be the ultimate checkers player, far superior to other human competitors. It was due to the lack of any real human challengers, Chinook was allowed to play for the title. Tinsley simply wanted to play someone able to give him proper resistance. After six games the match was drawn. However before the seventh match Tinsley resigned due to health concerns. In accordance with the rules Chinook was declared world champion. One week later Tinsley was diagnosed with cancer. He died April 3, 1995.

Chinook consists of four aspects:

- Search algorithms for traversing through the $O(10^{20})$ search space.
- Evaluation function to decide how good a position is.
- Endgame database holding perfect information for which endgame positions are won, lost and drawn
- An opening book database containing good openings to start a game with.

Search

Chinook uses a parallel alpha-beta search algorithm with minimum search depth of 21 moves. This ability has been obtainable because Chinook uses a search algorithm that has been adjusted such that it can select interesting lines of play for a much deeper search, while ignoring other less interesting lines of play.

Evaluation

The evaluation function consists of around 24 components, each containing several heuristic weights. Attempts to tune these weights using neural nets and genetic algorithms have been tried. They experience is however that the results are less successful then a hand tuned evaluation function. The evaluation function has been the biggest source of errors in Chinook. Several bugs have been discovered over the time, also during tournament play.

Endgame database

Chinook includes a complete 8 piece database. Every possible combination of play with 8 or less checkers left on the board has been completed. This database holds 440 billion positions stored in a 6 gigabyte database.

Opening book

The opening book is a database containing the opening plays that has been published in the literature over the years. After the initial database had been build several month ware used to let Chinook check the positions for errors. As a result, several hundred positions were discovered to have errors.

4.3 Evolutionary Checkers

Evolutionary checkers [7], [8] and [11] is a checkers game developed bye Kumar Chellapilla and David Fogel at the University of California. Although it does not hold the world championship of computer checkers, it has turned out to be quite successful. The approach taken in creating the checkers player differs from the approach taken in creating Chinook and Deep Blue. They argue that Chinook and Deep Blue can not be considered intelligent as everything they know, it knows because they ware explicitly told. Chinook and Deep Blue learn nothing on its own, and how can a system that never learns be considered intelligent.

The challenge when developing evolutionary checkers was to create an algorithm that instead of relying on human expertise, explicitly put into the program, had the ability to learn the game simply by playing it. The hope was that after several games a competent algorithm capable of playing checkers would emerge.

The experiment

The approach taken was to use evolutionary computing to evolve neural networks that would represent strategies for the game of checkers. 30 neural networks where initialized in a random state, and then set to compete against each other using co-evolution for 250 generations. In each generation the neural net played a series of games. The 15 best scoring neural networks where then maintained as parents for the next generation.

The neural network used was a standard feed forward network with 32 input nodes, corresponding to the 32 possible positions of a board. It had two hidden layers consisting of 40 and 10 nodes and one output. The input to the neural net consisted of 32 values that could have the values from {-K, -1, 0, 1, K}. K represents a king (initially

set to 2.0), 1 a checker and 0 an empty square. Positive indicates the piece belongs to a player, negative that it belongs to the opponent. A move was determined by feeding the neural net with possible new board positions from the current one. The output was then a value in the range of -1 to 1. The higher the value, the better the board was estimated to be. The highest scoring board was then selected as the move.

Results

The best evolved network was used to play against human opponents on www.zone.com. Human player's login and then play against other human opponents. Initially new players are assigned a rating of 1600. The rating then increase or decrease depending on a player's performance against other players. Over a two week period, 90 games where played, none of the opponents where told they where playing with a computer. The final rating for the neural network was 1901.98 rating it as a class 'A' player. It easily defeated players rated below 1800, was playing even with players rated between 1800 and 1900, but was not able to match players rated more than 2000, which was in the expert and master categories.

4.4 TD-Gammon

TD-Gammon [9] is a backgammon (reefer to appendix section 10.2.3 for the rules) game developed by Gerald Tesauro at IBM research. The author claims that TD-Gammon is by far the most successful backgammon program developed. As evolutionary checkers, TD-Gammon is based on a neural network. It does however take a slightly different approach when learning.

Architecture

The neural network architecture is a feed forward network which as input takes the number of white and black checkers at each location. The neural net has four output values indicating whether white or black is estimated to win a normal victory or a gammon. Due to its rarity, a backgammon victory is ignored.

TD-Gammon does however not rely solely on a neural network to play. Explicitly programmed features are also put into the program. This includes such features as evaluation of the strength of a blockade, probability of being hit, and a search algorithm looking two moves ahead.

Training

Where evolutionary checkers relied on evolution to create a capable player, TD-Gammon relies on a training algorithm that can be somewhat classified in the category of reinforced learning. After each move the current board is evaluated. The weights within the neural net are then adjusted accordingly using a formulary that among other things takes a heuristic value between 0 and 1 as input. The higher the value the poorer the board is estimated to be, and the more the weights are adjusted. At the end of each game a final adjustment is done on behalf of the final outcome. During training the neural net is used to select moves on both sides.

Results

TD-Gammon has been tested against top world players in a series of exhibition games at the world cup of Backgammon. Rather surprisingly the general experience was that the top players of the world were only able to pull some very narrow victories. The general opinion is that TD-Gammon plays at a strong master level where most commercial programs play at a weak intermediate level. It is even believed that in long game sessions or tournaments the game could pull a win as it does not gets tired as humans does.

4.5 Conclusion on related work

The game agents presented in this section rely on different approaches. In one category Deep Blue and Chinook can be placed. In a second category Evolutionary checkers can be placed. TD-Gammon can be considered a hybrid that can be placed a little both categories. Although all agents have been highly successful, it seems that Deep Blue and Chinook have been the most successful. However if these can be called intelligent is at discussion. They can probably better be classified as expert systems. All their knowledge has been explicitly put into the program by human experts; they do not learn anything on their own. Evolutionary checkers and TD-Gammon on the other hand, has been able to learn the games by playing them, and can better be classified in the category of intelligent game agent.

However, an interesting point is that although Deep Blue and Chinook relied on expert knowledge, this knowledge was also encoded in a way that was understandable in their given domain. Evolutionary checkers and TD-Gammon however instead relied on arbitrary weights within a neural net. Getting any information out from this network an eventually combine the obtained knowledge with other learning and decision making techniques would be hard, if not impossible.

An interesting approach could be to let the agent it self build a major database of knowledge (knowledge base) in a domain-understandable way, and somehow let the agent itself also encode its learning or evaluation function into it. This would also at a later point enhance chances for successfully apply further techniques to the agent to assist in its decision making.

5. Analysis

Previous chapters can be viewed as an analysis of existing research and techniques in the two areas of AI and games. This was to establish a fundament and get an insight into these fields. Beginning from this chapter a more personal approach will be taken. This analysis is to lay the fundament for the creation of an intelligent agent implementing a solution model capable of playing the game of tic-tac-toe.

This chapter consists of the following sections:

- Intelligence
- Requirements
- Domain
- Agent by classical AI
- A thesis for learning
- Goal directed learning
- Solution complexity
- Conclusion on analysis

5.1 Intelligence

This section will take look on what intelligence is. This is to help when defining requirements for an intelligent agent. But before going to deep into intelligence and discussing the capabilities for an intelligent system, it should be determined if it is at all possible to create intelligence on a machine. Although most should agree that at has not been successfully done yet, it still needs to be determined if it is at all possible. The answer to this question lies very much in ones own view of intelligence. Two views can be taken on intelligence, a mechanistic or romantic [2].

In the mechanistic view intelligence is believed to be an algorithm. This means that there is a recipe for intelligence, which can be broken down into a finite step by step instruction. The invention of human like artificial intelligence is just a matter of understanding it, and implementing it. This does not mean that intelligence is something humans currently is able to create, but it is believe to be possible to create intelligent machines in the future, as it is just a matter of understanding it. One supporter of this belief is Bill Gates which as said: “I don't think there's anything unique about human intelligence. All the neurons in the brain that make up perceptions and emotions operate in a binary fashion.”

In the romantic world view intelligence is believed to be something very unique. It is not governed by any rules, and can as a consequence not be implemented as an algorithm. All supporters of this belief do not necessarily believe that humans will not be able to create intelligence. They just believe that it has to be created in some other way than rules implemented on machines. This means that artificial intelligence is believed to have a greater chance of success, in for example a scientific branch like biology.

Depending on what view one supports, it should lay down the fundament for what the goal is, and requirements are from an intelligent agent. As a result, the view that is to govern this project is the romantic. This is because catching the essence of intelligence in an algorithm, and creating a human like intelligent system, as supporters of the mechanistic view should find possible, is not seen as a realistic goal for this project. Instead the goal is, like most existing research in AI, to find ways to simulate intelligence.

To obtain an initial understanding of intelligence and what is to be simulated, let us highlight some of the interesting terms when looking up intelligence, and its synonym *mind* on Dictionary.com:

Intelligence
1. a. The capacity to **acquire and apply knowledge**. b. The faculty of **thought** and **reason**. c. Superior powers of mind. *See Synonyms at mind.* 2. …

Synonyms at **mind**
1. The human consciousness that originates in the brain and is manifested especially in **thought**, **perception**, **emotion**, **will**, **memory**, and **imagination**. 2. The collective conscious and unconscious processes in a sentient organism that direct and **influence mental and physical behavior**. 3. The principle of intelligence; the spirit of **consciousness** regarded as an aspect of reality. 4. The faculty of **thinking**, **reasoning**, and **applying knowledge**: *Follow your mind, not your heart.* 5. …

So intelligence gives, according to its definition, the following capabilities:

- Having thoughts, perceptions, emotions, will, memory, imagination and consciousness.
- Ability to obtain knowledge
- Thinking and reasoning
- Influencing the mental and physical behavior.

These capabilities are however not to be seen as isolated from each other, but instead as capabilities that cooperate, to create what can be classified as intelligence. A way to illustrate this can be seen in Figure 24.

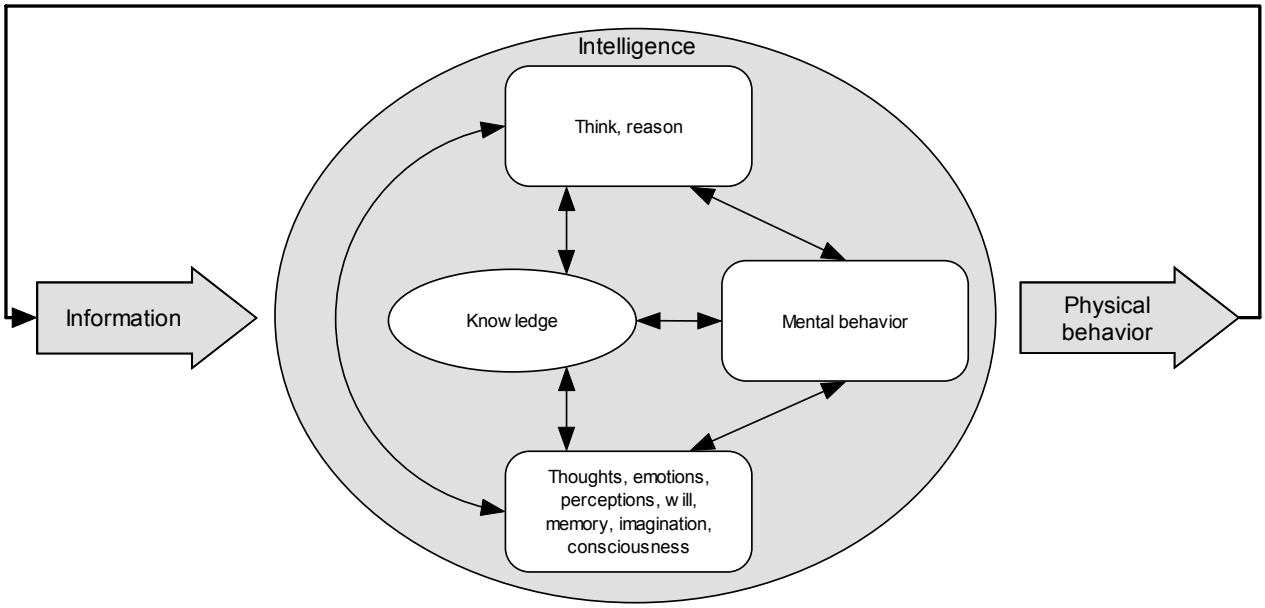


Figure 24: Illustration of intelligence

Information is continually given to the mind. This information is then if understood, and someone's intelligence is capable of it, converted to knowledge by use of existing knowledge and the capabilities represented by the rounded rectangles. All the capabilities mutually influence each other and ultimately govern the physical behavior which is how the intelligent entity acts. This physical behavior will ultimately create new information that is given to the mind.

Intelligence can for the entity possessing it, be viewed as a rather complex process. But the processes within the mind are of minor interest for entities observing other intelligent entities. The only part other is capable of observing is the information (or stimuli) given and the response obtained. It is as a consequence extremely important that the physical behavior seems logical in relation to the information given. Because we human has defined our self as intelligent, we measure intelligence against our own definition of intelligence, and that is how it to some degree has been the norm for human's to act, given that information. However what is interesting is that previous information is able to govern future actions; this can also be referred to as learning. Intelligent entities are aware of that and as a consequence knowledge and learning should play a major role in artificial intelligence. Many of the other capabilities are of minor importance. For example emotions can not directly be observed. Emotions might trigger someone to be sad, that again may result in one crying. But it is the physical behavior of crying that can be observed by other, not the emotion itself. As such the oval in Figure 24 can be considered a black box, and the challenge in artificial intelligence, is then to create this black box known as intelligence.

The physical behavior, that is the output from an intelligent entity, can be categorized in one of two categories, which are reflexes and deliberate actions. Deliberate actions, which are the one governed by the mind, and the interesting in this context, will in most (if not all) cases have the purpose of helping the entity reaching a desired goal. That could be win a game, obtain more knowledge, have fun, get from A to B, solve

an equation, etc. and with greater intelligence comes the ability to reach more complex goals. Reaching a goal is however not to be distinguished with luck. The goal might be reached, but if the intelligent entity has no idea on how it was reached, and is not able to repeat it, it cannot be defined as intelligence.

In essence AI is to create some system that is feed with information. Given this information it should be able to obtain knowledge and thereby learn about the domain from which the information is given. Given some information the system should, if necessary, decide on an output, which is the response to the problem at hand and a suggested solution to how the entity will try to reach its goal. To be able to learn it is necessary for the intelligent entity to receive feedback on its ability to reach the goal such the entity is able to distinguish good decisions from bad. Eventually the system is then to become better and better at solving a given task and to reach the desired goal. Ultimately intelligence can then be defined simply as *an entities ability to achieve goals*.

5.2 Requirements

In this section desirable requirements for the intelligent agent will be presented. These requirements are to work as a guideline in the creation of an agent. It might not be possible to fulfill all desirable requirements and as a consequence a tradeoff that ultimately can lead to the failure of fulfilling some requirements might be unavoidable.

The ultimate requirement is that the agent should be able to learn. The implementation of some expert system like Deep Blue (section 4.1) and Chinook (section 4.2) where all knowledge is put explicitly into the agent is not a satisfying goal for this project. The agent should be able to adapt to new and unforeseen strategies practiced by opponents. However if the element of machine learning is supported this requirement should be trivial fulfilled as well. A small amount of explicit coding of the simplest domain rules could however be required, especially to guide the agent while the obtained knowledge is still sparse early on in its process of learning.

The agent should if possible both be able to learn by playing human opponents as well as by co-evolution. Co-evolution is the process of letting the agent(s) making decisions for all players. The agent should then be able to be set to play several subsequent games without human interference and ultimately become better and better at playing the game.

The overall model of the agent should be somehow flexible such that it with a minimum of adjustments can be applied to different domains. An example of such a model is artificial neural networks where even though the specific design such as inputs, outputs, layers, etc. may differ from domain to domain, the overall model of a neural net is still the same.

The requirements are summarized below:

- Include the element of machine learning so it is able to learn and adapt to situations.
- Ability to learn by playing human opponents as well as learn by use of co-evolution.
- An overall model that is not just specialized for one domain.

These three basic requirements are to govern the future development of an intelligent game agent.

5.3 Domain

The initial domain selected is the game of tic-tac-toe. This is a rather simple game most kids are able to play rather well. By adults it is however often considered boring because most are able to play the game rationally, and all games played rationally will end in a draw. It is however also this fact that makes it excellent as initial test domain, as it is obvious that a solution model must be rather effective in this domain, to have any chance of success in more complex domains such as chess, checkers or backgammon. If the created agent, and its solution model, then become able to play tic-tac-toe successfully, it can at a later point be integrated in one of these more complex game domains. The fact that tic-tac-toe has the ability to always end in a draw, also makes evaluation of the agent relatively easy.

Tic-tac-toe is a standard two-person board game where the goal for the player X is to get three X's in a row horizontal, vertical, or diagonal before the player O gets three O's in a row. The game starts by placing the X or O by the respective players until the goal is achieved or the game is a draw because all squares are taken and none of the player has won. A tic-tac-toe board where X has won is illustrated in Figure 25.

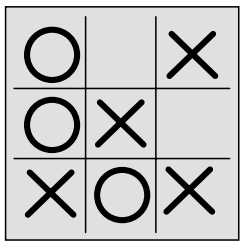


Figure 25: Tic-tac-toe board where X has three on row

This simplicity of the game also makes it an excellent for explanation, as examples on how the solution model works, can be kept at a fairly manageable level.

Tic-tac-toe is not to be confused with noughts and crosses. In this game each player has only three pieces. When all three pieces has been placed on the board one is to be removed and placed at another square. This gives noughts and crosses the ability to be played forever and never end. The board layout and the object of getting three in a row do however also apply in noughts and crosses.

5.4 Agent by classical AI

This section will describe one method, fulfilling the requirements given in 5.2, for how a tic-tac-toe agent can be created by use of classical AI techniques. The overall method presented in this section will rely on artificial neural networks as model. Two training methods will be used to let the network learn the game. The two training methods are the back-propagation algorithm and genetic algorithms.

5.4.1 Overall architecture

The neural network architecture can be a standard feed-forward network with 9 inputs and 9 outputs, one for each square on the board. The amount of hidden layers and node in each hidden layer is harder to decide without ability to test the networks ability to learn. So for this description we decide in one hidden layer with 18 nodes. Each input is connected to every node in the hidden layer and all nodes in the hidden layer are connected to every node in the output layer. With the decided network architecture that gives a total amount of 324 adjustable weights. For the step function we decide on a standard sigmoid function. This ensures the output from each node and the network itself, will be in the range greater than 0 and lower than 1.

The input to the network is the current representation of the board. The value 1.0 represents an O and the value 0.5 represents X. 0.0 indicates an empty square. The output from the network is 9 values in the range greater than 0 and lower than 1. The output with the highest value, and where the square is empty, is where the agent decides to place its piece. Figure 26 illustrates the network. It should be noted that for simplicity the hidden layer is illustrated as one “super” node instead of 18 separate nodes. Further more the output values are rounded at one decimal.

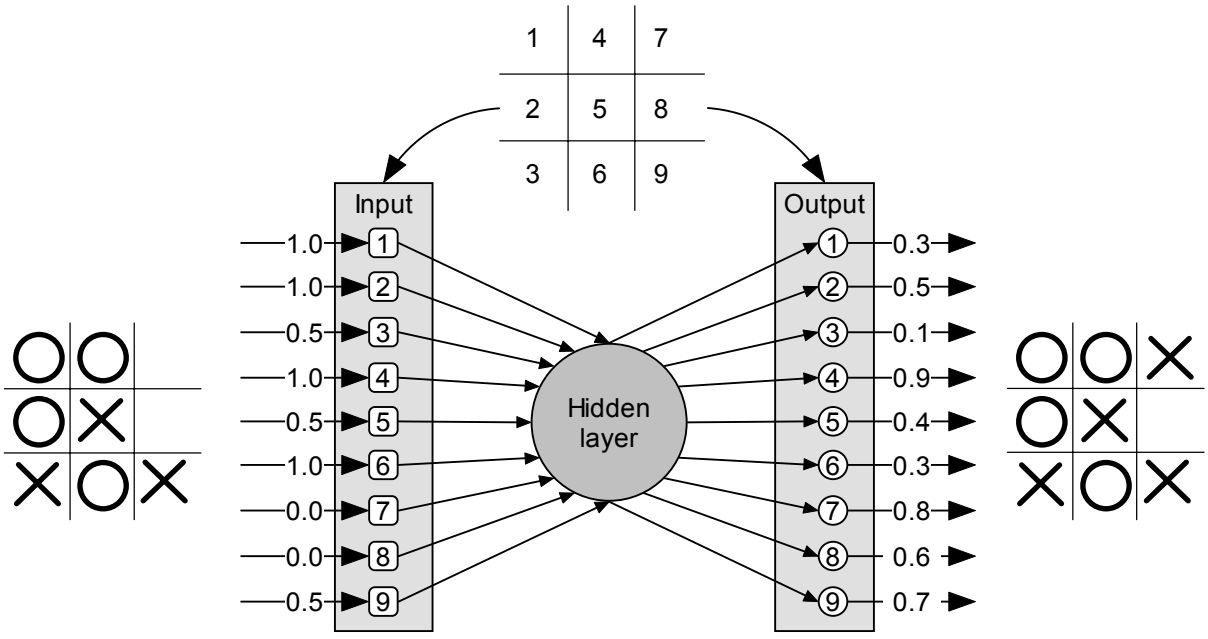


Figure 26: Neural network capable of playing tic-tac-toe

As illustrated the board layout is converted to its corresponding input values. Values will then flow through the network to obtain an output. In the above case, output

number 4 has the highest output of 0.9. This square is however already taken and as a result the square where the agent decides to put its piece is 7. This is the free square with the highest output value.

5.4.2 Learning by playing humans

For the agent to be able to play tic-tac-toe it is necessary for it to learn it. This can be done when playing humans by use of the back-propagation training algorithm described in section 2.6.4.

Consider the board layout illustrated in Figure 27. The human player plays X and it is X's turn.

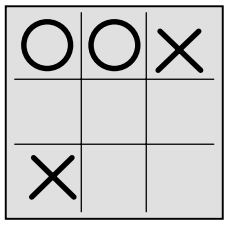


Figure 27: X's turn

X should clearly put its piece in the center square to win the game. To train the network to learn this strategy the network has to be feed with this board. The obtained output is then compared to the desirable output, which is obtained by the move the human player plays. The network is then trained by the back-propagation algorithm. This process is explained below.

First step is to switch the board such the agent sees the situation as its own. This is illustrated in Figure 28.

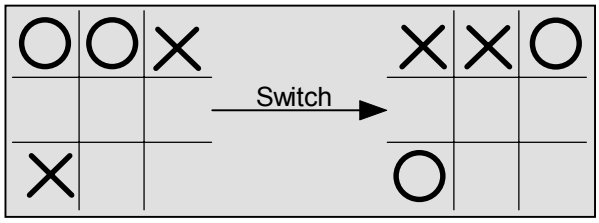


Figure 28: Switching board layout to match opponents view

The switch board is then feed as input when training the network to obtain an output. As the human player (hopefully) would put the piece in the center square the desirable output can be identified as, as low values as possible in all outputs except for the selected center square, where the output value should be as high as possible. This is illustrated in Figure 29. Remember that because the sigmoid function is used as step function, an output value can never be exactly 0 or 1; as a result the training values should never be exactly one of these values.

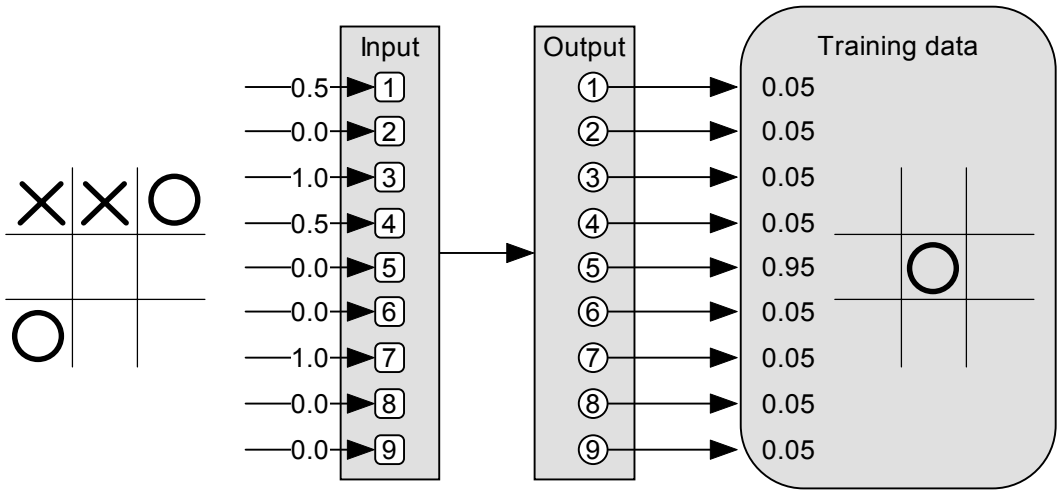


Figure 29: Training of neural network

As the desirable output for a specific input can be obtained from the player move, the network can be trained with the back-propagation algorithm. This process is then repeated every time the human player makes a move. As more game is played and the network is trained with more possible board combinations, the agent should hopefully become better and better over time. A big drawback with this method is that the agent learns by watching a human player that might make mistakes, not by the outcome of the games.

5.4.3 Learning by co-evolution

To learn in co-evolution genetic algorithms can be used to evolve the weights within the network. An amount of agents can then be initialized and set to compete against each other in a series of games. The fitness score used in the genetic algorithm evolving the network weights could then be calculated depending on how well an agent is performing. 3 points could for example be assigned for a victory, 1 point for a draw, and 0 point for a loss. The process is illustrated in Figure 30.

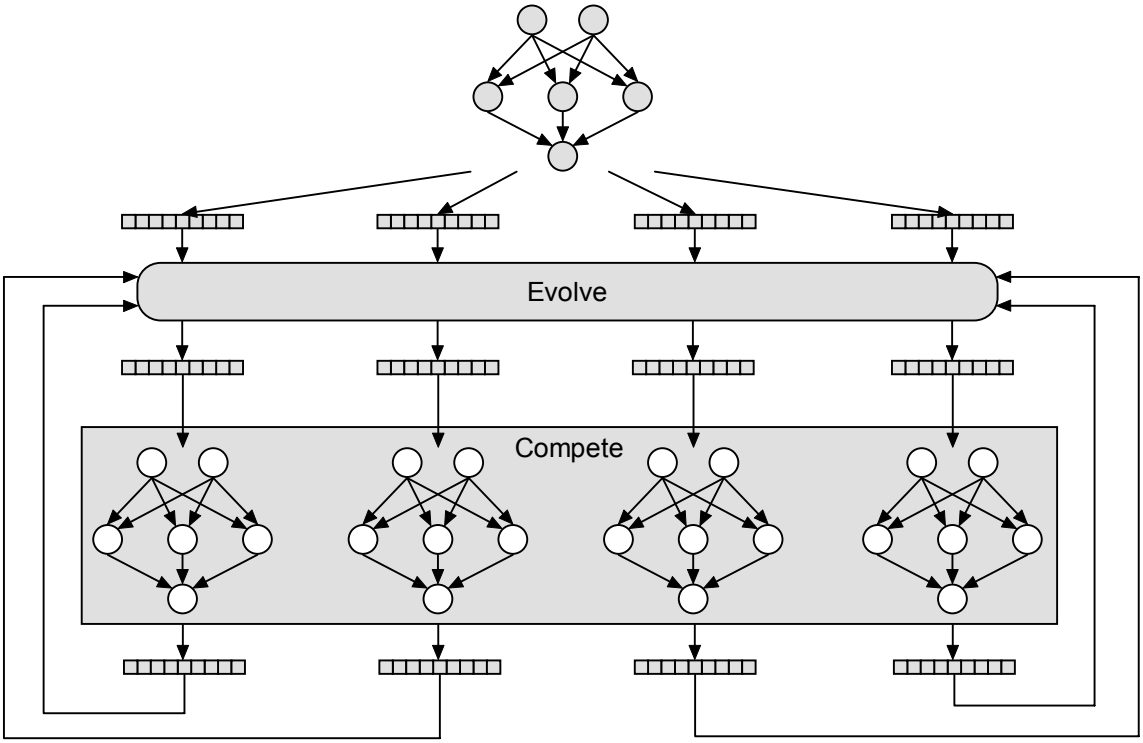


Figure 30: Tic-tac-toe training by use of a genetic algorithm

The agent's network weights are to begin with copied into a series of vectors that are to be the chromosomes in the genetic algorithm. Because all these chromosomes at first are the exact same we start by evolving them. This is to make sure the mutation step in the genetic algorithm will make a slightly alteration in some of the chromosomes. These evolved chromosomes or vectors of network weights, are then copied back into a series of neural networks. The networks are then set to compete against each other in a series of games. After the competition the network weights are again copied into a series of vectors. Depending on how well a network was performing, a fitness score is assigned to the chromosomes. After this process has been repeated a series of times, the best performing network in the final competition, is selected as the agent.

5.5 A thesis for learning

Existing AI techniques has in chapter 4 showed to be useful for the creation of intelligent game agents. In section 5.4 it was described how classical AI techniques, fulfilling the requirements in section 5.2, could be used for the creation of a tic-tac-toe playing agent. In this section a new idea that learns by the outcome of a series of decisions taken to reach a certain goal will be presented. The ambition is to create a new model for machine learning in classical board games.

Consider some inexperienced player about to play the game of tic-tac-toe against an experienced player. The inexperienced player has never played the game before and does not know its rules. The experienced player starts by putting a piece somewhere on the board. The two player alternate placing their pieces until the game is over and

the inexperienced player is told that he has lost. In the next game the inexperienced player starts. The inexperienced player starts out with an empty board and has 9 possible new moves. Of these 9 moves one of them can be recognized to create a board pattern that for the opponent lead to a victory in the previous game. A possible best move could than be to play the exact same move. However after this move the experienced player, plays a move that creates a board pattern not recognized from the previous game, and as a consequence the inexperienced player does not know what a smart move could be. However suddenly after some moves, the experienced player makes a move that creates a board layout where one of the possible next moves can be identified to have a matching board pattern from the previous game. This board layout was however created by the player that ultimately lost the game, and as a consequence this specific move would probably be smart to avoid. So a move creating another board pattern is played. Ultimately the inexperienced player probably looses again. The inexperienced player has however by playing this second game learned even more board combinations and their ultimate outcome. This process can be continued game after game. After each game a series of board pattern, and the final outcome of the games they have been played in, will be known. This process is illustrated in Figure 31. X is considered to be a human player; O is the computer agent about to learn the game of tic-tac-toe. The two players alternate starting the game.

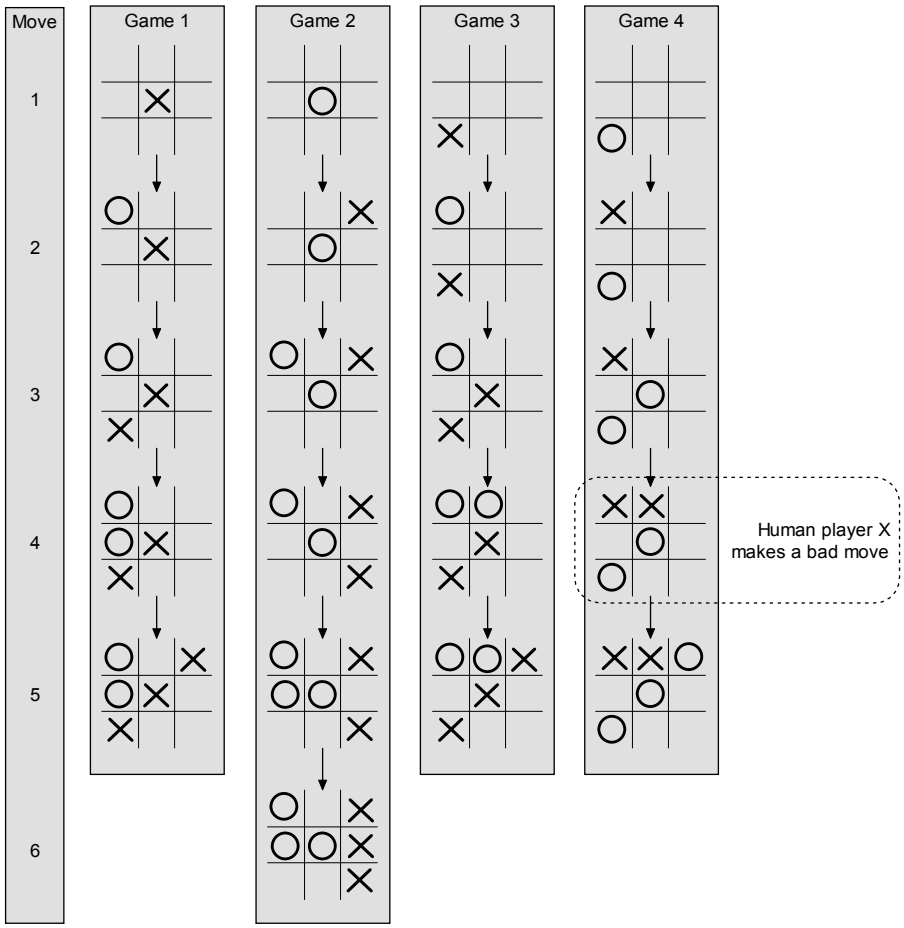


Figure 31: Machine learning by recognizing patterns and their outcome

Let's start by defining the strategy the computer agents uses for playing when knowledge about a possible next move not yet has been gathered. Then it will try to place its piece in the upper left corner of the board and then downwards and to the right as illustrated in Figure 32.

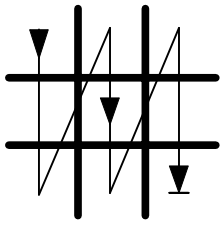


Figure 32: Tic-tac-toes agents' initial playing strategy

I game 1 the human player X starts by making a move. At this point the computer agent has no knowledge and it simply plays its standard strategy until X win the game.

In game 2 the agent is able to identify and create an opening pattern in move 1 that has lead to a victory in a previous game and play this move. At this point the human player X plays a move that creates a new pattern from which no known patterns can be identified. As a result the agent again relies on its standard strategy and looses the game.

In game 3 the human player X again starts the game. In move 2 the agent can not identify any patterns and again relies on its standard strategy. When about to play move 4, the agent is however able to identify a pattern that previously (game 1, move 4) has lead to a loss. This move is then avoided and a different move is played instead. Ultimately the human player X wins anyway.

In game 4 the agent starts by creating a pattern that again has been identified to lead to victory in all the cases where it has been played (Game 3, move 1). Luckily the human player X makes a move that creates a pattern from which another recognizable pattern can be derived. In move 3 the agent is as a result able to play a move that creates a pattern that previously (game 1 and 3, move 3) has lead to victory. At this point Human player X makes a major mistake in move 4, going for a victory in the top row and completely overlooking O's possibility for a diagonal win. As the winning pattern further more can be directly identified from a previous game (game 3, move 5) this move is played and as a result the agent wins this game.

Thesis
The more games with a desirable outcome for a player the board pattern has been represented in, the better the board is estimated to be, and the more games with a non desirable outcome for a player the board has been represented in, the worse the board is estimated to be.

5.6 Goal directed learning

In the previous section a thesis for machine learning was identified. This section will explain a way of modeling this method.

One thing to make sure is that the agent is able learn by all the information it is presented with. That means that in a perfect information game like tic-tac-toe, the agent should not only learn by its own decisions, but also the opponents decisions. It is also important that the agent does not rely on explicitly playing a specific piece and is able to play X in one game and O in another and still use the same obtained knowledge. For example, if in Figure 33 the agent in the leftmost plays O and it is O's turn, and plays X in the rightmost and it is X's turn, the boards should not be identified differently.

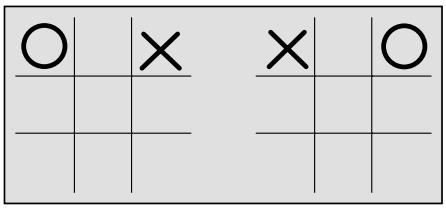


Figure 33: Two boards that can be either different or the same depending current players turn

To solve this, the pieces X and O will instead be represented as letters. The letter *A* will represent the player starting the game; *B* will represent the second player. Therefore the four games illustrated in Figure 31 can instead be represented as in Figure 34. The use of letters also insures a more general representation of the domain. In cases where games have more then two players, following letters can be used to represent them.

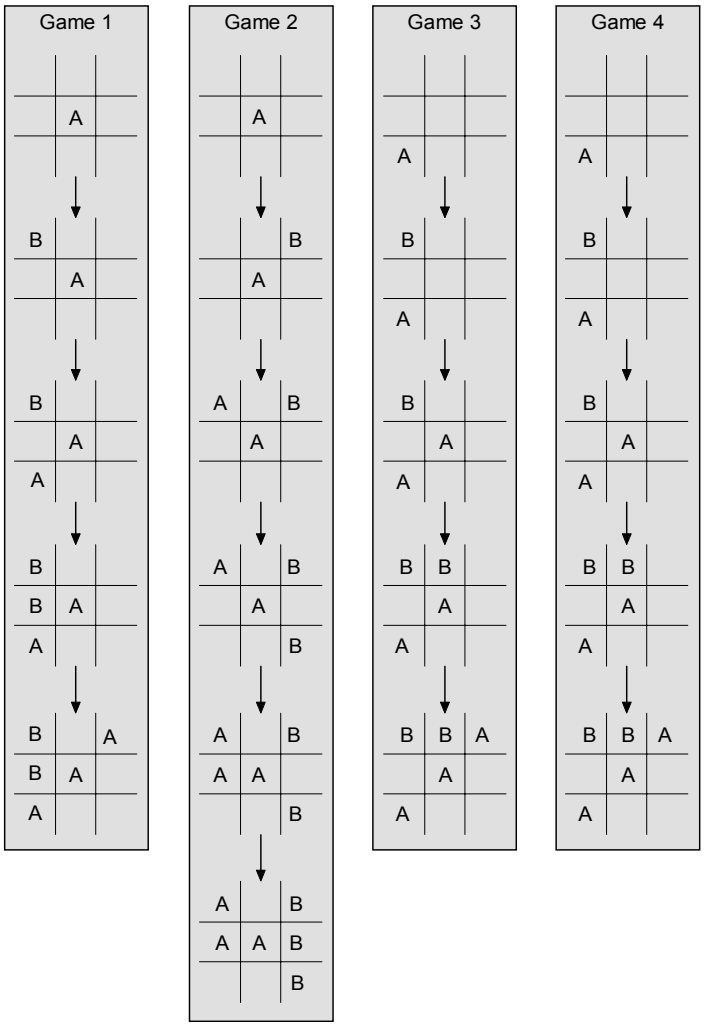


Figure 34: Tic-tac-toe represented by letters (A is the starting player).

The goals the agent can reach are also to be defined. Tic-tac-toe has three goal states; that is a win, a draw and a loss. Some way of representing the desirability for reaching each of these goals should also be defined. This will be done by assigning a numerical value to each goal, the higher the value, the more desirable the goal is considered to be. In this example we will assign 2 points to a board pattern if it ultimately reached the win goal, 0 points for reaching the draw goal, and -1 point for reaching the loss goal. The chance for a win is in this example therefore considered to be of higher priority, then the risk of a loss.

The knowledge base, which is how the learning is represented within the agent, is also to be defined. Theoretically every game could simply be stored. It is however desirable to keep the size of the knowledge base at a minimum. As such a board pattern should only be represented once. This should also make decision making faster as only one board pattern needs to be extracted from the knowledge base instead of a whole range of identical patterns from a range of different games. To explain how the knowledge base can be represented the four games represented by letters in Figure 34 will be used.

First thing to note is that the agent's knowledge base should not be updated before at least one of the goal states has been reached. This is because no usable knowledge can be extracted from an incomplete game in this model. Therefore the whole game and its progress are to be kept isolated from the knowledge base until it is over. Let us consider the case where game 1 has just been completed and the knowledge base has been updated. This can be represented as in Table 19.

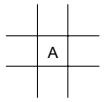
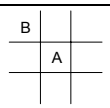
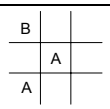
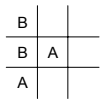
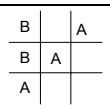
Pattern	Count	Score A	Score B
	1	2	-1
	1	2	-1
	1	2	-1
	1	2	-1
	1	2	-1

Table 19: Knowledge base after game 1

Count is the total number of times the board has been observed by the agent in all the games it has been playing. *Score A* and *Score B* is the total score for that board pattern for each of the corresponding players, remember A is the starting player B is the player not starting. The method is than each time a new board pattern has been identified in a game, to add it to the knowledge base along with its corresponding *Score A* and *Score B* (2 for a win, -1 for a loss), *Count* will always in tic-tac-toe be 1. If a pattern already exists in the knowledge base, the values are instead updated with the sum of the current values and the new values. In the case where the topmost pattern for example has been identified in a subsequent game where B won. *Count* is updated to 2, *Score A* to 2 + (-1) = 1, and *Score B* to -1 + 2 = 1.

The final knowledgebase after the four games in Figure 34 can be seen in Table 20.

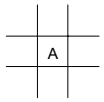
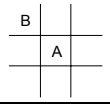
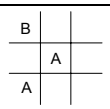
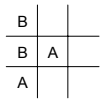
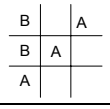
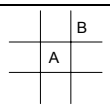
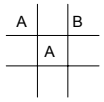
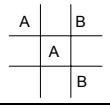
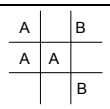
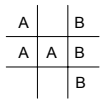
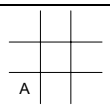
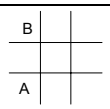
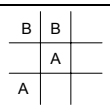
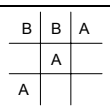
Pattern	Count	Score A	Score B
	2	1	1
	1	2	-1
	3	6	-3
	1	2	-1
	1	2	-1
	1	-1	2
	1	-1	2
	1	-1	2
	1	-1	2
	1	-1	2
	2	4	-2
	2	4	-2
	2	4	-2
	2	4	-2

Table 20: Knowledge base after game 4

Calculating the actual and final score of a board is then simply dividing the score of a board for the corresponding player with the *count*. This is to normalize the final score to a value between 1 and -1. In the case where the agent is presented with the board in Figure 35 it is able to identify 7 possible moves. Looking in its knowledge base it is

able to recognize one of these moves, which is pattern 3 in its knowledge base illustrated in Table 20.

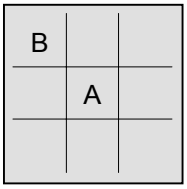


Figure 35: Board presented to agent

Calculation of the desirability for this recognized board is then done as illustrated in Table 21.

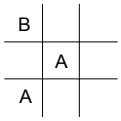
	A: 6 / 3 =	2
	B: -3 / 3 =	-1

Table 21: Calculation of board patterns desirability

All patterns not contained in the knowledge base are assigned the value 0 for *A* and *B*. As it is the agent turn and it is playing *A*, this pattern could be a possible good move since the score for *A* (the agent) is higher, and the score for *B* (opponent) is lower then 0. The agent will then play this move.

Although not possible in practice as *A* always plays first, but has it instead been *B* that has been presented with this board it should instead have been avoided.

Modeling machine learning in the presented way should have some advances over the neural network and genetic algorithm method presented in section 5.4.

- Knowledge is contained in a domain understandable way, not as arbitrary weights. This could make it possible to more easily add additional methods supporting an agent in its decision making.
- The same basic method can be used weather the agent is playing humans or another computer agent and does not have to rely on two different learning approaches as the model presented in 5.4.
- Wrong or bad decisions by the opponent does not learn the agent to play this flawed strategy as the agent learns by the outcome of decisions, not by watching and copying the opponents decisions.

More generally it is a model that is useful in cases where the possible outcomes and desirability of the outcomes is known, but where the problem is to find the best solution to reach these outcomes, which is the exact case with a wide variety of games. This is achieved by recognizing patterns between the initial state and a final state, and then rating these patterns depending on the final outcome. This model will be refereed to as: *Goal directed learning*.

5.7 Solution complexity

This section will take a look on the goal directed models complexity since a possible weakness with a model could be its complexity. For example game trees described in 3.2 is theoretically an excellent solution; its complexity is however of such a size that it is not a possible solution in games such as chess, checkers or backgammon although it would be possible for the game of tic-tac-toe.

The extreme complexity of game trees lies in the fact they look at all the possible games, and ignores the fact that there in different branches lay boards there is completely identical. For example by looking at the very limited part of the game tree illustrated in Figure 36, it can be seen that the two boards at the bottom are completely identical although they lie in two different branches.

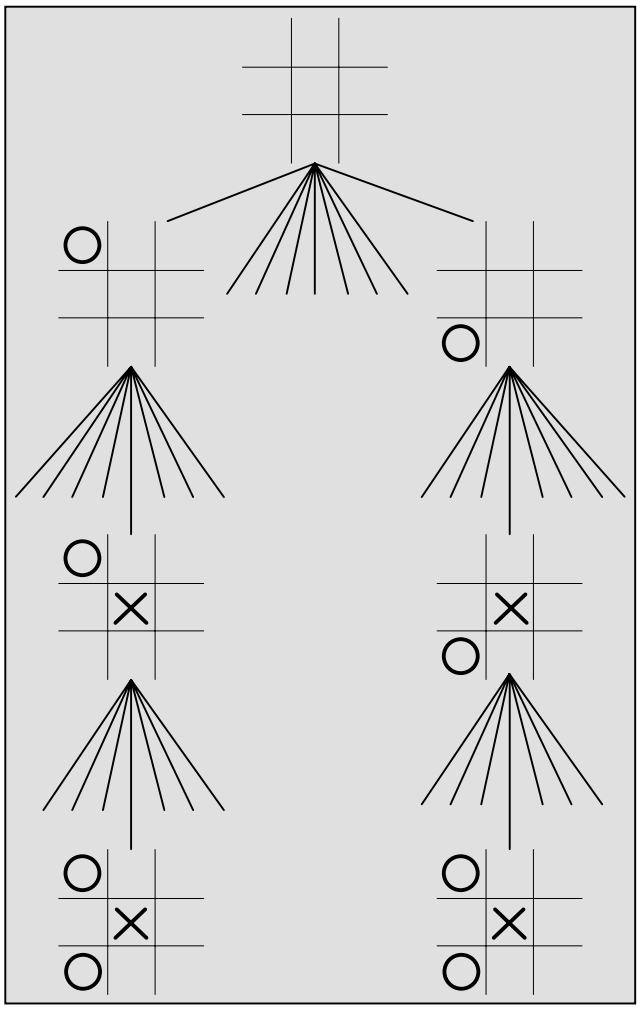


Figure 36: Very limitid part of a tic-tac-toe search tree

To be more specific then the theoretically number of games in tic-tac-toe is bounded by 9! = 362,880, since there at each move are as many options as there are empty squares. It should however be emphasized that this is an upper limit as games may end early because a player has won. This number can be compared to the number of legal board states that is bounded by 3^9 = 19,683 since each square can have three

possible states (X, O and empty). Again this is an upper bound as the number of X and O must either be equal or have a difference of one in favor of the starting player as well as board created after one player has won will never be created. Although non of the presented numbers are exact, it does show that the complexity of considering boards, as the goal directed learning method does, is considerably lower then considering possible games.

To have some form of comparison of the knowledge base size when implementing and testing the model in the domain of tic-tac-toe an algorithm creating a tic-tac-toe search tree was implemented. Every unique board was then identified. The upper bound was then identified to be 6,046 different boards. This was however without taking into account that board created after one player had three in a row would never be created. Adding the feature such these boards was not generated made the final amount of different boards drop to only 5,478. This is without taking symmetry into account.

For complex games the amount of theoretically possible boards will however still be large. It is however believed that the majority of board patterns is only theoretically and will never be played in practice. As a result this model is still believed to be practically possible. For example one of the identified patterns in the four example games, is presented in Figure 37.

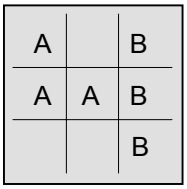


Figure 37: An identified pattern

At is should be evident subsequent combinations will never happened in practice. In this specific case 12 board (not 15 as some are identical) combinations will be cut off as illustrated in Figure 38.

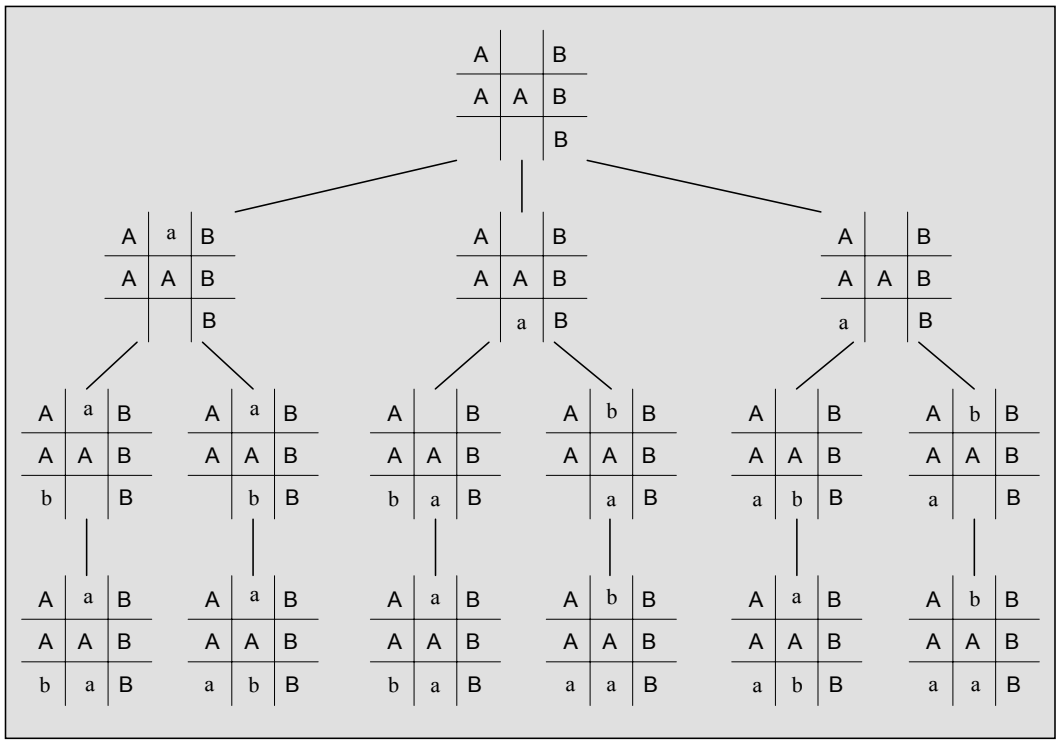


Figure 38: Tree illustrating board patterns that will be cut off

This is however a very specific case. But what it should illustrate is that one board has the power to cut of a whole range of other boards.

Let us consider another case illustrated in Figure 39.

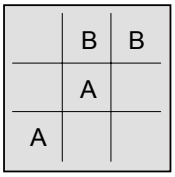


Figure 39: Pattern with power to cut of a whole sub tree

B has just created this pattern and it is *A's* turn. What interesting with this pattern is that it from *B's* point of view is such a bad pattern that has it first been played once, it should never be created again. From *A's* view point it is however and excellent pattern as the way to victory is rather easy. But it is not *A* that has the power to create this pattern, it is *B*. What *A* should do now is simply to but its piece in the upper left corner and it will have two possibilities of a victory as illustrated in Figure 40.

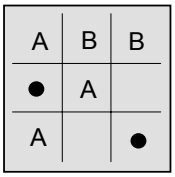


Figure 40: A has two winning possibilities, B will loose no matter what

As a result the board pattern illustrated in Figure 39 should only create a very few sub patterns. A personal belief is that in most games the range of patterns must be of such a type, that has they first been played once or a few times, they will be identified as very poor for one of the players and never be played again. As a result each of them will cut of a large sub tree of patterns and as a result the model is still practical possible. This is however one of the issues that is to be tested during the project.

But further methods can be used to keep the amount of boards down. In more complex games explicit coding of some of the simplest domain rules could be done to guide the agent from playing completely irrational strategies, and in that way ensure that completely irrational boards will be ignored and only somewhat rational boards will be played. This will however not be done when implementing and testing the method in the domain of tic-tac-toe.

Although and exact size of a knowledge base for complex games is hard to estimate section 4.1 describing Deep Blue hold one fact that might be of some use. It is described that Deep Blue holds a database of 700,000 grandmaster games. If it is estimated that each game in average has 90 different board positions, that gives a database of 63,000,000 boards. Many of these boards will undoubtedly be the same such the exact number of unique boards will be much lower. Comparing this to the around 2,758,547,353,515,625 boards to examine for a ten move look ahead in chess using a game tree, and the goal directed learning model starts to look rather good in relation to complexity.

Given the above descriptions the goal directed learning model is believed to be a practically possible and highly usable solution in even complex games.

5.8 Conclusion on analysis

In this section a definition of intelligence was given as an entities ability to achieve goals. Further more the concept of learning and knowledge was identified to be of high importance to be able to call (in the author's opinion) something intelligent. This analysis of intelligence along with requirements and knowledge obtained in previous chapters; assisted in the process of modeling a method for learning the game of tic-tac-toe. The model is general and should have the ability to be easily applied to a range of other game domains. The method is refereed to as goal directed learning and is a model that fulfills requirements given in 5.2. A possible problem concerning the size of the agents knowledge base in complex domains was also considered, it is however believed that will not be a problem in practice. Additionally one (of probably many) method of modeling an agent by use of classical AI techniques, using neural networks and genetic algorithms, was presented.

6. Prototype implementation

This chapter will give on overview on the general architecture of the developed prototype. This prototype is to be used for test and evaluation of the solution model previously described in 5.6. An in depth description of the implemented solution models can be found in chapter 7, as this chapter mainly is to give an introduction to the overall tic-tac-toe prototype program.

Sections of this chapter are:

- Design criteria
- Architecture
- Program
- Conclusion on prototype implementation

6.1 Design criteria

To be able focus on the most important aspects of a prototype that can test the goal directed learning model, relevant design criteria [3] is evaluated in Table 22.

Criteria	**Importance**
Useful, the adaptation of the system in the end-user environment.	The main objective is not to make a useful system for potential end users. As such an intuitive an easy to use graphical user interface can be ignored. Usefulness for anything else then the evaluation of the solution model is therefore considered to be of less importance.
Secure, securing against unwanted access to the systems data and facilities.	Considered irrelevant.
Efficient, the exploitation of facilities in the technical platform.	As a knowledge base structure, there might become relatively large, are to be updated and queried frequently, efficiency of its design is to be of importance. However as the program is only to be a prototype, top notch efficiency is not essential.
Correct the fulfilment of the system requirements.	As the prototype is to be used for testing and evaluation of a solution model, correctness of the implemented model in relation to the theoretically suggested model, is considered to be of high importance.
Reliable, the fulfilment of the demanded functionality with the wanted precision.	Minor bug and glitches in the “edges” of the prototype can be allowed. However in its core the system should be reliable as an unreliable system might result in failure to test the solution model.
Maintainable, The cost of localising and correcting errors in the running system.	As changes and tweaks of the agent, and its solution model, might be necessary, maintainability of this part of the program is of high importance.
Testability, the cost of testing the system in coherence to the requirements.	As the whole concept with the prototype is to test a solution model, testability is of major importance.

Flexible, the cost of changing the running system.	Flexibility in changing adding and removing tic-tac-toe playing agents for evaluation and improvements is important.
Understandable, the problem of getting an overview and understanding the system.	As the system as such has no end users this is considered to be of minor importance. Understand ability of the design is also of minor importance as the program only has one developer.
Reusable, the cost of moving the system to other related systems.	As implementation is a prototype, that potentially is to be the foundation for a larger project, it would be desirable to be able to reuse code. It is however not of great importance
Moveable, the cost of moving the system to other technical platforms.	Ability to move the program to another platform is not considered to be of importance.
Integrative, the amount of problems emerging when merging the system with other systems.	The system is to be an isolated system and as a result ability to integrate the system is not important.

Table 22: Design criteria for prototype

More generally the prototype is to be useful for test and evaluation of machine learning techniques in the domain of tic-tac-toe. The prototype shall be viewed as an application which main purpose is to evaluate the goal directed learning model, which is the solution model presented in the previous chapter. It is not to be targeted against a group of users, although the implementation of a tic-tac-toe game that can be published to the public will be pursued if possible.

More specific the requirements are to implement a tic-tac-toe game where it is possible for humans to play against, both a self trained and a human trained agent. The human training of an agent should simply be done automatically with out any special notice when playing against it. For evaluation purpose an additional agent relying on a neural network and genetic algorithm will also be implemented. This is to be able to test and evaluate the goal directed learning model against an already existing model for learning.

6.2 Architecture

This is a rather technical section explaining the architecture of the tic-tac-toe prototype, for a more general explanation of the program reefer to section 6.3. The program has been implemented in the object oriented programming language C#. The unified modeling language (UML) will therefore be used to describe the connections between the programs classes. It should be noted that the architecture presented here is the final architecture. This is because the final architecture resembles the initial planed architecture almost exactly, except for some very limited interface changes and additions.

6.2.1 Tic-tac-toe game

The central part of the prototype program, is the game of tic-tac-toe. A UML diagram describing the architecture of this part of the application is illustrated in Figure 41.

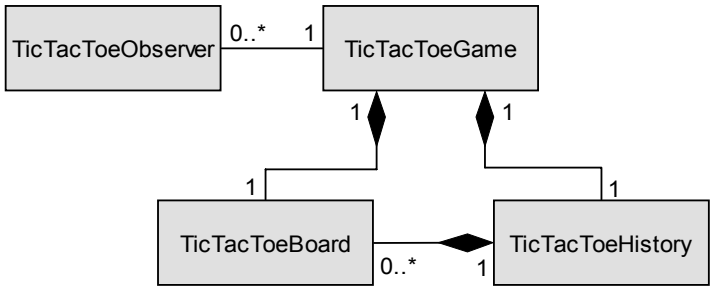


Figure 41: UML of tic-tac-toe game classes

The main interface to the tic-tac-toe game is the class *TicTacToeGame*. It aggregates the class *TicTacToeBoard,* which is a representation of the current board layout for a game in progress. *TicTacToeHistory* holds a history of previous boards that has been played in the game currently in progress. *TicTacToeObserver* is a class representing an object or entity that wish to observe or influence the game in progress, this is typically a player. Below important methods for each class will be presented.

TicTacToeBoard

```
public void Clear()
public bool Put( int column, int row, char piece )
public char Get(int column, int row )
public int PieceCount()
```

Pieces are simply represented by the characters X and O which can be put on the board. The method *get(…)* returns a type of piece on a specific square. Additionally the board can be cleared and the amount of pieces currently on the board can be obtained.

TicTacToeHistory

```
public void SetStartPiece( char piece )
public void SetWinner( char winner )
public void AddBoard( TicTacToeBoard board )
public char GetStartPiece()
public char GetWinner()
public TicTacToeBoard GetBoard( int idx )
public int Count()
public void Clear()
```

When the class *TicTacToeGame* starts a new game the piece starting the game is set with the method *SetStartPiece(…).* Every time a new board is created it is added to the history until the game is over. If the game has a winner this is also set. Additionally when a game is over there are get methods to obtain the different boards as well as the starting player and winner.

TicTacToeGame

```
public void NewGame()
public char GetWhoseTurn()
public char GetStartingPlayer()
public bool PutPiece( int column, int row )
public TicTacToeBoard GetBoard()
public bool GameOver()
public char FindWinner()
public TicTacToeHistory GetHistory()
public TicTacToeBoard[] GeneratePossibleBoards()
public void AttachObserver( TicTacToeObserver observer )
public void DetachObserver( TicTacToeObserver observer )
```

This is the main interface class representing the game of tic-tac-toe. Most methods should be pretty self explaining. *NewGame()* starts a new game. *GetWhoseTurn()* returns whose player turn it is to make a move. *GetStartingPlayer()* return what player started the current game in progress. *PutPiece(…)* puts a piece on the board. *GetBoard()* returns a reference to the *TicTacToeBoard* object, representing the board in the game. *GameOver()* returns true when the game is over and *FindWinner()* returns the winner of the game if any. *GetHistory()* returns a reference to the *TicTacToeHistory* object. *GeneratePossibleBoards()* will generate and return an array of possible next boards from the current board. *AttachObserver(…)* and *DetachObserver(…)* is used to add players or observers of the tic-tac-toe game. This is explained below.

TicTacToeObserver

```
void Notify( TicTacToeGame game )
```

This class or interface just has one method *Notify(…)*. All classes that need to access, observe or play the tic-tac-toe game, must inherit this class and then be added to the *TicTacToeGame* by its *AttachObserver(…)* method. Every time there is a change in the state of *TicTacToeGame,* it will call *Notify(…)* on all the observers (players) added to the game and pass itself along as a parameter. This design pattern is also often referred to as observer observable or publisher subscriber and ensures a flexible design where different players (agents) can be easily added and removed.

6.2.2 Goal directed learning agent

The most interesting part of the program is the goal directed learning agent. This is the implementation of the tic-tac-toe playing agent that should be able to learn the game of tic-tac-toe by implementing the goal directed learning model and thereby learn by the outcome of the games. It is this part of the program that is to be tested and evaluated. UML is illustrated in Figure 42.

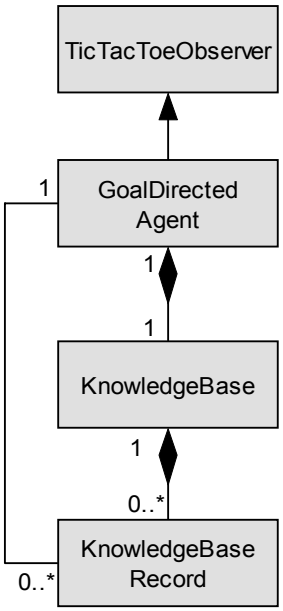


Figure 42: UML of the goal directed learning agent

For the agent to be able to access the *TicTacToeGame* it must inherit from the *TicTacToeObserver* class. For learning and decision making the agent relies on *KnowledgeBase* containing *KnowledgeBaseRecords*. For performance purpose the knowledge base relies on a hash table for storage of records using board patterns as unique key. Board patterns are encoded simply as a string with nine characters.

KnowledgeBaseRecord

```
public KnowledgeBaseRecord( string pattern, int wonA, int wonB, int draws )
public string GetPattern()
public int GetDraws()
public float GetWonA()
public float GetWonB()
public int GetPlayedTimes()
public bool UpdateRecord( TicTacToeKnowledgeBaseRecord record )
```

For each different board pattern the agent identifies when playing, it creates a *KnowledgeBaseRecord*. A specific board pattern is when stored in the knowledge base simply encoded as a string. Besides basic get methods, a *KnowledgeBaseRecord* contains a method to update itself with data from another *KnowledgeBaseRecord*.

As it might has been observed the *KnowledgeBaseRecord* does not exactly match a record implementing the model explained in 5.6. During testing and evaluation it was however decided to slightly change the structure of the knowledge base. This is explained in details in section 7.4 where the final solution model is explained.

KnowledgeBase

```
public void AddRecord( KnowledgeBaseRecord record )
public KnowledgeBaseRecord GetRecord( string pattern )
public int Size()
public void ClearKnowledgeBase()
public static void SaveKnowledgeBase( KnowledgeBase kb, string file )
public static void LoadKnowledgeBase( KnowledgeBase kb, string file )
public static void MergeKnowledgeBases( KnowledgeBase from, KnowledgeBase to )
```

Besides basic methods to add and get records from the knowledge base three static methods has been added to the class. *SaveKnowledgeBase(…)* and *LoadKnowledgeBase(…)* will save and load a knowledge base in a file. *MergeKnowledgeBases(…)* will take one knowledge base and merge this knowledge into another knowledge base. This is one of the powerful features of containing the obtained knowledge in a domain understandable way instead of arbitrary weights within a neural network. This feature will make it possible to create a whole range of agents each obtaining their own isolated knowledge, and then at some point integrate all this knowledge into just one knowledge base in an agent.

GoalDirectedAgent

```
public void SetPlayingPiece( char piece )
public void Notify( TicTacToeGame game )
public void ResetKnowledge()
public void SaveKnowledge( string file )
public void LoadKnowledge( string file )
public void MergeKnowledge( KnowledgeBase mergeFrom )
public KnowledgeBase GetKnowledgeBase()
```

This is the key class in this project. This is the class that is to contain the solution model used for learning. Changes concerning how an individual agent learns are to be isolated here. *SetPlayingPiece(…)* tells the agent what piece it should be playing in a game. *Notify(…)* is the method the *TicTacToeGame* is responsible for calling every time a change is made in the state of the game. *Notify(…)* then implements all the logic for selecting a move to make in the case it is the agents turn, as well as the logic for learning when a game is over. Additionally methods necessary to reset, save, load and merge knowledge exists on an agent.

6.2.3 Neural network agent

For evaluation purpose an additional agent relying on a neural network are also implemented. For a description of neural networks refer to section 2.6. This agent has been implemented to obtain the ability to compare the goal directed learning method against an already existing method for learning. UML for this agent is illustrated in Figure 43.

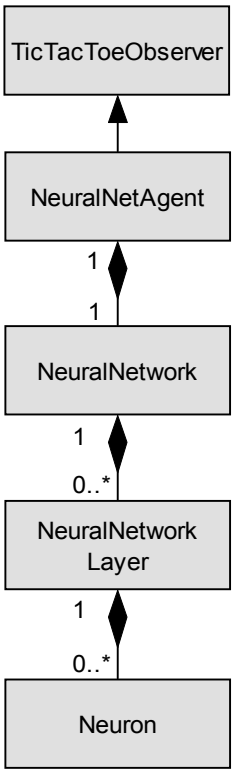


Figure 43: UML of agent relying on a neural network

Also the *NeuralNetAgent* must inherit from the *TicTacToeObserver* to be able to play the game of tic-tac-toe. A *NeuralNetAgent* aggregates a *NeuralNetwork* responsible for learning and decision making. This network is made up of *NeuralNetworkLayers* containing *Neurons*.

Neuron

```
public float[] Weights;
public float Output;
```

Neuron has no methods; instead it has just two public variables. That is an array of weights into the neuron and the output value from the neuron. The calculation of the output from a neuron is done in the *NeuralNetwork* class.

NeuralNetworkLayer

```
public Neuron[] Nodes;
public NeuralNetworkLayer( int dimension, int neuronInputs )
```

Also a rather simple class, it has one public variable which is an array of the *Neurons* in the layer. The class constructor *NeuralNetworkLayer(…)* will initialize this array corresponding to the given dimension and inputs for each neuron.

NeuralNetwork

```
public float[] GetWeights()
public void PutWeights( float[] weights )
public void Train( float[] data, float[] target, float maxMeanSquareError, float learningRate )
public void Run( float[] data, float[] output )
```

The method *Train(…)* will train the network using the back-propagation training algorithm described in section 2.6.4. *Run(…)* will run data through the neural network to obtain the networks output. *GetWeights()* will return an array of all the weights in the network, *PutWeights(…)* will replace these weights.

NeuralNetAgent

```
public void SetPlayingPiece( char pieceToPlay )
public float[] GetWeights()
public void PutWeights( float[] weights )
public void Notify( TicTacToeGame game )
```

SetPlayingPiece(…) tells the agent what piece it should be playing in a game. *Notify(…)* is the method the *TicTacToeGame* is responsible for calling every time a change is made in the state of the game. *Notify(…)* then implements all the logic for feeding the *NeuralNetwork* with input and obtaining the output to select a move to make in the case it is the agents turn. *GetWeights()* will return an array of all the weights in the agents contained network, *PutWeights(…)* will replace these weights.

6.2.4 Co-evolution

One central part of the testing and evaluation is to see if an agent is able to learn the game simply by playing other agents (or itself), and not only by playing humans. Both for the goal directed agent and for the neural network based agent co-evolution as been implemented. In the case of the agent based on goal directed learning, this form of learning will however often in this paper be reefed to as training or self play instead of evolution. This is because the term evolution is, in the author's opinion, more suitable for a learning model based on genetic algorithms. UML can be seen in Figure 44.

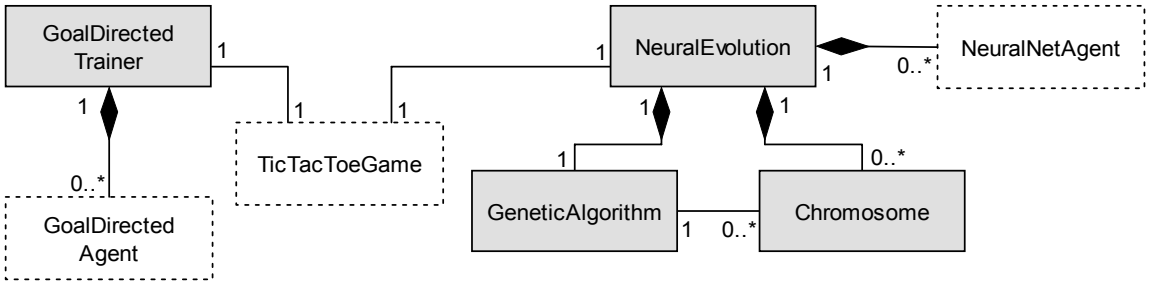


Figure 44: UML illustration of classes responsible for evolving tic-tac-toe playing agents

GoalDirectedTrainer is just one class, and is responsible for learning the *GoalDirectedAgent* the game of tic-tac-toe. *NeuralEvolution* uses a genetic algorithm in the hope that a *NeuralNetAgent* agent capable of playing reasonable tic-tac-toe will evolve. For an in depth descriptions of genetic algorithms refer to section 2.5.

GoalDirectedTrainer

```
public void SelfPlayTrain( GoalDirectedAgent agent, TicTacToeGame game, int count )
public void TournamentTrain( GoalDirectedAgent agent, TicTacToeGame game, int
        sessions, int opponentsCount )
```

GoalDirectedTrainer is responsible for training a goal directed learning agent. *SelfPlayTrain(…)* will let the agent play itself in a number of games. The same agent will simply be responsible for selecting moves for both sides. *TournamentTrain(…)* will use a range of other agents to learn the agent to play tic-tac-toe and at the end merge all this knowledge into one agent.

Chromosome

```
public float Fitness;
public float[] CromosomeBits;
```

Chromosome has just two public variables. That is an array of floating point values representing the chromosome bits to evolve and another floating point value representing the fitness of that array. The *ChromosomeBits* are to be the weights in the neural network.

GeneticAlgorithm

```
public void Evolve( ref Chromosome[] population )
```

The method *Evolve(…)* takes an array of *Chromosomes* and evolve it to the next generation, for an in depth description of how this method works, reefer to section 2.5.1.

NeuralEvolution

```
public void Learn( NeuralNetAgent agent, int generations, int populationSize )
```

The method *Learn(…)* will create a population of agents that will play against each other in a series of games. Depending on an agents performance a fitness score will be assigned to be used when evolving agents using the genetic algorithm

6.2.5 Complete architecture

The complete UML architecture can be seen in Figure 45. In the center connecting all the pieces is the *TicTacToeGUI* which is the graphical user interface of the program. This class also needs to inherit from the *TicTacToeObserver* such it can get access to the game and draw the board on the graphical user interface, as well as representing the human player.

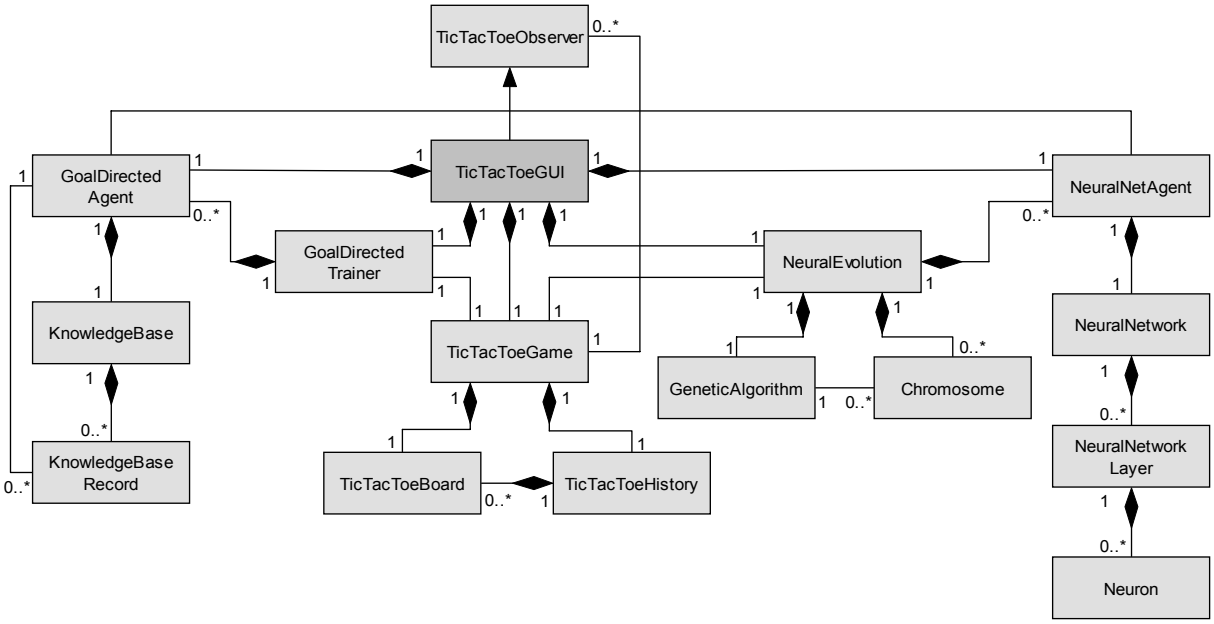


Figure 45: Complete UML of tic-tac-toe prototype

6.3 Program

For a more understandable description of the program, this section will explain its user interface and general capabilities. This is because it is the hope that the reader of this report will have an easier time understanding chapter 7 concerning testing and evaluation, if a general and more easily understandable presentation of the program has been given. It should be noted that the program presented here is the final program, initial testing was however possible and started, as previously mentioned, before all presented features were fully implemented. To run the program the Microsoft .NET framework version 1.1 is required.

When launching the program, the user interface presented in Figure 46 will show up.

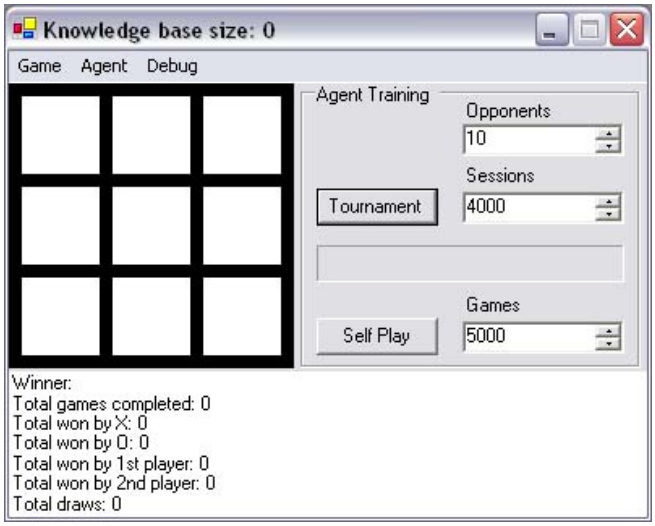


Figure 46: User interface for tic-tac-toe prototype

To the left the tic-tac-toe board can be seen. In the bottom some general statistic about played games can be seen. In the top the size of the agent's knowledge base can be seen. This is how many different board layouts the agent currently has been presented with and had the ability to learn from. The panel to the right is used to automatically train (co-evolve) an agent by playing either itself or other agents. The tournament button will let the agent play a selected number of opponents in a series of games. The self play button will let the agent learn by playing itself. It will simply select moves for both X and O. A special feature for training the agent by playing humans does not exist, as this will be automatically done when playing against it.

The items contained in the main menu can be seen in Figure 47

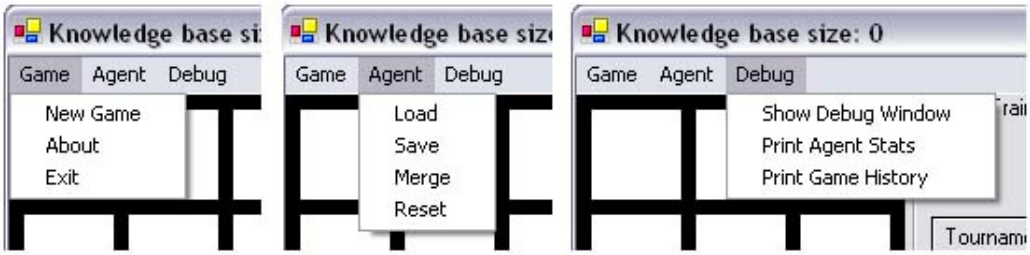


Figure 47: Main menu items

In the *Game* menu it is possible to start a new game. Some general info about the program is also available as well as the possibility to exit the program. In the *Agent* menu it is possible to load and save the agents knowledge from and to a file. The *Merge* feature will merge a saved knowledge base into the agent. *Reset* will delete all knowledge from the agent. The *Debug* menu contains some features used during development of the agent. The whole knowledge base of an agent can be written out, as well as the history of the game in progress.

When the program has been launched it is possible to begin playing against, and train the goal directed agent. Initially the agent has no knowledge how to play the game in a clever way. Either loading an agent or training it will therefore be advisable if a decent opponent is wanted. The human player will always play X and the game is started simply by pressing one of the squares on the board. Subsequently the agent will place a piece in one of the free squares. This continues until the game is over and the agent updates its knowledge base. After the game is over, selecting new game will start a new game where the agent starts out by placing an O on the board. The game has been created in such a way that the two players will alternate to start the game.

From this interface it is not possible to play against a neural network based agent. To play against this agent, the program must be executed in the following way:

[Application].exe nn [generations] [population size]

nn tells the application it should use the neural network based agent instead of the goal directed agent. Population size is parameters telling the amount of agents to use when evolving, generations is the amount of times the agents will evolve using the

genetic algorithm. In the case where the application name is TicTacToe.exe the following example will evolve an agent using 10 agents for 2000 generations.

TicTacToe.exe nn 2000 10

Many of the features on the graphical user interface specifically related to the goal directed agent will be disabled in this mode.

6.4 Conclusion on prototype implementation

A tic-tic-toe prototype capable of testing both a neural network based agent evolved by a genetic algorithm, as well as the goal directed agent has been successfully implemented and integrated into the game of tic-tac-toe. The architecture and design is flexible and modifications and changes to solution models implemented by agents can be easily changed for evaluation purpose.

7. Models, test and evaluation

The process of finding the exact solution model for the goal directed agent has been an iterative process starting with the initial solution model explained in section 5.6. This model has been implemented, tested and evaluated. Experience obtained from the testing and evaluation of this solution model then laid the foundation for an improved solution model that then again was implemented, tested and evaluated. Again experience obtained from this improved solution model laid the foundation for the implementation of the final solution model.

This section will however start out by describing the test and evaluate of the neural network based agent, this is to have some form of comparison to an existing models used for machine learning in games.

It should also be noted that in all cases, training of an agent by playing humans quickly proved to be a time consuming task. As a result the focus on test and evaluation is manly on self taught (co-evolved) agents. However as it is the same basic model (in the goal directed agent) there is used whether the agent is playing itself, another agent or a human player, it is quite evident that if it can learn by plying itself or another agent, it can also learn by playing a human opponent.

It should also be noted that the models primarily has been tested and evaluated simply by playing against it. However, since tic-tac-toe is a game that if played rational by both players will end in a draw, evaluation in this fashion can be done effectively.

Sections of this chapter are:

- Neural network based agent
- Initial solution model
- Improved solution models
- Final solution model
- Future work
- Conclusion on models, test and evaluation

7.1 Neural network based agent

Before describing and evaluating the goal directed agent, the neural network based agent will be described and evaluated. This is to gain the ability to refer to, and compare the different goal directed models, to this model.

7.1.1 Design

The design of the model used in the neural network based agent differs a little from the model earlier described in section 5.4. In this earlier described model the network had 9 input and 9 output nodes and one hidden layer containing 18 nodes. It was then feed with the current board layout. The output node with the highest value was then were the agent would put its piece. In the implemented model the network only has 1 output. It does however still have 9 input and 18 nodes in one hidden layer. It is then feed with the possible next boards. The board with the highest score is then selected

to represent the agents move. The input to the network is nine values either being -1 representing an opponent, 0 representing a free square or 1 representing one of the players own pieces. This model is illustrated in Figure 48.

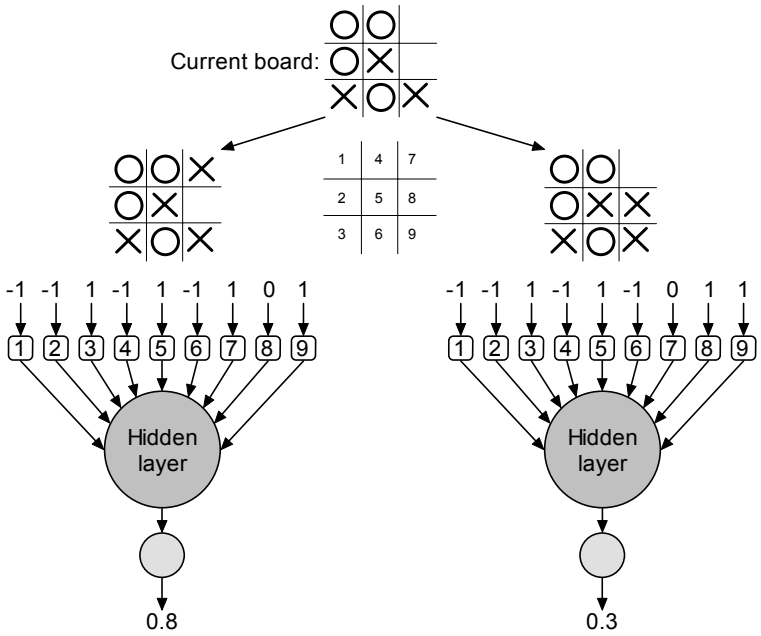


Figure 48: Neural network model used in agent

Testing and evaluation has only been done with the process of co-evolution as decent testing and evaluation by playing humans would take to much time. The genetic algorithm responsible for evolving the neural network weights was set to have a crossover rate of 70% and a mutation rate of 0.1%. During match play the whole population of neural net based agent was set to play two matches against each other in every generation, one game where each of them was starting the game. Fitness score was assigned such that 2 fitness points was given for a victory, 1 for a draw and 0 for a loss. For this example a population of 10 agents evolving for 30,000 generations was created. The process of evolution with these settings took around twenty minutes on a PC with 512 MB of memory and an AMD-Athlon processor running at 1.1 gigahertz. Twenty minutes might be argued to be a very short amount of time. However when reading section 7.4 it will be evident that twenty minutes is quite some time, compared to the final goal directed solution model.

7.1.2 Evaluation

Playing against the agent, quickly revels that its performance is rather poor and pulling a victory as human is extremely easy. That being said it does also seems to have learned something, and it does seem to perform slightly better then an entirely random based agent. The agent has learned the importance of owning the center square and when starting the game it will select it as its opening move. If the human player instead is starting the game, and not taking the center square, the agent will in its first move put its piece there. The agent has also learned that the corner squares often are considered better squares then the middle top, middle bottom, middle left and middle right squares. This is however also a weakness as it in many situations rather stupidly will go for one of the corner squares in cases where it is a completely

irrational move. This is illustrated in a rather typical, but not isolated example in Figure 49, where the neural network agent is playing O and starting.

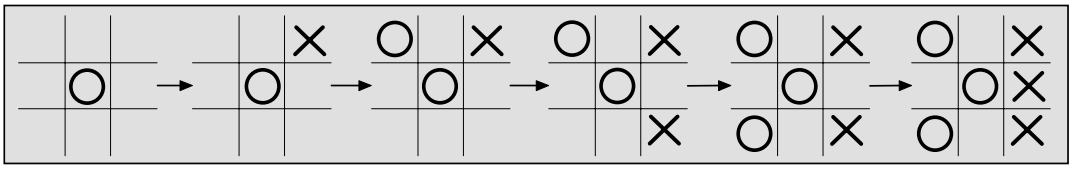


Figure 49: Example game against neural network based agent

It might be that a population of 10 agents evolving for 30,000 generations is not enough. It could also be that changing some values like crossover or mutation rate as well as the way fitness score is assigned might yield better results. It might also be that using another step function then the sigmoid within the neural net could improve results. None the less it does show that neural networks and genetic algorithms is not a straightforward process that can be used to in a quick way to train an agent capable of playing even the simplest form of games on the same level as a typical human. But it does give some small indications of learning.

7.2 Initial solution model

The original solution model is the one explained in section 5.6. For an in depth description reefer to this section, else the model will be briefly summarized here.

7.2.1 Model

In this model the agent has a knowledge base record for each identified board pattern. Each record contains 3 values besides a string representing the board pattern. The tree values are *Count* which is the amount of times the board has been represented in games the agent has been playing, and *ScoreA* and *ScoreB* which represents a score for that board for each of the two types of players the agent can play. *ScoreA* is a score if starting the game; *ScoreB* is the score if not starting the game. Each knowledge base record for a specific board pattern can then be illustrated like:

Pattern	Count	ScoreA	ScoreB

Let us consider a specific example using the board illustrated in Figure 50.

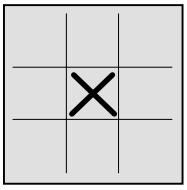


Figure 50: A tic-tac-toe opening board

Remember that a winner board was given 2 points and a looser board given -1 point. Then if the board illustrated in Figure 50 has been represented in four games, and in

two of these games the starting player has won, and in one of the games the starting player has lost. Then the record for this board would look like.

Pattern	Count	ScoreA	ScoreB
----A----	4	3	0

Then when the agent is to start a game and it consults its knowledge base, it will for this board, calculate it to have a score of 3 divided by 4, which is 0.75. The score is divided by *Count* to normalize the score to a value between 1 and -1. This will be done for all possible boards. The board with the highest score will then be selected. Untried boards are initially given the score 0. In the case that there is more then one board with an equal highest score, a predefined, strategy is played.

7.2.2 Evaluation

The very first testing of this model was by playing and training the agent manually. First impression was very positive. The agent seemed to learn, and seemed to be repeating the play leading to victory for a human player. It was however quickly evident that it would take quite a lot of time to train it to level (if at all possible) where success could be concluded.

The feature of self play, such an agent was able to select moves for both sides, and thereby play it self, was therefore quickly implemented and tested. This was however a very disappointing experience, after 100 games, 1,000 games or even 10,000 games, the agent did not show the slightest indication of learning at all. Closer examination of the problem revealed that the agent would start out by playing 22 different games, and from that point begin repeating the exact same game over and over again. At that point the total knowledge base size, which is the amount of different boards the agent has learned from, was only 85 boards.

What happens is that each of the two players (the starting player and non starting player) quickly identifies boards that either gets a positive score or a negative score. Boards with a positive score will be selected and boards with a negative score avoided. However at the point where the first game is ending in a draw, none of the scores in the knowledge base will change, except that the *Count* for a board will increase. This means that the exact same game as the previous one most likely will be played again; as the score for that board only will be a little lower than in the previous game. In the case that another board gets a higher value this will be played instead, but if this game also ends in a draw the *Count* for boards represented in this game will also increase without *ScoreA* or *ScoreB* changes. Ultimately all boards will end with a score of 0 or less. Each player will then simply play the default strategy from the boards with a score of 0, game after game.

To summarize then this model seems to give even worse results then the neural network based solution when an agent has been trained by self play. By playing humans, it however gives indication of some form of learning. It is also possible to get the agent out of the deadlock of the same played game by playing some games against it as human, it will however short after be trapped in a new game that will be played over and over again. However as the agent is not able to learn by solely

playing itself and requires quite a lot of human interference, this model is not satisfying.

7.2.3 Minor modifications

The experience described above lead to some modification of the model. First of all was the play of a default strategy replaced with a random strategy in the case that there was more then one board with an equal highest score. To increase the chances of boards getting an equal score the result of the calculation was also rounded at three decimals. The score for a win was also lowered from 2 to 1 point. Finally was the score for an untried board raised from 0 to 1 to force the agent to play a new strategy the first time it has the opportunity for it.

Once again results are very disappointing. After 20,000 training games the agent is still easy to beat, although it like the neural network based agent seems, in most cases, to have learned the importance of the centre and corner squares. Once in a while it might play some intelligent looking way, but such a game is probably based more on luck then learning. The model also seems to have a tendency to get stuck in a smaller group of games (typically 5-10) that for a long time is played over and over until the agent finally breaks out of this pattern, just to get stuck in a new group of patterns short after.

Although not an exact measurement on how well a model is performing the, size of the knowledge base gives some indication on how much has been learned. The growth of the knowledge base in relation to the amount of games played, also gives some indication on the models ability to vary its play and not play the same strategies over and over again. A typical example on knowledge base size and growth can be seen in Table 23.

Games played	Knowledge base size
10	67
100	441
500	453
1,000	453
5,000	498
10,000	536
20,000	557
40,000	557
80,000	557

Table 23: Knowledge base size after an amount of training games

As should be evident from Table 23, the model seems to very quickly letting the agent play the same group of games over and over again; such further knowledge with more training time cannot be obtained.

How to exactly evaluate this model is a little hard. It seems to be performing on the somewhat same level as the neural network and genetic algorithm based model. But as this model also in the author's opinion is performing poorly, this solution model is not satisfying in its current form. It does however seem to show some evidence of

learning when playing humans. But it then relies on the human player's ability to vary the games, such that it will not get stuck. However because of time concerns, an in depth test and evaluation of a human trained agent has not been performed.

7.3 Improved solution models

The problem with the initial solution model is that it very quickly concludes how good a board is, and this conclusion might be completely wrong. Consider the example where a rather good board might be played for the first time. Ultimately the player playing the board looses because an unlucky random move is made at a later point. The score for that board will then be calculated to be -1. The board will in other words always be avoided when the agent has the possibility to play the board. Now consider another situation where a very bad move by the same player by some later lucky random move leads to victory. This board will then have the score 1 and be played the next time there is a possibility for it. In subsequent games the player might play this board over and over again and loose. However after 1000 games with 1 victory and 999 defeats its score will be -0.998, which still is higher then the board that has just been played once and lead to a loss. The model will in other words during self play quickly cut of a group of pattern, and concentrate on another group. This group will ultimately be played again and again until a little group of situations ultimately leading to a draw has been identified. These games will then be played over and over again.

It is evident that to be able to let the agent learn by self play the model must somehow make sure that the conclusion of boards, is not made to quickly. A rather experimental but successful formula for awarding scores for a winner pattern and looser pattern is therefore tried out. Except for the changes specifically mentioned here, the model is equal to the minor modified initial version explained in section 7.2.3.

7.3.1 Changes

Instead of simply awarding 1 point for a win and -1 point for a loss, points is added using two formulas. The amount to update the score for a win is calculated with the formula:

$$DeltaScoreWinner = \frac{10}{10 + Count}$$

Count is number of times the board has been played so far. So the more a board has been played, the less its score for a win is updated.

The amount to update the score for a loss is calculated with the formula:

$$DeltaScoreLooser = -\left(1 - \frac{10}{10 + Count}\right)$$

Again *Count* is the number of times the board has been played. This formula will make sure that the more a board has been played, the more a loss will be punished.

Finally to avoid the agent of getting stuck and play the same group of drawn games over and over again, a draw will for a player starting the game, punish a played board by half the *DeltaScoreLooser,* while the score for a draw for the player not starting the game, will reward played boards by half the *DeltaScoreWinner*. In this way the agent will when starting, continually try out new patterns, and not accept a draw to be satisfying.

To better explain the model, consider an example where the opening move illustrated in Figure 51, has just been played for the first time.

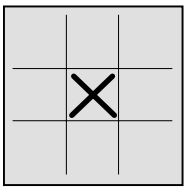


Figure 51: Opening move

Let us in this example say that the starting player X looses this first game. The following record will then be added to the knowledge base:

Pattern	Count	ScoreA	ScoreB
----A----	1	0	1

Although the board has lead to a loss for the starting player, it still has a rather good chance of selection as its score will be calculated to 0. In the case the starting player looses again playing this board, the record in the knowledge base will be changed to:

Pattern	Count	ScoreA	ScoreB
----A----	2	-0.091	1.909

Let us for the above example include the calculations for clarity. It was the player not starting the game that won so the value to update *ScoreB* with is found by:

$$\frac{10}{10+1} = 0.909$$

And the value to update the value for *ScoreA,* which was the player loosing the game is found by:

$$-\left(1-\frac{10}{10+1}\right) = -0.091$$

These values are then added to the values already contained in the record. The final score when the agent are about to start a game, and the score for that board is calculated, will then be -0.091 / 2 = -0.046. The move therefore still has a rather good chance of being selected at a later point. In the case that it in the third play will lead to victory for the starting player, the record will be updated to:

Pattern	Count	ScoreA	ScoreB
----A----	3	0.742	1.742

After two losses and one win, it will instead get a positive score of 0.742 / 3 = 0.247 and a chance of getting played again has increased considerably.

7.3.2 Evaluation

It very soon became clear that with this model, the agent was learning the game of tic-tac-toe very well. Training the agent for 5,000 games and it has become rather hard to beat. The first impression was actually that it was impossible. It however soon after became clear that it was not impossible if as human playing totally irrational against it. So when beginning to take chances and playing oddly, it was suddenly beatable. Two typical examples are illustrated in Figure 52. In both cases the agent is playing O and starting.

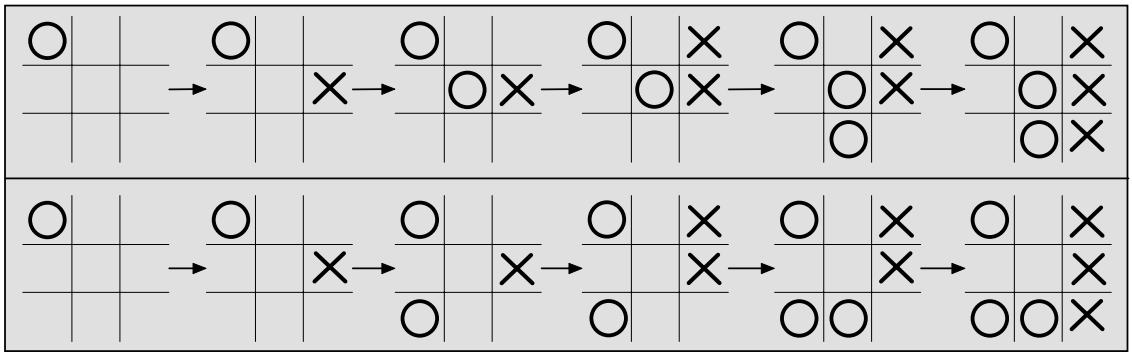


Figure 52: Odd strategies that could fool the agent

In both cases the agent has the chance to win the game. But for some reason it fails to do it and instead makes the game winnable for its opponent.

However an interesting observation can be done from Figure 52. It should be noted that the two example games illustrated are two subsequent games only separated by one game where human player X was starting. The interesting observation is that the agent in move number three is changing its strategy. It has in other words learned from the previous game that it might be clever to change strategy as the last time it played centre square it lost. Given enough time playing against a human the agent should also learn how to play claver in the above situations.

Training the agent up to 100,000 games without human interference, does however not help overcome this problem. It actually did not seem to have learned anything more when trained for 100,000 games compared to the agent 5,000 trained for games.

The reason for that might be that this model simply is so fast at trying out different solutions that at around 5,000 games played, it has learned how it in most situations is clever to play. This can be seen in Table 24 where it can be seen that growth of the knowledge base is non existent beyond 5,000 games.

Games played	Knowledge base size
100	574
250	1,189
500	2,103
1,000	3,276
2,000	4,397
3,500	4,844
5,000	4,847
10,000	4,847
50,000	4,847
100,000	4,847

Table 24: One example of games played and knowledge base size

The final knowledge base size of 4,847 in the above example is actually also rather large as the largest possible size in section 5.7 was found to be 5,477 possible different boards when not including the initial empty board. Still it seems that the claim made in 5.7 might be true. The claim was that some patterns will be identified as being of such a type, that has they first been played once or a few times, they will be identified as very poor for one of the players and never be played again, and as a result cut of a range of other boards. This does however has the obvious drawback that when a human player then starts to play these rather odd strategies, the agent implementing the solution model can be fooled because the agent suddenly then is caught in situations it does not recognize.

7.4 Final solution model

Although the previous model work very well in the domain of tic-tac-toe there is still room for improvements to increase it flexibility and improve chances for success in a more complex domain.

7.4.1 Model

First of all the layout of the knowledge base has changed. Instead of containing a score for each of the two types of (starting and none starting) player the agent can play. The knowledge base now contains how many times each pattern has lead to one of the domains possible goals for each of the two players (starting and non starting player). A knowledge base record can in this model then be illustrated as:

Pattern	WonA	WonB	Draws

The amount of times the board in all has been played is the sum of *WonA*, *WonB* and *Draws*. The record does not need to contain any records of lost games as the amount of lost games is equal to the amount of won games by the opponent. As such a theoretical value such as *LostA* would be equal to *WonB*.

Containing knowledge in this way should give some added flexibility. Once again knowledge bases can be merged without trouble. This was not possible without the creation of errors in the model described in 7.3 as this scoring system was not linear,

and a board played ten times with ten wins would get a higher score then a board played 100 times with 100 wins. Also because the knowledge base does not contain results of calculations, the exact way calculations are done, can be changed without throwing away an existing knowledge base.

Secondly the calculation of a value that controls the models desire to test board played a few times has been separated from the calculation of a boards score. The formula for calculating a board score is as follows:

$$BoardScore = \frac{1}{TotalCount} \cdot \sum_{goal} GoalCount \cdot GoalWeight$$

- *BoardScore* is the calculated score for that board.
- *TotalCount* is the total amount of times the board has been played. This is the sum of each of the possible goals.
- *GoalCount* is the total amount of times a board has reached that goal.
- *GoalWeight* is a weight for that goal and must be a value between 1 and -1.

We in other words take the amount of time a goal has been reached and multiply with a weight for have reached that goal. The weight must be between 1 and -1. The sum of all these goals is then multiplied with 1 divided by the total amount of times the board has been played to normalize and get a score between 1 and -1.

The score to control how willingly the solution is to test a board played a few times, is calculated by the formula:

$$TryScore = \frac{x}{x + TotalCount}$$

- *TryScore* is the calculated value indicating a score for playing a board.
- *x* is a value indicating how much a board should be tried out. A higher value will test boards more thoroughly.
- *TotalCount* is the total amount of times the board has been played.

BoardScore is a value between 1 and -1. *TryScore* is a value between 1 and 0. This final score of a board is then the sum of these formulas divided by the amount of formulas which is (*BoardScore* + *TestScore*) / 2.

Consider an example where X has started the game and O is about to calculate the score for the board illustrated in Figure 53.

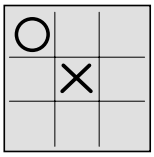


Figure 53: Board O is about to calculate score for

This board might have the following associated record in the agent knowledge base:

Pattern	WonA	WonB	Draws
B---A----	9	5	11

The weight for each of the possible goals is shown in Table 25. This is also the weights the final solution model is implemented with.

Goal:	Win	Loss	Draw (if starting)	Draw (not starting)
Weight:	1	-1	-0.5	0.5

Table 25: Weight for reaching a goal

Then the final score for the board for O, that is the player not starting the game, is calculated as:

$$\frac{1}{25} \cdot ((9 \cdot (-1)) + (5 \cdot 1) + (11 \cdot 0.5)) = 0.06$$

Additionally a score for how willingly the agent is to further test that board and try it out also needs to be calculated. In the tic-tac-toe prototype setting a value of 10 has proved to give rather good tradeoff between amounts of training games to play and intelligent play. The score for test the board is then calculated as:

$$\frac{10}{10 + 25} = 0.286$$

Finding the final score for the board, is then calculated to (0.06 + 0.286) / 2 = 0.173. The sum is divided by two because two different models to calculate the score of a board are used. In this model the final value is still rounded at three decimals to improve chances for different boards getting an equal score. In this case a random strategy is played. In more complex game domains it might very well be that the calculation of a *TryScore* should not be included at all or it at least should reach zero faster then in the tic-tac-toe domain.

A strength with the solution is that the calculation of *BoardScore* and *TryScore* is two different formulas helping in calculating the final score of a board. Both of these formulas can be added and removed from the agent's decision making calculation as desired. For example when playing against humans, it might be that *TryScore* should not be included. In the above case the board score will instead just be 0.06.

This model also makes it possible to add additional methods when calculating the score of a board. Only requirement is that its output for a board is a value in the range of 1 to -1. In more complex game domains it could for example be that there also existed a method that by some rules of thumb could, if given a board as input, give an output value in the range of 1 to -1 indicating how good this board is estimated to be. Then this rule of thumb method can in collaboration with the described solution model for learning help in the agent's decision making. One could imagine a whole range of these decision making methods co-exist within an agent. The solution model described here will then be the part of the agent responsible for memory and learning.

7.4.2 Evaluation

The performance of this solution is better then the model explained in section 7.3. After 5,000 training games the agent is showing indication of learning although it still is possible to beat especially if playing irrational and forcing it into odd strategies. Has the agent instead been trained for 100,000 games, it is becoming rather hard to beat. Again playing irrational as human it is possible to beat it, but it is not easily done and the agent is able to perform well in many of these odd situations. Also because the model still learns when it is playing against humans, irrational playing strategies identified by a human player to lead to victory, can not be repeated for many games before the agent has learned to handle the situation.

The models ability to try out new boards and thereby learn from new situations seems to perform better then the previous model. Where the previous model seemed to try out a lot of boards extremely quick, and then stopped playing new boards after 5,000 games. This final model instead seems still test new boards once in a while beyond 100,000 games. This gives some indication of a model that is better at varying its game, instead of playing a small amount of games again and again. A typical example of size of knowledge base in relation to games played can be seen in Table 26.

Games played	Knowledge base size
100	579
250	1,203
500	1,744
1,000	2,611
2,000	3,475
3,500	4,599
5,000	4,785
10,000	5,075
50,000	5,121
100,000	5,122
200,000	5,124

Table 26: Knowledge base size in relation to games played

Again as the agent starts to perform well around 5,000 trained games, the claim made in 5.7, that some patterns will be identified as being of such a type, that has they first been played once or a few times, they will be identified as very poor for one of the players and never be played again. Still seems to somewhat hold, although the amount of boards there has been cut off is not that many. Still this has the obvious drawback that when a human player then starts to play these rather odd strategies, the agent implementing the solution model can be fooled because the agent suddenly then is caught in situations it does not recognize. However as this model is better at varying its game play more training time will minimize this problem as more situations is learned from.

Another interesting observation is how many games played by the agent, will end in a draw depending on how much it has been trained before it is set to train even further. A typical example of this can be seen in Table 27.

Training games	Drawn
0 – 2,000	521
2,000 – 4,000	1,210
4,000 – 6,000	1,304
6,000 – 8,000	1,587
8,000 – 10,000	1,823
10,000 – 12,000	1,826
12,000 – 14,000	1,941
14,000 – 16,000	1,933
16,000 – 18,000	1,963
18,000 – 20,000	1,968
54,000 – 56,000	2,000
56,000 – 58,000	1,995
58,000 – 60,000	1,999

Table 27: Amount of draw games in relation to training

The amount of games ending in a draw is slowly increasing. At around 50,000 games trained, it will begin to almost only play draw games. This shows that the agent implementing the model will learn to play the game of tic-tac-toe in a quite rational manner.

Although playing rational the agent is still good at varying its game play such it will not become boring to play against. It will not play the same strategy again and again. It will instead continually try out new strategies giving the agent a very human like playing fashion.

7.4.3 Speed and space performance

The model is also performing extremely fast. On a standard PC with 512 MB of memory and an AMD-Athlon processor running at 1.1 gigahertz, training 100,000 games takes just below 90 seconds, also making the solution both much faster and much better at learning then the neural network based solution. Also as the index structure used in the knowledge base is a hash table, learning and decision making has a time complexity close to O(1) meaning that more boards stored in the knowledge base will not make the solution perform noticeably slower.

The physical size of the knowledge base has also shown to be acceptable. When saving the knowledge base to a file on the hard drive a knowledge base containing all the possible 5477 different boards, it has a size just below 100 kilobytes (97.3 kilobytes to be exact). That gives the ability to theoretically store roughly 54,770,000 board patterns in a one gigabyte file. Remember back in the related work chapter section 4.1 where Deep Blue was described, it had a database of 700,000 grandmaster games. If then estimating that a chess game in average consists of 90 different boards. Its database would contain 63,000,000 boards with the amount of unique boards even lower.

Let us now make a hypothetical example where the goal directed learning model has been implemented in a more complex game domain. Then we instead of saving to a

text file use a database. The combination of a database and a more complex game domain might do that each pattern record takes up three times as much space then in its current form in the tic-tac-toe game. Slightly more then 1825 boards can then be stored in a 100 kilobyte database, giving the model the power of having more then 182,500,000 boards stored in a ten gigabyte database. This should mean that if the model in a complex game domain is guided by some rules, ensuring somewhat rational play, the database should not grow to a size where the space required for storage should become a problem.

7.4.4 Merging and competitive training

Besides the strength of the models flexibility of being able to work in co-existence with other decision making models, as well as the fact that it actually works end seems to be practically possible. One other strength is that knowledge obtained by one agent can be merged with knowledge from another agent. In relation to normal situations and often played boards this does not give any noticeable advantages. However to be able to identify “normal” play it has to play some irrational strategies to learn this. This means that an agent plays some, but not all, sub patterns of a board before it learns a weak board, and starts to avoid it. However these played sub patterns might differ from agent to agent. One agent can then learn a whole range of these odd strategies by having knowledge from a range of other agents merged into its own knowledge.

This feature can be used for creating an agent that is able to learn the game even better. So instead of letting one agent learn only by itself. The agent learns by having knowledge merged from a range of other agents, which in turn play each other or itself in a range of games. At the end all knowledge obtained by all agents is merged into the one agent the human is to play against.

The process of merging can be illustrated as below. Consider the case where Knowledge base A in Table 28 is to be merged into knowledge base B.

Knowledge base A

Pattern	WonA	WonB	Draws
----A----	4	2	7
B---A----	2	2	4
B--AA----	2	1	3

Knowledge base B

Pattern	WonA	WonB	Draws
----A----	5	2	8
B---A----	3	4	4
--B-A--B-	1	3	2

Table 28: Two knowledge bases that is to be merged

The process is then simply to take every record from A and copy into B. In the case that a record exists, the values are added together. The resulting knowledge base B in the above case will be the one illustrated in Table 29.

Pattern	WonA	WonB	Draws
----A----	9	4	15
B---A----	5	6	8
--B-A--B-	1	3	2
B--AA----	2	1	3

Table 29: Resulting knowledge base after merge

Using the above described method where knowledge bases are merged, ten different agents is often able to learn all the 5,477 existing board patterns (excluding the empty

board) after a total of 100,000 games played, making the tic-tac-toe agent almost impossible to beat. Actually at that point, the challenge for the human player is more about avoiding a loss, then wining the game.

The feature of merging does however have additional strength. One could imagine a range of people playing and training game agents, and then exchange knowledge bases. It would simply just be a matter of having a web site where knowledge bases could be submitted to and downloaded from. In that fashion people from all over the world could in co-existence help in the training of an agent.

Also games could be initialized from different key situations. The agent could then train and learn from this sub situation of the game. In this way a complex game has the possibility to be divided into a range of different sub situations. A whole range of agents could then be set to train different situations that at a later point could be merged. In this way games can be learned in a divide and conqueror fashion.

7.5 Future work

A model that has been shown to be highly effective for teaching an agent to play the game of tic-tac-toe has been identified. However as work done in this project, is to lay the foundation for an implementation in a more complex game domain (possibly backgammon), there is still improvements to be done to increase changes for success.

First of all the final solution model in its current form might be taking a little to search based approach to obtain a wide variety of knowledge about the game domain. Although this search based approach is not even close to the complexity of a game tree, it is not possible without some form of guidance in more complex game domains. As a consequence some way of narrowing down the search space is needed. Therefore the feature of expert knowledge, explicit put into the agent will be unavoidable. This expert knowledge is then to help guiding the agent such that it from the start is able to play reasonably well, and not playing totally irrational strategies. However although some rules of thumb is followed and a game is played somewhat rationally, some strategies will still be better then other. The solution model proposed in this project is then believed in co-existence with some expert knowledge, to be practically possible such computer agents are able to learn even complex games.

This combination of models should also correspond quite well with how humans think as we also possess the ability to discard a wide range of moves, simply because we know that breaking some simple rules, is completely irrational. We then focus our attention on a small group of possible moves that is tried out. Depending on the outcome we then learn which strategies are good and bad, and then become better at playing the game.

Implementing some simple rules should also ensure that when a human player eventually starts to play irrational against an agent, it will still have some form of guidance that should ensure that it deals with the situation in a hopefully satisfying way.

One possible way of modeling this expert knowledge, could be by the use of fuzzy controllers explained in section 2.4. The fuzzy controller could then as input receive a possible board. The output from the fuzzy controller would then be a control signal in the range of -1 to 1, telling how good that board is estimated to be.

Future work and improvements can also be done to the goal directed learning model itself. An obvious improvement could be to preserve the structure of played games. In its current form it is not possible get any information about the flow of games. Adding this feature could possibly make the agent even better at making decisions. Returning to the four example games illustrated in Figure 34, page 60 in section 5.6, these games could be stored in a tree or graph like fashion as illustrated in Figure 54.

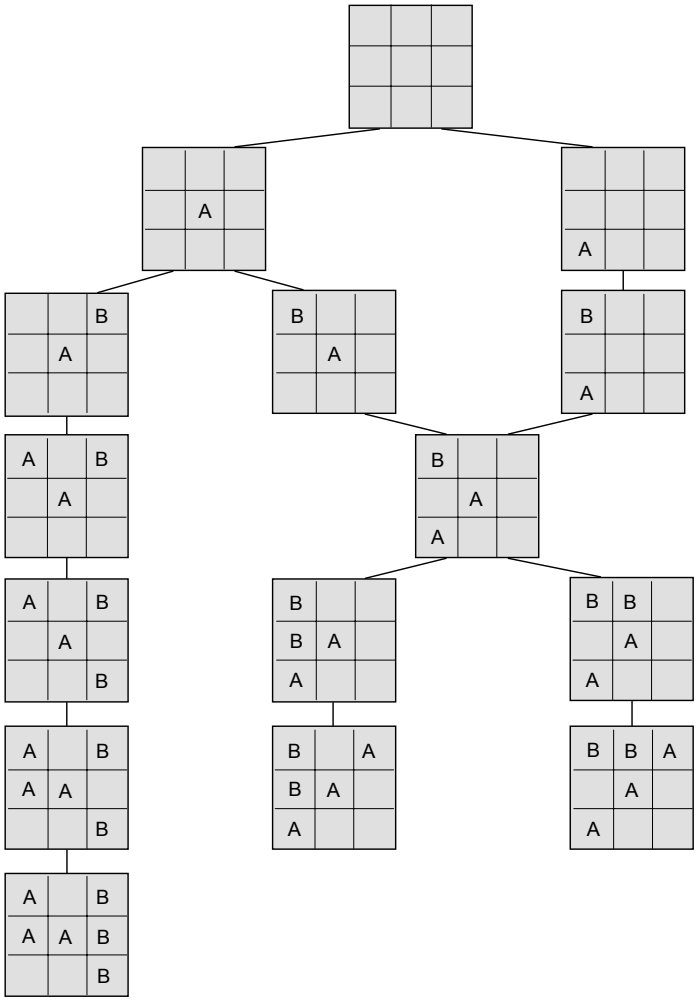


Figure 54: Preserving structure of games

However as the amount of possible games is larger then the amount of possible boards this might not be possible after all. Future work is to investigate if it is possible or in that case it is worth its cost in complexity.

Also the ability to make decisions in a situation that does not exactly match previous situations could be a feature to add. The agent would then be able to, from a range of other somewhat identical situations, be able to make a hopefully clever decision.

If this feature can be successfully added, it might also be possible, to compact the knowledge base to only hold a range of typical and generalized situations from which the agent is able to make decisions.

7.6 Conclusion on models, test and evaluation

The development of a solution model became and iterative process. All though the initial model did not perform well the final solution has proved to be extremely effective in the domain of tic-tac-toe. The agent is able to learn the game and play extremely well. It is also good at varying its game play such it will not become boring to play against, giving it a very human like playing fashion.

The final solution has been divided in two pieces, one responsible for calculating the actual score of a board, and one responsible for forcing the agent to try out new strategies. Both of these methods has been made such a board is graded with a value between 1 and -1. Modeling the agent's decision making in this way ensures a lot of flexibility such that additional methods can be easily added to the agent. Only requirement is that the output of a board's strength is graded with a value between 1 and -1. In a more complex game domain it will then be possible to add a method that uses human expert knowledge to grade a board with a score between 1 and -1, such completely irrational playing strategies can be avoided and still handled by the agent.

Performance in relation to speed and space has shown to be satisfactory. Time complexity for both learning and decision making is close to O(1) and space complexity for the knowledge base is O(n), where n is the number of different boards identified by the agent. The amount of information stored for each board is however so limited that with the amount of hard drive space available today, and still rapidly increasing, this should not possess any trouble in even complex game domains, as long some method is used for guidance to sort out completely irrational playing strategies.

8. Conclusion

This dissertation had to objectives. First objective was to study and demonstrate understanding of previous and current research in the area of artificial intelligence as well as artificial intelligence related to games. During this research knowledge and understanding of techniques used was obtained. Further more it was established that artificial intelligence manly seems to be about solving problems for which there is no straight forward solution.

This study helped guiding the fulfillment of the second main objective of this dissertation. The second objective was the development of a solution model useful for learning computers to play a wide variety of classical board games. A new method for machine learning in classical board games was defined, tested and evaluated. The new model in its final form showed to be highly effective in the game of tic-tac-toe. The final model was also made in a very flexible manner such that it easily can co-exist with other learning and decision making methods.

Extensions can still be made to the model to increase its effectiveness even more. Also the model needs to be tested in other more complex game domains. During the second part of the dissertation work on the model will continue and hopefully show it to be a new and highly effective method for machine learning useful in a range of classical board games.

9. References

Books

[1]. A. Konar, “Artificial Intelligence and Soft Computing: Behavioural and Cognitive Modelling of the Human Brain,” ISBN 0-8493-1385-6, 1999

[2]. B. Dahlbom, L. Mathiassen, “Computers in Context: The Philosophy and Practice of Systems Design,” ISBN 1-55786-405-5

[3]. L. Mathiassen, A. Munk-Madsen, P.A. Nielsen, J. Stage, ”Objekt Orienteret Analyse og Design,” ISBN 87-7751-129-8

[4]. S. Russell, P. Norvig, “Artificial intelligence: a modern approach,” ISBN 0-13-103805-2, 1995

Papers

[5]. A. M. Turing, “Intelligence machinery,” *Machine intelligence*, **5**: 2–23, Edinburgh University Press, 1950

[6]. B. D'Ambrosio, “Inference in Bayesian Networks,” *AI Magazine* **20**(2): 21–36, 1999

[7]. D. B. Fogel, “Evolutionary Entertainment with Intelligent Agents,” *IEEE Computer* **36**(6): 94–96, 2003

[8]. D. B. Fogel, “Evolving a checkers player without relying on human experience,” *Intelligence* **11**(2): 20–27, 2000

[9]. G. Tesauro, “Temporal Difference Learning and TD-Gammon,” *Communications of the ACM* **38**(3): 58–68, 1995

[10]. J. Schaeffer, R. Lake, P. Lu, M. Bryant, ” Chinook: The world man-machine checkers champion,” *AI Magazine* **17**(1): 21–29, 1996

[11]. K. Chellapilla, D. B. Fogel, “Evolving neural networks to play checkers without relying on expert knowledge,” *IEEE Transaction on Neural Networks* **10**(6): 1382–391, 1990

[12]. M. Campbell, “Knowledge Discovery in Deep Blue,” *Communications of the ACM* **42**(11): 65–67, 1999.

[13]. M. Wooldridge, N. R. Jennings, “Intelligent Agents: Theory and Practice,” *The Knowledge Engineering Review* **10**(2): 115–152, 1995

[14]. Y. Seirawan, H. A. Simon, T. Munakata, “The implications of Kasparov vs. Deep Blue,” *Communications of the ACM* **40**(8): 21–25, 1997

Web links (Links validated May 21, 2004)

[15]. B. Moreland, “Alpha-beta search,” http://www.seanet.com/~brucemo/topics/alphabeta.htm

[16]. I. Goldberg, “Falcon: Fuzzy Adaptive Learning Control Network,” http://www.generation5.org/content/2001/falcon.asp

[17]. J. Chen, S. Lu, D. Vekhter, “Game theory,” http://cse.stanford.edu/classes/sophomore-college/projects-98/game-theory

[18]. J. Jantzen, “Design of Fuzzy Controllers,” http://www.iau.dtu.dk/~jj/pubs/design.pdf

[19]. J. Jantzen, “Tutorial on Fuzzy Logic,” http://www.iau.dtu.dk/~jj/pubs/logic.pdf

[20]. J. Matthews, “An Introduction to Fuzzy Logic,” http://www.generation5.org/content/1999/fuzzyintro.asp

[21]. J. Matthews, “An Introduction to Neural Networks,” http://www.generation5.org/content/2000/nnintro.asp

[22]. J. Matthews, “How do Genetic Algorithms Work,” http://www.generation5.org/content/2001/ga_math.asp

[23]. M. Buckland, “Genetic Algorithms in Plain English,” http://www.ai-junkie.com/gat1.htm

[24]. M. Buckland, “Neural Networks in Plain English,” http://www.ai-junkie.com/nnt1.html

[25]. O. M. Halck, F. A. Dahl, “On Classification of Games and Evaluation of Players – with some sweeping generalizations about the literature,” http://www.ai.univie.ac.at/icml-99-ws-games/papers/halck.ps.gz

[26]. P. Crochat, D. Franklin, “Back-Progagation Neural Network Tutorial,” http://pcrochat.online.fr/webus/tutorial/BPN_tutorial.html

[27]. P. Y. Chan, H. Y. Choi, Z. Xiao, “Game trees. Alpha-beta search,” http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic11

10. Appendix

10.1 Content of enclosed CD

A screenshot of the enclosed CD can be seen in Figure 55.

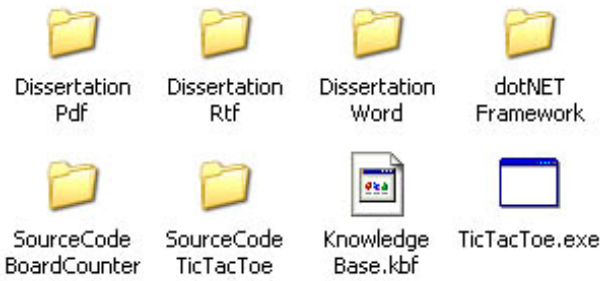


Figure 55: Screenshot of enclosed CD

Dissertation Pdf: This directory contains this paper in Adobe Portable Document Format (PDF) as well as an installer for Adobe Reader 6.01.

Dissertation Rtf: This directory contains this paper in Rich Text Format (RTF).

Dissertation Word: This directory contains this paper in Microsoft Word 2003 Format (DOC).

dotNET Framework: This directory contains the Microsoft .NET Framework 1.1 portable installer.

SourceCode BoardCounter: This directory contains source code and program mentioned in section 5.7 used for counting the exact number of unique and possible boards in the game of Tic-tac-toe.

SourceCode TicTacToe: This directory contains source code for the Tic-tac-toe prototype program used for implementation, test and evaluation of the goal directed learning model.

Knowledge Base.kbf: This is a knowledge base file that can be loaded into an agent in the Tic-tac-toe prototype program.

TicTacToe.exe: A compiled and ready to use version of the Tic-tac-toe prototype program. It requires Microsoft .NET Framework 1.1 to be able to run.

10.2 Game Rules

10.2.1 Chess

Rules taken from http://www.conservativebookstore.com/chess/index.htm

The ultimate aim in the game of chess is to win by trapping your opponent's king.

White is always first to move and players take turns alternately moving one piece at a time. Movement is required.

Each type of piece has its own method of movement (described in the following sections). A piece may be moved to another position or may capture an opponent's piece. This is done by landing on the appropriate square with the moving piece and removing the defending piece from play.

With the exception of the knight, a piece may not move over or through any of the other pieces.

Setup

The chessboard is made up of eight rows and eight columns for a total of 64 squares of alternating colors. When the board is set up it should be positioned so that a light square is positioned on the extreme lower right hand side of the chess board. See Figure 56.

Figure 56: Initial board position in a game of chess

When setting up, make sure that the light queen is positioned on a light square and the dark queen is situated on a dark square. The two armies should be mirror images of one another.

The Pawn

There are eight pawns situated on each side of the board. They are the least powerful piece on the chess board, but have the potential to become equal to the most powerful.

Pawns cannot move backward or sideways, but must move straight ahead unless they are taking another piece.

Generally pawns move only one square at a time. The exception is the first time a pawn is moved, it may move forward two squares as long as there are no obstructing pieces. A pawn cannot take a piece directly in front of him but only one at a forward angle.

Should a pawn get all the way across the board to reach the opponent's edge of the table, it will be promoted. The pawn may now become any piece that the moving player desires (except a king or pawn). Thus a player may end up having more than one queen on the board.

The Rock

The rook, shaped like a castle, is one of the more powerful pieces on the board. The rooks, grouped with the queen, are often thought of as the "major pieces".

The rook can move any number of squares in a straight line along any column or row. They cannon move diagonally. The simplicity of the rook's movement is indeed what makes it powerful. It can cover a significant area of the board and there are no areas which an opponent's piece - moving one square at a time - can slip through.

The rook may also make a move in conjunction with the king. This manoeuvre will be explained in the section called castling.

The Knight

The knight is the only piece on the board that may jump over other pieces. This gives it a degree of flexibility that makes it a powerful piece.

Since obstructions are not a bar to movement (unless there is a friendly piece on the square where the knight would move) the knight's path of movement has never been well defined.

The knight can be thought of as moving one square along any rank or file and then at an angle. The knight's movement can also be viewed as an "L" laid out at any horizontal or vertical angle.

Note that the squares to where the knight can move are all of the opposite coloured squares two steps away from his starting square. This may help you visualize the knights’ range of influence on the board.

The Bishop

The bishop may move any number of squares in a diagonal direction until it is prevented from continuing by another piece.

Each player begins with two bishops, one originally situated on a light square, the other on a dark square. Because of the nature of their movement, the bishops always remain on the same colored squares.

Bishops are a powerful piece (though less so than the queen or rooks). Nevertheless, the bishop is a great piece to have in open situations when it can range the board. The knight is better in cluttered situations where it can utilize its ability to jump over other pieces.

The Queen

The queen is, without doubt, the most powerful piece on the chessboard. She can move as many squares as she desires and in any direction (barring any obstructions).

She captures in the same way that she moves, replacing the unlucky opposing piece that got in her way. (She must, of course, stop in the square of the piece she has captured - unlike the knight the queen does not jump other pieces.)

The queen's power is so great that she is considered to be worth more than any combination of two other pieces (with the exception of two rooks). Thus it would be better, under normal circumstances, to sacrifice a rook and a bishop (for example) than to give up a queen.

The King

Though not the most powerful piece on the board, the king is the most vital; for once he is lost the game is lost.

The king can only move one square in any direction. There is only one restriction on his movement - he may not move into a position where he may be captured by an opposing piece. Because of this rule, two kings may never stand next to each other or capture each other.

Castling

Castling is a special defensive maneuver. It is the only time in the game when more than one piece may be moved during a turn.

This move was invented in the 1500's to help speed up the game and to help balance the offence and defense.

It can only occur if there are no pieces standing between the king and the rook.
Neither king nor rook may have moved from its original position.
There can be no opposing piece that could possibly capture the king in his original square, the square he moves through or the square that he ends the turn.
The king moves two squares toward the rook he intends to castle with (this may be either rook). The rook then moves to the square through which the king passed.
Hopefully, the diagram to the left makes this clear.

End game

The game ends when one of the players captures his opponent's king, when one of the player's resigns or there is a stalemate.

When a player's king is threatened by an opposing piece, it is said to be "in check". When a player places the opposing king in check he should announce, "Check". The object of a player is not merely to place his opponent's king in check but to make certain that every square where the king has a possibility of movement is also covered. This is called checkmate. The king is considered captured.

Either player may resign at any time. This generally happens when a player loses a major piece and the outlook for victory in his case appears bleak.

Stalemate is considered a tie. A stalemate occurs when a player's only move is to place his own king in check, but its current square is not threatened. As long as he can move another piece or the king can move to an open square, stalemate may not occur.

A draw also results when the only two pieces on the board are Kings, regardless of their position. If the pieces remaining on the board make check mate impossible, for example one cannot checkmate an opponent with only a king and a bishop a draw would also result.

10.2.2 Checkers

Rules taken from MVP Checkers http://www.mvpsoft.com

The objective of checkers is to capture all your opponent’s pieces or block them so they are unable to move.

Black always moves first.

Figure 57 is a screenshot of the initial board position of a checkers game about to start.

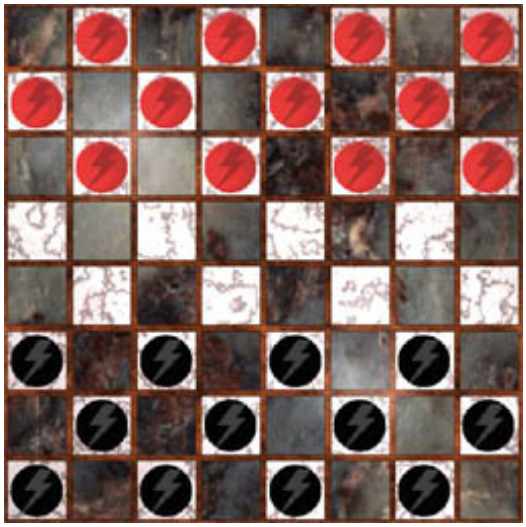


Figure 57: A game of checkers about to start

Pieces move diagonally forward. Black starts at the bottom of the board and moves up towards Red; Red starts at the top of the board and moves down towards Black.

Captures are enforced. If you can capture a piece, you must do so. If you have more then one possible capture, you may choose which to make.

To capture an opponent he must be in one of the squares diagonally in front of you, and the square beyond him must be empty. The captured pieces are removed from the game. If the piece can make another capture in its new location, it must continue to jump until it either reaches the last row or has no more possible jumps.

When a piece reaches the last row, it is promoted to a King. Kings can move and capture backward as well as forward

The game is over when a player can no longer make a move either because he has no more pieces, or his opponent has him totally blocked and he has no valid moves.

10.2.3 Backgammon

Rules taken from Backgammon Galore http://www.bkgm.com

Backgammon is a game for two players, played on a board consisting of twenty-four narrow triangles called points. The triangles alternate in colour and are grouped into four quadrants of six triangles each. The quadrants are referred to as a player's home board and outer board, and the opponent's home board and outer board. The home and outer boards are separated from each other by a ridge down the middle of the board called the bar (see Figure 58).

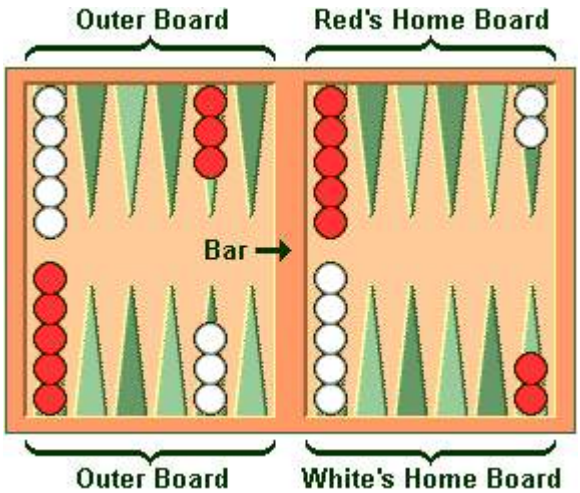


Figure 58: Initial board layout of a game of backgammon

The points are numbered for either player starting in that player's home board. The outermost point is the twenty-four point, which is also the opponent's one point. Each player has fifteen checkers of his own color. The initial arrangement of checkers is: two on each player's twenty-four point, five on each players thirteen point, three on each player's eight point, and five on each player's six point.

Both players have their own pair of dice and a dice cup used for shaking. A doubling cube, with the numerals 2, 4, 8, 16, 32, and 64 on its faces, is used to keep track of the current stake of the game.

Object of the Game

The object of the game is for a player to move all of his checkers into his own home board and then bear them off. The first player to bear off all of his checkers wins the game.

Movement of the Checkers

To start the game, each player throws a single die. This determines both the player to go first and the numbers to be played. If equal numbers come up, then both players roll again until they roll different numbers. The player throwing the higher number now moves his checkers according to the numbers showing on both dice. After the first roll, the players throw two dice and alternate turns. The roll of the dice indicates how many points, or pips, the player is to move his checkers. The checkers are always moved forward, to a lower-numbered point. The following rules apply:

A checker may be moved only to an open point, one that is not occupied by two or more opposing checkers.

The numbers on the two dice constitute separate moves. For example, if a player rolls 5 and 3, he may move one checker five spaces to an open point and another checker three spaces to an open point, or he may move the one checker a total of eight spaces to an open point, but only if the intermediate point (either three or five spaces from the starting point) is also open.

A player who rolls doubles plays the numbers shown on the dice twice. A roll of 6 and 6 means that the player has four sixes to use, and he may move any combination of checkers he feels appropriate to complete this requirement.

A player must use both numbers of a roll if this is legally possible (and all four numbers of a double). When only one number can be played, the player must play that number. Or if either number can be played but not both, the player must play the larger one. When neither number can be used, the player loses his turn. In the case of doubles, when all four numbers cannot be played, the player must play as many numbers as he can.

Hitting and Entering

A point occupied by a single checker of either color is called a blot. If an opposing checker lands on a blot, the blot is hit and placed on the bar.

Any time a player has one or more checkers on the bar, his first obligation is to enter those checker(s) into the opposing home board. A checker is entered by moving it to an open point corresponding to one of the numbers on the rolled dice.

For example, if a player rolls 4 and 6, he may enter a checker onto either the opponent's four point or six points, so long as the prospective point is not occupied by two or more of the opponent's checkers.

If neither of the points is open, the player loses his turn. If a player is able to enter some but not all of his checkers, he must enter as many as he can and then forfeit the remainder of his turn.
After the last of a player's checkers has been entered, any unused numbers on the dice must be played, by moving either the checker that was entered or a different checker.

Bearing Off

Once a player has moved all of his fifteen checkers into his home board, he may commence bearing off. A player bears off a checker by rolling a number that corresponds to the point on which the checker resides, and then removing that checker from the board. Thus, rolling a 6 permits the player to remove a checker from the six point.

If there is no checker on the point indicated by the roll, the player must make a legal move using a checker on a higher-numbered point. If there are no checkers on higher-numbered points, the player is permitted (and required) to remove a checker from the highest point on which one of his checkers resides. A player is under no obligation to bear off if he can make an otherwise legal move.

A player must have all of his active checkers in his home board in order to bear off. If a checker is hit during the bear-off process, the player must bring that checker back to his home board before continuing to bear off. The first player to bear off all fifteen checkers wins the game.

Doubling

Backgammon is played for an agreed stake per point. Each game starts at one point. During the course of the game, a player who feels he has a sufficient advantage may propose doubling the stakes. He may do this only at the start of his own turn and before he has rolled the dice. A player who is offered a double may refuse, in which case he concedes the game and pays one point. Otherwise, he must accept the double and play on for the new higher stakes. A player who accepts a double becomes the owner of the cube and only he may make the next double.

Subsequent doubles in the same game are called redoubles. If a player refuses a redouble, he must pay the number of points that were at stake prior to the redouble. Otherwise, he becomes the new owner of the cube and the game continues at twice the previous stakes. There is no limit to the number of redoubles in a game.

Gammons and Backgammons

At the end of the game, if the losing player has borne off at least one checker, he loses only the value showing on the doubling cube (one point, if there have been no doubles). However, if the loser has not borne off any of his checkers, he is gammoned and loses twice the value of the doubling cube. Or, worse, if the loser has not borne off any of his checkers and still has a checker on the bar or in the winner's home board, he is backgammon and loses three times the value of the doubling cube.