FFI Norwegian Defence Research Establishment

22/02423

FFI-RAPPORT

Imitation learning for modelling air combat behaviour

– an exploratory study

Patrick Gorton
Martin Asprusten
Karsten Bråthen

Imitation learning for modelling air combat behaviour

— an exploratory study

Patrick Gorton
Martin Asprusten
Karsten Bråthen

Norwegian Defence Research Establishment (FFI) 13 January 2023

Keywords
Kunstig intelligens
Maskinlæring
Datagenererte styrker
Kampfly
Modellering og simulering

FFI report
22/02423

Project number
1554

Electronic ISBN
978-82-464-3454-4

Approvers
Ole Martin Mevassvik, *Research Manager*

The document is electronically approved and therefore has no handwritten signature.

Summary

Fighter pilots commonly use simulators to practice their required tactics, techniques and procedures. The training may involve computer-generated forces controlled by predefined behaviour models. Such behaviour models are typically manually crafted by eliciting knowledge from experienced pilots and take a long time to develop. Nonetheless, these behaviour models generally fall short due to their predictable nature and lack of adaptivity, and the instructors must spend time manually monitoring and controlling aspects of these forces. However, recent advances in artificial intelligence (AI) research have developed methods capable of producing intelligent agents that beat expert human players in complex games such as Go and StarCraft II.

Similarly, one may use methods from AI to compose advanced behaviour models for air combat, allowing the instructors to focus more on the pilots' training progression rather than manually controlling their opponents and teammates. Such intelligent behaviour must perform realistically and follow the correct military doctrines to prove useful for pilot training. One possible way of achieving this is through imitation learning, a machine learning (ML) type where agents learn to imitate examples given by expert pilots.

This report summarizes work on optimizing air combat behaviour models using an imitation learning technique. These behaviour models are expressed as behaviour transition networks (BTNs) controlling the computer-generated forces, simulated by the Next Generation Threat System (NGTS), a military simulation application aimed mainly toward the air domain. An adapted version of the genetic algorithm Neuroevolution of Augmenting Topologies (NEAT) optimizes the BTNs to behave similarly to demonstrations of pilot behaviour. As with most ML methods, NEAT requires many consecutive behaviour simulations to yield satisfying solutions. NGTS is not designed for ML purposes, so a system was developed around NGTS that automatically handles simulation and data management and controls the optimization process.

A set of experiments were performed in which the developed ML system optimized BTNs to imitate example behaviours across three simple air combat scenarios. The experiments show that the adapted version of NEAT (BTN-NEAT) produces BTNs that successfully imitate simple demonstrations. However, the optimization process took considerable time, up to 44 hours of computation or 92 days of simulated flight time. The slow optimization was mainly influenced by NGTS's inability to run fast while remaining reliable. This reliability issue is caused by NGTS's lack of time management, which would have associated the agents' states with simulation time stamps. To achieve successful behaviour optimization with more complex scenarios and demonstrations, one should simulate the behaviours much faster than in real-time with high reliability. Therefore, we consider NGTS not to be well-suited for future ML work. Instead, a lightweight air combat simulation designed for ML purposes capable of running fast and reliably is needed.

Sammendrag

Kampflypiloter trener på taktikker, teknikker og prosedyrer ved hjelp av simulatorer. Der inngår datagenererte styrker med forhåndsbestemte oppførsler. Å utvikle slike oppførsler er omfattende. Tradisjonelt lages de for hånd, i tett samarbeid med erfarne piloter. Dessverre har de resulterende modellene ofte svakheter, for eksempel ved at de framstår forutsigbare eller er lite tilpasningsdyktige. Slike problemer gjør at simulatorinstruktørene blir nødt til å følge opp og kontrollere aspekter ved de datagenererte styrkene manuelt. Simulatorinstruktørene bør avlastes fra denne jobben, slik at de best mulig kan bidra til at pilotene tilegner seg nødvendige ferdigheter.

Behovet for denne typen støtte har ført til en økning av forskningsaktiviteter knyttet til modellering av luftstridsoppførsel. Nylige framskritt innenfor kunstig intelligens har bidratt med metoder som gjør agenter i stand til å spille komplekse spill som Go og StarCraft II. Slike metoder kan også brukes til å lære avansert kampflyoppførsel, til bruk i simuleringsbasert trening. Likevel må kunstige, intelligente piloter være nyttige. I så måte må de kunne opptre realistisk, og i tråd med militær doktrine. Dette krever at de er i stand til å etterligne ekspertoppførsel.

Denne rapporten oppsummerer arbeid knyttet til imiteringslæring for luftstridsoppførsel ved bruk av Next Generation Threat System (NGTS), som er et simuleringssystem med særlig vekt på luftdomenet. Oppførselsmodellene er uttrykt som oppførselstransisjonsnettverk (BTN). For å optimalisere dem ble en tilpasset utgave av den genetiske algoritmen Neuroevolution of Augmenting Topologies (NEAT) benyttet. Den fikk navnet BTN-NEAT. På lik linje med andre maskinlæringsmetoder krever NEAT at mange oppførsler simuleres og evalueres, for å kunne finne gode løsninger. NGTS er ikke rettet spesifikt mot maskinlæringsformål. Derfor ble det utviklet et eget maskinlæringssystem som kobler sammen simulering, oppførselsmodeller og optimaliseringsmetode.

Maskinlæringssystemet ble benyttet i eksperimenter med utgangspunkt i tre enkle luftkampscenarioer. I disse eksperimentene lærte agenter å handle i tråd med demonstrert oppførsel, ved å bruke imiteringslæring. BTN-NEAT lyktes i å lage BTN-er som etterlikner demonstrasjonene, selv om læringsprosessen krevde mer tid enn antatt. Årsaken er at NGTS ikke kan kjøre raskt, og samtidig gi pålitelige data. Det tok opp mot 44 timer med beregning å lære egnede oppførsler. Det tilsvarer 92 dager med simulert flytid. Eksperimentene avslørte at NGTS har svakheter knyttet til tidsstyring. Det gjør simuleringssystemet lite attraktivt i videre arbeid knyttet til modellering av luftstridsoppførsel med kunstig intelligens. I stedet er det behov for å utvikle en enkel og beregningsmessig hurtig luftstridssimulering til bruk for maskinlæring.

Contents

Abbreviations

AI	Artificial intelligence
ANN	Artificial neural network
API	Application programming interface
BTN	Behaviour transition network
CAP	Combat air patrol
CGF	Computer-generated forces
DIS	Distributed interactive simulation
DRL	Deep reinforcement learning
EA	Evolutionary algorithm
EC	Evolutionary computing
FSM	Finite state machine
GA	Genetic algorithm
HLA	High level architecture
HPC	High-performance computing
IL	Imitation learning
ML	Machine learning
NEAT	Neuroevolution of Augmenting Topologies
NGTS	Next Generation Threat System
PDU	Protocol data units
RL	Reinforcement learning
SDK	Software development kit
SME	Subject matter expert
TTP	Tactics, techniques and procedures
YAML	YAML Ain't Markup Language (prev. Yet Another Markup Language)

1 Introduction

Fighter pilots learn and maintain their tactical skills through rigorous training. A considerable amount of the training is simulation-based, during which trainees face friendly and opposing forces that should preferably behave in a manner that accelerates training and builds desired competence. Computer-generated forces (CGFs), which are autonomous, computer-controlled entities, are used to play the role of these friendly and opposing forces. Ideally, using CGFs in simulation-based training should provide benefits such as increasing training availability for pilots and reducing the need to involve subject matter experts (SMEs) during training. However, manually modelling CGF behaviours sufficiently representative for the instructional role is tedious and has proven challenging. As a result, current handcrafted behaviour models are often predictable, fail to adapt to new situations or behave realistically concerning military doctrines, tactics, techniques, and procedures (TTP). Maintaining a realistic experience in simulation-based air combat training is crucial to ensure that trainees attain the necessary skills. Still, since the performance and behaviour of CGFs are considered inadequate, SMEs tend to micromanage the CGFs during training sessions, which is unfortunate since SMEs are costly, their time is valuable, and their numbers are limited.

Recent advances in AI research have developed methods capable of producing intelligent agents that beat expert human players in complex games such as Go [1] and StarCraft II [2]. With these advances, learning instructive and adaptive agent behaviour for air combat has become a growing research field of interest. However, in order to be useful, the pilots' simulated opponents and allies must behave realistically and in line with military doctrines, as opposed to, e.g. attempting to win an engagement at any cost. Several contributions in the research field focus on reinforcement learning approaches, and some promising results have been shown. However, even with careful design of objective functions, reinforcement learning agents risk learning policies that are suboptimal for use in pilot training, meaning they act differently from what is expected according to established doctrine and TTP. An alternative approach is to provide ML algorithms with expert demonstrations from which they can extract pilot-specific knowledge and integrate this into the behaviour models used by agents. To our knowledge, little or no prior research has explored this approach in the air combat domain.

This report presents work in which an imitation learning algorithm based on the Darwinian principles of natural selection is used to produce air combat behaviour models expressed as behaviour transition networks (BTNs). While BTNs have appeared in prior work related to air combat behaviour modelling using reinforcement learning, this work investigates whether BTNs are suitable for imitation learning. Next Generation Threat System (NGTS) is used to simulate the BTNs, and assessments are made to consider the suitability of this simulation system for machine learning (ML). An ML system has been developed, comprising the tools and methods needed to successfully produce air fighter agents using NGTS and the selected learning algorithm. This ML system automatically handles simulation and data management and controls the learning algorithm. Simple air combat scenarios were defined and used in a series of experiments conducted using this ML system where BTNs that reflect demonstrated pilot behaviour were produced.

A few delimitations are made to limit the scope of the work. The ML system developed is not production grade but rather a proof of concept. Thus, the scenarios and pilot demonstrations used in the experiments are kept simple. Specifically, these are one vs one scenarios with demonstrations restricted to movement in two-dimensional space. Further, the behaviour demonstrations are based on BTNs hand-crafted by the report authors, not by professional pilots.

This report is intended for researchers working on topics related to military training and AI, preferably with knowledge of air combat and behaviour modelling, and is organized as follows. Chapter 2 presents the background for the work, covering concepts relevant to air combat training and simulation, artificial intelligence theory, and related work. Chapter 3 covers the selected learning algorithm used in the experiments and its configurations, while Chapter 4 presents the processes and tools that make up the ML system. Chapters 5 and 6 review the setup and execution of the experiments by defining air combat scenarios and behaviour demonstrations and presenting the results. These results, as well as the performance of the ML system and NGTS, are discussed in chapter 7. Chapter 8 brings the report to a close with concluding remarks and thoughts on future work.

2 Background

This chapter outlines background information related to the research documented in this report. The first part introduces the application domain of fighter pilot training, specifically fighter pilot simulation-based training. For simulation-based fighter pilot training, a simulation system must provide a synthetic environment, including the simulation of adversaries and own forces. Elements of such a synthetic environment are covered, including the process typically employed today for modelling these entities' behaviours. Next Generation Threat System, which is used in this study, is an example of a simulation system that provides such a synthetic environment. NGTS is briefly introduced, including how the system expresses behaviour models of entities using BTNs. Since this work studies ML as an alternative to the traditional behaviour modelling process, the chapter also covers general ML concepts and specific ML methods and tools relevant to this work. The chapter also covers software virtualization and containerization tools, which can help automate and parallelize simulations. Lastly, a section on related work brings the chapter to a close.

2.1 Simulation-based fighter pilot training

Fighter pilots must perform a comprehensive annual continuation training programme to stay combat-ready. Strict requirements for the number of flying hours are in place. Although parts of the training programme, such as emergency procedures training, have long been fulfilled using simulators, simulator-based training has come to make up a much larger part of training in recent years. Due to cost, time, operation security, air space restrictions, safety regulations, and environmental and climate considerations, simulators are expected to replace live training to an even larger extent in the coming years. This trend is enabled by developments in simulation technologies, including realistic synthetic environments and the ability to network simulators so that larger forces can be trained. Thus today, a full mission simulator provides effective training for a large part of the training elements constituting a continuation training programme.

A full mission simulator must include both a virtual simulator and representations of the fighter aircraft environment. The virtual simulator constitutes the cockpit and the mission systems, a high-fidelity flight dynamics model, an out-of-the-window visual system, sensors, weapons, communication systems, and possibly a motion platform. Such a full mission simulator is shown in figure 2.1. Additionally, the surrounding environment must be represented; the physical environment (natural and man-made ground environment and the atmospheric environment) and the entities, both friendly and threat entities (e.g. aircraft, ground-to-air missile systems, etc.). These entities are normally created by constructive simulation, i.e. simulated humans operating simulated systems, also called CGFs and synthetic agents.

A CGF is an autonomous or semi-autonomous actor in military simulation-based training and decision support applications [3]. According to the NATO Long-Term Scientific Study LTSS/48, CGF is defined as “A generic term used to refer to computer representations of entities in simulations which attempts to model human behaviour sufficiently so that the forces will take some actions automatically (without requiring man-in-the-loop interaction)“ [4]. Generally speaking, CGFs can be considered intelligent simulated elements operating in a simulated environment, similar to how

Figure 2.1 *An example of a full mission simulator. Source: [5].*

humans would do in the real world. Equipped with autonomy and intelligence, a CGF may perceive, communicate and coordinate with other entities.

The main challenge of constructing a CGF is to design its behaviour, which should be accurate and realistic to best cope with and reflect the nature and uncertainty of an authentic battle environment. Representing human aspects in CGFs is especially challenging. For a fighter aircraft, these human aspects include how the pilot behaves in order to obtain situation awareness, manoeuvre the aircraft, use sensors and weapons, and communicate with other pilots and command and control elements. The current constraints in the behaviour models force instructors and role players to micromanage the constructive entities, restricting the complexity of scenarios that can be managed and trained realistically.

Today, behaviour models are based on symbolic AI. These models are handcrafted and typically rule-based [6] or scripted. The rules and scripts of these models lay the foundations of CGFs, and are typically derived from standard doctrine and TTP publications, and through tedious interview processes eliciting knowledge from experts on the behaviour of both own forces and adversaries.

2.1.1 Next Generation Threat System

NGTS is a synthetic environment generator owned by the Naval Air Warfare Center Aircraft Division (NAWCAD) of the United States Armed Forces [7]. The software is used to support military training, testing, analysis, research, and development and is targeted especially towards the air combat domain. It can model both friendly and opposing units across most military domains and aims to provide a realistic theatre environment.

NGTS consists of a set of program components that run separately from each other and perform different roles. The NGTS Core Simulation Engine performs all simulation, but lacks an interface for performing anything more than simple tasks, such as loading, starting and stopping a scenario. The Battle Monitor provides more detailed control through a visual interface, and allows – among other things – giving orders to simulated units and adding and removing units from a scenario. Other components include an after-action review tool and the Behavior Editor, in which behaviours for computer-generated forces can be specified using the BTN modelling language.

NGTS can communicate with other simulation systems using several communication protocols, such as the Distributed Interactive Simulation (DIS) [8] protocol. NGTS is also able to communicate using tactical data links, such as Link 16 [9] over the Joint Range Extension Application Protocol (JREAP) [10].

2.1.2 Behaviour transition networks

BTNs are similar to finite state machines (FSMs) but include several augmentations. They also share some similarities with Petri Nets. Stottler Henke Associates developed the BTN in the early 2000s [11, 12, 13], but, to our knowledge, no formal definition of BTN has been provided.

Syntactically, BTNs are bipartite-directed multigraphs. They are bipartite since their vertices may be divided into two sets, *action* and *transition* vertices, such that each edge is directed from an element in one set to an element in the other set. A transition vertex expresses a condition that, when fulfilled, transitions between action vertices. Thus, edges connect action vertices to transition vertices and transition vertices to action vertices. BTNs are multigraphs since action vertices may connect to multiple transition vertices expressing different conditions. In this event, the transition vertices are assigned priorities defining the order to evaluate them in case more than one condition is fulfilled. All BTN must have a *start action* vertex and an *end action* vertex. An example of a BTN is shown in figure 2.2.

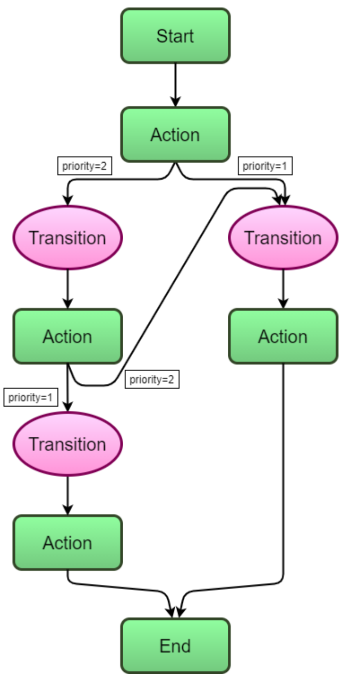


Figure 2.2 *This example BTN shows how actions connected to more than one transition have assigned priorities.*

Definition A behaviour transition network $G = (V, E)$ is a bipartite-directed multigraph, where $V = \{Start, End, v_1, v_2, ..., v_s\}$ is a set of vertices and $E = \{e_1, e_2, ..., e_r\}$ is a set of directed edges $e_i = (v_j, v_k)$ with $v_j, v_k \in V$. The set V can be partitioned into two disjoint sets B and T such that $V = B \cup T, B \cap T = \emptyset$, and for each directed edge, $e_i \in E$, if $e_i = (v_j, v_k)$, then either $v_j \in B$ and $v_k \in T$ or $v_j \in T$ and $v_k \in B$.

Furthermore, BTNs may be hierarchical, wherein one or more action vertices refer to other BTNs. This property of BTNs is essential to model complex behaviour and avoid the state explosion of FSMs. Graphically, actions are expressed as rectangles (often with rounded corners), while transitions are expressed as ellipses, with the name of the specific action or transition located inside these shapes. They have input and output ports at the top and bottom for connecting to other vertices using edges.

The execution of a BTN is relatively simple, beginning at the start vertex node and proceeding to the following vertex. If the following vertex is an action, the action gets executed. If the following vertex is a transition, the transition occurs only when the condition is met. If the following vertex branches to more than one transition, their conditions are evaluated in the order given by their priority scheme as described above. Concurrency is not supported. Note that in hierarchical BTNs, an action vertex may refer to another BTN, in which case any subsequent transition vertex still gets evaluated and can interrupt this sub-BTN execution. This behaviour is an important aspect of BTNs that helps reduce their complexity. Finally, the execution terminates in the end action vertex. When controlling simulated entities, each entity will have its own BTN. The BTNs can read and write messages to and from blackboards enabling information sharing and cooperation between the entities.

As mentioned, modelling BTNs in NGTS is done using its Behaviour Editor. In NGTS, action vertices are called commands, and transition vertices are called triggers. Note that NGTS follows a slightly less strict definition of BTNs than the one outlined in this section: NGTS allows edges that connect action nodes to action nodes and transition nodes to transition nodes, a property contrary to the definition given above. All the other rules for BTNs still hold.

Another type of software for modelling BTNs is SimBionic [14]. SimBionic is an open-source tool comprising a run-time system that reads and executes BTNs, and a debugger for testing and debugging behaviour.

2.2 Machine learning

Machine learning is a branch of artificial intelligence which concerns the design and use of algorithms that make computers able to learn from data and use this knowledge to make predictions or decisions. Typical ML applications include data classification, clustering, dimensionality reduction, agent control, and more.

Conceptually, most ML methods deal with creating some function that takes a certain input and generates a usable output. For instance, an image classification function could take a vector containing the greyscale values of each image pixel as input, and return a vector containing probabilities that the image fits a given category from a set of possible categories. Machine learning involves automatically refining the parameters of such functions to provide useful outputs. Different ML methods use different types of functions and often different methods of refining the function parameters.

Machine learning can be divided into three main categories: supervised, unsupervised, and reinforcement learning [15]:

- Supervised learning consists of learning mappings from input to output, and the ML model is trained using example data that shows the correct output for a given input. Supervised learning is often used for data classification: a typical task is to classify the contents of an image.
- Unsupervised learning aims to find structures in data, like clusters of data points in a data set. Unsupervised learning algorithms are not trained using example data but instead tasked to minimize some heuristics of the data. For instance, if the algorithm is tasked with finding a certain number of clusters of data points, the algorithm would have to find the best location

for each cluster centre and assign each point to a cluster. The heuristic could then be to find these such that it minimizes the mean-squared distance from each data point to the centre of its assigned cluster. Unsupervised learning is obviously used for data clustering but can also be used for dimensionality reduction.

- Reinforcement learning (RL) deals with learning goal-directed behaviour for agents. Typically, an RL algorithm will be given some state as input and then choose an action based on that state. State-action pairs are given a reward value depending on whether taking that action leads to a desirable outcome. The RL model is refined in order to pick the action giving the highest reward for a given state. RL is often used in order to control agents in environments where it is difficult to know whether the outcome of a particular action will be desirable without attempting it first.

This report focuses on using a special kind of supervised learning, *imitation learning* (IL), to generate behaviour models for CGFs in the air-combat domain.

2.2.1 Imitation learning

An intelligent agent is an autonomous entity that perceives its environment through sensors and carries out actions that affect the environment and help the agent achieve its goals [16]. Such agents may perform various tasks in different environments, e.g. in the field of robotics. However, as the complexity of the environment increases, so does the challenge of manually modelling the agent's behaviour. Instead of defining increasingly complex behaviours that consider every eventuality, it might be easier to just provide examples of how to solve a task and let the agent learn from these examples. This is the goal of IL.

IL is the study and application of algorithms that use *demonstrations* to make agents learn certain behaviours [17]. The behaviour of an agent is determined by a function that takes the agent's state as input and returns the preferred action for the agent to take. This function is normally referred to as the agent's *policy*. In addition to using demonstrations, policies can be learned using *experiences*. These experiences can originate from the agent itself, as is the case when IL is combined with RL, or they can be created by another agent or human. An experience consists of a previously encountered state, the action that was taken in that state, and the reward that was gained from that action. While the actions of a demonstration are assumed to be optimal, the actions from experiences are usually not.

Russel and Norvig [16] define the terms demonstration and experience as follows.

- **A demonstration** is presented as a pair of input and output (s, a) where s is a vector of features describing the state at that instant and a is the action performed by the demonstrator.
- **An experience** is presented as a tuple (s, a, r, s') where s is the state, a is the action taken at state s, r is the reward received for performing action a, and s' is the new state resulting from that action.

There exist various approaches to IL, but the two major classes of methods are *behavioural cloning* and *inverse reinforcement learning*. Behaviour cloning aims to learn a policy by directly mapping states to actions or trajectories in a standard supervised learning fashion [18]. The demonstrations used in behaviour cloning are considered optimal; thus, the agent's performance will not exceed that of the demonstrator. Inverse reinforcement learning is closely related to RL in that it also uses a reward function. However, in inverse reinforcement learning, the reward function is unknown, and the learner attempts to recover it from a given policy or demonstrations of this policy [17]. When a suitable reward function is found, the agent's policy may be refined further using RL and the new reward function, and the resulting policy may even be better than the demonstrator's policy.

Although IL and supervised learning are similar in many circumstances, some properties distinguish the two approaches. For example, when creating a demonstration for IL purposes, the demonstrator may perform actions that are impossible for the learning agent to recreate. A robot may, for instance, not be able to perform the same actions as a human due to the limitations of robot joints [17]. Such properties are not typical in the standard supervised learning formulation.

2.2.2 Artificial neural networks

Most modern ML methods take inspiration from biological processes, such as biological neural networks. An artificial neural network (ANN) is a composition of artificial neurons, or *nodes*, that are connected to one another in layers. Each node takes input from the output values of nodes in the previous layer, computes some weighted combination of the input values, and passes it on as output to be used by the next layer. These weighted connections loosely resemble the behaviour of brain synapses between biological neurons, and the output value of a given node is known as the *node activation*, in an analogy to the electrical activation of biological neurons.

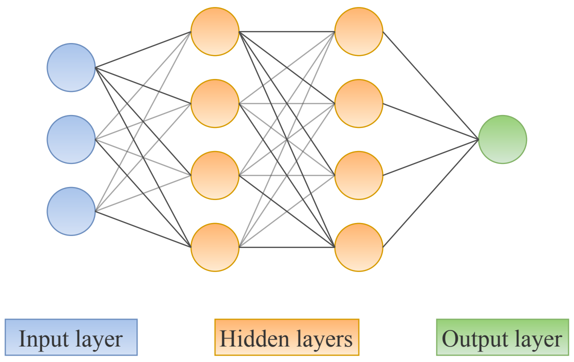


Figure 2.3 *A simple artificial neural network illustrating neurons, layers and weighted connections. Note that each layer, including the input and output layers, can have an arbitrary number of nodes. Source: [19].*

In essence, an ANN is used to approximate some function or system that gives the desired output for a given input. This function does not need to be known. An example could be an image classification function that takes a vector containing the colour values of each pixel in the image as input, and gives a vector containing probabilities that the image belongs to a set of categories as output, e.g. the category of images containing a cat, a dog, and so forth. Such a function would be exceedingly difficult to find analytically. However, the neural network is able to approximate this unknown function by refining its parameters until it gives desirable outputs.

The term *network topology* is used to describe how many layers a network has, how many nodes it has, and how the nodes are connected [20]. Figure 2.3 shows a *fully connected network*, sometimes referred to as a dense neural network, in which every node is connected to all nodes in its preceding and following layer. Other network structures exist, like recurrent networks, that are often used to process time series data, or residual networks, which use shortcuts to skip certain layers. These structures are not discussed here. Networks with more than one *hidden layer* (as shown in figure 2.3) are known as deep neural networks and can, under the right circumstances, approximate more complex functions than shallower networks.

The *activation* of a given node k is calculated using the following expression:

$$a_k = b_k + \sum_j w_{jk} a_j \tag{2.1}$$

Here, b_k is the *bias* of node k, which is a constant value that is specific to each node in the neural network. j represents the nodes that provide input to node k; in a fully connected network, this is every node in the previous layer to k. w_{jk} is the weight given to the input from node j by node k. Finally, a_j is node j's output value.

It is also common to represent equation 2.1 without the bias term, b_k. In this case, a *bias node* is instead added to each layer of the neural network, with a constant output of 1. The bias term b_k then instead becomes a part of the weights w_{jk}; since every node has a constant input of 1 from a *bias node*, the weight a node gives to this constant input can replace the b_k term.

In equation 2.1, the b_k and w_{jk} values of each node are *trainable* parameters; ANNs learn to approximate a function by optimizing these parameters so that the network gives the desired output. Normally, this optimization is performed by initializing the trainable parameters to some set of values according to some initialization scheme. A set of inputs for which the desired output is known is then passed through the network, and the outputs are compared with the desired outputs using a *loss function*, which represents how different the actual output is from the desired output. The training variables are then tuned in order to minimize the loss function. How this is done depends on the optimization method chosen; often, the loss function is differentiated with respect to the trainable parameters, and this differential is used to update the parameters. The procedure of optimizing the trainable parameters is normally referred to as *training* the network.

The activation values given by equation 2.1 are strictly linear. However, it is not always the case that the unknown function to be approximated will be linear. In order to make ANNs represent nonlinear functions, some nonlinear transformation $\phi(a_k)$ is normally applied to the activation values before they are passed to the next node [21, p. 165]. This type of transformation may be referred to as a nonlinear activation function or simply a nonlinearity. Examples of commonly used nonlinearities are the sigmoid function:

$$\phi(a_k) = \frac{1}{1 + e^{-a_k}} \tag{2.2}$$

the hyperbolic tangent function:

$$\phi(a_k) = \frac{e^{a_k} - e^{-a_k}}{e^{a_k} + e^{-a_k}} \tag{2.3}$$

and the rectified linear unit function:

$$\phi(a_k) = \begin{cases} a_k, & a_k > 0 \\ 0, & a_k \leq 0 \end{cases} \tag{2.4}$$

It is common to choose a function that is easily differentiable since most optimization methods depend on differentiating the output of the neural network.

2.2.3 Evolutionary algorithms

Evolutionary computing (EC) is a subfield of AI dealing with a class of algorithms inspired by evolutionary processes. These algorithms are called *evolutionary algorithms* (EA) and are applied to a wide range of computational problems. EAs do not guarantee optimal solutions but are designed to deliver satisfactory solutions within an acceptable time [22, p. 21]. Generally, an EA creates a population of candidate solutions to the problem that is to be solved. The solutions are then scored according to some evaluation function, and the best solutions get a chance to contribute to the creation of new solutions. One of the most widely used EAs is the *genetic algorithm* (GA). Inspired by the Darwinian principles of natural selection, GAs allow members of a population to interact with each other and reproduce [23, p. 301]. The reproduction generally involves combining aspects of members from the current generation's solutions. By introducing random alterations to their offspring, a large space of possible solutions is explored. Usually, EAs continue creating new generations until a termination condition is reached. This could be that a satisfactory solution has been reached, that the evolution has stagnated, or that too much time has elapsed. The type of EA one is dealing with is characterised by whether and how the following fundamental components and procedures are defined:

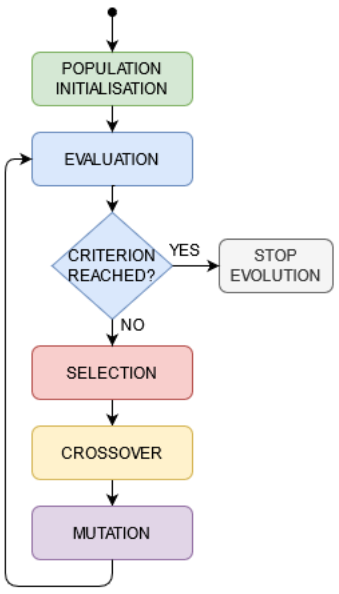


Figure 2.4 *A flowchart describing the typical steps of an evolutionary algorithm.*

- **Representation**. EAs are population-based search algorithms where individuals in a population yield candidate solutions to a given problem. The initial step of designing an EA is to define a way to meaningfully represent these individuals in a way that allows the EA to mutate them or splice them with other individuals. This representation is known as an individual's *genotype* and could, for instance, be a list of integers or floating point numbers. An individual's genotype representation must be translatable into the kind of object that could solve the problem. The type of object depends on the problem, but could for example be executable computer programs, neural networks, or time schedules. The actual solution object that an individual is translated into is referred to as that individual's *phenotype*. Finding a good representation means finding a suitable genotype for the problem, and finding a translation that means that changes in the genotype leads to meaningful changes in the phenotype.
- **Evaluation function**. The evaluation function, normally referred to as the *fitness function*, is a function or procedure that assigns a quality measure to the genotypes. This quality measure, or *fitness*, represents an individual's ability to solve the problem at hand. Since EAs are commonly used to solve optimization problems, they usually involve minimizing or maximizing the fitness function.
- **Population**. The EA's population is the full set of individuals, or candidate solutions that the EA is currently working on. The population size is usually more or less constant throughout

the evolutionary search, which means that only a subset of individuals in each generation will be propagated to the next, which in turn leads to competition between the individuals [22, p. 31]. In some applications, specific population measures may be used, such as *diversity* measures that distinguish distinctive solutions. Before evolution may start, the *initialization* of the first generation's individuals must take place. The initialization is usually based on creating randomly composed genotypes.

- **Selection mechanisms**. There exist two types of selection mechanisms for EAs; *parent selection* and *survivor selection*. These are the processes responsible for improving the population's overall quality. The parent selection mechanism picks strong individuals from the current generation to become parents that produce offspring for the next generation. The selection procedure is usually probabilistic but based on fitness in some way. However, to prevent the evolutionary search from becoming too greedy and stranding in local optima, weak individuals are typically also given a chance to become parents. The survivor selection mechanism brings a few individuals forward unchanged from one generation to the next. The process takes place after new individuals have been created, and is usually performed in a deterministic manner by selecting the top-ranking individuals from the previous generation [22, p. 33].
- **Variation operators**. While the selection mechanisms effectively reduce the population's diversity, variation operators are responsible for creating diversity by generating new individuals [24]. There are two types of variation operators; *recombination* and *mutation*. Some EAs apply both operators, while others use only one of them. Exactly how these operators work will depend on the genotype representation. The recombination operation, also called *crossover*, combines genetic aspects, usually from two or more parent individuals, into one or more offspring. The operation is probabilistic, meaning that which parts of each parent get selected and how they are used depend on random processes. The parents that get selected for mating presumably possess some desirable traits reflected by the fitness values, and their offspring will hopefully carry a combination of these traits, which results in further improved behaviour. Of course, there is also a chance that the new genetic composition is disadvantageous and decreases performance. The mutation operator is applied only to a single individual at a time and alters that individual's genotype. The choice of both individual and type of alteration is usually random. Together with the selection mechanisms, the variation operators *exploit* a population's existing resources and *explore* the search space by creating new solutions.
- **Termination condition**. Normally, the evolution process is set to terminate once some condition is reached. If there is a known optimal fitness level for the given problem, the condition would be that an individual reaches this value. However, since EAs are stochastic, arriving at this optimum is unlikely. Therefore it is more common to define an acceptable fitness threshold that, once is reached, will cause the evolution to terminate. Other common termination conditions could be that the performance of the population stagnates (meaning no improvement has occurred over some number of generations), that a maximum number of generations are completed, or that the population diversity drops below a minimum.

While there exist many types of EAs, this report focuses on GAs. For further reading on EC and EAs, we recommend looking into the introductory book by Eiben and Smith [22].

2.3 Neuroevolution of augmenting topologies

Neuroevolution of augmenting topologies (NEAT) is a GA used to evolve artificial neural networks. The method was developed by Kenneth O. Stanley in 2001 [25] and laid the groundwork for many neuroevolution methods used today. While predecessing neuroevolution methods focus on optimizing the parameters of fixed ANN structures, NEAT evolves both network topologies and optimizes their parameters, hence the name. This section addresses the NEAT method as described in the original paper, while subsequent work offers various extensions [26].

NEAT seeks to answer the following questions:

1. Is there a genetic representation that allows different neural network topologies to crossover in a meaningful way?
2. How can one protect topological innovations that have yet to be optimized so that they do not disappear from the population prematurely?
3. How can the size of topologies be minimized throughout evolution without the need for a fitness function that measures structure complexity?

The key ideas behind NEAT are as follows:

- **Genetic encoding with historical markings.** In GAs, the individuals have a genotype, which is the encoding that undergoes mutation and crossover, and a phenotype, which results from transforming a genotype into some object that solves the problem at hand, like a neural network. When genotypes become complex, recombining individuals in a meaningful way may prove challenging. NEAT combats this by adopting historical markings to keep track of the origin of all genes, allowing recombination without the need for expensive topological analysis. Whenever a new gene appears as a result of structural mutation, a unique id, or *innovation number*, is assigned to that gene. These innovation numbers are the historical markings used to match the genetic material between any individuals in a possibly topologically diverse population.
- **Protect and promote structural innovation.** NEAT aims to protect structural innovation by dividing the population into species based on the genes' historical markings. Only individuals within the same species may compete and mate with each other. To preserve the diversity of solutions, individuals get their fitness adjusted according to their species' size, such that sparsely populated species holding innovative solutions reproduce more than densely populated species with well-established solutions.
- **Genome initialization with minimal structure.** Completely random initialization of genomes is not uncommon among GAs. However, this could result in a base of solutions with large amounts of complexity that might not directly improve with crossover and mutation. The size of topologies may be controlled with a fitness component measuring their complexity to prevent them from bloating. However, NEAT instead creates initial network structures that are as simple as possible and incrementally expands the solutions through genetic operations.

2.3.1 The NEAT algorithm

Being a genetic algorithm, NEAT follows the standard process of initializing a population of solutions that undergoes selection pressure, genetic crossover and mutation. In addition, NEAT incorporates the three key ideas presented above. An overview of the NEAT algorithm is presented step by step in algorithm 1 below.

Algorithm 1: An overview of each step of the NEAT algorithm.

```
(1) initialize a population of $N_{pop}$ random individuals with minimal structures;
(2) while the termination criterion is not satisfied do
    (3) evaluate all genomes in the population;
    (4) divide the population into species;
    (5) reproduce genomes and replace the population with offspring;
```

Note the following remarks regarding the steps of algorithm 1:

1. The size of the minimal structure is predefined (i.e. the number of hidden layers and nodes).
2. The termination criterion could, for example, be N_{gen} generations or a fitness threshold.
3. Evaluating a genome means calculating its fitness. First, a genotype-to-phenotype conversion is performed, meaning the genome is transformed into a neural network. A set of data is propagated through the neural network, and the genome's fitness value is the result of a cost function applied to the set of output values (and possibly some target values)[1].
4. At the first generation, only one species exists, containing all individuals in the population.
5. Reproduction involves several steps:
   - Adjust the fitness of each genome in each species (explicit fitness sharing).
   - Eliminate the lowest-performing members from the population (predefined threshold).
   - Every species receives a number of offspring in proportion to its members' sum of adjusted fitnesses. For each species, repeat until this number is reached:
     1. Select two parents for crossover[2].
     2. Produce a child that inherits genes from the parents.
     3. With some probability, perform a structural mutation to the child.
     4. With some probability, mutate one of the child's connections.

2.3.2 Genetic encoding

NEAT encodes individuals such that the genes from two parents may be conveniently lined up when producing new offspring. The encoding is a linear representation of a neural network's connectivity, consisting of a node gene set and a connection gene set. The node genes represent the set of inputs, hidden nodes, bias nodes and outputs that may be connected. Each connection gene specifies which two nodes are connected, the weight of this connection, whether the connection is enabled, and an innovation number. This innovation number is the historical marking used to find corresponding genes when performing a crossover between two individuals, described in more detail in section 2.3.4.

[1]The choice of cost function, or objective function, depends entirely on the problem at hand. For example, in the case of classification, the cost function could be cross-entropy, or in the case of regression, it could be mean squared error.

[2]There exists various ways to select parents for reproduction: Examples are random selection, fitness proportionate selection, rank selection, tournament selection, etc. There is no default parent selection method specified in the NEAT paper.

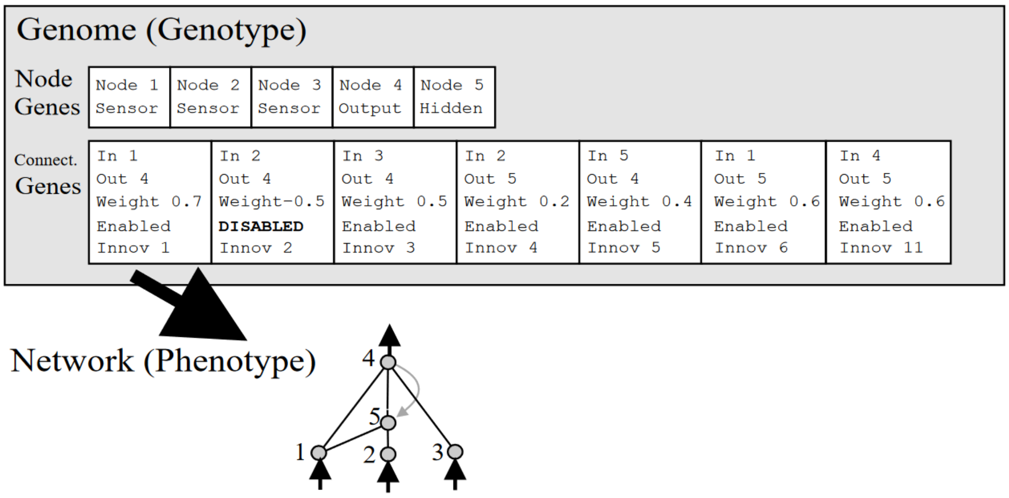

Genome (Genotype)
Node Genes
Node 1 Sensor
Node 2 Sensor
Node 3 Sensor
Node 4 Output
Node 5 Hidden
Connect. Genes
In 1 Out 4 Weight 0.7 Enabled Innov 1
In 2 Out 4 Weight-0.5 DISABLED Innov 2
In 3 Out 4 Weight 0.5 Enabled Innov 3
In 2 Out 5 Weight 0.2 Enabled Innov 4
In 5 Out 4 Weight 0.4 Enabled Innov 5
In 1 Out 5 Weight 0.6 Enabled Innov 6
In 4 Out 5 Weight 0.6 Enabled Innov 11
Network (Phenotype)
4
5
1
2
3

Figure 2.5 *A genotype to phenotype mapping example. The genotype (top) produces the depicted phenotype (bottom). Source: [25].*

2.3.3 Mutation

There are two types of NEAT mutations; structural mutations, which change a neural network's topology, and connection weight mutations. Each structural mutation adds a gene to the genome, making it expand, as shown in the figure above. Adding a connection by mutation adds a connection with random weight connecting two previously unconnected nodes. Adding a node by mutation disables an existing connection and places a new node with two new connections where it used to be. The new connection going to the new node gets a weight of 1.0, while the new connection going out gets the same weight as the old connection. This way of adding new nodes protects the genome from decreasing its performance when mutating. New genes receive an innovation number. The innovation number is global and gets incremented with each new gene, thus tracking gene appearances' chronology.

2.3.4 Crossover

The historical markings allow genomes of different structures and sizes to mate in a sensible way. During crossover, the parents' matching genes are lined up; that is, the genes that share innovation numbers. The genes that do not match are categorised as either disjoint or excess genes, depending on whether their innovation number is lower or higher than the highest common gene. When producing new offspring, the matching genes are chosen randomly from the parents, while disjoint and excess genes are selected from the fittest parent.

2.3.5 Maintaining diversity through speciation

NEAT protects topological innovation by dividing the population into species and letting individuals compete within their species. The historical markings between a pair of genomes are used to

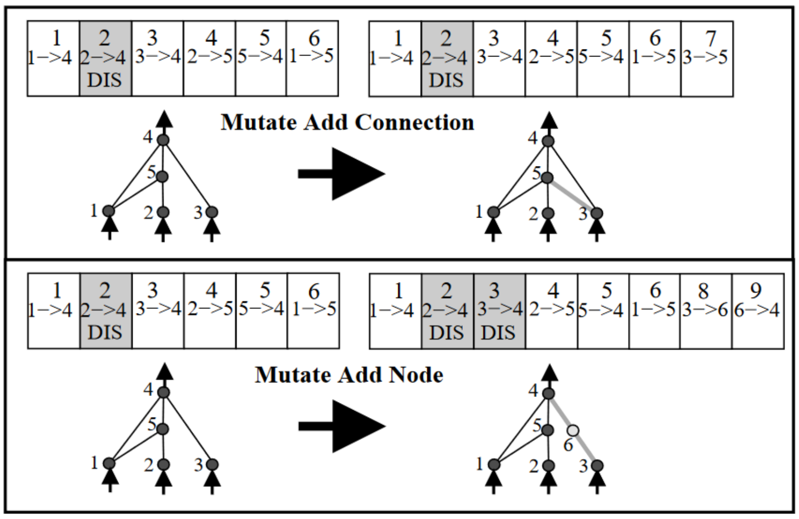


Figure 2.6 *The two types of structural mutation in NEAT. Both types, adding a connection and adding a node, are illustrated with the connection genes of a network shown above their phenotypes. The disable flag, denoted DIS, indicates that the connection is disabled. Source: [25].*

measure their compatibility, which determines whether they belong to the same species. Largely disjoint genomes share little evolutionary history and are thus less compatible. The compatibility measure, δ, is a linear combination of the number of excess E and disjoint D genes and the average weight differences of matching genes W.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 W \tag{2.5}$$

N is the number of genes in the larger genome, and the coefficients c_1, c_2 and c_3 control the importance of the three factors. In each generation, genomes are placed into species using a compatibility threshold. Each species is represented by a random genome from the species of the previous generation. A given genome in the current generation is placed in the first species with which it is compatible. Further, NEAT uses explicit fitness sharing to preserve diversity by placing less emphasis on solutions in dense species. Genomes in the same species share the fitness of their niche, a function that is necessary for speciated evolution to work [25]. Every individual gets their fitness f_i' adjusted according to its distance δ from every other individual j in the population.

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{2.6}$$

The sharing function sh yields 0 when the distance $\delta(i,j)$ is above a threshold δ_t, and 1 otherwise. Thus, f_i' is equivalent to dividing the fitness of genome i by the number of individuals in its species, which results in good solutions in densely populated regions having lower fitness values than comparably good solutions in less populated regions.

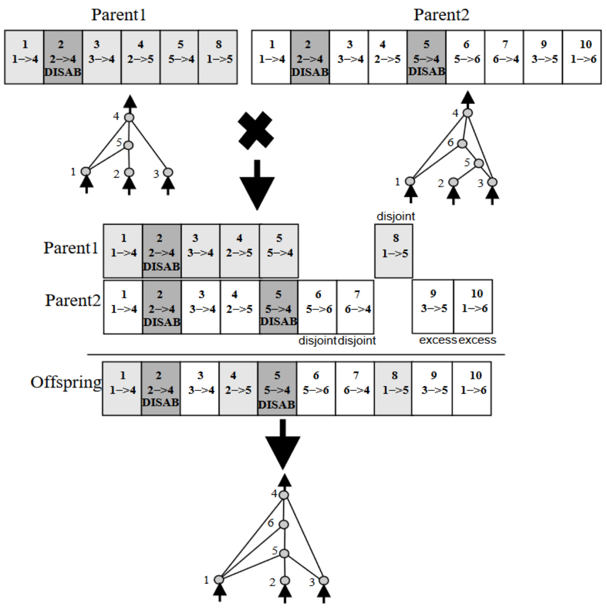


Figure 2.7 *Using innovation numbers to line up the genes of genomes yielding different network topologies. During crossover, matching genes are inherited randomly, whereas disjoint genes and excess genes are inherited from the more fit parent. In this example, equal fitnesses are assumed, so the disjoint and excess genes are also inherited randomly. Source: [25].*

2.4 Software virtualization and containerization

Computer programs are traditionally run inside a host operating system on a physical machine. However, there are drawbacks to this: if a program is dependent on a particular library, then this library must be installed and shared with every other program running on the operating system. If another program is dependent on a conflicting version of that library, then it might become impossible to run. If a program crashes, errors might propagate through the host system, in the worst case causing it to crash entirely.

Virtual machines are a method used to handle these problems, and the concepts underpinning their operation can trace their origins back to the late 1950s. However, virtual machines in the modern sense only became popular in the late 1990s and early 2000s, at the time of the Disco research project at Stanford University, and its commercial spin-off VMWare [27].

A virtual machine does not access the hardware of the physical machine it is running on directly. Instead, it has access to emulated hardware. Software running inside a virtual machine should ideally not be able to tell that it's not running on an actual physical machine. The virtualization software in which the virtual machine runs provides the abstraction layer between the virtual machine and the physical hardware, making sure the virtual machine can not access memory or hard drive space outside of what has been allocated to the virtual machine. This virtualization software could either run as software in an operating system or be an operating system on its own. It is not uncommon to run several virtual machines at the same time on the same physical machine.

Since the virtualization software works to ensure that processes inside a virtual machine can't affect anything outside it, a program that crashes inside a virtual machine will usually only crash the virtual machine at worst, while the host machine and any other virtual machine that runs on it should remain unaffected. Each virtual machine is also given its own virtual hard drive, on which it must install its own operating system and software. This means that two virtual machines can run different library versions and even entirely different operating systems, all while running on the same physical machine.

However, virtual machines come with drawbacks: since each virtual machine needs to provide its own entire operating system, they take up a lot of disk space and are computationally expensive. Initializing a virtual machine can also take a long time, as it needs to go through almost all the same boot procedures as a physical machine.

Instead of abstracting away the hardware of a physical machine, it is also possible to provide an abstraction layer that uses the computer's existing operating system. Programs are still given restricted access to the host computer's hardware, such as memory and hard drives, but they are able to use the operating system kernel of the host machine. This paradigm is known as *containerization* [28].

Ever since Docker was released by Docker, Inc. in 2013, software containerization has been a rising trend. Docker is the most common containerization framework and provides a standardized environment in which to run software across a range of different platforms. Docker containers are also easily distributed, making it simple to run a program quickly across a range of different host machines.

Docker containers are "lightweight, standalone, executable packages"[3]. They contain every library and file a program needs to run, depending only on the host operating system's kernel. They are portable, and there is no need to install any software or libraries on the host machines they run on apart from the Docker engine itself. This makes it very easy to deploy Docker containers to virtually any machine, makes it easier to upgrade applications by replacing them with newer containers, and makes it possible for developers to run their programs in environments that are almost exactly like a live server [29].

However, although running software in containers removes the need to install software and dependencies on a host machine, starting a large set of containers on a set of different hosts, making sure they can communicate with each other, and restarting them if they fail is still a complex task. In order to handle this task automatically, *container orchestration* tools are used.

Container orchestration tools are tools that manage containers running in a cluster automatically. This involves starting and stopping containers across several computers, but also monitoring their health, and restarting them if they crash. Orchestration tools also need to manage the network connections between containers and allow outside access to containers when required [30].

Kubernetes is the most popular container orchestration tool and is more or less a de facto standard [31]. It was originally released by Google but is now maintained by the Cloud Native Computing Foundation. Several cloud computing providers offer services based on Kubernetes, such as Azure

[3]`https://www.docker.com/resources/what-container`, accessed 2021-04-20.

Kubernetes Service[4], Google Kubernetes Engine[5], and Amazon Elastic Kubernetes Service[6].

A containerized application running in Kubernetes consists of several parts, the most important of which are pods and services. Pods are conceptually the smallest units in a Kubernetes service and consist of a set of containers that are always run together. Services define the interfaces that other programs can use to connect to Kubernetes pods by specifying which ports the Kubernetes host should forward to a Kubernetes pod. A service can provide an interface to other pods, but also provide an interface for external programs or end users to connect to.

Pods, services, and the other Kubernetes building blocks are expressed through the use of YAML[7] files [32]. These are configuration files that specify options for the different building blocks, for instance, which containers should be part of a pod or which ports should be forwarded in a service. Each YAML file normally describes only a single building block, and they all need to be submitted individually to build a complete Kubernetes application.

In order to simplify this process, it is common to specify Kubernetes applications using Helm charts[8]. Helm charts consist of a set of YAML files that together describe a full Kubernetes application. These can then all be installed and uninstalled from a Kubernetes cluster in a single operation.

2.5 Related work

Early attempts at constructing intelligent agents for the air combat domain not based on simple rule-based systems date back to the nineties, if not earlier. Some of this research includes supervised manoeuvre classifier models based on neural networks [33, 34], and agents implemented using the Soar cognitive architecture [35, 36]. The Soar agents incorporate knowledge based on interviews with experts in flight tactics and analysis of the tactical domain. Artificial intelligence research has come a long way during the last thirty years, and improved methods and techniques have emerged. With these improvements, the use of AI for training purposes has increased, and several research contributions can be found in military, industry and medical domains.

This related work section briefly summarizes some recent research contributions and other work on using ML methods to construct behaviour models for use in the air combat domain. The research methods will not be described in detail; thus, we encourage readers interested in learning more to take a closer look at the provided sources. To our knowledge, there is little or no research focusing specifically on the application of IL techniques in this field, which this report focuses on. However, a great deal of the reviewed work applies RL, some of which use reward shaping, which shares similarities with IL if used extensively. We categorize the reviewed work based on their choice of ML methods or applications.

[4]`https://azure.microsoft.com/en-us/services/kubernetes-service/`, accessed 2021-04-20.

[5]`https://cloud.google.com/kubernetes-engine/`, accessed 2021-04-20.

[6]`https://aws.amazon.com/eks/`, accessd 2021-04-20.

[7]YAML was originally short for *Yet Another Markup Language*, but has since been changed to the recursive acronym *YAML Ain't Markup Language*.

[8]`https://helm.sh/docs/topics/charts/`, accessed 2021-05-12

Evolutionary algorithms and fuzzy systems

Grammatical evolution [37], which is an evolutionary algorithm designed to evolve computer programs, has been used to generate behaviour trees for CGFs in the air combat domain [38]. A behaviour tree is a common way to represent AI behaviours, using control flow nodes and execution nodes [39]. In this study, subject matter experts provided domain knowledge encoded into behaviour trees in a knowledge base. The behaviour trees were converted to bit-strings using a pre-defined grammar and got improved through variation operators in an evolutionary process guided by reward-based feedback from the simulation environment. This framework was proven to work quite well using simple one vs one beyond visual range scenarios.

Another line of research applied a combination of genetic algorithms and fuzzy logic to air combat control problems involving autonomous unmanned combat aerial vehicles (UCAVs) [40, 41, 42]. Fuzzy logic is a form of many-valued logic that handles degrees of truth, as opposed to Boolean logic, in which truth values are either zero or one [16]. In their work, the researchers introduced the method of genetic fuzzy trees, which uses a genetic algorithm to train multiple small fuzzy inference systems that combine to solve complex tasks at lower computational costs than a single genetic fuzzy system would [43]. The genetic fuzzy trees learned to successfully control UCAVs in scenarios involving enemy threats, such as air interceptors, surface-to-air systems and electronic warfare systems.

Evolutionary algorithms have also been applied to problems within other simulated military domains, such as for learning cooperative movement tactics of ground troops [44, 45]. Although not focusing on the air combat domain, this work is closely related to ours since they aim to learn agent behaviour by providing demonstrations in an IL configuration.

Dynamic scripting

Dynamic scripting is an RL method that has been studied extensively in this field. The method uses an adaptive rulebase for the generation of behaviours in the form of scripts [46]. Each rule in the rulebase has a probability of being included in the behaviour scripts, and a weight-update function changes these probabilities using feedback from the environment.

Dynamic scripting has proven to be an efficient method for learning behaviours targeted towards air combat training simulations. It has been used to study reward functions that consider the expected outcome of agents' actions [47], the transfer of learned behaviours between distinct scenarios [48], and creating behaviour models in the form of finite-state machines for increased adaptivity [49]. Dynamic scripting has even been used for the generation of BTNs with NGTS as the simulation environment [50].

Deep reinforcement learning

Due to its recent success, deep reinforcement learning (DRL), which is a combination between RL and deep learning [51], has become one of the most popular classes of algorithms used for learning agent behaviour today. Researchers have successfully used DRL algorithms to learn complex agent behaviour for various tasks and games such as Go [1] and StarCraft II [2]. These achievements have likely contributed to an increasing interest in applying DRL to the air combat domain.

Among recent activities, [52] has reviewed open-source DRL frameworks that may be applicable in learning CGF behaviours intended to support the training of fighter pilots. RAND Corporation explored different DRL methods for learning to plan simple operations involving suppression of enemy air defences, and one of these methods proved successful [53].

Others have focused on learning to manoeuvre unmanned aerial vehicles (UAVs) [54, 55]. Furthermore, [56] demonstrated the effectiveness of applying reward shaping, a technique where supplemental rewards are provided to make a problem easier to learn [57], to DRL for air combat using a simple within visual range one vs one scenario. Pope et al. applied a hierarchical DRL method to train an agent to control a fighter jet engaging in dogfight scenarios within visual range [58]. This agent came second in a tournament held by the Defense Advanced Research Projects Agency (DARPA), further described below.

Benchmarking intelligent air combat behaviour

As can be seen above, there are many approaches to learning behaviour models for the air combat domain. However, as of writing this report, there are no standardized procedures for training and testing the performance of CGF behaviour models that researchers can easily employ. A couple of projects have identified the need for such procedures and have contributed to their emergence. One of these is a testbed developed by Air Force Research Laboratory called the Agent Generation & Evaluation Network Testbed (AGENT) [59]. The AGENT testbed aims to provide developers of fighter pilot agents with the ability to generate large amounts of tactical data for ML of such agents' capabilities. The testbed employs enemy CGF simulated by NGTS, whose behaviours are implemented as BTNs. The developers may exercise their agents against these GCFs in scenarios defined in a shared scenario library.

Another line of work is the Air Combat Evolution (ACE) programme of DARPA. The ACE programme seeks to gain fighter pilots' trust in combat autonomy by exploring human-machine collaborative dogfighting [60]. As part of this programme, DARPA carried out a benchmark of AI-controlled fighters in a simulated environment called the AlphaDog trials. Eight companies participated in the AlphaDog Trials, in which agents faced one another in one vs one dogfight. Although dogfighting might not be the most relevant scenario used when training pilots today, the tournament created an arena where current approaches to AI could take on a complex problem relevant to military applications. The winner of the trials was an agent developed by Heron Systems, which employed a neural network-based behaviour trained using DRL. This winning agent outperformed the other agents in addition to a human fighter pilot [61].

3 Methods

The following chapter argues for the choice of simulation application and machine learning methods used to approach the imitation learning problem.

3.1 Simulation environment and behaviour representation

We use NGTS because it is designed especially for the air combat domain, for its DIS support and its built-in implementation of BTNs.

We would have preferred to represent behaviour using neural networks. However, this was not possible when using NGTS because we lacked the API access necessary to control entities within NGTS from an external program. Instead, we chose to use BTNs for representing behaviours, which allowed us to use NGTS' built-in BTN support. BTNs offer, among other things, the following assets:

1. **Explainability**: BTNs are made of by human-readable components, which result in highly explainable behaviour models.
2. **Complexity**: Although BTNs are simple, human-readable structures, the number of possible node combinations is vast, making it possible to express very complex behaviours using BTNs.

3.2 Evolving BTNs with NEAT

NEAT was the algorithm of choice for learning behaviour models for the air combat domain. While NEAT was originally intended to create ANNs, the algorithm can also apply to other tasks. ANNs and BTNs are quite different because one represents some mathematical function, while the other defines a behaviour model composed of action nodes and condition nodes. However, in essence, both ANNs and BTNs are networks of nodes (vertices) and connections (edges). Therefore, NEAT's ability to evolve network topology and parameters might generate BTNs. We used the NEAT-Python implementation [62] as a basis and added some extensions to make it able to evolve BTNs in place of ANNs. These extensions are described in section 3.2.1. For clarity throughout the rest of this report, we name the modified NEAT algorithm *BTN-NEAT*. Further, for simplicity, the terms nodes and connections are used from here on when referring to BTNs rather than vertices and edges.

Although BTNs may be hierarchical, in that a node can be used to call a different BTN, evolving hierarchical BTNs is dramatically more complex than evolving simpler, self-contained BTNs. Therefore, for simplicity, BTN-NEAT evolves self-contained BTNs and not hierarchical BTNs. In NGTS, BTNs consist of two types of nodes: commands (action nodes) and triggers (conditional nodes). Recall from section 2.1.2 that BTNs employed by NGTS do not strictly follow the BTN definition and allow connecting commands to other commands and triggers to other triggers. When traversing a BTN, if it is possible to continue into more than one node at any point, the possible

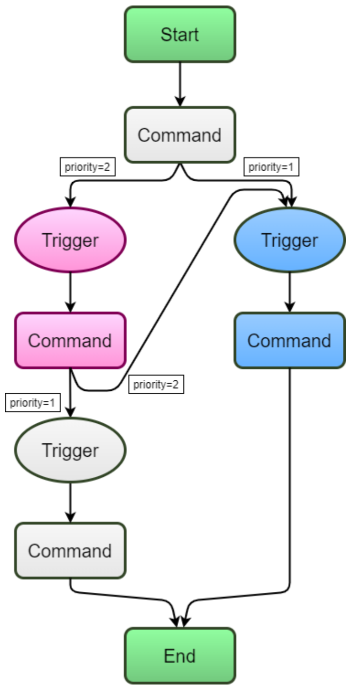


Figure 3.1 *A valid behaviour transition network with adjacent trigger nodes. In this BTN example, two adjacent pairs are formed, one by the pink and blue trigger nodes and the other by the grey and blue trigger nodes. The priorities of the connections determine which trigger is evaluated first. Trigger nodes are depicted as ellipses, and command nodes as rectangles.*

nodes are called adjacent to each other. Adjacency is not necessarily associative: in figure 3.1, the blue trigger node is adjacent to both the pink trigger node and the lower grey trigger node, but the pink and grey trigger nodes are not adjacent to each other.

Three rules define the validity of BTNs:

Rule 1. Command and triggers may not be adjacent.
Rule 2. Commands may not be adjacent.
Rule 3. If two or more triggers are adjacent, they must have different priorities.

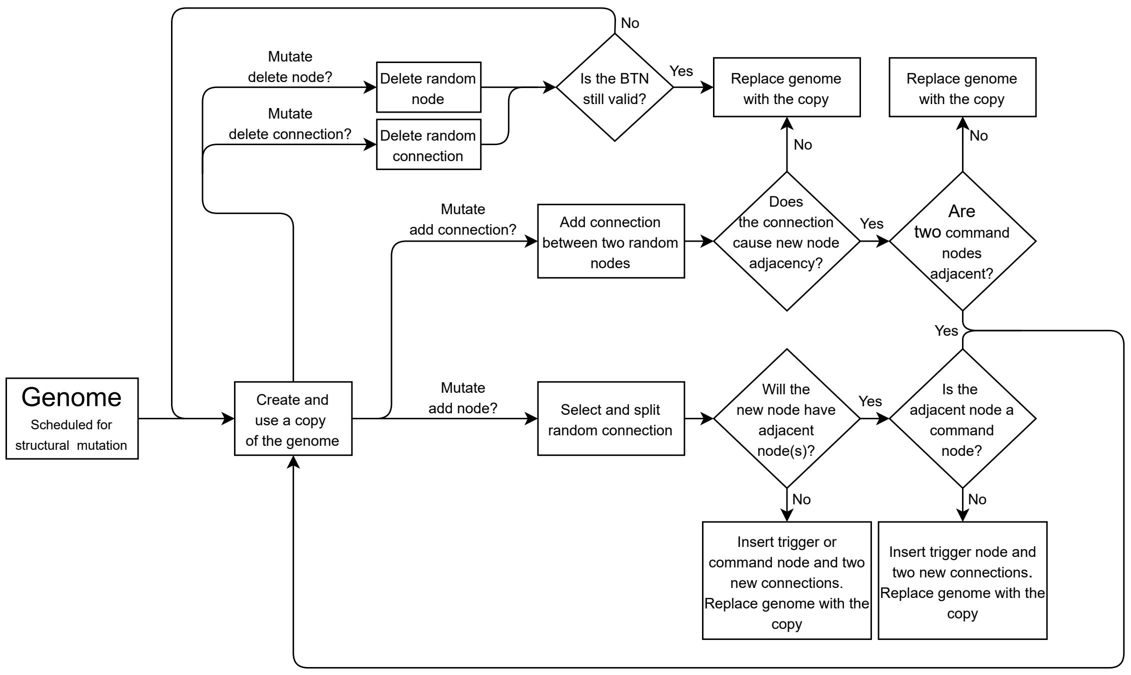
These three rules ensure that no ambiguity occurs within a BTN. In other words, a node's output can only connect to more than one node if these are all triggers. Yet, if two or more triggers are adjacent, the priorities of the connections leading to them determine the order of their evaluation. Figure 3.1 shows how connections leading to adjacent trigger nodes have different priorities. Since it is only possible to execute valid BTNs, the above rules must be satisfied during the crossover and mutation of genomes. While mutation of network parameters remains as described in section 2.3.3, structural mutation and the crossover operation require some modification.

3.2.1 Adapting NEAT to evolve BTNs

In order to make NEAT evolve BTNs in place of the ANNs for which it was originally intended, we slightly modified two of the algorithm's procedures: the structural mutation procedure and the crossover procedure. These modified procedures are described below. The genes that code for connections between nodes are kept as when evolving neural networks, and the weights of the connections are used to determine the priority when two trigger nodes are adjacent.

3.2.1.1 Modified structural mutation

For the most part, structural mutations of BTN genomes remain as described in section 2.3.3, except for the following consideration. If a structural mutation is performed on a genome, for example, adding a node, the operation must be carried out such that the genome remains valid. A structural mutation that violates any of the three rules may not be performed. This is solved by creating a copy of the genome, structurally mutating it and then checking its validity. If valid, the original genome is replaced by the mutated copy. If the resulting genome is not valid, another structural mutation is attempted. This process is only repeated for a predefined number of attempts to avoid infinite loops. The new structural mutation operation is shown in figure 3.2. Note that structural mutations may also involve deleting nodes or connections, although these operations are not described in the original NEAT paper.

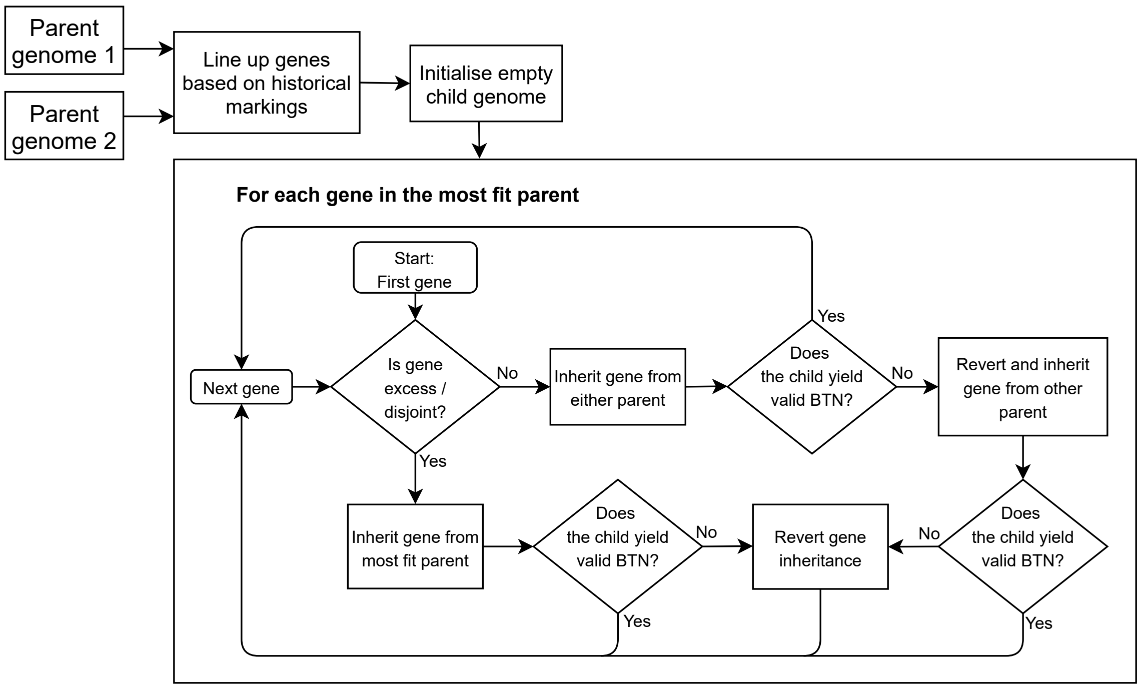


Genome
Scheduled for structural mutation
Create and use a copy of the genome
Mutate delete node?
Mutate delete connection?
Delete random node
Delete random connection
Is the BTN still valid?
No
Yes
Replace genome with the copy
Replace genome with the copy
Mutate add connection?
Add connection between two random nodes
Does the connection cause new node adjacency?
Yes
No
Are two command nodes adjacent?
No
Yes
Mutate add node?
Select and split random connection
Will the new node have adjacent node(s)?
Yes
No
Is the adjacent node a command node?
No
Insert trigger or command node and two new connections. Replace genome with the copy
Insert trigger node and two new connections. Replace genome with the copy

Figure 3.2 *A flowchart depicting the modified structural mutation procedure, which complies with the defined rules and allows BTN-NEAT to evolve BTNs in place of ANNs. The procedure affects a genome that has already been scheduled for a structural mutation. The mutation operator (add node, add connection, delete node or delete connection) gets selected based on the mutation probabilities set by the user.*

3.2.1.2 Modified crossover

Like the modified structural mutation, the crossover procedure between BTN genomes remains mostly as described in section 2.3.4, except for the following consideration. ANN genomes have only one type of node, which means that inheriting any gene from any parent genome during crossover is unproblematic, even though the new configuration may reduce its fitness. On the other hand, BTNs consist of two types of nodes; commands and triggers. Thus, a child BTN genome may not inherit genes that break any of the adjacency rules. When a child BTN genome is produced, only the parents' genetic combinations that yield a valid BTN can be inherited. The new crossover operation is shown in figure 3.3.



Parent genome 1
Parent genome 2
Line up genes based on historical markings
Initialise empty child genome
For each gene in the most fit parent
Start: First gene
Next gene
Is gene excess / disjoint?
No
Yes
Inherit gene from either parent
Does the child yield valid BTN?
Yes
No
Revert and inherit gene from other parent
Inherit gene from most fit parent
Does the child yield valid BTN?
No
Yes
Revert gene inheritance
Does the child yield valid BTN?
No
Yes

Figure 3.3 *A flowchart depicting the modified crossover procedure, which complies with the defined rules and allows BTN-NEAT to evolve BTNs in place of ANNs.*

Note that the order of the historical markings will also affect which parent genes are added to the child genome.

3.2.2 Fitness functions

Like in other genetic algorithms, BTN-NEAT individuals compete against each other, and the strongest individuals of the population are most likely to pass on their genetic composition. The strength of an individual is measured using a fitness value, which indicates how well the individual can solve the problem at hand. The fitness value of each individual is calculated using a fitness function.

We developed primarily one fitness function for the imitation learning task, termed the *trajectory* *fitness function*. In addition, we experimented with an alternative fitness function, a *semantic fitness function*, which aims to offer increased robustness and generalization to the learned BTN solutions.

3.2.2.1 Trajectory fitness function

The trajectory fitness function is used to calculate the difference between a provided trajectory generated by some behaviour and the demonstration trajectory we wish to imitate. It involves adding up the Euclidean distances between these two trajectories at every logged data point, in addition to two terms that punish the behaviour for using the radar and missiles in a different way compared to the provided demonstration. The optimization consists of minimizing the following fitness function:

$$f = d_{\text{trajectory}} + p_{\text{radar}} + p_{\text{missile}} \tag{3.1}$$

The trajectory component is defined as the sum of distances at $t_0, ..., t_k$:

$$d_{\text{trajectory}} = \sum_k d_k e^{-c_e t_k} \tag{3.2}$$

where

$$d_k = \sqrt{(x_{t_k} - \hat{x}_{t_k})^2 + (y_{t_k} - \hat{y}_{t_k})^2 + (z_{t_k} - \hat{z}_{t_k})^2} \tag{3.3}$$

and

$$c_e = \frac{r \ln 2}{\lambda}$$

where

$x_{t_k}, y_{t_k}, z_{t_k}$ = the position of the trajectory being evaluated at time t_k
$\hat{x}_{t_k}, \hat{y}_{t_k}, \hat{z}_{t_k}$ = the position of the demonstration trajectory at time t_k
r = simulation time scale factor
λ = exponential decay factor

The $e^{-c_e t_k}$ term in equation 3.2 leads to a data point at time $t + \lambda$ being given half the weight of a data point at time t, and emphasises the importance of correctly imitating the first parts of the demonstrated trajectory, before focusing on subsequent parts. The simulation time scale factor r determines the relationship between the rate of advance in simulation time and wallclock time [63, p. 29]. For example, simulations with a simulation time scale factor of $r = 2$ execute twice as fast as real-time, while simulations with $r = 1$ execute in real-time.

The first parts of a demonstrated trajectory are emphasised because behaviours tend to evolve towards more complexity: in early generations of evolution, behaviours only have a few conditions and commands and are fairly simple, while later generations can become increasingly complex. The early parts of a demonstration will also usually be characterised by simple behaviours, such as flying towards or away from an enemy, while later parts of the demonstration behaviour typically become more complex, such as turning away from the enemy at a given distance. The early parts of the trajectory are therefore emphasised in order to make the BTN-NEAT algorithm learn the overarching, simple parts of some behaviour before adding on complexity later.

The radar punishment component punishes the agent for using its radar in a way that is dissimilar to the demonstrated behaviour and is defined as

$$p_{radar} = \sum_k s_{t_k} \hat{s}_{t_k} c_p \tag{3.4}$$

Here, s_{t_k} is the state of the controlled aircraft's radar at time t_k, $\hat{s}_{t_k}$ is the radar state of the demonstration at time t_k, and c_p is a coefficient weighting the impact of the radar punishment. The radar state is 1 when the radar is on and 0 otherwise.

We also considered implementing a punishment term for firing weapons in an incorrect manner but did not have time to explore this further.

3.2.2.2 *Semantic fitness function*

While the trajectory fitness function described above may be appropriate for the given learning problem, it requires the initial conditions of the scenario to be identical to those of the demonstration. Running even the demonstration behaviour itself using different starting positions for the aircraft would lead to a very low fitness score. As an attempt to overcome this limitation, we did some experimental development of a fitness function that would compare two sets of events instead of comparing the trajectories directly. These events do not describe exactly *what* each entity is doing, as in the case of comparing trajectories, but rather what each action *means*. For this reason, we call it a semantic fitness function.

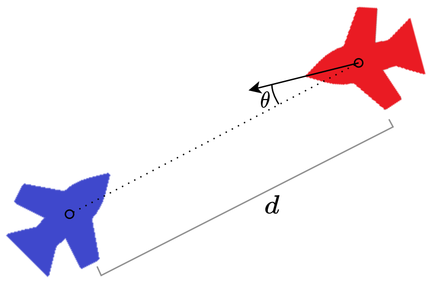


Figure 3.4 *Finding the relative distance and heading between the entities for use with the semantic fitness function.*

These events describe changes in relative heading and distance between the red and blue entities, in addition to changes to their radar and missile states. The possible events are listed in table 3.1.

Event	**Description**
e_1	Facing towards target
e_2	Turning away from target
e_3	Facing away from target
e_4	Turning towards target
e_5	Tracking target with sensor
e_6	Launching missile

Table 3.1 *An overview of the different semantic events that may be registered.*

A semantic sequence takes the following form

$$[(e_1, d_1), (e_2, d_2), (e_3, d_3), \cdots, (e_n, d_n)]$$

where $e_1, \ldots, e_n$ are the events in the semantic sequence, and $d_1, \ldots, d_n$ are the distances between the aircraft when the events occurred. The different semantic events are derived using a set of rules that examines the rate of change of the relative distance and heading between the red and blue aircraft.

To derive the similarity between two event sequences, the event sequences are turned into strings of letters, where each letter represents a type of event, and the Levenshtein string matching distance [64] is used to measure how similar the event strings are. The Levenshtein distance computes the minimum number of edits (insertions, deletions or substitutions) required to change one word into another. The complete semantic fitness function is defined as follows.

$$f_{\text{semantic}} = c_L d_{\text{Levenshtein}} + c_d d_{\text{aircraft}} \tag{3.5}$$

where $d_{\text{Levenshtein}}$ is the normalised Levenshtein distance yielding a value in the range $[0, 1]$, d_{aircraft} is the sum of distances between the aircraft at all the events, and c_L and c_d are coefficients weighting the contribution of the two fitness terms.

4 The machine learning system

Machine learning requires large amounts of data to train models effectively. It would therefore be unfeasible to evaluate generated BTNs by manually running NGTS and extracting data logs to be used for fitness evaluation. Instead, the machine learning system developed in this work consists of a set of tools that automate these processes, presented in this chapter.

4.1 System overview

The overarching system consists of four parts: the BTN evolution, simulation control, the simulation, and the data logger. When evolving behaviours, these four parts are used in a loop, illustrated in figure 4.1. First, BTN Evolution evolves a set of new behaviours for the CGF to employ. Then, the simulation control, which is configured to use a specific scenario, configures the simulated entities to use these behaviours and starts the simulation. The data logger logs the events of the simulation, which the BTN evolution system then uses to calculate the fitness of each generated behaviour. This information is then used to evolve behaviours further.

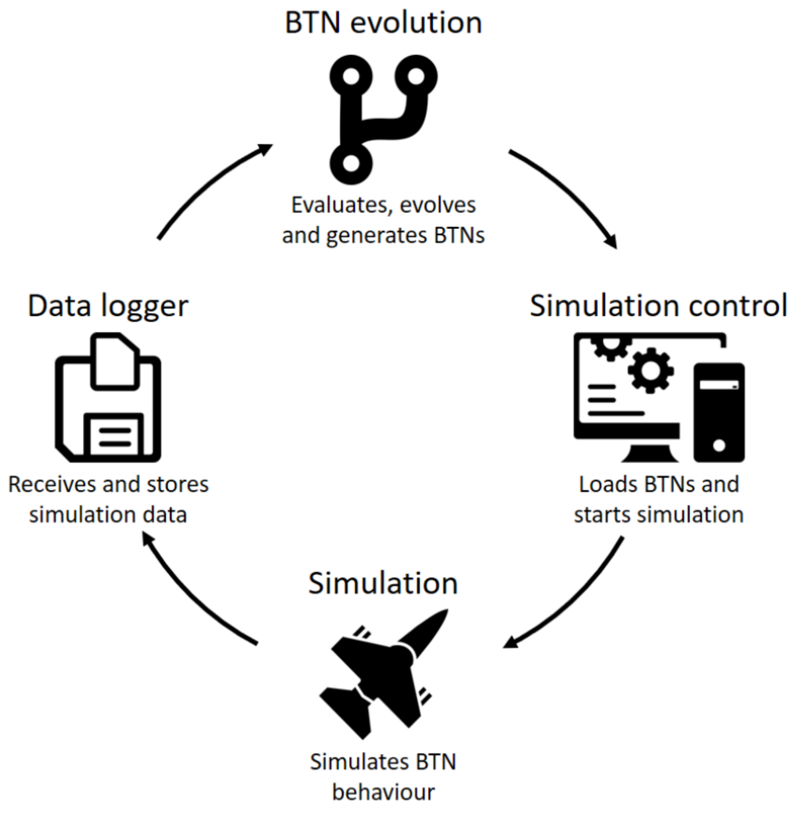

BTN evolution
Evaluates, evolves and generates BTNs
Simulation control
Loads BTNs and starts simulation
Simulation
Simulates BTN behaviour
Data logger
Receives and stores simulation data

Figure 4.1 *The loop that is used to evolve BTNs.*

Figure 4.2 shows how these parts were implemented, as well as the data formats used to exchange information between them. Each of the components are described in more detail in the following sections.

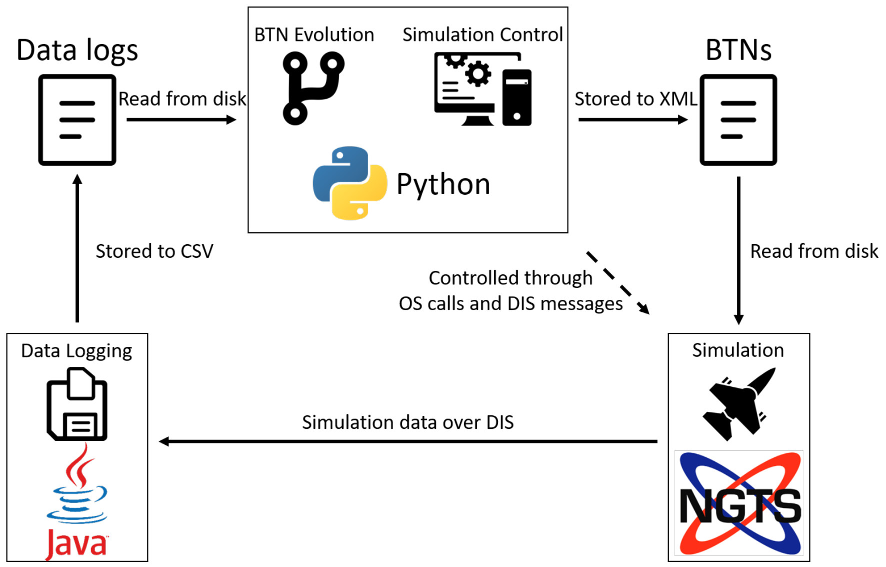


Figure 4.2 *The BTN evolution loop from figure 4.1, with implementation details and data formats shown.*

4.1.1 BTN evolution and simulation control

The BTN evolution and simulation control parts were both implemented as a single Python application. This combined BTN evolution and simulation control program is the driving force behind the machine learning loop. This application is hereafter referred to as the *evolution controller* for simplicity.

BTN evolution is achieved using BTN-NEAT, which is a modified version of the NEAT algorithm, described in more detail in chapter 3. Once a BTN is created, it is saved to disk in an XML format in order for NGTS to be able to read it.

In order to get NGTS to use the new behaviours, the simulation control needs to assign the behaviour to a simulated entity. This is done by automatically generating new scenario files, in which simulated entities and their behaviours are specified. The scenario files are generated based on a scenario template; the initial positions and types of the simulated entities are defined, and only the names and behaviours are modified.

Both scenarios files and behaviours need to be stored in specific folders where NGTS can find them. However, NGTS only discovers new files when restarted. For this reason, the NGTS process itself is controlled using the Python Subprocess library[9]. For each generation, NGTS is restarted once a batch of behaviours and scenarios is generated and placed into the correct directories. Simulation control calls are then sent over DIS in order to load new scenarios into NGTS and start simulations,

[9]https://docs.python.org/3/library/subprocess.html

and stop the simulations after having run for a given amount of time. At the end of each generation, the NGTS process is killed before a new generation is started.

Once the simulation is finished, the BTN evolution reads that generation's data logs from a set of files that the data logger has stored to the hard drive. A fitness function compares the content of the new data logs to the provided demonstration to derive a fitness score for each behaviour of that generation. The fitness functions used are described in section 3.2.2.

NGTS is remote-controlled using protocol data units (PDUs) that are transmitted over the DIS protocol. Standard PDUs are used for starting and stopping the simulation. Saving and loading scenarios, on the other hand, are performed using a custom PDU proprietary to NGTS, based on the SetData PDU.

In order to simplify the control of NGTS, we have written a remote control application that can be called by the evolution controller. This program can create and send the DIS PDUs that are needed to remote control NGTS, as well as listen to DIS traffic to determine when NGTS has finished its startup procedures. The program is written in Java and can be used in two ways: either by providing command-line arguments or by sending commands across a TCP connection.

4.1.2 Data logger

The data logger is a Java program that uses either HLA or DIS to collect data about simulated entities and stores this data in an internal database. When the logger is done recording, this data is written to log files. The log files for a given behaviour are used when calculating a fitness score for that behaviour.

Currently, the logger is capable of logging an entity's usage of sensors, the firing of weapons, and its trajectory. This can be done for either a single entity, a set of entities or all entities in the simulation. Each entity's logs are written to a separate folder, and the sensor usage, weapon fire, and trajectory logs are stored in separate files.

The sensor usage log contains data about which other entities were being tracked by a specific entity's sensors. Whenever an entity's sensor gains or loses a track, a new line is written to the log file. Each line contains two columns: the (simulated) time the track was acquired or lost, and a list of the entities that are currently being tracked. A line is also written when the entity loses all tracks; in this case, the list of entities that are being tracked is empty. An example of what a log file might look like is shown in table 4.1.

The weapon fire log describes when a specific entity fired its weapons. Whenever a weapon is fired, a new line is added to the log. These lines consist of four columns: the (simulated) time at which the weapon was fired, the name of the weapon being fired, the type of weapon being fired, and the name of the target that is being fired upon. An example weapons log is shown in table 4.2.

The trajectory log consists of a set of observations of the entity's physical state. Each observation corresponds to a single row in the log file, and each row is split into six columns. The first column contains the (simulated) time of the observation. The next three columns contain the X-, Y-, and Z-coordinate of the entity's position at that time. The final two columns contain the entity's speed (in m/s) and it's heading (in degrees) at that time. An example trajectory log can be seen in table 4.3.

Time [s]	Targets
7.30	Red_1
10.48	Red_1, Red_2
13.50	Red_2
14.78	
Done	

Table 4.1 *An example sensor usage log. In an actual log file, columns are separated by tab characters. The log shows the aircraft starting to track entity Red_1 at 7.30 s, and starting to track Red_2 at 10.48 s. The track of Red_1 is lost at 13.50 s, and the track of Red_2 is lost at 14.78 s.*

Time [s]	Weapon name	Weapon Type	Target
7.55	AIM-120D_1	2 1 225 1 2 4 0	Red_1
Done			

Table 4.2 *An example weapon usage log. In an actual log file, columns are separated by tab characters. The log shows a weapon with the name AIM-120D_1 being fired at target Red_1 at 7.55 s. The weapon type is expressed as a DIS enumeration [65].*

Time [s]	X-coordinate [m]	Y-coordinate [m]	Z-coordinate [m]	Speed [m/s]	Course [deg]
0.00	373647.31	440144.02	18483.60	278.7	6.7
0.10	373306.53	440104.02	18460.84	278.7	7.2
0.20	372956.55	440062.95	18437.46	278.2	8.4
Done					

Table 4.3 *An example trajectory log. In an actual log file, columns are separated by tab characters.*

The coordinates of the entity are given in local tangent plane coordinates with respect to a configurable point on the Earth's surface. This point is also used as the origin of the coordinate system. The axes of this coordinate system are chosen such that the X axis points due north, the Y axis points due east, and the Z axis points downwards. This coordinate system is illustrated in figure 4.3. The course of an entity is also calculated with respect to this coordinate system.

The logger can be run using either a graphical user interface or from the command line. The main window of the graphical user interface can be seen in figure 4.4. This graphical user interface allows the user to connect to DIS or HLA federations, start and stop recording, set the origin of the coordinate system used in trajectory logs, and write log files concerning specific entities from previous recordings. It is also possible to set the logger to automatically start and stop recording upon receiving start and stop interactions, and to automatically write log files when the recording is stopped. The logger can be configured to automatically write logs for a single entity or all entities with names matching a specified prefix.

The logger's command line mode also allows the user to set most of these options when starting the program. However, it is not possible to manually start and stop recordings and write logs using the command line mode, so the automatic mode is always used. This means that the user is required to

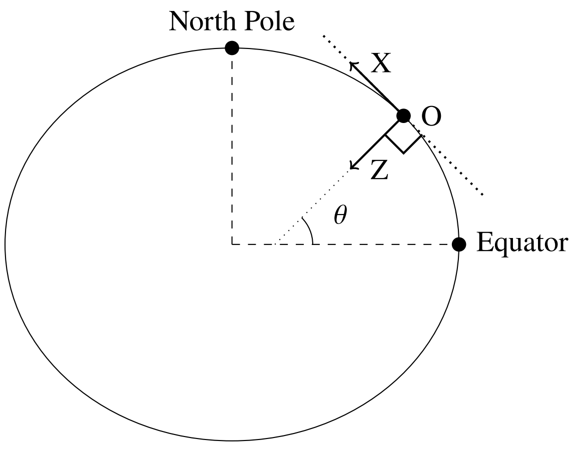


Figure 4.3 *A 2D example demonstrating local tangent plane coordinates. The figure shows a cross-section of Earth (with exaggerated eccentricity). The origin of the local tangent plane coordinate system at point O has latitude θ. The X axis is tangent to the surface of the Earth at point O, and points towards the North Pole. The Z-axis is normal to the surface tangent and points towards the centre of the Earth. (Note that a normal line drawn through O does not generally pass exactly through the centre of the Earth. This is due to the Earth's eccentricity.) The Y-axis is not shown in this 2D example, but on a 3D globe, it is perpendicular to both the Z-axis and X-axis, pointing towards the East.*

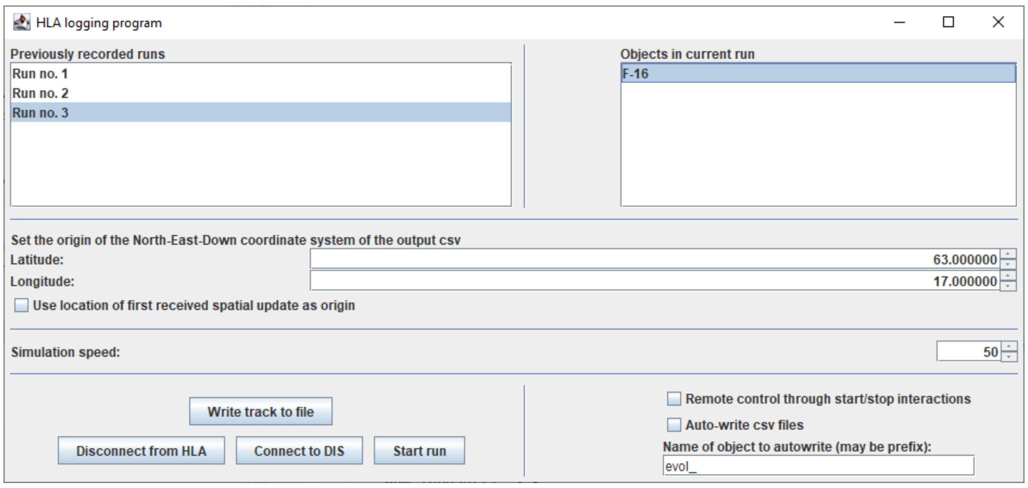
provide a name or prefix to determine which entities' logs should be written to file.

The DIS[8] standard specifies that entity updates should only be sent if the entity's state has changed significantly since the previous update. This is also specified in the Guidance, Rationale, and Interoperability Modalities (GRIM) [66] document that accompanies the very commonly used Real-time Platform Reference Federation Object Model (RPR-FOM) [67]. This means that the logger may receive updates at uneven intervals and that these intervals may also vary between entities.

Since updates for an entity's state are received infrequently, and at uneven intervals, a receiving system needs to extrapolate the previous state of the entity until the next update is received. This process is known as dead reckoning. New updates should only be sent when the difference between the entity's actual state and its dead reckoned state from the previous update exceeds a certain threshold. This threshold is most commonly set to an orientation difference of 3° or a position difference of 1 metre [8][66]. This means that if an entity's trajectory is determined by extrapolating updates, it will have discontinuities each time a new update is received, since this very discontinuity is the reason a new update was sent.

In sum, this means that if the data logger only saves the entity states that are received in updates, the trajectory log will have infrequent data points with uneven intervals. In order to achieve a constant interval between the data points, it is possible to create a trajectory by extrapolating each received update and then sampling this trajectory at even intervals to create the trajectory log. This extrapolated trajectory will, however, contain discontinuities at each received update.

During simulation, a simulation system has no choice but to extrapolate an entity's state based on the last received update. However, since an entity's trajectory log is written only after the logger has



Previously recorded runs
Run no. 1
Run no. 2
Run no. 3

Objects in current run
F-16

Set the origin of the North-East-Down coordinate system of the output csv
Latitude: 63.000000
Longitude: 17.000000
☐ Use location of first received spatial update as origin

Simulation speed: 50

☐ Remote control through start/stop interactions
☐ Auto-write csv files
Name of object to autowrite (may be prefix): evol_

Figure 4.4 *The main window of the graphical user interface of the logger program.*

already finished recording and all updates have been received, it is possible for the logger to also use the next update when determining an entity's trajectory. A smooth trajectory may be achieved by interpolating between consecutive updates rather than extrapolating from single updates. This interpolated trajectory may then be sampled at regular intervals, giving a trajectory log that contains both a constant interval between data points and no discontinuities.

The following calculations interpolate between updates along a single spatial dimension. Multidimensional trajectories can be interpolated by interpolating for each dimension independently.

Since each update contains values for an entity's position, velocity and acceleration, in order to interpolate between two updates, a trajectory needs to be found that matches the received position, velocity and acceleration at the time of both updates. This can be done by assuming the acceleration between the two updates follows a curve given by

$$a(t) = c_0(t-t_0)^3 + c_1(t-t_0)^2 + c_2(t-t_0) + a_0 \tag{4.1}$$

where t_0 is the time of the first update, a_0 is the acceleration value received for the first update, and c_0, c_1, and c_2 are unknown variables that need to be determined. Integrating this equation gives the following functions for velocity and position

$$v(t) = \frac{c_0}{4}(t-t_0)^4 + \frac{c_1}{3}(t-t_0)^3 + \frac{c_2}{2}(t-t_0)^2 + a_0(t-t_0) + v_0 \tag{4.2}$$

$$x(t) = \frac{c_0}{20}(t-t_0)^5 + \frac{c_1}{12}(t-t_0)^4 + \frac{c_2}{6}(t-t_0)^3 + \frac{a_0}{2}(t-t_0)^2 + v_0(t-t_0) + x_0 \tag{4.3}$$

where v_0 and x_0 are the velocity and position values received in the first update, respectively. If the second update was received at time t_1, and the values for acceleration, velocity, and position at this update are denoted as a_1, v_1 and x_1 respectively, then this gives the following system of equations:

$$a_1 = c_0(t_1-t_0)^3 + c_1(t_1-t_0)^2 + c_2(t_1-t_0) + a_0 \tag{4.4}$$

$$v_1 = \frac{c_0}{4}(t_1-t_0)^4 + \frac{c_1}{3}(t_1-t_0)^3 + \frac{c_2}{2}(t_1-t_0)^2 + a_0(t_1-t_0) + v_0 \tag{4.5}$$

$$x_1 = \frac{c_0}{20}(t_1-t_0)^5 + \frac{c_1}{12}(t_1-t_0)^4 + \frac{c_2}{6}(t_1-t_0)^3 + \frac{a_0}{2}(t_1-t_0)^2 + v_0(t_1-t_0) + x_0 \tag{4.6}$$

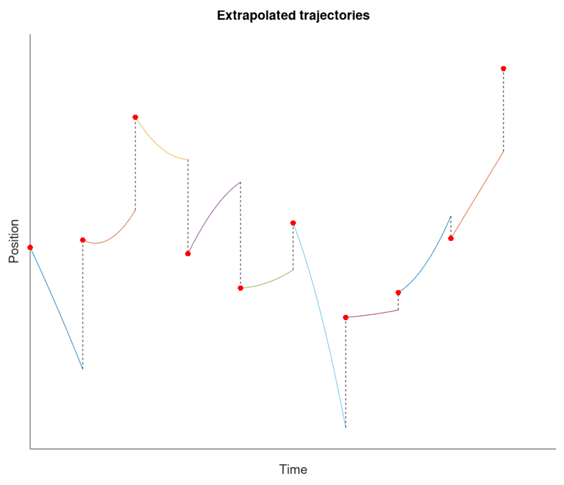
This system of equations may be solved for c_0, c_1 and c_2, giving the following values:

$$c_0 = \frac{10(a_1 - a_0)(t_1 - t_0)^2 - 60(v_1 + v_0)(t_1 - t_0) + 120(x_1 - x_0)}{(t_1 - t_0)^5} \tag{4.7}$$

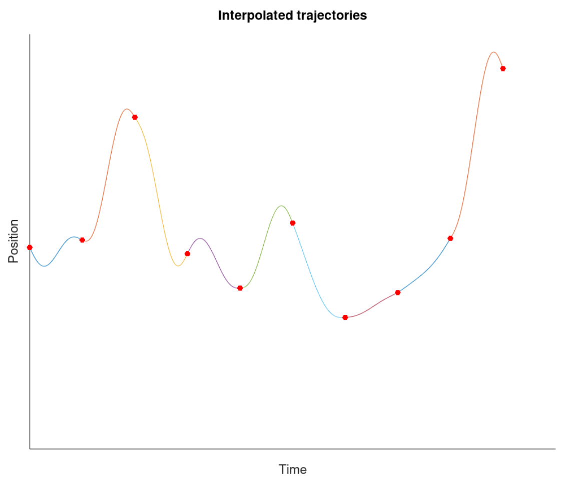
$$c_1 = \frac{(18a_0 - 12a_1)(t_1 - t_0)^2 + (96v_0 + 84v_1)(t_1 - t_0) + 180(x_0 - x_1)}{(t_1 - t_0)^4} \tag{4.8}$$

$$c_2 = \frac{(3a_1 - 9a_0)(t_1 - t_0)^2 - (36v_0 + 24v_1)(t_1 - t_0) + 60(x_1 - x_0)}{(t_1 - t_0)^3} \tag{4.9}$$

This solution was found using the software library SymPy[10]. These values may then be plugged into equations 4.1, 4.2, and 4.3 in order to interpolate between two updates. In order to calculate the full trajectory of an entity, these values need to be calculated for each pair of consecutive updates.



(a) Each update is extrapolated using the received position, velocity and acceleration values until the next update is received. This leads to discontinuities in the trajectory.



(b) The trajectory is interpolated between adjacent updates. The trajectory is continous. The interpolated velocity and acceleration values are also continuous, but this is not shown here.

Figure 4.5 *A comparison of extrapolated and interpolated trajectories for a set of random data points. Each red dot represents a received update. The set of updates is the same in both examples, and each update contains a random position, velocity and acceleration value. The actual units are unimportant since this figure is only meant for illustration, and so the axes are not labelled.*

Figure 4.5 shows a comparison between extrapolated and interpolated trajectories. Since the position, velocity, and acceleration values are random for each update, the extrapolation discontinuities in this figure are likely exaggerated when compared to a more realistic example.

[10]https://www.sympy.org/en/index.html

4.2 Speeding up simulations through parallelization

In order for BTN-NEAT to generate suitable behaviours, it is necessary to evaluate a large number of individual behaviours. Since it is possible to evaluate individuals in a generation in any order, and their fitness values are not used until the creation of the next generation, the evaluation can be performed in parallel.

Kubernetes is a good candidate for doing this due to its orchestration capabilities. However, Kubernetes requires containerized applications, and a Docker image was therefore created for each of the applications shown in figure 4.2.

Although there exists a Windows version of Docker, it is more common to run Docker on the Linux kernel, which is what we did in this work. The data logger and evolution controller described in section 4.1 could easily be run on Linux since both Python and Java are platform-independent. On the other hand, NGTS is a Windows program and could not easily be run on Linux. To solve this, we ran NGTS using Wine, which is "a compatibility layer capable of running Windows applications on [...] Linux".[11] The downside to this was that NGTS ran a lot slower under Wine than it would have on Windows.

In order to run the ML system in Kubernetes, the simulation control component of the evolution controller needed to be expanded. When run on a single computer, the simulation control directly starts and stops NGTS processes running on the same computer. However, inside a Kubernetes cluster, the simulation control has to directly control Kubernetes instead, in order to create and destroy Kubernetes pods containing NGTS. This was done by using the Python Kubernetes Client library.[12]

The Kubernetes-based system was defined using a Helm chart[13], consisting of two pods: one containing the evolution controller and one containing NGTS, the remote control program, and the data logger. When the evolution controller system generated new behaviours and scenarios for NGTS to use, these files were stored to a shared drive. The evolution controller system then used the Kubernetes client library to start several instances of the NGTS pod. These pods could then read behaviours and scenarios from the shared drive, and store log files to the drive when the simulation was complete. Finally, the evolution controller could read the logs, evaluate the behaviours, and create a new generation of behaviours and scenarios.

[11] `https://www.winehq.org/`, accessed 2021-08-23
[12] `https://github.com/kubernetes-client/python`, accessed 2021-08-24
[13] `https://helm.sh/`, accessed 2022-08-02

5 Experiment setup

This chapter outlines the setup and delimitations for our imitation learning experiments. This includes defining the learning objective, describing the air combat scenarios used in our experiments, the BTNs that the demonstration behaviours were based on, the set of BTN nodes available to BTN-NEAT, and the configuration of the learning algorithm.

5.1 Learning objective

In this work, the goal was to use imitation learning to produce agents operating in one vs one scenarios that behave in line with a set of provided demonstrations. These scenarios, described in section 5.3, involve a friendly (blue) aircraft whose behaviour was static and a hostile (red) aircraft whose behaviour was learnt by imitation. This necessitates a set of demonstrations for the algorithm to learn from. Our algorithm, BTN-NEAT, was used to create BTNs that imitate two aspects of the demonstration behaviour: its dynamic state, specifically its position, speed, and course, and its usage of sensors. These aspects of the demonstration behaviour were captured and stored by the data logger, which is described in section 4.1.2. This data logger was also used to capture the same aspects of the evolved behaviours, allowing comparisons between the evolved behaviours and the demonstration.

Although behaviour cloning is a type of imitation learning, it was unsuitable for use with NGTS and BTNs. This is because BTNs do not directly map states to action, and behaviour cloning involves approximating a state-to-action function. Instead, we attempted to learn behaviours by maximizing the similarity between the evolved behaviours and the demonstrated behaviour by applying the fitness functions defined in section 3.2.2.

There was also no fitness threshold used to determine whether the learning algorithm had evolved a good enough behaviour. Instead, we manually inspected the trajectories of the BTNs with the best fitness values during the experiment, and stopped the learning process when these trajectories were judged to be sufficiently similar to the demonstration. Additionally, we manually inspected the best BTNs. If an evolved BTN was found to encode the same behaviour as the demonstration BTN, regardless of any vestigial nodes left over by the evolution process, it was found to be satisfactory.

5.2 BTN configuration and search space

In all the following experiments, the first three BTN nodes were kept static for all evolved genomes. These static nodes are illustrated in figure 5.1. The BTN-NEAT algorithm would then add subsequent nodes through its genetic operations. Furthermore, all node parameters were pre-defined, meaning the learning algorithm was not able to change the parameter values of the nodes it introduced, such as the amount of time to wait when using a *Delay* node or the angle to turn when using a *Steer Course* node.

NGTS contains more than 400 unique types of BTN nodes, some of which are designed for platforms other than aircraft, such as ground or naval platforms. Allowing BTN-NEAT to select among all

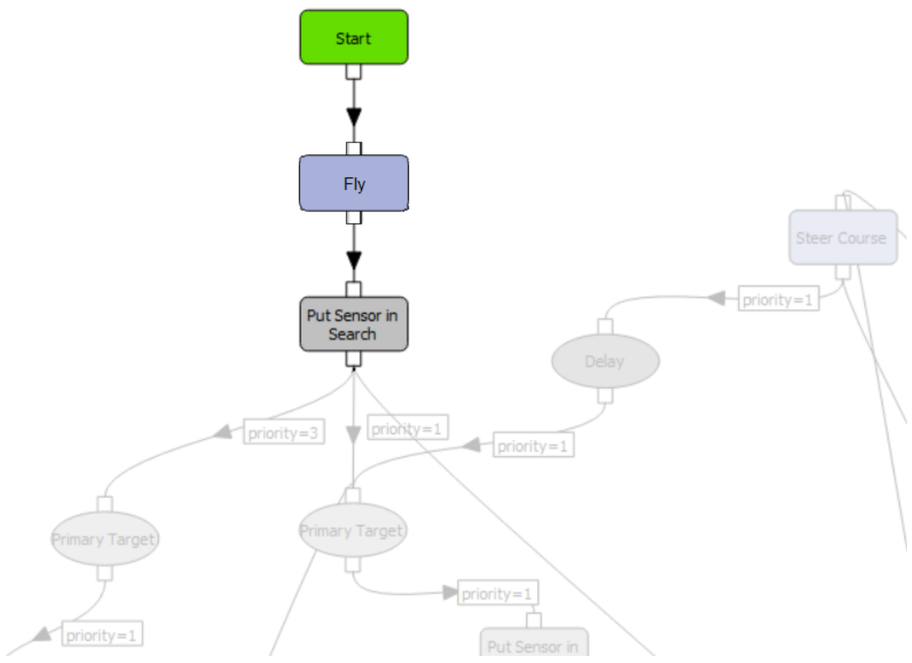


Figure 5.1 *An illustration of the three hard-coded BTN nodes present in all genomes; Start, Fly and Put Sensor in Search. The shaded nodes represent an arbitrary composition of subsequent nodes.*

nodes would result in an unreasonably large search space for the evolutionary algorithm, meaning a satisfactory behaviour would almost certainly not evolve for a very long time. Additionally, some nodes represent very similar actions. In order to keep the problem simple, BTN-NEAT was restricted to selecting from only eight different nodes. These eight nodes are summarised in table 5.1. As can be seen from the table, the parameter of the *Steer Course* node is set to 179° rather than 180°. The reason is that an angle of 180° results in the aircraft turning in arbitrary directions (left or right) each time the node is called, resulting in an unstable trajectory outcome, while an angle of 179° will make the aircraft always turn right.

Even with the number of available nodes reduced to eight, there is still a large space of possible behaviours for the evolutionary algorithm to search through. Approximating the size of this search space can give an indication of whether the algorithm is actually learning anything useful; if it were possible to find a satisfactory behaviour in a reasonable time by generating random behaviours, there would be no need for the evolutionary algorithm.

Although a BTN can, in theory, become infinitely large and complex, most BTNs consist of somewhere between 10 and 100 nodes in practice. One of our demonstration BTNs (shown in figure 5.5) consisted of seven command nodes and eight trigger nodes. If both the type and number of nodes and the connections between them are taken as fixed, this still gives a search space of (number of possible command nodes)^7×(number of possible trigger nodes)^8 = $3^7 \times 5^8 \approx 8.5 \times 10^8$. Given an evaluation time of 1 s per behaviour, a random search would take over 13 years on average to find the original behaviour in a search space of this size. In reality, the number of nodes and connections is not fixed, making the actual search space much larger, while the evaluation of behaviours takes on the order of minutes per behaviour. If BTN-NEAT is able to find satisfactory

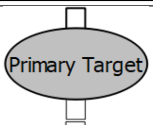
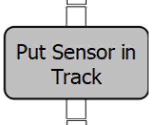
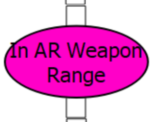
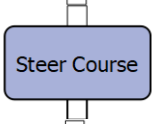
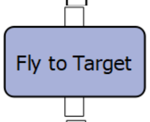
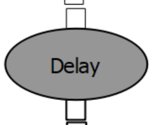
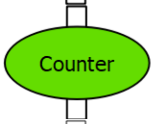
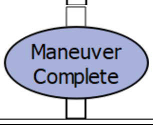
Node		Type	Parameter	Description
1		trigger	N/A	A primary target exists
2		command	N/A	Command sensors to track target
3		trigger	N/A	See if any AR weapon is in range
4		command	179°	Command a heading goal in degrees true
5		command	N/A	Fly towards the specified target
6		trigger	180 s	Delay for specified number of seconds
7		trigger	max = 3	Triggers for specified number of counts
8		trigger	N/A	Route maneuver is complete

Table 5.1 *An overview of the BTN nodes used in the experiments. The descriptions are taken directly from the NGTS Behavior Editor. These nodes make up BTN-NEAT's search space. The Steer Course node's parameter is set to* 179° *to make the aircraft always turn right, as opposed to* 180°, *which results in arbitrary turning directions. Note that the* Start, Put Sensor in Search, *and* LOOP START *nodes were hard-coded into each BTN but were not a part of BTN-NEAT's search space.*

solutions within a reasonable time, the algorithm is much more efficient than a random search.

5.3 Scenarios

Three separate one vs one air combat scenarios involving one friendly aircraft (blue) and one hostile aircraft (red) were used in the imitation learning experiments, as well as a demonstration of how the red aircraft *should* behave for each scenario. The scenarios range from simple to slightly more complex. The recurring idea is for the red aircraft to safely intercept its enemy by getting close enough to use short-range weapons without receiving any damage. The initial conditions, which are summarised in table 5.2, are equal for all three scenarios.

	Blue aircraft	Red aircraft
Speed	0.5 Mach	1.2 Mach
Lat	63°30′46.04″N	63°58′04.10″N
Long	17°09′52.91″E	19°13′47.47″E
Distance	113.5 km	
Altitude	25128 ft MSL	25296 ft MSL
Course	59°	244°

Table 5.2 *The initial conditions of the red and blue CGFs for the three scenarios. The altitude is given in feet above mean sea level (MSL). The course is given in degrees relative to the north direction.*

5.3.1 Scenario 1: Flee

The first scenario is the simplest of the three, and only the BTN nodes numbered 1–5 in table 5.1 are required to compose the desired behaviour. In the demonstration for this scenario, both aircraft fly straight towards each other, but once the red aircraft has its enemy within missile fire range, it turns and flees in the opposite direction. The red aircraft has a considerably higher velocity compared to the blue aircraft, so the distance between the two increases after the red aircraft has turned away. No missiles are fired in this scenario.

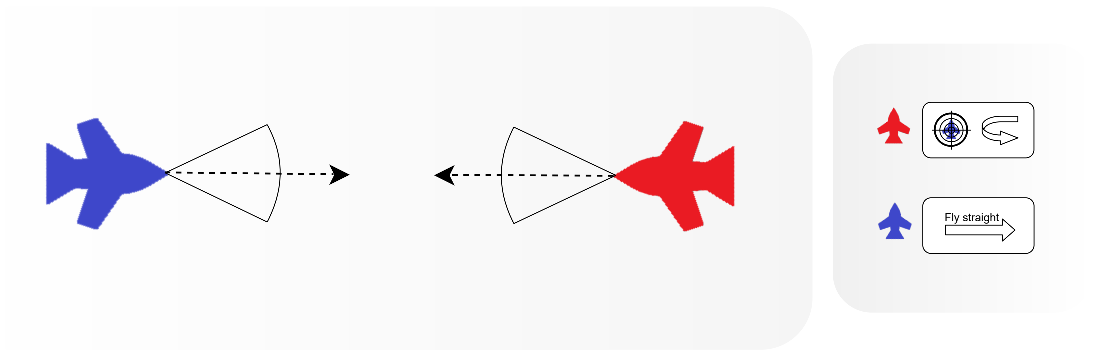


Figure 5.2 *A summary of the first scenario: Flee. The CGFs fly towards each other from their initial positions. Once the blue aircraft comes within the red aircraft's missile fire range, the red aircraft turns and flees in the opposite direction.*

The BTN used to produce the demonstration behaviour for this scenario is shown in figure 5.3, and the duration of the scenario is 10 minutes.

5.3.2 Scenario 2: Feint

The second scenario extends scenario 1: *Flee* in the following way. When the red aircraft turns away from the blue aircraft, it does so only for a little while before turning back to face the blue aircraft again. This is repeated three times, resulting in a trajectory of a series of rounded rectangles. This behaviour is meant to feint an attack on the blue aircraft, causing it to expend a missile. By turning

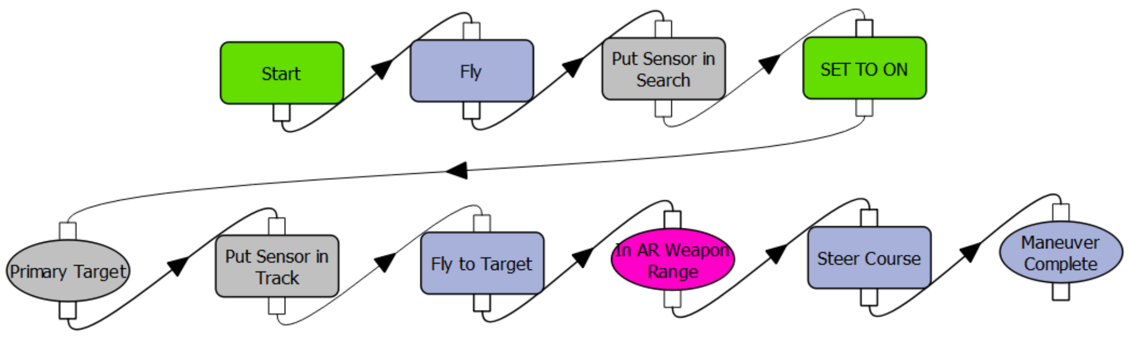


Figure 5.3 *The handmade BTN used to produce the demonstration for scenario 1: Flee.*

away early, the red aircraft should be able to fly out of range, thus depleting the blue aircraft's missile stores. Finally, after the third feint, the red aircraft intercepts the blue aircraft. This behaviour requires the full range of BTN nodes from table 5.1.

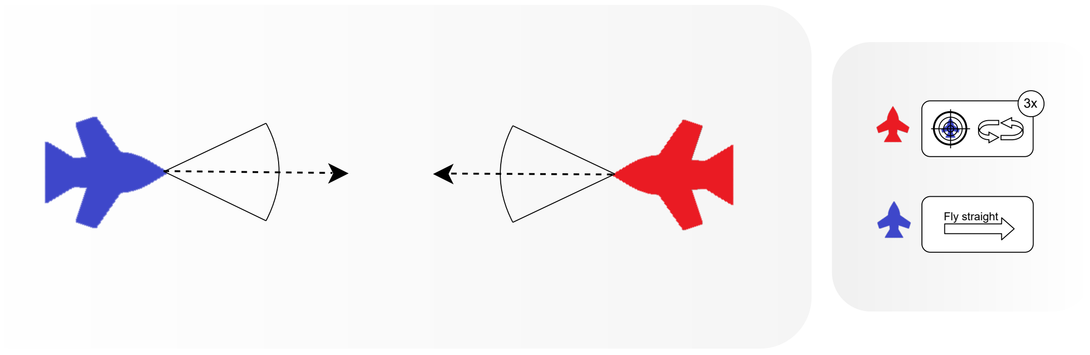


Figure 5.4 *A figure depicting scenario 2: Feint. The blue aircraft flies straight at a considerably lower speed than the red aircraft. The aircraft initially fly towards each other. Once the red aircraft has the blue aircraft within its missile fire range, the red aircraft evades by turning in the opposite direction. After a while, the red aircraft turns back towards the blue aircraft. This is repeated three times.*

The BTN used to produce the demonstration for this scenario is shown in figure 5.5, and the scenario's duration is 20 minutes.

5.3.3 Scenario 3: CAP

The third and final scenario is similar to scenario 2: *Feint*, except that instead of flying straight ahead, the blue aircraft performs a combat air patrol (CAP) with a leg length of approximately 18 km, initially flying north. The BTN used to produce the demonstration behaviour for the red aircraft is identical to the one used in scenario 2: *Feint*, shown in figure 5.5. The example scenario lasts for about 27 minutes.

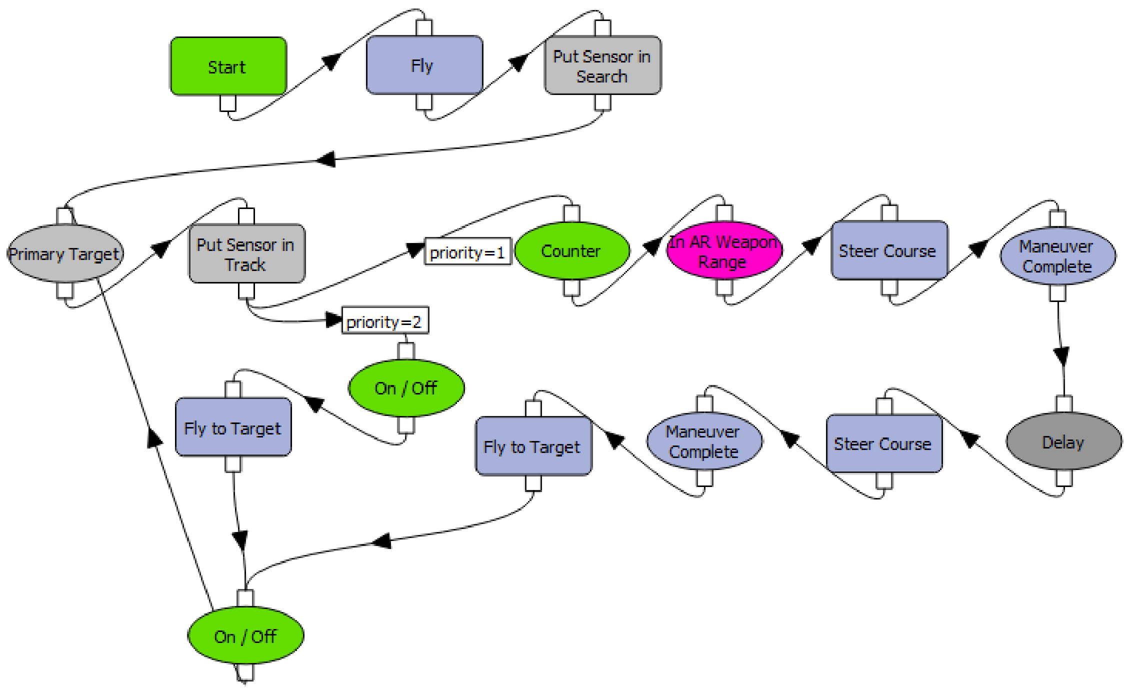


Figure 5.5 *The handmade BTN used to produce the demonstrations for both scenario 2: Feint and scenario 3: CAP.*

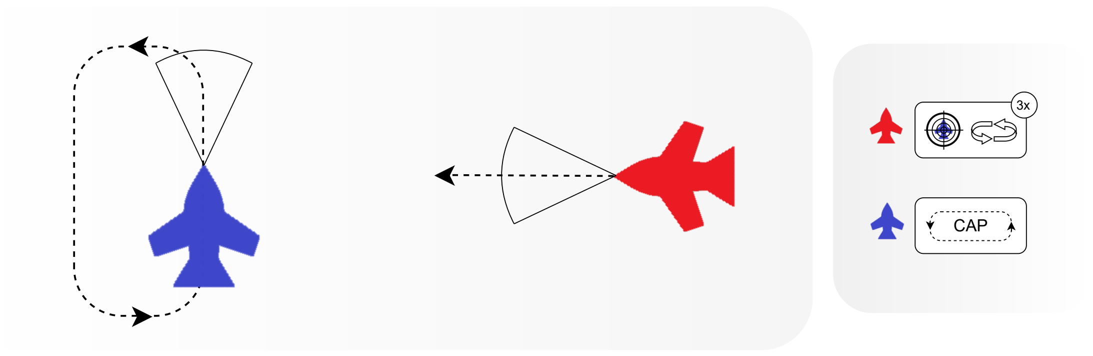


Figure 5.6 *A figure depicting scenario 3: CAP. The blue CGF performs a combat air patrol (CAP). The red aircraft initally flies towards the blue aircraft. Once the blue aircraft comes within the red aircraft's missile range, the red aircraft evades in the same way as in scenario 2.*

5.4 BTN-NEAT hyperparameters

Hyperparameters are the parameters of the machine learning algorithm itself and are normally not adapted by the learning algorithm [21, p. 117]. NEAT has a considerable number of hyperparameters whose values influence the performance and learning outcome, and a few BTN-specific hyperparameters were also introduced to the BTN-NEAT algorithm. The values used for the hyperparameters throughout the imitation learning experiments are summarised in table 5.3.

NEAT hyperparameter	**Value**
Population size	128
Number of elites	10
Default node type	random $\in$ $\{\text{trg}, \text{cmd}\}$
Number of initial BTN nodes	2
Add connection probability	0.4
Delete connection probability	0.2
Add node probability	0.4
Delete node probability	0.4
Enable/disable connection probability	0.0
Disjoint coefficient	1.0
Compatibility threshold, δ	4.2

Table 5.3 *An overview of the hyperparameters used with NEAT in the imitation learning experiments.*

- *Population size* decides how many individuals there should be in each generation.
- *Number of elites* determines how many of the top-ranking individuals from the current generation are transferred to the next generation.
- *Default node type* determines the node type when a new node gene is inserted into a genome, which in this case is either a trigger or command chosen at random.
- *Number of initial BTN nodes* controls the maximum number of nodes a BTN consists of upon initialization.
- *Add connection probability* sets the probability of adding a connection to a genome through mutation.
- *Delete connection probability* sets the probability of deleting one of a genome's connections through mutation.
- *Add node probability* sets the probability of adding a node to a genome through mutation.
- *Delete node probability* sets the probability of deleting one of a genome's nodes through mutation.
- *Enable/disable connection probability* sets the probability of toggling the disabled flag of a genome's connection through mutation, either disabling or re-enabling the connection.
- *Disjoint/excess coefficient* weights the contribution of both disjoint and excess genes when calculating the compatibility between individuals according to equation 2.5.
- *Compatibility threshold*, δ determines which individuals that belong to the same species.

6 Experiment results

This section presents the results of the experiments and how different computational resources and fitness functions affected the results. The scenarios used in the experiments apply the initial conditions summarized in table 5.2. The section concludes by discussing some simulation issues that arose while the experiments were conducted, mainly related to time management.

6.1 Computational resources

Different computational resources were employed when using the machine learning system (section 4) throughout the following three experiments. These computational resources ranged from running the system sequentially on a single desktop computer to run it in parallel on computer clusters. As will become apparent in later sections, these computational resources posed their various pros and cons.

Desktop computer

This setup consisted of a single desktop computer running Windows 10. Since NGTS is a Windows native program, this allowed us to run NGTS much faster than in any other setup, and experiments could be conducted with simulation time scales up to $r = 50$. However, it was only possible to run a single instance of NGTS at a time using this setup. The desktop computer contained 12GB of RAM and an Intel Xeon CPU with six cores running at 2.8GHz.

Small-scale computer cluster

In order to run several simulations in parallel, a small-scale computer cluster consisting of 32 CPU cores was used with Docker containers. Because the cluster was distributed across two machines, Kubernetes was used to orchestrate the Docker containers. Kubernetes was set up using Rancher, an open-source system that is able to set up Kubernetes clusters almost automatically [68]. With this setup, it was possible to run up to 32 NGTS instances in parallel. However, each NGTS instance had to be run using Wine, causing slower simulations compared to running them natively in Windows.

High-performance computer cluster

The final and theoretically most powerful setup was a high-performance computing (HPC) cluster consisting of tens of computers, each with tens of CPU cores. The total amount of CPU cores would have allowed us to evaluate an entire generation of BTNs in parallel. These machines were running SUSE Linux Enterprise Server 12[14], and utilised PanFS[15], which is a special file system allowing all computers in the cluster to access a shared network drive. Unfortunately, running Kubernetes on this cluster proved problematic, and in the end, this large-scale computer cluster could not be used

[14]`https://www.suse.com/products/server/`, accessed 2022-08-02
[15]`https://www.panasas.com/products/panfs/`, accessed 2022-08-02

in any of the experiments. Nevertheless, the setup is presented here to share the experiences made when attempting to make it work.

PanFS was not compatible with the Docker containers: these rely on a larger amount of symbolic links than the shared file system would allow. This meant we had to install both Kubernetes and each container image locally on the machines we would use; however, since these were intended to use the large shared network drive, each machine only had a small amount of local disk space.

Once we managed to install and run Kubernetes, we ran into more problems: Machines in the cluster would suddenly reboot themselves. This seemed to happen at random, and it often happened to a different computer than the one we were using. We could not find the source of these problems, and since these reboots risked adversely affecting the other users of the cluster, we decided to stop working with the HPC.

6.2 Experiment 1

The first experiment was conducted to verify that the components of the learning system were interacting as expected. The learning system was prepared with scenario 1: *Flee* (section 5.3.1), running on a single desktop computer as described in section 6.1. Initially, neither the radar component of equation 3.1 nor the decay term of equation 3.2 was used when calculating fitness scores. This meant that the weights were effectively $c_e = c_p = 0$, reducing the fitness function to simply adding up the distances between the two trajectories at every time step.

However, it quickly became apparent that all evolved BTN solutions would fly straight past the blue aircraft rather than turning away when reaching the missile fire range. This undesired behaviour occurred because the learning algorithm had no incentive to add the *Put Sensor in Track* node to the genomes early on. The lack of this node meant that other nodes that depended on the aircraft's sensors being in tracking mode, such as the *In AR Weapon Range* node, would not be able to work. We realised that a very careful composition of nodes is required for a BTN to yield the desired behaviour, but without explicitly motivating BTN-NEAT, this composition was unlikely to occur within a reasonable amount of time.

For this reason, we introduced the radar punishment aspect of 3.1 and set its coefficient to $c_p = 10000$. This added a penalty of 10000 fitness points to a genome's total fitness for every time step its radar status failed to match the radar status of the demonstration. The reason for selecting such a large value was to force the learning algorithm to quickly evolve genetic compositions that included the *Put Sensor in Track* node, before focusing on the manoeuvring aspects. With the updated fitness function in place, the learning algorithm found BTNs producing the correct pattern, and the initial experiment was considered a success. Details about the experiment are summarised in table 6.1. The BTN-NEAT algorithm needed approximately 600 hours, or 25 days, of simulated flying in order to learn the demonstrated behaviour.

6.3 Experiment 2

The second experiment was performed to test the learning system against a problem of increased complexity. This experiment used scenario 2: *Feint* (section 5.3.2). First, the experiment was

	Value
c_e	0
c_p	10000
r, simulation time scale factor	50
time elapsed per simulation run	12 s
time to learn demonstrated behaviour	~12 h

Table 6.1 *Important aspects of experiment 1.*

conducted using the single desktop computer, similar to the previous experiment, and the learning algorithm did indeed find solutions that gave the desired behaviour. However, as can be seen in table 6.2, the learning algorithm spent much more time learning the demonstrated behaviour than for the first scenario. The BTN-NEAT algorithm needed approximately 2200 hours, or nearly 92 days, of simulated flying in order to learn this behaviour. This increase was partly because the *Feint* scenario lasted twice as long as the *Flee* scenario, and partly because the demonstrated behaviour in the *Feint* scenario was more complex than the behaviour demonstrated in the *Flee* scenario. Thus, BTN-NEAT required an increased number of evolutionary steps to find a satisfactory BTN solution.

	Value
c_e	0
c_p	10000
r, simulation time scale factor	50
time elapsed per simulation run	24 s
time to learn demonstrated behaviour	~44 h

Table 6.2 *Important aspects of experiment 2.*

We discovered that the choice of hyperparameters might dramatically influence the number of evolutionary generations required for BTN-NEAT to find adequate solutions. Therefore, we sought to explore combinations of hyperparameters that would produce better BTNs in fewer steps. However, as discussed in section 7, it proved challenging to perform simulations fast enough for a search of hyperparameters to become feasible.

Running NGTS sequentially on a single computer proved to be a bottleneck. This restriction motivated us to transition to running simulations in parallel using containers, which could, in theory, greatly reduce the time needed to find good behaviour. This new architecture used Docker, Kubernetes, and Rancher, running on the small-scale computer cluster, to speed up the optimization process. As described in section 6.1, NGTS was not able to run at as high time scale factors inside a container as when running directly on a Windows computer. Thus, we reduced the simulation time scale factor from 50 to 8[16]. With 32 CPU cores available and each NGTS instance requiring 2 CPU cores to run at a time scale factor of $r = 8$, it should, in theory, be possible to simulate 128 seconds for each wall-clock second.

The experiment was conducted again, using Kubernetes to distribute and parallelize the simulations. We obtained similar results to the previous run but in much less time, as can be seen in table 6.3.

[16]Later, further analyses would reveal that NGTS simulations might not be run reliably in Docker containers at simulation time scale factors $r > 4$ (see section 6.5). This matter was discovered when conducting experiment 3.

The 2200 hours of simulated flying could now be performed in about 17 hours. The process was run three times to verify the learning system's reliability, all of which yielded satisfactory BTN solutions after stagnating at 47, 43 and 44 generations, respectively. Figure 6.1 on page 57 shows the resulting BTN and trajectory from one of these runs.

The evolved trajectory differed from the demonstration trajectory because the evolved behaviour caused the aircraft to turn back towards the enemy sooner than the demonstration aircraft. In order to match the demonstration trajectory more closely, the BTN-NEAT algorithm would have needed to evolve a *Maneuver Complete* node between the *Steer Course* node and the *Delay* nodes. Nevertheless, experiment 2 was considered a success.

We also attempted to run scenario 2: *Feint* using the semantic fitness function described in section 3.2.2.2. This fitness function compares two sequences of semantic events rather than two trajectories. This was done in order to be able to vary the initial conditions like position, speed and course during the optimization of BTNs, thus making them more robust to different environmental circumstances. These initial conditions could not be varied while using the trajectory fitness function, as this would have large, negative impacts on the fitness values, even for BTNs that exactly matched the demonstrated behaviour. Unfortunately, we did not achieve good results using the semantic fitness function. It requires further development and testing before it can be used for our purpose.

	Value
c_e	0
c_p	10000
r, simulation time scale factor	8
time elapsed per simulation run	150 s
NGTS instances (parallel)	16
number of generations	~45
time to learn demonstrated behaviour	~17 h

Table 6.3 *Important aspects of experiment 2, using Docker containers and the small-scale computer cluster to distribute simulations.*

6.4 Experiment 3

For the third experiment, we used the same containerised setup that was used by the end of experiment 2 (see section 6.1 for more details about the computational resources used). The learning system was prepared with scenario 3: *CAP*, the trajectory fitness function with coefficients $c_e = 0, c_p = 10000$, and BTN-NEAT was run using the hyperparameters shown in table 5.3. We found that behaviours evolved to exploit a local optimum of the fitness function, causing the aircraft to fly in circles, as can be seen from figure 6.2 on page 58.

Flying in circles gives a reasonably high fitness score because by flying in circles, the aircraft always stays near the demonstration trajectory, which consists of a set of rounded rectangles all centred near the aircraft's starting point (see figure 6.2). In contrast, flying directly towards the opposing aircraft will initially match the demonstration trajectory exactly, but soon deviate much more from the rounded rectangles than just flying in circles will.

However, in order to learn to turn away from the opposing aircraft, the agent would first need to learn to fly towards it. Since flying towards the opponent gave lower fitness scores than flying in circles, this meant the evolutionary algorithm would have to go through a step that initially seemed sub-optimal in order to evolve better behaviours later. This did not seem to happen, and even after running for a very long time, the evolutionary algorithm never evolved past the local optimum of flying in circles.

This problem did not arise during the *Feint* scenario because here, the opposing aircraft was flying in a straight line. Since the demonstration aircraft always turned at a given distance from the opposing aircraft, the turning point of each rounded rectangle in the demonstration trajectory moved with the opposing aircraft. In this scenario, flying in circles would lead to progressively worse punishments as the expected position of the aircraft, as given by the demonstration trajectory, got farther and farther away. On the other hand, flying towards the enemy at all times (which eventually leads to circling it) keeps the aircraft closer to the demonstration trajectory than flying in circles, and thus leads to a lower punishment. In this case, flying towards the enemy was a better behaviour than circling, and this behaviour would therefore be chosen as one the evolutionary algorithm could propagate and mutate, eventually leading to the evolution of turning away from the opposing aircraft.

To make flying towards the enemy seem less sub-optimal in this scenario, we introduced an exponential term to the distance punishment term of equation 3.3 that made the punishment decay over time. Thus, the Euclidean distance component was weighted depending on how much simulation time had elapsed, as described in section 3.2.2. This exponential term should lead the evolutionary algorithm to try to learn behaviours in a sequential fashion: matching for instance the first minute of a demonstration behaviour is given more weight than matching the second minute, but once that first minute is learned, the only way to increase the fitness is to match the second minute, which is given more weight than the third minute. Using this method, flying towards the enemy results in a higher fitness score than flying in circles. This is because the fitness function gives a higher score to a trajectory that initially matches the demonstration exactly, and later diverges, than it gives to a trajectory that stays in the general vicinity of the demonstration track for the entire duration of the scenario.

The time constant c_e was calculated according to equation 3.3 using a simulation time scale factor of $r = 8$, setting the half-life (i.e. the time it takes for the punishment to decay by half) to the value of $\lambda = 60$. This value was arbitrarily chosen but yielded good results. The value of c_e was then found as:

$$c_e = \frac{r \ln 2}{\lambda} = \frac{8 \ln 2}{60} = 0.0924$$

This value was then used when calculating the Euclidean distance punishment.

Scenario 3: *CAP* was rerun using the updated fitness function, and the optimization stagnated after 51 generations, yielding a much-improved result. The trajectory of the evolved behaviour now resembled the demonstration reasonably well, indicating that the exponential decay of punishment is indeed impactful. However, the behaviour would not send the red aircraft to intercept its opponent after the third turnaround like in the demonstration, but rather keep flying in ellipses due to the lack of a *Counter* node and a *Fly to Target* node. Figure 6.3 shows the BTN and trajectory produced

	Value
c_e	0.0924
c_p	10000
r, simulation time scale factor	8
time elapsed per simulation run	200 s
NGTS instances (parallel)	16
number of generations	51
time to learn demonstrated behaviour	~23 h

Table 6.4 *Important aspects of experiment 3 using containers to distribute simulations.*

by the best genome from this run, and figure 6.4 shows the fitness value of each generation's best genome until the evolutionary process stagnates. An overview of the experiment can be seen in table 6.4. As can be seen, the BTN-NEAT algorithm needed approximately 3000 hours, or 125 days, of simulated flying in order to learn the demonstrated behaviour.

6.5 Time management issues

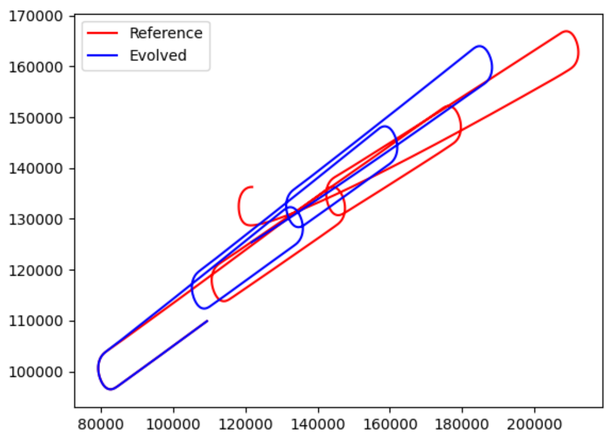
Analyses revealed that when running under Wine, NGTS simulations will occasionally lag, either due to a lack of CPU resources or network issues. However, the DIS standard provides no mechanism to synchronize the simulation time of different simulation systems. It is possible to mark DIS messages with a simulation timestamp, but most simulation systems, NGTS included, use the wallclock time at which the data was valid as the simulation timestamp. This means that the timestamp does not necessarily coincide with the simulation time, and timestamp errors can become significant when the system experiences lag, causing the simulation time to drift further out of synchronization with the wallclock time.

Time stamp errors mean that even if an evolved behaviour behaves exactly the same as the demonstration behaviour, a punishment might still be applied to it because its trajectory was given incorrect simulation timestamps. The system lag introducing these errors is random and unpredictable, leading to random and unpredictable variations in the fitness score applied to a behaviour. Figure 6.5 on page 61 shows how the choice of simulation time scale factor affects the reliability of the states of the agents while running scenario 2: *Feint* in Docker containers. Larger simulation time scale factors especially increase the standard deviation of the agents' X, Y and Z coordinates.

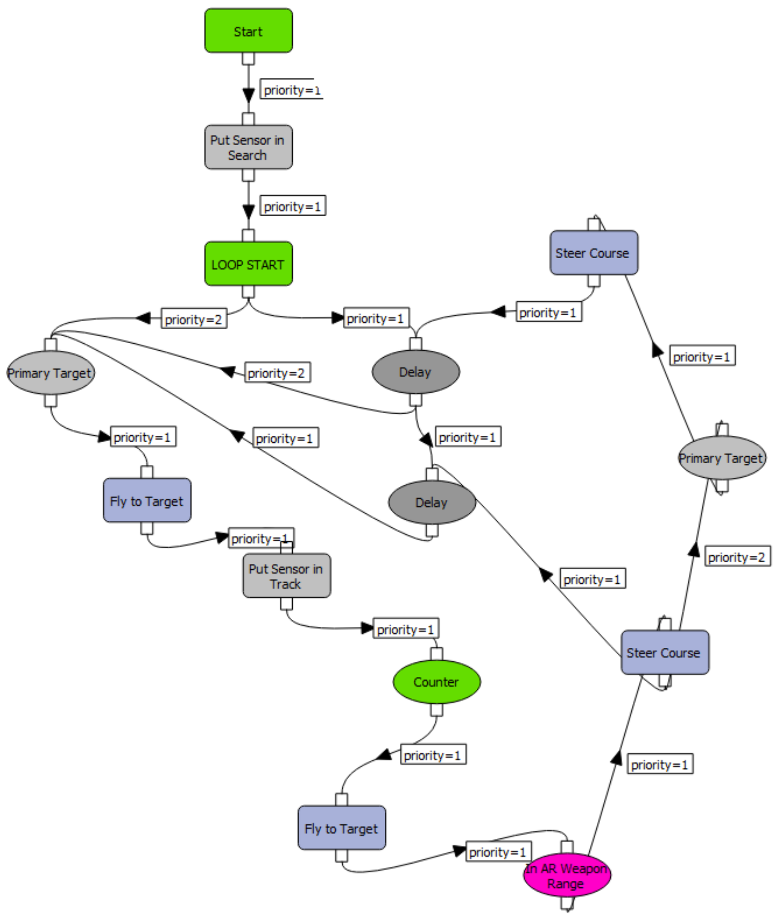
These findings reveal that careful consideration must be made when selecting values for the simulation time scale factor, r, especially when NGTS is containerized. Scaling factors $r > 1$ drastically increases the probability that NGTS will struggle to keep the execution stable, causing lags in the simulation. When simulating much faster than in real time, even the same behaviour is likely to yield significantly different fitness scores over different runs. Thus, the fitness value calculated after simulating an evolved BTN could be significantly different from what it should have been had the simulation been running without lag at a lower simulation time scale factor.

With some further experimentation, we found that when running the machine learning system using the small-scale computer cluster (section 6.1), the simulation could not be run at a time scale factor of $r > 4$ without introducing errors. Thirty-two instances running at a time scale factor of $r = 4$ gives a total simulation speed of 128x faster than real-time, which is faster than running on a single desktop computer, but not a substantial increase. However, it is possible to expand this setup by adding more machines to the cluster, allowing even more NGTS instances to run in parallel.

Ultimately, this lack of reliability makes it unfeasible to run NGTS simulations much faster than real-time but instead requires the user to select slower and more stable simulation speeds. The drawback, of course, is that small values for r increase the time needed to evolve a satisfactory behaviour.

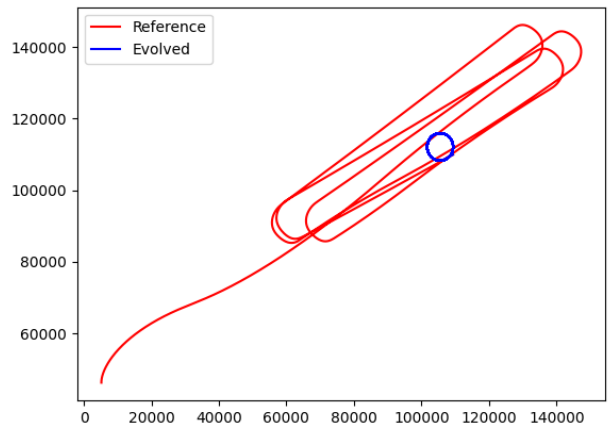


(a) The path of the demonstration aircraft (red) and the path produced by the best evolved behaviour model (blue).

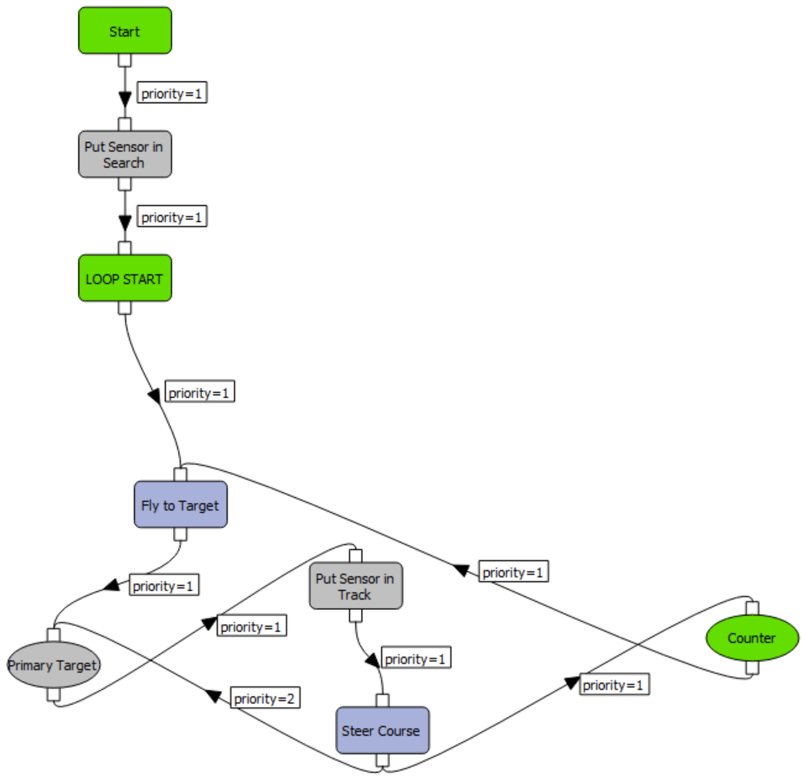


(b) The BTN produced by the best genome in experiment 2.

Figure 6.1 *The result from one of the three runs of experiment 2 that were conducted to verify the learning system's reliability.*

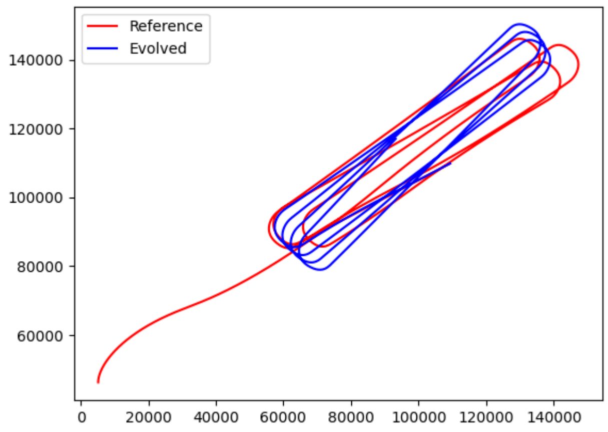


(a) The path of the demonstration aircraft (red) and the path produced by the evolved behaviour model (blue). The behaviour has reached a local optimum of only flying in a circle.

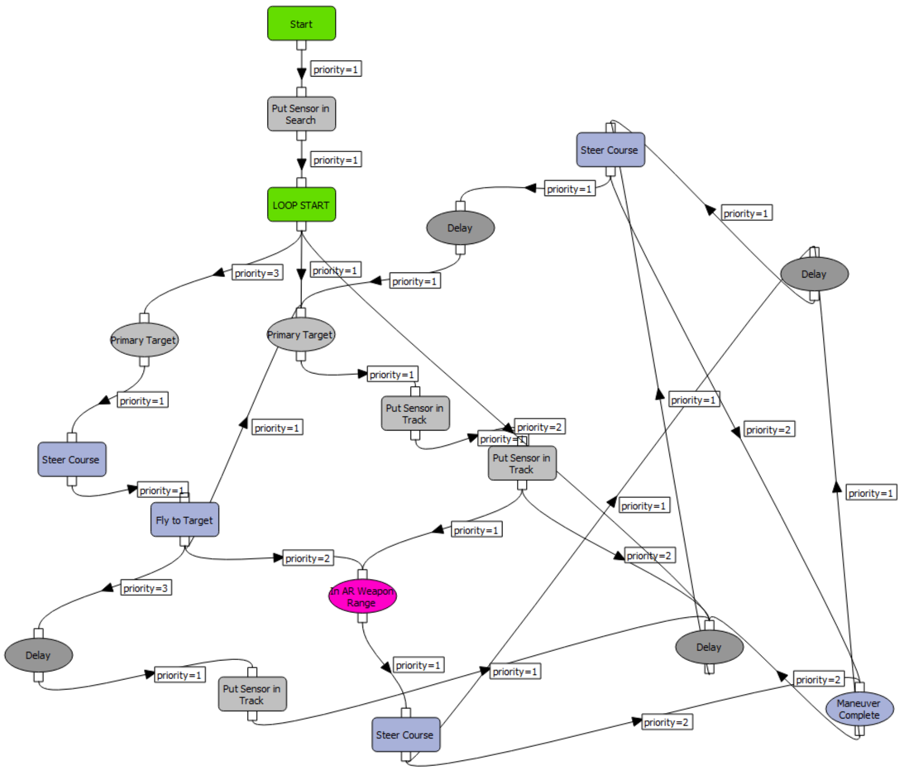


(b) The evolved BTN that lead to the aircraft flying in circles.

Figure 6.2 *The result from a run of experiment 3, prior to introducing the time decay factor of the fitness function, which generated a behaviour that caused the aircraft to fly in circles.*



(a) The path of the demonstration aircraft (red) and the path produced by the best evolved BTN (blue).



(b) The BTN produced by the best genome in experiment 3.

Figure 6.3 *Figures showing the BTN and its resulting trajectory produced by the best genome evolved in experiment 3 after introducing the time decay factor to the fitness function.*

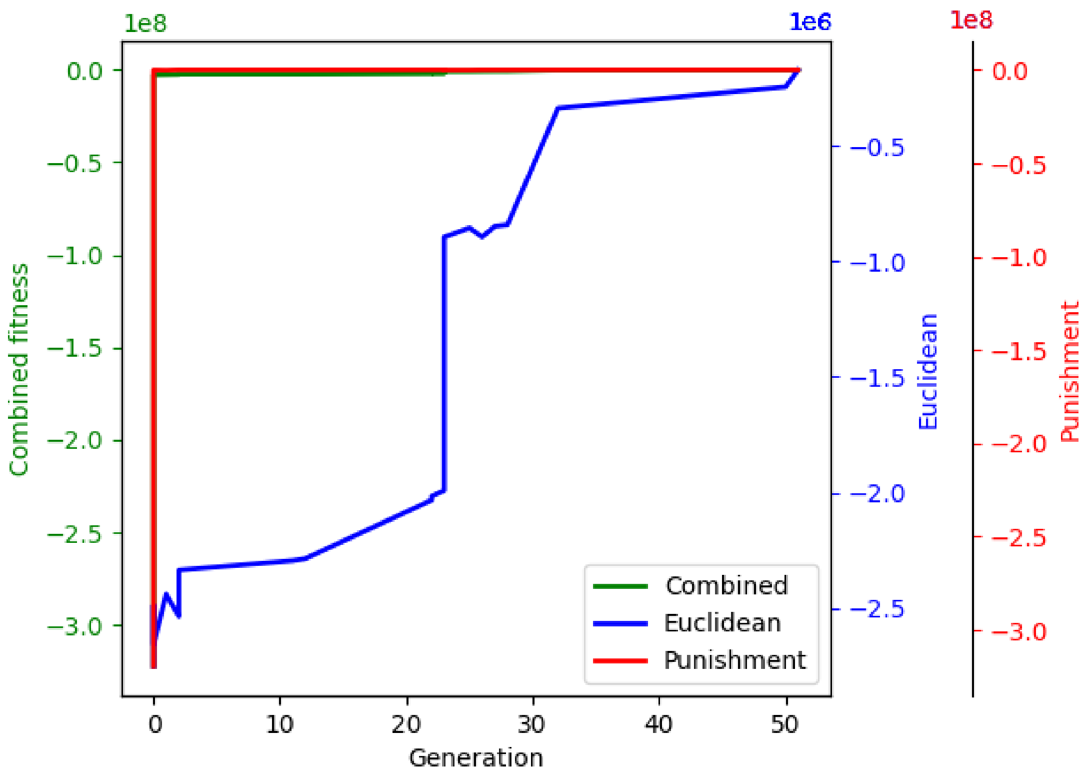
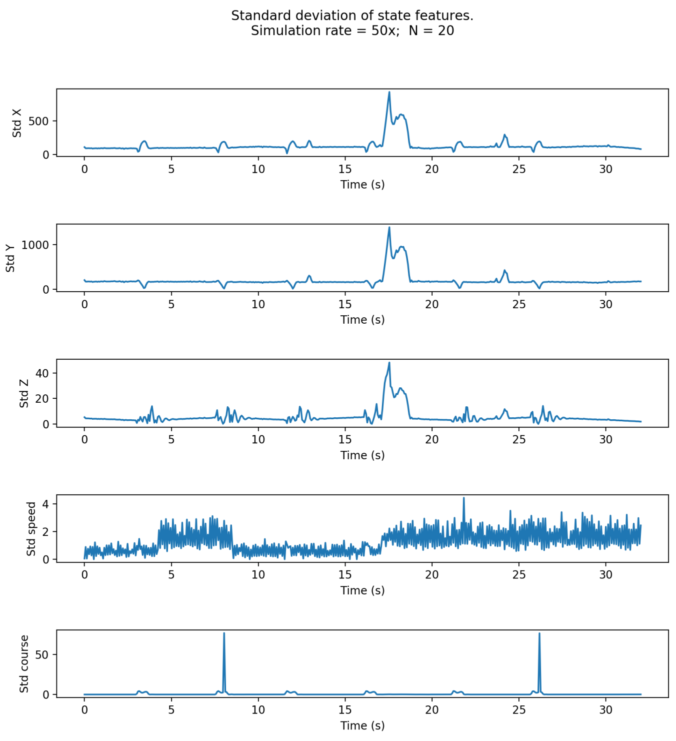
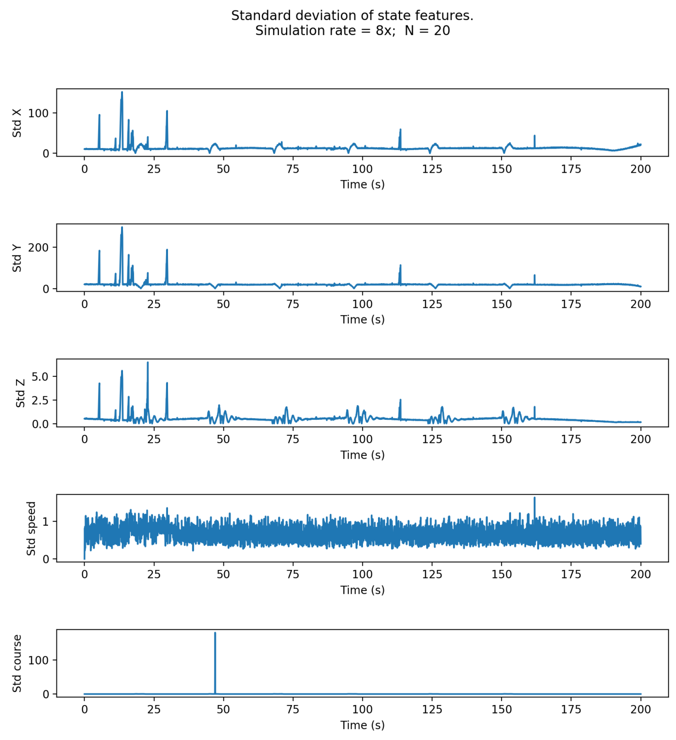


Figure 6.4 *A plot of the fitness values of each generation's best genome in experiment 3. The blue graph shows the trajectory fitness component, the red shows the radar punishment component, and the green shows them both combined. The evolutionary process stagnates after 51 generations.*



(a)



(b)

Figure 6.5 *A figure showing the standard deviations for the red agent's X, Y and Z coordinates, speed and course across 20 simulations of Scenario 2: Feint running in Docker containers. The figure shows standard deviations for simulation time scale factors of 50 (a) and 8 (b). Note that the y axes of the different figures are not the same scale.*

7 Discussion

This report describes our work on using an imitation learning algorithm to compose behaviour transition networks that control fighter aircraft in NGTS, and our experiences related to using the chosen methods and tools to solve the imitation learning problem throughout this work. This chapter discusses our findings when performing these experiments, especially the challenges we encountered while conducting the experiments described in chapter 6. We also aim to propose possible solutions to the problems we encountered.

The fitness functions and hyperparameters

While performing experiment 1, we added a radar punishment term to the fitness function in order to force the algorithm to add a *Put Sensor in Track* node. This could be seen as hard-coding the fitness function in order to get the expected result. However, in our experience, this might be unavoidable; a good fitness function needs to be finely tuned in order to yield good behaviours. It can also be argued that sensor usage is an important part of doctrine, and any good fitness function would need to consider the pilot's use of radar as a part of air combat tactics. Although, when using more complicated scenarios, extensive fine-tuning of fitness functions may prove challenging.

We also found that giving more weight to early parts of the demonstration trajectory than later parts helped the evolutionary algorithm learn more efficiently (see section 6.4). This was especially true for Scenario 3: *CAP*, in which both aircraft stayed near their initial positions, but the modified fitness function also worked optimizing BTNs for the *Feint* scenario. In our experience, adding a time dependence to the fitness function generally improved the evolutionary algorithm's ability to learn demonstrated behaviours. However, this cannot be said with certainty, as we did not try to seek out scenarios where this might not hold true.

We also found that hyperparameters had a large impact on how much time it took for the algorithm to find satisfactory BTNs, or whether such BTNs could be found at all. To our knowledge, there exists no analytical way to determine the best hyperparameter values, and so these must be set using guesswork, trial and error, grid search, or similar. Even when running simulations in a parallelized Kubernetes solution, BTN-NEAT spent many hours evolving a good behaviour. In order to find the best hyperparameters, it is necessary to run the full evolution process many times. This means that in order for a hyperparameter search to be feasible, the simulation system will need to run significantly faster than the NGTS-based system we used.

We also performed some initial tests of a semantic fitness function, as described in section 3.2.2.2. The trajectory fitness function used throughout this work is too dependent on both the initial conditions and the behaviour itself being exactly equal to the demonstration. We believe a fitness function that focuses on the decisions that affect the aircraft trajectory, rather than exactly where it is at a given time, has the potential to provide more appropriate fitness values, and can thus be used in a larger variety of scenarios. However, we could not get our initial semantic fitness function to work as expected and did not have enough time to experiment further. We think this kind of approach could be a better way forward than only employing a trajectory-based fitness function.

NGTS for machine learning purposes

Over the course of our work, we gradually realised that NGTS was not well suited for our use in machine learning. We had no access to any Application Programming Interfaces (APIs) or Software Development Kits (SDKs) and had to reverse engineer several of the commands used to remote control it. Additionally, without an API to directly control simulated entities, the only way to control behaviours was through generating BTNs.

NGTS has a network plugin to support HLA, but we could not get this to work, forcing us to use DIS. The biggest disadvantage of DIS was that there was no way to annotate data messages with the simulation time at which they were valid. This meant that lag in the system could lead to inconsistent fitness scores (see section 6.5).

Additionally, NGTS could not run directly under Linux and running it under Wine led to a substantial loss of performance. Even when running on a powerful Windows desktop computer, the maximal simulation time scale factor achievable was about 50. This is too slow for machine learning algorithms, where it might be necessary to run a scenario thousands or tens of thousands of times.

Nevertheless, this does not mean that NGTS *cannot* be used for machine learning purposes if given the right conditions, by which we mean having access to its API and SDK. As mentioned in section 2.5, previous work has applied NGTS in combination with dynamic scripting to learn behaviour scripts converted to BTNs [50]. Moreover, [59] uses NGTS to control opposing CGF using BTNs in the AGENT testbed, which was designed to evaluate the performance of AI behaviours for use with pilot training. NGTS contains a large collection of possibly suitable scenarios and behaviours. Besides, the system offers an efficient way to manually create new scenarios and agent behaviours using its built-in BTN design tool.

The machine learning method

Because of the aforementioned restrictions of NGTS, we were only able to control simulated entities using BTNs. Only a subset of machine learning methods, such as our modified BTN-NEAT algorithm, is suitable for generating BTNs. In contrast, the latest developments in reinforcement learning and imitation learning centre around deep neural networks. However, while we acknowledge that other suitable machine learning methods for evolving BTNs might exist, we did not have time to investigate such alternatives, particularly because testing several algorithms would have taken too much time given our current system with NGTS. Had we managed to reduce the optimization time further, a comparison between other machine learning methods for learning BTNs could have been made.

The system architecture

Our final system architecture was based on running both the machine learning algorithm and simulation systems in Docker containers in a Kubernetes cluster. This solution proved to be very portable, as we could easily move and run Docker containers on new hardware. It was also versatile and scalable: the number of simulations to run in parallel could be controlled by a single configuration variable, and the Kubernetes cluster would automatically start and control the correct amount of containers.

The drawback to using a system like this is that it requires a much larger amount of initial configuration than simply running systems manually on a single computer. It also requires a system architecture that is compatible with available containerization technology, meaning that a lot more consideration needs to be put into system design. Finally, maintaining and running Docker containers and Kubernetes clusters requires specialized knowledge.

Despite these drawbacks, we think the scalability and portability gained by making the system container-based outweigh the additional work required. We will almost certainly use Docker and Kubernetes in future ML projects.

8 Conclusion and future work

The simulation-based training of fighter pilots may become more efficient if the instructors and subject matter experts are relieved of having to control friendly and hostile players manually. This would instead allow them to focus more on the educational aspects of pilots in training. Artificially intelligent pilots may provide this function to the instructor if they act realistically concerning military doctrines, tactics, techniques, and operations. This report summarizes work on using the simulation system Next Generation Threat System combined with imitation learning to make agents learn to act in accordance with demonstrated air combat behaviour. The machine learning algorithm used was a modified version of the neuroevolution of augmenting topologies (NEAT) method, which we named BTN-NEAT.

The BTN-NEAT algorithm is an evolutionary algorithm, which creates a population of behaviours, evaluates the fitness of each individual in the population, and combines and mutates traits from the best-performing individuals to create a new generation of behaviours. As the goal was to imitate a demonstrated behaviour, an individual behaviour's fitness score depended on its ability to imitate the movements and sensor usage of the demonstrator aircraft. Each generated behaviour was simulated in NGTS in order to evaluate its fitness. Simulations were run in parallel across multiple CPUs using the Docker and Kubernetes virtualization technologies in order to reduce the time needed to find the optimal behaviour.

The BTN-NEAT algorithm managed to create behaviours that were judged to be sufficiently similar to the demonstration behaviours exhibited across three different scenarios. However, the algorithm spent more time finding satisfactory behaviour than expected. The scenarios also only involved simple one-versus-one engagements, and it remains to be seen how well BTN-NEAT will perform given more complex scenarios.

While conducting the experiments, we identified problems introduced by lag when running NGTS at high time scale factors. Combined with the lack of time management when logging aircraft data, this results in the timestamps captured not necessarily coinciding with the simulation time at which the data was valid. This was especially true when running NGTS at large time scale factors. Even an aircraft behaving identically to the demonstrated behaviour might be penalized when evaluated because the lag might cause the timestamps of its logged trajectory to differ from the actual simulation time. This finding forced us to reduce the simulation time scale factor to improve consistency, and reducing the simulation rate comes at the cost of increased optimization times. However, we recognize that some of these problems may be dealt with if one has access to the software development kit, application programming interface, and extensible documentation of NGTS.

Furthermore, the performance of BTN-NEAT, along with other machine learning algorithms, is highly dependent on the configured hyperparameter values. Estimating suitable hyperparameter values that could improve upon learning would involve an additional optimization process, which is impractical unless the simulation time scale factor is sufficiently large.

In future work, we will explore other simulation systems that allow us to express behaviours in ways different from BTNs, thus allowing us to explore other machine learning methods. We are especially interested in exploring imitation and reinforcement learning using deep neural networks. Preferably, we will develop a configurable, lightweight simulation system that can simulate air combat entities and other platforms at large simulation time scale factors. This simulation should also support time management and parallel execution using containers. Increased simulation speeds will allow us to integrate hyperparameter optimization methods as part of a new machine learning system.

References

[1] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[2] O. Vinyals *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[3] R. A. Løvlid *et al.*, "Data-driven Behavior modeling for computer generated forces," Forsvarets forskningsinstitutt, FFI-RAPPORT 17/01510, 2017.

[4] U. Dompke, "Computer Generated Forces - Background, Definition and Basic Technologies," Defense Technical Information Center, Tech. Rep., 2003. [Online]. Available: https://ntrl.ntis.gov/NTRL/dashboard/searchResults/titleDetail/ADA485518.xhtml

[5] European Defence Review, "Elbit Systems Delivers M-346 Simulators to the Polish Air Force," 2018, accessed December 2022. [Online]. Available: https://www.edrmagazine.eu/elbit-systems-delivers-m-346-simulators-to-the-polish-air-force

[6] R. M. Jones *et al.*, "Automated Intelligent Pilots for Combat Flight Simulation," *AI Magazine*, vol. 20, no. 1, p. 27, Mar. 1999. [Online]. Available: https://ojs.aaai.org/index.php/aimagazine/article/view/1438

[7] Naval Air Systems Command, "Next Generation Threat System (NGTS)," 2018, accessed: 2021-04-13. [Online]. Available: https://www.navair.navy.mil/nawctsd/sites/g/files/jejdrs596/files/2018-11/2018-ngts.pdf

[8] IEEE, *IEEE Standard for Distributed Interactive Simulation – Application Protocols*, 2012, IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995).

[9] NATO Standardization Agency (NSA), *Standardization Agreement (STANAG) 5516 Edition 4 – Tactical Data Exchange – Link 16*, 2008.

[10] NATO Standardization Agency (NSA), *Standardization Agreement (STANAG) 5518 Edition 1 – Interoperability Standard for Joint Range Extension Application Protocol (JREAP)*, 2014.

[11] D. Fu and R. Houlette, "Putting AI in entertainment: An AI authoring tool for simulation and games," *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 81–84, 2002.

[12] D. Fu *et al.*, "A Visual, Object-Oriented Approach to Simulation Behavior Authoring," in *Proceedings of the Industry/Interservice, Training, Simulation & Education Conference (I/ITSEC 2003)*, 2003.

[13] C. Cox and D. Fu, "AI for Automated Combatants in a Training Application," in *Proceedings of the second Australasian conference on Interactive entertainment*, 2005, pp. 57–64.

[14] D. Fu, R. Houlette, and J. Ludwig, "An AI Modeling Tool for Designers and Developers," in *2007 IEEE Aerospace Conference*. IEEE, 2007, pp. 1–9.

[15] A. Dosovitskiy and V. Koltun, "Learning to act by predicting the future," *arXiv preprint arXiv:1611.01779*, 2016. [Online]. Available: https://arxiv.org/abs/1611.01779

[16] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[17] T. Osa *et al.*, "An Algorithmic Perspective on Imitation Learning," *Foundations and Trends in Robotics*, vol. 7, no. 1-2, p. 1–179, 2018. [Online]. Available: http://dx.doi.org/10.1561/2300000053

[18] C. Sammut, *Behavioral Cloning*. Boston, MA: Springer US, 2010, pp. 93–97. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_69

[19] P. R. Gorton, "Backpropagating to the Future: Evaluating Predictive Deep Learning Models," Master's thesis, University of Oslo, 2020. [Online]. Available: https://www.duo.uio.no/handle/10852/78816

[20] R. Miikkulainen, *Topology of a Neural Network*. Boston, MA: Springer US, 2017, pp. 1281–1281. [Online]. Available: https://doi.org/10.1007/978-1-4899-7687-1_843

[21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[22] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*, 2nd ed. Springer Publishing Company, Incorporated, 2015.

[23] P. M. Pardalos and H. E. Romeijn, *Handbook of Global Optimization: Volume 2*. Springer Science & Business Media, 2013, vol. 62.

[24] A. E. Eiben and C. A. Schippers, "On Evolutionary Exploration and Exploitation," *Fundamenta Informaticae*, vol. 35, no. 1-4, pp. 35–50, 1998.

[25] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. [Online]. Available: http://nn.cs.utexas.edu/?stanley:ec02

[26] S. Risi and K. O. Stanley, "An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons," *Artificial life*, vol. 18, no. 4, pp. 331–363, 2012.

[27] A. Randal, "The ideal versus the real: Revisiting the history of virtual machines and containers," *ACM Computing Surveys*, vol. 53, no. 1, Feb 2020.

[28] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[29] S. J. Vaughan-Nichols, "What is Docker and why is it so darn popular?" Mar 2018, accessed: 2021-04-13. [Online]. Available: https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/

[30] A. Khan, "Key characteristics of a container orchestration platform to enable a modern application," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, 2017.

[31] Datadog, "11 Facts About Real-World Container Use," Nov 2020. [Online]. Available: https://www.datadoghq.com/container-report/

[32] O. Ben-Kiki, C. Evans, and I. döt Net, “YAML ain’t markup language (YAML) version 1.2,” Tech. Rep., October 2009. [Online]. Available: https://yaml.org/spec/1.2/spec.html

[33] D. McMahon, “A neural network trained to select aircraft maneuvers during air combat: a comparison of network and rule based performance,” in *1990 IJCNN International Joint Conference on Neural Networks*, 1990, pp. 107–112 vol.1.

[34] E. Y. Rodin and M. Amin, “Maneuver prediction in air combat via artificial neural networks,” *Computers and Mathematics with Applications*, vol. 24, no. 3, pp. 95–112, 1992.

[35] R. M. Jones *et al.*, “Intelligent automated agents for flight training simulators,” *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*, 1993. [Online]. Available: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA278641

[36] M. Tambe *et al.*, “Intelligent agents for interactive simulation environments,” *AI Magazine*, vol. 16, no. 1, pp. 15–39, 1995.

[37] C. Ryan, J. J. Collins, and M. O’Neill, “Grammatical evolution: Evolving programs for an arbitrary language,” in *Proceedings of the First European Workshop on Genetic Programming*, ser. LNCS, W. Banzhaf *et al.*, Eds., vol. 1391. Paris: Springer-Verlag, 14-15 Apr. 1998, pp. 83–96.

[38] J. Yao, Q. Huang, and W. Wang, “Adaptive CGFs Based on Grammatical Evolution,” *Mathematical Problems in Engineering*, vol. 2015, 2015.

[39] M. Colledanchise and P. Ögren, *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[40] N. Ernest *et al.*, “Learning of intelligent controllers for autonomous unmanned combat aerial vehicles by genetic cascading fuzzy methods,” *SAE Technical Papers*, vol. 2014-Septe, no. September, 2014.

[41] N. Ernest *et al.*, “Genetic fuzzy trees and their application towards autonomous training and control of a squadron of unmanned combat aerial vehicles,” *Unmanned Systems*, vol. 3, no. 3, pp. 185–204, 2015.

[42] N. Ernest and D. Carroll, “Genetic Fuzzy based Artificial Intelligence for Unmanned Combat Aerial Vehicle Control in Simulated Air Combat Missions,” *Journal of Defense Management*, vol. 06, no. 01, pp. 1–7, 2016.

[43] N. Ernest *et al.*, “Perspectives on Genetic Fuzzy Based Artificial Intelligence for Cooperative Control of Unmanned Fighter Aircraft,” no. July, 2017.

[44] R. A. L. Farzad Kamrani, Linus J. Luotsinen, “Learning objective agent behavior using a data-driven modeling approach,” in *Proceedings of 2016 IEEE International Conference on Systems, Man and Cybernetics (SMC)*. IEEE, 2016, pp. 2175–2181.

[45] G. Berthling-Hansen *et al.*, “Automating Behaviour Tree Generation for Simulating Troop Movements (Poster),” in *2018 IEEE Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA)*, 2018, pp. 147–153.

[46] P. Spronch *et al.*, “Adaptive game AI with dynamic scripting,” *Machine Learning*, vol. 63, pp. 217–248, 2006.

[47] A. Toubman *et al.*, “Rewarding Air Combat Behavior in Training Simulations,” *Proceedings - 2015 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2015*, vol. 0, pp. 1397–1402, 2016.

[48] A. Toubman *et al.*, “Transfer Learning of Air Combat Behavior,” *Proceedings - 2015 IEEE 14th International Conference on Machine Learning and Applications, ICMLA 2015*, pp. 226–231, 2016.

[49] A. Toubman *et al.*, “Rapid adaptation of air combat behaviour,” in *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, 2016, pp. 1791–1796.

[50] J. Ludwig and B. Presnell, “Developing an Adaptive Opponent for Tactical Training,” in *International Conference on Human-Computer Interaction*. Springer, 2019, pp. 532–541.

[51] V. François-Lavet *et al.*, “An introduction to deep reinforcement learning,” *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, p. 219–354, 2018. [Online]. Available: http://dx.doi.org/10.1561/2200000071

[52] J. Källström and F. Heintz, “Reinforcement Learning for Computer Generated Forces using Open-Source Software,” in *Interservice/Industry Training, Simulation, and Education Conference*, 2019, pp. 1–11.

[53] L. A. Zhang *et al.*, “Air Dominance Through Machine Learning: A Preliminary Exploration of Artificial Intelligence-Assisted Mission Planning,” RAND Corporation Santa Monica, Tech. Rep., 2020.

[54] Y. Chen *et al.*, “Design and Verification of UAV Maneuver Decision Simulation System Based on Deep Q-learning Network,” *16th IEEE International Conference on Control, Automation, Robotics and Vision, ICARCV 2020*, pp. 817–823, 2020.

[55] K. Zhou *et al.*, “Learning System for Air Combat Decision Inspired by Cognitive Mechanisms of the Brain,” *IEEE Access*, vol. 8, pp. 8129–8144, 2020.

[56] W. Kong, D. Zhou, and Z. Yang, “Air Combat Strategies Generation of CGF Based on MADDPG and Reward Shaping,” *Proceedings - 2020 International Conference on Computer Vision, Image and Deep Learning, CVIDL 2020*, no. Cvidl, pp. 651–655, 2020.

[57] E. Wiewiora, *Reward Shaping*. Boston, MA: Springer US, 2010, pp. 863–865. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_731

[58] A. P. Pope *et al.*, “Hierarchical Reinforcement Learning for Air-to-Air Combat,” 2021. [Online]. Available: http://arxiv.org/abs/2105.00990

[59] J. Freeman, E. Watz, and W. Bennett, *Adaptive Agents for Adaptive Tactical Training: The State of the Art and Emerging Requirements*. Springer International Publishing, 2019, vol. 11597 LNCS. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-22341-0_39

[60] R. Hefron, “Air Combat Evolution - DARPA,” accessed April 2021. [Online]. Available: https://www.darpa.mil/program/air-combat-evolution

[61] F. Wolfe, "Heron Systems Shows Future Potential of AI in Win Over Top F-16 Weapons Instructor," Avionics International, 8 2020, accessed April 2021. [Online]. Available: https://www.aviationtoday.com/2020/08/24/heron-systems-shows-future-potential-ai-win-top-f-16-weapons-instructor/

[62] A. McIntyre *et al.*, "neat-python." [Online]. Available: https://github.com/CodeReclaimers/neat-python

[63] R. M. Fujimoto, "DVEs: Introduction," in *Parallel and Distributed Simulation Systems*. New York, NY: John Wiley & Sons, Inc., 1999, ch. 7, pp. 195–221.

[64] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, feb 1966, doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[65] SISO, *Reference for Enumerations for Simulation Interoperability*, 2015, SISO-REF-010-2015.

[66] SISO, *Standard for Guidance, Rationale & Interoperability Modalities (GRIM) for the Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0*, 2015, SISO-STD-001-2015.

[67] SISO, *Standard for Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0*, 2015, SISO-STD-001.1-2015.

[68] Rancher Labs, "A Guide to Kubernetes with Rancher," 2019, accessed August 2021. [Online]. Available: https://more.suse.com/fy21-global-web-landing-page-guide-to-kubernetes

About FFI

The Norwegian Defence Research Establishment (FFI) was founded 11th of April 1946. It is organised as an administrative agency subordinate to the Ministry of Defence.

FFI's mission

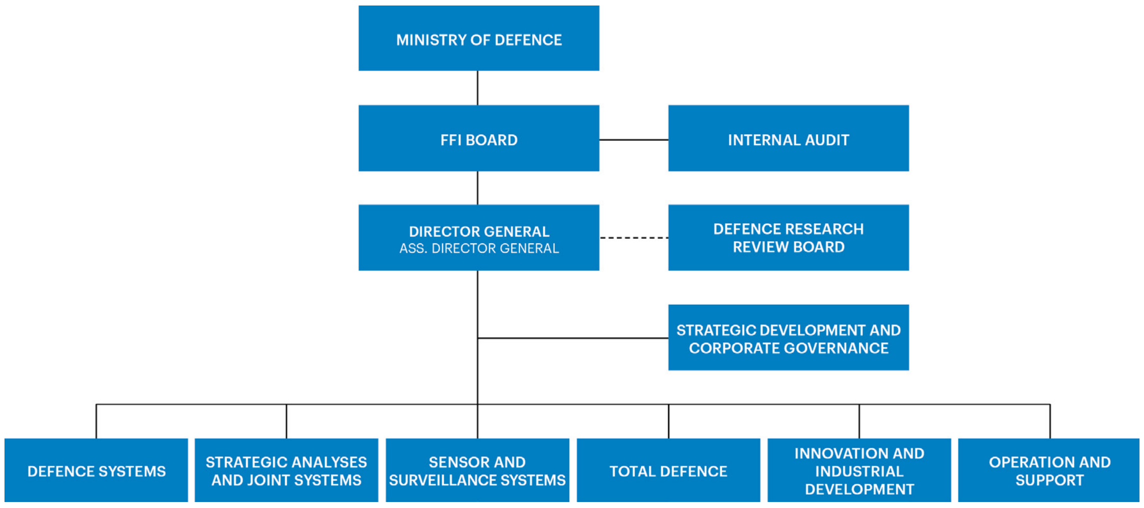
FFI is the prime institution responsible for defence related research in Norway. Its principal mission is to carry out research and development to meet the requirements of the Armed Forces. FFI has the role of chief adviser to the political and military leadership. In particular, the institute shall focus on aspects of the development in science and technology that can influence our security policy or defence planning.

FFI's vision

FFI turns knowledge and ideas into an efficient defence.

FFI's characteristics

Creative, daring, broad-minded and responsible.


MINISTRY OF DEFENCE
FFI BOARD
INTERNAL AUDIT
DIRECTOR GENERAL
ASS. DIRECTOR GENERAL
DEFENCE RESEARCH REVIEW BOARD
STRATEGIC DEVELOPMENT AND CORPORATE GOVERNANCE
DEFENCE SYSTEMS
STRATEGIC ANALYSES AND JOINT SYSTEMS
SENSOR AND SURVEILLANCE SYSTEMS
TOTAL DEFENCE
INNOVATION AND INDUSTRIAL DEVELOPMENT
OPERATION AND SUPPORT