e-print http://www.gm.fh-koeln.de/ciopwebpub/Kone17a.d/TR-GBG.pdf

# The GBG Class Interface Tutorial: General Board Game Playing and Learning

Wolfgang Konen

Computer Science Institute,
TH Köln,
Cologne University of Applied Sciences,
Germany

wolfgang.konen@th-koeln.de

June 2017

**Abstract**

This technical report introduces GBG, the general board game playing and learning framework. It is a tutorial that describes the set of interfaces, abstract and non-abstract classes which help to standardize and implement those parts of board game playing and learning that otherwise would be tedious and repetitive parts in coding. GBG is suitable for arbitrary 1-player, 2-player and $n$-player board games. It provides a set of agents (AI's) which can be applied to any such game. This document describes the main classes and design principles in GBG. GBG is written in Java.

# Contents

# 1 Introduction

## 1.1 Motivation

General board game (GBG) playing and learning is a fascinating area in the intersection of machine learning, artificial intelligence and game playing. It is about how computers can learn to play games not by being programmed but by gathering experience and learning by themselves (self-play). The learning algorithms are often called AI agents or just „AI"'s (AI = artificial intelligence). There is a great variety of learning algorithms around, e.g. reinforcement learning algorithms like TD($\lambda$), Monte Carlo tree search (MCTS), different neural network algorithms, Minimax, ... to name only a few.

Even if we restrict ourselves to board games, as we do in this paper (and do not consider other games like video games), there is a plethora of possible board games where an agent might be active in. The term „General" in GBG refers to the fact that we want to have in the end agents or AIs which perform well on a large variety of games. There are quite different games: 1-person games (like Solitaire, 2048, ...), 2-person games (like Tic-Tac-Toe, Othello, Chess, ...), many-person games (like Settlers of Catan, Poker, ...). The game environment may be deterministic or it may contain some elements of chance (like rolling the dices, ...).

A common problem in GBG is the fact, that each time a new game is tackled, the AI developer has to undergo the frustrating and tedious procedure to write adaptations of this game for all agent algorithms. Often he/she has to reprogram many aspects of the agent logic, only because the game logic is slightly different to previous games. Or a new algorithm or AI is invented and in order to use this AI in different games, the developer has to program instantiations of this AI for each game.

Wouldn't it be nice if we had a framework consisting of classes and interfaces which abstracts the common processes in GBG playing and learning? If someone programs a new game, he/she has just to follow certain interfaces described in the GBG framework, and then can easily use and test on that game *all* AIs in the GBG library.

Likewise, if an AI developer introduces a new learning algorithm which can learn to play games, she has only to follow the interface for agents laid down in the GBG framework. Then she can test this new agent on *all* games of GBG. Once the interface is implemented she can directly train her agent, inspect its move decisions in each game, test it against other agents, run competitions, enter game leagues, log games and so on.

The rest of this document introduces the class concept of GBG. GBG is

written in Java. After a short (and probably incomprehensive) summary of related work in Sec. 1.2, Sec. 2 gives an overview of the relevant classes and Sec. 3 discusses them in detail. Sec. 4 discusses some use cases and FAQs for the GBG class framework. Appendix A lists the methods of the important classes and interfaces.

### 1.2 Related Work

Epstein [2001] presented with *Hoyle* an early general board game playing program. It learned to play 18 diverse board games, where its strategic principles are general and not game specific. The software implementation of *Hoyle* is not further described in Epstein [2001].

There is the discipline **General Game Playing (GGP)** (Genesereth and Thielscher [2014], Mańdziuk and Świechowski [2012]) which has a long tradition in artificial intelligence: A GGP competition organized by the Stanford Logic Group is held annually at the AAAI conferences since 2005 (Genesereth et al. [2005]). Given the game rules written in the so-called *Game Description Language* (GDL, Love et al. [2008]), several AIs enter one or several competitions. As an example for GGP-related research, Mańdziuk and Świechowski [2012] propose a universal method for constructing a heuristic evaluation function for any game playable within the GGP framework. Méhat and Cazenave [2010] show a study of single player games in the GGP context where they use variants of Monte Carlo Tree Search. The program is based on *Ary*, which is written in C with a Prolog interface engine. The program was tested on 22 single player games. Michulke and Thielscher [2009] proposes a GGP learning framework where the AI's learn from experience with TD($\lambda$) reinforcement learning.

Why then do we need GBG if we have GGP already? – GGP usually solves a tougher task, each agent is a *Tabula Rasa*, i. e. no game specific features are known to the AI's at compile time. But then GGP is usually only concerned with rather simple games or – if it tackles more complex games like Connect Four or Othello – it reaches only a very modest playing strength on them. Another restriction is that GGP can only deal with deterministic games. We aim with GBG at a slightly different goal: A framework where the game or AI implementer has the freedom to define features or symmetries at compile time which she believes to be useful for her game, but where the learning of the game tactics (when to perform which action) is completely left to the AIs (e. g. learning through self-play). Yet the features are embedded in a generic interface, so that the agents are general and can be applied to any game. We then aim at developing AI's which learn to play perfectly on simple games and which exhibit a decent playing strength on games of larger complexity. We include in GBG

explicitly non-deterministic games with stochastic elements as well.

Other work which is somewhat related to GBG: *OpenAI Gym* (Brockman et al. [2016]) is a toolkit for reinforcement learning research which has also a board games environment supporting a (small) set of games. *General Video Game Playing* (GVGP, Levine et al. [2013]) is a related field which does not tackle board games as in GGP or GBG, but instead video games.

We define a **board game** as such a game being played with a known number of players $n$, usually on a game board, but the game board may be the table as well. The game proceeds through actions (moves) of each player. This differentiates board games from video games where usually each player can take an action at any point in time. Note that this definition of board games includes (trick-taking) card games (like Skat, ...) as well.

To summarize we propose with GBG a framework which differs from existing frameworks in the following aspects:

- It is written in Java and thus available on most OS platforms.
- It allows the user to define game specific features and symmetries which are embedded into the agent framework in a generic way (see Sec. 3.5).
- It allows to tackle non-deterministic games.
- It offers an n-tuple based TD($\lambda$) agent (see Sec. 3.6 and Sec. 4.6).
- It allows agent-agent competitions and human-agent play.

## 2 Class and Interface Overview

Interface StateObservation is the main interface a game developer has to implement once he/she wants to introduce a new game. A class derived from StateObservation observes a game state, it can infer from it the available actions, knows when the game is over, can advance a state into a new legal state given one of the available actions. If a random ingredient from the game environment is necessary for the next action (of the next player), the advance function will add it.

The second interface a game developer has to implement is the interface GameBoard, which realizes the board GUI and the interaction with the board. If one or more humans play in the game, they enter their moves via GameBoard.

The interface an AI developer has to implement is the interface PlayAgent It represents an „AI“ or agent capable of playing games. If necessary, it can be trained by self-play. Once trained, it has methods for deciding about the best

next action to take in a game state StateObservation and getting the agent's estimate of the score or value of a certain game state.

The heart of GBG are the abstract classes Arena and ArenaTrain. In the Arena all agents meet: They can be loaded from disk, they play a certain game, there can be competitions. In ArenaTrain, which is a class derived from Arena, there are additional options to parametrize, train, inspect, evaluate and save agents.

The helper classes Feature, XNTupleFuncs, and Evaluator support the abstraction in the classes Arena and ArenaTrain.

# 3 Classes in Detail

## 3.1 Interface StateObservation

Interface **StateObservation** observes the current state of the game, it has utility functions for

- returning the available actions (`getAvailableActions()`),
- advancing the state of the game with a specific action (`advance()`),
- copying the current state
- getting the score of the current state
  (`getScore()`, `getScore(StateObservation referingState)`)
- signaling end and winner of the game

If a game has random elements (like rolling the dices in a dice game or placing a new tile in 2048), `advance()` is additionally responsible for invoking such random actions and reporting the results back in the new state. Examples:

- For a dice-rolling game: the game state is the board & the dice number.
- For 2048: the game state is just the board (with the random tile added).

Implementing classes: StateObserverTTT, StateObserver2048, ...

As an example, StateObserverTTT is a state observer for the game Tic-TacToe: It has constructors with game-specific parameters (`int [][] table`, `int Player`). It has access functions `getTable()` and `getPlayer()`. The latter returns the Player who has to move in the current state.

### 3.2 Interface PlayAgent and class AgentBase

Interface **PlayAgent** has all the functionality that an AI (= game playing agent) needs. The most important methods are:

- `getNextAction(sob,...)`: given the current game state `sob`, return the best next action.
- `double getScore(sob)`: the score (agent's estimate of final reward) for the current game state `sob`.
- `trainAgent(sob,...)`: train agent for one episode[1] starting from state `sob`.

Some more methods, e.g. setters and getters, have their defaults implemented in class **AgentBase**.[2] It might be useful to design a new agent class with the signature `... extends AgentBase implements PlayAgent`.

There is an additional method `double estimateGameValue(sob)` which has the default implementation `getScore(sob)` in **AgentBase**. This method is called when a training game is stopped prematurely because the maximum number of moves in an episode ('Episode length') is reached.[3] See Sec. 3.3 and 3.4 for more details on game score and game value.

Classes implementing interface **PlayAgent** and derived from **AgentBase**:

- RandomAgent: an agent acting completely randomly
- HumanPlayer: an agent waiting for user interaction
- MinimaxAgent: a simple tree search (max-tree for 1-player games, min-max-tree for 2-player games)[4]
- MCTSAgent: Monte-Carlo Tree Search agent
- MCAgent: Monte-Carlo agent (no tree)
- TDAgent: general TD($\lambda$) agent (temporal difference reinforcement learning) with neural network value function (see Sec. 4.5 for more details). This agent requires a Feature object in constructor, see Sec. 3.5.

[1] An *episode* is one specific game playout.

[2] **AgentBase** does not implement all methods of the interface PlayAgent, so it has no `... implements PlayAgent`.

[3] Or, for agents MCTS or MC, when the maximum rollout depth ('Rollout depth') is reached.

[4] Note that Minimax in this simple implementation may not be appropriate for games with random elements, because Minimax follows in each tree step only *one* path of the possible successors that `advance()` may produce.

- TDNTupleAgt: TD($\lambda$) agent (temporal difference reinforcement learning) using n-tuple sets as features (see Sec. 4.6 for more details). This agent requires an object of class XNTupleFuncs in constructor, see Sec. 3.6.

Each agent has an **AgentState** member, which is either RAW, INIT or TRAINED.

More details on TD($\lambda$) (temporal difference learning, reinforcement learning for games, eligibility traces) can be found in the technical report Konen [2015].

Some of the agents (RandomAgent, HumanAgent, MinimaxAgent, MCTSAgent, MCAgent) are directly after construction in a TRAINED state, i.e. they are ready-to-use. Minimax, MCTS and MC make their observations on-the-fly, starting from the given state. Some other classes (TDAgent, TDNTupleAgt, ...) require training, they are after construction in state INIT.

Classes derived from PlayAgent should implement the `Serializable` interface. This is needed for loading and saving agents. Agent members which should be *not* included in the serialization process have to be flagged with keyword `transient`. Agent members which are user-defined classes should implement the `Serializable` interface as well.

### 3.3 Some Remarks on the Game Score

Although the game score (the final result of a game, e. g. „X wins“ or „O wins with that many points“) seems to be a pretty simple and obvious concept, it becomes a bit more confusing if one wants to define the game score consistently for a broader class of states, not just for a terminal state. We use the following conventions:

- `StateObservation.getGameScore()` returns the sum of rewards for the player **who has to move** in the current state. Most 2-player games will give the reward only in the end (win/tie/loss), so that for those games `getGameScore()` is usually 0 as long as the game state is non-terminal. If the game state is terminal, a negative reward will be returned if the player loses and a positive reward if the player wins. For other games there might be also rewards during the game.

- If a state is terminal (e. g. „X wins“) then the „player who moves“ has changed a last time (i. e. to player O, although the game is over.). Thus the score will be -1 („O loses“). This seems a bit awkward at first sight, but it is the only way to guarantee in a succession of actions for 2-player games that the current score is always the negative of the next state's score (negamax principle). Fig. 1 shows an example.

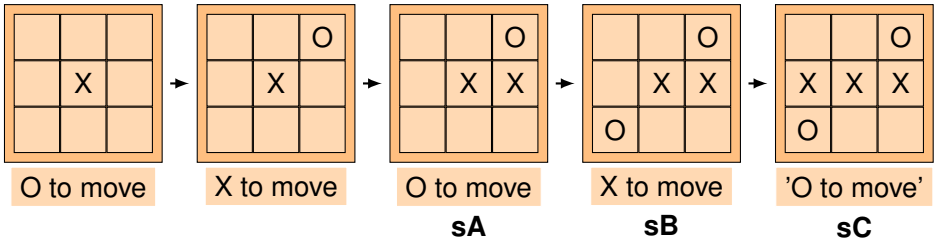


Figure 1: A succession of states in TicTacToe: If O makes in state **sA** the losing move leading to **sB**, then **sB** is a clear win for X, so terminal state **sC** should be a clear loss for O to be consistent. The game score for **sC** is $-1$. The game values (see Sec. 3.4) for **sB** and **sC** are $+1$ and $-1$, resp.

- Example: A 2-player game like TicTacToe is terminal when X makes a winning move. On this terminal state O would have to move next (if it were not terminal). So the game score for this terminal state (**sC** in Fig. 1) is a negative reward $-1$ for player O. It turns out that TicTacToe is always terminated with either a negative reward or a tie.

- What differentiates `sC.getGameScore(StateObservation referingState)` from `sC.getGameScore()`? – The latter is the plain game score for the state **sC**, while the former is the game score of **sC** viewed from the perspective of the player to move in the predecessor state `referingState`. In 2-person games this amounts to a factor $-1$ if the players in **sC** and `referingState` are different, that is:

  ```
  sC.getGameScore(sC) = -1 = sC.getGameScore();
  sC.getGameScore(sB) = +1;
  sC.getGameScore(sA) = -1;
  ```

  This is advantageous for agents like MC or MCTS which perform rollouts from `referingState`. With method `sC.getGameScore(referingState)` they get in all settings a quantity which they have to *maximize*, no matter which player ends the game.

- `StateObservation.getGameWinner()` may only be called if the game is over for the current state (otherwise an assertion fires). It returns an enum `Types.WINNER` which may be one out of {PLAYER_WINS, TIE, PLAYER_LOSES}. The player is always the player who has to move. The method `Types.WINNER.toInt()` converts these enums to integers which correspond to $\{+1, 0, -1\}$, resp.

StateObservation defines two methods

```
public double getMinGameScore();
public double getMaxGameScore();
```

These methods should return the minimum and maximum game score which can be achieved in a specific game. This is needed since some PlayAgent (e.g. TDAgent) make predictions of the estimated game score with the help of a neural network. Since a neural network has often a sigmoid output function which can emit only values in a certain range (e.g. [0,1]), it is necessary to map the game scores to that range as well. This can only be done if the minimum and maximum game score is given.[5]

### 3.4 Difference between Game Score and Game Value

There is a subtle distinction between game score and game value. The **game score** is the score of a game according to the game laws. For example, TicTacToe has the score 0 for all intermediate states, while a terminal state has either +1/0/-1 as game score for win/draw/loss of the player to move. In 2048, the game score is the cumulative sum of all tile merges. Each player usually wants to maximize the expectation value of 'his' score at the end of the game.

But the score in an intermediate game state is not a good indicator of the *potential* of that state. Two states in 2048 might have the same score, but the **game value** of these states can be different. While the first state might be close to the terminal state, the second one might have a higher mobility and thus 'last' longer and receive a higher final score. The precise game value of a state is often not known / not computable, but it is of course desirable to estimate it. An estimate can be based on a simple heuristic like the weighted piece count in chess.

The main possibility to deliver a game value is:

- `PlayAgent.getScore(StateObservation so)` returns the agent's estimate of the final score for the player **who has to move** in StateObservation `so` – assuming perfect play of that player. That is what we call the **game value** of `so`. The game value for 2-player games is usually +1 if it is expected that the player wins finally, 0 if it is a tie and -1 if he loses. Values in between characterize expectation values in cases where different outcomes are possible or likely (or where the agent has not yet gathered enough information or experience).

[5] If a precise maximum game score for a certain game is not known, a reasonable 'big' estimate is usually also sufficient.

There are other methods to deliver a game value

- `StateObservation.getGameValue()`
- `PlayAgent.estimateGameValue(StateObservation so)`

but they are only for advanced users and their description is deferred to Appendix B.

### 3.5 Interface Feature

Some classes implementing PlayAgent need a game-specific feature vector. As an example, consider **TDAgent**, the general TD($\lambda$) agent (temporal difference reinforcement learning) with neural network value function. To make the neural network predict the value of a certain game state, the network needs some feature input (e.g. specific board patterns which form threats or opportunities, number of them, number of pieces and so on). These features are usually game-specific. We assume here that every feature can be expressed as double value (neural networks can only digest real numbers as input), so that the whole feature vector can be expressed as `double[]`.

To create an **Feature** object within the general Arena-code, the *factory method pattern* is used: Arena defines an abstract method

```
public Feature makeFeatureClass(int featmode);
```

The argument `featmode` allows to construct different flavors of Feature objects and to test and evaluate them.

In all cases where Arena or ArenaTrain needs a Feature object, it will call this method `makeFeatureClass(int)`. This will take place whenever a `TDAgent` object is constructed, because the `TDAgent` constructor needs a Feature object as parameter.

Interface Feature has the method

```
public double[] prepareFeatVector(StateObservation so);
```

which gets a game state and returns a double vector of features. This vector may serve as an input for a neural network or other purposes.

Implementing classes: FeatureTTT, Feature2048, ...

### 3.6 Interface XNTupleFuncs

There is one special agent **TDNTupleAgt**, which realizes TD-learning with n-tuple features. N-tuple features or n-tuple sets (Lucas [2008], Thill et al. [2014], Bagheri et al. [2015], Thill [2015]) are another way of generating a large number of features. An n-tuple is a set of board cells. For every game state StateObservation it can translate the position values present in these cells into a double score or value. In order to construct such n-tuples, the user has to implement the interface XNTupleFuncs. See Sec. 4.6 for more details on the member functions of XNTupleFuncs and Appendix C for more details on n-tuples.

To create an XNTupleFuncs object within the general Arena-code, the *factory method pattern* is used: Arena defines an abstract method

```
public XNTupleFuncs makeXNTupleFuncs();
```

In all cases where Arena or ArenaTrain needs a XNTupleFuncs object, it will call this method `makeXNTupleFuncs()`. This will take place whenever a **TDNTupleAgt** object is constructed, because the **TDNTupleAgt** constructor needs an XNTupleFuncs object as parameter.

*Note*: If you do not plan to use **TDNTupleAgt** in your game, you do not need to implement a specific version of class XNTupleFuncs. The default implementation of `makeXNTupleFuncs()` in Arena will just throw a `RuntimeException`.

### 3.7 Interface GameBoard

Interface **GameBoard** has the game board GUI (usually in a separate JFrame). It provides functionality for:

- Maintaining its own StateObservation object `m_so`. This object is after construction in a default start state (e. g. empty board). The same state can be reached via `clearBoard()` or `getDefaultStartState()` as well. The associated GUI will show the default start state.
- Showing or updating the current game state (StateObservation) in the GUI and enabling / disabling the GUI elements (`updateBoard(...)`).
- Human interaction with the board: see Sec. 3.8.
- Returning its current StateObservation object (`getStateObs()`).
- `chooseStartState01()`: This method returns randomly one out of a set of different start states. This is useful when training an agent so that not

always the same game episode is played but some variation (exploration) occurs.

Example for TicTacToe: The implementation in GameBoardTTT returns with probability 0.5 the default start state (empty board) and with probability 0.5 one out of the possible next actions (an 'X' in any of the nine board positions).

Implementing classes: GameBoardTTT, GameBoard2048, ...

### 3.8 Human interaction with the board and with Arena

During game play: How is the integration between user actions (human moves) and AI agent actions implemented?

If GameBoard request an action from Arena, then its method `isActionReq()` returns `true`. This causes the selected AI to perform a move. If on the other hand a human interaction is requested, Arena issues a `setActionReq(false)` and this causes `isActionReq()` to return `false` as well. GameBoard then waits for GUI events until a user (human) action is recorded. GameBoard is responsible for checking whether the human action is legal (`isLegalAction()`).[6] If so, then GameBoard issues an `advance()`. Method `advance()` opens the possibility for invoking random elements from the game environment (e. g. adding a new tile in 2048), if necessary.

When all this has happened, GameBoard sets its internal state such that `isActionReq()` returns `true` again. Thus it asks Arena for the next action and the cycle continues. Finally, Arena detects an `isGameOver()`-condition and finishes the game play.

### 3.9 Abstract Class Evaluator

Class **Evaluator** evaluates the performance of a PlayAgent. Evaluators are called in menu item 'Quick Evaluation', during training and at the end of each competition in menu item 'Multi-Competition'. It is important to note that Evaluator calls have no influence on the training process, they just measure the (intermediate or final) strength of a PlayAgent.

In the constructor

```
public Evaluator(PlayAgent e_PlayAgent, int stopEval,
                 int mode, int verbose);
```

[6] see method `HGameMove(x,y)` in GameBoardTTT for an example.

the argument `mode` allows derived classes to create different types of evaluators. These may test different abilities of PlayAgent.[7]

A normal evaluation is started by calling Evaluator's method `eval` which calls in turn the abstract method

```
abstract protected boolean eval_Agent();
```

and counts the consecutive successful returns from that method. The argument `stopEval` sets the number of consecutive evaluations that the abstract method `eval_Agent()` has to return with `true` until the evaluator is said to reach its *goal* (method `goalReached()` returns `true`). This is used in XArenaFunc's method `train()` as a possible condition to stop training prematurely.

Method `eval_Agent()` needs to be overridden by classes derived from Evaluator. It returns `true` or `false` depending on a user-defined success criterion. In addition, it lets method `double getLastResult()` return a `double` characterizing the evaluation result (e. g. the average success rate of games played against Minimax player).

Concrete objects of class Evaluator are usually constructed by the factory method

```
abstract public Evaluator makeEvaluator(PlayAgent e_PlayAgent,
                int stopEval, int mode, int verbose);
```

in Arena or ArenaTrain.

Implementing classes: EvaluatorTTT, Evaluator2048, ...

### 3.10 Abstract Class Arena

Class **Arena** is an abstract class for loading agents and playing games. Why is it an abstract class? – **Arena** has to create an object implementing interface GameBoard, and this object will be game-specific, e. g. a GameBoardTTT object. To create such an object within the general Arena-code, the *factory method pattern* is used: Arena defines the abstract methods

```
abstract public GameBoard makeGameBoard();
abstract public Evaluator makeEvaluator(...);
```

[7] For complex games it is often very difficult or impossible to have a perfect evaluator. Remember that (a) that the game tree can be too complex to retrieve the perfect action for a certain state and that (b) a perfect Evaluator should evaluate the actions of PlayAgent to *every* possible state, which would take too long (or is impossible) for games with larger state space complexity. A partial way out is to have different Evaluator modes which evaluate the agent from different perspectives.

The first method is a factory method for GameBoard objects. The second method is a factory method for Evaluator objects. Both will be implemented by classes derived from Arena. That is, a derived class ArenaTTT can be very thin, it just implements the methods `makeGameBoard()` and `makeEvaluator()` and lets them return (in the example of TicTacToe) GameBoardTTT and EvaluatorTTT objects, resp.

Class Arena has in addition the factory method

```
public Feature makeFeatureClass(int);
```

If it is not overridden by derived classes, it will throw a `RuntimeException` (no game-tailored Feature object available). If a class derived from Arena wants to use a trainable agent requiring Feature (e. g. `TDAgent`) then it has to override `makeFeatureClass`.

Class Arena has similarly the factory method

```
public XNTupleFuncs makeXNTupleFuncs();
```

which can be used to generate a game-tailored XNTupleFuncs object, if needed (if agent **TDNTupleAgt** is used ). If not overridden, it will throw a `RuntimeException`.

Class Arena has the following functionality:

- choice of agents for each player (load)
- playing games (AI agents & humans)
- inspecting the move choices of an agent
- logging of played games (option for later replay or analysis)
- a slider during agent-agent game play to control the playing velocity
- (TODO) undo/redo possibilities
- (TODO) game balancing
- (TODO) game leagues, round-robin tournaments, ...

Derived abstract class: ArenaTrain. Derived non-abstract classes: ArenaTTT, Arena2048, ...

### 3.11 Abstract Class ArenaTrain

Class **ArenaTrain** is an abstract class derived from Arena which has additional functionality:

- specifying all parameters for an agent
- training an agent (one or multiple times)
- evaluating agents, competitions (one or multiple times)
- saving agents
- (TODO) replay memory for better training

The helper classes XArenaFuncs, XArenaButtons, XArenaMenu, XArenaTabs contain functionality needed for Arena and ArenaTrain.

Derived non-abstract classes: ArenaTrainTTT, ArenaTrain2048, ...

## 4 Use Cases and FAQs

### 4.1 I have implemented game XYZ and want to use AI agents from GBG – what do I have to do?

As a game developer you have to implement the following four interfaces for your game:

- `StateObserverXYZ implements StateObservation`
- `GameBoardXYZ implements GameBoard`
- `EvaluatorXYZ extends Evaluator`
- `FeatureXYZ extends Feature` (only needed, if the game wants to use trainable agents like **TDAgent**).
- `XNTupleFuncsXYZ extends XNTupleFuncs` (only needed, if the game wants to use the n-tuple agent **TDNTupleAgt**).

Once this is done, you only need to write a very 'thin' class ArenaTrainXYZ with suitable constructors, which overwrites the abstract methods of class ArenaTrain with the factory pattern methods

```
public GameBoard makeGameBoard() {
   gb = new GameBoardXYZ(this);
   return gb;
}
public Evaluator makeEvaluator(PlayAgent pa, GameBoard gb,
                int stopEval, int mode, int verbose) {
   return new EvaluatorXYZ(pa,gb,stopEval,mode,verbose);
}
```

If needed, you should overwrite the methods (see Sec. 4.5 and Sec. 4.6)

```
public Feature makeFeaturClass(int featmode) {
   return new FeatureXYZ(featmode);
}
public XNTupleFuncs makeXNTupleFuncs() {
   return new XNTupleFuncsXYZ();
}
```

as well.

If you do not want to use the agents **TDAgent** and **TDNTupleAgt** needing these factory methods, you may just implement stubs throwing suitable exceptions:

```
public Feature makeFeaturClass(int featmode) {
    throw new RuntimeException("Feature not implemented for XYZ");
}
public XNTupleFuncs makeXNTupleFuncs() {
    throw new RuntimeException("XNTupleFuncs not implemented for XYZ");
}
```

Finally you need a class with `main()` to launch ArenaTrain. You may copy and adapt the example in `LaunchTrainTTT`.

Then you can use for your game all the functionality laid down in ArenaTrain and all the wisdom of the AI agents implementing PlayAgent. Cool, isn't it?

## 4.2 How to train an agent and save it

1. Create an ArenaTrain object
2. Select an agent and set its parameters
3. Set training-specific parameters:

- `maxTrainNum`: 'Training games' = number of training episodes,
- `numEval`: after how many episodes an intermediate evaluation is done,
- `epiLength`: 'Episode length' = maximum allowed number of moves in a training episode. If it is reached, the game is stopped and `PlayAgent.estimateGameValue()` is returned (either up-to-now-reward or estimate of current + future rewards). If the game terminates earlier, the final game score is returned.

4. Train the agent & visualize intermediate evaluations.
5. Optional: Inspect the agent (how it responds to certain board situations).
6. Save the agent

### 4.3 Which AI's are currently implemented for GBG?

The following AI's (agents) are currently implementing interface PlayAgent:

- Class TDAgent (Temporal difference reinforcement learning)
- Class TDNTupleAgt (Temporal difference learning with n-tuples)
- Class MCTS (Monte Carlo Tree Search)
- Class MC (pure Monte Carlo search)
- Class MinimaxAgent (Minimax tree search of prescribed depth)
- Class RandomAgent (an agent acting completely randomly)
- Class HumanAgent (an agent waiting for human input on the game board)

### 4.4 How to write a new agent (for all games)?

Of course your new agent `NewAgent` has to implement the interface PlayAgent. You may want to derive your new agent from AgentBase to have a few basic functions already with their default implementations. These functions can be overridden if necessary.

The new agent should as well implement the interface Serializable (java.io) to be loadable and savable.

There are a few places in the code where the new agent has to be registered:

- `Types.GUI_AGENT_LIST`: Add a suitable agent nickname `"nick"`. This is how the agent will appear in the agent choice boxes.
- `XArenaFuncs.constructAgent()`: Add a suitable clause
  ```
  if (sAgent.equals("nick")) ...
  ```
- `XArenaFuncs.fetchtAgent()`: Add a suitable clause
  ```
  if (sAgent.equals("nick")) ...
  ```
- `XArenaMenu.loadAgent()`: Add a suitable clause
  ```
  if (td instanceof NewAgent) ...
  ```
- `LoadSaveTD.loadTDAgent()`: Add a suitable clause
  ```
  if (obj instanceof NewAgent) ...
  ```

### 4.5 How to specialize the TD agent to a new game?

Suppose you have implemented a new game XYZ and want to write a TD agent (temporal difference agent) which learns this game. What do you have to do? – Luckily, you can re-use most of the functionality laid down in class **TDAgent** (see Sec. 3.2).

1. Write a new Feature class

   ```
   public class FeatureXYZ implements Feature, Serializable
   ```

   This is the only point where some code needs to be written: Think about what features are useful for your game. In the simplest case this might be the raw board positions, but these features may characterize the win- or loose-probability for a state only rather indirectly. Other patterns may characterize the value (or the danger) of a state more directly. For example, in the game TicTacToe any two-in-a-line opponent pieces accompanied by a third empty position pose an immanent threat. A typical feature may be the count of those threats. N-tuple sets (Lucas [2008], Thill et al. [2014], Bagheri et al. [2015], Thill [2015]) are another way of generating a large number of features in a generic way (but they are not part of **TDAgent**, see Sec. 4.6, Appendix C and **TDNTupleAgt** instead).

2. Add to ArenaXYZ and ArenaTrainXYZ the overriding method

   ```
   public Feature makeFeatureClass(int featmode) {
       return new FeatureXYZ(featmode);
   }
   ```

**TDAgent** will generate by reinforcement learning a mapping from feature vector to game value (estimates of the final score, see Sec. 3.4) for all relevant game states.

The class ArenaTrainTTT (together with FeatureTTT) may be inspected to view a specific example for the game TicTacToe.

### 4.6 How to write a new TDNTupleAgt agent for a specific game?

Suppose you have implemented a new game XYZ and want to write a TD (temporal difference) agent using n-tuples which learns this game. What do you have to do? – Luckily, you can re-use most of the functionality laid down in class **TDNTupleAgt** (see Sec. 3.2). As a game implementer you have to do the following:

1. Write a new XNTupleFuncs class (Sec. 3.6)

   ```
   public class XNTupleFuncsXYZ
   implements XNTupleFuncs, Serializable
   ```

   Here you have to code some rather simple things like the number of board cells in your game (`getNumCells()`) and the number of position values (`getNumPositionValues()`) that can appear in each cell. This is for example 9 and 3 (X/O/empty) in the game TicTacToe.

   Next you implement

   ```
   int[] getBoardVector(so)
   ```

   which transforms a game state `so` into an `int[]` board vector.

   If your game has symmetries (the game TicTacToe has for example eight symmetries, 4 rotations $\times$ 2 mirror reflections), the function

   ```
   int[][] symmetryVectors(int[] boardVector)
   ```

   should return for a given board vector all symmetric board vectors (including itself). If the game has no symmetries, it returns just the board vector itself.

   The method

   ```
   HashSet adjacencySet(int iCell)
   ```

returns the set of cells adjacent to the cell with number `iCell`. Whether adjacency is a 4-point- or an 8-point-neighborhood or something else is defined by the user. This function is needed by **TDNTupleAgt** if the shape of the n-tuples is to be created by random walk.

Finally you implement

```
int[][] fixedNTuples()
```

a function returning a fixed set of n-tuples suitable for your game. If you do not need fixed n-tuple sets, you may leave `fixedNTuples()` unimplemented (i. e. let it throw an exception) and chose in the `NTPar` (n-tuple params) tab 'Random n-tuple generation'.

2. Add to ArenaXYZ and ArenaTrainXYZ the overriding method

```
public XNTupleFuncs makeXNTupleFuncs() {
    return new XNTupleFuncsXYZ();
}
```

The class ArenaTrainTTT (together with XNTupleFuncsTTT) may be inspected to view a specific example for the game TicTacToe.

**TDNTupleAgt** offers several possibilities to construct n-tuples:

(a) using a predefined, game-specific set of n-tuples (see `fixedNTuples()` above),

(b) random n-tuples generated by random-cell-picking (the cells in an n-tuple are in general not adjacent), and

(c) random n-tuples generated by random walk (every cell in each n-tuple has adjacent at least one other cell of this n-tuple; needs method `adjacencySet`, see above).

A cell may (and often should) be part of several n-tuples.
See Appendix C for further information on n-tuples.

# 5 Open Issues

The current GBG class framework is still in its test phase. The design of the classes and interfaces may need further reshaping when more games or agents are added to the framework. There are a number of items not fully tested or not yet addressed:

- The GUI for Arena and ArenaTrain is just a quick hack adapted from earlier programs and may need further refinement.
- The above-mentioned elements for Arena and ArenaTrain and further elements that are planned but not yet implemented:
  - Add Arena and ArenaTrain launchers which allow to select between the different implemented games and then launch the appropriate derived Arena and ArenaTrain class.
  - undo/redo possibilities
  - optional game visualization and game logging during competitions as well
  - game balancing
  - game leagues, round-robin tournaments, ...
  - option to enable/disable game value display during game play
  - client-server architecture for game play via applet on a game page. Option for a 'hall of fame'. An example for the game Hexi is found under http://www.dbai.tuwien.ac.at/proj/ramsey.
  - Implement the game Sim (= Hexi) in the Arena and ArenaTrain framework. A Java code example of the Sim board GUI may be found under http://www.dbai.tuwien.ac.at/proj/ramsey. Generalize the number of nodes (not only 6). Later, one may create a 3-player variant of Sim and test the framework on this.
  - Replay memory for better training: This is the idea used by DeepMind in learning Atari video games. Played episodes are stored in a replay memory pool and used repeatedly for training.
- The extension to $n$-player games ($n > 2$) is not fully functional yet. An example to fully implement and test $n$-player games can be the 3-player variant of the game Sim. The cases $n = 1$ and $n = 2$, which are fully functional, need also to be tested on a variety of 1- and 2-player games.
- Another 3-player game which could be tested is Skat.
- The n-tuple agent developed for C4 (Connect Four) needs to be ported to GBG.
- Allow only trained agents to be saved.
- Clarify: Is the parameter data flow safe, if we issue a 'play' or 'compete' for 2 agents of same type but with different parameters?

# A Appendix: Interface Summary

## A.1 Interface StateObservation

```
/**
 * Class StateObservation observes the current state of the game,
 * it has utility functions for
 * <ul>
 * <li> returning the available actions (getAvailableActions()),
 * <li> advancing the state of the game with a specific action (advance()),
 * <li> copying the current state
 * <li> signaling end, score and winner of the game
 * </ul>
 *
 * @author Wolfgang Konen, TH Köln, Nov'16
 */
public interface StateObservation {

public StateObservation copy();
public String stringDescr();
public double getGameScore();
public double getGameScore(StateObservation referingState);
public double getMinGameScore();
public double getMaxGameScore();
public void advance(ACTIONS action);
public boolean isLegalState();
public boolean isGameOver();
public Types.WINNER getGameWinner();
public ArrayList<ACTIONS> getAvailableActions();
public void setAvailableActions();
public int getNumAvailableActions();
public Types.ACTIONS getAction(int i);
public void storeBestActionInfo(ACTIONS actBest, double[] vtable);
public int getNumPlayers(); // n
public int getPlayer();     // (0,1,...,n-1)
public int getPlayerPM();   // (+1,-1) for a 2-player game
}
```

## A.2 Interface GameBoard

```
/**
 * Each class implementing interface GameBoard has the board game GUI.
 *
 * It has an internal object derived from StateObservateion which
 * represents the current game state. It can be retrieved (getStateObs()),
 * reset and retrieved (getDefaultStartState()) or a random start
 * state can be chosen (@link #chooseStartState01()).
 * @author Wolfgang Konen, TH Köln, Nov'16
 */
public interface GameBoard {

public void clearBoard(boolean boardClear, boolean vClear);
public void updateBoard(StateObservation so, boolean showStoredV,
            boolean enableOccupiedCells);
public void showGameBoard(ArenaTrain ticGame);
public boolean isActionReq(); // action requested from Arena?
public void setActionReq(boolean actionReq);
public void enableInteraction(boolean enable);
public StateObservation getStateObs();
public StateObservation getDefaultStartState(); // empty-board state
public StateObservation chooseStartState01();
}
```

## A.3 Interface PlayAgent

```
/**
 * The abstract interface for the game playing agents.
 *
 * @author Wolfgang Konen, TH Köln, Nov'16
 */
public interface PlayAgent {
public enum AgentState {RAW, INIT, TRAINED};

public Types.ACTIONS getNextAction(StateObservation sob, boolean random,
                    double[] vtable, boolean silent);
public double getScore(StateObservation sob);
```

```
public double estimateGameValue(StateObservation sob);
public boolean wasRandomAction(); // was last getNextAction random?
public boolean trainAgent(StateObservation so); // for one episode
public boolean trainAgent(StateObservation so,
               int epiLength);        // with episode length limit
public String printTrainStatus();
public String stringDescr();
public String getName();
public void setName(String name);
public AgentState getAgentState();
public void setAgentState(AgentState aState);
public int getMaxGameNum();
public void setMaxGameNum(int num);
public int getGameNum();
public void setGameNum(int num);
}
```

### A.4 Interface Feature

```
/**
 * Interface Feature translates game states into feature vectors.
 *
 * Method prepareFeatVector(so) returns the feature vector
 * for state so. Child classes have usually constructors accepting a
 * single argument 'featmode'. The argument 'featmode' allows to
 * construct different flavors of Feature objects.
 * The acceptable values for 'featmode' in a certain child class
 * are retrieved with getAvailFeatmode().
 */
public interface Feature {
public double[] prepareFeatVector(StateObservation so);
public String stringRepr(double[] featVec);
public int getFeatmode();
public int[] getAvailFeatmode();
public int getInputSize(int featmode);
}
```

## A.5 Interface XNTupleFuncs

```
/**
 * Interface XNTupleFuncs contains game-specific functions
 * for using n-tuple sets.
 */
public interface XNTupleFuncs {
public int getNumCells();
public int getNumPositionValues();
public int getNumPlayers();
public int[] getBoardVector(StateObservation so);
public int[][] symmetryVectors(int[] boardVector);
public int[][] fixedNTuples();
public HashSet adjacencySet(int iCell);
}
```

## A.6 Abstract class Evaluator

```
/**
 * Evaluates the performance of a PlayAgent in a game.
 */
abstract public class Evaluator {
protected PlayAgent m_PlayAgent;
protected int verbose=1;
public Evaluator(PlayAgent e_PlayAgent, int stopEval);
public Evaluator(PlayAgent e_PlayAgent, int stopEval, int verbose);
public boolean eval();
public boolean goalReached(int gameNum);
public boolean setState(boolean stateE);
public boolean getState();
public String getMsg();
public String getMsg(int gameNum);
abstract protected boolean eval_Agent();
abstract public double getLastResult();
}
```

## A.7 Abstract class Arena

```
/**
 * This class contains the GUI and the task dispatcher for the game.
 * The GUI for buttons and choice boxes is in {@link XArenaButtons}.
 *
 * Run this class from the {@code main} in {@link LaunchArenaTTT})
 * for the TicTacToe game.
 *
 * @author Wolfgang Konen, TH Köln, Nov'16
 */
abstract public class Arena extends JPanel implements Runnable
{
public enum Task {PARAM, TRAIN, MULTTRN, PLAY, INSPECTV
                  , COMPETE, SWAPCMP, MULTCMP, IDLE  };
protected GameBoard gb;
public Task taskState;
public Arena();
public Arena(JFrame);
public void init();
public void run();
public void PlayGame();
public void enableButtons(boolean state);
public void setStatusMessage(String msg);
public StatusBar getStatusBar();
public Feature makeFeatureClass(int featmode);
public XNTupleFuncs makeXNTupleFuncs();
abstract public String getGameName();
abstract public GameBoard makeGameBoard();
abstract public Evaluator makeEvaluator(PlayAgent pa, GameBoard gb,
               int stopEval, int mode, int verbose);
abstract public void performArenaDerivedTasks();
}
```

## A.8 Abstract class ArenaTrain

```
/**
```

```
 * This class contains the GUI for the arena with train capabilities.
 * It extends the task dispatcher of Arena with method
 * performArenaDerivedTasks() which contains tasks to trigger functions
 * for agent learning, parameterization, inspection and so on.
 *
 * @author Wolfgang Konen, TH Köln, Nov'16
 */
abstract public class ArenaTrain extends Arena
{
public ArenaTrain();
public ArenaTrain(JFrame frame);
public void performArenaDerivedTasks();  // extend task dispatcher
protected void InspectGame(); // inspect agent X on game positions
}
```

# B Appendix: Other Game Value Functions

Sec. 3.2 and 3.4 have introduced with `PlayAgent.getScore(StateObservation so)` the main function to retrieve a game value. There are two other functions delivering a game value; they are only required for more advanced needs:

- Interface StateObservation delivers with `getGameValue()` a game-specific (and hard-coded) version of a game value heuristic. For example, the game of chess has a heuristic based on weighted piece counts which delivers a rough estimate of the game value. If a good heuristic is known, this option is simple to implement. [8] If such a heuristic is not known, `getGameValue()` might simply return `getGameScore()`.

- Interface PlayAgent delivers with `estimateGameValue(sob)` a more flexible approach: Such a function can be made trainable / adjustable from previous experiences that the agent has made. For example, a class implementing PlayAgent might learn the game value function from self-play (via trainable weights or via a neural net). The weights have of course to operate on something, therefore a set of features extractable from StateObservation is necessary.[9] These features, normally represented by

[8] It is of course a bit against the idea of general game playing, since knowledge about the game tactics is in one way or the other implicitly coded in such a heuristic. But the same argument applies to feature construction as well.

[9] The raw state could be used as feature in principle, but it will be in most cases too difficult to establish a mapping from raw features to game value.

an object implementing Feature, can be something like number and kind of pieces, number of empty cells and so on. See Sec. 3.5 for more information about interface Feature.

A potential use of `sob.getGameValue()` or `pa.estimateGameValue(sob)` is to compute in MC or MCTS the final value of a random rollout in cases where the rollout did not reach a terminal game state (since the episode lasts longer than the 'Rollout depth' as it is for example in 2048 often the case).

It is dependent on the class implementing PlayAgent what

```
estimateGameValue(sob)
```

actually returns. If it is too complicated to train a value function (or if it is simply not needed, because for a game like TicTacToe we come always to an end during rollout), then `estimateGameValue(sob)` may simply return `sob.getGameValue()` or `sob.getGameScore()`.

If we integrate a trainable game value estimation into a class implementing PlayAgent, then agents that formerly did not need training (Minimax, MC, MCTS, ...) will require training. They should be after construction in AgentState INIT. How the training is actually done depends fully on the implementing agent.

Is there a need to distinguish between an PlayAgent's `pa.getScore(sob)` and `pa.estimateGameValue(sob)`? – Yes, it is! For a PlayAgent that uses `estimateGameValue(sob)` inside `getScore(sob)` (as it may be the case for Minimax, MCTS and MC), it is necessary to override its method

```
estimateGameValue(sob)
```

in such a way that it does not make a call to `getScore(sob)`. Otherwise an infinite loop would result.

# C Appendix: N-Tuples

## C.1 Board Cell Numbering

Each n-tuple is a list of board cells Lucas [2008]. Board cells are specified by numbers. The canonical numbering for a rectangular board is row-by-row, from left to right. For example, a $4 \times 4$ board would carry the numbers

```
00 01 02 03
04 05 06 07
08 09 10 11
12 13 14 15
```

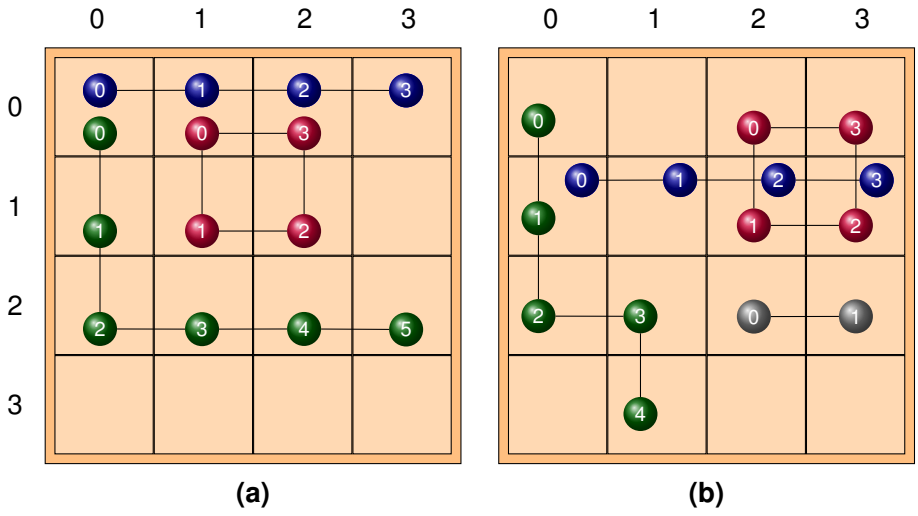


Figure 2: Two examples for n-tuples: (a) 3 n-tuples, (b) 4 n-tuples of varying length and placement.

Other (irregular) boards may carry other (user-specified) cell numbers. Each choice of numbering is o.k., it has only to be used consistently throughout the game.

### C.2 N-Tuple Creation

Fig. 2 shows two examples of $4 \times 4$ boards with fixed (user-specified) n-tuple sets. The canonical cell number is obtained from

$$4 \times \text{row\_number} + \text{col\_number}$$

Example (a) would be coded in a class derived from XNTupleFuncs as

```
public int[][] fixedNTuples() {
  int nTuple[][]={ {0,1,2,3}, {1,5,6,2}, {0,4,8,9,10,11} };
  return nTuple;
}
```

Example (b) would be coded as

```
public int[][] fixedNTuples() {
```

```
    int nTuple[][]={ {4,5,6,7}, {2,6,7,3}, {0,4,8,9,13}, {10,11} };
    return nTuple;
  }
```

Each n-tuple contains each cell at most once. But different n-tuples may (and often should) contain the same cell multiple times.

Fixed n-tuples are a user-specified way of creating n-tuples. It is also possible to let the n-tuple factory build **random** n-tuples:

1. **Random points**: Cells are picked at random, no cell twice[10], no topographical connection. This is often not advantageous because in many board games the neighborhood of a cell is more important for determining its value than an arbitrary other more distant cell.

2. **Random walks**: Cells are picked at random, no cell twice, with adjacency constraint. That is, each cell of the n-tuple list must be *adjacent* to at least one other cell in the n-tuple. What *adjacent* actually means in a certain game is specified by the user through the XNTupleFuncs method

   ```
   /* Return all neighbors of cell number iCell */
   public HashSet adjacencySet(int iCell);
   ```

### C.3 N-Tuple Training and Prediction

How are the n-tuples used to generate features? – Each n-tuple has an associated look-up table (LUT) of length $P^n$ where $n$ is the n-tuple length and $P$ is the number of position values[11] each cell might have. Consider the example of TicTacToe with 3 cell position values `{X,-,O}` ($P = 3$) leading for an n-tuple of length $n = 2$ to $3^2 = 9$ possible LUT entries

```
XX,   -X,   OX,
X-,   --,   O-,
XO,   -O,   OO
```

These are features. Even for a small number of n-tuples this will generate quite a large number of features. For example in Fig. 2(a), the number of features is $3^4 + 3^4 + 3^6 = 891$. On a larger board, a more realistic setting would be 40 n-tuples of length 8, resulting in $40 \cdot 3^8 = 262\,440$ features.

Each feature $i$ in n-tuple $\nu$ has an associated weight $w_{\nu,i}$. Given a certain board state, we look first which of those features are active ($x_{\nu,i} = 1$) or inactive

[10] within the same n-tuple

[11] the number that XNTupleFuncs method `getNumPositionValues()` returns

($x_{\nu,i} = 0$) in that board state. Then the n-tuple network computes its estimate $V^{(est)}$ of the game value through

$$V^{(est)} = \sigma\left(\sum_{\nu=1}^{m}\sum_{i=0}^{P^n-1} w_{\nu,i}x_{\nu,i}\right) \tag{1}$$

which is simply a linear neural net with sigmoid function $\sigma(\cdot)$. We compare the estimate generated by this net with the target game value $V$ prescribed by TD-learning. A $\delta$-rule learning step with step-size $\alpha$ (gradient descent) is made for each weight in order to decrease the perceived difference $\delta = V - V^{(est)}$ between both game values (Thill et al. [2014], Thill [2015]).

For complex games it might be necessary to train such a network for several hundred thousand or even million games in order to reach a good performance. The so-called **eligibility traces** are a general technique from TD-learning to speed up learning. They can be activated in the GBG framework by setting parameter $\lambda > 0$ in the *TD pars* parameter tab. Further details on eligibility traces are found in Thill et al. [2014].

Once the network is trained, the game value estimate $V^{(est)}$ is used to decide about the next action.

To further speed up learning, symmetries may be used: **Symmetries** are transformations of the board state which lead to board states with the same game value. For example in the case of TicTacToe, there are eight symmetries: the board state itself plus its three $90^o$ rotations and the mirrored board state with its three $90^o$ rotations. Instead of performing only *one* learning step with the board state itself, one can do *eight* learning steps with all symmetric states. This may greatly speed up learning, since more weights can learn on each move and the network generalizes better.

The knowledge of the symmetric states is game-specific. The user has to code it in XNTupleFuncs method

```
public int[][] symmetryVectors(int[] boardVector);
```

See Sec. 3.6 and Sec. 4.6 for further information on this method.

## References

S. Bagheri, M. Thill, P. Koch, and W. Konen. Online adaptable learning rates for the game Connect-4. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):33–42, 2015. doi: http://dx.doi.org/10.1109/TCIAIG.2014.2367105. URL http://www.gm.fh-koeln.de/~konen/Publikationen/Bagh15.pdf. 13, 20

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. 6

S. L. Epstein. Learning to play expertly: A tutorial on Hoyle. *Machines that learn to play games*, pages 153–178, 2001. 5

Michael Genesereth and Michael Thielscher. General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(2):1–229, 2014. URL https://wiki.eecs.yorku.ca/course_archive/2014-15/F/4412/_media/game_playing.pdf. 5

Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2): 62, 2005. URL https://vvvvw.aaai.org/ojs/index.php/aimagazine/article/download/1813/1711. 5

W. Konen. Reinforcement learning for board games: The temporal difference algorithm. Technical report, Research Center CIOP (Computational Intelligence, Optimization and Data Mining), Cologne University of Applied Sciences, 2015. URL http://www.gm.fh-koeln.de/ciopwebpub/Kone15c.d/TR-TDgame_EN.pdf. 9

John Levine, Clare Bates Congdon, Marc Ebner, Graham Kendall, Simon M Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General video game playing. Technical report, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013. 6

Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification, 2008. 5

S. M. Lucas. Learning to play Othello with n-tuple systems. *Australian Journal of Intelligent Information Processing*, 4:1–20, 2008. 13, 20, 30

J. Mańdziuk and M. Świechowski. Generic heuristic approach to general game playing. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 649–660. Springer, 2012. 5

J. Méhat and T. Cazenave. Combining UCT and nested Monte Carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277, 2010. 5

D. Michulke and M. Thielscher. Neural networks for state evaluation in general game playing. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 95–110. Springer, 2009. 5

M. Thill. Temporal difference learning methods with automatic step-size adaption for strategic board games: Connect-4 and Dots-and-Boxes. Master thesis, Cologne University of Applied Sciences, June 2015. URL http://www.gm.fh-koeln.de/~konen/research/PaperPDF/MT-Thill2015-final.pdf. 13, 20, 33

M. Thill, S. Bagheri, P. Koch, and W. Konen. Temporal difference learning with eligibility traces for the game Connect-4. In Mike Preuss and Günther Rudolph, editors, *CIG'2014, International Conference on Computational Intelligence in Games, Dortmund*, 2014. URL http://www.gm.fh-koeln.de/~konen/Publikationen/ThillCIG2014.pdf. 13, 20, 33