POLITECNICO
MILANO 1863

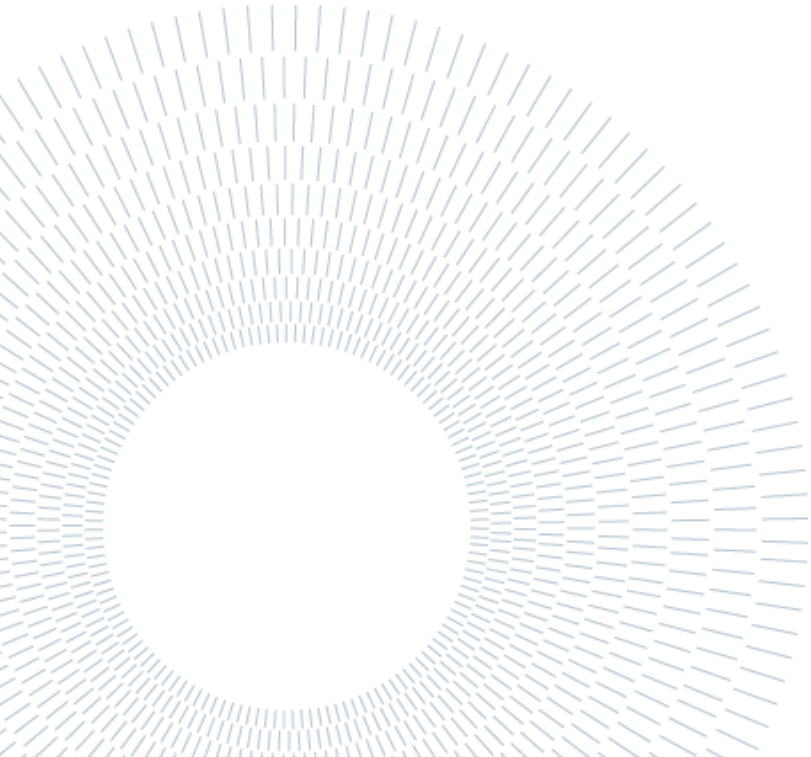
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Extension of the ADAPTA architecture applied to the videogame Advance Wars

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING
INGEGNERIA INFORMATICA

Author: **Lorenzo Carnaghi**

Student ID: 920740
Advisor: Daniele Loiacono
Co-advisor: Pier Luca Lanzi
Academic Year: 2020-2021



Abstract

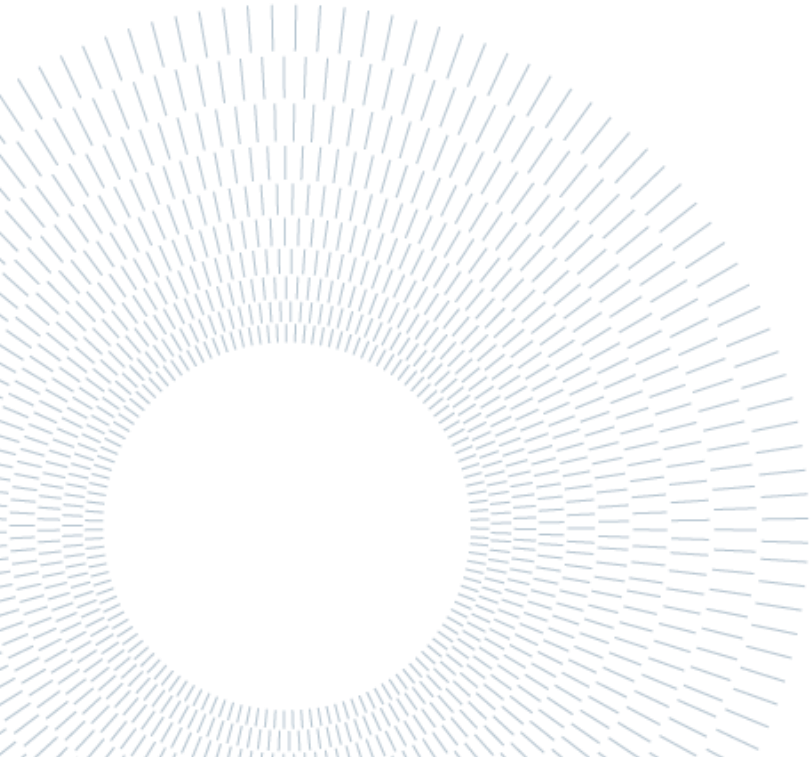
In 2008, Maurice Bergsma and Pieter Spronck proposed an AI architecture, named ADAPTA, for a deeply simplified version of the Turn Based Strategy (TBS) videogame Advance Wars™. The aim of this thesis is to extend the concepts of such architecture and apply them to the original game. The created AI is able to change its behavior and strategies between matches: its aim is not necessarily to win, but is parametrized to give the player a customized challenge, in line with the Game Design principle of Flow. Since the ADAPTA architecture features modularity as one of its strongest points, this work also attempts to lay the foundations for future research. As an example of this, a new type of approach in this field is also proposed, which makes use of a Convolutional Neural Network applied to a TBS game.

Keywords: Artificial Intelligence, TBS Game, Influence Map, Convolutional Neural Network.

Abstract in italiano

Nel 2008, Maurice Bergsma e Pieter Spronck proposero un'architettura, nominata ADAPTA, per un Intelligenza Artificiale applicata ad una versione estremamente semplificata del videogioco strategico a turni Advance Wars™. Lo scopo di questa tesi è di estendere i concetti di tale architettura e applicarli al gioco originale. La IA creata può modificare il suo comportamento e le sue strategie tra una partita e l'altra: lo scopo non è necessariamente vincere, ma è parametrizzato per poter fornire al giocatore un'esperienza personalizzata, in linea con il principio di Flow nell'ambito di Game Design. Dato che l'architettura ADAPTA è caratterizzata da una forte modularità, questa tesi vuole anche proporsi come possibile fondamento per ricerche future. A dimostrazione di questo viene anche proposto un nuovo tipo di approccio in questo campo, che utilizza Convolutional Neural Networks applicate ad un gioco strategico a turni.

Parole chiave: Intelligenza Artificiale, Giochi Strategici a Turni, Mappa di influenza, Convolutional Neural Network.



Contents

1 Introduction

Context

Artificial intelligence (AI) applied to strategy videogames has captured the interest of many researchers through the years. While concepts are similar, a lot of researches in the field of AI for strategy games are dedicated to board games. This may be due to the fact that board games are overall more popular, generally easy to develop in a digital version for AI studies, and usually simpler in terms of rules, mechanics, and situations.

Most of the commercial strategy videogames adopt behavioral AIs because creating behavioral AIs is overall a simpler task to manage, and usually due to budget restrictions and deadlines there is simply no reason to spend time on researching or developing a particular framework for an interesting Artificial Intelligence. Also being online gameplay very common, AI is often a secondary problem. Accordingly, while the player may be free to play against one among a set of AIs, usually available just as different levels of difficulty (this is not always the case, in Advance Wars for instance there is the possibility of choosing a defensive or offensive AI), in the end it is possible that with time the player understands the general behavior of the AIs and hence could adapt his gameplay to exploit it. This results in a less interesting experience.

In the Game Design context this is known as Flow. This is the state in which the player is positively focused solely on the game with clear goals and no distractions. Since games are not in general short experiences, the flow perceived is divided in micro and macro flow. The first is relative to short time windows, the second is relative to the overall game experience [1] [2].

On the micro flow side, the game proceeds flatter because mostly anything is expected, while on the macro level, if the player recognizes the patterns of the AI they have probably already developed enough knowledge of the game to be more than beginner, which means that the skills they developed now are matched with an easier challenge: the flow is broken, and the game tends to become boring. In following Figure 1-1, the player in this scenario would be in the bottom of the

chart, in the center or on the right depending on their skills; either case they would be in the boredom zone [1].

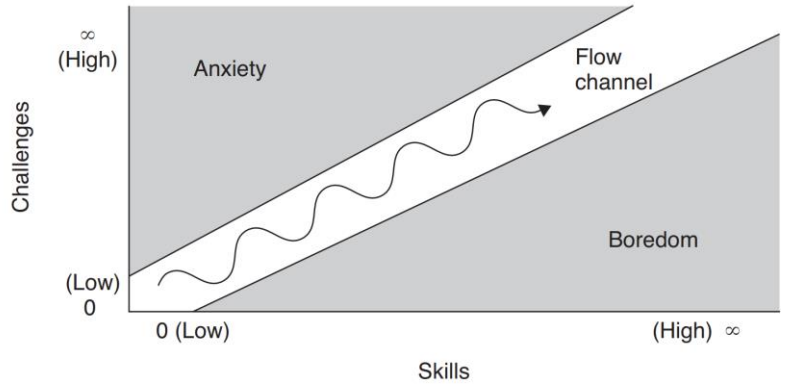


Figure 1-1 Flow graph. (from [1], p.121)

Objective

In 2008, Maurice Bergsma and Pieter Spronck proposed the ADAPTA architecture for a heavily simplified version of Advance Wars [3].

The proposed ADAPTA (Allocation and Decomposition Architecture for Performing Tactical AI) featured an interesting concept with modularity in mind. However, the idea was developed marginally on a simplified case, named Simple Wars.

The objective of this thesis is to develop and extend the proposed architecture to a more realistic videogame scenario and experimenting with its modularity, in order to create an interesting AI whose objective can be configured to adapt to the player in different ways, ultimately creating a more interesting AI to play against.

Outline

- In chapter 2 we present a brief scenario on AIs applied to strategy games, both board and video games, and how they relate to the case of Advance Wars;
- In chapter 3 we explain in detail the rules of Advance Wars. At the beginning there's also a quick reference containing the key features of Advance Wars, which can be useful to understand quickly how the games work without knowing all the details (3.2);
- In chapter 4 we illustrate the ADAPTA Architecture at a general level, compared with the original proposal;
- In chapter 5 we describe the Multi Influence Network Module, which is one of the modules developed for the ADAPTA, and its evolution process;
- In chapter 6 we describe another module which is the CNN module, and the NEAT technique for evolution;
- In chapter 7 we discuss the training environment, challenges and results obtained;
- In chapter 8 we draw conclusions about this work and propose topics for further researches.

2 AI in strategy games

There is a good amount of literature on the topic of Machine Learning applied to strategy games. The majority of it focuses on more classical strategy board games, which generally involve definitely less possible actions and states than Turn Based Strategy (TBS) videogames.

The literature on TBS videogames is much more limited, especially if we narrow the typology of TBS game, for instance considering the "War chess" games, a term which loosely defines strategy games with focus on units moving on a board, which is a representation of sort of a war scenario [4]. As explained in detail in chapter 3, Advance Wars is a series of TBS games featuring a 2D, squared grid with buyable/expendables units, where the objective is to destroy the opponent's army or capture their Headquarters, which is a precise tile on the map. The very core of the gameplay is about deciding an action for each unit on each turn, where most of the actions are simply move or attack. It is a multi-player, single-agent, (marginally) stochastic, fully observable (if Fog of War is not present, which is not in the experiments made. See 3.5.4 for FoW details) game.

2.1. Board Games

Most concepts of AIs developed for board games can be applied or adapted to a TBS videogame. A TBS game where the state is fully observable would be in fact fully playable as a board game[1], albeit not being practical because of the substantially increased number of mechanics or calculations which would slow down too much the flow of a match. In fact, the game studied in this thesis and traditional board games share in common the property of being played on a square grid board and at the core resembling the mechanic of moving a unit from a tile to another, attacking the opponent or controlling space in some way.

[1] Some of the concepts adopted in the extension of the ADAPTA architecture are in fact presented in a board game environment [6]

The main difference in terms of tactics and techniques between Advance Wars and traditional board games is that traditional board games have always the same starting board configuration. This includes board size, starting unit disposition (both number and positioning) and game map arrangement, which is plain in each of these games, since tiles have no special properties or if they have one, they are always in the same positions (like in chess or checkers the tiles which allow promotion of units are always in the same rows).

At the same time, albeit figuring a vastly higher number of configurations, if we consider a specific configuration in Advance Wars and one in any of the mentioned board game and change it by a single element (1 unit moved/changed stats, or a terrain changed for Advance Wars and 1 piece changed position for the other board games) the impact it can have on the game is drastically increased in the board games. In short each configuration holds in general a bit more importance in the mentioned board games w.r.t. a configuration in Advance Wars. It has to be mentioned that this holds true also inside Advance Wars itself, because as the map increases in size, the single tile and unit contained loses importance to the overall possible outcome of the game.

The main challenge in War chess games like Advance Wars, compared to the majority of board games, is that the branching factor explodes very easily, since in a turn all the units should be moved (leaving them where they are is still a particular case of a move) in any order, and that may also why the literature on such games is scarcer.

2.1.1. Checkers

Checkers is a very popular and well-known board game, and has been officially solved in 2007 [5], being as today the most complex board game solved.

It can be however useful to experiment techniques on such game due to its simpler rules. In fact, it was adopted to assess the potential of the HyperNEAT technique in combination with CPPNs [6], which will be shortly explained in 6.5.1.

The main difference with Advance Wars is that the input itself is way deeper since each tile contains more information, but most importantly that the size of the map is variable, while is not in checkers. This introduces more challenges because it cannot be used a fixed-size input, which means that a standard Neural Network cannot be trained, considering that the input is interconnected spatially.

2.1.2. Chess

Chess is probably the most famous and played board game up to present day.

A lot of chess engines have been developed since the 1950s, when Alan Turing and Claude Shannon devised the first programs playing chess [7] [8]. Worth to be mentioned are Deep Blue defeating the world champion Garry Kasparov in 1997, Stockfish, which is an open source engine released in 2008 (v 1.0) [9] and still updated in new versions, and AlphaZero, which in 2017 beat Stockfish, the strongest AI at the time and still one of the strongest, beaten only by few other engines [10]. Stockfish adopts a technique for evaluating states and doing searches in the space of states [9]. AlphaZero in contrast is at the core a deep neural network, deprived of domain-specific knowledge [7].

In relation with Advance Wars, exploring the space of states by an approach similar to the one adopted by Stockfish would prove to be very difficult: this is mainly because the branching factor is enormously higher, and utilizing heuristics to reduce the options could fail to capture some potential tactics, limiting the "creativity" that could be developed in training. Keeping in mind that the objective is to produce an "interesting" opponent, this is an important factor to consider.

Regarding AlphaZero's approach, the main problem in creating a deep neural network would lie mainly in the problem that, as already mentioned in the checkers subchapter, the size of the input map is variable. This creates a series of problems not easily solvable with a NN, and in any case it increases the complexity of a problem spanning already in a very higher-sized space of states (see 7.1.2)

2.1.3. Shogi

Known also as Japanese Chess, it is in fact the most popular chess variant in Japan. Among the most prominent differences, it features different types of units and movements, a 9x9 board and the ability to return captured pieces in game. [11]

Similarly to chess, Shogi software in recent years (in 2010 the first program defeated a shogi champion [12]) have surpassed human players and their tactics are studied by skilled players. Among the software worth mentioning there are Ponanza, Elmo and AlphaZero. The latter in 2017 as well as in standard chess it was also able to defeat Elmo, which has been the champion among Shogi software in May of the same year [13] [14]. Elmo is an advanced evaluation function and book file to be used with an Alpha-Beta search engine [15] . For the same reasoning as in chess, a similar approach would be way harder or limiting in Advance Wars.

2.1.4. Go

Invented in China more than 2500 years ago, it is believed to be the oldest board game continuously played to the present day [16] [17]. It differs from Checkers, Chess, and Shogi, as pieces are only placed on the board, not moved, and there are no different types of pieces. The standard board size is 19x19, although it can be played with different sizes.

Go features a vastly higher complexity even than Chess and Shogi, due to its board size and branching factor [18]. AIs created in the 1980s to early 2000s were not better than an intermediate level human player. After 2007, Go algorithms improved with the application of Monte Carlo tree search [19], but could still be beaten by professional players. In 2015 DeepMind produced AlphaGo, which managed to beat in 2017 the best player in the world [20] [21].

Similarly to its Chess counterpart AlphaZero, it's an application of Machine Learning and Deep Learning, hence the same reasoning about the application of a similar structure to Advance Wars applies.

2.2. Strategy Videogames

As mentioned before for Advance Wars, strategy videogames in general feature a deeper complexity in terms of space of states, with respect to board games. Strategy videogames can have a finer granularity for some mechanics, because the videogame itself takes care of in terms of automation and calculations, such as getting a certain amounts of coins each turn based on the owned properties, damage calculation, and so on. These type of mechanics cannot be applied to a board game because they would be too complex to follow and evaluate frequently. Board games in general tend to create a very simpler representation of a given concept because of this reason.

One example is Chess vs Advance Wars: units in Chess are a simpler representation in terms of gameplay of a unit in a battlefield (taking away the context of classical vs modern war, which is not strictly impactful here): in Chess units are different almost solely on their type of movement, while in Advance Wars a unit has a set of characteristics which create a more realistic representation of the corresponding military unit, having a weapon with ammunitions and its properties, a type of movement which allow it to move or not onto specific terrains, and so on. This comes with the fact that every variable in a videogame is in general less impactful on the overall outcome, as changing slightly how a piece moves in chess is most of the times very impactful on the optimal strategy to

adopt, but in Advance Wars a slight change in any parameter most likely doesn't affect a lot the overall outcome of the game.

Some videogames however can take advantage of real time computation, which is a feature no board game can make use of to simulate battles. Such strategy games are called Real Time Strategy games (RTS).

2.2.1. StarCraft II

StarCraft II is one of the most famous RTS, also present in the competitive scene, developed by Blizzard Entertainment, published in 2010. Specific mechanics asides, it shares a lot of similarities with of most of RTS, such as in Age of Empires. In particular a set of resources gatherable by specific units on the game map, units produced by spending these resources and unlocked/powered through skills, techs and/or buildings.

RTS games require a different set of skills, as timing is a factor that doesn't exist in board or TBS games, such as being able to adjust the strategy and doing both micro and macro management on the fly. Furthermore, in any typical RTS the full state of the game is not known. In StarCraft there is Fog of War, but in general the full state is not known because the player views always only a portion of the map, which limits the timing of actions they can perform.

On this topic, the first version of AlphaStar, a deep NN made by DeepMind (the same one which made AlphaZero), won in December 2018 against StarCraft pro players, but was considered unfair, mainly due to the fact that the AI had access to the complete view of the map all the time [22].

Later AlphaStar was adjusted to be limited like a human player in their view of the game and input rate, and in August 2019 the AI managed to reach rank Grandmaster, outperforming 99,8% of human players [23].

2.2.2. Other TBS games and frameworks

As mentioned, TBS games' AIs literature is not as vast as RTS or board games one. Among the titles worth to be mentioned, there is for sure Civilization, which has been subject of study [24] [25].

The game is however deeply different from War chess -like games, tackling different types of challenges and problems.

A recent attempt of creating a framework for TBS games worth to be mentioned is Tribes [26] [27]. Tribes is more similar to games like Polytopia, and shares some similarities with games like Civilization or Age of Empires, in the sense that there

are also actions that can be performed by cities and technologies to be unlocked within a game, mechanics which are not present in standard War chess games. There are however some similarities[2]. As today the framework has yet to be used by third parties for research on this topic.

Figure 2-1 Tribes (left), Polytopia (right) [27]

A similar case is STRATEGA [28], another recent framework (2020) for general strategy games. It differs from Tribes because its focus is more on units, rather than civilization, technologies or cities like Tribes. This makes it interesting for other types of studies, relative to games more similar to pure War chess games. As today it is in the same situation as Tribes, and there aren't third party studies which adopts this framework.

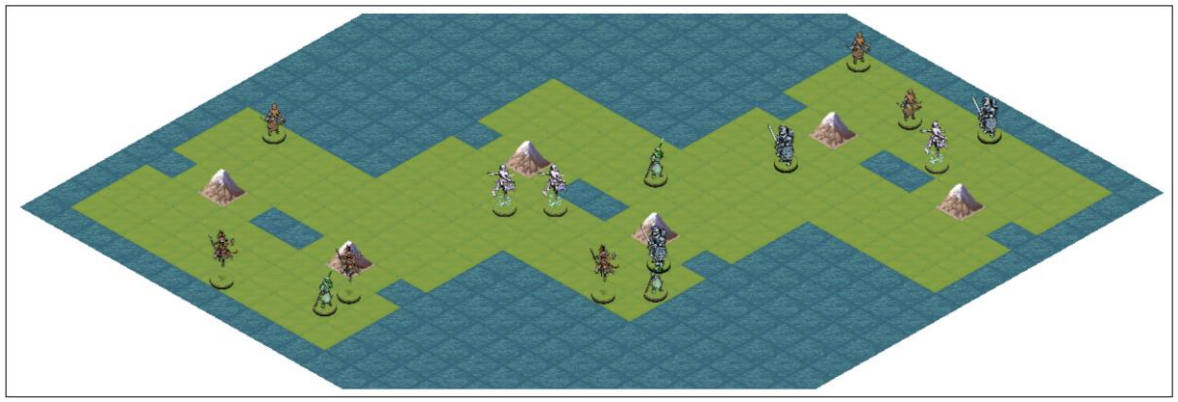


Figure 2-2 STRATEGA [28]

[2] Similarities are in particular creation of units, attack and counterattack actions from and to units, and choosing the actions in any order in a player's turn.

3 Advance Wars: rules

This chapter contains a detailed description for the game on which this thesis project is based, Commander Wars, which is a C++ open source [29] clone of Advance Wars™, a series of video games developed by Intelligent Systems and published by Nintendo.

Commander Wars includes all the content of all 4 games of the series (*Advance Wars, Advance Wars 2: Black Hole Rising, Advance Wars: Dual Strike, Advance Wars: Dark Conflict*), plus some extra custom content, but the core mechanics are the same of the official games, so the description of the game itself works for all games, although the last has more differences w.r.t. the previous three. The main difference is the number of units and terrains each game contains, with Commander Wars having all of the 4 chapters plus some extra custom units.

3.1. General description

Advance Wars is a 2D Turn-Based Strategy game (for short TBS), where from 2 up to 4 players control each one his[3] army of various military units (although in the open-source version more than 4 players can play).

Players control their own army through a character, called CO (Commanding Officer), which gives passive buffs/debuffs to their army, as well as 2 active abilities, so in general they require a different playstyle and won't be covered by this research.

The objective of a standard match is to remain the last army alive. An army is defeated when its last unit is destroyed or when its HQ (Headquarters) is captured by an enemy unit.

[3] The Player will be referred only as "He" for readability purposes, avoiding the use of the gender neutral they (for better clarity) or the She/He option each time. Using one of the other could have been chosen as well, but "He" has been randomly selected among the two.

Figure 3-1 [AW rules] example of gameplay

3.2. Quick Reference

Here we provide a quick reference for the rules of Advance Wars. The details of all the rules are in the subsequent sections.

- **2 or more players**, each controls an army.
- The game plays in a **2D square-grid environment.**
- In their **turn**, players can move their units in whichever order, one action at most per unit. First a unit is moved, then it may perform an action, such as attacking or waiting. After an action the unit cannot be moved until next turn.
- **Attacks** performed by units can be direct or indirect. **Direct attacks** target one bordering (on axis not diagonally) enemy unit and can be done after moving. **Indirect attacks** have a range but can't be performed if the unit moves in that turn. Some attacks require **ammunition** to be used. Each unit has a different number of ammos.
- **Attack effectiveness** is **proportional to the health** of the unit. A unit which is directly attacked will **counterattack** after the attack with the remaining

hp if it can do a direct attack itself. Base effectiveness depends on the attacker's weapon and defender, described in a **damage chart.**

- **Movement** of units is only vertical and horizontal. Each unit can move of a different amount each turn.
- Each unit also has a **movement type** (foot, tires, wheels, air, ship, etc.), which determines how much they can move on certain terrains.
- **Terrains** are the type of tile in the grid. They **determine movement of units** (they determine for each movement type a cost for entering that tile) and can give different amounts of **defense bonuses**.
- Units are divided into **Land**, **Air** and **Naval units.**
- Units have a **fuel amount**, which is consumed at each movement by the movement cost for each tile. When it reaches 0, the unit can't move anymore if it's a land unit (but can still attack), or it is destroyed if it's an air or naval unit.
- **Cities** are special terrains which can be neutral or **belong to a player**. They can be **captured** (in at least 2 turns) by a player by using an infantry unit and using the capture command on it. If they are captured they have 2 properties: give a **bonus income** (1000 Funds) each turn, and **repair** (give 2 hp) the **friendly land unit** on it at the start of the turn (paying a cost). They also give 3* of defense.
- **Factories** function like cities but additionally they can be used to **build** a **new** land **unit**, by paying its cost (in Fund). The newly built unit can't perform an action the same turn it is built. **Airports** and **Ports** work similarly but can build and repair only Air units and Naval units respectively.
- **HeadQuarters** (HQ) are a special terrain, similar to a city but there's one per player and **if captured** by an opponent **the owner loses**, and all its cities pass under control of the capturing player.
- **Win conditions**: A player wins if is the last player remaining in the game. Players lose by getting their **HQ captured** or by getting their **last unit** alive **destroyed**. In this latter case, all their cities and their HQ become neutral cities.

Extra mechanics which are **not considered for the thesis** but present in the game:

- **Special actions** of certain units can be performed:
  - **Carry other units** in them and unload on other tiles. For instance, the transport copter can carry 1 infantry unit through air. If the carrying unit is destroyed, the carried unit is destroyed as well.
  - **Supply** other units with fuel and ammos.
  - **Repair** other units, paying the repair cost.

- **Fog of War** (FoW): players can see the types of terrains but not the units in them, and need nearby units to have visibility on tiles. Additionally, some terrain can conceal units until there's an adjacent enemy unit to discover them.
- **Commanding Officers** (COs): one of the core mechanics in the game. Players play as a CO, which alters considerably their armies, by giving passive bonuses and maluses to all their units or having special properties (e.g., more power to indirect units but less on direct ones). Additionally, they can use a special power when charged. Each CO is unique.
- **Weather**: it can be set before each match and can be randomized each turn. It gives additional conditions on movement or visibility (e.g., rain creates FoW even if is not a FoW match, snow doubles the fuel and slows down movement in some terrains, sunny is the default).

3.3. Basic mechanics

3.3.1. Introduction

Advance Wars is played in a 2D top-down square-grid.

Each player can move all their units, one at a time, and make them do at most 1 action each. Players can also build new units in their turn only in specific buildings, based on their cost.
Whenever they want, they can pass the turn to the next player.

Units can attack other units, and units can be destroyed if they take too much damage in total.

A player wins if he controls the last army remaining in the game.

A team battle is possible. Players can be on any team, and if they are on the same team, they can't attack or capture each other's cities, but act as different players in every other aspect (they can't share funds/repair other armies' units for instance)

Figure 3-2 [AW rules] example of 4p vs in AW 1

3.3.2. Playing a Turn

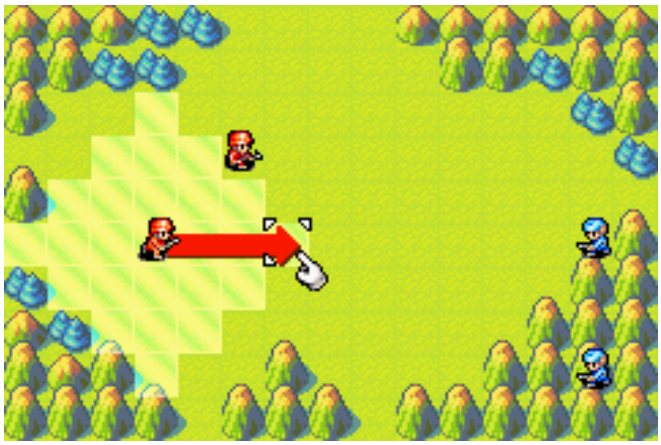


Figure 3-3 [AW rules] moving a unit

In their turn, a player can choose whichever units he controls (in the image above the current player controls the red units) and make them move or perform actions.

A unit which has performed an action becomes inactive and cannot be selected until the current player's next turn. A unit can move or not but will terminate its turn always and only when it will perform an action. Note that waiting is considered an action.

Performing an action will always terminate the turn, so a unit can't perform an action and then move.

Figure 3-4 [AW rules] moved units

Units which have done an action are colored out. In the example above the red units with 8 and 9 on them have already used their action, while the other 2 red

units haven't. Opponent's units (blue here) can't move since it's not his turn, so they are not colored out.
A player can pass whenever he wants, even if he didn't move any unit.

3.3.3. Movement, Terrains, Units

Movement

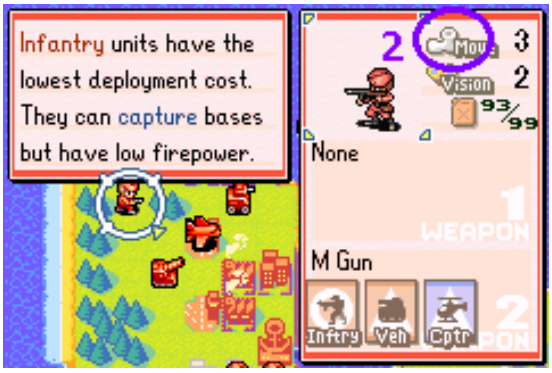
Units have a movement value (image below, 1) which determines how much it can move each turn.

(a) Infantry has 3 movement points (b) So it can travel 3 spaces

Figure 3-5 [AW rules] movement points

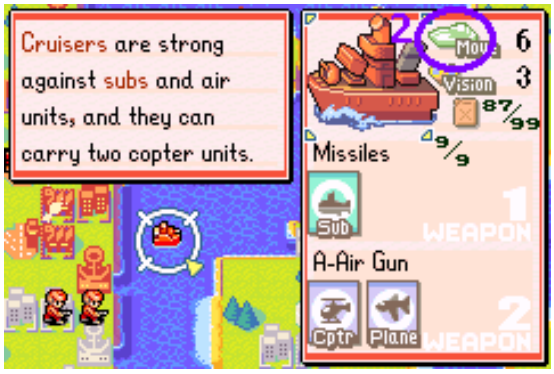
For instance, an infantry has 3 move points and can move up to 3 tiles, only horizontally or vertically.

Units spend 1 or more movement points to travel each tile in the grid.
Different units have a different movement type (images below, 2), as can be seen in the next figures.

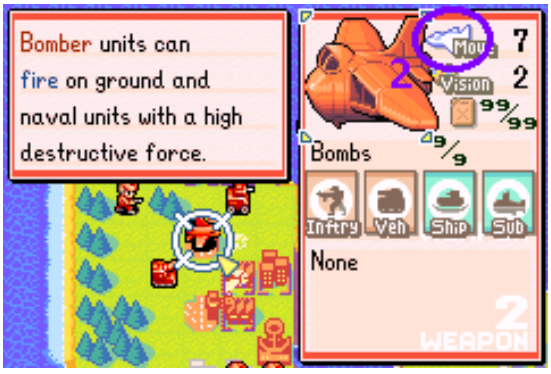


(a) Infantry movement type: feet

(b) Recon Movement type: tires



(c) Cruiser movement type: ship



(d) Bomber movement type: air

Figure 3-6 [AW rules] different movement types

This movement type affects how much movement points a unit will have to spend to enter a specific tile, as well as if the unit is able to traverse a tile. The cost is determined by the type of **terrain** present on a tile.

Terrains

Terrains are the type of tile of a map. They give defense to units in them (see later subchapter) and determine the movement cost of each movement type in the game.

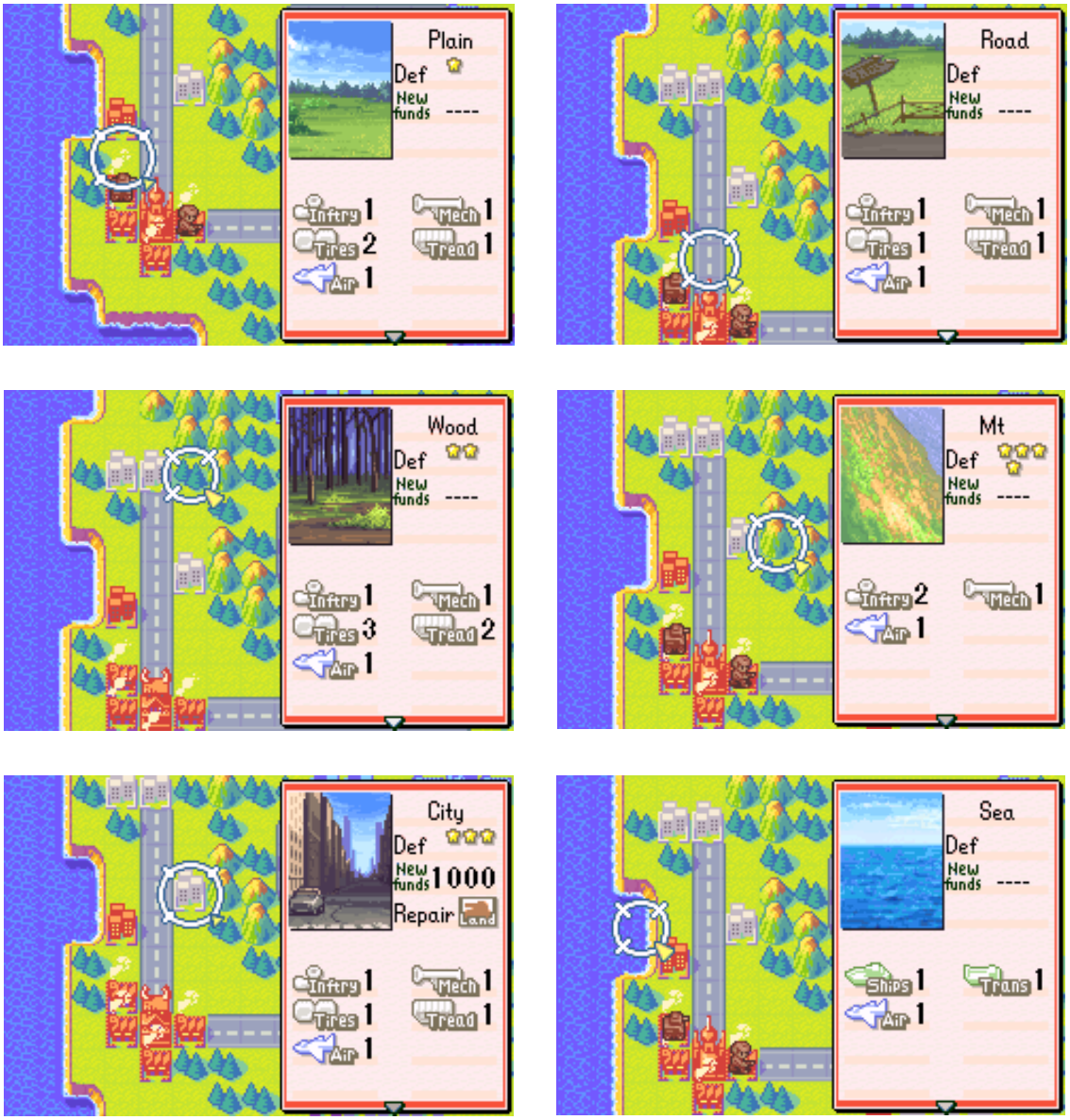


Figure 3-7 [AW rules] common terrains and movement costs

Above the most common types of terrains: plain, road, woods, mountain, city, sea. As is visible for instance only infantry (mechs are a type of infantry) can go on mountains, units with tires have more trouble moving outside road and cities, treads move well enough everywhere on ground, air units can go anywhere and ships can only traverse sea.

Unit Types

There are 3 types of units in the game: **Land units, Air units** and **Naval units.**

They differ mostly for the type of terrain they can travel: ground for land units, sea and rivers for naval units and everywhere for air units. Air units however don't gain the terrain defense bonus, and since no unit can traverse a tile where there's an enemy unit, also air units cannot traverse tiles with enemy ground units.

Fuel

Fuel is a resource every unit has in different quantities, and represents how much in total that unit will be able to move before having to be resupplied.

For each movement point a unit spends, the fuel decreases by 1. When the fuel reaches 0, a unit cannot move anymore. Additionally, since movement is calculated using fuel, if a unit for instance could move 3 tiles in 1 turn but has 2 units of fuel, then it will be able to move only 2 tiles (or 1 tile with cost 2 for that unit, like treads in forest or infantries in mountains).

Different types of units behave differently w.r.t. fuel and when is over:

- **Land units** consume fuel only when they move, 1 for each movement point spent. When fuel is over, they cannot move anymore and must be resupplied to move again. They can still attack and perform all other actions, like capture.
- **Naval** and **Air units** consume 1 fuel when they move for each movement point spent, and additionally they consume some fuel at the start of each turn, depending on the unit. Furthermore, if a naval or air unit starts its turn with 0 fuel (before it must be decremented by its daily usage), it gets automatically destroyed.

Figure 3-8 [AW rules] fuel amount of various units

3.3.4. Combat System

Health

Units have a **health** (HP, Health Points) status represented in percentage, for a total of 100%. When the health reaches 0%, the unit is destroyed. In the game is shown as a single digit number in the corner of the unit sprite, since the actual HPs used for damage calculations are a value which is the percentage floored to the tens (hence goes from 0 to 10).

Attack

Units can deal damage by choosing the **Fire action**, which is an attack action.

Attacks can be of 2 types: **Direct** and **Indirect**. The differences will be explained later.

The damage dealt by an attack depends on the attacking unit, which has a table of effectiveness against other units, and the category to which the defending units belong.

Figure 3-9 [AW rules] effectiveness of a unit's weapon example

For instance, recon units can directly attack infantry, against which they deal good damage, and vehicles and copters, against which perform poorly (image above).

Although in the game the indication is only “good” or “bad”, a full-HP unit of one type will always deal (on average) the same amount of damage to the same type of enemy unit.

E.g., a tank will always deal 55% to another tank (without additional defense) or 75% to an infantry. The actual damage is subject to a small degree of RNG, but the average damage is fixed given the attacking and defending units.

Figure 3-10 [AW rules] effectiveness of a unit's weapon example 2

The proper RNG name mechanic is called **Luck**, and in general luck gives a variation in damage from ±0 to +9% of damage at most, so it's only positive in general.

Here's an example of a damage chart. This is based on *Advance Wars: Dual Strike*.

DEFENDING UNITS \ ATTACKING UNITS	Infantry	Mech	Recon	Tank	Md Tank	Neotank	Megatank	Artillery	Rockets	Anti-Air	Missiles	Pipe-runner	BCopter	Fighter	Bomber	Stealth	Cruiser	Sub	Bship	Carrier
Infantry	55%	65%	70%	75%	105%	125%	135%	90%	95%	105%	x	95%	75%	x	110%	90%	x	x	95%	x
Mech	45%	55%	65%	70%	95%	115%	125%	85%	90%	105%	x	90%	75%	x	110%	90%	x	x	90%	x
Recon	12%	85%	35%	85%	105%	125%	185%	80%	90%	60%	x	90%	55%	x	105%	85%	x	x	90%	x
Tank	5%	55%	6%	55%	85%	105%	180%	70%	85%	25%	x	80%	55%	x	105%	75%	x	x	85%	x
Md Tank	1%	15%	1%	15%	55%	75%	125%	45%	55%	10%	x	55%	25%	x	95%	70%	x	x	55%	x
Neotank	1%	15%	1%	15%	45%	55%	115%	40%	50%	5%	x	50%	20%	x	90%	60%	x	x	50%	x
Megatank	1%	5%	1%	10%	25%	35%	65%	15%	25%	1%	x	25%	10%	x	35%	15%	x	x	25%	x
APC	14%	75%	45%	75%	105%	125%	195%	70%	80%	50%	x	80%	60%	x	105%	85%	x	x	80%	x
Artillery	15%	70%	45%	70%	105%	115%	195%	75%	80%	50%	x	80%	65%	x	105%	75%	x	x	80%	x
Rockets	25%	85%	55%	85%	105%	125%	195%	80%	85%	45%	x	85%	65%	x	105%	85%	x	x	85%	x
Anti-Air	5%	65%	4%	65%	105%	115%	195%	75%	85%	45%	x	85%	25%	x	95%	50%	x	x	85%	x
Missiles	26%	85%	28%	85%	105%	125%	195%	80%	90%	55%	x	90%	65%	x	105%	85%	x	x	90%	x
Pipe-runner	5%	55%	6%	55%	85%	105%	180%	70%	80%	25%	x	80%	55%	x	105%	80%	x	x	80%	x
Oozium	5%	30%	20%	20%	30%	35%	45%	5%	15%	30%	x	15%	25%	x	35%	30%	x	x	20%	x
Bcopter	7%	9%	10%	10%	12%	22%	22%	x	x	105%	120%	105%	65%	100%	x	85%	105%	x	x	115%
TCopter	30%	35%	35%	40%	45%	55%	55%	x	x	105%	120%	105%	95%	100%	x	95%	105%	x	x	115%
Fighter	x	x	x	x	x	x	x	x	x	65%	100%	65%	x	55%	x	45%	85%	x	x	100%
Bomber	x	x	x	x	x	x	x	x	x	75%	100%	75%	x	100%	x	70%	100%	x	x	100%
Stealth	x	x	x	x	x	x	x	x	x	75%	100%	75%	x	85%	x	55%	100%	x	x	100%
Black Bomb	x	x	x	x	x	x	x	x	x	120%	120%	105%	x	120%	x	120%	120%	x	x	120%
Lander	x	x	x	10%	35%	40%	75%	55%	60%	x	x	60%	25%	x	95%	65%	25%	95%	95%	x
Cruiser	x	x	x	5%	30%	30%	65%	50%	60%	x	x	60%	25%	x	50%	35%	25%	25%	95%	x
Sub	x	x	x	1%	10%	15%	45%	60%	85%	x	x	80%	25%	x	95%	55%	90%	55%	95%	x
Bship	x	x	x	1%	10%	15%	45%	40%	55%	x	x	55%	25%	x	75%	45%	5%	65%	50%	x
Carrier	x	x	x	1%	35%	15%	45%	45%	60%	x	x	60%	25%	x	75%	45%	5%	75%	60%	x
Black Boat	x	x	x	10%	10%	40%	105%	55%	60%	x	x	60%	25%	x	105%	65%	25%	95%	95%	x
BH Structure	1%	15%	1%	15%	55%	75%	125%	45%	55%	10%	x	55%	25%	x	95%	70%	x	x	55%	x

Figure 3-11 [AW rules] damage table of Advance Wars: Dual Strike

Attack/HP scaling

Attack effectiveness also scales with the attacker's HP, proportionally. A unit with 5 HP (50% of life) will deal 50% damage. (since HP represents how much of the unit is left).

In the next figure, an example of Damage scaling with HP: if a tank has 4 HP it will deal 40% of the original damage, which was 55%, and hence it will do 22% damage.

Figure 3-12 [AW rules] example of damage scaling with HP

Direct attack and Counterattack

A **direct attack** is an attack performed by a unit, possibly after a movement, against an adjacent enemy unit.

A direct attack will trigger after the attack a **counterattack** by the defending unit, only if the defending unit is a unit which can perform a direct attack itself (these are called direct units).

A counterattack works exactly as an attack in terms of damage, but since it's performed *after* the first attack, its damage is scaled by the remaining HP of the defending unit. This also means that if the defending unit is destroyed, the attacker won't receive any counterattack damage.

An example below, before and after the attack action performed by the red tank:

Figure 3-13 [AW rules] counterattack example

The attacking tank (red) attacks the blue tank, dealing to it around 55% damage, possibly 60% damage. The blue tank counterattacks, dealing 40% of the same damage, which is 4/10 * 55% = 22% (plus a bit of RNG), and the hence red tank remains with ~80% HP.

Indirect attacks

An **indirect attack** is instead an attack which can hit far units.

Indirect units have a **range** property, which indicates the tiles they can attack to, both as minimum and maximum.

Differently from direct units, indirect units can't move in order to attack in the same turn.

Furthermore, an indirect attack doesn't trigger a counterattack, even if the attacked unit has a range which would allow it to fire at the attacker.

Below is an example of an indirect unit with range 2~3 (minimum 2, maximum 3), which can attack any one unit in the selected range, here a tank or an infantry.

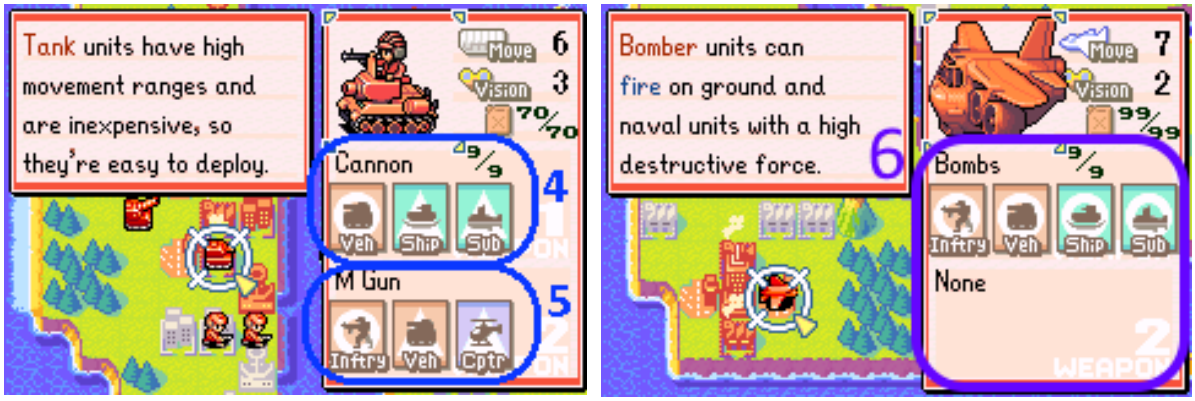
Figure 3-14 [AW rules] indirect attack example

Ammos

Units can have up to **2 weapons** equipped, weapon 1 (image below, 4) and weapon 2 (5). In some cases, a weapon requires 1 **ammo** to be used, and they can carry a different maximum number of ammos, depending on the unit.

If there are no more ammos for a weapon, that attack can't be performed, and if a unit has no other weapons it won't be able to attack (6).

In the example below on the left, light tanks have 9 ammos for the cannon, which is effective against vehicles and can deal damage to ships, and a basic weapon which can attack land units and copters. Note that although both weapons can attack vehicles, if the tank has at least 1 ammo it will always attack with the cannon (4) against a vehicle, so choosing the weapon before an attack is not an option, as the game will always choose the most effective weapon.



(a) a unit with 2 weapons, one with limited and one with unlimited ammos

(b) a unit equipped only with 1 weapon with limited ammos

Figure 3-15 [AW rules] weapons and ammos example

Defense from terrains

Terrains give a defense bonus to units on them, with the exception of air units, which never get a defense bonus (neither a defense malus, which some rare special terrain might have).

Terrains have a value indicated in stars which goes from 0 to 4 (7). Each star gives a reduction of 10% of the received total damage.

This reduction however is proportional to the health of the unit: if a unit has 6HP (60% of life), then it will reduce incoming damage of 6% per star.

Figure 3-16 [AW rules] highlight of some terrain defense value

Damage Formula

The final damage formula is calculated as follow [30]:

$$f = \left(\left(b * \frac{s}{d}\right) * (a * 0.1)\right) - r * \left((t * 0.1) - (t * 0.1 * h)\right)$$

(final damage is indicated in % in game)

f = Final damage

b = Base damage (in damage chart)

s = Strength modifier of CO (COs are explained later but are an extra mechanic)

d = Defense modifier of CO (same for Defense modifier)

a = HP of attacker

r = Defense rating (number of stars)

t = Total damage. This is the first part of the equation, (b * s / d) * (a *. 1).

h = HP lost by defender (from 0 to 9)

3.4. Advanced Mechanics

3.4.1. Basic buildings and capture

Buildings are special terrains which have additional interaction properties. The basic buildings have an additional property of belonging to a player or being neutral.

A building neutral or belonging to an opponent can be captured by infantry units.

Capture is an action that infantry units can perform on buildings to convert them to their team. When under capture, buildings have 20 "HP", and infantry units which use the capture action will decrease the building's HP by their own HP value. When the building HP reaches 0, the building is captured and will belong to the capturer player.

Since the max value of HP of a unit is 10, it will take at least 2 turns to capture a building in general, and at most 20 turns if the capturing units has 1 HP.

If the unit leaves the building or is destroyed while capture is not complete, the capture will fail, and the building's HP will return to 20

In the images below it can be seen:

(a) The capture action;

(b) A unit with 10 HP after the capture command will leave the neutral building with 10 HP;

(c) The captured city is now a red player's building;

(d) A unit with 6 HP after the capture command will leave the neutral building with 14 HP, and will require 4 turns in order to capture it (unless the player merges units. See join units, 3.5.1).

(a) – capture action

(b) – capturing with a 10 HP unit leaves the city with 10 “HP” left

(c) – next turn city is captured, its HP are now 20 again

(d) capturing with a 6HP unit leaves the city with 14 “HP” left

Figure 3-17 [AW rules] capture action

3.4.2. Cities

Cities are the most common and basic type of building. They can be captured, and they have some additional properties w.r.t. a normal terrain (they also give 3* of defense as a normal terrain).

1. Each city possessed by a player increases the income by 1000 War Funds (see next chapter) a player receives at the start of each turn. If a player possesses 4 cities, he will earn 4000 extra War Funds each turn.
2. A player’s unit which starts the turn on a player’s city, gets a free **supply** (restores all fuel and ammos) and is **repaired** by 2 HP, which means it restores 2HP to the normal cap of 10HP. The repair will use War Funds

proportionally to the unit's cost (see next chapter) and the HP recovered. If a unit recovers 2HP then the repair will cost 20% of the original unit's cost. The repair cost is based on the true percentage of health of the unit and not the HP.

3.4.3. War Funds and Factories

War Funds (WF, or just Funds) represent the main resource of players in general. War Funds act like money, and can be used to create new units or repair existing ones.

Before the player can move any unit, each turn a player receives an **income** in War Funds, which is determined by the buildings he possesses. The HQ gives a standard 1000 WF, and each possessed city an additional 1000. Factories, Ports and Airports give an additional 1000 each.

There's no WF (reasonable) cap.

Factories (also called Bases) are a special building which work like cities and additionally allow the player to build land units. To do so, the player selects the building, which must have its tile free from any unit, and choose a unit from the menu. Each unit has its own fixed cost.

Figure 3-18 [AW rules] factory and building units

The unit built will spawn on the same tile of the factory and will be inactive, meaning that in normal circumstances at most 1 unit can be built from each factory.

Ports (also called Harbors) and **Airports** are similar to the factories but produce Naval and Air units respectively, and are the only type of building which can repair Naval and Air units (respectively).

Figure 3-19 [AW rules] port and airport

3.4.4. Headquarters and Win Condition

Headquarters (**HQ**) is the most important building for a player, since if it's captured by any opponent, the match is immediately lost. Apart from this, it works like a city, except that it gives 4* of defense instead of 3*. In general, the HQ must be protected at all cost, and since it's possible to just capture it a basic strategy is to place at least 1 unit on top of it so that no infantry can sneak over the HQ and capture it while the main forces are on the front line.

In a multiplayer scenario the owner of a captured HQ will get defeated, and his HQ will turn into a city.

Win Conditions:

A player wins if is the last player remaining in the game.

A player is defeated and removed from the game if his HQ is captured or if its last unit alive is destroyed in any way.

If a player captures an enemy HQ, that player loses and additionally all his cities become the capturer player's property.

If a player instead has his last unit destroyed by anyone, his buildings will become neutral no matter who was the attacker.

Below an example of a Headquarter. After it has been captured, it becomes a city and all the blue player's buildings become the red player's property.

(a) before capture (b) after capture

Figure 3-20 [AW rules] HQ capture example

3.5. Additional Mechanics

Here are listed some extra mechanics which are in the game but are **not considered for the thesis** project, as well as the reason why they are not considered.

3.5.1. Special actions

Some units can perform other special actions, described here.

Join units

If a unit is moved onto a unit of the same type which has not full HP, it can be joined with it. This will add their HP, never exceeding 10. The excess HP is converted in War Funds depending on the unit's base cost. A unit can join a unit which has moved or that has still its action available, but in any case the 1 joined unit remaining will have its action spent.

Load/Unload units

Some units are able to load other units inside them. **Load** is an action performed by the unit which has to be carried. The carried unit must go on the same tile of the carrier unit and use the Load action.

Since the action is performed by the carried unit, the carrier unit can move in the same turn, and the carrier unit doesn't have to be active to load units.

To unload the carried unit, the carrier, also after its movement, can choose the **Unload** (or **Drop**) action, which unloads the carried unit in an adjacent tile, chosen by the player. The unloaded unit is inactive. The tile where the unit is dropped must be a tile where that unit can walk on (e.g. a unit can't unload an infantry on sea).

A carrier unit can choose to drop the carried unit on any of its turns. This is useful in general to boost movement of units or to move across sea or air other units. If the carrier unit is destroyed when carrying another unit, that unit is destroyed as well.

Here's a simple example with the most common unit with the ability to load, which is the APC. In the example, the infantry goes on the same tile of the APC and uses the load action.

The APC now contains that infantry, then moves to some tiles on the right and uses the Drop action, chooses the tile on which it will drop the infantry and confirms. Now both units are inactive and the infantry has moved more tiles than it could have in 1 turn.

Figure 3-21 [AW rules] load/drop example

Depending on the unit more than 1 unit can be loaded at the same time on the same unit. For instance, Landers (which are transporter ships) can carry any 2 ground units and unload 1 or both on the same turn. If the carrier carries units of different types than itself (like Landers or Transport copters which can carry 1 infantry unit), in order to drop the carried unit, the unit itself must be also on a tile where the carried units can stand.

Air units can be dropped anywhere by design and naval units can't be carried, so the carrier in general must be on a tile which allows ground movement. For ships there is the shore, which is a terrain that can be walked on both by land and sea units.

Supply

Supply is an action performed by APCs. It refills all fuel and ammos to all adjacent friendly units, both active and inactive. APCs also do an automatic supply at the start of each turn.

APCs can supply any type of unit, as long as they are adjacent, so they can also supply fuel for naval and air units.

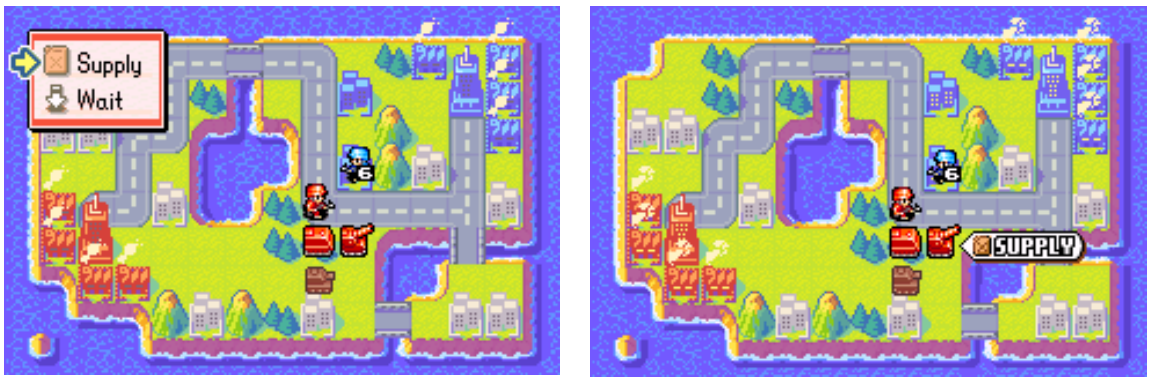


Figure 3-22 [AW rules] supply example

Repair

The black boat is the only unit which can repair other units. The repair action targets only 1 unit and repairs the target of at most 1HP, draining the cost (up to 10% of the original unit cost) from the player's funds. If the player hasn't enough funds, the repair action won't repair the unit or will repair it only partially.

The repair action performed by the black boat also acts as a supply, but targets only the repaired target unit, and doesn't work passively as for apcs.

Hide

Submarines can perform a dive action which makes them disappear from the opponent view. When submerged they can't be seen unless an enemy unit is adjacent to them, and can't be attacked anymore by weapons that can attack ships, but only with special weapons (in particular only the cruisers and the submarine itself).

They also consume more fuel, making them more susceptible to sink if they remain submerged for too long.

Stealth Fighters work in a similar way but are air units.

3.5.2. Commanding Officers

Commanding Officers (shortened **CO**s) are arguably one of the main mechanics of the game.

Each player which controls his army actually controls a CO.

A CO is a character which will modify statistics in various ways to define his/her style of combat.

Some examples of COs:

- Max has stronger direct units but weaker indirect ones, and the weaker indirect units also have reduced range.
- Colin has units' costs reduced by 20%, but all its units are a bit weaker than normal.
- Lash's units increase attack by 5% for each star of defense they have (10% in AW 2)

- Jess has increased firepower for vehicles units but has weaker naval and air units.
- Grimm's units have 130% firepower but 80% defense.
- Andy has all standard units.

Since different COs can require different tactics, and the thesis is focusing on a generic AI behavior, the training will be performed without the use of COs. Locally Commander Wars has the possibility of playing without a CO, differently from the real games (although one could just use Andy, which is the balanced CO which has standard behavior for everything).

In Advance Wars DS and hence in Commander Wars it is possible also to use **two COs** per player, which at the end of each turn can choose to switch his CO for the next turn. This mechanic is not used as well since neither one CO is used.

3.5.3. COs Power and Super Power

COs have an additional mechanic: in addition to providing passive bonus/malus to their units, they have special abilities which can be charged and activated when they are ready at every moment in their turn.

COs during the fight charge their power meter depending on how much damage they inflict and receive in terms of funds (if a unit which costs 9000 is damaged by 5HP, they will charge the meter proportionally to 4500 funds, and so on). Having units damaged boosts more than damaging.

The meter has 2 levels of charge: at the first level the player can activate the CO's power, at second level he can activate the power or the superpower.

Power and Superpower are unique and depend on the CO (and the game), but all in general improve a bit defense and/or offense of all units. Some examples are (in square brackets is the superpower version)

- Andy: Restore 2 HP to all friendly units, increase firepower and defense by 10% [restores 5 HP, increases firepower by 30%, defense by 10%, movement by 1]

- Grit: Increase range of indirect units by 1 and their firepower to 160%, and defense of all units to 110% [increases the range of indirect units by 2, the rest is the same]
- Sasha: Reduces the enemy CO's Power Meter proportionally to her funds, and gets a boost of 10% in attack and defense [Super Power doesn't decrease opponent Power Meter. Instead for this turn 50% of all damage dealt will be turned in War Funds. Get a boost of 10% in attack and defense.
- Sensei: Increases the firepower of all battle copters to 180% and creates Infantry units with 9 HP at every allied city, ready to move [Same but creates mech units with 9HP]

Again, since powers are very different and the purpose is to create a generic AI regardless of COs and powers (and also because to have a Power a player would need a CO), they are not taken into account for the training of the AI.

3.5.4. Fog of War

In Advance Wars there exists the possibility to play with **Fog of Wars** (FoW for short) on. In this mode all the terrains are always visible but not the units on them or who's the owner of non-player's buildings, or how many Funds opponents have. Units have a **vision** property which states how many tiles of range they can see for the player.

Furthermore, there are some terrains like woods which conceal the unit inside them until a unit goes on an adjacent tile. Because of the considerably different working of the game, w.r.t. information and favored units, it would be better to train 2 separate AIs and consider them as separate problems, hence FoW is not accounted for.

3.5.5. Weather

Weather is a mechanic that can change how the map will behave for all players. In a normal match, one can choose random weather which with some probability changes each day and stays so for 1 day, returning to sunny, which is the standard weather with no modifiers.

Weather effects examples are:

- Rain: increases movement cost in plains and woods for tires and treads. In AW DS creates FoW for the duration, and if FoW is on it reduces vision by 1

- Snow: increases movement cost for some tiles and types of movement and doubles the consume of fuel.

To learn a more stable behavior, the AI will be trained not taking into account weather, so all games will always have sunny weather.

4 The ADAPTA architecture

In this chapter we present the detailed structure of the implemented ADAPTA architecture and its components, in relation to the original concept and the additional developed features, and it is explained how it processes the game state and produces a set of actions to play a turn.

4.1. Original ADAPTA

The rationale behind the original conceptualization of the ADAPTA [3] is to create a modular structure which can be configured to work with different objectives and possibly a different number and type of modules. This allows to create Tactical (sub)Modules which focuses on only one aspect of the game. These modules are managed by a Strategic Module, which acts as an arbitrator between the Tactical Modules. The Tactical Modules must share a finite number of resources (or assets), namely in Advance Wars units (but also War Funds for building or repairing units), which means they could have conflicts if more than one tactical module requires the same unit. To obviate this problem, each module instead of simply requiring or not a unit {0,1}, proposes a bid for each unit [0,1] which should represent the utility gained by using that unit in the current turn, according to that module's objectives.

The Asset Allocation Module then tries to maximize "social welfare", by assigning the units to the modules based on all the bids.

Finally, Movement Order module chooses the turn order of each unit.

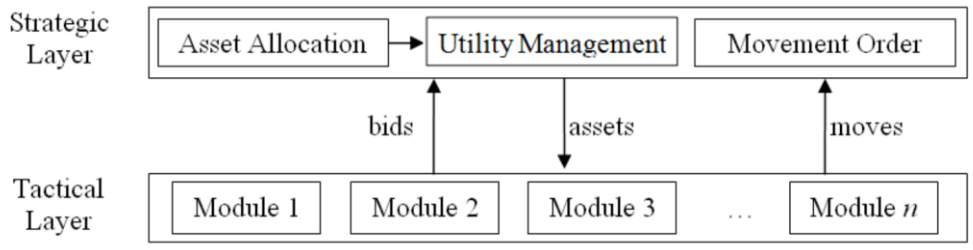


Figure 4-1 The ADAPTA Game AI architecture [3]

4.2. Implementation

The implementation adopted for a more real case scenario of the Adapta AI is at the core the same as the one presented in the previous subchapter.

At the macro level, the Adapta AI is composed of a number of Adapta Modules, which serve the role of the previously described Tactical Modules. Since however deciding which actions should be performed for each unit and choosing which unit to build are substantially different actions, The Adapta AI has two categories of Tactical Modules, which are:

- Adapta Modules, the ones in charge of deciding which action a unit should perform and for each one generating a bid each turn;
- Building Modules, the ones which can only control factories, airports, and ports (in general any structure which can produce units) and choose which unit should be produced.

A Strategic Module here decides the weights of the Adapta and Building Modules at the start of every match. Weights of modules are additional multipliers on top of the bids proposed by the Tactical Modules, and can be used to control the importance of every module and adjust in a finer way the overall adopted strategy, as described in 4.3.

A Movement Order Module was not developed, as the single Tactical Modules decide the ordering of the units it by ordering the bids in descending order, if they have to do so. From one side this doesn't allow a completely free bidding process, from the other side however it grants the ability for each Tactical Module to be sure to control units in their evaluated order.

A general Movement Order Module would have to be trained with specific modules, since changing the modules would likely produce suboptimal behaviors. The Movement Order Module in fact can't know the specific objectives of each Tactical Module, hence any order produced by it should in the end consider the modules under it. This is why the simpler approach of leaving this to the single submodules was adopted.

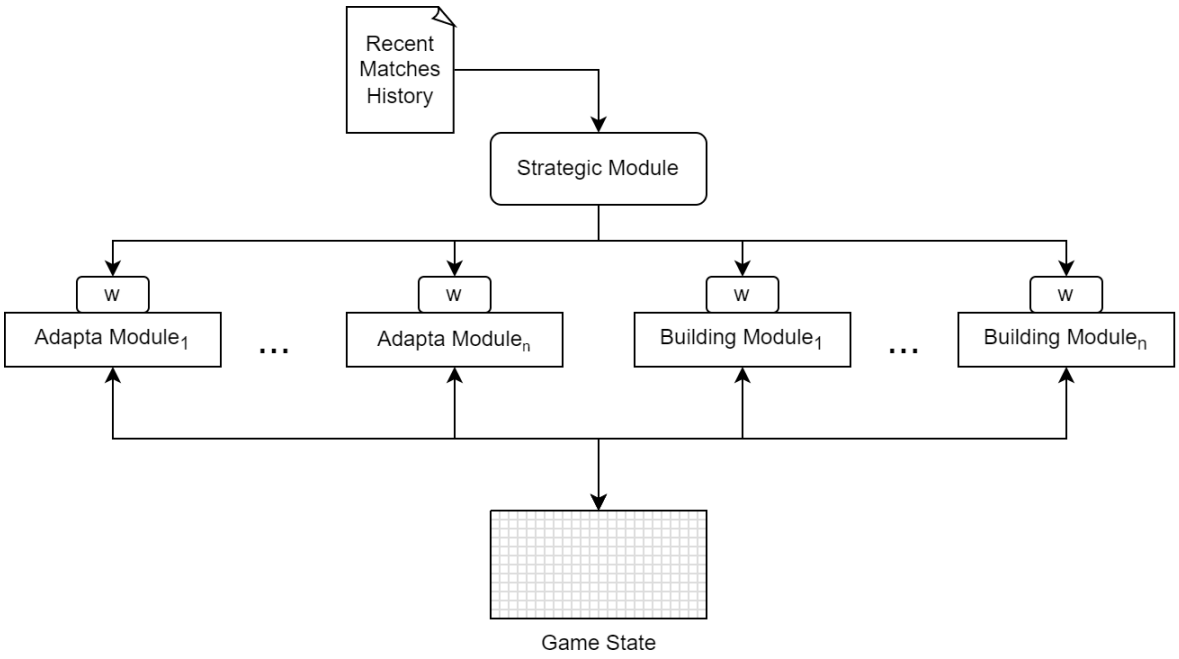


Figure 4-2 Implemented Adapta architecture

4.2.1. Adapta Modules

At the core an Adapta module has each turn to generate a bid for each unit it's able to use, and then perform an action with the unit requested by the Adapta AI when it's called to do so.

The specific working of each module is not defined at this point, since a Module can work with whichever behavior is programmed to or trained for. Each module could support the control of every unit in the game, only a subcategory of any kind, and could be able to perform with the same unit different actions.

A module could use infantry but only deciding Fire actions (attack), while another could also Capture buildings, or do only that. Since every unit can perform different actions depending on their type, and one could divide the game units in any number of subgroups which perform different actions, it is possible to create any type of module. This would also make it possible, in the domain of behavioral AIs, to create several smaller tactical modules, dedicated to subgroups of units or actions, which can also overlap in their domain of units used. This concept is not new, being similar to dynamic scripting [31], and could allow to focus on smaller subtasks, create a greater variety of tactics and divide the work among several developers easily.

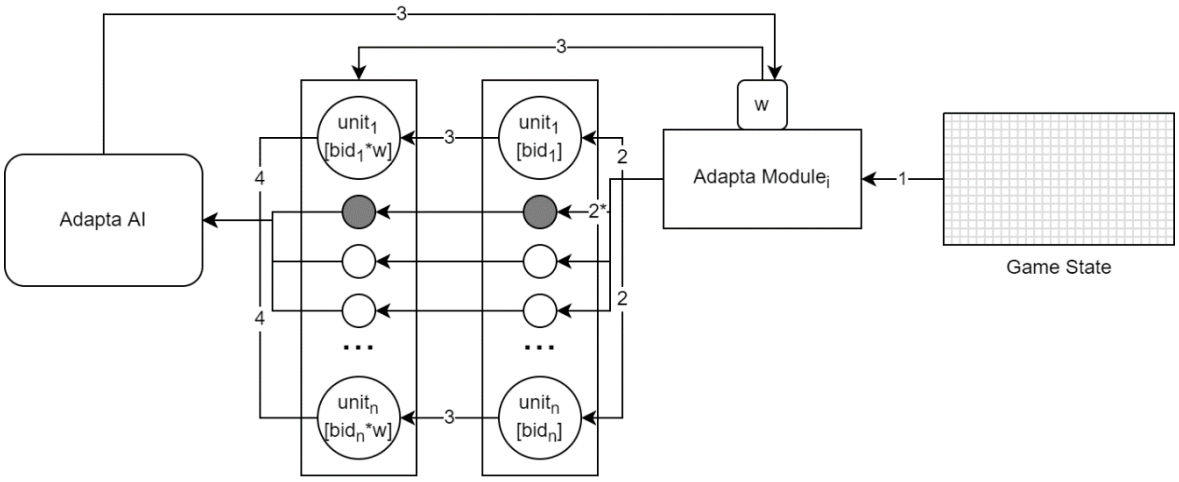


Figure 4-3 Adapta Module turn process (1/2)

At the start of each turn an Adapta module processes the map (Figure 4-3, 1) and generates bids for each unit on the map (in the range [0,1]) (Figure 4-3, 2). Due to the versatile system, a module may be trained to work with only some units. In such a case a module will always generate a bid of 0 for units it cannot use (Figure 4-3, 2*).

A bid of 0 should be used if and only if a unit is not supported. Even if the best move a module could perform for a unit results in a negatively evaluated outcome, the module has to use a minimum bid > 0 (for instance 0.001), since is still better to perform the least damaging move than to not move at all a unit (unless it is considered the optimal move by the module).

After having generated a bid, the Adapta AI gather the bids made by that module and multiplies them by the module's weight (Figure 4-3, 3), assigned by the Strategic Module (see 4.3), collecting then all bids from all modules (Figure 4-3, 4).

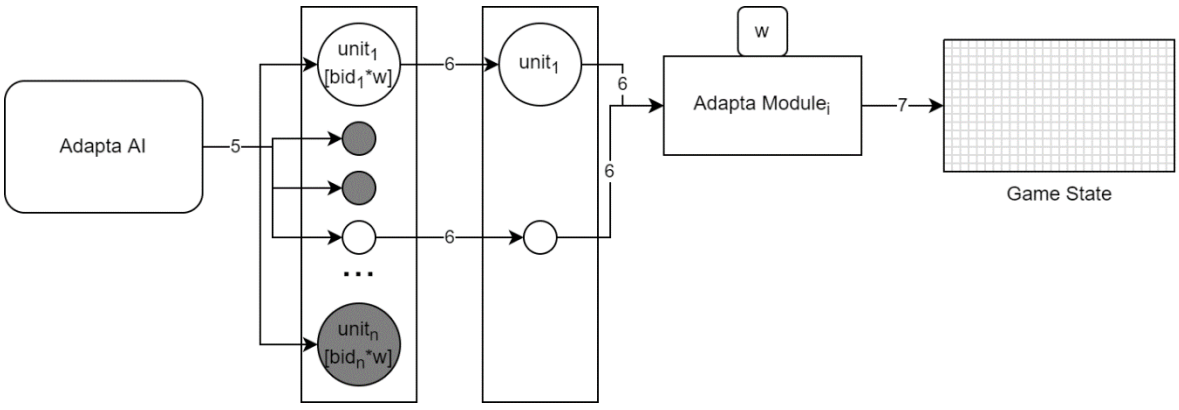


Figure 4-4 Adapta Module turn process (2/2)

The Adapta AI then assigns to every Adapta Module the designated units for the turn (Figure 4-4, 5). Units not assigned to Module *i* are not processed at all by that module to perform an action (Figure 4-4, 6). A module behaves only when requested by the Adapta AI and with the unit it chose to make it move. The module is free to re-process anything that has changed since the start of the turn and possibly alter the bids, but in any case it must make the unit requested by the Adapta AI perform an action.

It's guaranteed by the architecture that a module will never be ordered to process a unit for which it bade 0.

4.3. Strategic Module

The Strategic Module is the module which is in charge of developing an overall strategy for the match. To do so, in practicality it decides the weights to be assigned to each Tactical Module at disposition.

The decision is made based on the recent history and on the configuration adopted.

Since winning is not necessarily the objective of the Adapta, but rather adapting its gameplay to the player, the objective and the type of adaptation has been parametrized to be a more fluid configuration, instead of a precise aim as win, tie or lose.

The *aim* of the Strategic Module is a value which ranges in [-1, 1], and represents the desired outcome of the AI for a match. If the aim is set to 0, then it tries to tie with the opponent, if set to 1 it will try to win, and -1 to lose. It is however possible to choose any value in-between the interval, so for instance 0.2 means that the aim is to slightly win.

Before each match, the Strategic Module operates in this way:

Algorithm Strategic Module weight assignment

```
1:  modulesValues[]← historic_values(modules) //evaluation of modules based on history
2:  Foreach moduleValue in moduleValues
3:    moduleValue ← gaussian(moduleValue, aim, sqrVariance) //true value
4:    moduleValue ← random_variate(moduleValue) //random variation
5:  end foreach
6:  normalize(moduleValues) //now sum of module Values is 1
7:  applyConfiguration(moduleValues, modules)
```

Algorithm 1 - Strategic Module weight assignment

First every module is evaluated given the past history. This is done with the following equation (for each module m):

$$historic\_value(m) = \frac{\sum_{h=0}^{H} w_{m,h} * r_h * p^h}{\sum_{h=0}^{H} p^h}$$

Where:

- $h \in [0, H]$ is the index representing the history of the previous h matches (the previous one corresponds to 0, and H is the history capacity assigned to the Strategic Module);
- $w_{m,h}$ is the weight assigned to module m at previous match h (range [0,1]). This allow the outcome of a previous match to be weighted according to the importance of a module in that match;
- r_h is the outcome of the match of previous match h, which is in range [-1,1], where -1 is total loss and 1 is total win, while the in-between values represent estimates of how close to one side or the other the match was (for pre deployed it was used the predeployed evaluation function as explained in 7.2.2);
- p^h is the past multiplier, which is a value ranging in [0,1]. This just gives more or less importance to older outcomes. If 1 all recorded outcomes have same importance, if 0 only the previous match matters

$historic\_value(m)$ returns a value in the range [-1,1].

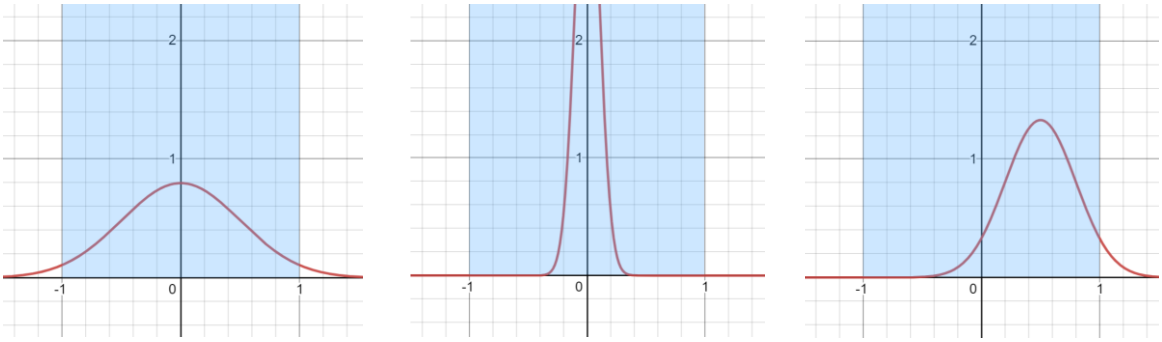
This value is used to estimate the overall value in the previous matches of a module. The closer to 1 the more a module was responsible for winning, the closer to -1 to losing and so on.

The Strategic Module uses this information to generate new weights according to a gaussian function in the range [-1,1]:

$$gaussian(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Since we are interested in the range [-1, 1] only, we can choose μ and σ^2 accordingly:

- μ is in fact the *aim* of the Strategic Module: the weight of a module closer to the aim is always higher;
- σ^2 is the sqrVariance used in the previous algorithm (line 3), and is a configurable parameter (as described later).



(a) $\mu = 0, \sigma^2 = 0.5$

Tries to tie, smooth weight assignment

(b) $\mu = 0, \sigma^2 = 0.1$

Tries to tie, closer to binary assignment

(c) $\mu = 0.5, \sigma^2 = 0.3$

Tries to moderately win, slightly smooth assignment

Figure 4-5 Examples of gaussians for the Strategic Module and their effects

Above 3 examples of configuration for the Strategic Module (in blue is the interested region). The aim (μ) chooses which modules to weight more in the next match based on expected performance: if the aim is 0, like in (a) and (b), modules which generated an $historical\_value(m)$ closer to 0 will have in the next match a higher weight. If the aim is positive, like in (c), modules which performed toward a good margin of victory will be weighted more. If a module scores a $historical\_value(m)$ toward 1 it would be penalized and valued exactly as one which performed overall 0, because the aim is to win by a certain margin. The same reasoning applies toward negative values.

The square variance σ^2 decides the distribution of the weights around the aim. The higher, the less significative the aim becomes. A good starting point are values greater or equal than 0.3, because tend to distribute better weights also for values "far" from the aim. The opposite tends to cause a single pick among the modules each time, because weights almost only values close to the aim, while giving values near to 0 to other modules. This can have, in some scenarios, the effect of choosing almost randomly, if all modules obtain an historical value far from the aim (since after this step is applied a random variation, as explained now).

After having evaluated the true weight that should be assigned to each module, this value is variated randomly to a parametrized degree, but in percentage of the weight itself. The higher it is the more noise the less impactful the choices will be. Since the variation is in percentage of the true value found at the previous step,

the more similar the true values are the higher is the randomicity of the next configuration. More formally the value obtained with the gaussian is modified by:

$$random\_variate\ (x) = \max\ (0.001, x * \big(1 + random(-v, v)\big))$$

Where v is the parameter chosen and $random(-v, v)$ returns a uniformly random value between $-v$ and v (v is greater or equal to 0). The max ensures a minimum positive value greater than 0, so that all modules have always a minimum weight assigned, or there could be the risk of having all modules set to 0 in some particular cases (this could happen anyway only if v is set at least to 1, which is already a pretty high value for randomly modifying the results, in general).

Having a random modifier is useful for two reasons: the first is to introduce a bit of unpredictability to the Strategy adopted. But most importantly it avoids stalls in evolution of the strategy. If for instance two modules start with the same value, the Strategic Module cannot attribute wins and losses to one or the other. This would result in having the 2 (or more possibly) modules assigned at each match with the same exact weight, reducing de facto the strategies that can be adopted. Any small value of v avoids this behavior and keeps randomicity low, possibly not impactful at all.

Finally the weights (moduleValues in the previous algorithm) are normalized:

$$w_{normalized} = \frac{w}{W}$$

where

$$W = \sum_{m}^{M} w_m$$

Since all w_m (weights, or values, of module m) are positive, the result is positive, and the sum of the weights will be 1. This means that also the final bids seen by the Adapta AI will be always in the range [0,1].

If the Strategic Module starts with no history it just generates a random distribution. It can be also set to generate a random distribution with a given chance at each match.

To summarize, the parameters that can be chosen for configuring a Strategic Module are:

- History Capacity H: the total number of matches to record, after which the oldest is dropped at every new record;
- Past Multiplier p: value between [0,1] which tells how much weight to give to older results;
- Aim μ: represents what the Strategic Module should try to achieve;
- Squared Variance σ^2: determines the distribution of weights around the aim. Can make the Strategic Module more favorable toward more homogeneous weights assignments or toward more heterogenous ones;
- Random Variation Percentage v: modifier in percentage of values obtained with the deterministic evaluation made based on the previous parameters;
- Full Randomize Probability p_{fr}: chance that the Strategic Module will just randomly select a distribution of weights over evaluating them based on past histories.

5 Multi Influence Network Module

In this chapter we describe the Multi Influence Network Module, which is an implementation of the Extermination module proposed in the original ADAPTA paper [3] adapted to work in the full version of the game, its features and its structure. We also present how this module is evolved through genetic algorithms.

5.1. General structure

5.1.1. Description

The Multi Influence Network Module, or **MIN** for short, is the first Adapta Module that was implemented, and reflects in logic the one proposed in Spronck's paper [3], which was used as a simple extermination module, extending some concepts to adapt it to a more realistic case.

This means that by design the only actions it will take are **Move** and **Attack**, although as it can be seen later, it's easy to add a behavior to include a Capture action.

The core of the MIN module is the use of **influence maps** [32]. An influence map assigns a value to each tile of the game's grid, which typically represents the goodness of a certain tile.

Since the goodness of a tile is not unique overall, but depends on several factors, multiple networks are generated with different logics and grouped into several output maps.

To balance complexity and accuracy, it was made the choice to create an output map for each different type of unit (same type means all units with the same name), since they all have in general different roles, albeit some of them being similar, and can threaten and be threatened by different units or terrains.

The MIN module is itself very customizable. For each type of unit, it computes the different local influence maps associated to them, then sum them together, each one weighted with respect to a weight that is determined by training, and summed also to a number of global maps (possibly none to any number). Global

maps are the same as the local ones, except they are only evaluated once per turn (or only once for the entire match, depending on the map type). Each global map is still weighted locally depending on the unit, so different units may weigh differently global influence from different sources.

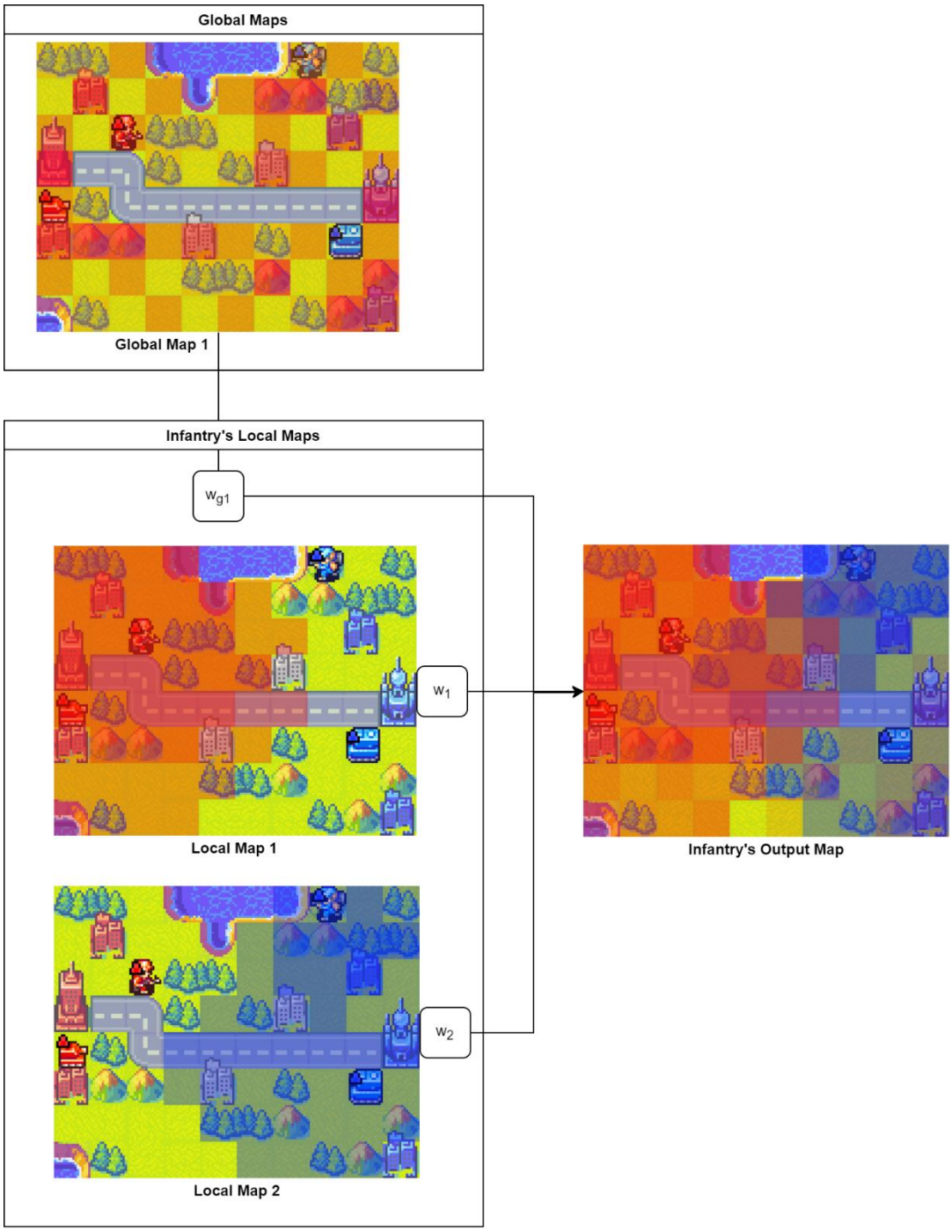


Figure 5-1 MIN output map evaluation

Assuming that each influence map has the same width and height (which is the width and height of the game map in tiles), the final influence value in tile (x, y) is calculated as follows:

$$I(x,y) = \sum_{l}^{L} i_{l(x,y)} * w_l + \sum_{g}^{G} i_{g(x,y)} * w_g$$

The final influence map is just a sum of all local (l) and global (g) maps. The difference between the two is that $i_{g(x,y)}$ (which is influence evaluated in tile (x, y) by global map g) is evaluated only once (per turn or per match depending on the map), while $i_{l(x,y)}$ is evaluated once per type of unit.

In any case the weights are still local to the type of unit and map, so given a specific global map g, $w_{g,infantry}$ can be different from $w_{g,light\_tank}$, and is a learned weight.

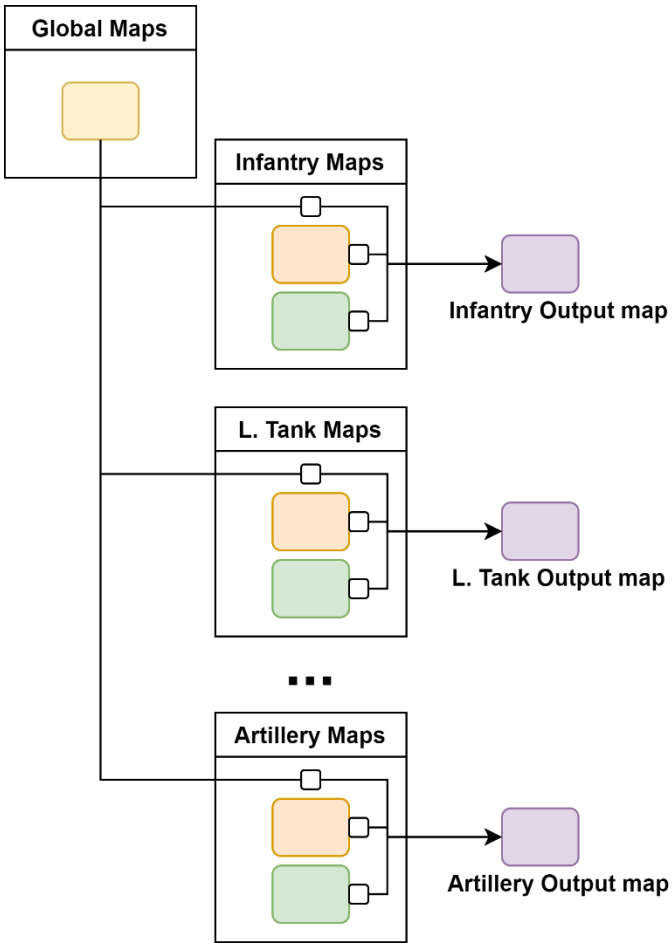


Figure 5-2 MIN module all output maps example view

In Figure 5-1 is shown an example of output map evaluation for the Infantry type of unit. In this specific example, the Infantry type has two local maps and uses a global map. Of the two local maps, one evaluates a influence for all the allies and the other evaluates one for each enemy. The global map evaluates an influence for each tile on the map (so based on the type of terrain). Note that the weight of the global map used for the output map evaluation is still local to the Infantry, since the output is valid for all and only Infantry units. The output map is simply a weighted sum on each tile. Also, each influence map can have in general positive or negative values, and the fact that in the figure enemies propagate a negative value is just to be used as an example, it is not true in general.

The evaluated output map is valid for all Infantry units, which means that the maps contained in the "Infantry's Local Maps" box exist replicated once for each unit. A graphical representation of this is in Figure 5-2: The global map is unique but is locally multiplied by each unit type, while local maps exist locally to each unit type and are evaluated each turn. The weights of the maps and used by the maps to evaluate their influences are learned.

Local maps are updated at each move to account for moved units. The update reevaluates the influence of all and only the elements that changed since last computation when the control of a unit is requested, for clear performance reasons but this is done to increase accuracy of influence for all units moved at any point in the turn. At the start of the turn every unit's output map is computed, because the module must be able to give a bid as indication for each unit. After that, each time a unit is requested to move, the map of its type is updated w.r.t. all the elements that changed, without recomputing it from start.

A unit requested to be moved by the Adapta AI, after having possibly updated its map, will be moved in the highest influence-value reachable tile, and will attack a unit if possible, since the MIN module is implemented as an Extermination Module, as in [3]. In case of multiple attackable units, the unit will attack the best one w.r.t. net fund damage done, which is:

$$netFundDamage(a, d) =$$

$$damageFromTo(a, d) * unitValue(d) -$$

$$virtualDamageFromTo\big(d, a, HP(d) - damageFromTo(a, d)\big) * unitValue(a)$$

Where:

- a is the attacking unit, which is the unit controlled by the AI;

- d is the defending unit, which is each attackable unit from the chosen tile;
- $damageFromTo(u_1, u_2)$ is a function which returns the damage dealt by unit u_1 to unit u_2 according to the damage chart (see Figure 3-11, chapter 3.3.4) and units' HP; the damage is in HP, so a value in range [0, 10]; the function uses only the damage chart and doesn't include luck because luck is evaluated only when the attack is performed but this doesn't change the overall net fund damage expected;
- $virtualDamageFromTo(u_1, u_2, virtualHP)$ returns the same value as the function $damageFromTo(u_1, u_2)$, but the damage is evaluated simulating u_1 with $virtualHP$ HP;
- $HP(u)$ returns the HP of unit u;
- $unitValue(u)$ returns the value in War Funds of a unit, which is unique and fixed, and is known when the unit is bought from its relative Base.

So once the influence map has chosen the point, in a behavioral manner the MIN module will attack the best target in general.

When controlling indirect units, the tile where the unit stands is multiplied by a parameter to increase its influence if there are available targets, so that unit is favored to stand still to attack.

5.1.2. Configuration

The MIN module can be configured to have:

- **A list of supported units**: this is the list which contains all units this module **can control**. If there is in game a unit which does not belong to this list, it won't generate any influence map, hence always generating a bid of 0 for that unit.
- **A list of all expected units**: since having more units in game can increase complexity, it is possible to configure a list of what this module can see as enemy or ally. If a unit does not belong to this category, it will never influence any map of any unit in any way, nor if it is an ally or if it is an enemy. In a general case, this list should be configured to include every unit the game has, so that even if this module can control a subset of units, it can be influenced by all types of units. However, since including more units increases complexity, it is left as a possibility to include only a subset as expected opponents (see experiments section, 7.3).
- **A list of local maps**: a list which specifies how many and which types of maps each unit will have to compute. For each map in the list, in game will exist N copies of these maps, where N is the number of supported units (the first list), each with their own weights (see 5.2).

- **A list of global maps**: a similar list to the previous, except it specifies global maps, which are bound to the whole MIN and not to a single type of unit. (see 5.2)
- **Additional parameters**: to compensate for complexity there were also included a minor number of parameters, which can regulate the propagation function in the map evaluation (see 5.2) and some minor empirical weights.

5.2. Maps evaluation

5.2.1. Evaluation

An influence map is at the essence a function which maps the game map into a 2D array of the same size, in which in each tile assigns a value, depending on, in general, any subset of game elements which define the unique current game state. See chapter 3 for what defines a unique game state exactly. In general, for each tile is the unit, all its values and the terrain with all its properties. There also can be more if money is accounted for, but here is not since the match is predeployed.

In the same general formula as in Bergsma and Spronck's paper [3], the evaluation on each tile is the following function:

$$I(x,y) = \sum_{o}^{O} p(w(o), \delta(o,x,y))$$

Where:

1. O is the set of all objects (essential game elements defining a unique state) accounted for while evaluating the influence in a map;
2. $p(W,d)$ is a propagation function of a weight vector W and a distance d;
3. $w(o)$ is a function which converts the object into a vector of weights;
4. $\delta(o,x,y)$, is a distance function which simply evaluates the distance in game tiles of object o from the tile with coordinates (x,y).

Each type of influence map is defined by (2) and (4). The weight-conversion function $w(o)$ depends on the MIN module itself, which has possibly several functions associated to the same object, since for each different type of unit the weight conversion may change.

The distance function (4) in this case is always simply a Manhattan distance.

The propagation function (2) propagates the weights depending on the distance, and in the MIN module, for units this is always done by propagating the associated weight constantly in the range of attack of the unit, then divided by a parametrized factor from each turn it takes to attack a certain tile, with a (parametrized) limit, after which the influence is 0.

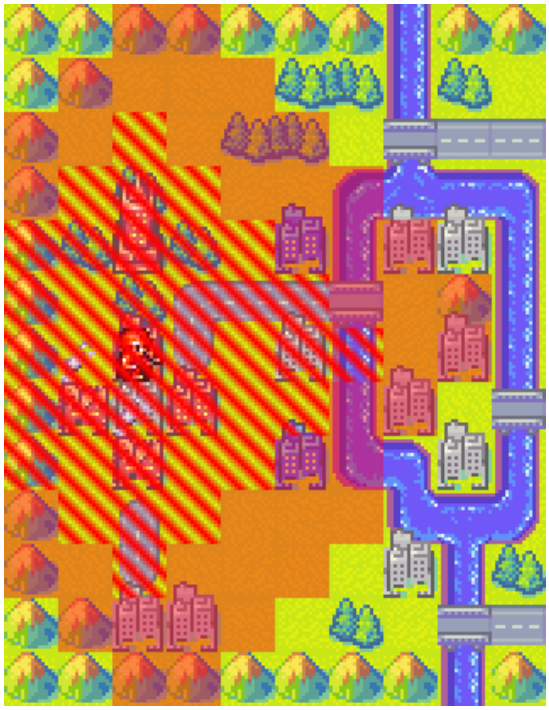
More formally, considering W as a single weight w:

$$p(W, d) = w * zero\_if\_higher(s^{turns(d)-1}, turns(d), t_{max})$$

- w is the weight to be propagated;
- $zero\_if\_higher(a, b, b_{max})$ is a function which returns a if b is not higher than b_{max}, after which returns 0;
- s is the step multiplier, which multiplies the weight w by a value (in the range (0,1] since influence decreases with distance). This is a parameter of the MIN;
- $turns(d)$ is a function returning number of turns t required to attack a certain tile at distance[4] d, where the minimum is considered 1 if the unit can attack a tile in that same turn;
- t_{max} is the maximum number of turns in which a unit can attack a tile before that tile is not considered influenced anymore by that unit. This is also a MIN parameter, and in practice is divided into $t_{direct_{max}}$ and $t_{indirect_{max}}$, where the first is for direct units and the second for indirect, since they behave differently (see 3.3.4).

[4] Manhattan distance

(a) tiles reachable in 1 turn (striped) and in 2 turns (filled)



(b) tiles attackable in 1 turn (striped) and in 2 turns (filled)

Figure 5-3 movable and attackable range of a unit

On the left (a) the range of tiles in which the unit can move in 1 turn (blue-striped cells) and in 2 turns (blue transparent cells).

On the right (b) the tiles the unit can attack in 1 turn (red-striped cells) and in 2 turns (red transparent cells). This latter is the one used by the MIN module, since it is focused on extermination only and attacking and receiving damage is considered influencing a tile, while strictly being able to reach a tile is not.

Using Figure 5-3 movable and attackable range of a unit Figure 5-3 (picture b) as an example, if t_{max} is 2 and s is 0.5, in the red-striped tiles, which are the tiles attackable by the Infantry unit in 1 turn, will have a value of w, while the red transparent ones, attackable in 2 turns, will have a value of $0.5 * w$. If t_{max} would have been 3, there would have been an additional set of tiles with weight $0.5^2 * w = 0.25 * w$. All other tiles have a value of 0.

Each map sums this evaluation over all objects o in O. In this case all evaluations are performed either only on units or only on buildings, since to account for both it's just sufficient to sum the 2 different maps.

5.2.2. Types of maps

This formula is used for all the types of custom maps, which are:

- **Custom**: has a weight for each type of unit. Propagates the weight positively for allied units, and negatively for enemy units. The weight itself can be negative.
- **Custom Allies**: propagates the weight but accounts for only allied units.
- **Custom Enemies**: same as previous but only for enemy units

Hence having both a Custom Allies and a Custom Enemies map gives higher freedom.r.t. a single Custom map, but increases complexity.

There are also standard maps, which can be added to the list of maps and will always be evaluated in the same way, although their weight will be learned. The standard maps are:

- **Standard Attack**: adds for each enemy unit its influence of attack against the type of unit chosen. The influence added is negative, since in general the more dangerous an enemy unit (for the given type of unit) the less the ally unit would want to move there. (As always the weight of the map itself can still be negative, so it can be learned to instead move forward dangerous zones for a type of unit, which could make sense for their archetype or strategies developed by the MIN module).
- **Standard Damage**: adds for each enemy unit an influence decreasing with step-distance (number of turns, as explained in picture 4.2.b), where the influence is evaluated based on how much damage the selected type of unit does to the enemy unit (again multiplied by the map's weight).

The only standard map with the property of be computed only once implemented is:

Once Map Defense: this map simply adds at the start of the match an influence based on the defense value (number of stars, see rules, section 3.3.3). It also multiplies the influence for friendly buildings and friendly factories by two parameters set in the MIN configuration: this is to in general avoid units moving on friendly factories but favor moving them on friendly buildings. The influence per star too is a parameter of the MIN module and thus not learned, but as the rest

of the maps the weight is local to the type of unit: for instance an aerial unit will probably learn to adapt this weight toward 0 since air units don't benefit from cover. In Advance Wars 2/DS do exist some uncommon tiles which can change their defense value during the game, namely breakable pipes and both small and big black cannons, which can be an obstacle but breakable, after which they become a terrain. However, by choice training was not done with such elements, since only breakable pipe sections can be typically present in a versus map, while cannons are usually relegated to challenges, and are still not that common.

Standard Attack and Standard Damage can be only local maps because they need a precise type of unit to do their evaluation, while Once Map Defense can only be global since its evaluation doesn't depend on the type of unit. While it's true that aerial units gain no benefit from defense, all other units gain the same defense value, and it is more performant to evaluate it only once instead of once per type of unit. As stated before aerial units can still learn to set it to 0 by themselves anyway.

The three custom maps can be both global and local.

5.3. Evolution

5.3.1. Encoding

A MIN module is fully described in their configuration, as specified in 5.1.2. Once the configuration is fixed, the weights assignment can be represented as a simple vector of weights, of a precise length which is then used by the MIN module to assign each index to a precise weight.

The specific weight assignment is stored with this format, where capital letters represent the total in a module, and lowercase letters followed by an index represent a specific index of the same category (e.g., if N is the total number of units, n_0 is the unit with index 0. The maximum index would be then $N-1$).

- N: number of units used by this module
- M: number of total units supported by this module (the list of expected units)
- S: number of local standard influence maps
- K: number of local custom influence maps
- G: number of global influence maps
- GK: number of global custom influence maps

$$[n_0^{k_0 m_0}, \dots, n_0^{k_0 m_{M-1}}, n_0^{k_1 m_0}, \dots, n_0^{k_{K-1} m_{M-1}}, n_1^{k_0 m_0}, \dots, n_{N-1}^{k_{K-1} m_{M-1}},$$
$$gk_0^{m_0}, \dots, gk_0^{m_{M-1}}, gk_1^{m_0}, \dots, gk_{GK-1}^{m_{M-1}},$$
$$n_0^{s_0}, \dots, n_0^{s_{S-1}}, \dots, n_0^{k_0}, \dots, n_0^{k_{K-1}}, \dots, n_0^{g_0}, \dots, n_0^{g_{G-1}}, n_1^{s_0}, \dots, n_{N-1}^{g_{G-1}}]$$

In the first position there is $n_0^{k_0 m_0}$, which is for unit type 0 the weight of custom map 0 relative to unit 0, up until all unit types ($n_0^{k_0 m_{M-1}}$). Then the same for all other custom maps, and this once for all the N units this module can control.

Then there are the weights of all M units for each global custom map ($gk_0, \dots, gk_{GK-1}$).

Finally, for each type of unit ($n_0, \dots, n_{N-1}$), their weight of all (in order) standard maps, custom maps and global maps.

Every value is always clamped between 2 parameterized values, so invalid codifications can't exist.

5.3.2. Evolution of Weight Vectors

Weight vectors are evolved in a classical genetic algorithm environment. They are organized in a population of fixed, parametrized size.

In a classical manner, evolution is performed by:

Algorithm 1 Simple genetic evolution

```
1:    population ← random_population()
2:    do
3:      foreach WeightVector wv in population do
4:        evaluate_fitness(wv)
5:      end foreach

6:      if population.bestFitness() > targetFitness
7:        return population.bestWeightVector()
8:      end if

9:      newPopulation ← empty_population()
10:     while newPopulation.size() < population.size()
11:       parent1, parent2 ← population.getRandomParentsProportionallyToFitness()
12:       offspring ← crossover(parent1, parent2)
13:       mutate(offspring, mutate_probability)
14:       newPopulation.add(offspring)
15:     end while
      while true
```

Algorithm 2 - simple genetic evolution

Evaluate each weight vector with a fitness function (see 7.2.2)

With the addition of having an **elitism** mechanism which copies at each new generation the best N (parameter of the MIN) weight vectors into the new population, with the addition note that the copied offspring are not evaluated and automatically obtain the same fitness. This allows to keep always some current best vectors in the evolution process.

It is also added the possibility to have a (parameterized) number of **random** new weight vectors at each new generation, to insert a small probability of having completely different types of vectors which explore the solution space.

As a **selection method** a custom function was adopted, which first creates a custom fitness for each weight vector with a fitness, then applies a simple roulette wheel selection.

The fitness is evaluated as explained in 7.2.2. In short for MIN modules it is a function for every match returning a value in the range [-1, 1], where the value is positive if the match is won and negative if lost, and the precise value is the ratio of the army's strength left w.r.t. the army's strength on first turn: if the match is won it is used the trainee's army, if lost the enemy army. This allow to distinguish better different types of wins and losses. For N matches the final fitness is the sum of fitnesses of each match, which hence ranges in [-N, N].

Since fitnesses can be negative by design, and a simple positivization may result in samples with probabilities too similar with each other and which results in selecting less the best offspring. On the other hand, since it is not desirable to increase too much the probabilities of the current winners to avoid stagnation, the probabilities are boosted only partially.

In the example, which is a real case w.r.t. fitness values, the fitnesses can go from -2 up to +2. If we add 2 to every probability to make it positive and allow a proportional selection, the result is that probabilities are a bit too similar, in general.

To boost them partially, a custom function was adopted to modify original fitnesses:

$$f_{cust} = \left(\frac{(f + f_{range})}{f_{range}}\right)^4$$

$$f'_{cust} = (f + f_{range})^4$$

f_{range} is $f_{max} - f_{min}$, which is the maximum variation range of a fitness. Given that is fixed, f'_{cust} is equivalent in the choice of the two vectors, since vectors are selected in a direct proportional manner, hence dividing them by a constant is equivalent. Both are shown in the table, and the graph of the custom function correctly represents both. Since fitnesses in general could be any, the first function was used (optimized), although in practice it doesn't happen to have overflows since fitnesses in this problem are always values in the magnitude of 10^1.

Table 5-1 Example of values with the custom fitness function

default fitness	+2	f custom	f' custom
-1,2	0,8	0,2401	61,4656
-1,4	0,6	0,17850625	45,6976
-1,1	0,9	0,2762816406	70,7281
-0,9	1,1	0,3607503906	92,3521
-0,1	1,9	0,9036878906	231,3441
-0,75	1,25	0,4358062744	111,5664063
-1,06	0,94	0,2918430506	74,71182096

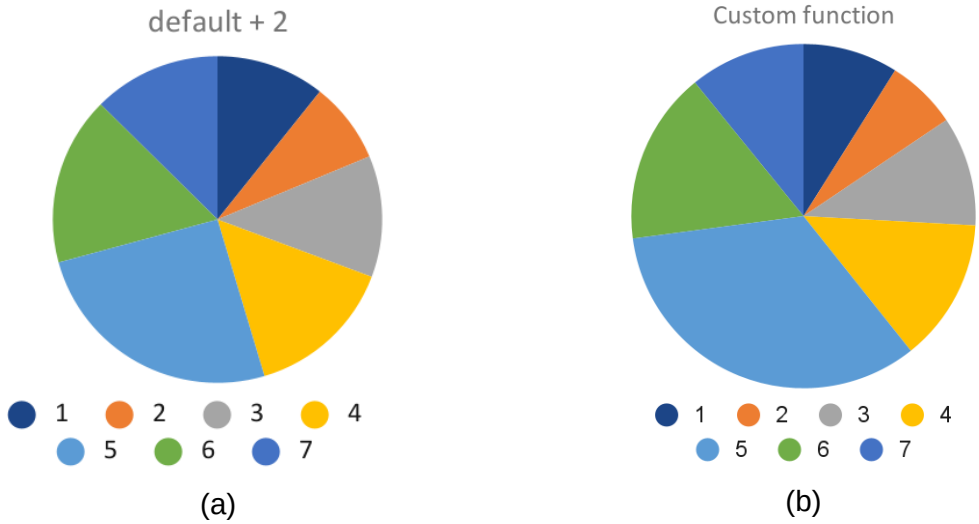


Figure 5-4 graphic view of custom fitness function vs default positive

On the left (a) the fitnesses made positive by subtracting the minimum fitness (so custom fitnesses are always positive or zero).

On the right (b) the fitnesses generated by the custom function, which moderately boosts better offspring.

As **crossover operator** it was made possible to choose one among:

- Split middle: simply splits the weight vector in half, and assigns one half of each parent (in the same parent positions)
- Split random: same but split point is chosen randomly, uniformly along the weight vector
- Multi split random: splits the parents in multiple random points, then alternates between the parents when assigning the split sections.
- Random mix: at each position decide randomly (random but with even odds in this phase) the parent from which take a weight

As **mutation operator**, a simple mutation probability, parameterized, on each weight, has been used. The mutated weight can be any in the range of valid weights and doesn't depend on the previous weight.

5.3.3. Transfer learning

Since a MIN module can be trained to use a specific set of units and to face another set of units, it has been implemented the possibility to transfer the learned weights to another MIN module.

The functionality remaps all the weights of the trained vector to the correct positions in the new vector.

A weight is always relative to a type of map, a type of unit used (one in the set N, in section **Error! Reference source not found.**), and possibly a type of unit faced (set M).

The remapping copies a weight in a new position, if all the criteria are met:

- The **type** and **index of map** is the **same**: the type must be the same, but also if there are multiple maps of the same type (for instance multiple custom or custom allies/enemies maps) the weight is transferred only once and to their relative number of maps. If the former MIN module has 2 custom maps and the new trainee has 1, only the first map will be transferred. Conversely, if the former MIN module has 1 and the new trainee has 2, the weight will be transferred only on the first map
- The **type of unit used** is the **same**: If in the new module there are new units supported, these weights are left untouched. If some are missing, the weights relative to these units will simply be not transferred.
- The **type of unit faced** is the **same** (**if any**): If a weight is also relative to a unit M (the ones in the set of units the module expects to fight against and for which it will generate an influence, even if they are enemies and the module itself can't control them), then the weight is transferred only if is in the new map against the same unit. This criterium doesn't apply if the weight is relative to the map as a whole, for which is weighted only by units of the set N.

The new trainee and the former MIN module can share any number of maps and units, so any amount of knowledge which should be represented in the same form is passed to the new module.

There is also the possibility of fixing the transferred weights, so that they can't be modified during the new training.

It can be useful to train MIN modules in smaller subsets and then transferring the knowledge to modules to be trained in bigger subsets of units, or just to transfer partial information and speed up the process of training.

Since it's possible that by adding different units, maps, or simply changing the training game maps, the weights learned could be less optimal w.r.t. the setting they were learned in, fixing the transferred weights is not always the optimal choice, even if it reduces complexity. It is still possible to keep the weights fixed in a first phase and unlock them later in another training, to start with more stable solutions.

6 CNN Module

In this chapter we introduce the Convolutional Neural Network Module, an experimental approach in the AI applied to TBS games, explaining the rationale and how it was implemented. Subsequently it is described its evolution process, which makes use of the HyperNEAT technique [33] [6].

6.1. General structure

The Convolutional Neural Network Module, for short CNN Module, is an experimental approach in the field of TBS games.

At a high level, it consists of a CNN which is applied in a special type of segmentation problem, where the input is the game map, and the output is the same map in which each tile represents the desirability to perform an action on that tile.

The CNN itself works in the classical way, with a series of layers which convolute the input to a higher-level state, then deconvolute it to the output map.

Just as the rationale behind a general CNN problem, for instance classification or segmentation, the idea is to read the state of the game locally under different types of views, and incrementally build a deeper and unified model which should conceptualize the game state, from which the network should be able to then decode the optimal actions to be performed in this turn.

Below an example of CNN model used for segmentation. Here the input is an image, while in Advance Wars, or in general a TBS, is a game state. Nevertheless, conceptually the working is very similar.

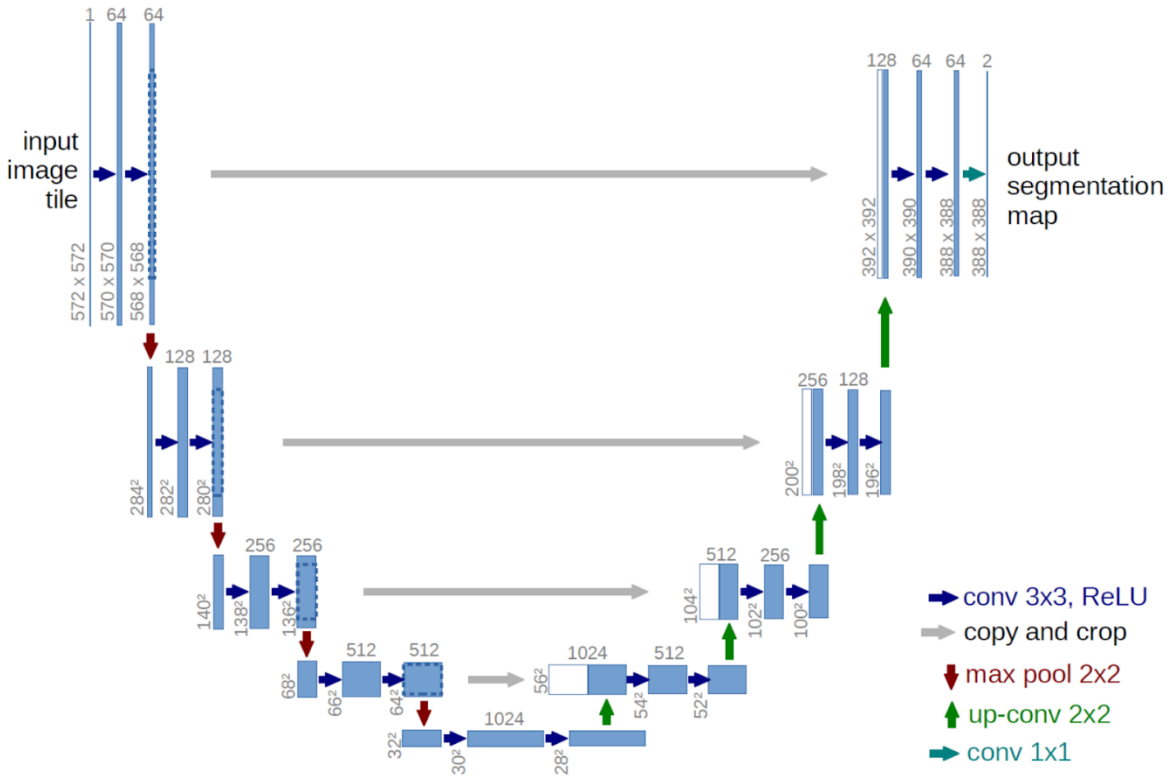


Figure 6-1 example of CNN model for segmentation (source: [34])

Similarly to the MIN module, the CNN module can support a defined number of units. The higher this number, the higher the number of parameters. However, it doesn't have another list of units which includes the units expected to face, since, as explained in 6.3, the parameters are for the most part evaluated not based on weights or parameters (and if they are they all resort to default values).

6.2. Layers

The CNN Module is built with a CNN, which can be configured with any number of layers in order, as long the output is as the same size as the input, in width and height.

Input and output maps' depths are obtained and used as described in 6.3 and 0 respectively.

The CNN itself is simple and classical in literature. The layers that can be added are:

- **Convolutional** layer: traditional convolutional layer composed by kernels of a certain size, can be parametrized in stride, padding, number, and size of kernels;
- **MaxPool2D**: reduces by half the width and length of the map in input by keeping for each 4 tiles the one with the maximum value. Has a parameter to determine on each dimension, in case of odd input, to floor or not the output size;
- **Activation** layer: keeps the input and output map of the same size, modifies the weights by a function specified among ReLu, Sigmoid and Tanh;
- **Transpose Convolutional**: the decoder part of the convolutional layer. Has the same parameters, but works in the "opposite" way w.r.t the convolutional module;
- **MaxUnpool2D**: decoder part of MaxPool2D. Has a similar parameter but with opposite effect, which can increase the size of the output by one on none, one or two dimensions. It works with the logic of Bed of nails, but can be configured to also retrieve the positions in the corresponding MaxPool2D layer in the decoder part.

6.3. Input adaptation

In a TBS game environment, the minimum unit is a tile. Tiles are drastically less than pixels if we were comparing image segmentation with this problem. However, each tile carries more information, and each single tile can affect a lot more the overall problem, while this is not true for pixels. Furthermore, if a tile were to carry all the exhaustive information which characterize it, it would be deeper and at each depth there could be different logics as values (such as some range of values in some layers, one hot encoding in others, etc.).

A tile in Advance Wars is fully defined by its Terrain and Unit. For each there are a set of characteristics which define their state. See rules section 3.3.3 for further explanation of a specific mechanic/feature.

For **Terrains:**

- **Defense value:** defense value of the terrain, in stars, in general ranging from 0 to 4;
- **Movement values:** for **each** type of movement value (feet, mech, treads, tires, helicopters, etc.) the cost of movement on that tile (typical range [0,9]);
- **Building:** which is the building on it, if any;

- **Building's owner:** who controls the building. Can be any player or none: {-1,0,1};
- **Building's capture points:** how much the building has been captured (20 is captured, starts at 0).

For **Units:**

- **Type of unit:** which determines the **damage received** by **and dealt** to other units. As well as other parameters, but in the pure term, "type of unit" doesn't determine strictly what a unit can do, so from this perspective it only determines damage dealt and received;
- **Owner**: the player who controls the unit;
- **Movement type:** the category of movement which determines how much it costs to pass on each tile;
- **Movement points:** how many movement points the unit can spend in 1 turn;
- **HP:** a value between 10.0 and 0.1, which indicates the overall power of the unit, both offensive and defensive. At 0 the unit is destroyed, so it cannot exist a unit with 0 HP;
- **Direct/Indirect unit:** this deeply affects the influence of the unit, since an indirect unit cannot attack if it moves;
- **Min and Max Range:** this is only used for indirect units since direct units only have 1 as min and max range, nonetheless in general is a parameter of a unit, and determines how much area an indirect unit can threaten with their attack;
- **Ammos:** both weapons of a unit have an ammo amount. If it reaches 0, a unit can't attack anymore with that weapon until it gets resupplied;
- **Fuel:** similarly to ammos, fuel determines how much a unit can move. It is a high value in general, and costs 1 for each movement point is spent by moving the unit (see 3.3.3);
- **Actions:** while most of the units can do the same actions (move, wait, attack, join), some can or cannot do some of them, like APCs not being able to fire or infantry and mechs being able to capture buildings. In general, a unit has a set of actions it can perform;
- (Vision): Tiles of visibility in Fog of War. This parameter is not important however since the research didn't focus on FoW mechanics.

Given that every parameter increases the input map dimension, and hence the whole CNN's complexity, a CNN module can be configured to generate the input map accounting only for a subset of parameters. Even so, the configuration itself

was simplified accounting certain aspects, to reduce the overall complexity without losing too much information about the game state.

It has also been included the possibility to define **custom static** features for each unit and for each terrain. This feature is meant to define some heuristic values which summarize certain features, for example the overall effectiveness against a general category of units, like grounded vehicles, which is not a unique feature and could be different depending on which units are selected for training.

If a static feature is defined, it must be defined for each unit supported by the module, and additionally a default value to be used if an unknown unit (w.r.t. the supported units) is faced.

For each feature included for terrains or units in the CNN module, it corresponds a channel (which is an additional layer of depth in the input map) which can take its own range of values and has its meaning.

The list of **terrain features** that can be included in the CNN module is:

- **Defense value**: as described above in the full list of terrain features;
- **Move_***: each type of movement, encoded as "MOVE_*" (e.g., MOVE_FEET, MOVE_TANK, MOVE_AIR…), can be included or not, each type is included at most once. For each, the cost in movement points to walk on the terrain with the specified type of movement;
- **Owner**: the owner of the building, if the terrain has one. It is a channel with values $\{o, n, p\}$, where o is opponent's, n neutral and p is player's. The three values are by default {o = -1, n = 0, p = 1}, but can be defined custom;
- **Static**: this is a custom static feature as described above, for a terrain. This is the only feature which can be included in any number, since each time it has a custom meaning. If defined, it must be defined a list of terrains, and for each of them N values, where N is the number of static features included. It has also to be included a default section with N values for any unknown terrain.

The list of **unit features** that can be included in the CNN module is:

- **Move points**: simply the amount of movement points the unit has;
- **Can capture**: a boolean channel with values {0, 1} if the unit can capture or not;

- **Move_***: same as units, but is a boolean channel. All the move features together can be seen as a one-hot encoding, since a unit has only one type of movement at any time;
- **Ammo 1** and **Ammo 2:** channels including the amount of ammos of weapon 1 and 2 respectively;
- **Static**: same as for terrains, and as explained above, can be included multiple times.

It has not been included a direct way to encode damage dealt and received by certain units, because it would add a lot of depth to the input map, and thus to the CNN. Instead, as reported in the list of experiments section (7.3), static maps were used to group categories of units and simplify the problem.

Additionally, although unit features don't include the Owner and HP info, at runtime their static values, which are used for units as the main information versus other units, are multiplied by a value in the range [-1, 1], which is {-1, 1} for the owner and [0, 1] for the HP ratio, multiplied together. The information about hp and owner is hence implicitly included without adding 2 more layers to the input map.

$$s'_f = s_f * owner(u) * hp\_ratio(u)$$

Where:

- s'_f is the static feature adjusted;
- s_f is the static feature;
- $owner(u)$ returns -1 if the unit u is an opponent's one, $+1$ if it is a player's one;
- $hp\_ratio(u)$ returns the ratio of health of unit u, which is $\frac{hp(u)}{MAX\_HP(u)}$ (where $MAX\_HP(u) = 10$).

The input adapter component of the CNN module hence converts the input map in an input map for the CNN of size $w * h * d$, where w and h are width and height of the game map respectively, and d is the number of terrain and unit features that were included.

A graphical representation of this is shown in Figure 6-2. In the figure the computation of the input is shown for a single tile as an example. The process to compute the full input map is simply repeated for each tile in the game map.

Static maps of units are adjusted with s'_f as aforementioned before being computed.

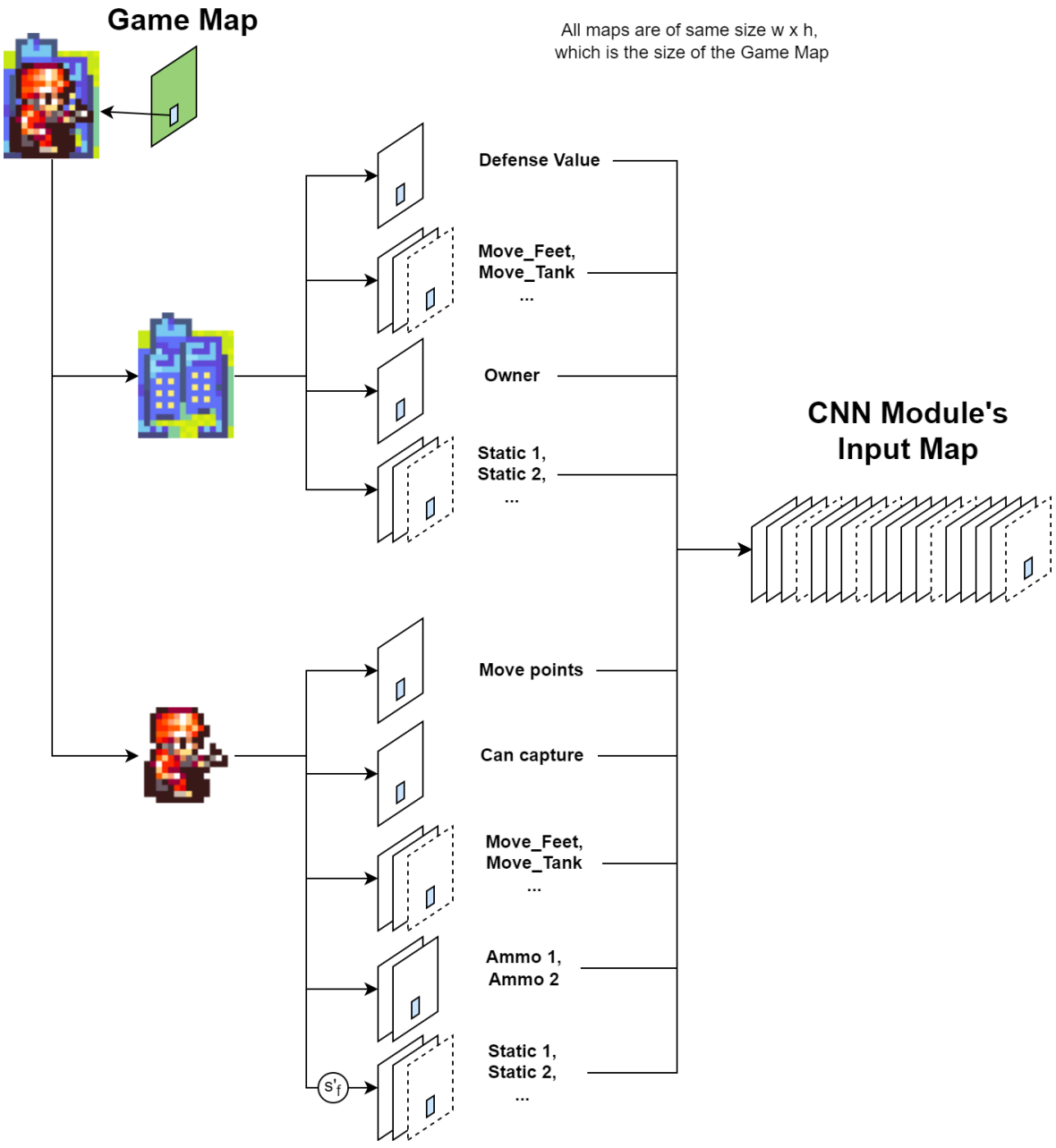


Figure 6-2 Computation of CNN Module's input map

6.4. Output adaptation

In order to balance output complexity without sacrificing too much space of possible actions, each supported unit has its own map of the size of the game map, in which similarly to the MIN module, each tile represents the goodness of moving toward that tile.

However, since the goodness of moving toward a tile is not derived directly from a set of influences, the actions taken by the CNN modules can be more generic. In particular, the CNN module attacks a unit if the cell with the enemy unit is the one with the highest value for a precise type of unit (e.g., tanks), while the MIN moves towards empty cells only and attacks from there the best target.

This implies that for attacking unit is lost a bit of control on movement: if the unit can attack an opponent's unit from more tiles, it will simply move in a behavioral way to the nearest one.

Furthermore, the CNN module can take actions such as capturing buildings. Since it's not a pure extermination module, it could also theoretically perform most of the other actions, such as supply, repair, join and load. This has been not accounted for in practice due to unnecessary increase of complexity, mainly because most of these actions are performed by other units not used in this project.

Given the number of supported units N, the output is hence of size $w * h * N$, where w and h are the game map's dimensions.

The meaning is a segmentation performed on each slice of depth of the output map, where each segmentation represents the goodness for the type of unit x to act toward a tile. The type of action the unit will perform on that tile and the path taken to act on that tile are selected in a behavioral.

6.5. Evolution and NEAT

6.5.1. Evolution with NEAT and CPPN

To evolve the CNN module, it was attempted an experimental approach, generating a CNN's weights using the NEAT technique [35].

The process is similar to the one described in the HyperNEAT approach [33] [6], in particular it was adopted a technique using a Compositional Pattern Producing Network (CPPN) [6] [36].

The idea of CPPN is to use a network which generates patterns in a defined N-dimensional space, through a function generating a weight in every point in such space.

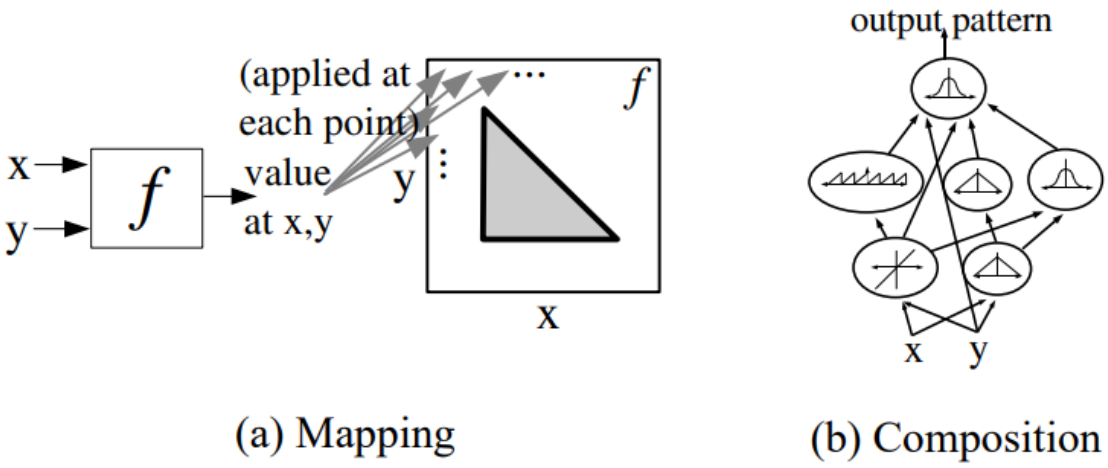
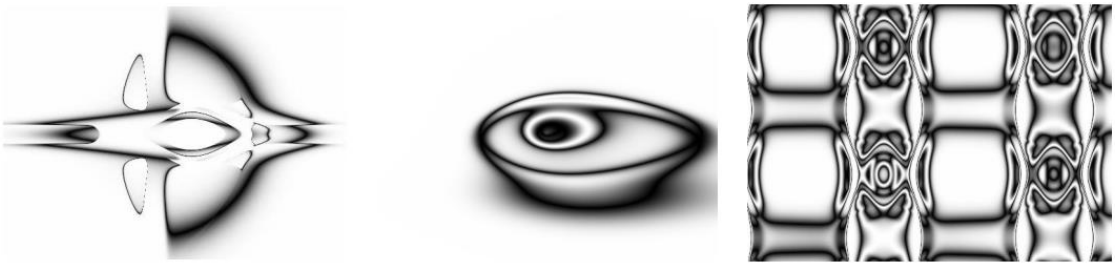


Figure 6-3 CPPN Encoding in a 2D scenario. (a) Function f maps a value to each 2D point (x, y). Applied to every point in a defined space, it results in a spatial pattern, interpreted as phenotype of the genotype f. (b) The CPPN is a graph of any topology, which determines which functions are connected and how through weighted connections. (source: [33])



(a) Symmetry (b) Imperfect Symmetry (c) Repetition with Variation

Figure 6-4 CPPN-generated regularities. Examples of spatial patterns exhibiting (a) bilateral symmetry, (b) imperfect symmetry, and (c) repetition with variation. This demonstrates that CPPNs can encode fundamental regularities of different types (source: [33])

Here in a CNN setting, the concept is the same, but the weights are evaluated by "disposing" the kernels in a 3D space, always in the same manner, and then let NEAT generate functions for a 3D space to create ideally patterns in the weights of the CNN.

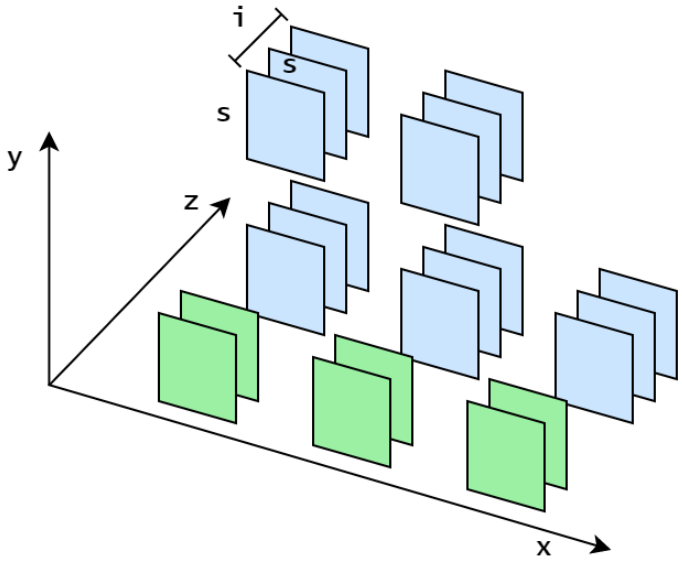


Figure 6-5 CNN Module's visual kernels disposition for CPPN's weight assignment

In Figure 6-5 is shown visually how the kernels are arranged in the 3D space. For each Convolutional 2D layer in the CNN its kernels are disposed on the same starting z, and disposed on N rows and columns, according to configuration parameters. Kernels are separated by a parameter of choice, and so is the gap which separates the kernels of the next layer. In the image the green kernels are ones of the first Conv2D layer of the CNN, the blue ones are of the second Conv2D

layer. Their size is $s * s$ and is a parameter, while their number (i) depends on their relative input (in this simplified example since green kernels are the first ones, the hypothetical input map has depth 2). To set up this formation biases of kernels are set to 0.

6.5.2. Evolution with genetic algorithm

Since using CPPNs to generate weights has proven to be challenging in terms of computation (see chapter 7), it has been implemented also the possibility to train the CNN with classical genetic algorithms.

The algorithm is the same as in 5.3.2 (Algorithm 2), the only difference is that now a Weight Vector encodes a CNN.

Since the CNNs used has only kernels with their weights (and biases) to set as weights, the weight vector is simply encoded as the weights of the kernels in order of appearance in the CNN (from encoder to decoder).

7 Experiments and results

In this chapter we explain how and with which rationale the experiments were performed, evaluating the training environment, the complexity of the game and the evaluation. After this it is provided a list of experiments related to the relevant features developed for this thesis.

7.1. Training description

7.1.1. Training environment

The training process of every Adapta module consists always in a series of matches made in custom maps, differentiated in some aspect to cover a basic variety of situations.

The larger is a map the longer it takes to train, since in general modules have to process each tile, even if there were the same number of units.

The number of units also is a factor but in a "standard", pre-deployed match (which is a match where the whole army is pre-deployed and no units can be built) typically the number of units is proportional to the map's size.

For this reason, the training maps tested can vary in size but they are always on the **small** side of the spectrum of **maps' sizes**, where the spectrum is intended with normal skirmish maps included in the game.

To evaluate Adapta modules, since they have only to decide actions of units, only **pre-deployed** maps were used.

Given that the complexity was reduced, also the **set of units** used for testing was **reduced**. The objective of the training is to assert that the Adapta system can create an interesting AI in a realistic game environment, and by reducing the space of possibilities within the game the training is made easier but the rationale is not changed.

In particular all special units introduced with Advance Wars: Dual Strike, with their more peculiar mechanic, have been excluded. It is always possible to train a model specifically to make them work, however.

The Adapta can be built with any number of modules which don't learn and one that does. This was made to allow for instance training of building modules and training of specific modules which support a smaller number of units (e.g., a MIN trained to use only air units). However, this feature was not used in practical training.

Next follows the list of maps adopted for training. Note that the trainee is not player 1 or 2, but it depends on training. Typically, maps are not symmetrical and the trainee is trained on both sides, for several reasons. One is that in any case a trainee has to play a set of matches since playing only 1 includes a little degree of randomness (see 7.2). The other reason is to make it play different scenarios or asymmetric scenarios on both sides to develop a less overfit solution.

Figure 7-1 Training map 1 - 18x11, 17 units p.p. - 10 types of units

The training map 1 is relatively big w.r.t. training maps, and features several types of units p.p. (per player). As every training map used, it is not symmetrical. It could give a bit of defensive advantage to player 1 (red), but it depends on tactics adopted.

Figure 7-2 Training map 2 - 12x8 - 10 units p.p. - 5 types of units

This map is instead very small and with very few units. It is useful to test trainees in a more contained environment more quickly. This map is almost fully symmetrical, not giving a considerable advantage to neither side.

Figure 7-3 Training map 3 - 10x8 - 11 units p.p. - 7 types of units

This map is even more compact, but features more unit variety. The deployment is symmetrical, but the map again is slightly not, and features a middle of the map with a few choke points for the vehicle units which can make the artillery a precious unit, against all grounded units or vehicles, which overall are the more menacing ones.

Figure 7-4 Training map 4 - 8x10 - 14 units p.p. - 7 types of units

This vertical map features a very dense unit deployment and a mostly free field, which makes it hard for both parties to engage the enemy without sacrificing too many units. The set of units is on purpose the same as map 3, although the amount for each is different.

As opponents, in the project Commander Wars there are also 4 behavioral AIs, one named “Very Easy” and the other “Normal”, which comes in 2 additional variants, “Offensive” and “Defensive”. The difference is in the parameter configuration which leads to different decision-making. Having several types of behavioral AIs at disposal is very useful to allow the trainee to face from the start a good opponent instead of itself (although training vs itself could be good for different reasons).

To avoid stalls, the ruleset includes a turn limit, after which the opponent wins automatically.

As can be seen naval units were avoided completely, and air units only marginally. The reasoning behind this is that naval units are in general a support type of category and require dedicated maps, which in general are larger maps. A naval only map would have not added anything significant, mainly because the 3 AW 2 naval units, excluding the Lander, which is a transport unit, act in a rock-paper-scissor manner, which doesn't add anything meaningful per se w.r.t. a more complex system of grounded units where there boundaries of picks and counter picks are less defined.

7.1.2. Complexity

The number of possible states in Advance Wars is extremely high. In theory it is infinite since a generic map has no strict limits in size, but in practice most of the maps are sized from 8x8 to a maximum of 40x40 (maps can be of any set of dimensions, so not only squared).

To do a comparison against chess and checkers (and shogi, which however has a 9x9 board), we'll perform an estimate on a 8x8 generic map for 2 players, and considering the unit of Advance Wars 2 (so not including the AW:DS ones, which were not included in the experiments anyway) and only the most common tiles.

A note to be mentioned, as said in 2.1 is that albeit having a higher number of configurations, in games like Advance Wars, w.r.t. to classic board games, a change in the configuration is more likely to have lower impact on the outcome of the overall match. A map which has a single different tile is likely to be almost none of impact to the outcome of the game, and so is a unit which is placed differently and/or which has a slightly different health or ammos, or in some cases if is missing entirely. Clearly the smaller the map the more important everything is, however this is true for most slight changes possible given any configuration. In games like chess, moving any unit even of 1 tile in general affects way more considerably the match.

Considering the map alone, the type of tiles that can be found are: Plains, Road, Forest, Mountain, Sea, River, Beach, City*, HQ*. City can be neutral of belonging to a player, hence can be configured in 3 states. They also have their Capture Points, which is a value from 1 to 20 (see 3.4.2). HQ also shares this property of Capture points: in a 1v1 match there exist 1 HQ for each player, and if is captured the game is over, so they are exactly 1 per player and of their property. A map in general can have no HQ, but we include it because a typical map does have it.

Assuming an average number of 6 cities for a very small map, and removing the 2 HQs, $64 - 6 - 2 = 56$ generic tiles remain (so we assume an average of $c = 6$ cities)

So, the total maps configurations, approximately, are:

$$C_{map\_raw} = 7^{62-c} * (3 * 20)^c * \ 20^2 \cong \ 2.1 * 10^{47} * \ 4.6 * 10^{10} * \ 400 \cong 3.8 * 10^{60}$$

Which must be multiplied by the number of disposition of the 3 groups of tiles, which is

$$\frac{64!}{56! * 6! * 2!} \cong 10^{11}$$

For an approximate total of $C_{map\_raw'} = 10^{71}$

This is a raw estimate, in fact this doesn't account the fact that for most configurations the game would be probably unplayable. For a typical map, let's assume a 66% made with Forests, Plains, Roads, Beach, which are the ones crossable by all land units, and a 33% made with Mountains, Sea and Rivers, crossable only by specific units. Assuming again $c = 6$:

$$C_{map\_balanced} = 4^{38} * 3^{18} * (3 * 20)^c * 20^2 \cong 7 * 10^{22} * 3 * 10^8 * 4.6 * 10^{10} * 400 \cong 3.8 * 10^{44}$$

Again, multiplied by the number of permutations of each category of tiles, which is

$$\frac{64!}{38! * 19! * 6! * 2!} \cong 10^{24}$$

For a total of

$$C_{map\_balanced'} \cong 10^{68}$$

This has to be multiplied by the combinations of units and their states. In AW 2 there are 19 units [37], but for simplicity we'll consider again only the most common ones/the ones used in these experiments, which are 10: Infantry, Mech, Recon, Anti Air, Light Tank, Mid Tank, Neotank, Artillery, Missile launcher, Helicopter. This omits completely naval units, as mentioned in the previous subchapter 7.1.1.

We consider that on average on the maps adopted for experiments 1/5 of the tiles is occupied by units (1/10 for each player), we evaluate for 12 units total, which

can be disposed on 12 tiles for each army for simplicity. Also we assume the army is the same, so the number of possible armies with 6 units and this list of units available is simply

$$C_{army6} = 10^6$$

While the number of dispositions of a generic 12 units on 46 tiles (46 is 64 – 18 tiles assumed not crossable by anybody as mentioned above; some units can go over other tiles so the number should be higher but this is a lower estimate), is:

$$\frac{46!}{12! * 34!} \cong 3.8 * 10^{10}$$

Assuming an army of 2+2+1+1 units (each number is a different type of unit), the possible armies combinations number is:

$$\frac{12!}{(2!)^4 * (1!)^4} \cong 3 * 10^7$$

For a total of

$$c_{army\_configurations} = 10^{23}$$

Which has to be multiplied by the number of states possible for each unit, which should account:

- HP: 10^1 (10^2 in theory but the difference between single percent is almost always negligible)
- Fuel: 10^2
- Ammo 1: 10^1 (on average a unit has only 1 expendable ammo and the total is 9)

Hence the total is

$$c_{army\_configurations'} = 10^{27}$$

Which brings a final total of estimated plausible game states, for a map 8x8 of

$$c_{total\_configurations} = 10^{95}$$

The estimation is lower than the actual number of possible game states since is based on a lot of simplifying assumptions. It is however clear how the complexity explodes when the size of the map increments, which is the typical case. Even in the trainings the smallest maps are 10x8, while in typical gameplay such maps are a small subset w.r.t. to the totality of the available maps.

Table 7-1 number of configurations by game

Game	Possible configurations	Board size	Sources
Checkers	$5 * 10^{20}$	*8x8*	[5] [38]
Chess	$8.7 * 10^{45} (upper\ bound)$	*8x8*	[38] [39] [40]
Shogi	$\cong 10^{71}$	*9x9*	[38] [41] [42]
Go	$\cong 2.089 * 10^{170}$	*19x19*	[18]
Advance Wars*	$\cong 10^{95} (raw\ estimate)$	*8x8*	(this chapter)

*as estimated in a general scenario in a map of comparable size with the other games.

7.2. Evaluation

7.2.1. General method and randomness

Given the structure and the objective Adapta modules, evaluation is always performed by making the trainee play a set of N custom matches, then evaluating them based on performance metrics.

The objective of this phase is to assign a useful fitness to a trainee so that the evolution process can work properly.

To do this it must be noted that any match is always subject to a small **degree of randomness**.

The randomness can be derived mainly from 2 factors:

- **Luck**: as explained in the game rules section, luck (see 3.3.4, Attack section) always affects any attack made by units. Although luck is a minor factor in determining the final damage and its fluctuation is 10% at most, which in practicality doesn't affect much the strategy adopted in a game, it can in some cases result in outcomes which alter the trainee's or the opponent's decisions, which could lead to different final outcomes.
- **AI behavior**: the AIs are subjected to a degree of randomness in the way they behave. Although the general play is the same, turn-order can make an impact on the outcome of certain situations. This is especially true when unit density on the front line is high. From one side this kind of randomness is good because it avoids having an opponent which always plays in the same way, resulting in a less overfit model. From the other side it adds noise to the fitness evaluation.

Considering the randomness present intrinsically (and not) in the game, the choice made to assign a more robust fitness was to create a fitness based on the outcome of multiple matches. At the cost of increasing training time, this also increases selective pressure.

7.2.2. Evaluation types

All the experiments were done in **pre-deployed** maps (7.1), hence the fitness was evaluated in different ways but always with the **army strength ratio** (a_r) as base measure.

$$a_r = \frac{A_{last\_turn}}{A_{first\_turn}}$$

Where A_t is the **value** of the army at a certain turn t. The value of an army is defined simply as:

$$A_t = \sum_{u}^{U} val(u) * hp\_ratio(u)$$

Where:

- U is the set of units u composing the army of a player.

- $val(u)$ returns the value of unit u in terms of *war funds* (see 3.4.3), which can be considered the most abstract "power" or "value" measure of a type of unit.
- $hp\_ratio(u)$ is the same as seen in 6.3: it returns the ratio of health of unit u, which is $\frac{hp(u)}{MAX\_HP(u)}$ (where $MAX\_HP(u) = 10$)

Hence for each unit its value is its cost multiplied by a number in the range (0,1] according to its hp percentage left.

At the start of the game every unit has full hp, hence A_{first_turn} is just the sum of the costs of every unit a player has.

Since the maps are pre-deployed, $A_{last_{turn}}$ is always equal or less than A_{first_turn}. Units can be repaired in buildings, but they can never exceed in value their original value by definition of army strength At.

The list of evaluation types, for pre-deployed maps is:

- **Win count only -** *{-1, 1}, N matches {-N, +N}*: it just counts +1 if won and -1 if lost. Being too simplistic, it has not been used for meaningful experiments. For N matches it sums the total for each match.

- **Army value pre-deployed -** *[-1, 1], N matches [-N, +N]*: returns a_r^{ally} in case of victory and $-a_r^{opponent}$ in case of loss. If the match is lost the more the opponent's army is strong the worse is the fitness. Conversely, if the match is won the more the trainee's army is strong the better is the fitness. For N matches the total is just the sum of the fitnesses.

- **Army value pre-deployed positive** *[0, 2], N matches [0, 2N]*: a simple variant of the previous one which gives an additional +1 to avoid negative fitnesses. For N matches the total is the sum of all the fitnesses.
- **Army value pre-deployed positive win bonus** *[0, 2], N matches: [0, 2+N]*: a variant of the pre-deployed positive, but when evaluating the total fitness the partial fitnesses are averaged and *then* is added a +1 bonus for each victory in the set of N matches.

7.3. List of experiments

Here follows a list of meaningful performed experiments and their results and a discussion [To be noted: that names of this modules are not the same as the ones in the actual project, and here they are named in order for simplicity].

7.3.1. MIN Module 1

- Maps: Training 2
- Matches: 4, two for each side of map Training 2
- Opponent: Normal AI
- Evaluation: Player value pre-deployed ([-4, 4])
- Best fitness: 3.109268 (gen < 50)
- Generations: 177
- Evolution: Genetic Algorithm

MIN module configuration:

- Unit List: Infantry, Mech, Artillery, Light Tank, Anti Air
- Full (Expected) Unit List: Infantry, Mech, Artillery, Light Tank, Anti Air
- Unit Influence Maps: Custom Allies, Custom Enemies, Standard Damage, Standard Attack
- Global Influence Maps: Once Map Defense
- Parameters:
    - Step multiplier: 0.5
    - Steps for indirect units: 1
    - Steps for direct units: 2
    - Weight per star: 2
    - Friendly Building Multiplier: 1.5
    - Friendly Factory Multiplier: 0.1

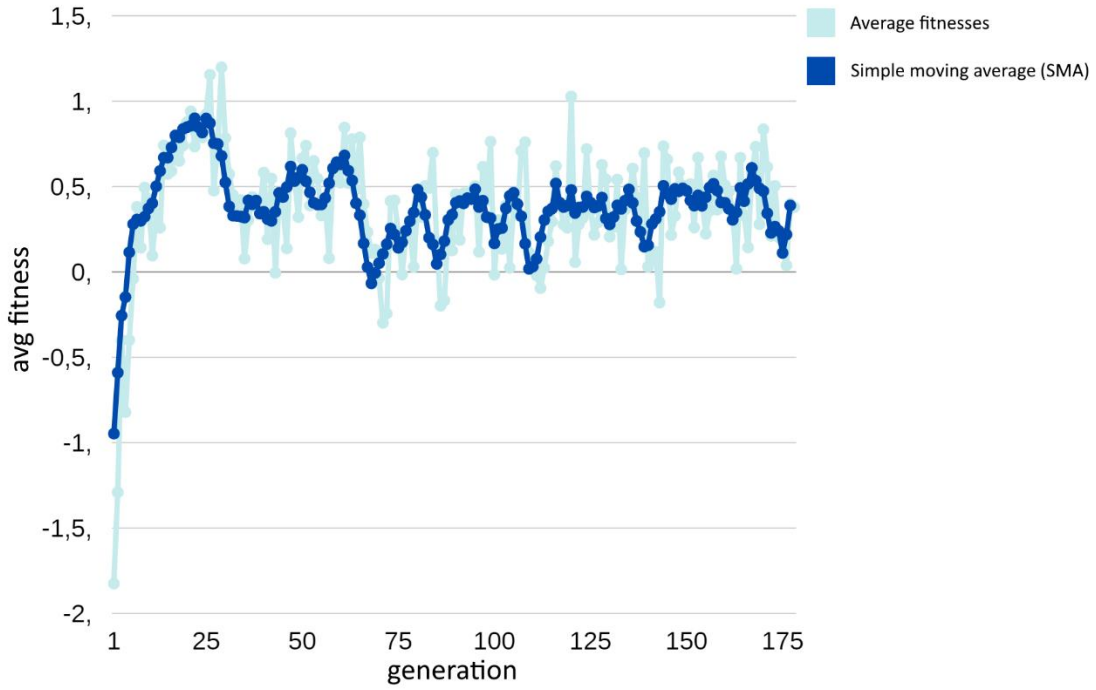


Figure 7-5 Average fitnesses of MIN Module 1 by generation

Results: the average fitness of each generation showed a fast increase trend followed by relatively noisy values around an almost constant value around ~0.4. The best result was in fact obtained early, however comparable results (~3.0795 and several ~2.9 on generations after 159) were obtained in later generations.

This module was trained in a simple environment (Training map 2) with few units, so that the MIN Module 3 could inherit the knowledge for its training on a bigger map with more types of units.

In a very contained environment the MIN module proved to be capable of being very effective with a solid margin of win, which is a good indicator of its potential.

7.3.2. MIN Module 2

- Maps: Training 3
- Matches: 4, two for each side of map Training 3
- Opponent: Normal AI
- Evaluation: Player value pre-deployed ([-4, 4])
- Best fitness: 1.577321 (gen 130)
- Generations: 153
- Evolution: Genetic Algorithm

MIN module configuration:

- Unit List: Infantry, Mech, Artillery, Light Tank, Heavy Tank, Anti Air, Helicopter
- Full (Expected) Unit List: Infantry, Mech, Artillery, Light Tank, Heavy Tank, Anti Air, Helicopter
- Unit Influence Maps: Custom Allies, Custom Enemies, Standard Damage, Standard Attack
- Global Influence Maps: Once Map Defense
- Parameters:
  - Step multiplier: 0.5
  - Steps for indirect units: 1
  - Steps for direct units: 2
  - Weight per star: 2
  - Friendly Building Multiplier: 1.5
  - Friendly Factory Multiplier: 0.1

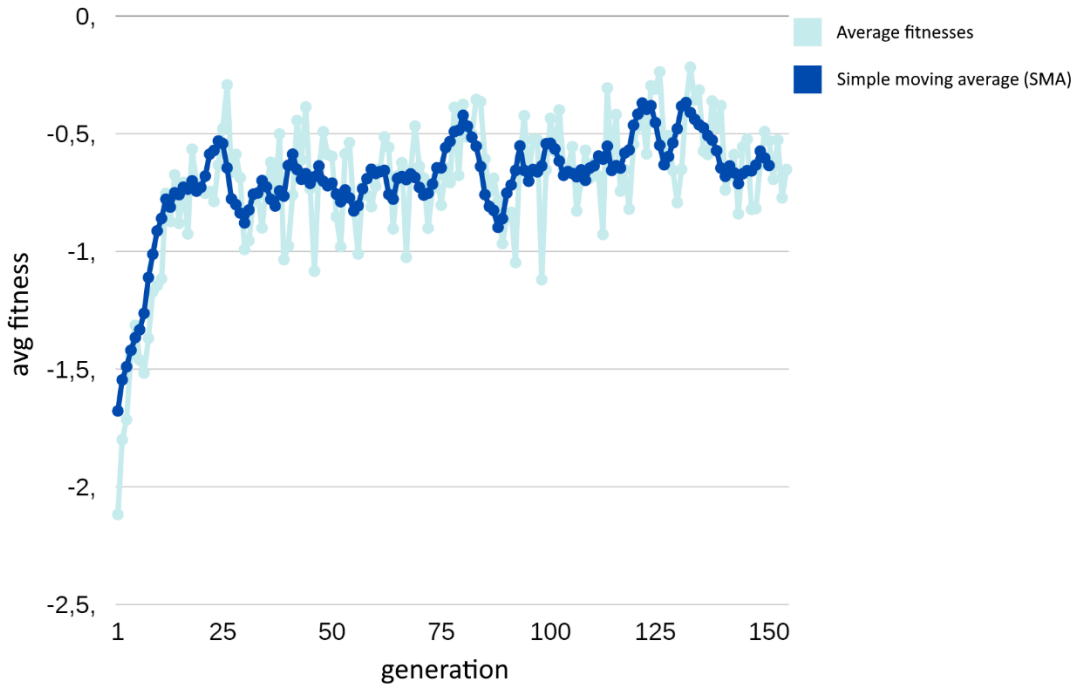


Figure 7-6 Average fitness of MIN Module 2 by generation

Results: It is very clear that on average the same configuration applied to a more complex scenario struggled more to find a valid solution. It managed to find a set of weights granting an overall consistent victory but considering the results obtained by the MIN Module 2 it is clear that there is a good improvable margin.

The trend of average fitnesses is similar however, maybe slightly growing in later generations, which overall tend to feature more and more average fitnesses above 0.5. None of them however has been positive.

This may be an indicator of how much the complexity of the training grows with few more units and in a slightly bigger map.

7.3.3. MIN Modules 3 and 4

- Maps: Training 1
- Matches: 4, two for each side of map Training 1
- Opponent: Normal AI
- Evaluation: Player value pre-deployed ([-4, 4])
- Best fitness 0.34300002 (Module 3, gen 273), 0.60486734 (Module 4, gen 370)
- Generations: 432 for MIN Module 3, 378 for MIN Module 4
- Evolution: Genetic Algorithm

MIN module configuration:

- Unit List: Infantry, Mech, Artillery, Light Tank, Heavy Tank, Anti Air, Recon, Neotank, Rocket Thrower
- Full (Expected) Infantry, Mech, Artillery, Light Tank, Heavy Tank, Anti Air, Recon, Neotank, Rocket Thrower
- Unit Influence Maps: Custom Allies, Custom Enemies, Standard Damage, Standard Attack
- Global Influence Maps: Once Map Defense
- Parameters:
    - Step multiplier: 0.5
    - Steps for indirect units: 1
    - Steps for direct units: 2
    - Weight per star: 2
    - Friendly Building Multiplier: 1.5
    - Friendly Factory Multiplier: 0.1

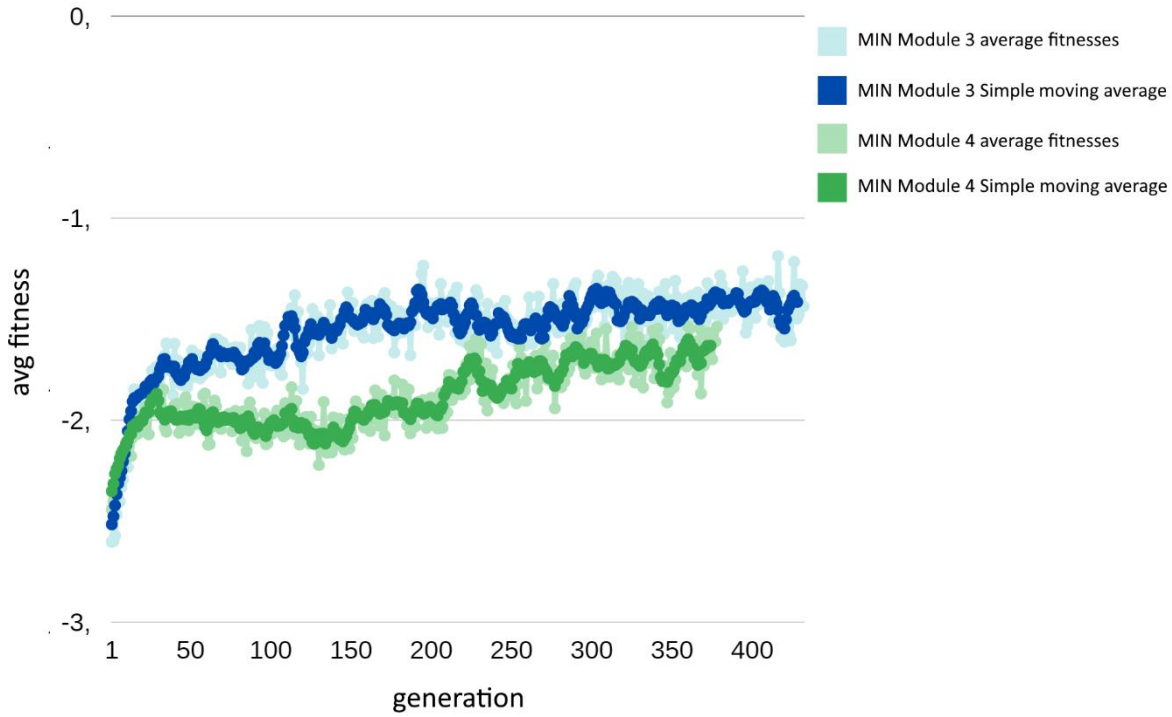


Figure 7-7 Average fitness of MIN Module 3 (blue) and 4 (green) by generation

Results: The MIN Module 3 and 4 were trained for testing transfer learning. They have the same exact configuration and training environment, but Module 3 was trained by having a basis of weights transferred from the MIN Module 1. It is clear that the higher number of units and map size did slow down progresses of MIN Modules, with both modules having obtained a result which is an overall victory but still close to a tie. Albeit having found a better solution MIN Module 4 (which can be attributed to some degree of luck in any case, since the second better result of Module 4 is less than the best one of Module 3), it is clear from Figure 7-7 that the transfer learning did indeed help to improve and stabilize more the average results.

7.3.4. CNN Module 1

- Maps: Training 3
- Matches: 4, two for each side of map Training 3
- Opponent: Very Easy AI
- Evaluation: Player value pre-deployed win bonus ([0, 6])
- Best fitness: 0.622089
- Generations: 181
- Evolution: NEAT

CNN module configuration:

- Units used: Infantry, Mech, Light tank, Heavy tank, Artillery, Anti Air, Helicopter
- Terrain Features: Defense value, Owner, Move_Feet, Move_Tank, Move_Air
- Unit Features: Move points, Can capture, Static (goodness against infantry), Static (goodness against vehicles), Static (goodness against air), Move_Feet, Move_Tank, Move_Air, Ammo_1
- total input depth: 14
- Layers:
  - Conv2D, 16x14 kernels 3x3
  - MaxPool2D
  - Activation, Tanh
  - Conv2D, 32x16 kernels 3x3
  - Activation, Tanh
  - [AutoDecoder]

Results: no module has been ever able to win even once, since the best fitness ever is < 1. This is believed to be caused by the NEAT evolution to a CNN, which is unstable, since a slight modification in the genome can cause lots of changes in kernels evaluations. More complex system didn't apparently bring any considerable benefit

7.3.5. CNN Module 2

- Maps: Training 3
- Matches: 4, two for each side of map Training 3
- Opponent: Very Easy AI
- Evaluation: Player value pre-deployed win bonus ([0, 6])
- Best fitness: 1.80892
- Generations: 146
- Evolution: NEAT

CNN module configuration:

- Units used: Infantry, Mech, Light tank, Heavy tank, Artillery, Anti Air, Helicopter

- Terrain Features: Defense value, Owner, Move_Feet, Move_Tank, Move_Air
- Unit Features: Move points, Can capture, Static (goodness against infantry), Static (goodness against vehicles), Static (goodness against air), Move_Feet, Move_Tank, Move_Air, Ammo_1
- Total input depth: 14
- Layers:
  - Conv2D, 6x14 kernels 3x3
  - MaxPool2D
  - Activation, Tanh
  - Conv2D, 12x6 kernels 3x3
  - Activation, Tanh
  - [AutoDecoder]

Results: A simpler model proved on average to perform better. However, given that the best model, and in total only 3 models across all generations won each at most 1 match proves that is still too inconsistent to consider the result positive. It is believed that the reason behind the failure is the same as the one of CNN module 1. Maybe since the model was simpler however it helped to stabilize evolution. This is notable because most of the best records are consistently better than the ones in the CNN module 1, although not enough to win consistently.

7.3.6. CNN Module 3

- Maps: Training 2
- Matches: 4, two for each side of map Training 2
- Opponent: Offensive Normal AI
- Evaluation: Player value pre-deployed win bonus ([0, 6])
- Best fitness: 1.84752 (gen < 228)
- Generations: 390
- Evolution: NEAT

CNN module configuration:

- Units used: Infantry, Mech, Light tank, Artillery, Anti Air
- Terrain Features: Defense value, Owner, Move_Feet, Move_Tank
- Unit Features: Move points, Can capture, Static (goodness against infantry), Static (goodness against vehicles), Move_Feet, Move_Tank
- Total input depth: 10

- Layers:
  - Conv2D, 8x10 kernels 3x3
  - MaxPool2D
  - Activation, Sigmoid
  - Conv2D, 14x8 kernels 3x3
  - Activation, Sigmoid
  - [AutoDecoder]

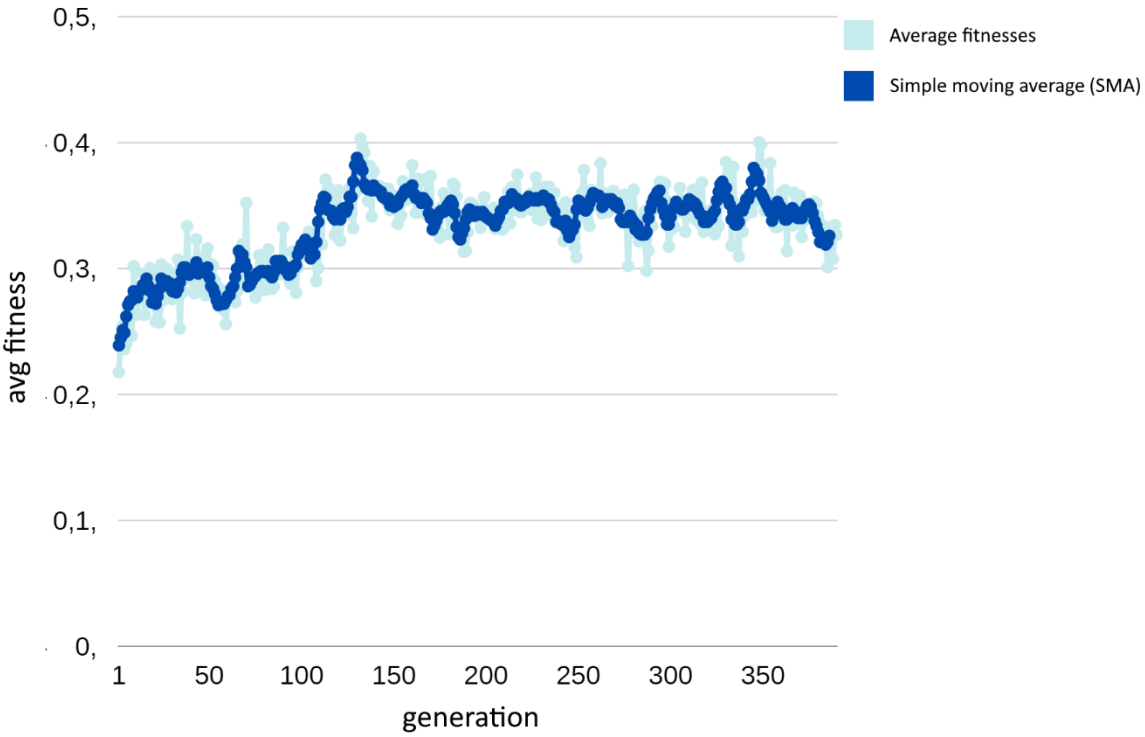


Figure 7-8 Average fitness of CNN Module 3 by generation

Results: In the simplest environment adopted in this experiments, the CNN Module 3 managed to win once out of the 4 matches at best. The trend of average fitness by generation shows that more complex models can bring improvements, however after an initial grow overall the fitness does not increase, which is in line with results obtained by the previous CNN modules.

7.3.7. Strategic Module

To test the strategic module, it was made play against several different AIs to see how it would adjust its behavior during different matches. First 12 matches are against the Normal AI, then 12 matches against Very Easy, and then Offensive Normal AI. After that the cycle repeats Chosen map was the Training Map 2.

Configuration of the Strategic Module adopted:

- Adapta Modules: Normal Offensive, Normal Defensive, MIN Module 1, MIN Module 3, CNN Module 1, CNN Module 3
- Building Module: Normal (not used, maps were predeployed)
- History Capacity: 16
- Past Multiplier: 0.8
- Aim: 0 (tries to overall tie)
- Square Variance: 0.2
- Random Variation Percentage: 0.25
- Randomize Probability: 0.25

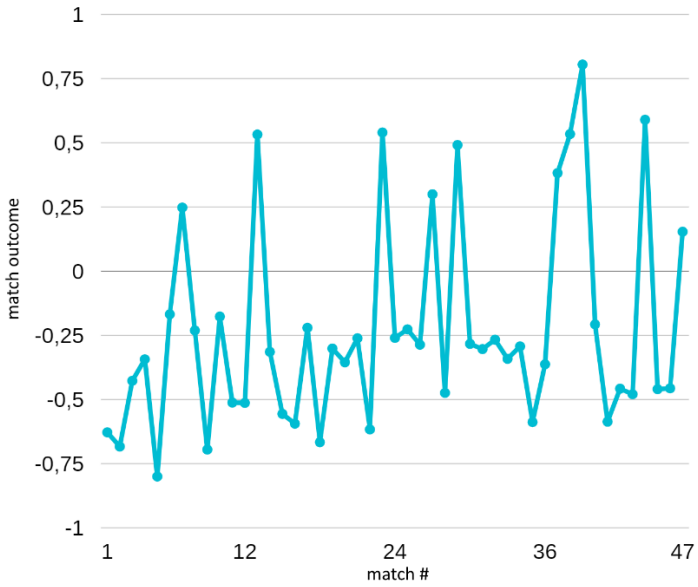


Figure 7-9 Strategy Module performance on a series of matches against different AIs

Results: the Strategic Module was tested with an Adapta AI configured with 2 modules per category: behavioral, MIN and CNN, in an attempt to summarize a more complete AI in terms of possible strategies. It managed overall to change its behavior between the matches, resulting in wins and losses sometimes by a good margin, which is considered to be a good result since it was against a behavioral AI, which supposedly has some tactics which are more or less effective against them. The fact that the opponent changed but the overall results didn't is it also considered a good indicator that the Adapta AI can adapt to the opponent, clearly given that it depends on the modules that are loaded in.

Similar experiments in other maps obtained comparable results.

8 Conclusion and future developments

The overall built architecture has shown the possibility of being in line with the aim of the thesis. This is promising for future expansions in higher complexity cases.

Multi Influence Network Modules has proven to be more stable w.r.t. experimental CNN Modules, which instead didn't obtain the same results overall.

The main challenge is definitely a matter of complexity, which can be tackled in a lot of different or new approaches in future researches.

The modularity of the ADAPTA architecture is its strongest point: it allows to train any type of modules, as its described in its chapter.

From a computational point of view for commercial game however it is not very practical to load a high number of modules because it will increase the computational time, especially with modules like MIN. In general it depends on the type of modules, and in such cases a different Strategic Module may be developed, which for instance activates only a maximum number of modules per match.

Hopefully this work can be used as a prompt for expansions or researches about this architecture, applied to Advance Wars or other strategy games.

For future researches on the same game, some possibilities are to expand concepts accounting for higher-sized maps or new/special units, or to attempt to utilize in a general way any new type of unit added by mods. CNN Modules can be expanded and revisited, and new typologies of modules can be created. It can be also considered to create custom Building Modules and create interesting behaviors also for building units.

9 Bibliography

[1] J. Schell, "The Experience is in the Player's Mind - Focus," in *The Art of Game Design: A Book of Lenses*, Burlington, Morgan Kaufmann Publishers (Elsevier), 2008, pp. 118-123.

[2] "THE FLOW THEORY APPLIED TO GAME DESIGN," [Online]. Available: https://thinkgamedesign.com/flow-theory-game-design/.

[3] M. Bergsma and P. Spronck, "Adaptive Spatial Reasoning for Turn-based Strategy Games," Tilburg centre for Creative Computing, Tilburg University, The Netherlands, Tilburg, 2008.

[4] H. Nan, B. Fang, G. Wang, W. Yang, S. Carruthers and Y. Liu, "Turn-Based War Chess Model and Its Search Algorithm per Turn," Hindawi, 2016.

[5] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu and S. Sutphen, "Checkers Is Solved," Sciencexpress, Alberta, Canada, 2007.

[6] K. O. Stanley and J. Gauci, "Autonomous Evolution of Topographic Regularities in Artificial Neural Networks," Orlando, Florida, 2010.

[7] P. Grünke, "Chess, Artificial Intelligence, and Epistemic Opacity," Információs Társadalom, 2019.

[8] billwall, "Computers and Chess - A History," 2017. [Online]. Available: https://www.chess.com/article/view/computers-and-chess---a-history.

[9] "Stockfish - GitHub," [Online]. Available: https://github.com/official-stockfish/Stockfish.

[10] K. Bharath, "AI In Chess: The Evolution of Artificial Intelligence In Chess Engines," 2 4 2021. [Online]. Available: https://towardsdatascience.com/ai-in-chess-the-evolution-of-artificial-intelligence-in-chess-engines-a3a9e230ed50.

[11] H. L. Bodlaender and F. Duniho, "Shogi (将棋): Japanese Chess," 9 9 1996. [Online]. Available: https://www.chessvariants.com/shogi.html.

[12] takodori, "Shogi computer beats female champ Shimizu," 12 10 2010. [Online]. Available: https://web.archive.org/web/20110708143357/http://blog.chess.com/view/shogi-computer-beats-female-champ-shimizu.

[13] "第27回 世界コンピュータ将棋選手権は新星「elmo」が制覇！～評価関数と定跡が公開 [tr: Il 27° World Computer Shogi Championship è vinto dall'astro nascente "elmo"! -La funzione di valutazione e il libro vengono rilasciati]," 11 5 2017. [Online]. Available: https://forest.watch.impress.co.jp/docs/serial/yajiuma/1058898.html.

[14] [Online]. Available: http://www2.computer-shogi.org/wcsc27/index_e.html.

[15] "Elmo (shogi engine) - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Elmo_(shogi_engine).

[16] "Wikipedia - Go (game)," [Online]. Available: https://en.wikipedia.org/wiki/Go_(game).

[17] P. Shotwell, "The Game of Go: Speculations on its Origins and Symbolism in Ancient China," 1994, rev. 2008.

[18] J. Tromp and G. Farnebäck, "Combinatorics of Go," 2016.

[19] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," Springer, Turin, Italy, 2007.

[20] "Wikipedia - Computer Go," [Online]. Available: https://en.wikipedia.org/wiki/Computer_Go.

[21] C. Metz, "Wired - Google's AlphaGo Continues Dominance With Second Win

in China," Wired, 25 05 2017. [Online]. Available: https://www.wired.com/2017/05/googles-alphago-continues-dominance-second-win-china/.

[22] V. James, "DeepMind's AI agents conquer human pros at StarCraft II," The Verge, 24 01 2019. [Online]. Available: https://www.theverge.com/2019/1/24/18196135/google-deepmind-ai-starcraft-2-victory.

[23] S. Ian, "AI becomes grandmaster in 'fiendishly complex' StarCraft II," The Guardian, 30 10 2019. [Online]. Available: https://www.theguardian.com/technology/2019/oct/30/ai-becomes-grandmaster-in-fiendishly-complex-starcraft-ii.

[24] M. Rohs, "Preference-based Player Modelling for Civilization IV," 2007.

[25] S. Branavan, D. Silver and R. Barzilay, "Non-Linear Monte-Carlo Search in Civilization II," MIT, University College London, Cambridge & London, 2013.

[26] D. P. Liebana and R. Gaina, "Tribes: a Turn-Based Strategy Games Framework," UK EPSRC, 2020. [Online]. Available: https://gaigresearch.github.io/projects/Tribes.

[27] D. Perez-Liebana, Y.-J. Hsu, S. Emmanouilidis, B. D. A. Khaleque and R. D. Gaina, "Tribes: A New Turn-Based Strategy Game for AI Research," School of Electronic Engineering and Computer Science, Queen Mary University of London, London, 2020.

[28] A. Dockhorn, J. Hurtado-Grueso, D. Jeurissen and D. Perez-Liebana, "STRATEGA - A General Strategy Games Framework," School of Electronic Engineering and Computer Science, Queen Mary University of London, UK, London, 2020.

[29] R. Muller, "CommanderWars project: https://github.com/Robosturm/Commander_Wars".

[30] "Advance Wars Wikia," [Online]. Available: https://advancewars.fandom.com/wiki/Damage_Formula.

[31] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper and E. Postma, "Adaptive game AI with dynamic scripting," Springer Science + Business Media, LLC, 2006.

[32] P. Tozour, "Using a Spatial Database for Runtime Spatial Analysis," in *AI Programming Wisdom 2*, Charles River Media, 2004, pp. 381-390.

[33] K. O. Stanley, J. Gauci and D. D'Ambrosio, "A Hypercube-Based Indirect Encoding for Evolving Large-Scale Neural Networks," Orlando, Florida, 2009.

[34] O. Ronneberger, P. Fisher and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on medical image computing and computer-assisted intervention*, 2015.

[35] K. O. Stanley, "Evolving Neural Networks through Augmenting topologies," The MIT Press Journals, Austin, 2002.

[36] K. O. Stanley, "Compositional pattern producing networks: A novel abstraction of developement," Springer Science+Business Media, LLC, 2007.

[37] "Advance Wars Wiki Fandom," [Online]. Available: https://advancewars.fandom.com/wiki/List_of_Units.

[38] "Game Complexity - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Game_complexity.

[39] J. Tromp, "ChessPositionRanking - Github," [Online]. Available: https://github.com/tromp/ChessPositionRanking.

[40] "Shannon number - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Shannon_number.

[41] M. S. J. R. Hiroyuki Iida, "Computer Shogi," Elsevier, Shizouka & Oxford, 2002.

[42] S.-J. Yen, J.-C. Chen, T.-N. Yang and S.-C. Hsu, "Computer Chinese Chess," International Computer Games Association Journal, Taipei, Taiwan, 2004.

[43] "Lockyard/Commander_Wars," [Online]. Available:

https://github.com/Lockyard/Commander_Wars.

[44] "Draw.io," [Online]. Available: https://app.diagrams.net/.

[45] "Normal Distribution Plot - Desmos," Desmos, [Online]. Available: https://www.desmos.com/calculator/0x3rpqtgrx.

[46] "Canva," [Online]. Available: https://www.canva.com/graphs/.

List of Figures

List of Tables

Tools and Source Code

- The code of this project on GitHub: [43]
- The code of the original project Commander Wars: [29]
- For the ADAPTA schemes (chapters 4, 5, 6): Draw.io [44]
- For the gaussians graphs in 4.3: Desmos [45]
- For average fitnesses graphs in 7.3: Canva [46]

Acknowledgments

It's been a wonderful journey.

I want to thank every professor I had during my journey at Polimi, for having taught me amazing things. Each one with their own unique way of teaching. I am thankful for everything I learned and the friends and experiences I made along the way.

I want to thank, among my professors, especially professor Daniele Loiacono for following me during my thesis and professor Pier Luca Lanzi for organizing the GGJs, the VDP course and related events.

I want to thank every member of my family for always supporting me, for being a good and amazing person, each in their own way.

I want to thank every friend and good person I met during these years, with which I shared good life experiences and memories.

I want to thank Rob Müller, aka Robosturm, for creating Commander Wars, the wonderful and still running and updating project, and for being so nice and available.

I want to thank every person which has contributed to creation of incredible videogames, music, movies, series, stories, pictures, food, art, able to leave a mark in me and changed me a bit for the better.

I want to thank myself for every mistake I made and lessons I learned, for every good thing I've done and for who I am today.

To all of you, Thank you.

Lorenzo

Your past is fractured and your future is not yet written, but you have traveled through this world as a beacon of good, and that is all that matters.

Ahrah - Dust: An Elysian Tail