Computer Gaming: Foundations and Technologies of Artificial Intelligence

Fabien J. E. Zwick[a], B.Sc.

[a]Augsburg University of Applied Sciences, An der Hochschule 1, 86161 Augsburg, Germany

Abstract

AI in video games is a very important topic for both the games industry and academic research. There are many different applications of AI in both fields, but the focus of this paper is mainly on AI methods used to play games. First, the reasons why it makes sense to use AI to play games are discussed. The possible roles and goals the AI can have are here considered. Then a broad overview of some of the most commonly used AI methods is given. These include methods mostly used in academia, such as supervised learning, tree search and reinforcement algorithms, as well as some of the most commonly used methods in industry, such as finite state machines and behaviour trees. Then, some characteristics of games and algorithms are discussed that should be considered when developing and selecting an AI method. Last but not least, some examples are given of how the methods described can be used.

Keywords

Video Games, Artificial Intelligence, Foundations, Technologies, Augsburg

1. Introduction

Since their beginning in the 1950s, artificial intelligence has been a vital feature of video games. AI in games in general is a very big topic. Most people think of AI in video games primarily in terms of controlling non-player characters, so-called NPCs. But there is much more to AI in games. Besides controlling NPCs, AI can also be used for content creation and player modelling. Content creation includes things like procedural content generation of levels, assets or game rules, and in the case of player modelling, AI is used to construct a computation model of the player. In this paper, the focus is on the use of AI to play games. For playing games, there is still a gap between industrial and academically used AI. In the game development industry, most of the AI used belongs to expert-knowledge-systems. This kind of AI is to control NPCs through behaviours that are mostly scripted or designed in advance by the game designer. The designers mostly want complete control over the behaviour of the AI agent. Academic approaches like learning a policy through reinforcement learning would be usually be too unpredictable for this use case. Many papers dealing with AI methods are either very specific to the application of a particular AI method or deal with the methods in a very superficial way and do not focus on playing games, as in [1] and [2], but on the general use of AI. Therefore, the goal of this work is to give a good overview of the fundamentals of AI methods that can be used to play games, so that after reading this work, the reader knows which methods are best suited for a particular case and where to go for further reading about them.

The following section describes some reasons why it may make sense to use AI for games. Then the foundations of some of the most important AI methods are explained. These include methods that are used in industry as well as methods that are mainly used in academia. However, the focus here is primarily on the academic methods. Some methods are better, others less suitable for certain games. The characteristics to consider when choosing an AI method are discussed in the next section. And last but not least, examples for the application of the methods described are given in the last section.

2. Reasons to Use AI to Play Games

Artificial intelligence can improve games in a few different ways by simply playing them. Regardless of whether the basic AI depends on a basic behaviour tree, a utility-based AI or then again on the other hand, on a modern machine learned reactive controller is of restricted pertinence as long as it fills its needs. AI plays games with two main goals: 1) to play well (to win) and/or 2) to play believably (for experience). In addition, the AI can control either the player character or the non-player character (NPC) of the game. In the following, these objectives and the role as a player or NPC will be discussed in more detail.

Märtin, C. (ed.): Current Trends in Computer Science and Business Information Systems, Proc. of the M.Sc. Seminar, Summer Semester 2021, Augsburg University of Applied Sciences, April 2021

fabien.zwick@hs-augsburg.de (F. J. E. Zwick)

https://eobanz.github.io/ (F. J. E. Zwick)

2.1. Win in the Player Role

The task of an AI that takes on the role of a player in order to win is to achieve as much performance as possible while playing. Such an AI can be very useful when it comes to the (automatic) testing of the game or the evaluation of the game design. Through such an AI can be tested whether a level is solvable/playable at all. Another reason why AI is often used in the player role to win is because of research. In the academic field, games are often used as a test bed for AI algorithms. Many AI milestones have been achieved by beating the best human players in some games. Another reason for using an AI that tries to win in the player role is to provide a challenge for the players. For example, in classic games like chess or in strategy games where the AI can take the place of a real player like in the strategy game Civilization.

2.2. Win in a Non-player Role

An AI that tries to win in a non-player role can also be used to provide a challenge to the player. Just in a role that a real human can not take over. An example of this would be the enemies in XCOM: Enemy Unknown (2K Games, 2012). Sometimes an AI that is trying to win is needed to play in tandem with another AI that plays for experience. An example of this would be a racing AI that uses rubber-banding. The agents controlling the cars could use an AI that plays to win. Another AI can then control the speed of the cars to match the player, so they are never too far behind or in the lead.

2.3. Experience in the Player Role

An AI that does not try to win in the player role can be used to automatically test games. In this case, the AI tries to play as human-like as possible. As in an earlier point, it can also be used to test for example if a procedurally generated level is playable for a human player. In addition, an AI that plays for experience can detect bugs in the game that real players would also encounter. Another way to use AI is to demonstrate how a level or a certain section of a level can be mastered. In Super Mario Bros (Nintendo, 2006), for example, the player is shown how to complete a part of the level if he fails several times. The AI takes over the control for a few seconds and then lets the player continue.

2.4. Experience in a Non-player Role

An AI used in an NPC role to play for experience is the most commonly used type of AI in the industry. This includes very simple AI like the right-to-left walking of the Koopas in Super Mario Bros, as well as very complex AI behaviour like that of the alien in Alien Isolation. In this role, there is usually an attempt to create an illusion of intelligence, although the controlling code behind it is rather simple. Often human behaviour is not desired at all. In a stealth game, for example, you want the NPC to be somewhat predictable so that the player can remember the movements. Also, in some cases where you might think you can use an NPC AI that plays to win can sometimes lead to problems. This is because the way this AI plays can turn out to be boring or unsporting. For example, in FIFA 99, an algorithm has found that the best way to score a goal is to force a penalty kick. This is very effective, but it is no fun to play against such an AI.

3. Fundamentals of Algorithmic Approaches and AI Methods

Before the characteristics of games and algorithms and the application of AI methods can be discussed, some basics of AI methods and algorithms should be known. In the following, an overview of a selection of relevant AI methods used to play games is given. Since it is not possible to go into great detail here, additional literature for further reading is suggested for each method.

3.1. Ad-hoc Behaviour Authoring

This section covers some of the most widely used AI methods in the gaming industry. When talking about AI in video games in the context of game development, these ad-hoc behaviour authoring methods are usually meant. These methods have little to do with the academic meaning of Artificial Intelligence, as they belong to the expert-knowledge systems and are usually created in advance by a designer. They mostly do not use search or any form of learning and are primarily used to control NPCs.

3.1.1. Finite State Machines

Finite State Machines (FSMs) [3] were the most widely used AI method until the mid-2000s. FSMs belong to the expert-knowledge systems and are represented as graphs. An FSM is a computational model based on a hypothetical machine consisting of one or more states. Since only one state can be active at a time, the machine must switch from one state to the next in order to perform actions. In short, FSMs are defined by the following 3 components.

- A number of states which contain information, e.g., you are currently on the patrol state.
- A number of transitions between states described by a condition that must be met, for example, if you see an enemy or hear a gunshot, you move to the "alerted" state.

- A set of actions to be performed in each state, e.g. in patrol state, *move to random waypoints* and *search for enemies*

FSMs are very easy to design, implement, visualize and debug. For more advanced AI, however, they become too limited at some point because they can quickly become very complex. Another limitation of FSMs is that they are not very flexible and dynamic in their standard form. This creates a very predictable behaviour. To counter this, fuzzy rules [4] or probabilities [5] can be used, for example. Details on how to build FSMs can be found in [6].

3.1.2. Behaviour Trees

Behaviour Trees (BT) [7, 8, 9] are also expert-knowledge systems, and they also transition between a number of tasks (or behaviours). The strength of BTs in contrast to FSMs is their modularity. If they are well-designed, very complex behaviours can be created from simple tasks. The main difference between BTs and FSMs is that BTs are composed of behaviours instead of states. Like an FSM, a BT is very easy to design and debug, which is why they became a standard in the game development scene since their success in games like *Halo 2* and *Bioshock*. Behaviour trees use a tree structure with a root node and a number of corresponding child nodes representing the behaviour. A BT is traversed starting from the root node. The parent-child pairs are then executed as specified in the tree. A child node then returns one of the following values in a given time step: *success* if the behaviour is finished, *failure* if the behaviour failed, and *run* if the behaviour is still active. Behaviour Trees are composed of three node types: Sequence, Selector and Decorator. The sequence node succeeds when all its child behaviours succeed. Otherwise, the sequence node fails. There are two types of selector nodes. In the probability selector node, the child node is selected on the basis of a probability and succeeds if its child node also succeeds. In the priority selector, each child node is executed one after the other, and it succeeds as soon as the first child behaviour succeeds. Only if no child succeeds, the priority selector fails. The Decorate node can be used to increase the complexity and capacity of a single child behaviour. For example, this type of node can be used to specify how often a child behaviour is executed or to specify the amount of time the behaviour has to complete a task. Standard BTs are more flexible to design than FSM, but suffer from similar limitations as they are quite undynamic in their standard form. As with FSMs, there are ways to address this through extensions. For this, see [10, 11, 12]. More detailed information on how to create BTs can be found in [12].

3.1.3. Utility-Based AI

With utility-based AI, instances in the game get assigned utility functions that gives a value for the importance of the instance [13, 14]. For Example, the importance of low health when an enemy is present in a particular distance. Given all utility functions and options an agent has, it decides which is the most important option it should execute. A utility function can measure anything in a game. For example, enemy health, threat, emotions or mood. These utilities about possible actions can be aggregated into non-linear and linear formulas. With FSMs and BTs we only consider a decision at a time. In contrast, utility-based AI looks at all available options, gives them a utility value, and chooses the one that suits best. For further reading about utility-base AI [15] is a good starting point.

3.2. Tree Search

Most AI problems can be cast as a search problem, which can be solved by finding the best plan, path, model, function, etc. Search algorithms are therefore often seen as the core of AI. The following algorithms can be characterized as tree search algorithms, where the root node is the state where the search starts. The edges are the actions an agent can perform to move from one state to the next. Because an agent can usually perform multiple actions in a state, the tree branches.

3.2.1. Uninformed Search

Uninformed search algorithms are the algorithms that search the state space without having more information about the target. These algorithms include two algorithms that are fundamental in computer science: **Depth-first search** and **Breadth-first search**. The Depth-first algorithm explores each branch as far as possible before backtracking and trying another branch. When used to play a game, it explores the consequences of a single action until the game is won or lost and then explores the consequences of taking a different action close to the end state. The breadth-first algorithm does the opposite. Instead of exploring all consequences of a single action, it explores all actions of a single node before moving to a deeper node. More information about uninformed algorithms can be found in [16].

3.2.2. Best-First Search

In contrast to uninformed search algorithms, best-first search algorithms expand nodes based on information about the target state. In general, the nodes closest to a target state are expanded first. The best known algorithm of this kind is the A* algorithm. This keeps a list of "open" nodes which are next to an explored node but

which have not themselves been explored. For each open node, the distance to the target is estimated. New nodes to explore are chosen on a lowest cost basis, where the cost is the distance from the originating node plus the estimated distance to the destination. The A* algorithm is used in video games, mainly for pathfinding of AI NPCs. Details about different implementations and modifications of the algorithm can be found in the following sources: Real-time heuristic search [17], 3D pathfinding [18], navigation meshes [18, 19], grid-based [20]. Besides pathfinding, the A* algorithm can also be used to search in the space of games states, i.e. it can be used for planning instead of just for navigation. The difference here is that the changing state of the world is taken into account, not just the changing state of a single agent. Planning with A* can be very effective. For example, the winner of the 2009 Mario AI Competition used an A* planner that tried to get to the right side of the screen all the time [21]. More information and a good description of the A* algorithm can be found in [16].

3.2.3. Minimax

Uninformed and informed search algorithms are suitable for finding a path to an optimal state in a single-player game, but not for two-player games in which players play against each other. In this case, the actions of the two players depend on the actions of the opponent. An algorithm that can be used in such cases is **minimax**. This is well suited for classical perfect-information two-player board games like Checkers and Chess. With minimax, the algorithm constantly switches between the two players (called min and max player). For one player all possible actions are explored and then based on these nodes the actions that the opponent can perform as well. This is done until all possible combinations of actions have been explored and the game ends (e.g. with a win, a loose or a draw). At the end of this process, you get a whole game tree. A utility function is then applied to the leaves. This estimates how good the current game configuration is for a player. E.g. if the game is won, the utility function returns 1, if it is lost -1 and if it is a draw 0. Then, the algorithm traverses the tree to determine what action each player would have taken at a given state by returning values from leaves through the branching nodes. By doing so, it assumes that each player plays optimally. This means that from the point of view of the max player the score is maximized and from the point of view of the min player the score is minimized. The optimal winning strategy is reached for max if, on min's turn, a win can be achieved for max for all the moves min can make. This basic algorithm works with low complexity and action space. Exploring all actions and counter actions for a game of interesting complexity, the size of the search tree increases exponentially with the depth of the game or the number of moves that are simulated. For example the game tree size of tic-tac-toe is $9! = 362880$ which is feasible to traverse through. Chess on the other hand has approximately 10^{154} nodes which is infeasible to search through. Therefore, almost all applications of the minimax algorithm stop searching at a give depth, and use a state evaluation to evaluate the game state at that depth. A simple example for a state evaluation in chess would be counting the amount of black and white pieces. In addition, the basic minimax algorithm can be optimized by alpha-beta pruning, which requires fewer nodes to be explored.

3.2.4. Monte Carlo Tree Search

The Algorithms shown before will not play games well, that have a high branching factor. Also, there are games where it is hard to find good state evaluation functions that an algorithm like the Minmax can use to avoid exploring trees too deeply. An example of this is the game Go. In Go it is not so easy to tell during the game which player is winning. You can't just count the pieces to get a state evaluation function like in Chess. Also, Go has a 10x higher branching factor than chess (about 300). Another limitation of minimax is that it cannot be used for **imperfect information** games such as Battleship, Poker and/or **non-deterministic** games such as backgammon. For these problems, the **Monte Carlo Tree Search** (MCTS) [22] was invented in 2007. Instead of searching all branches of the search tree to an even depth, the MCTS concentrates on the more promising branches. This makes it possible to search certain branches quite deep, even though the branching factor is high. To address the problem with evaluation functions, determinism, and imperfect information, the standard formulation of MCTS is used to estimate the quality of the game state by randomly playing from a game state to the end of the game to see the expected outcome (win, loss, draw, or score). The core loop of MCTS algorithms can be divided into four steps. The first step **selection**. In this phase, a decision is made as to which node should be expanded. The starting point of this process is always the root of the tree and continues until a node is selected that has unexpanded children (i.e., represents a state from which actions can be performed that have not yet been attempted). To select the node from which to expand, a formula such as the UCB1 is used.

$$\text{UCB1} = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

where $\overline{X_j}$ is the average reward of all nodes beneath this node, C_p is an exploration constant, n is the number of times the parent node has been visited, and n_j is the number of times the child node j has been visited. In

addition to UCB1, there are several other options that include Thompson sampling, epsilon greedy, and Bayesian bandits. The second step is **expansion**. In this step a child of the currently selected node is expanded, i.e. an available action is executed. This is often done randomly. The next step is **simulation**. After a node is expanded, a simulation (roll-out) is done starting from that node until the end of game. Usually, random actions are taken until a terminal state is reached. The state at the end of the game is used as the reward for this simulation, and propagated up the search tree. The last step is **back propagation**. In this step, the reward of the simulation is added to the overall reward X of the new node. It is also added to the total reward of its parent nodes until the root of the tree. In cases where the simulation can be very long or will never enter an end state, a state evaluation function can be used like in the Minmax algorithm. More information about MCTS can be found in [23].

3.3. Evolutionary Algorithms

In contrast to tree search algorithms, evolutionary algorithms do not build a search tree based on the available actions, but only consider complete solutions and not the path to them. In evolutionary algorithms, a so-called **fitness function** is used to assign a numerical value to a solution. Using this function, the algorithm searches a search space for solutions that have the highest value for this function. The general idea of this type of algorithm is to optimize by "breeding" solutions: generate many solutions, dump bad ones, keep the good ones, and generate new solutions from them. The idea of keeping a population of the best solution and create new ones based on them is taken from Darwinian evolution by natural selection. A basic template for an evolutionary algorithm is as follows:

1. Initialization: The population is filled with randomly created solution.
2. Evaluation: Through the fitness function all solutions become a fitness value assigned
3. Parent selection: Based on the fitness value, the population members that will be used for reproduction are selected
4. Reproduction: New solutions are generated through crossover from parents or though copying parent solutions.
5. Variation: To some or all of the parents and/or offspring, mutation is applied
6. Replacement: The parents and/or offsprings are selected that will make it to the next generation. Popular replacement strategies are the generational (offspring replace the parents), elitism (best $x\%$ of parents survive) and steady state (replacement of worst parents if offspring is better) approaches.
7. Termination: Decide if termination condition is satisfied. E.g. a number of generations or evaluation have elapsed, a target fitness value is reached or/and some other termination condition
8. got to step 2

Every iteration of this loop is called a **generation**. This template can be expanded and implemented in a lot of different ways. There exist thousands of evolutionary-like algorithms, and many of them rearrange the overall flow and remove/add existing steps. Detailed information about evolutionary computation can be found in [24, 25, 26].

3.4. Supervised Learning

The algorithmic process of approximating the underlying function between labeled data and their corresponding attributes of features is called supervised learning [27]. It's called "supervised" because in order to learn it requires a set of supervised labeled training data. A simple example for this is a machine that distinguish between two objects. To train this model the algorithm needs features (inputs) of this object e.g. the size and colour and the corresponding labels (output). With these input-output pairs, a supervised learning algorithm can derive a function that approximates their relationship. This function should then be able to map well to new and unseen instances of input and output pairs. Some examples in the context of game AI would be the input-output pairs:

- {NPC health, player health, distance to player} -> {action(idle, flee, shoot)}
- {number of kills, ammo spent} -> {skill rating}
- {position of obstacles, self (NPC) and player} -> {action(move left, up, right or down)}

In this paper we only give an overview over Artificial Neural Networks because they are most commonly used for the purpose of game playing. Note that there are other supervised learning methods like Support Vector Machines and Decision Tree Learning.

3.4.1. Artificial Neural Networks

Artificial Neural Networks (ANN) are a very complex and mathematical topic. To go in detail would go beyond the scope of this paper. For this reason, only the most basic characteristics and one of the simplest types of ANNs will be discussed here. More precisely, a Multilayered Perceptron (MLP) is looked at without going too much into the mathematical details. The most important key point to take away from this section is that ANNs are universal function approximators [28]. ANNs are in essence simple mathematical models defining a function $f : \boldsymbol{x} \rightarrow \boldsymbol{y}$ Given sufficiently large ANN architectures and computational resources, they can approximate any continuous

real-valued function. A fundamental component of an ANN is a neuron. Each neuron has a number of **input** $\boldsymbol{x}$ each with an associated **weight** parameter $\boldsymbol{w}$, a bias b and an **output**. In addition, each neuron has a processing unit that combines the inputs with the associated weights via an inner product and adds a bias weight b to the weighted sum, as follows: $\boldsymbol{x} * \boldsymbol{w} + b$. This is then fed to an **activation function** g. The result of that function is the output of the neuron. The most commonly used function for ANN training is the sigmoid-shaped logistic function ($g(x) = 1/(1 + e^{-x})$)). One reason for this is that it can be used with gradient-based optimization algorithms like back propagation. Other activation functions for training deep architectures include the **rectifier** (ReLU) and **softplus** [29], because they allow faster training of deep ANNs, which are usually trained on large datasets. To build an ANN, the just described neurons need to be structured and connected. The most common way is to structure the neurons is in layers. The simplest form is the **Multilayer Perceptron** (MPL) architecture. In this architecture the neurons are layered across one or more layers but not connected to neurons in the same layer. The output of the neurons in one layer are the inputs for all the neurons in the next layer. The last layer is called the **output layer,** and all layers between the input and the last layer are known as **hidden layers**. The input of an ANN is connected to all neurons of the first layer. But this so-called **input layer** does not contain neurons, as it only distributes the inputs to the first layer of neurons. To calculate the output of the ANN, the output is calculated for each neuron until the output layer is reached. To calculate the output for each neuron j, this function is used: $\alpha_j = g(\Sigma_i\{w_{ij}\alpha_i\} + b_j)$. α_i is the input α_j is the output of the neuron. g is the activation function, w_{ij} is the connection weight from neuron i to neuron j and b_j is the bias of the neuron. For the ANN to approximate $f : \boldsymbol{x} \rightarrow \boldsymbol{y}$, the weights ($\boldsymbol{w}$ and $\boldsymbol{b}$) of the neurons need to be adjusted. For a training algorithm that does that, it requires two components: 1) a cost (error) function that evaluates the quality of any set of weights. 2) a search strategy within the space of possible solutions. The cost function is used to measure the MLP performance. The most commonly used performance measure for ANNs is the squared Euclidean distance (error) between the vectors of the desired labeled output and the actual output of the ANN. The calculated error is then used for backpropagation. It calculates weight updates that minimize the error function all the way from the output layer to the input layer. Without going in too much detail, backpropagation computes the partial derivative of the error function with respect to each weight and adjusts the weights of the ANN following the gradient that minimizes the error. More information about the mathematical background and backpropagation can be found in [30]. So the basics steps for each input-output pair of the backpropagation algorithm are as follows:

1. Present input pattern $\boldsymbol{x}$.
2. Compute the actual ANN outputs a_j.
3. Compute the error E.
4. Compute error derivates with respect to each weight and bias from the output to the input layer.
5. Update weights and bias using a learning rate and the errors derivatives

More details about artificial neural networks can be found in [31]. Information about deep architectures can be found in the deep learning book [29].

3.5. Reinfocement Learning

In **Reinforcement Learning**[32], unlike supervised learning, no labeled input-output pairs are required. An RL agent can learn just by the actions it performs in an environment and the rewards it receives for them. RL involves an **agent**, a set of **states**, and a set of **actions** per state. At a particular point in time, an agent is on a state, decides an action from all the available actions in the current state and gets an immediate **reward** from the environment for taking that action. The goal of the agent is to discover a **policy** (π) for selecting actions that maximize the long-term reward, such as the expected cumulative reward. The policy is then used by the agent for selecting actions, giving the state it is in. There are different types of RL algorithms like **Dynamic Programming**, **Monte Carlo methods** and **Temporal Difference learning** (TD) [32]. In this paper the focus is only on TD learning because the most popular RL algorithm for playing gmaes is the Q-Learning TD algorithm.

3.5.1. Q-Learning

The **Q-learning** [33] algorithm is a TD algorithm that relies on a tabular representation of $Q(s, a)$. $Q(s, a)$ represents how good it is to take the action a in state s. More precisely, it is the expected discounted reinforcement of executing the action a in state s. The agents learns from experience by receiving rewards for picking actions via bootstrapping. Bootstrapping means that it estimates how good a state is based on how good we expect the next state is. The Q-learning algorithm updates the Q values in the table in an iterative fashion. Every time the agent takes an action a in the state s, moves to the next state s' and receives an immediate reward r it updates its value for $Q(s, a)$ in the table like this:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\{r + \gamma \max_{a'} Q(s',a') - Q(s,a)\}$$

where α is the **learning rate**, and γ is the **discount factor** which are both values between 0 and 1. The learning rate is used to define, to what extent the new estimate

for Q will override the old estimate. The discount factor is used to weight the importance of earlier versus later rewards. Based on the Q-values in the table, the optimal policy can then be estimated. Is the agent in state s, it selects the action a with the highest $Q(s, a)$. Given a reward function r and a table of $Q(s, a)$ for each state and all possible actions, the basic loop for learning in the Q-learning algorithm are as follows:

1. Choose an action α based on a policy (e.q., ϵ-greed)
2. Execute the action, transit to state s', and receive an immediate reward r
3. Update the $Q(s, a)$ value in the table with the formula
4. $s \leftarrow s'$

Before starting the loop the Q-table has to be initialized with arbitrary Q values. The termination condition for this loop is either when the obtained policy is satisfactory or a certain number of iterations have been executed. During learning, an RL algorithm must also find a balance between exploration and exploitation. To make sure that the algorithm does not only execute the already known actions but also new unknown actions, the ϵ-greedy mechanism is often used. In short, the ϵ can be used to specify a probability of selecting a random action instead of the action with the highest reward. Further information about RL and Q-learing can be found in [32]. The Q-learning algorithm also has some limitations, which are mainly related to the tabular representation. Through this representation, large state-action pairs can become computationally very expensive. Furthermore, convergence can take a very long time, since the learning time is exponential to the size of the state-action space. One way to overcome these limitations is to use the hybrid algorithms described below.

3.6. Hybrid Algorithms

Many of the algorithms and methods shown until now can be interwoven in numerous ways. For example, you can use genetic algorithms to evolve finite state machines or behaviour trees or use an ANN estimator for tree pruning in MCTS. In this section, we will give a brief overview of the popular hybrid algorithms: **TD learning with ANN function Approximator** and **Neuroevolution**.

3.6.1. TD learning with ANN function Approximator

As mentioned earlier, RL typically uses tabular representations to store values and can be very expensive to calculate and store depending on the size of the state-action pairs. The most popular way to deal with this problem is to use an ANN as (Q-)value approximator and thus replace the table. The use of an ANN also allows handling continuous state spaces, which are infinitely large. Two very well-known examples that have used ANNs as approximators for TD learning are **TD-Gammon** and **deep Q network** (DQN)[34]. The former was used to play backgammon and the latter to play Atari 2600 games at a master level. It would be too much for this paper to go into detail how these algorithms work, but the key point is that the tables of RL algorithms can be replaced by training an ANN. For interested readers, DQN and TD-Gammon are covered in detail in [34] and [35].

3.6.2. Neuroevolution

With neuroevolution the design of an artificial neuronal network can be evolved through evolutionary algorithms. By design is meant their topology and their connections. This evolutionary reinforcement approach is useful, for example, when there are no target outputs or when the existing cost (error) function is not differentiable. In these cases, instead of back propagating the error, neuroevolution designs the ANN via evolutionary search. So neuroevolution does not require a dataset of input-output pairs to train like supervise learning does. It only requires a measure of the performance of the problem, like a score of the game, in that an agent is controlled by the ANN. The two typical approaches for neuroevolution are: 1) consider only the network's weight and 2) evolve the connection weights and the topology of the network (including activation functions and connection types). More information about Neuroevolution can be found in [36].

4. What to consider when choosing an AI method to play a game

When choosing a suitable AI method to play a game, it is important to first consider which methods are well suited. Some methods are more, others less suitable for certain types of games. In this section, we will look at the characteristics of games and algorithms that should be considered before choosing an AI method.
The first thing to consider is the **number of players** a game has. Some AI methods are better and some less suitable. Classical tree search algorithms such as depth-first, breath-first are particularly well suited for the single-player case (also for games with NPCs that are so simple and predictable that they can be seen as part of the environment) that have a low branching factor. For two-player zero-sum adversarial games, where there are exactly two players, the minimax algorithm described in 3.2.3 is particularly suitable. However, when the number of players increases, the algorithm is no longer suitable,

because for each move that is made, all counter moves of the opponents must be considered. Most of the time, multiplayer games are treated like single-player games in that some sort of model is used for what other players would do. This can be either be a learned model, a random one or the assumption that the other players will oppose the player. With a good model of what the opponents will do, many single-player methods can be used in multiplayer settings.

The next thing to consider is the **stochasticity**. Stochastic means that some processes cannot be predicted. In completely deterministic games like chess, it is possible to predict exactly what the state of the game will be after an action. In a game with stochasticity, if you perform the same action in a certain state, you may get different results. So the result of the game is not only dependent on the actions the player performs. Performing the same actions at the same time in several playthroughs of a game does not guarantee the same result. Many tree search algorithms have a problem with this. There are ways to modify the search algorithms to make them work, but as with the determinization modification for the MCTS where the possible outcomes are explored separately, the computational cost is increases significantly. In reinforcement learning, because of stochasticity, it is no longer possible to say for sure how good a policy is, since a good policy can sometimes produce bad results and a bad policy can sometimes produce good results. Stochasticity in RL sometimes leads to more robust algorithms, because no policies are learned for specific game states. For example, instead of attacking an opponent at a specific location at a specific time, it learns to deal with opponents coming from no specific direction at any time.

Another characteristic to consider is the **observability**. Observability refers to how much information about the state of the game is available to a player. In games like Chess or Go, all information about the board and the state is available to the player. Such games are called perfect information games. On the other hand, most video games have hidden information. For example, in Super Mario Bros, you only see a part of the game world. In strategy games, the commonly used term for this is *fog of war*. The same is true for classic games like poker, where players keep their hands of card private. In Such games are therefore only partially observable. When creating an AI for this type of game, the easiest approach is to simply ignore the hidden information. This can work well for some games like simple Super Mario levels. But as soon as complicated levels with backtracking come into play, this approach doesn't work anymore. So to develop an effective AI often some kind of modelling of the hidden information is needed.

Another characteristic that should be considered is the **Branching Factor and Action Space**. First, a few examples of branching factors in popular games. The mobile game Flappy Bird has a branching factor of 2, Pac-Man has 4, Super Mario Bros has 32, Chess has about 35 and Go has 400 in the very first move. In strategy games, where you have several units that have their own actions, the branching factor of the game is the product of the branching factors of each unit. I.e. if there are 6 units that can perform 10 actions, you get a branching factor for the game of one million. It gets worse when the input space is continuous, e.g. input from controllers and mice. The effectiveness of tree search algorithms depends very much on the branching factor. A high branching factor makes it impossible to look more than a few steps into the future. Even search algorithms that are better able to cope with larger branching factors than standard search algorithms such as MCTS will reach their limit at some point. High branching factors also can become a problem for reinforcement learning algorithms. If for the representation of a policy a function approximator like a neural network is used, it can be that for each action an output is needed. If the network is used to assign values to the actions, it must be iterated over all actions. In both cases, many possible actions can become a costly problem. Furthermore, as the number of possible actions increases, so does the time required to explore them during learning.

The next characteristic to look at is **time granularity**. Time granularity is about how many times a player can perform an action. Here, a fundamental distinction is made between real-time and turn-based games. In classic turn-based games like Chess, players take turns performing actions and the time that elapses between them is usually unimportant. In real-time games like modern fps, jump and runs or strategy games, the number of times an action can be executed often varies. Actions can be executed e.g. every render frame. However, the rendering frequency is not the same for every player. Two very contrasting examples in the context of time granularity are Chess and the game Star Craft. In chess, a game can be over after 40 actions. In Star Craft, just maybe only a second has passed after 40 actions. If a Star Craft game lasts 10 to 20 minutes, this means that thousands of actions have been executed in this time. So when using a tree search algorithm, the usefulness of a search at a certain depth is very dependent on the time granularity. For example, a chess game can be completely simulated with a search depth of 10. A search of this depth in Star Craft, on the other hand, will hardly be useful because the state of the game will hardly have changed. So a very deep search would have to be performed to deliver useful results, and this is mostly computationally infeasible. There are ways to address this problem like using macro-actions [37, 38].

Another important thing to think about when designing an AI that plays games is **how the game state is represented**. Games differ greatly in how they present

information to the player. Board games can be described by the positions of their characters, text adventures output text and most video games provide the video output along with sound. For each game there are different ways in which the information can be presented to an algorithm, and in which form this is done has a big influence on it. For example, if you want to use AI to control a car in a racing game, information about the current state of the game can be presented to the algorithm in different ways. It could get a 3d rendered first-person view through the windshield of the car, a 2d overview of the map showing all cars, or a set of angles and distances to other cars and the edges of the road. This decision about the input representation is very important when it comes to the design of an AI. Learning a policy that has as input three continuous variables for speed and distance to the edges of the road to the left and right of the car is comparably simple. Learning a policy that gets as input the whole unprocessed pixel image is much harder.

When designing an AI, another very important factor is whether you can simulate the game. More precisely, this so-called **forward model** is a model that when you are in a state s and execute the action a, you reach the same new state s' that the real game would reach. With such a model, it is possible to explore the results of different actions before they are executed in the real game. Without such a model, no tree search-based AI method can be used, since they are based on simulating the outcome of different actions. An important characteristic of a forward model to play a game in real time is that it is fast. The simulation should be at least a thousand times faster than real-time. If the computational complexity of the game loop is too high and therefore the forward model is too slow, a simplified model can be created, or an approximated model can be learned. In this case, of course, it cannot be guaranteed that the resulting states correspond exactly to the states of the real game. As mentioned before, tree search algorithms cannot be applied without a forward model. Supervised learning or reinforcement learning can still be used. To learn a RL agent, it should be possible to speed up the game, otherwise it can take a long time to learn the policy.

The last suggested aspect to consider is whether you have **time to train**. AI methods can be broadly distinguished between methods such as tree search algorithms, which examine actions and future states to decide which action to take, and machine learning algorithms, which learn a model over time to do so. In the context of games, there are algorithms that are designed in such a way that they do not need to learn about the game, but require a forward model (tree search). Others do not need a forward model and learn a policy instead (model-free reinforcement learning). And others need a forward model and time to learn (tree search with adaptive hyperparameters and model-based reinforcement learning).

5. Application of the AI methods

In section 3.1, some of the most important AI algorithms and methods have already been shown. In this section, we will briefly review how these can be used to play a game and give real-world examples. The most common methods in industry, shown in the section 3.1, such as FSMs and BTs, are not considered here, since they belong to the expert knowledge systems and their application, as already mentioned, usually don't apply artificial intelligence in an academic sense. The behaviour of the AI is determined in advance by a designer and is usually scripted.

One way AI can play games is through **Planning-Based** approaches. Among these methods are the tree search algorithms explained in 3.2. This type of method chooses actions by simulating future actions and therefore requires a forward model. Unless they are used for simple tasks like path finding in physical space. Classical informed and uninformed Tree Search includes algorithms like minimax, breath-first and depth-first. These algorithms are generally easy to use when the games have full observability, a low branching factor and a fast-forward model. Best-first algorithms like A* are usually used for path-planning, but can also be used to control the behaviour of an NPC. Here, instead of searching in physical-space, the search is done in state-space. A good example is the winner of the 2009 Mario AI Competition [21]. Robin Baumgarten used an A* algorithm in state-space, trying to get to the right edge of the screen.

For larger branching factors, a type of Monte Carlo Tree Search (MCTS) is best suited. MCTS overcomes the problems of standard tree search algorithms, such as high branching factor or the evaluation of the game state, by executing random rollouts until the end of the game and building the tree imbalanced. A well-known example that was built around the MCTS algorithm is AlphaGO [39]. AlphaGo managed to defeat some of the world's best Go players. This algorithm has also been successfully used in video games [40, 41, 42, 43].

Besides tree search, optimization algorithms such as evolutionary algorithms can also be used for planning. Here, the utility of a complete action sequence is evaluated. Since no tree has to be created, this approach is well suited for large branching factors. An evolutionary algorithm can be used for planning as follows. A plan is represented as a sequence of a number of actions (e.g. 10-20 actions). One can then use a standard evolutionary algorithm to obtain a satisfactory fitness value for a sequence using mutation, crossover, etc. or stop after a given number of iterations. The first action of the highest scoring sequence is then executed. In [44], an

evolutionary algorithm was used to choose actions in the turn-based game Hero Academy. In [45], Wang even showed that an evolutionary planner performed better than some variants of tree search algorithms in the game Star Craft.

Another possibility to let an AI play a game is through **reinforcement learning,** as already described in 3.5. As also described before, classical TD learning algorithms like Q-learning have a problem with their tabular representation. Apart from very simple games and some board games, interesting games have far too many possible states to use such a representation. Therefore, in such cases, a functions approximator like a neural network is often used. A major success in which an RL algorithm where a function approximator was used was in 2015 by Google DeepMind. The team trained deep neural networks that could play several games from the Atari 2600 console [34]. Each network was trained with raw pixels as input to play a single game. The output of the networks were the buttons of the console controllers. To train the networks, the deep Q networks method was used. Essentially this is standard Q-learning applied to neural networks with many layers (some of them were convolutional).

Another way to use reinforcement learning to play games is in combination with evolutionary algorithms. As described in 3.6, this so-called **neuroevolution** evolves the weights and/or topology of neural networks. In this type of algorithms, the network is used to play the game and the result (e.g. the score) is used as a fitness function. A disadvantage of these algorithms is that they have problems with very large input spaces that require large, deep neural networks. Like for example for pixel input. That is why most successful examples for this use case currently rely on deep Q-learning or similar methods. Neuroevolution algorithms have already been used in many game genres. For example, in first-person shooters [46], strategy games [47], real-time strategy games [48], car racing games [49, 50], and classic arcade games like Pac-Man [51].

As mentioned in 3.4, another method for AI to play games is through **supervised learning**. More precisely, policies or controllers can be learned through supervised learning that play the game. The fundamental thought here is to record gameplay of human players and train some function approximator like a neural network to act like the player. When the function approximator is trained well enough, the game can be played in the style of the human it was trained on by just executing whatever action the trained approximator returns when given the current game state. Alternatively, supervised learning can be used to learn the value of a game state and thus be used as a state evaluator in combination with the previously mentioned search algorithms.

AI methods can be interwoven. Some ways to do this like neuroevolution and deep q learning have already been mentioned in 3.6. Another example is dynamic scripting [52]. Here the importance of scripts is learned and adapted during the runtime by reinforcement learning. Dynamic scripting has been used for example in real-time strategy games [53] and fighting games [54] to adapt the AI to the skill level of the player. Another very well known example is AlphaGo. At its core, the agent uses an MCTS algorithm (planning). For the rollouts, however, a neural network is used to evaluate the state, and a position estimation network is used for node selection. Both of these networks were initially trained through recorded games between grand masters (supervised learning). Later, the networks were further trained by self-play (reinforcement learning) [39].

6. Summary

This paper gave a good overview of some of the most relevant AI methods for video games (mainly in the academic sense) and how they can be used. It also covered what should be considered when choosing a method. First, it explained in 2 what reasons there are for using AI to play games. The roles and goals that an AI agent can have were discussed here. After that, some of the most used AI methods and algorithms were presented. Here their basic concepts were covered and sources for more information were provided. It covered methods that are primarily used in the game development industry, but the focus was on the application of academic methods such as reinfocement learning, supervised learning and tree search. It has been shown that hybrid methods can significantly increase the performance and application areas of AI. After that, in 4, the characteristics of games and algorithms that should be considered when deciding on a method were discussed. In this section, it has become clear that the characteristics of a game have a great influence on which AI methods can be used and which are the most appropriate. Finally, the possibilities of applying the methods presented in 5 were addressed. With this knowledge, the reader should now be able to choose a suitable method for his or her specific application, or at least know where to look for more information.

References

[1] S. Risi, M. Preuss, From chess and atari to starcraft and beyond: How game ai is driving the world of ai, Springer, 2020.

[2] B. Xia, X. Ye, A. O. Abuassba, Recent research on ai in games, IEEE, 2020.

[3] A. Gill, Introduction to the theory of Finite-State Machines, 1962.

[4] M. Pirovano, The use of Fuzzy Logic for Artificial Intelligence in Games, Technical Report, University of Milano, Milano, 2012.

[5] A. J. Champandard, AI game development: Synthetic creatures with learning and reactive behaviors, New Riders, 2003.

[6] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines—a survey, volume 84 of *Proceedings of the IEEE*, IEEE, 1996, pp. 1090–1123. doi:10.1109/5.533956.

[7] A. J. Champandard, Behavior trees for next-gen game ai, in: Game Developers Conference, Audio Lecture, 2007.

[8] A. J. Champandard, Getting started with decision making and control systems, in: AI Game Programming Wisdom, volume 4, 2008, pp. 257–264.

[9] A. J. Champandard, Understanding behavior trees, AiGameDev.com, 2007.

[10] R. B. Chong-U Lim, S. Colton, Evolving behaviour trees for the commercial game, in: European Conference on the Applications of Evolutionary Computation, Springer, 2010, pp. 100–110.

[11] K. Dill, Introducing gaia: A reusable, extensible architecture for ai behavior, in: Proceedings of the 2012 Spring Simulation Interoperability Workshop, Springer, 2012.

[12] A. Shoulson, F. M. Garcia, R. M. Matthew Jones, N. I. Badler, Parameterizing behavior trees, in: International Conference on Motion in Games, Springer, 2011, pp. 144–155.

[13] B. Alexander, The beauty of response curves, in: AI Game Programming Wisdom, 2002, p. 78.

[14] K. Dill, A pattern-based approach to modular ai for games, in: Game Programming Gems, volume 8, 2010, pp. 232–243.

[15] D. Mark, Behavioral Mathematics for game AI, Charles River Media, 2009.

[16] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice-Hall, 1995.

[17] V. Bulitko, Y. Bjornsson, N. R. Sturtevant, R. Lawrence, Real-time heuristic search for pathfinding in video games, in: Artificial Intelligence for Computer Games, volume 8, Springer, 2011, pp. 1–30.

[18] D. Brewer, Tactical pathfinding on a navmesh, in: Game AI Pro: Collected Wisdom of Game AI Professionals, 2013, p. 361.

[19] P. Tozour, I. S. Austin, Building a near-optimal navigation mesh, in: AI Game Programming Wisdom, volume 1, 2002, pp. 298–304.

[20] S. Rabin, N. Sturtevant, Combining bounding boxes and jps to prune grid pathfinding, in: AAAI Conference on Artificial Intelligence, 2016.

[21] J. Togelius, S. Karakovskiy, R. Baumgarten, The 2009 mario ai competition, in: Evolutionary Computation (CEC), IEEE, 2010.

[22] R. Coulom, Efficient selectivity and backup operators in monte-carlo tree search, in: International Conference on Computers and Games, Springer, 2006, pp. 72–83.

[23] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of monte carlo tree search methods, in: Computational Intelligence and AI in Games, IEEE, 2012.

[24] A. E. Eiben, J. E. Smith, Introduction to Evolutionary Computing, Springer, 2003.

[25] D. Ashlock, Evolutionary computation for modeling and optimization, Springer, 2006.

[26] R. Poli, W. B. Langdon, N. F. McPhee, A field guide to genetic programming, lulu.com, 2008.

[27] C. M. Bishop, Pattern Recognition and Machine Learning, 2006.

[28] H. W. Kurt Hornik, Maxwell Stinchcombe, Multilayer feedforward networks are universal approximators, 1989.

[29] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.

[30] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning representations by back-propagating errors, Nature, 1986.

[31] S. Haykin, Neural Networks: A Comprehensive Foundation, Macmillian College Publishing Company Inc., 1998.

[32] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT Press, 1998.

[33] C. J. C. H. Watkins, P. Dayan, Q-learning, volume 8 of *Machine Learning*, 1992, pp. 279–292.

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep reinforcement learning, volume 518 of *Nature*, 2015, pp. 529–533.

[35] G. Tesauro, Temporal difference learning and td-gammon, volume 38 of *Communications of the ACM*, 1995, pp. 58–68.

[36] D. Floreano, P. Durr, C. Mattiussi, Neuroevolution: from architectures to learning, volume 1 of *Evolutionary Intelligence*, 2008, pp. 47–62.

[37] T. Pepels, M. H. M. Winands, M. Lanctot, Real-time monte carlo tree search in ms pac-man, volume 6 of *IEEE Transactions on Computational Intelligence and AI in Games*, 2014, pp. 245–257.

[38] D. Perez, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, S. M. Lucas, Solving the physical traveling salesman problem: Tree search and macro actions, volume 6 of *IEEE Transactions on Computational Intelligence and AI*

in Games, 2014, pp. 31–45.

[39] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, Mastering the game of go with deep neural networks and tree search, volume 529 of *Nature*, 2016, pp. 484–489.

[40] E. J. Jacobsen, R. Greve, J. Togelius, Monte mario: platforming with mcts, Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, 2014, pp. 293–300.

[41] N. Justesen, B. Tillman, J. Togelius, S. Risi, Script- and cluster-based uct for starcraft, in: Computational Intelligence and Games (CIG), IEEE, 2014.

[42] J. R. Quinlan, C4. 5: programs for machine learning, Elsevier, 2014.

[43] J. Fischer, N. Falsted, M. Vielwerth, J. Togelius, S. Risi, Monte-carlo tree search for simulated car racing, in: Proceedings of FDG, 2015.

[44] N. Justesen, T. Mahlmann, J. Togelius, Online evolution for multi-action adversarial games, in: European Conference on the Applications of Evolutionary Computation, 2016, pp. 590–603.

[45] C. Wang, P. Chen, Y. Li, C. Holmgard, J. Togelius, Portfolio online evolution in starcraft, in: Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference, 2016.

[46] M. Parker, B. D. Bryant, Visual control in quake ii with a cyclic controller, in: Computational Intelligence and Games, 2008, pp. 151–158.

[47] B. D. Bryant, R. Miikkulainen, Evolving stochastic controller networks for intelligent game agents, Evolutionary Computation, IEEE, 2006, pp. 1007–1014.

[48] K. O. Stanley, B. D. Bryant, R. Miikkulainen, Real-time neuroevolution in the nero video game, volume 9 of *Evolutionary Computation*, IEEE, 2005, pp. 653–668.

[49] J. Togelius, S. M. Lucas, Evolving controllers for simulated car racing., in: IEEE Congress on Evolutionary Computation, IEEE, 2005, pp. 1906–1913.

[50] D. Loiacono, P. L. Lanzi, J. Togelius, E. Onieva, D. A. Pelta, M. V. Butz, T. D. Lonneker, L. Cardamone, D. Perez, Y. Saez, M. Preuss, J. Quadflieg, The 2009 simulated car racing championship, volume 2 of *Computational Intelligence and AI in Games*, IEEE, 2010, pp. 131–147.

[51] S. M. Lucas, Evolving a neural network location evaluator to play ms. pac-man, in: Proceedings of the IEEE Symposium on Computational Intelligence and Games, 2005, pp. 203–210.

[52] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, E. Postma, Adaptive game ai with dynamic scripting, volume 63 of *Machine Learning*, 2006, pp. 217–248.

[53] A. Dahlbom, L. Niklasson, Goal-directed hierarchical dynamic scripting for rts games, in: AIIDE, 2006, pp. 21–28.

[54] K. Majchrzak, J. Quadflieg, , G. Rudolph, Advanced dynamic scripting for fighting game ai, in: International Conference on Entertainment Computing, 2015, pp. 86–99.