TODOs for GBG-Framework – WK/08/2017

## From TR-GBG.pdf (is commented out there)

* The n-tuple agent developed for C4 (Connect Four) needs to be ported to GBG.
* Allow only trained agents to be saved.
* Optional game visualization and game logging during competitions as well
* Clarify: Is the parameter data flow safe, if we issue a ’play’ or ’compete’ for 2 agents of same type but with different parameters?
* Or what is with the TD pars tab when we have a TD agent for X and a TDNTupleAgt for O? Both have their (possibly different) TD parameters (!)

## Solved items

* OK Bug fix: The current LoadAgent version ‘hangs’ if an agent with incompatible serialID is read (TTT agents TDS & TD-NTuple).
* OK Bug fix in TDAgent.java: We needed to replace the hard-coded   
  private int inpSize[] = { 6, 6, 10, 19, 13, 19, 0, 0, 0, 9 };  
  by the flexible m\_feature.getInputSize(m\_feature.getFeatmode()).
* OK TD-pars tab: Constrain ‘Feature set’ (= value for featmode) to the set of allowed featmode values for the current game (m\_feature.getAvailFeatmode()) and set the initial value to a sensible value 🡪 KG fixed this and generated a pull request
* OK: BUG1 + BUG2 fix for MCAgent: In some versions of Hex, the MC agent does not choose the best action, but **instead the worst** action (!!) This happens for (3x3, 1.), (4x4, 2.), (5,5, 1.), … and so on. The same behavior for TTT (3x3, 1.) – This was due to 2 errors: (1) in RandomSearch there could be actions on newSob even if newSob is already a game-over-state (BUG1) and (2) newSob.getGameScore has to be called with the referingState sob to get the sign right in every case (BUG2)
* OK Improved TR-GBG around game score and game value: Made it simpler!
  + Concentrate Chap. 3.3 on so.getGameScore() and so.getGameScore(sob). Explain the difference in pictures. Give examples showing explicit values for both functions.
  + Move pa.getGameScore(so) to Chap. 3.4 (Game Value), because it is a game value, not a game score (think about renaming it to pa.getGameValue).
  + Move the mind-buggling complicated so.getGameValue() and pa.estimateGameValue(so) to appendix.
* OK XArenaFuncs::multiCompete writes Arena.comp.csv now to agents/<gameName> (and not to root of GBG)
* OK XArenaFuncs::multiCompete has the evaluator mode hard-wired (9). This might fail for other games where the evaluator mode 9 is not present. Extend it that it fetches static Evaluator.getDefaultEvalMode().
* OK Why has XArenaFuncs::multiCompete the evaluator part excluded in case of agent MCTS? – This was due to a wrong (incomplete) implementation of MCTSAgentT::getScore (the part with nextActionScore, same as in MCAgent, was missing). -- Additionally, there was another bug in both MCAgent and MCTSAgentT: Both would have problems when getScore(sob) was called with an already game-over sob (as it regularly happens inside Evaluator9). Then MCTSAgentT resulted in a NullpointerException, since the tree of a root state with game-over condition is never expanded in treePolicy. The fix is to check in getScore(sob) for sob.isGameOver(). If true, return directly sob.getGameScore(sob). Now both bugs are fixed and MCTS is correctly evaluated in multiCompete.
* OK Improve Evaluator chapter in TR-GBG.
* OK Write for TR-GBG.pdf an appendix on N-tuples (with figures): Explain for what the several functions in XNTupleFuncs are needed.
* OK Types.ACTIONS: replace enum (which needs to be extended whenever we need more ACTIONS) by some class construct
  + a class having an int as member
  + we delete fromString, fromVector, they are never used in GBG
  + older logs (with enum ACTIONS) are not readable anymore
* OK Bugs in Hex (for KG):
  + OK When playing a human-agent game, then the scores of the agent are not shown during play (they are however shown when replaying it with the game log)
  + OK Hex: extend to logging with subdirs 🡪 KG
  + OK Under Hex, both Arena and ArenaTrain open on “Load Agent” the wrong directory (it should be agents/Hex/<num> where <num>=BOARD\_SIZE)
* OK Saving (serializing) of things (logs, agents) was not safe, if something changes in the classes being serialized. Is it possible (with minimal effort) to read older versions of serialized objects as well? – It has become better with the proper definition of [serialVersionUID](#serialVersionUID).
  + A further improvement might be to catch an InvalidSerialVersionUID exception and react on this with transforming from an older version? – Does not work, because the class name is stored and will not fit to the new version. If we want to re-use older versions, we have to transform them, see **TransformTdAgents** for an example transforming v12-TDNTupleAgt’s to v13.
* OK Added   
   static final long **serial****VersionUID**

to serializable classes to ease the serialization process. Then the **serialVersionUID** is no longer compiler-dependent. We may add functions to a serializable class w/o unwanted change in serialVersionUID. We may add or delete single members and still read old versions, as long as the new members can be set from suitable defaults.

* OK MCTSAgent-Design-Flaw: It has params, SinglePlayer m\_mcPar and the single elements 🡪 too much!!! See if we can delete params 🡪 yes, we did. And we let the setters for single params change m\_mcPar as well. And we added getters for MCTSAgentT. Do we need setters in MCTSAgentT (or do we go always through constructor)?
  + We have some getters in MCTSAgentT, but we can use mctsPlayer.getParMCTS() for the others
  + We do not need setters.
  + We have now less parameters in MCTS since we upgraded to the ParMCTS version.
* OK MCTSParams: bug in getK\_UCT(): intValue() 🡪 now corrected to doubleValue()
* OK MCTSParams tab: when changing params and directly saving MCTS, the params are not taken. Only after a “Play” the params are taken over (!) 🡪 saveAgent in XArenaMenu now uses fetchAgents first!
* OK An object of class Evaluator is currently constructed in several places (multiCompete, multiTrain, train, and taskState TRAIN, menu item ‘Quick Evaluation’ (fct evaluate)), often with different objects (different evaluator modes), and with modes tied to game TicTacToe. Make the modes as simple as possible, then generalize it in such a way that it is for arbitrary games. Perhaps with user-adjustable mode selectors in the ‘Other pars’ tab, perhaps with sensible defaults set in the files defining constants. 🡪 see [#New Evaluator Concept](#_Evaluator_concept)
* OK During TDNTupleAgt-training: Replace the Minimax-evaluation by proper general evaluation (replace JFreeChart plot title)
* OK Disable “Output sigmoid” in TD pars when TDNTupleAgt is selected (but check it)
* OK What is NORMALIZE in TD pars? Add tooltip text. – NORMALIZE is for normalizing CurrentScore = so.getGameScore(…) in getNextAction, such that it fits to the range of the prediction of TD- or TD-NTuple-agent ([0,1] for Fermi fct in TD, [-1,1] for tanh in TDNTupleAgt)
* OK Update HelpGUI-Arena-GBG.htm
* OK [MCTSParams-vs-ParMCTS fixes](#MCTSParams_vs_ParMCTS_fixes). Now also for MCAgent (ParMC). Added member ParOther m\_oPar to MCAgent and MCTSAgentT. Now also for TDAgent: member TDParams m\_tdPar transformed to ParTD.
* OK Minimax on 4x4-Hex: InspectV shows an initial value function with some values ‘0’. Why? – Because the Minimax default depth 10 is not enough. Enhancing to depth=12 solves the problem.
* OK Extend all classes derived from XNTupleFuncs by a new member serialVersionUID. Use this serialVersionUID which is stored in the saved agents. (We need this serialVersionUID, otherwise the tiniest change in XNTupleFuncsXX will invalidate all saved agents.)
* OK When replaying a human-agent game with the game log, then any ADVANCE will move two plies forward (one agent, one human ply).[[1]](#footnote-1) When replaying an agent-agent game, it is correctly only one ply per ADVANCE. – This was a bug in the logging implementation: logManager.add(…) was only part of the if(gb.isActionReq())-part in the play-while-loop. It was missing in the human-move-part. Now solved: Added a proper statement logManager.add(…) to all GameBoard implementations in function HGameMove (or similar). This required two new getter-functions in Arena: getLogManager() and getLogSessionID(). Done.
* OK Hex:
  + OK If agents/Hex does not exist (Arena.comp.csv), then create it
  + OK Wrote in the Javadoc of StateObserverHex (a) how the tile [i,j] translates to board cells and (b) how the board numbering in XNTupleFuncs::getBoardVector translates to the board.

## Partly solved items

* (OK) It is very disturbing that TDAgent cannot learn the trivial 2x2 Hex well. Even with the so far best feature mode 2, the value functions for the clearly loosing start moves are still too high. Why? – And even worse, if we add with feature mode 99 a set of features exactly equivalent to TDNTupleAgt (only w/o symmetry), we do not get good results, although we have a lot of features. What is different then between TDAgent and TDNTupleAgt? – It has another sigmoid (but the difference happens also w/o sigmoid). It has another epsilon-descent. What else??  
  🡪 now partly solved, see [#Debugging TDS (TDAgent)](#_Debugging_TDS_(TDAgent))
* (OK) NTuple[2]::getBoard is still specific to TTT (vector of length 9). Generalize! – Now set to @Deprecated, since this function is never used (both classes)

## Open items

* Prepare TR-GBG.pdf for publication on CIplus server.
* TR-GBG.pdf: new sections on important reoccurring questions
  + How to find good features?
  + How to evaluate an agent?
  + Extend Evaluator chapter
* (OK) Think about the game logic for so.getGameScore(referingState) in 3-player games 🡪 see [notes\_MCTS.docx](file:///C:\WUTemp\FH-MassenDaten\svnSoma\trunk\doc\CaseStudies.d\201314.d\CIG2014\MCTS.literature\notes_MCTS.docx), Chap. “Getting The Score For n-Player Games”. – Still some thinking about intermediate game scores to do.
* Extend the logic of getGameScore(StateObservation referingState) accordingly to 3- or 4-player games
* If MCTS has several actions with the same value (all are a ‘Win’), then it does not take the shortest path to victory. Instead it takes a random among all winning moves. 🡪 KG will think about an add-on to value function which breaks ties in favor of a shorter path.
* Some elements of measurement in methods train, multiTrain, and compete are still TicTacToe specific or at least 2-player specific. Generalize them to arbitrary 2-player games and later to arbitrary n-player games.
* Improve GUI layout for Arena and ArenaTrain
  + Arena: fill the empty space with s.th. sensible (e.g. game name)
  + Arena: announce which column is for which player (‘Param X’ and so on from ArenaTrain is missing here). Optional: show the color of the player
  + Move button MultiTrain in the ‘Agents trained’ row and move button ‘Logs’ in the blue buttons row.
  + Arena: skip the two rows ‘Train games’ and ‘Agents trained’
  + Arena: there is no way to select Human Player via agent select box. Should we allow to select all agents (or all agents which need no training) via select box?
  + think about nicer colors
* Hex, Multi-Competition: seems that wrong agents compete (MCTS when MC is desired) 🡪 check multiCompete, check whether its training is o.k.
* Things for MCTSExpectimax (for JK):
  + OK TreeDepth <= 3 leads to NullPointerException (s.th. around   
    return uct().treePolicy().treePolicy();) 🡪 Bug is gone in new version
  + Why is TreeNode-constructor called exactly nIterations (3500) times (at least for TreeDepth=10)?
  + Why is MCTS(Expectimax) with TreeDepth=1 not as good as MC? Which role plays K (UCT)? Does it help to set it to large values?
* ? Competitor class to make the code parts dealing with competitions more similar / more safe
* Bug (GUI hangs) when trying to load an old MCTS agent (v12). Happens only sometimes, only when running, not (!) in debugger. Probably gone with v13.
* Bug (GUI hangs) when loading with Hex several time an TDNTupleAgt. It passes the [LoadAgent] status message, but then it returns from the action listener in such a way that the GUI does no longer respond to any events.
* Replace PlayAgent::wasRandomAction() with ACTION::isRandomAction() (simpler interface, cleaner code). Delete agent’s member randomSelect, where present.
* Think about the parameter ‘player’ in the getGameScore and getScore functions for the different TD agents. Is it correctly and consistently used everywhere?

## equals() or hasEqualState()?

* Design discussion: Is it safe to have StateObservation::equals() for checking that two states are the same?
  + pro: If we have it this way, we can use all other code (e.g. Set, List) which builds upon equals()
  + con: There is always the default implementation Object::equals() which has a different meaning (‘equal’ means that two objects are at the same location in memory. Nevertheless then, two different objects can have the same state). A developer of a new class implementing StateObservation might forget to re-implement equals()
* Another possibility would be to add a new function   
   hasEqualState(StateObservation arg0)

to the StateObservation interface. Then no one can forget to implement this. To have also the “pro” of the 1st solution, it is possible to define equals() as well, if it is needed:

public Boolean equals(Object arg0) { return hasEqualState(arg0);}  
(with an appropriate cast from Object to StateObservation to be added)

So we should change any usage of equals to the mandatory usage of hasEqualState()

Similarly with toString() and stringDescr()

## New Evaluator Concept

Status 06/2017: Where and how is Evaluator used?

* ArenaTrain, derived task TRAIN: after training, an Evaluator with mode=2 is constructed and evaluated 🡪 should be mode getQuickEvalMode()
* XArenaFuncs, train() & multiTrain():
  + m\_evaluator1: Evaluator with mode 9
  + m\_evaluator2: Evaluator with mode 2
  + m\_evaluator3: Evaluator with mode 0 (only multiTrain())
* XArenaFuncs, multiCompete():
  + m\_evaluatorX: Evaluator with default mode (for X)
  + m\_evaluatorO: E valuator with default mode (for O)
* XArenaMenu, evaluate(), which is only called by ‘Quick Evaluation’:
  + evaluator2: Evaluator with mode 2
* Disadvantage: The mode values are hard-coded (perhaps only appropriate for TicTacToe)
* The different compete menu items should be disabled for 1-player games and generalized appropriately for (n>2)- player games
* Arena2048, ArenaTrain2048: both have some strange makeEvaluator functions which compare the constant (!) ConfigEvaluator.Evaluator with 0 or 1 (🡪 compiler warning & can only be changed after recompile)
* Nicer design: it would be better to use parameter mode of Evaluator constructor / of makeEvaluator()
* EvaluatorHex: eval\_Agent chooses between three different functions competeAgainstMinimax, competeAgainstMCTS, and competeAgainstRandom at compile time
* Nicer design: Have three different Evaluator objects which decide via a mode variable which competeAgainst… function is used. This has two advantages
  + Several competeAgainst… functions can be evaluated in parallel
  + The decision, which Evaluator object should be taken can be made at run time

**The new and more general Evaluator concept**:

How should the new Evaluator usage be?

* There are five new helper functions in Evaluator interface
  + int getQuickEvalMode()
  + int getTrainEvalMode()
  + int getMultiTrainEvalMode()
  + String getPrintString() (e.g. "success rate (randomAgent, best is 0.9): ")
  + String getPlotTitle() (e.g. "success against Minimax")
* When starting ‘Quick Evaluation’, only one Evaluator with mode=getQuickEvalMode() is constructed and evaluated
* When starting TRAIN, there might be two evaluators, if getQuickEvalMode() and getTrainEvalMode() return different int’s. If they return the same int, only one evaluation is done. In any case, the QuickEval Evaluator is used to add a point to the JFreeChart plot and to check whether goal is reached (and if so, break out of training loop). The other evaluator with mode getTrainEvalMode() is only used for additional reporting at the end of training.
* When starting MULTITRAIN, there might be similarly one, two or three evaluators, depending on whether getQuickEvalMode(), getTrainEvalMode() and getMultiTrainEvalMode() return different int values.
* When starting multiCompete(), both agents, X and O, are evaluated with Evaluator having mode getQuickEvalMode().
* The helper functions getPrintString() and getPlotTitle() are for convenience, to customize printout and plot axis title.

So the simplest usage of the Evaluator concept would be: have only one mode value implemented and let getQuickEvalMode(), getTrainEvalMode() and getMultiTrainEvalMode() return this one mode value.

More advanced: two or three different evaluator modes, allowing evaluation from two or three different perspectives simultaneously.

TODO: It would be nice, if a Train Evaluator is shown in JFreeChart plot as well, but on a secondary y-axis. (This needs to understand how to plot in JFreeChart *dynamically* one item to the first, one to the secondary y-axis)

OK: Have in the “Other Pars” tab two choice boxes “Quick Eval Mode” and “Train Eval Mode”, with initial values getQuickEvalMode() and getTrainEvalMode(), resp. This offers the possibility to change these eval modes to any available eval mode (what getAvailableModes() returns). With the effect that the mode values used in ‘Quick Evaluation’ and during training change accordingly to the values in the choice boxes.[[2]](#footnote-2)

A possible later add-on: Let eval() return an object of class EvalResult which has getters getSuccess(),goalReached(), getLastResult(), getMsg(). Why? Consistent object, all information is kept together.

## MCTSParams vs. ParMCTS

Kevin Galitzki correctly pointed out that MCTSParams with its GUI being part of MCTSAgentT (and similarly in other agents) is not a good idea. Indeed, this may be the reason that GBG crashes after many MCTSAgentT constructions (“Too many handles in Windows OS”).

This is clearly a design flaw of GBG. MCTSParams is far too big to be part of other objects. And it is unpleasantly to debug, one cannot ‘see’ the parameter values in the debugger.

The new solution:

* Have for MCTSParams a sister class ParMCTS with only the parameter values as members. If there are elements static <type> DEFAULT\_... in MCTSParams, put them into ParMCTS.
* Have a constructor ParMCTS(MCTSParams) and a method MCTSParams::setFrom(ParMCTS) to copy the parameter data in either direction.
* Replace member MCTSParams in MCTSAgentT with ParMCTS. Change the serialVersionUID of MCTSAgentT to a higher ID. (Write down beforehand the parameters of all saved agents you want later to reconstruct).
* For the moment we live with the situation that older serialized objects won’t be loadable after the change XXParams 🡪 ParsXX. [[3]](#footnote-3)
* Adapt the constructors of MCTSAgentT (at least a new one with ParMCTS instead of MCTSParams)
* Eliminate all calls to MCTSParams other than the real fetching from the multi-tabbed window

Similar solutions for other parameter classes XXParams.

OK Done for MCTSAgentT, MCTSParams 🡪 ParMCTS.

OK Done for TDNTuple[2]Agt (TDParams, NTParams). In addition, with the help of TransformTdAgents and TDNTupleAgt\_v12 and a one-time call from LaunchTrainTTT constructor: all TDNTupleAgt’s saved to disk are transformed to the new version with serialVersionUID=13L. For safety, the old version v12 of TDNTupleAgt and all agent files v12 are stored in GBG-backup/v12 (can be deleted later).

OK There was a bug in the transformed TDNTupleAgt agents: The members m\_MaxGameNum, m\_NumEval and epochMax from AgentBase were not transformed properly to the new agents. Now the constructors TDNTupleAgent(TDNTupleAgent\_v12) and TDNTupleAgent\_v12(TDNTupleAgent) are extended accordingly and the whole transformation was repeated.

OK Done for MCAgent and TDAgent. In addition, TDAgent, MCAgent and MCTSAgentT get new parameter m\_oPar, for the sake of completeness

TODO: MCTSExpectimaxParams and associated agents. Minimize the calls and functions using still NTParams and TDParams to only the places directly connected with the GUI. In other places we may substitute ParTD.

## Debugging TDS (TDAgent)

Kevin Galitzki found out that TDS cannot learn even the simplest Hex game, although TDNTupleAgt can. It is not well understandable that TDS has even on the most trivial 2x2 Hex such a bad behavior (value function in InspectV). This is even more pronounced when we add to FeatureHex a feature mode 99 which should be equivalent to a 4-tuple in the case 2x2. Still no good result (frankly speaking, even worse than feature mode 2).

So what are the reasons?

We compared in detail the code of TDAgent and TDNTupleAgt and found the following differences / bugs.

1. TDS had the Fermi function as **sigmoid** (if “with sigmoid”) is activated, while TDNTupleAgt has **always** tanh sigmoid. We extended TD\_func.java with a source code switch boolean FERMI\_FCT = true, false to allow either Fermi function or tanh. We corrected also a bug around (unconditional) normalization in TDAgent. Now we do a normalization only if NORMALIZE, and with min/max values appropriate to the actual FERMIC\_FCT value. With this we get for feature mode 2, 50.000 training games:
   1. good results for linear, bp, without sigmoid (value func >580 for winning tiles, <90 for loosing tiles)
   2. bad results for linear, bp, FERMI\_FCT=true sigmoid (value func >700 for winning tiles, but ≈ 600 (!) for loosing tiles)
   3. good results for linear, bp, FERMI\_FCT=false sigmoid (value func ≈ 800 for winning tiles, <90 for loosing tiles)

The results in b. are even slightly worse (≈ 900 (!) for loosing tiles) if we activate NORMALIZE.  
The results in c. are the same if we activate NORMALIZE. No wonder, since NORMALIZE does not change anything here (min and max of game score are the same as min and max of sigmoid function \in [-1,1])

So the result of this part is: Better have tanh-sigmoid (FERMI\_FCT=false) or no sigmoid, but definitely not Fermi function sigmoid

1. Rethinking of the **reward logic**: We have in both agents two places where we fetch the reward (game score from StateObservation when game is over)
   1. getNextAction
   2. trainAgent

In getNextAction, we advance the state by one of the available actions and want to select the one with highest CurrentScore. In the case of Hex (or other 2-player games) the score matters only if ‘Game over’. So we fetch in that case NewSO.getGameScore() (which is ‘-1’ in case of a win) and multiply it by (-1). Equivalently, we fetch NewSO.getGameScore(so), where ‘so’ is the predecessor of NewSO. Since ‘so’ and NewSO are always opposite, this multiplies the raw game score by (-1) as well. Either way, CurrentScore is +1 whenever a game is won.   
If the game is not over, we set CurrentScore=player\*getScore().

In trainAgent, we advance the state ‘so’ via the action suggested by getNextAction. If the game is over, we want to give a reward to the net. But the reward is not exactly the same as CurrentScore (!): reward = player\*so.getGameScore(oldSO). This is +1 for an X-win, and -1 for an O-Win (X: player=+1, O: player=-1).

Problematic in trainAgent: If game is not over, we set reward=0.0. This is no problem for 2-player games like Hex or TTT. But is it correct for 1-player games like 2048??? Shouldn’t it be the score accumulated so far? S.th. for later, we are now concerned with Hex.

Anyway, the agent is trained to let the score of predecessors of the game-over state ‘so’ become as similar as possible to the reward.

If the game is not yet over in getNextAction, we have CurrentScore=player\*getScore(). Should this be normalized as well? And if so, what is the source range, what is the target range? – TODO.

So the result of this part is: It is OK that CurrentScore and reward have slightly different formulas. We replace in all 4 instances (2 agents \* 2 functions) the complicated switch statement with one getGameScore(referingState). We have now exactly the same reward logic in TDNTupleAgt and TDAgent, if NORMALIZE==false. Needs some thinking for the case NORMALIZE==true, but for NORMALIZE==false everything is OK.

1. Why does TDS with feature mode 99 (a complete n-tuple for 2x2 board) not get the same performance as TDNTupleAgt?? Each state has a different and separate weight, so it should learn the true value function perfectly from the successor state. Why does it not work?
   1. Check the development of certain weights, whether they get a change in the same direction each time they are activated.
   2. Compare with the development of certain weights in TDNTupleAgt
   3. TDNTupleAgt has one weight table for each player, but this should be redundant here, since each board state is either only activated by X or by O.

First of all, we fixed a bug in createFeatureVector99 (wrong index calculation).

Secondly, we need much larger ALPHA, if input feature vector is long (81 values in case of feature mode 99)!! If we multiply the standard alpha values by about 100, i.e. ALPHA\_INIT=10, ALPHA\_FINAL=0.1, we get perfect results! So the **reason for not learning the trivial 2x2 board was a too small ALPHA**. This might be due to the fan-in division by length of input vector, which is active in TD\_Lin (TDAgent), but not active in TDNTupleAgt.

Thinking about the fan-in division: The divide by fan-in (which happens in TDAgent::setTDParams) is perhaps a good idea, if we expect each input unit to be (at least partly) active and if the sum of all N inputs is in the order of “N \* <AverageInput>”. But this is not the case in the “one-hot”-encoding of n-tuple-like features. There the activation sum is equal to the one active input, and alpha should NOT be divided by N. TODO: Have a parameter in the tab or a compiler switch to activate or deactivate alpha-fan-in-division.

So the result of this part is: **If we set the learning step size to ALPHA\_INIT=10, ALPHA\_FINAL=0.1**, we get perfect results for feature mode 99 on the 2x2 board. This holds for with / w/o sigmoid, linear or bp, NORMALIZE or not (!). To make adjustment of the learning rate easier in the general case, there should be a boolean switch “alpha-fan-in-division”.

Overall results: With the right ALPHA and feature mode 99 we get perfect results on 2x2 board. So it seems there is no bug in TDAgent. – But feature mode 99 is of course not transferable to larger Hex boards. Feature mode 2 shows a fair result on 2x2 board, but it is not perfect. And for larger boards we have not yet found reasonable “hand-made” features.

1. And when logging a 1-player human game there was no action recorded at all (!). [↑](#footnote-ref-1)
2. The third eval mode getMultiTrainEvalMode() is without choice box because it is probably only seldom used. [↑](#footnote-ref-2)
3. Several ideas to recover older saved agent versions do not work:

   * **Idea 1**: Save beforehand a sample MCTSAgentT with old serialVersionUID. Try to write code to load such an older version:
     + Catch the InvalidVersionUID exception
     + If we are in the catch block, try to get the version ID of the older object, cast it to a loader class MCTSAgentT\_v12 (a copy of the old class with old serialVersionID and old MCTSParams)
     + Copy from MCTSAgentT\_v12 to MCTSAgentT (is a shallow copy enough for all parts that did not change?). Use constructor ParMCTS(MCTSParams) for the part that changed.
   * A first analysis shows that it does not work this way:
     + The InvalidClassException is already thrown at ois.readObject() when the serialVersionUID of the class on file and the serialVersionUID of the local class MCTSAgentT do not match. That is, as soon as we change this, the object on disk is no longer readable.
     + The serialization mechanism can handle adding or deletion of class members, but not a change of field type. **Idea2**: But wait: If serialization can read an older MCTSAgentT object (with MCTSParams object but no ParMCTS object) when the serialVersionUID does not change (and it initializes ParMCTS with null), then we can read an older version w/o change in serialVersionUID. – No, this does not work really, since if the new local class has no MCTSParams object, then the information on the old MCTSAgentT object on disk w.r.t. MCTSParams is ignored (not read in), and thus no transformation is possible.
     + The only way would be to override the readObject function for the agents.   
       Or to transform any agent object to a special load/save object, e.g. MCTSAgentT\_v12, MCTSAgentT\_v13 prior to saving it to disk. If objects are saved to disk in this way, we first load the object from disk (deserialization), then check in a long switch statement the class of the loaded object (instanceof) and transform according to this class on to the ‘real’ class MCTSAgentT by copying (cannot be done through a simple cast).
     + This would require a lot of work and some maintenance costs. Perhaps something for later.

   [↑](#footnote-ref-3)