These are the slides accompanying the book Artificial Intelligence and Games through the gameaibook.org website.
About the Book

Welcome to the Artificial Intelligence and Games book. This book aims to be the first comprehensive textbook on the application and use of artificial intelligence (AI) in, and for, games. Our hope is that the book will be used by educators and students of graduate or advanced undergraduate courses on game AI as well as game AI practitioners at large.

Final Public Draft
The final draft of the book is available here!

Your readings from gameaibook.org

Chapter: 3
Making **computers** able to do things which currently only **humans** can do **in games**

AI as employed to games – A reminder from the intro lecture
Here is a non-inclusive list of things humans can do with games. What if AI could take these roles?

In this lecture we will focus on play!
As a reminder: “Play” is identified as one of the three major roles of AI in games in this book

- When we think of AI in games we think of an AI playing the game, or controlling an NPC. Maybe due to the associations between AI and autonomy, or between game characters and robots.
- Playing games is a very important role for AI and the role with the longest history.
- Many methods for content generation (Chapter 4) and player modeling (Chapter 5) are dependent on methods for playing games,
- Therefore it makes sense to cover play before content generation and player modeling.
So, let us focus on “play” and start by asking the question “why would one use AI to play games?”
Why use AI to Play Games?

- Playing to **win** vs playing for **experience**
  - For experience: human-like, “fun”, believable, predictable...?
- Playing in the **player role** vs. playing in a **non-player role**

[see Section 3.1 for more details]

The question can be reduced to two more specific questions:

- “Is the AI playing to win?”
- “Is the AI taking the role of a human player?”
AI could be playing a game to **win** or for the **experience of play** either by taking the role of the **player** or the role of a **non-player character**. This yields four core uses of AI for playing games as illustrated in the Figure. The figure illustrates the two possible **goals** (win, experience) AI can aim for and the two **roles** (player, non-player) AI can take in a gameplaying setting.

With these distinctions in mind, we will now look at each of the four key uses in more detail.
<table>
<thead>
<tr>
<th>Win</th>
<th>Player</th>
<th>Non-Player</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Motivation</strong></td>
<td>Playing roles that humans would not (want to) play, Game balancing</td>
</tr>
<tr>
<td></td>
<td>Games as AI testbeds, AI that challenges players, Simulation-based testing</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
<td>Rubber banding</td>
</tr>
<tr>
<td></td>
<td>Board Game AI (TD-Gammon, Chinook, Deep Blue, AlphaGo, Libratus), Jeopardy! (Watson), StarCraft</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Experience</th>
<th>Player</th>
<th>Non-Player</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Motivation</strong></td>
<td>Believable and human-like agents</td>
</tr>
<tr>
<td></td>
<td>Simulation-based testing, Game demonstrations</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Game Turing Tests (2kBot Prize/Mario), Persona Modelling</td>
<td>AI that: acts as an adversary, provides assistance, is emotively expressive, tells a story, ...</td>
</tr>
</tbody>
</table>

[see Section 3.1.1 for more details]
[see Section 3.1.2 for more details]
[see Section 3.1.3 for more details]

<table>
<thead>
<tr>
<th></th>
<th>Player</th>
<th>Non-Player</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Win</strong></td>
<td><strong>Motivation</strong>&lt;br&gt;Games as AI testbeds, AI that challenges players. Simulation-based testing</td>
<td><strong>Motivation</strong>&lt;br&gt;Playing roles that humans would not (want to) play. Game balancing</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong>&lt;br&gt;Board Game AI (TD-Gammon, Chinook, Deep Blue, AlphaGo, Libratus), Jeopardy! (Watson), StarCraft</td>
<td><strong>Examples</strong>&lt;br&gt;Rubber banding</td>
</tr>
<tr>
<td><strong>Experience</strong></td>
<td><strong>Motivation</strong>&lt;br&gt;Simulation-based testing, Game demonstrations</td>
<td><strong>Motivation</strong>&lt;br&gt;Believable and human-like agents</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong>&lt;br&gt;Game Turing Tests (2kBot Prize/Mario), Persona Modelling</td>
<td><strong>Examples</strong>&lt;br&gt;AI that: acts as an adversary, provides assistance, is emotively expressive, tells a story, ...</td>
</tr>
<tr>
<td>Win</td>
<td>Player</td>
<td>Non-Player</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Experience</strong></td>
<td><strong>Motivation</strong> Games as AI testbeds, AI that challenges players, Simulation-based testing</td>
<td><strong>Motivation</strong> Playing roles that humans would not (want to) play, Game balancing</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong> Board Game AI (TD-Gammon, Chinook, Deep Blue, AlphaGo, Libratus), Jeopardy! (Watson), StarCraft</td>
<td><strong>Examples</strong> Rubber banding</td>
</tr>
<tr>
<td></td>
<td><strong>Motivation</strong> Simulation-based testing, Game demonstrations</td>
<td><strong>Motivation</strong> Believable and human-like agents</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong> Game Turing Tests (2kBot Prize/Mario), Persona Modelling</td>
<td><strong>Examples</strong> AI that: acts as an adversary, provides assistance, is emotively expressive, tells a story, ...</td>
</tr>
</tbody>
</table>

[see Section 3.1.4 for more details]
Some Considerations

[see Section 3.2 for more details]
When choosing an AI method for playing a particular game it is crucial to know the characteristics of the game you are playing and the characteristics of the algorithms you are about to design. These collectively determine what type of algorithms can be effective.
We list a number of characteristics of games and we discuss the impact they have on the potential use of AI methods. All are tied to the design of the game but a few (e.g., input representation and forward model) are also dependent on the technical implementation of the game and possibly amenable to change. Much of our discussion is inspired by the book Characteristics of Games by Elias et al. which discusses many of these factors from a game design perspective.
Number of Players

- **Single-player** – e.g. puzzles and time-trial racing
- **One-and-a-half-player** – e.g. campaign mode of an FPS with nontrivial NPCs
- **Two-player** – e.g. Chess, Checkers and *Spacewar!*
- **Multi-player** – e.g. *League of Legends* (Riot Games, 2009), the *Mario Kart* (Nintendo, 1992–2014) series and the online modes of most FPS games.

[see Section 3.2.1.1 for more details]
Stochasticity

- The degree of randomness in the game
- Does the game violate the Markov property?
  - **Deterministic** (e.g. Pac-Man, Go, Atari 2600 games)
  - **Non-deterministic** (e.g. Ms Pac-Man, StarCraft, ...)

[see Section 3.2.1.2 for more details]
• How much does our agent know about the game?
  • **Perfect** Information (e.g. *Zork, Colossal Cave Adventurer*)
  • Imperfect (**hidden**)
    Information (e.g. *Halo, Super Mario Bros*)

[see Section 3.2.1.3 for more details]
Action Space and Branching Factor

- How many actions are there available for the player?
  - From two (e.g. Flappy Bird) to many (e.g. StarCraft)....

[see Section 3.2.1.4 for more details]
Time Granularity

- How many turns (or ticks) until the end (of a session)?
  - Turn-based (e.g. Chess)
  - Real-time (e.g. StarCraft)

[see Section 3.2.1.5 for more details]
For illustrative purposes, this Figure places a number of core game examples onto the three-dimensional space of observability, stochasticity and time granularity. Note that the game examples presented are sorted by complexity (action space and branching factor) within each cube. Minimax can theoretically solve merely any deterministic, turn-based game of perfect information (red cube in the figure)—in practice, it is still impossible to solve games with substantially large branching factors and action spaces such as Go via Minimax. Any AI method that eventually approximates the Minimax tree (e.g., MCTS) can be used to tackle imperfect information, non-determinism and real-time decision making (see blue cubes in figure).
Characteristics of Games: Some Examples

Chess
- Two-player adversarial, deterministic, fully observable, bf ~35, ~70 turns

Go
- Two-player adversarial, deterministic, fully observable, bf ~350, ~150 turns

Backgammon
- Two-player adversarial, stochastic, fully observable, bf ~250, ~55 turns

Some board game examples
Characteristics of Games: Some Examples

**Frogger (Atari 2600)**
- 1 player, deterministic, fully observable, \( bf \) 6, hundreds of ticks

**Montezuma's revenge (Atari 2600)**
- 1 player, deterministic, partially observable, \( bf \) 6, tens of thousands of ticks

Arcade game examples
AAA digital game examples

Imagine having 6 different units that can each take 10 different actions at a given time—a rather conservative estimate compared to typical games of, say, StarCraft (Blizzard Entertainment, 1998) or Civilization (MicroProse, 1991)—then your branching factor is a million!
Key questions

- How is the game state represented?
- Is there a forward model available?
- Do you have time to train?
- How many games are you playing?

[see Section 3.2.2 for more details]

Answers to these four questions lead to core algorithmic decisions. We will look at each one of them in detail
Game State Representation

- **Games differ with regard to their output**
  - Text adventures → Text
  - Board games → Positions of board pieces
  - Graphical video games → Moving graphics and/or sound
- **The same game can be represented in different ways!**
  - The representation matters greatly to an algorithm playing the game
- **Example: Representing a racing game**
  - First-person view out of the windscreen of the car rendered in 3D
  - Overhead view of the track rendering the track and various cars in 2D.
  - List of positions and velocities of all cars (along with a model of the track)
  - Set of angles and distances to other cars (and track edges)

[see Section 3.2.2.1 for more details]
A forward model is a simulator of the game
- Given $s$ and $\alpha \rightarrow s'$
- Is the model fast? Is it accurate?
- **Tree search** is applicable *only* when a forward model is available!

[see Section 3.2.2.2 for more details]

A very important factor when selecting an AI algorithm to play a game is whether there is a simulator of the game, a so-called **forward model**, available.

For many games, however, it is impossible or at least very hard to obtain a fast forward model.
In some cases the computational complexity of the core game loop might still be so high that any forward models built on the core game code would be too slow to be usable.

In some of such cases, it might be practical to build and/or learn a simplified or approximate forward model, where the state resulting from a series of actions taken in the forward model is not guaranteed to be identical to the state resulting from the same series of actions in the actual game.
Life without a forward model...

- Sad...!
- We could learn a direct mapping from state to action
- Or some kind of forward model
- Even a simple forward model could be useful for shallow searches, if combined with a state value function
What type of algorithm you will want to use depends largely on your motivation for using AI to play games. If you are using the game as a testbed for your AI algorithm, your choice will be dictated by the type of algorithm you are testing. If you are using the AI to enable player experience in a game that you develop then you will probably not want the AI to perform any learning while the game is being played, as this risks interfering with the gameplay as designed by the designer. In other cases you are looking for an algorithm that can play some range of games well, and do not have time to retrain the agent for each game.

[see Section 3.2.2.3 for more details]
General game playing is typically motivated by a desire to use games to progress towards artificial general intelligence, i.e., developing AI that is not only good at one thing but at many different things.

Frameworks for general game playing:
- General Game Playing Competition
- General Video Game AI Competition
- Arcade Learning Environment

[see Section 3.2.2.4 for more details]
Problem: Overfitting!
Can we construct AI that can play many games?
How Can AI Play Games?

- Different methods are suitable, depending on:
  - The characteristics of the game
  - How you apply AI to the game
  - Why you want to make a game-playing
- There is no single best method (duh!)
- Often, hybrid (chimeric) architectures do best

[see Section 3.3 for more details]
“Surely, deep RL is the best algorithm for playing games...”

Given the success of deep reinforcement learning in Atari games and Go (predominately by Deepmind) one might conclude that it is the best algorithms available out there
It is however just an opinion – let’s see why
Here is an example of a AI playing Super Mario Bros. This is the winner entry of Robin Baumgarten (Imperial College at the time) that won the 2012 Mario AI competition. Surprisingly enough this is merely an A*-controlled agent which at any point simply tries to get to the right edge of the screen.

https://www.youtube.com/watch?v=DIkMs4ZHHR8
Here is an overview of methods that can be used to play games (and their corresponding requirements).

- Some algorithms do not need to learn anything about the game, but do need a forward model (tree search)
- Some algorithms do not need a forward model, but instead learn a policy as a mapping from state(s) to action (model-free reinforcement learning)
- And some algorithms require both a forward model and training time (model-based reinforcement learning and tree search with adaptive hyperparameters).

Let us start with the assumption that a forward model is available and cover the planning based methods first
[see Section 3.3.1 for more details]
Algorithms that select actions through planning a set of future actions in a state space are generally applicable to games, and do not in general require any training time. They do require a fast forward model if searching in the game’s state space, but not if simply using them for searching in the physical space (path-planning).

Tree search algorithms are widely used to play games, either on their own or in supporting roles in game-playing agent architectures.
A different taxonomy of the same algorithms [see Section 3.3.1 for more details]
- Algorithms that require a forward model appear in **green**
- Algorithms that do not require a forward model appear in **red**

- Planning-Based
  - **Classic Tree Search** (e.g. best-first, breadth-first, A*, Minimax)
  - **Stochastic Tree Search** (e.g. MCTS)
  - **Evolutionary Planning** (e.g. rolling horizon)
  - **Planning with Symbolic Representations** (e.g. STRIPS)
Classic tree search methods, which feature little or no randomness, (Minimax and α-β pruning) have been used in game-playing roles since the very beginning of research on AI and games.

In general, classic tree search methods can be applied in games that feature
- full observability
- low branching factor
- fast forward model

Theoretically they can solve any deterministic game that features full observability for the player; in practice, they still fail in games containing large state spaces.
In the following slides we will focus on a popular algorithm of informed search (and classic tree search) in games: \textit{A*}

Path-planning and A*: Best-first search, in particular a myriad variations of the A* algorithm, is very commonly used for \textbf{path-planning} in modern video games. When an NPC decides how to get from point A to point B, this is typically done using some version of A*. In such case no forward model is required.

NPC control: best-first algorithms such as A* can be used for controlling all aspects of NPC behaviour. To take an example, the winner of the 2009 Mario AI Competition was entirely based on A* search in state space [see Figure]
Let us now see how A* worked in Super Mario Bros in more detail. Further details can be found in section 3.3.1.1 and in the following paper:

*Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 Mario AI competition. In Evolutionary Computation (CEC), 2010 IEEE Congress on. IEEE, 2010*

The figures in the following slides illustrate the key steps of A* search for playing the game. Super Mario’s goal (heuristic to be maximised) is to reach the right boarder of the screen.
The agent considers a maximum of nine possible actions at each frame of the game, as a result of combining the jump and speed buttons with moving right or left.
Then the agent picks the action with the highest heuristic value.
But this action gets a high negative reward as an enemy is threatening Mario.
Hence, Mario takes the action second highest heuristic value (i.e., right, jump, speed in this example),
Then Mario takes the next move to a new state
A* in Mario: Create Child Nodes

From this new state Mario considers all new possible action states
And finally Mario evaluates the new action space by considering “unexplored” states in the stack
Potential question for the class

Potential answers:

- Super Mario Bros is **deterministic**
- The game has locally **perfect information** (at any instant the information in the current screen is completely known)
- A good **forward model is available**: if the A* would not have used a complete model of the game including the movement of enemies, it would have been impossible to plan paths around these enemies.
- **Levels** are fairly **linear**; in a later edition of the competition, levels with dead ends which required back-tracking were introduced, which defeated the pure A* agent (see next slide).
An example level generated for the 2010 Mario AI Competition (a year after Robin’s agent won).

Note the overhanging structure in the middle, creating a dead end for Mario; if he chooses to go beneath the overhanging platform, he will need to backtrack to the start of the platform and take the upper route instead after discovering the wall at the end of the structure. Agents based on simple A* search are unable to do this.
[see Section 3.3.1.2 for more details]

Stochastic tree search refers primarily to Monte Carlo Tree search and its myriad variants
Monte Carlo Tree Search

- The best new tree search algorithm you hopefully already know about
- When invented, revolutionized computer Go

<table>
<thead>
<tr>
<th>Year</th>
<th>Program</th>
<th>Description</th>
<th>Elo</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006</td>
<td>INDIGO</td>
<td>Pattern database, Monte Carlo simulation</td>
<td>1400</td>
</tr>
<tr>
<td>2006</td>
<td>GNU GO</td>
<td>Pattern database, α-β search</td>
<td>1800</td>
</tr>
<tr>
<td>2006</td>
<td>MANY FACES</td>
<td>Pattern database, α-β search</td>
<td>1800</td>
</tr>
<tr>
<td>2006</td>
<td>NEUROGO</td>
<td>TDL, neural network</td>
<td>1850</td>
</tr>
<tr>
<td>2007</td>
<td>RLGO</td>
<td>TD search</td>
<td>2100</td>
</tr>
<tr>
<td>2007</td>
<td>MoGo</td>
<td>MCTS with RAVE</td>
<td>2500</td>
</tr>
<tr>
<td>2007</td>
<td>CRAZY STONE</td>
<td>MCTS with RAVE</td>
<td>2500</td>
</tr>
<tr>
<td>2008</td>
<td>FUEGO</td>
<td>MCTS with RAVE</td>
<td>2700</td>
</tr>
<tr>
<td>2010</td>
<td>MANY FACES</td>
<td>MCTS with RAVE</td>
<td>2700</td>
</tr>
<tr>
<td>2010</td>
<td>ZEN</td>
<td>MCTS with RAVE</td>
<td>2700</td>
</tr>
</tbody>
</table>
The basic steps of MCTS are given here as a reminder.

We recommend an extensive lecture on MCTS.

[see Section 2.3.4 for more details on the MCTS algorithm]
Child nodes are chosen based on the upper confidence bound (UCB) criterion

In brief

- With UCB we choose which node to explore based so as to balance exploration and exploitation

- UCB only calculates average reward for all children of a node, and number of visits
There are certain challenges MCTS faces when it is applied for real-time control of fast-paced games (e.g. Ms Pac-Man or Super Mario Bros)

- Limited roll-out budget
  - Heuristic knowledge becomes important
- Action space is fine-grained
  - Take *macro-actions* otherwise planning will be very short-term
- Maybe no terminal node in sight
  - Use a heuristic
  - Tune simulation depth
- Next state function may be expensive
  - Consider making a simpler abstraction
An Example: Jabosen et al. (2014) applied a number of MCTS variants for controlling Mario (video: best result obtained)

https://www.youtube.com/watch?v=01j7pbFTMXQ

Jacobsen, Greve, Tögelius: Monte Mario: Platforming with MCTS. GECCO 2014.
The various modifications on the paper by Jacobsen et al. (2014) that led to dissimilar results

<table>
<thead>
<tr>
<th>Modification</th>
<th>Mean Score</th>
<th>Avg. T Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vanilla MCTS (Avg.)</td>
<td>3918</td>
<td>131</td>
</tr>
<tr>
<td>Vanilla MCTS (Max)</td>
<td>2098***</td>
<td>153</td>
</tr>
<tr>
<td>Mixmax (0.125)</td>
<td>4093</td>
<td>147</td>
</tr>
<tr>
<td>Macro Actions</td>
<td>3869</td>
<td>142</td>
</tr>
<tr>
<td>Partial Expansion</td>
<td>3928</td>
<td>134</td>
</tr>
<tr>
<td>Roulette Wheel Selection</td>
<td>4032</td>
<td>139</td>
</tr>
<tr>
<td>Hole Detection</td>
<td>4196**</td>
<td>134</td>
</tr>
<tr>
<td>Limited Actions</td>
<td>4141*</td>
<td>137</td>
</tr>
<tr>
<td>(Robin Baumgarten’s A*)</td>
<td>4289***</td>
<td>169</td>
</tr>
</tbody>
</table>
• Several MCTS configurations get the same score as A*

• It seems that A* is playing essentially optimally

• But what if we modify the problem?
Making a Mess of Mario

- Introduce action noise:
  - 20% of actions are replaced with a random action
- Destroys A*
- MCTS handles this much better

<table>
<thead>
<tr>
<th>AI</th>
<th>Mean Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCTS (X-PRHL)</td>
<td>1770</td>
</tr>
<tr>
<td>A* agent</td>
<td>1342**</td>
</tr>
</tbody>
</table>

See “Jacobsen, Greve, Togelius: Monte Mario: Platforming with MCTS. GECCO 2014.” for more details
Example: MCTS @ Total War Rome II

- Task Management System
  - Resource Allocation (match resources to tasks)
    - Typically many tasks, but few resources
    - Large search space, little time
  - Resource Coordination (determine the best set of actions given resources & their targets)
    - Large search space
    - Grows exponentially with number of resources
    - Expensive pathfinding queries
    - MCTS-based planner to achieve constant worst-case performance
Decision making through planning does not need to be built on tree search. Alternatively, one can use optimization algorithms for planning.

[see Section 3.3.1.3 for more details]
• Basic idea:
  — Don’t search for a sequence of actions starting from an initial point
  — Optimize the whole action sequence instead!
• Search the space of complete action sequences for those that have maximum utility.
• Evaluate the utility of a given action sequence by taking all the actions in the sequence in simulation, and observing the value of the state reached after taking all those actions.

[see Section 3.3.1.3 for more details]
Evolutionary Planning

- Any optimization algorithm is applicable
- Evolutionary algorithms are popular so far; e.g.
  - *Rolling horizon evolution* in TSP
  - Competitive agents in General Video Game AI Competition
  - “Online evolution” outperforms MCTS in *Hero Academy*
  - Evolutionary planning performs better than varieties of tree search in simple *StarCraft* scenarios
- A method at birth – still a lot to come!

[see Section 3.3.1.3 for more details and references to papers]
The last category of planning-based methods in games include classical planning with symbolic representations.

[see Section 3.3.1.4 for more details]
Planning with Symbolic Representations

- Planning on the level of in-game actions requires a fast forward model
- However one can plan in an abstract representation of the game’s state space.
- Typically, a language based on first-order logic represents events, states and actions, and tree search is applied to find paths from current state to end state.
- Example: STRIPS-based representation used in Shakey, the world’s first digital mobile robot
- Game example: F.E.A.R. (Sierra Entertainment, 2005) agent planners by Jeff Orkin

[see Section 3.3.1.4 for more details]
A very important factor when selecting an AI algorithm to play a game is whether there is a simulator of the game, a so-called forward model, available.

For many games, however, it is impossible or at least very hard to obtain a fast forward model.
When a forward model cannot be produced, tree search algorithms cannot be applied. It is still possible to manually construct agents, and also to learn agents through supervised learning or some form of reinforcement learning.

Note: the planning-based methods (described earlier) for playing games cannot be directly compared with the reinforcement learning methods described here. They solve different problems: planning requires a forward model and significant time at each time step; reinforcement learning instead needs learning time and may or may not need a forward model.
[see Section 3.3.2.1 for more details]
[see Section 3.3.2.2 for more details]
Evolutionary Algorithms

- Stochastic global optimization algorithms
- Inspired by Darwinian natural evolution
- Extremely domain-general, widely used in practice
Simple $\mu+\lambda$ Evolutionary Strategy

- Create a population of $\mu+\lambda$ individuals
- At each generation
  - Evaluate all individuals in the population
  - Sort by fitness
  - Remove the worst $\lambda$ individuals
  - Replace with mutated copies of the $\mu$ best individuals
Evolving ANNs
Ms. Pac-Man Example

Fitness value $f_2$
Fig. 2. Neuroevolution in Existing Games. (a) NE is able to discover high-performing controllers for racing games such as TORCS [11]. (b) NE has also been successfully applied to commercial games, such as Creatures [44]. Additionally, NE enables new types of games such as GAR (c), in which players can interactively evolve particular weapons [46], or NERO (d), in which players are able to evolve a team of robots and battle them against other players [119].

Sebastian Risi and Julian Togelius (2016): Neuroevolution in games. TCIAIG.
The various role neuroevolution has played across dissimilar games genres.
Procedural Personas

- Given utilities (rewards) show me believable gameplay
- Useful for human-standard game testing
- RL
  - MCTS
  - Neuroevolution
  - ...
- Inverse RL

Q-learning

- Off-policy reinforcement learning method in the temporal difference family
- Learn a mapping from (state, action) to value
- Every time you get a reward (e.g. win, lose, score), propagate this back through all states
- Use the $\max$ value from each state
Agent consists of two components:
1. Value-function (Q-function)
2. Policy
Representing $Q(s, a)$ with ANNs
Training is performed on-line using the Q-values from the agent’s state transitions.

For Q-learning:

input: $s_t, a_t$

target: $r_t + \gamma \max_a Q(s_{t+1}, a)$
TD-Gammon (Teusaro, 1992)

predicted probability of winning, $V_t$

TD error, $V_{t+1} - V_t$

hidden units (40-80)

backgammon position (198 input units)
Deep Q-learning

- Use Q-learning with *deep* neural nets
- In practice, several additions useful/necessary
  - Experience replay: chop up the training data so as to remove correlations between successive states

[see 3.4.3.3 for more details]
Arcade Learning Environment

Based on an Atari 2600 emulator

- Atari: very successful but very simple
- 128 byte memory, no random number generator

A couple of dozen games available (hundreds made for the Atari)

Agents are fed the raw screen data (pixels)

Most successful agents based on deep learning
Human-level control through deep reinforcement learning

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg & Demis Hassabis

Affiliations | Contributions | Corresponding authors

Received 10 July 2014 | Accepted 16 January 2015 | Published online 25 February 2015
Results: not bad! ...but not general
How Can AI Play Games?

- Supervised learning (requires play traces to learn from)
  - Neural networks, k-nearest neighbours, SVMs etc.

[see Section 3.3.3 for more details]
Which Games Can AI Play?

[see Section 3.3.4. for more details]
Which Games Can AI Play?

- Board games
  - Adversarial planning, tree search
- Card games
  - Reinforcement learning, tree search

[see Section 3.4.1 and 3.4.2 for more details]
Which Games Can AI Play?

- Classic arcade games
  - Pac-Man and the like: Tree search, RL
  - Super Mario Bros: Planning, RL, Supervised learning
  - Arcade learning environment: RL
  - General Video Game AI: Tree search, RL

[see Section 3.4.3 for more details]
**Which Games Can AI Play?**

- **Strategy games**
  - Different approaches might work best for the different tasks (e.g. strategy, tactics, micro management in StarCraft)

[see Section 3.4.4 for more details]
Which Games Can AI Play?

- Racing games
  - Supervised learning, RL

[see Section 3.4.5 for more details]
Which Games Can AI Play?

- Shooters
  - UT2004: Neuroevolution, imitation learning
  - Doom: (Deep) RL in VizDoom

[see Section 3.4.6 for more details]
Which Games Can AI Play?

- Serious games
  - Ad-hoc designed believable agent architectures, expressive agents, conversational agents

[see Section 3.4.7 for more details]
Which Games Can AI Play?

- Interactive fiction
  - AI as NLP, AI for virtual cinematography, Deep learning (LSTM, Deep Q networks) for text processing and generation

[see Section 3.4.8 for more details]
Thank you!
gameaibook.org