Calculated Moves: Generating Air Combat Behaviour

Toubman, A.

2 Foundations

In this chapter, we will discuss four topics that are related to our research. First, we will have a detailed look at the steps of the behaviour modelling process (Section 2.1). Second, we will discuss the potential benefits and drawbacks of the use of machine learning on training simulations (Section 2.2). Third, we will describe the three categories of machine learning tasks (Section 2.3). Furthermore, we will take a closer look at reinforcement learning, and the dynamic scripting technique. Fourth, we will give an overview of past research on generating air combat behaviour with machine learning (Section 2.4). Finally, we will conclude the chapter by a summary (Section 2.5).

2.1 The steps in the behaviour modelling process

In this section, we describe the four steps of the behaviour modelling process in detail. The four steps were briefly mentioned in Chapter 1 (see Figure 1.1). The description of the steps below serves to create a better understanding of (1) how modern behaviour models are created, (2) the dependencies in the process that lead to the obstacles (see Subsection 1.1.1), and (3) the context in which a machine learning solution for behaviour generation would operate.

Step 1. The *training specialist* identifies training needs, resulting in a collection of learning objectives (cf. Stacy and Freeman, 2016). To help trainees reach these learning objectives in the simulator, the training specialist requires a CGF with certain behaviour to interact with the trainees. The training specialist writes a behaviour specification containing the required behaviour. As a case in point, consider a training specialist who has set a learning objective for successfully evading long-range missiles. For this learning objective, the training specialist requires a CGF that fires long-range missiles, and therefore writes "long-range missile firing behaviour" in the behaviour specification.

Step 2. The *subject matter expert* refines the behaviour specification that was written by the training specialist. The subject matter expert knows the behaviour of the real world forces represented by the CGF, such as (1) the technical capabilities of the CGF's aircraft, and (2)

the principal strategies and tactics of the represented real world forces (see, e.g., Løvlid, Alstad, Mevassvik, De Reus, Henderson et al., 2013). Returning to the example from Step 1, the subject matter expert refines the behaviour specification by, e.g., (1) defining the specific type of missile that the CGF should fire, and (2) adding any manoeuvres that the CGF should make before and after firing.

Step 3. The *programmer* takes the refined behaviour specification and implements the specification as a computer model. We call the resulting program the behaviour model. The behaviour model commonly takes one of three executable forms: (1) the form of a script (cf. Abdellaoui, Taylor and Parkinson, 2009; Toubman, Roessingh, Van Oijen, Hou, Luotsinen et al., 2016a), (2) a finite-state machine (cf. Fu, Houlette and Jensen, 2003; Coman and Muñoz-Avila, 2013), or (3) a behaviour tree (cf Khatami, Huibers and Roessingh, 2013; Marzinotto, Colledanchise, Smith and Ögren, 2014; Zhang, Sun, Jiao and Yin, 2017). When loaded in a simulator, the behaviour model governs the behaviour of the CGF. In other words, the behaviour model selects and implements the CGF's actions, based on the observations that the CGF makes in the simulator. However, simulators usually only provide CGFs with a limited set of possible observations and actions. It is the programmer's responsibility to interpret the behaviour specification, and then to accurately translate the specified behaviour into an executable form using the possible observations and actions as provided by the simulator. For example, consider (1) a behaviour specification that calls for a CGF that patrols a section of the airspace, and (2) a simulator that only provides actions by which a CGF can set its own altitude and heading. In this example, the programmer has to interpret the specified patrol and translate it to the correct sequence of altitude and heading settings.

Step 4. The *professionals* work together to validate the behaviour model. Multiple methods exist for validating behaviour models, each method with its own advantages and disadvantages (cf. Petty, 2010; Birta and Arbez, 2013). Validating the behaviour models commonly involves testing and reviewing the behaviour produced by the behaviour models, when used by CGFs. Each of the three professionals validates the behaviour model from their own viewpoint: (1) the training specialist determines whether the behaviour model can be adequately used to help trainees reach their learning objectives, (2) the subject matter expert decides whether the displayed behaviour matches the behaviour of real-life forces, and (3) the programmer establishes whether his[1] interpretation of the behaviour specification was correct with help from the training specialist and the subject matter expert. Finally, the professionals work out and implement improvements to the behaviour model. After any improvements have been implemented, the behaviour model is ready to be used by a CGF in a training simulation.

[1] For brevity, we use "he" and "his" whenever "he or she" and "his or her" are meant.

2.2 Machine learning in training simulations: potential benefits and drawbacks

Because machine learning is a powerful technology, it is important to carefully consider the impact it has on each application domain. In this section, we look at the possible impact of the use of machine learning on training simulations in terms of the potential benefits (Subsection 2.2.1) and the potential drawbacks (Subsection 2.2.2). Note that we purposefully do so before we describe machine learning itself (see Section 2.3). It enables us to confine the description to details necessary for a good understanding of our research.

2.2.1 Potential benefits

Below, we identify three potential benefits that appear when using machine learning to generate air combat behaviour models. They are: (1) faster development of behaviour models, (2) detection of patterns, and (3) online behaviour adaptation.

The first potential benefit is *faster development of behaviour models* (cf. Doyle, Watz and Portrey, 2015; Oswalt and Cooley, 2019). Computers excel at (1) storing and retrieving knowledge, and (2) calculation. Therefore, given (1) a database that stores the knowledge of a subject matter expert, and (2) a suitable machine learning algorithm, a computer may be able to automatically generate a correct behaviour model based on a behaviour specification (Stytz and Banks, 2003b). Automating the behaviour development process makes the process less dependent on the availability of two of the required professionals, i.e., the subject matter expert and the programmer. Furthermore, the ability to develop behaviour models at high speed enables the training specialist to support learning objectives with multiple and varied behaviour models.

The second potential benefit is the *detection of patterns* in behaviour. Faster development of behaviour models enables training specialists to see how trainees react to CGFs that behave in varying manners. Large data sets of these reactions allow for computer programs by which training specialists can detect patterns in the behaviour of trainees (e.g., areas of improvement for the trainees), and then adjust the current learning objectives to the needs of the trainees (cf. Mittal, Doyle and Watz, 2013; Sottilare, 2013; Ososky, Sottilare, Brawner, Long and Graesser, 2015; Oswalt and Cooley, 2019). Using data sets to improve training as described above is known as educational data mining, adaptive tutoring, and adaptive training (cf. Peña-Ayala, 2014; Goldberg, Davis, Riley and Boyce, 2017). Furthermore, large data sets of behaviour in simulations allow searching for exploitable patterns in the tactics that are taught to trainees, by the use of machine learning and big data techniques. This practice is known as computational red teaming (cf. Abbass, Bender, Gaidow and Whitbread, 2011; Wang, Shafi, Ng, Lokan and Abbass, 2017).

The third potential benefit is *online behaviour adaptation*. With faster development of behaviour models, it becomes possible for a computer to change the behaviour of CGFs while a training simulation is taking place (viz. online) (cf. Olde and DiCola, 2014; Oswalt and Cooley, 2019).

By adapting the behaviour of a CGF to the behaviour of the trainee in the training simulation, the CGF is able to continuously challenge the trainee (Lopes and Bidarra, 2011). A similar use of machine learning is being investigated in the field of video games (i.e., tuning the behaviour of non-player characters to the player, while the player is playing the video game) (cf. Yannakakis and Togelius, 2014), a field that is strongly tied to the field of military simulations (Smith, 2010).

In our research we will focus on materialising the first potential benefit. It will give a sufficient insight into the impact of machine learning on training simulations.

2.2.2 Potential drawbacks

Already in 2003, Petty (2003) identified the potential drawbacks of using automatically generated behaviour models in simulations for training, analysis, and experimentation purposes. Below, we summarise Petty's work into what we consider to be the two main drawbacks for training simulations, namely (1) the emergence of unrealistic behaviour, and (2) the resulting loss of training control.

Petty (2003) warns against the use of machine learning programs with the goal of automatically generating behaviour models for use in training simulations, as the models may produce unrealistic behaviour. The behaviour of a CGF that represents a particular force (e.g., a specific military branch of a specific nation) should accurately follow the doctrine of that force, in order to be perceived as realistic (cf. Doyle and Portrey, 2014; Bolton, Tucker, Priest, McLean, Beaubien et al., 2016). The doctrine of a military force is a "guide to action", viz. a handbook for conducting operations that describes how and when the different capabilities of the military should be put to use (see, e.g., Paparone, 2017). Petty uses the term *doctrinal behaviour* to refer to the behaviour of CGFs that appear to follow their doctrine.

If a CGF fails to follow its doctrine, but the CGF still acts as though it could be operated by a human, then the CGF exhibits what Petty (2003) calls *non-doctrinal behaviour*. Such behaviour may be the result of a machine learning program that is trying to optimise behaviour models regarding some constraints. Although non-doctrinal behaviour may be physically realistic, real-world forces are unlikely to display such behaviour. In extreme cases, the behaviour of the CGF may surpass the capabilities of a human operator and become *non-human behaviour*. An example of non-human behaviour is the display of inhumanly fast reaction times to threats.

The emergence of unrealistic behaviour (in the form of either non-doctrinal behaviour, or non-human behaviour) may lead to a loss of training control. As a machine learning program changes the behaviour of CGFs away from doctrinal behaviour, it becomes possible that the CGFs no longer support the training goals that the training specialist has set for a particular simulation. Thus, new tools are required to keep the creative power of machine learning techniques under a permanent check (cf. Shaffer, Ruis and Graesser, 2015; Wray, Woods, Haley and Folsom-Kovarik, 2017). In our research, we aim to mitigate the potential loss of training control by the development of a proper validation method for generated behaviour models (see Chapters 6 and 7).

2.3 Machine learning

So far, we have mentioned machine learning without explaining its details. In this section, we introduce the three categories of machine learning tasks (Subsection 2.3.1). Next, we take a detailed look at reinforcement learning, one of the categories (Subsection 2.3.2). Finally, we discuss dynamic scripting, a reinforcement technique (Subsection 2.3.3).

2.3.1 The three categories of machine learning tasks

Machine learning tasks are commonly split up into three categories: (1) supervised learning, (2) unsupervised learning, and (3) reinforcement learning (cf. Bishop, 2006; Alpaydin, 2010; Jordan and Mitchell, 2015). We briefly describe the three categories of machine learning tasks below. Here, we separate the tasks from the specific techniques that are used to perform the tasks. For instance, neural networks and deep learning techniques can be used to perform tasks in each of the three categories.

Category 1: Supervised learning tasks. Supervised learning tasks are tasks in which the computer has to learn a function that maps the inputs to the desired outputs. To learn this function, the computer receives a data set containing the function's inputs and desired outputs. The *classification* of data (for instance, credit card fraud detection) is an example of a supervised learning task (see, e.g., Dal Pozzolo et al., 2014). The computer is provided with a data set containing credit card transactions that are labelled by human experts as *fraudulent* or *regular*, and then learns a function to identify fraudulent transactions. When the computer is given new transactions, it can detect fraudulent transactions using the function it has learnt.

Category 2: Unsupervised learning tasks. Unsupervised learning tasks are tasks in which the computer has to create automatically a model of the data it receives. In unsupervised learning tasks, there is no desired output. Instead, it is the newly created model that is the output of interest. The model captures the structure of the data in ways that human experts may not have foreseen. An example of an unsupervised learning task is *clustering*. Clustering is the grouping of data points. The computer learns a model of the most important properties of the data points, and divides the data points into a number of groups according to these properties. The groups capture a hidden structure in the data that was given as input. Unsupervised learning tasks such as clustering are commonly part of exploratory data analysis (see, e.g. Peña-Ayala, 2014).

Category 3: Reinforcement learning tasks. Reinforcement learning tasks are tasks in which an agent (e.g., a program or robot) that has to learn to act in some environment. The environment provides *rewards* to the agent when the actions of the agent affect the environment in some desirable way. In most reinforcement learning tasks, the environment

is dynamic. For example, each action performed by the agent changes the environment in some way. As a result, (1) the future states of the environment, (2) the actions that are possible in these future states, and (3) the rewards that are obtained because of these possible actions all depend (at least partially) on the agent's current actions. Therefore, the computer can only learn which actions lead to the most rewards by actually interacting with its environment. The desired result of reinforcement learning is a sequence of actions that leads to the most rewards.

An example reinforcement learning task is learning to play the video game *StarCraft II* (Blizzard Entertainment, 2010). The goal of playing this game is to defeat the (virtual or human) opponent. The actions of the agent are, for instance, (a) creating buildings and troops, and (b) using troops to attack the buildings and troops of the opponent. It is up to the agent to learn what to build and how to properly instruct the troops to make their attacks (see, e.g., Lee, Tang, Zhang, Xu, Darrell et al., 2018).

Given the three categories of machine learning tasks, how should the task of generating air combat behaviour be approached? An important consideration is the availability of data to learn from. Real-world air combat data sets are difficult to obtain because of their military nature. Furthermore, if such a data set were available, it is likely that it would not contain sufficient examples for a machine to learn a generalised model of air combat behaviour from. Air combat situations are subject to what is known as the *curse of dimensionality* (originally introduced by Bellman, 1957; see also Roessingh, Rijken, Merk, Meiland, Huibers et al., 2011; Liu and Ma, 2017). Air combat has so many variables (e.g., the number of participating aircraft, their positions, headings, speeds) that it is unfeasible to enumerate and label all possible states. Additionally, the labelling of air combat behaviour as desirable or not remains an open problem (see Chapter 7).

Simulation technology enables us to create an air combat data set on demand. By treating the task of generating air combat behaviour models as a reinforcement learning task within a simulation, the reinforcement learning agent is able to gradually explore the state space in the search for desirable behaviour. We supply the agent with a simulated fighter jet and a restricted set of actions that the agent can perform at any moment (e.g., changing heading or firing a missile). This way, the agent's creativity is not bound to a limited set of real-world examples of behaviour, but still restricted to a particular set of actions that it can perform. In the next subsection, we discuss reinforcement learning in detail.

2.3.2 Reinforcement learning

Reinforcement learning is "learning what to do – how to map situations to actions – so as to maximize a numerical reward signal" (Sutton and Barto, 1998, chap. 1). Below, we briefly review the most important concepts in reinforcement learning. We base our review on the works by Sutton and Barto (1998), Heidrich-Meisner, Lauer, Igel and Riedmiller (2007), Alpaydin (2010),

Grondman, Busoniu, Lopes and Babuska (2012) and Arulkumaran, Deisenroth, Brundage and Bharath (2017).

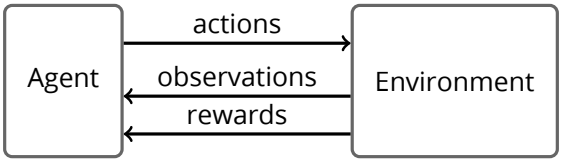

Figure 2.1 The reinforcement learning loop.

Descriptions of reinforcement learning commonly start by showing the reinforcement learning loop (see Figure 2.1). This loop shows the interaction between the agent and its environment. The agent performs an action. This action has some effect on the state of the environment. The environment provides a reward to the agent. Next, the agent observes the new state of the environment, and selects a new action to perform.

The agent selects its actions by means of its *policy* (i.e., its action plan). The policy is equivalent to what we so far have called the behaviour model. The agent learns by changing its policy in such a way that it can expect to receive more rewards in the future. The changes to the policy are guided by the rewards collected by the agent. However, the agent is never sure which specific action has lead to the rewards that the agent receives, as the reward may be the result of an earlier action. This is known as the *credit assignment problem*. The credit assignment problem is especially present in tasks where the reward is only presented to the agent after a sequence of actions (i.e., *delayed rewards*). Alpaydin (2010) summarises the use of rewards by noting that the agent is not dealing with a *teacher* that shows the agent how to act (such as in supervised learning), but rather with a *critic* that tells the agent how well it is doing. However, as Alpaydin (2010, chap. 18) notes, "the feedback from the critic is scarce and when it comes, it comes late."

The goal of collecting rewards presents a dilemma to the learning agent. Consider a game with two possible actions, action a and action b. Earlier, the agent performed action a, and then received a reward of $+50$. Therefore, action a is said to have a *value* of $+50$ to the agent. If the agent has not yet tried action b, that action will have a value of 0. The *state-action value* is the estimated reward the agent can expect to receive by performing that action in that state. The function that estimates the state-action values is called the *value function*. The agent's dilemma is as follows: Should the agent *exploit* action a to continue receiving rewards, or should it *explore* the use of action b, to see if action b leads to a higher reward than action a? Of course, by performing action b the agent also risks receiving (1) a lower reward or (2) no reward at all. Exploitation maximises the expected rewards in the short term, but exploration may yield more rewards in the long term (Sutton and Barto, 1998). Reinforcement learning methods that only allow the learning agent to exploit known action-values are called *greedy* methods. Alternatively, the agent can be allowed to explore some of the time. With small probability ϵ, the agent selects an action at random, rather than selecting the action with the highest known state-action value.

Methods that allow exploration in this manner are called *ϵ-greedy* methods.

Moreover, in reinforcement learning an important distinction is made between three classes of techniques: (1) actor-only techniques, (2) critic-only techniques, and (3) actor-critic techniques. Each class of techniques uses value functions and policies in different ways. We describe the three classes of techniques below.

First, *actor-only* (also called *policy-only*) techniques learn without a value function. Instead, they employ parametrised policies, and use any obtained rewards to follow a gradient descent over the parameters towards more rewards. Because of the parametrised policies, actor-only techniques are capable of learning policies in continuous action spaces. However, because of the use of gradient descent, information from earlier interactions with the environment is not retained (i.e., there is no real *learning* taking place).

Second, *critic-only* (also called *value-only*) techniques work by estimating the values of all states. Afterwards, the value function is used to derive the policy that is estimated to lead to the most reward. Critic-only methods learn optimal value functions, but the resulting policies are not guaranteed to be optimal (Grondman et al., 2012).

Third, *actor-critic* techniques try to combine the best features of critic-only and actor-only methods. In actor-critic techniques, the actor and the critic are modelled separately. Initially, the actor performs its actions based on some random policy. The critic detects the rewards that are received, and updates its value function. Next, the actor changes its policy based on the updated value function. The estimates of the critic lower the variance in the actor's policy updates. As a result, the combination of a separate actor and critic has two advantages: (1) increased learning speed and (2) good convergence properties compared to actor-only and critic-only techniques (Grondman et al., 2012). However, the trade-off is that the critic's estimates are inaccurate at the beginning of the learning process, when the critic has not yet seen many states and their values. Still, the two advantages have made actor-critic techniques popular in many domains.

In the last decade, an actor-critic reinforcement learning technique called *dynamic scripting* was introduced (Spronck et al., 2006). Two properties set dynamic scripting apart from other actor-critic techniques: (1) the behaviour models that it generates are accessible, and (2) by letting domain experts specify all relevant state-action pairs, the dynamic scripting is capable of counteracting the curse of dimensionality. In the next subsection, we further discuss the dynamic scripting technique.

2.3.3 Dynamic scripting

Dynamic scripting is a reinforcement learning technique (Spronck et al., 2006). Originally, dynamic scripting was designed as a behaviour generation technique for non-player characters in video games. The design of dynamic scripting was motivated by a discontent with the machine learning techniques that were available at the time (e.g., neural networks, evolutionary algorithms, and Q-learning). Specifically, the available machine learning techniques failed to fulfil eight

requirements that, according to Spronck et al. (2006, p. 219), were essential for maintaining the entertainment quality of video games. The eight requirements are as follows.

Requirement 1: Speed. Techniques must be computationally *fast*.

Requirement 2: Effectiveness. Techniques must be *effective* and produce adequate behaviour at all times.

Requirement 3: Robustness. Techniques must be *robust* against the inherent randomness in video games.

Requirement 4: Efficiency. Techniques must be *efficient* in learning, since each encounter between human players and non-player characters will most likely be different. Furthermore, the encounters are sparse, meaning there are few learning opportunities.

Requirement 5: Clarity. The policies that are generated must be easily *interpretable* by experts such as game developers.

Requirement 6: Variety. Techniques must be able to produce *variety* in the generated behaviour, to keep the video games entertaining.

Requirement 7: Consistency. Techniques must produce adequate behaviour from a limited number of learning opportunities with high *consistency*, independent of the behaviour of the player.

Requirement 8: Scalability. The technique must be able to *scale* the difficulty level of the generated behaviour to the skill level of the human player.

Of course, training simulations do not serve to provide entertainment. Still, there are three parallels between training simulations and video games that make the eight requirements relevant to training simulations as well. In both simulations and video games, a human participant (1) controls some avatar of themselves in a (somewhat realistic) representation of the world, and (2) encounters virtual agents that challenge the participant in some way. Furthermore, (3) the participant aims to reach some measurable goal: either setting high scores and completing levels (in video games), or honing and demonstrating his skills (in simulations). Therefore, it is important to investigate dynamic scripting as a technique for generating behaviour models for air combat CGFs.

Spronck et al. (2006) note that the key to fulfilling the eight requirements is the inclusion of domain knowledge in the machine learning technique. For this reason, they made predefined domain knowledge an integral part of dynamic scripting. Below, we review the workings of dynamic scripting, including the use of domain knowledge in the learning process.

The principal unit of behaviour in dynamic scripting is the *behaviour rule*. The policies generated by dynamic scripting are groups of behaviour rules. These groups are called *scripts*. We define behaviour rules and scripts below.

Listing 2.1 Example behaviour rule.

```
observe(radar(opponent)) → act(turn(180));
```

Definition 2.1 (Behaviour rule)**.** A behaviour rule is an *if-then* statement with an observation as the condition of the statement, and an action as the consequence of the statement.

Definition 2.2 (Script)**.** A script is a set of behaviour rules that is used as a policy.

A behaviour rule (henceforth: *rule*) directs an agent (e.g., a CGF) to behave in the following manner: *if* the agent makes the observation stated in the condition, *then* the agent takes the action stated in the consequence. Listing 2.1 shows an example behaviour rule. In Listing 2.1 and the remainder of the thesis, we use the two conventions for writing rules: (1) the condition and the consequence of a rule are separated by the arrow symbol →, and (2) rules end with a semicolon.

In normal words, the rule shown in Listing 2.1 means "if I observe the presence of an opponent using my radar, I turn around 180 degrees." Such a rule only constitutes a limited part of the behaviour that may be desired from a CGF inside simulations. Of course, more rules are required to provide behaviour that is applicable to other air combat situations. A script that only contains the rule shown in Listing 2.1 will cause a CGF to react only to opponents on its radar. However, other behaviour (e.g., offensive behaviour) may also be desired from the CGF.

The rules that dynamic scripting uses to form scripts are stored in a database called the rulebase. We define the rulebase below.

Definition 2.3 (Rulebase)**.** A rulebase is a database with (1) rules and (2) weight values associated to the rules.

The *weight value* (henceforth: *weight*) of each rule is akin to the state-action values that were mentioned previously (see Subsection 2.3.2). The weights of the rules in the rule base indicate the contribution of each rule towards desirable behaviour. Based on the rewards from the environment, dynamic scripting updates the weights of the rules that were used to obtain the rewards. Furthermore, the weight of each rule influences the probability that a rule is selected, whenever dynamic scripting generates a new script. This way, dynamic scripting learns which rules provide the behaviour that leads to the most rewards.

Additionally, each rule has an associated priority value. When two or more rules fire at the same time (e.g., because they have equal or logically overlapping conditions), the priority value is used to determine which rule takes precedence over the other rules. This way, it can be ensured that only the actions from one rule are executed.

The dynamic scripting learning process consists of three steps: (1) rule selection, (2) control,

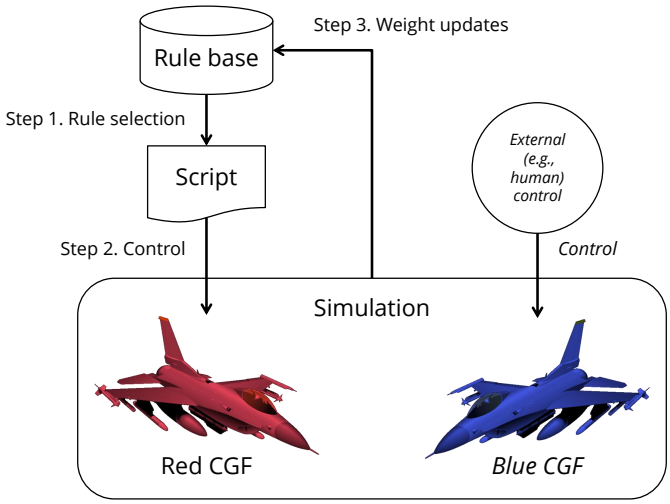

Figure 2.2 The three steps of the dynamic scripting learning process. Adapted from (Spronck, Ponsen, Sprinkhuizen-Kuyper and Postma, 2006).

and (3) weight updates, as shown in Figure 2.2. Below, we describe the three steps of the learning process.

First, dynamic scripting selects n rules from the rulebase. The selected rules form a script. The number of rules per script n depends on the domain (e.g., the number of possible states and actions, and how the states and actions are used in the rules). Therefore, n must be carefully chosen by a domain expert. The rules are selected from the rulebase based on their weights, by means of repeated *roulette wheel selection*. In roulette wheel selection, the probability of selecting a rule is equal to that rule's weight, divided by the sum of all weights in the rulebase.

Second, the script is used to control the behaviour of some agent in its environment. Here, we consider the case of a CGF that inhabits an air combat simulation. The CGF continuously observes its environment using its sensors (see Appendix A). The script checks whether the observations of the CGF match the conditions of one or more rules. Whenever the condition of a rule matches the observations of the CGF, the rule is said to *fire*, and the CGF performs the action that is defined by the rule.

Third, the behaviour of the CGF leads to updates of the weights in the rulebase. The weights are updated by means of two functions: (1) a fitness function, and (2) a weight adjustment function. The *fitness function* evaluates the behaviour that the CGF has displayed, and awards a *fitness value* (viz. a reward) to the CGF. In other words, the fitness function defines what behaviour is desirable (see Chapter 4). For example, the CGF might receive a fitness value of $+1$ if it completes some task, and a fitness value of -1 if it does not. The *weight adjustment function* takes the fitness value of the CGF, and then uses it to calculate the necessary adjustments to the

weights in the rulebase. Two mechanisms regulate the weight updates. These mechanisms are (1) restricting the growth of the weights in the rulebase, and (2) keeping the total sum of the weights in the rulebase constant. The growth of the weights is restricted by keeping each weight in the range $[W_{min}, W_{max}]$, where W_{min} and W_{max} are the minimum and maximum weights, respectively. Additionally, when the weights of certain rules must increase, they do so at the cost of the weights of the other rules in the rulebase (and vice versa). The constant *redistribution of weights* (1) allows the weights to converge to a set of well-performing (i.e., high-weight) rules, yet also (2) enables dynamic scripting to rapidly adapt to new situations. When a well-performing rule suddenly ceases to perform well, the weight that is taken from it is redistributed to other rules, thereby immediately increasing the probability that those other rules are selected for a new script.

An important feature that makes dynamic scripting stand out from other reinforcement learning techniques is the use of rules, and in particular, the origin of the rules. Spronck et al. intended for the rules in the rulebase to be manually written, based on domain knowledge. This way, the domain expert can define rules that make sense regarding the domain knowledge, and then let the dynamic scripting algorithm discover the combinations of rules (i.e., the scripts) that lead to the most desirable behaviour. On the nature of the rules, Spronck et al. (2006, p. 221) note that "it is imperative that the majority of the rules in the rulebase define effective, or at least sensible, agent behaviour."

The use of manually written rules can be viewed as both a drawback and an advantage. On one hand, writing the rules requires costly domain knowledge and human labour. On the other hand, rules only have to be written one time for each class of agent (e.g., CGFs that model a particular combination of pilot and fighter jet). Once the rules are stored in the rulebase, the rules can be used by each agent of that class. Furthermore, the rules are not edited by the dynamic scripting algorithm, and therefore remain accessible to the human professionals. Since the introduction of dynamic scripting, various methods have been introduced that automatically write behaviour rules (see, e.g., Thawonmas and Osaka, 2006; Ponsen, Spronck, Muñoz-Avila and Aha, 2007; Kanetsuki, Thawonmas and Nakata, 2015). While these methods have shown the capacity to generate effective rules (i.e., rules that lead to desirable behaviour), they also threaten the control that the professionals need to have over the resulting behaviour (see Subsection 2.2.2).

The combination of (1) rules and (2) domain knowledge makes dynamic scripting a versatile machine learning method. This versatility is shown by the diversity in the applications that can be found in the literature (see Table 2.1). Dynamic scripting has been used in multiple video game genres, each with distinctive features (e.g., real-time/turn-based, continuous moves/discrete moves, control of one or more agents). The demonstrated versatility of dynamic scripting provides a solid foundation for application in the air combat domain.

Table 2.1 A selection of dynamic scripting applications from the literature.

Application	Sources
Business simulation games	Bijlsma (2014)
Fighting games	Thawonmas and Osaka (2006), Kanetsuki, Thawonmas and Nakata (2015) and Majchrzak, Quadflieg and Rudolph (2015)
First-person shooter games	Policarpo, Urbano and Loureiro (2010)
Platform games	Ortega, Shaker, Togelius and Yannakakis (2013)
Real-time strategy games	Ponsen, Muñoz-Avila, Spronck and Aha (2005) and Dahlbom and Niklasson (2006)
Role-playing games	Spronck, Ponsen, Sprinkhuizen-Kuyper and Postma (2006), Timuri, Spronck and Van den Herik (2007) and Ludwig and Farley (2008)
Turn-based strategy games	Santoso and Supriana (2014)

2.4 Past approaches to generating air combat behaviour

The high stakes involved in air operations have invited multiple generations of computer scientists to support the training of fighter pilots by means of innovative machine learning programs. Furthermore, the complexity of the air combat domain (including the behaviour required of air combat CGFs), makes it an interesting application domain for machine learning algorithms. So far, past approaches to the generation of behaviour models for air combat CGFs has focused on neural networks (Subsection 2.4.1) and evolutionary algorithms (Subsection 2.4.2). Despite the continued interest in air combat behaviour modelling, we are unaware of any standardised tests or benchmarks for the performance of behaviour models for CGFs. The nearest example of such a test is a recent competition (Defense Advanced Research Projects Agency (DARPA), 2019) aimed at the creation of behaviour models for WVR air combat. Therefore, it remains difficult to assess the impact of each individual study performed in the air combat domain.

2.4.1 Neural networks

Neural networks have been applied in various forms to the generation of air combat behaviour. The strength of neural networks is the ability to emulate complex functions by learning from examples. Four of their weaknesses are (1) the need for long training phases, (2) the tendency to strongly converge toward a single solution, (3) trained neural networks are difficult to understand

and reason about, and (4) trained networks are practically impossible to manually edit. In other words, neural networks are powerful yet inaccessible. On multiple occasions, researchers have explored the potential of neural networks in the air combat domain. Below, we discuss four works that apply neural networks to air combat behaviour, viz. the works by (a) Rodin and Amin (1992), (b) McMahon (1990), (c) Teng, Tan and Teow (2013), and (d) Liu and Ma (2017).

Early work with neural networks includes the use of a three-layer back-propagation network by Rodin and Amin (1992) for predicting and countering within-visual-range tactical manoeuvres. With a single hidden layer, Rodin and Amin's network could not "satisfactorily distinguish" a set of simple one-versus-one manoeuvres from two-versus-one manoeuvres. Extensive testing of different architectures of the network resulted in a network with two hidden layers. This research exposes the third weakness of neural networks, which is the difficulty of reasoning about its construction. So far, trial and error has been the best way of finding optimal networks. Furthermore, Rodin and Amin report "successfully training" their network after 60,000 iterations.

Second, McMahon (1990) trained a neural network to recognise within-visual-range situations and choose appropriate manoeuvres. The neural network learned from examples. After 17,500 iterations, the network had learned to classify 36 out of 38 situations correctly. The network's classification capability was compared to that of a rule-based system containing expert knowledge. McMahon found that the neural network was able to classify situations correctly 2.5 times more often than the rule-based system. The high rate of correct classification was attributed to the generalising capability of neural networks. The capability to generalise enables neural networks to classify situations with noisy or incomplete data. Such situations are hard to classify for rule-based systems, unless the ability to deal with noisy or incomplete data is explicitly coded from the knowledge that is elicited from experts. Recently, research into new methods for the classification of air combat situations has been continued by Alford, Borck, Karneeb and Aha (2015).

Third, Teng et al. (2013) applied self-organising neural networks with a *Q*-learning component for online generation of within-visual-range behaviour. The resulting behaviour models were evaluated in small-scale human-in-the-loop experiments. The learning network was able to reach a 93% mean win rate after 120 episodes against a CGF with a fixed behaviour model. Furthermore, the network peaked at a 40% win rate against pilots in training, and below 10% against experienced pilots. Teng, Tan, Ong and Lee (2012) report using available air combat doctrine for building the state- and action-space for the *Q*-learning component by encoding expert knowledge as if-then rules.

Fourth, Liu and Ma (2017) applied deep reinforcement learning to generate air combat behaviour for an air combat agent. Deep reinforcement learning is a machine learning technique that combines (1) deep neural networks and (2) reinforcement learning. Deep neural networks are a class of neural networks that employ many layers of neurons (hence the term "deep"). The use of many layers allows the networks to not only learn (1) a mapping between input and output data, but also (2) their own feature detectors, by which the networks can adapt themselves to

the most important features in the input data. The combination of a deep neural network with a reinforcement learning technique, such as *Q*-learning, results in a form of machine learning that is known as deep reinforcement learning. In deep reinforcement learning, the deep neural network is used to approximate the value function for the reinforcement learning technique. This way, the network can use its adaptive feature detectors to learn the values of state-action pairs.

In the past years, deep reinforcement learning has been shown to be a versatile and powerful technique. Recently, a deep reinforcement learning agent called ALPHAGO ZERO has learned to play the game of Go purely by self-play, and then continued to repeatedly defeat a previous version of itself (Silver et al., 2017b). The previous version, known as ALPHAGO LEE, had earlier received acclaim for defeating a human world champion (Silver et al., 2016). Liu and Ma (2017) tested their air combat agent with deep reinforcement learning against another agent that used a minimax decision making algorithm. The rewards for the learning agent were based on (1) the relative positioning of the agents, and (2) the optimal firing range of the learning agent's weapon. In an experiment consisting of 100 encounters the agent learned to defeat its opponent nearly 60% of the time. These encounters took place after a training session consisting of 5000 encounters.

In the literature, we see neural networks applied in two ways: (1) since the 1990s as a model for partial control of a CGF's behaviour, such as situation recognition (McMahon, 1990; Rodin and Amin, 1992), and recently (2) as a model controlling the entire behaviour of a CGF (Teng et al., 2013; Liu and Ma, 2017). Because of the black box nature of neural networks, controlling only a part of behaviour increases the possibilities of complete validation of the resulting models. Below we briefly discuss models for partial control. The approach of partial control of behaviour using neural networks is advocated by, e.g., Henninger, Gonzalez, Georgiopoulos and DeMara (2000). A recent example of neural networks learning only a specific part of CGF behaviour is the work by Kamrani et al. (2016), in which CGF representing soldiers learn a troop movement pattern. The use of neural networks for partial control allows for completing complex sub-tasks (e.g., situation classification instead of "air combat" as a whole task) while limiting the effects of the inaccessibility of trained networks, since only the networks performance on the sub-task has to be explained and validated.

2.4.2 Evolutionary algorithms

A second type of algorithm that has been applied to the generation of air combat behaviour is the evolutionary algorithm. The strength of evolutionary algorithms is the ability to generate and try multiple creative solutions simultaneously. However, as is the case with neural networks, their weaknesses are (1) the need for extensive learning phases, and (2) the inaccessibility of the resulting models (depending on the specific evolutionary technique that is used). Below we discuss five lines of development in which evolutionary algorithms were applied to air combat behaviour, viz. the works by (a) Mulgund, Harper and Krishnakumar (1998), (b) Smith, Dike,

Mehra, Ravichandran and El-Fallah (2000a), (c) Kaneshige and Krishnakumar (2007), (d) Yao, Huang and Wang (2015), and (e) Koopmanschap, Hoogendoorn and Roessingh (2013).

Line (a): genetic algorithms. Mulgund et al. (1998) (continued by Mulgund, Harper and Zacharias, 2001) applied a genetic algorithm to find optimal formations for many-versus-many beyond-visual-range engagements. For this application, Mulgund et al. divided air combat tactics into three parts: (1) individual manoeuvres, (2) formations as a form of cooperation for small groups of aircraft, and (3) the use of individual manoeuvres and formations in large groups of aircraft. In their work, they focused on the latter as an optimisation problem, while using "conventional" individual tactics and small formations. Starting from a scenario with equal losses on both sides, the algorithm of Mulgund et al. was able to develop formations for as many as 16 aircraft. Using these large formations and the conventional tactics, all enemy CGFs were defeated without any defeats on the friendly side. While this is an impressive result, it only encompasses a small part of air combat behaviour generation. Furthermore, only a few parameters used by the algorithm are reported.

Line (b): learning classifier systems. Smith et al. (2000a) (see also the work by Smith, Dike, Ravichandran, El-Fallah and Mehra, 2000b) generated innovative one-versus-one within-visual-range behaviour for an experimental fighter jet using learning classifier systems (LCSs). The work by Smith and colleagues is a prime example of using evolutionary methods for the creative diversity of their solutions. It is explicitly stated that the goal of the study was the discovery of new behaviour, and not finding optimal behaviour. According to Smith et al., the automatic discovery of behaviour for a new aircraft allows simulation experts to give feedback to aircraft designers, customers, and operators about optimal ways to take advantage of the new aircraft's capabilities.

Line (c): artificial immune systems. An unconventional approach related to evolutionary algorithms was taken by Kaneshige and Krishnakumar (2007). The algorithm in this work was designed like an artificial immune system. The immune system selected manoeuvres (i.e., antibodies) to defeat detected intruders (i.e., antigens) in within-visual-range air combat. The parameters that were used were specifically chosen so that the algorithm was able to select manoeuvres within two seconds of calculation time. However, these parameters also appeared to limit the diversity of the solutions severely, as the algorithms quickly converged to the manoeuvres with the best performance.

Line (d): grammatical evolution. Yao et al. (2015) recently applied grammatical evolution to generate behaviour trees. At the core of their method was a genetic algorithm that operated on behaviour trees represented as bit-strings. The evolution of the behaviour trees was guided by a grammar. The grammar encoded three types of data: (1) the possible conditions and actions

that the behaviour tree could use, (2) the parameters of these conditions and actions, and (3) the structure by which the conditions and actions could appear in the behaviour tree. Use of the grammar served two purposes: (1) it guided the evolution, limiting the search space, and (2) it kept the resulting behaviour tree accessible to humans. However, even though the grammar served to limit the search space, there is no mention of any constraints placed on the creativity of the genetic algorithm. Yao and colleagues tested their method in a one-versus-one beyond-visual-range air combat simulation. In the best case, the agent using the grammatical evolution method learned to outperform its opponent after 60,000 simulations.

Line (e): optimising a cognitive model. Koopmanschap et al. (2013) optimised a cognitive model for air combat CGFs by means of an evolutionary algorithm (see also Koopmanschap, Hoogendoorn and Roessingh, 2015). This cognitive model took the form of a network that connected observations to beliefs. Starting with the observations made by a CGF, the network enabled the CGF to form beliefs about its situation. The CGF then selected its actions based on its beliefs. The cognitive model allowed for the modelling of human-like features in the CGF's reasoning process, such as (1) *situation awareness*, (2) *surprise*, and (3) *theory of mind* (Merk, 2013). The network used by Koopmanschap et al. was constructed by a human expert with domain knowledge. The evolutionary algorithm was used to optimise the connection strengths between the observations and beliefs in the cognitive model. This way, the evolutionary algorithm was able to determine how strongly (combinations of) observations contributed to the formulation of specific beliefs. The network was tested in simulations with air combat CGFs. Koopmanschap et al. assigned a high fitness to the CGF if it defeated an opposing CGF, and a low fitness if the CGF took out a friendly CGF. The evolutionary algorithm outperformed both a hill climbing algorithm and a random search strategy. Wilcke, Hoogendoorn and Roessingh (2014) built upon the work by Koopmanschap et al. by letting the evolutionary algorithm determine the connections between the observations and beliefs, rather than only the connection strengths.

As can be seen above, evolutionary algorithms may provide creative and interesting solutions to complex problems. The creativity of evolutionary algorithms can be a great asset in developing behaviour. However, two drawbacks are: (1) the learning process takes time, and (2) the creativity of evolutionary algorithms must be guided, to ensure no loss of training control occurs.

2.5 Chapter summary

In this chapter, we have provided background information on four topics.

First, we took a detailed look at the four steps of the behaviour modelling process (see Section 2.1). The behaviour modelling process is the process by which behaviour models for training simulations are created today. It is a lengthy process with interdependent steps.

Second, we discussed the potential benefits and drawbacks of using machine learning in training simulations (see Section 2.2). The potential benefits are: (1) faster development of

behaviour models compared to the behaviour modelling process, (2) the automatic detection of patterns in behaviour, and (3) online behaviour adaptation (see Subsection 2.2.1). The two potential drawbacks are: (1) emergence of unrealistic behaviour, and (2) the resulting loss of training control (see Subsection 2.2.2). In this thesis we focus on achieving the first potential benefit, and take care to avoid the two potential drawbacks.

Third, we introduced the three categories of machine learning tasks: (1) unsupervised learning tasks, (2) supervised learning tasks, and (3) reinforcement learning tasks (Section 2.3). Furthermore, we discussed the most important reinforcement learning concepts and reviewed the dynamic scripting reinforcement learning technique.

Finally, we reviewed past approaches to generating air combat behaviour models by means of machine learning (see Section 2.4). The two most commonly used techniques are (1) neural networks, and (2) evolutionary algorithms. However, these two techniques make it difficult to avoid the potential drawbacks of using machine learning in training simulations.