Strategy TDNTupleAgt:   
How does it work for 2048?

## Literature

[[Bagh14]](file:///C:\WUTemp\FH-MassenDaten\svnSoma\trunk\doc\CaseStudies.d\201314.d\IEEE-CIG-Paper.d\submission-REV2.d\TCL-C4-paper-R2.pdf): our TCIAIG-paper with TCL-EXP and other online algorithms

[[Thill14]](file:///C:\WUTemp\FH-MassenDaten\svnSoma\trunk\doc\CaseStudies.d\201314.d\CIG2014\Out.d\paperCIG2014.pdf): our CIG-paper with eligibility trace comparison

[[Jaskowski16]](file:///C:\WUTemp\FH-MassenDaten\svnSoma\trunk\doc\CaseStudies.d\201314.d\CIG2014\MCTS.literature\2048\Jaskowski2016-2048.pdf)

[[SzubertJaskowski-CIG2014]](file:///C:\WUTemp\FH-MassenDaten\svnSoma\trunk\doc\CaseStudies.d\201314.d\CIG2014\MCTS.literature\2048\paper2048-SzubertJaskowski-CIG2014.pdf)

[Konen2017b] ([TR-TDNTuple.tcp](file:///C:\user\wolfgang\www\Optimierung\TR_GBG\TR-TDNTuple.tcp))

[[Galitzki17]](file:///C:\user\datasets\Vorlesungen\FHK\DiplomArb\2014\BA%20Galitzki\BA-KevinGalitzki-final-2017.pdf.lnk)

[notes-WK-RubiksCube.docx](file:///C:\user\wolfgang\www\GameTheory\RubiksCube\notes-WK-RubiksCube.docx)

## Nomenclature

|  |  |
| --- | --- |
| st | state at time t |
| pt ∈ [-1,+1] | player **to move** at time t. For 1-player games: always pt=+1. |
| at | action taken at time t |
| s’t+1 | **afterstate** at time t+1 (applying at to st, but before random element) |
| st+1 | state at time t+1 (afterstate plus random element) |
| Rt+1 | total reward ( game score) for state st+1 at time t+1 (referring state: st) |
| rt+1 = Rt+1 - Rt | partial reward when moving from stto st+1 (referring state: st) |
| V(st+1) | game value, the agent’s expectation of the sum of future rewards (beyond rt+1) |

Initially, we had not yet the afterstate logic of [Jaskowski16] integrated, but meanwhile it is part of the new TDNTuple2Agt (switch AFTERSTATE). For deterministic games: s’t+1= st+1.

We describe the TDNTupleAgt-logic for both 1-player and 2-player games, to keep it general.

The total reward Rt+1 is the reward relative to st. That is, when player pt moves in st and the reward from the environment is =-1 (player pt+1 has lost), then Rt+1 (with st as reference) is (-1)\*(-1) = +1 (player pt has won). This is for 2-player games. For 1-player games, the reward relative to st is always equal to the reward from the environment.

For games which give a reward only in the terminal state (like TTT or Hex), the “expectation of the sum of future rewards” in V(st+1) is identical to the “expectation of the total reward”, i.e. the game value.

Currently V(st+1) is for 2-player games always the game value from the perspective of the 1st player (“X”). This is not very logical, since Rt+1 is measured with respect to referring state st and player pt. We will try in TDNTuple2Agt to get a more homogeneous realization of V(st+1).

## New Target Logic

Since the TDNTupelAgt did not work at all for 2048, but [Jaskowski16] had success with TD-Ntuple-learning, we re-investigated the target logic here and there and coded with switch NEWTARGET==true a version more similar to [Jaskowski16].

### getNextAction(st)

returns arg maxa CurrentScore(st,a), where CS = CurrentScore is:

1. OLD target logic (NEWTARGET==false)
   1. st+1 not terminal: CS = pt \* V(st+1)  
      (For 1-player games, this simplifies to V(st+1))
   2. st+1 terminal: CS = Rt+1 (with st as referring state)  
      (For 2-player games which are terminated by a winning move, like TTT or Hex, this simplifies to Rt+1=+1)
2. NEW target logic (NEWTARGET==true)
   1. CS = rt+1 + γ pt V(st+1)   
      (For 1-player games, this simplifies to rt+1 + γ V(st+1))

This is for the case when no random action is requested. If a random action is requested, each CurrentScore(st,a) will be set to a different random number, so getNextAction will return a random action.

The discount factor γ was missing in the old target logic of getNextAction(), for no specific reason. It is however sensible to include it in getNextAction() if we have it in trainAgent() (!). The impact on the results achieved so far is minimal, since we had nearly always γ=1.

### trainAgent(s0)

Learn or train the agent from one episode (an episode is a self-play-game starting from s0 and ending with a terminal state)

t=0

while (st not terminal) {

at = getNextAction(st)

st+1 = advance(st, at)

1. OLD target logic (NEWTARGET==false):
   1. st+1 not terminal: TDupdate(T= γV(st+1), s­t, λ)
   2. st+1 terminal: TDupdate(T= pt\*Rt+1, s­t, λ)
2. NEW target logic (NEWTARGET==true):
   1. TDupdate(T= pt rt+1+γV(st+1), s­t, λ)

}

if (NEWTARGET==true) TDupdate(T= 0, s­t, λ)

In the OLD target logic, the target for a pre-final state is to **predict the total reward** (times pt ∈ {-1,+1} for 2-player games, that means the total reward from the perspective of player +1).  
In the NEW target logic, the target is to **predict the extra reward** pt rt+1 for a pre-final state (that is a small number compared to the total reward in the case of 2048) and 0 for a final state.

This is for the case w/o random moves (ε=0).   
If ε>0, getNextAction() will be requested to return with probability ε a random action and (if learnFromRM==false) no TDupdate takes place after such a random action.

## Current drawbacks of TDNTupleAgt

Experiments with TDNTupleAgt and the game 2048 revealed that the current agent “TD-NTuple” has a number of drawbacks and deficiencies and can by no means create anything similar to [Jaskowski16]. These drawbacks are:

1. The current target logic is suspicious for games with continuous rewards like 2048. We implemented a new target logic (NEWTARGET==true) along the lines of [Jaskowski16] (see above), but this does not yet lead to success.
2. If we switch off the sigmoid (as it is done in [Jaskowski16]), we get with NEWTARGET after many training games diverging (infinite) weights (and thus a crash). What has to be done to avoid the unlimited weight growth?
3. [Jaskowski16] describes a better learning scheme with **afterstates s’** which should be implemented in TD-NTuple-2. We may use StateObservationNondeterministic for that purpose.
4. If we increase in 2048 the random n-tuples (number N, length L) from (N,L)=(10,3) to (10,4), the number of weights increases by a factor of 15 (due to the 15 position values in each n-tuple cell). This should cost more memory, but not more runtime. Instead we experience in training **runtime increase by a factor of 30** (!!??). Why is this so and can it be avoided by a better implementation? – The reason is that updateElig becomes very slow since there are as many traces as weights, and after a while many of them are non-zero. And that updateElig was erroneously called in NTupleValueFunc::calcScoresAndElig even in case λ=0 (now fixed in NTupleValueFunc 🡪 the case λ=0 becomes fast). But the problem remains for λ>0.
5. The current way of calculating eligibility traces in TDNTupleAgt is bound to be very slow for games like 2048 having many moves in one episode. [Jaskowski16] suggests another way of doing it by keeping only those traces where λk ≥ 0.1. We implemented this in TDNTuple2Agt.
6. Johannes Kutsch [reports a disturbing issue](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\mai-von-Kutsch-06\mai-von-Kutsch-06.msg) when training TDNTupleAgt for 2048: When training for 106 episodes with 20 3-tuples, there was an increase up to score 30.000 until episode 990.000, but then a sudden decrease to score 1.000 in the very last episodes. Why?
7. When all the deficiencies are resolved: Try to simplify the code where possible, in order to keep it maintainable.

To experiment on these issues, we create the copy classes TDNTuple2Agt, NTuple2ValueFunc and NTuple2 in subdir controller/TD/ntuple2 which we can modify and test in parallel to the former classes in subdir controller/TD/ntuple.

We first start to work on issues 4. and 5. For this, it is necessary to review the exact calculation of the n-tuple score in TDNTupleAgt (and connected classes).

It [turned out](#issue2And6), that these issues together with a few (other) bug fixes found when [#Debugging TDNTuple2Agt](#_Debugging_TDNTuple2Agt), solved issues 1., 2. and 6. as well. Finally, we will fix issue 3. as well in [#Afterstate Logic and Debugging](#_Afterstate_Logic_and), so that we have all issues solved when we solve issue 7. as well.

## N-tuple score

**getS****coreI()**, the n-tuple score function in class NTuple2ValueFunc, computes:

OLD **and** NEW target logic (NEWTARGET==false/true)

with S(st): the set of all states symmetric to st (including st itself), NS = |S(st)|

Indi(s): the index into the look-up table Vi[] of n-tuple i=1,…,m, given state s.

The **update function** **in case λ=0** (no eligibility trace) for each n-tuple i=1,…,m is

OLD target logic (NEWTARGET==false)

NEW target logic (NEWTARGET==true), **scaled α**:

with σ’(st): the derivative of the sigmoid σ() w.r.t. its argument at state st, that is (1-V2(st)) in case of σ()=tanh() and 1 in case of no sigmoid (σ()=1),

δt: the delta signal “T – V(st)” where the target T is the first argument of [TDupdate](#TDupdate) above.

**Special case no sigmoid and α=1**: In this case we have σ’(st)=1 and the NEW target logic update rule will add δt/(mNS) to each Vi. Plugging this into the [getScoreI-equation](#getScoreI_Eq) we see that the update rule with α=1 has consistently (for each number of n-tuples) the meaning: “Let become immediately the target ” :

This is why the scaling with 1/(mNs) (m=number of n-tuples) in the update rule makes sense.

(We note in passing that the OLD target logic misses the factor 1/mNS in the update rule, so a hypothetical update with α=1 will overshoot the target by a factor mNS (e.g. mNS = 80 for an 8-fold symmetry and 10 n-tuples).)

There was a possible bug in this update rule (see [below](#bugUpdateFormula)), which happened when two states in would activate the same LUT-entry of an n-tuple. We solved this by adding a bookkeeping mechanism which inhibited the update if it had been already updated once by another member of .

## Eligibility traces: The case λ ≥ 0

The general TD(λ) rule for weights wt is:

Normally, this formula is realized in TD(λ)-algorithms with the help of eligibility traces and t0=0 is used. But eligibility traces are unpractical (prohibitively slow) in cases like 2048 (long episodes, millions of weights).

[Jaskowsi16] has the brilliant idea to use instead a finite horizon t0=max(t-h, tmin),[[1]](#footnote-1) that is, we use in the TD(λ) formula above only the first h+1 terms (at most). If we choose then we retain only those terms with λk ≥ 0.1.

For n-tuple score: The weights wt are the entries Vi in the lookup tables. The gradient boils down to for all weights with and to 0 for all other weights. Thus we get the update rule (similar to [Jaskowsi16], Algorithm 1):

For k=t downto t0

For i=1 to m

In order to compute this at time step t, we have to store the states , k=t,…,t0 (in order to calculate the indices for all states equivalent (symmetric) to ) and the sigmoid values .

If is the state, we can start from t=0 theoretically. But if is replaced by the afterstate as done by [Jaskowski16], then we do not have the afterstate , therefore we start to learn only for t>0. This is the same reason why we have the “1” in t0=max(t-h,1).

Implementation (NTuple2ValueFunc::update): Use java.util.LinkedList for storage within horizon: we store in this list h+1 objects of class EligStates containing for each afterstate the array int[][] equiv coding and the value .

The good thing: We need no longer the eligibility traces, their storage, and the time for resetting them (after each episode) 🡪 much faster execution when LUTs are large!

Handling of random moves (RM) in elig traces:

* The current way of dealing with random moves (**standard traces [et]**) is just to skip the update step in case of RM (TDNTuple2Agt::train\*). But this means only that the state prior to RM is not added to the EligState list. The next state, if it has not RM as selected action, will be again added to the list, and the other prior states will be kept in the list until they are out-of-horizon. (Since the state prior to RM is missing, the other prior states will be even updated with a too large lamFactor.)
* A better way: **resetting traces [res]**. Whenever a random move occurs, clear the EligState list. This means fewer training updates, but possibly removes conflicting / wrong updates.

## Debugging TDNTuple2Agt

* OK: We realized that even with epsilon=0 frequent **random moves** occurred (??, which should not be!). The reason was suspicious (wrong) code around ‘progress’ and ‘randomSelect’ in TDNTuple2Agt::getNextAction (and likewise in TDNTupleAgt). – Now replaced with the simpler code

randomSelect = (rand.nextDouble() < m\_epsilon) ,   
which is used in TDAgent::getNextAction as well. With this, we observe no more random moves if m\_epsilon==0. Changed in TDNTupleAgt as well.  
The following plot shows the old and wrong random move rate (ε is constant, ‘progress’ is the proportion of all training games):  
   
Remember: When training a stored agent anew, the stored parameters for εinit and εfinal will probably not be valid or useful anymore. If we want a similar behavior, we have to select other values.   
For ε not being constant, see the [plots and text below](#_Parameter_at_time).

* With no sigmoid and ALPHA=1.0, we should get after each weight update a value being **identical to the target** used in that update. We find this valid when running the program with USESYMMETRY==false, but it is **violated** if USESYMMETRY==true. Why?? – This was a **fundam****ental problem in the formulas**: Whenever two equivalent states lead to the same Index for a certain n-tuple (e.g. 0, not activated at all), then the LUT entry for that Index will be updated twice. When later the new value (getScoreI) is calculated, the LUT for the same Index will be added twice as well. This leads to 4x delta\_w, where it should be only 2x delta\_w. – How to fix: When updating for a set of equivalent states, start with an array trainCounter[\*]=0. When a certain index is updated, increment its trainCounter. Update is only done if trainCounter[Index]==0. This leads perfectly to Vnew = Target in the case no sigmoid and ALPHA=1.0.
* The good news: With this new approach we get better results: A quick run with only 10k training games leads to avg. score 30k, highest score 60k in 50 eval games. The 2048 tile is reached in 36 out of 50 games.
* Drawback of the trainCounter remedy: Having to reset trainCounter[] for each n-tuple prior to updating slows down the whole process: for 500 training games from 2.5 sec to 51 sec (!!). We implemented a faster way with LinkedList indexList in NTuple2. This is much faster: only 7 sec instead of 51 sec. Results seem slightly worse, but this may be due to statistical fluctuations.
* It seems that the new changes and bug fixes also solve the issues 2. (**infinite weights**) and 6. (JK’s issue of **drop down in score** near the end of the training games) in the list [#Current drawbacks of TDNTupleAgt](#_Current_drawbacks_of) 🡪 at least we do not observe both effects, when we train agent ‘10 4Tupel 30k TDNT2 indexList.agt.zip’ for 40.000 games. See [training plot here](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\TDNTuple2Agt-10%204tuple%2040k.png).
* When having this agent playing, it is nice to see that (at least for some time) the **highest tile stays constantly in a corner**. This behavior was found through learning, it was not prescribed by any form of heuristics (!). When doing a quick eval, the highest-tile statistics fluctuates somewhat, but we get quick evals with 41 ‘2048’-tiles and even 2 ‘4096’-tiles out of 50 games.
* OK: Small bug: the number of training games is not saved when saving TDNTuple2Agt. – Fixed: suitable switch if(… || td instanceof TDNTuple2Agt) added in XArenaMenu::loadAgent.
* OK: Apparent bug: After the changes in TDNTupleAgt and TDNTuple2Agt, there seems to be something wrong with TTT: (a) not always 100% success (sometimes -0.2, sometimes -0.45) and (b) the first move on the empty board has all V(s)=0. Why?? There should be nothing different in TDNTupleAgt, except the new random move calculation. – This is no bug, the behavior has to do with this new random move calculation, which means that εinit=0.3, εfinal=0.0 triggers much less random moves than before. If we change to εinit=1.3, εfinal=0.0, we get very much the same results as before.
* OK: Is it still disturbing that the empty board has all V(s)=0 ? (The exact values are something like 1e-17, if we inspect them in debugger). Observation: If we switch from gb.chooseStartState01() in XArenaFuncs::train() to gb. getDefaultStartState (), we get a non-zero V(s). – It should not disturb us: an empty board with all V(st+1)=0 is not wrong (!). Because Minimax as an exact agent has exactly this V(st+1)=0. That is, we can reach a tie with EVERY initial move.
* OK: Another (upcoming) small problem with TDNTuple2Agt: in the new afterstate logic it will learn only for t>0, which is inferior for the first move in TTT. – We ‘fixed’ it with the logic, that the afterstate logic is ONLY allowed for nondeterministic games. For deterministic games like TTT or Hex it is hardcoded that AFTERSTATE==false. With this setting in effect, we learn for the first move as well.
* OK: **Is for Hex gb.chooseStartState01()** in XArenaFuncs::train() the right way to do it? – Probably yes! A good agent should not only win when it starts the Hex game, but it should also win when the opponent starts with one of the losing moves. If we start in training always from the default start state (empty board), it is very likely that self-play (after a while) will always start with one of the winning moves. Then it will not learn to exploit possible errors of the opponent in his first move. We have now the possibility in the OtherPar tab to check or uncheck the ‘Choose Start 01’ box.
* OK: Made a **tougher evaluator for Hex** (evalMode==10): (a) one game from empty board and (b) N games, when 1st player makes one of the N losing moves[[2]](#footnote-2) and the agent plays 2nd player. The evaluator calculates the average win rate for the agent. Ideally it should be 100%. – The drawback of this evaluator is: For each new Hex board size, the losing moves have to be determined manually. This makes it less easy to go to the next level.
* OK: implemented [afterstate logic](#_Afterstate_Logic_and)
* OK: member randomSelect is now part of Types.ACTIONS 🡪 simpler bookkeeping in train functions.

### Parameter settings in different software states

#### Parameter at time of writing BA Galitzki / Kutsch

ε and resulting random move rate: in TDNTupleAgt the ‘suspicious’ form

where the two numbers shown for ‘eps’ are εinit to εfinal. The strange form in the case ‘non-constant ε’ results from a strange ε-change rule in the form of a ‘half-sigmoid’ which drops quite fast from εinit/2 (which is wrong) to values close to εfinal. See [TDNTuple-eps.R](file:///C:\user\datasets\Vorlesungen\FHK\DiplomArb\2016\BA%20Kutsch\TDNTuple-eps.R) for details. In TDAgent: the random move rate is linearly decreasing from εinit to εfinal.

‘Learn from RM’: always false

‘Choose Start 01’: always true, that is 50% default start state, 50% one of the 1-ply moves start

AFTERSTATE: always false

[α-decay](#_Alpha_decay): exponentially decreasing from αinit to αfinal in TDNTupleAgt and TDAgent.

#### Parameter since 2017-08-20

ε and resulting random move rate: (at least) two options exist:

We currently implement only **epsL****inear**, since the resulting random move rates are not dramatically different for epsLinear and epsSigmoid. We do this consistently for all three agents TDAgent, TDNTupleAgt, and TDNTuple2Agt. Note that initial εinit>1 are also sensible settings for certain random move rate profiles (they are for an initial period saturated at 100% random moves). Likewise, εfinal<0 is possible as well: this results in a final period saturated at 0% random moves.

‘Learn from RM’: true/false in ‘Other pars’ tab

‘Choose Start 01’: true/false in ‘Other pars’ tab

AFTERSTATE: false for all deterministic games. true/false in ‘NT pars’ tab for nondeterministic games (currently only 2048)

[α-decay](#_Alpha_decay): exponentially decreasing from αinit to αfinal in TDNTuple[2]Agt and TDAgent.

## Hex and TDNTupleAgt Debugging

There are two problems with Hex and TDNtupleAgt

* OK: When re-loading a trained agent from Kevin Galitzki, it is **re-loaded with withSigmoid==false** (although at the time of training only withSigmoid==true was allowed). Why?? As a consequence we get unusual high values in InspectV. – This was due to a bug in the loading mechanism: Parameter tdPar.hasSigmoid() would get the right value during loading, but m\_Net.withSigmoid was not updated during load. Now fixed: suitable call to td.setTDParams() in XArenaMenu::loadAgent will set m\_Net.withSigmoid. Likewise, we need m\_Net.useSymmetry to be set properly: suitable call to td.setNTParams() in XArenaMenu::loadAgent. We had to add a new function NTupleValueFunc::setUseSymmetry. The same changes were done for NTuple2ValueFunc and TDNTuple2Agt.[[3]](#footnote-3)
* When re-training an agent for 5x5 boards or higher, the evaluation will not return (at least not for a long time). Probably it starts a Minimax evaluation which is too big to fit in memory. – Symptomatic remedy: Set Other Pars – Train Eval Mode = 0 prior to start of train.[[4]](#footnote-4) Then we do an evaluation with MCTS as counterpart, which is fast. However, it is less accurate.

## Afterstate Logic and Debugging

We add a runtime switch

boolean AFTERSTATE

in TDNTuple2Agt and decide based on this whether we expect StateObservation objects to be actually instances of StateObservation (AFTERSTATE==false) or StateObservationNondeterministic (AFTERSTATE==true).

**Important**: **Use AFTERSTATE=false for deterministic games**, only in this way we can learn for t=0 as well, and it might be important in games like Hex or other to start with the right initial move.   
**Use AFTERSTATE=true for nondeterministic games**, this should lead to a better value function, because the set of possible afterstates in a game like 2048 is much lower than the set of possible states. This allows faster learning.

The following things need to be changed if we implement AFTERSTATE==true:

### getNextAction

Assert that NewSO is an object of StateObservationNondeterministic and in the loop over actions replace

NewSO.advance(actions[i]);

with

NewSO.advanceDeterministic(actions[i]);

Then NewSO will contain the **afterstate** s’ (before adding a random tile in the case of 2048). We calculate the score r + p\*V(s’) (where p=+1/-1 is the player for 2-person games).

The rest remains the same.

### trainAgent

Assert that so is an object of StateObservationNondeterministic and in the while loop replace

so.advance(actBest);

nextBoard = m\_Net.xnf.getBoardVector(so);

with

so.advanceDeterministic(actBest);  
 nextBoard = m\_Net.xnf.getBoardVector(so);  
 so.advanceNondeterministic(actBest);

This lets nextBoard become the board vector of an **afterstate** s’ (so.advanceDeterministic) and in the next round, after

curBoard = nextBoard,

curBoard will be the board vector of an afterstate as well. On the other hand, so in line 3 is the **state** s’’ (with random tile added) which we need to continue and to decide whether the game is over or not.

Additionally, in case AFTERSTATE==true, set in the initialization part **before** the while-loop curBoard=null. Perform in trainNewTargetLogic the weight update only if curBoard!=null. This ensures the “if (t>0) …” part in [Jaskowski16], i.e. no update in the very first move of an episode.   
On the other hand, if AFTERSTATE==false, we keep the old line

**int**[] curBoard = m\_Net.xnf.getBoardVector(so);

Then the function NTuple2ValueFunction::update() is called even in the first pass, and this is in effect the same as if we replace in TD(λ)UPDATE() in [Jaskowski16] the element max(t-h,1) with max(t-h,0). This is the desired behavior for deterministic games, where we want an update for the first state (empty board) as well.

The rest of the code in trainAgent() remains the same.

As a whole, the changes which we need to implement for AFTERSTATE==true are pretty localized and simple.

However, in order to make the code cleaner, we introduce the new helper class **NextState** inside TDNTuple2Agt, which bundles the information needed for state advance. This requires more code changes, but the resulting code is better understandable and thus better maintainable. We use class NextState in trainAgent() and related functions.[[5]](#footnote-5)

### Results

Is there a measurable benefit of using afterstates in 2048? – Yes, it is! We run 10 times on a TDNTuple2Agt loaded from ‘10 4Tupel 10k TDNT2 afterState.agt.zip’, one time with AFTERSTATE, one time w/o. The results are stored in [multiTrain-2048-with[no]AFTERSTATE.csv](file:///C:\user\datasets\Vorlesungen\FHK\DiplomArb\2016\BA%20Kutsch\multiTrain-2048-withAFTERSTATE.csv), and processed with [multiTrainPlot.R](file:///C:\user\datasets\Vorlesungen\FHK\DiplomArb\2016\BA%20Kutsch\multiTrainPlot.R) to get this graph:



The AFTERSTATE==true case has results which are consistently 30%-60% better!

### Some other items

* OK: Quick Eval’s Average Score after 50 games is varying a lot in game 2048. Might be due to low outliers? We tested whether a Median Score would be more stable. But it turns out, that median has similar fluctuations. The only way seems to be that we repeat Quick Eval multiple times (e.g. 10 times) and take the average of the Average Score (resulting in a smaller standard deviation, reduced by a factor (e.g. sqrt(10))).
* OK: multiTrain: add intermediate evaluations and print results to file 🡪 visualization of multiple trainings in R: see [multiTrain-2048-with[no]AFTERSTATE.csv](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\2048\csv\multiTrain-2048-withAFTERSTATE.csv), and visualization with [multiTrainPlot.R](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\resources\multiTrainPlot.R) and similar R-files.
* OK: Extend multiTrain file saving to go in the right directory for each game. – Done: multiTrain.csv is saved to agents/<gameName>[/subDir]/csv.
* OK: Rename MaxScore in getNextAction() to BestScore (to disentangle from so.getMaxGameScore())
* OK: Extend pa.stringDescr() by the new elements: AFTERSTATE, learnFromRM. (chooseStart01 may be added when we have ParOther as part of TDNTupleAgt).
* OK: Make ε-update the same ([epsLinear](#epsLinear)) in TDAgent, TDNTupleAgt, TDNTuple2Agt.
* OK Simplify TDNTuple2Agt::trainAgent (withNS always true, comment out the two parts tagged with ‘this part will become obsolete’. But perform tests that the train behavior is exactly the same (no dangling reference to ‘old’ variables)
* OK: Parameters of tab ‘Other par’ are as well relevant for training (like ‘Choose Start 01’ or ‘Learn from RM’) and evaluation (like Quick Eval Mode). It is better if they are also part of the saved agent. – Done for TDNTuple2Agt, TDNTupleAgt, TDAgent: Added a new class ParOther. Added a member ParOther m\_oPar to all those agents. It does not invalidate the already stored agents. But some code is needed in XArenaMenu::loadAgent() in order to safely load older agents.   
  Added m\_oPar as well to MC, MCTS and MCTSExpectimax agents, it may be useful to store the quick evaluator mode used.
* OK: Simplified the interface for PlayAgent::trainAgent(so,learnFromRM,epiLength): both parameters learnFromRM and epiLength are no longer needed, they are retrieved from m\_oPar.
* Heap space: When loading the big 44MB-agent (FIXEDNTUPLEMODE=2) and then starting the training anew, we get sometimes a heap space error, even with VM-argument -Xmx1024M. Look for memory leaks!
* OK Possible extension: Currently, TD-NTuple plays always with 1-ply look-ahead. It would be an option to offer n-ply look-ahead. This requires more computational effort during game play, but probably results in stronger agent play. Needs some thinking how to establish n-ply look-ahead in nondeterministic games: Only one random playout? Multiple playouts and taking the average? Or the worst result? – Now solved with [ExpectimaxWrapper and MaxNWrapper](#_Max-N_and_Expectimax-N).
* OK Little bug: StateObserver2048 implements StateObservationNondeterministic, but StateObs2048BitShift and StateObserver2048Slow still implement StateObservation. – Done.
* OK Can we speed up TDNTuple2 training for 2048 if we create an invisible GameBoard class GBInvisible2048 which eliminates any drawing and painting on JFrame? – No, a quick implementation with GBInvisible2048 has the same runtime. We moved GBInvisble2048 to deprecated.

## Alpha decay

The learn step size α follows this decrease scheme from αi to αf:

This starts with αi at N=0 and ends with αf at N=Nmax.   
In between it follows an **exponential decay scheme**:

We see that β gets a large negative value as approaches 0. (The value is not allowed, it would lead to infinite β.) We have to keep in mind that an leads to large negative β and thus a steep and fast decline towards .

## Rewards other than game score

From discussion with Laurenz 2017/10: Currently we use the game score as reward signal. But the game score does not distinguish a good from a bad player **while the game is underway**: The score is mainly linearly rising with number of moves in an episode. A better indicator of good play might be the (cumulative) number of empty tiles.   
To test this hypothesis: Record during an episode both measures, the score and the cumulative number of empty tiles. Play one game with a good agent, another one with a bad agent. The expectation: As long as both game continue, the cumulative score plots look similar. On the other hand, the cumulative empty tiles plots should be significantly higher for the good agent than for the bad one. 🡪 partly fulfilled, see images in [resources\R\_plotTools\playStats.zip](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\resources\R_plotTools\playStats.zip).

## Games with multi-moves per turn

There are deterministic games like [Kalaha](https://de.wikipedia.org/wiki/Kalaha) where the game rules allow it that a player performs (in certain game states) more than one action (move) if it is their turn. We call a turn of a player with multiple moves a **multi-move**.

TODO: Think about Backgammon (nondeterministic, multi-move)

Multi-moves require a bit more complex action in getNextAction() and other places:

(actBest, V) = getNextAction(so, random, …)

see [TR-TDNTuple.pdf](file:///C:\user\wolfgang\www\Optimierung\TR_GBG\TR-TDNTuple.pdf)

(the **multi-move** part: If action ak calls for a next move of the same player, then search the best among these follow-up moves. Set V(ak) = V(best follow-up move). Do this recursively, if one of the follow-up moves allows another extra move!)

PlayAgent()

while (nextPlayer==curPlayer) do: another move!

trainAgent()

same as before, we have only to select always the right player via StateObservation. Each move in a multi-move gets a separate update of weights (however, the decision which next action to take during training are based on multi-move getNextAction()).

But question: What is the right next state V(st+1) for the target in an update step?

OK: make a better ACTIONS class

class ACTIONS\_VT extends ACTIONS with new members double[] vTable, double vBest

ACTIONS\_VT getNextAction2(…) is now the new alternative to ACTION getNextAction(…)

OK Changes needed for the other agents

* NEW\_GNA
* selector function getNextAction, old getNextAction becomes getNextAction1. New functions getNextAction2,3 returning an ACTIONS\_VT object
* normalize2()
* getGamma()

MCAgent: How does getNextAction1MultipleAgents() reflect the loop over multiple agents? - It returns in vtable[k] how many of the multiple agents selected k as the best next action

**TODO**:

* OK Think about the multiplication of reward with player pt during training. Shouldn’t it be the same pt when taking the difference “reward – oldReward” ?? – This was a problem of the old version (VER\_3P=false, NEW\_2P=false). It was correct, since reward was from a different player’s perspective than value function. However, it was not very ‘logical’ or easy to grasp. Now we have two alternatives (NEW\_2P=true or VER\_3P=true), where we do not need to multiply with a player-sign variable in an unsystematic way.   
  See [Konen2017b] ([TR-TDNTuple.tcp](file:///C:\user\wolfgang\www\Optimierung\TR_GBG\TR-TDNTuple.tcp)) for details.
* OK Add { trnMoves (pa.getNumTrnMoves()) } to each line of multiTrain.csv
* OK Add { alpha, epsilon } to pa.stringDescr2()) for TD-agents pa and print it as 2nd line of multiTrain.csv
* OK Transform MCTSExpectimaxAgt to use class ParMCTSE instead of MCTSExpectimaxParams (big class extending frames)
* OK When all agents have the new ACTIONS\_VT getNextAction2(…) returning an object of class ACTIONS\_VT:
* OK change all calls of ACTIONS getNextAction(…,VTable,…) to ACTIONS\_VT getNextAction2(…)
* OK handle the VTable part, where necessary
* OKIf everything works, remove the selector functions getNextAction(…) and the now obsolete ‘old’ functions getNextAction1 in every agent
* OK TD-NTuple-2: horizon printout in stringDescr()

## Debugging VER\_3P=true, MODE\_3P=0 (former OLD\_3P=true)

The VER\_3P=true, MODE\_3P=0 switch in TDNTuple2Agt is our first attempt to develop a TD-learning scheme for n-player games with arbitrary number n of players. I.e. it should allow the same algorithm for n=1,2,3,…

The technique and the equations behind VER\_3P and NEW\_2P are described in much more detail in [Konen2017b] ([TR-TDNTuple.tcp](file:///C:\user\wolfgang\www\Optimierung\TR_GBG\TR-TDNTuple.tcp)). Here we cover some software issues with VER\_3P=true, MODE\_3P=0:

1. The branch VER\_3P worked first for 2048 (1-player games), but not for 2-player games.
2. After fixing the 1st bug (see below), it works in principle, but there is a decrease in performance with MODE\_3P=0.

**ad 1)** Since VER\_3P=true, MODE\_3P=0 cannot even learn the trivial 2x2 Hex, there must be a fundamental sign bug.

Different debug printouts did not clarify the reason at first.

Suspicious items

* The DBG\_REWARD printout after the call to trainNewTargetOLD\_3P in trainAgent did show that reward and reward2 are NOT the same if nextSO is a win for the 2nd player (W). Is this correct?
* trainNewTargetOLD\_3P calls updateWeightsNew with curPlayer=nextPlayer=i (i=0,1). Is this correct?

We looked more precisely into the updates of the value function(s) in the states prior to a final state.

After 2 more hours of looking ☺, the bug was found: The problem was in g3\_evaluate with the line

agentScore = getScore(NewSO);

This was wrong for MODE\_3P=0, since getScore(so) always retrieves the score from the perspective of so.getPlayer(). This was the right thing to do as long as MODE\_3P≠0, since we need in this case according to Eq. (9) in [Konen2017b]. But it is the wrong thing to do if MODE\_3P=0. What we need then is which is given by

agentScore = getScore(NewSO, refer.getPlayer());

where refer is the state preceding NewSO.

**Now the results for VER\_3P=true or =false are much closer for TTT and Hex** (sizes 2 and 4 tested so far).

**ad 2)**

But the results are not identical.

A closer look with multiTrain reveals, that VER\_3P=true, MODE\_3P=0 (although being much better after the getScore bug fix above) is still not as good as NEW\_3P=false:



(GBG\agents\Hex\04\csv\multiTrain-learnFromRM-NEW\_3P-small.png, created with GBG\resources\R\_plotTools\multiTrainPlot-learnFromRM.R).

All curves are the average from 25 runs and the evaluator has mode 10 (success against MCTS when TD-agent starts from different winning start boards). It is clearly seen that the curves with “… 3P” (the ones with VER\_3P=true) are worse than those without (those are the ones with VER\_3P=false). The difference is more pronounced in the (learnFromRM==true)-case.

But why is there still this discrepancy?

We spent hours of looking and debugging: We built under switch DBG\_OLD\_3P inside TDNTuple2Agt a second m\_Net3 which updates according to VER\_3P=false (even if the global setting is VER\_3P=true) and compare the update steps. When do both nets start to deviate in their decisions?

It was seen that quite early the 3P-form would have Vold ≠ 0 for final states, while the 2P-form still had Vold=0.0. After more iterations, cases with Vold ≠ 0.0 would also appear in the 2P-form, but not so often and not so big. The update step in updateWeightsNewTerminal() would bring Vnew closer to 0.0, but it is of course a burden for the algorithm: As long as Vnew for final states deviates significantly from 0, the target for the preceding states (and their predecessors) will be wrong.

It is probably not a software implementation bug, but a ‘feature’ (or fundamental flaw) of the 3P-algorithm: This algorithm updates the n-tuple value functions twice as much (for 2 players), and it does so even for the value function of the player who is not the moving one. This might more often bring one of the n-tuples in a state where it gives a value different from zero to final states (unwanted cross talk). The 2P-form has fewer cross-talk[[6]](#footnote-6). For the general n-player form with arbitrary value functions we probably cannot do better than in the 3P-form. But for games with 2 players and strictly antagonistic reward functions[[7]](#footnote-7), it is probably better to stick with the 2P-form.

To put it otherwise: the 3P-form updates for twice as many states the value function away from zero. This leads to more ‘cross talk’ between n-tuples than necessary. This explains probably, why the 3P-curves in the plot above **decrease** in iterations 85.000-100.000: The random move rate goes to zero, and – since ‘Learn from RM’ is false – the update frequency increases and so the cross-talk increases. This leads to performance degradation.

Ways out:

* ~~Add a predicate StateObservation::has2OppositeRewards() to all games: This predicate is true, if it is a 2-player games with antagonistic reward function. It is false in all other cases.~~
* ~~When has2OppositeRewards()==true, run the 2P-form. In all other cases run the 3P-form.~~

[obsolete now, we use the new perspective (VER\_3P=true, OLD\_3P=false)]

* We advise with the new switch NEW\_2P=true an equivalent form to the old 2P-form (NEW\_2P=false). This new 2P-form avoids the p=+1/-1 parameter and is described in more detail in Appendix C of [Konen2017b] ([TR-TDNTuple.tcp](file:///C:\user\wolfgang\www\Optimierung\TR_GBG\TR-TDNTuple.tcp)). If the new form is tested and performs as well as the old form, eliminate the old form, i.e. have always NEW\_2P=true. Later simplify the code.
* Another perspective: Think about other alternatives to VER\_3P=true, MODE\_3P=0, which are valid for 3 and more players, but which reduce to the NEW\_2P-form (or produce results equivalent to it) in the case of 2-player games with antagonistic reward functions:
  + like in TD-Gammon: for an n-player game make an n-ply look-ahead and take this action which maximizes the Z-score Z(self, one-round-ahead)
  + for antagonistic-reward n-player games[[8]](#footnote-8): make an (n-1)-ply look-ahead (all opponents of self) and take that action which maximizes

Both alternatives need to be worked out in more detail. They are perhaps even algorithmically equivalent. In both cases there would be only **one** update of the value function (that of the current state). Both cases should reduce to the NEW\_2P-form in the case of 2-player games with antagonistic rewards.

TODO:

* OK Work out the VER\_3P alternative cases in more detail.
* We probably need to rebuild all Hex agents stored in agents/Hex/, because agents created with NEW\_2P=false will not work under NEW\_2P=true (other meaning of V()).
* OK We should later delete all Hex agents stored in agents/Hex which are from class TDNTupleAgt, they lead to confusion with TDNTuple2Agt. In the future we should use only class TDNTuple2Agt.

## Debugging VER\_3P=true, MODE\_3P=1 (former OLD\_3P=false)

We worked out the TD-Gammon alternative. Details are in [Konen2017b] ([TR-TDNTuple.tcp](file:///C:\user\wolfgang\www\Optimierung\TR_GBG\TR-TDNTuple.tcp)).

After some initial debugging: It **works nicely for nply≤NP** for all three games TTT, 2048, Hex. The case (nply=1, NP=2) is equivalent to the former (NEW\_2P==true)-form. The case (nply=2, Np=2) is for 6x6-Hex costly to calculate (10h training instead of 1h (!!)) and is only slightly better than the (nply=1)-case. This is due to the high branching factor of 6x6-Hex (18 on average).

We expected nply=2 to work faster on 2048 than on 6x6-Hex, but we found a BUG: **nply=2 leads to infinite weights**, infinite getScore (and finally a crash) on 2048. **Reason not yet found**. May be that 2048 is nondeterministic and nply-branch cannot handle this (i.e. the former problems before afterstate are back)? When [[Jaskowski16]](file:///C:\WUTemp\FH-MassenDaten\svnSoma\trunk\doc\CaseStudies.d\201314.d\CIG2014\MCTS.literature\2048\Jaskowski2016-2048.pdf) talks about “n-ply search”, he always means ‘coupling of the TD-n-tuple network (afterstate value function of the 1-ply afterstate) with n-ply expectimax search. So it is probably not worth to do n-ply in training, but it is better to couple a trained agent with an n-ply tree search (minimax in the case of deterministic games, expectimax in the case of nondeterministic games). OK, done with [MaxNWrapper and ExpectimaxWrapper](#ExepectimaxWrapper), works nicely for 2048: our best TD-NTuple2 agent with score 100.00 is boosted with 5-ply ExpectimaxWrapper to score **182.000** (!!)

OK Bug: nply=3 in 4x4-Hex leads to wrong moves covering already occupied tiles. Fixed: It was a missing “g3BestScore = -Double.MAX\_VALUE;” in g3\_Eval\_NPly. We get these results for 4x4-Hex

* nply=1 is fastest in computation time, but slower-learning in terms of training games and less stable (evalQ=1.0 may be fall back to lower values in later games).
* nply=2 or 3 is slower in computation, but learns more reliably to win after 20.000 games. Once evalQ reaches 1.0, it usually stays there. There is no perceivable difference between 2 and 3.

(These are first results only, repeated runs are necessary to confirm results)

## Summary VER\_3P

In summary, we have different alternatives for generalizing TD-NTuple2 to arbitrary N-player games:

* **MODE\_3P=0**: update N independent V(s­t|p(i)) ∀i=0,…,N-1, in each time step t. Works for arbitrary N, but weaker results for 2-player games than in 2P-logic. The reason is probably [unwanted cross-talk](#crosstalk_3P) in the n-tuples due to the more frequent updates.
* **MODE\_3P=1**: update only for V(st|pt). As a consequence, only V(s­t+k|pt+k) is known. There-fore player pt takes the action **minimizing** V(s­t+k|pt+k) if pt+k≠pt and **maximizing** V(s­t+k|pt+k) if pt+k=pt. To get a target for V(st|pt) we make an **N-ply** evaluation, so that V(s­t+N|pt+N) is a valid target for V(st|pt).
* **MODE\_3P=2**, synthesis of 0 and 1:
  + 1-player games: take mode 0 (which is for 1-player games equivalent to mode 1)
  + 2-player games: take mode 1 with **1-ply** evaluation[[9]](#footnote-9), that is make an update only for V(st|pt) and use the symmetry V(st+1|pt) = - V(st+1|pt+1) to infer the otherwise unknown V(st+1|pt).
  + N-player games with N>2: take mode=0 (and live with the cross-talk). Since we update N independent V(s­t|p(i)), we know V(s­t|p(i)) ∀i.

Drawbacks of MODE\_3P=1:

1. Taking a 2-ply evaluation in 2-player games during training is costly for games with high branching factor and does not yield better results. It is better to use 1-ply and the symmetry V(st+1|pt) = - V(st+1|pt+1).
2. Minimizing V(s­t+k|pt+k) may be too short-sighted in 3-player games. Example: If player 3 has to move and finds a state st+1 where player 1 loses, player 3 will take that state. But if st+1is a win for player 2 (and not player 3), this option is clearly disadvantageous for player 3. He should instead take an action maximizing the return for player 3
3. We do not have the score tuple V(st|p(i)) available, which we need for wrapper algorithms.
4. Running 2048 with nply=2 (instead of nply=N=1) leads to infinite weights and crash. This may be a consequence of the nondeterministic nature of 2048, i.e. it might happen as well for other nondeterministic games.

Advantages of MODE\_3P=2:

1. Best results for 1- and 2-player games.
2. Avoids the costs of n-ply-training with n>1.
3. For **all** N-player games, the score tuple V(st|p(i)) is available.

## Max-N and Expectimax-N

We added two new agents MaxNAgent and ExpectimaxNAgent which are the generalization of Minimax to N players for deterministic and nondeterministic games.

We added two wrapper agents MaxNWrapper and ExpectimaxWrapper. Each wrapper has a parameter “Wrapper nPly” which means that the tree is recursively build from the current state up to depth nPly. When the leaves are reached, the wrapped agent is used to determine the game value. When using ExpectimaxWrapper for 2048, we have a high branching factor (2\*numEmptyTiles). Therefore, the wrapper gets costly for larger nPly. For QuickEval & nPly>0 we implemented a **parallel version on 6 cores** for faster execution, yielding 10 min for nPly=5 instead of 1h single-threaded. The moves/second measurement should still be realistic.

QuickEval results for a wrapped TD-NTuple-2 agent with FIXEDNTUPLEMODE=2 [Jaskowski2016, 4 6-tuple] and 200k training games and 50 QuickEval games:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| λ=0.0 | 200k training games | | | 300k training games | | 400k training games | |
| nPly | avg score | highest tile | moves/second | avg score | highest tile | avg score | highest tile |
| 0 | 108.000 | 8192: 21/50 | 94.620 | 116.000 | 8192: 21/50 | 113.000 | 8192: 31/50 |
| 1 | 108.000 | 8192: 25/50 | 56.800 | 109.000 | 8192: 28/50 | 120.000 | 16384: 1/50 |
| 3 | 150.000 | 16384: 1-2/50 | 20.048 | 150.000 | 16384: 1-2/50 | 170.000 | 16384: 4-6/50 |
| 5 | 182.000 | 16384: 9/50 | 582 | 188.000 | 16384: 9/50 | 196.000 | 16384: 14/50 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| λ=0.5 | 200k training games | | | 300k training games | | 400k training games | |
| nPly | avg score | highest tile | moves/second | avg score | highest tile | avg score | highest tile |
| 0 | 115.000 | 8192: 25/50 | 120.000 | 115.000 | 8192: 25/50 | 118.000 | 8192: 31/50 |
| 1 | 110.000 | 8192: 27/50 | 56.000 | 115.000 | 8192: 30/50 | 115.000 | 16384: 1/50 |
| 3 | 154.000 | 16384: 2-4/50 | 20.000 | 157.000 | 16384: 1-4/50 | 165.000 | 16384: 5-7/50 |
| 5 | 185.000 | 16384: 11/50 | 580 | 181.000 | 16384: 6-14/50 | 202.000 ± 8.500 | 16384:16-21/50 |

That is, nPly=5 leads to nearly a **doubling (!) of the avg. score** as compared to nPly=0.

Surprising results:

* nPly=1 is not better than nPly=0. In fact, it seems sometimes slightly worse. Why?
* 100k training games have only a bit weaker results score = 78.000/128.000/159.000 for nPly=0/3/5. We reach the 16384-tile only in 4/50 trials.
* 300k or 400k training games is not (or only weakly) better than 200k training games. This is in some contrast to [Jaskowski16]. Do we need λ>0? Actually, λ=0.5 gives only slightly better results. And for both λ the results for 400k and nPly=5 are significantly better than those for 200k. The parameter settings are (others as in [standard parameter settings](#StandardParams_2048_TDNT2)):

|  |  |  |
| --- | --- | --- |
|  | V2 | V5 |
| filename | fixed 4 6-Tupels 200k\_V2 TDNT2.agt.zip | fixed 4 6-Tupels 200k\_V5 TDNT2.agt.zip |
| λ | 0.2 | 0.5 |

* Training times for V5 (λ=0.5) are 3.0h/5.0h/6.5h for 200k/300k/400k training games. V2 only slightly below. All training is done on a single core.

Overall: needs to be repeated to get a more reliable std. dev. and average.

## TC on 2048

First setting: TC-Immediate, α=1.0, λ=0, NO output sigmoid, USESYMMETRY, AFTERSTATE, FIXEDNTUPLEMODE=2.

Very large program during training (**4.0 GB!!**), needs Run Configuration -Xmx4096M 🡪 check whether we can reduce the memory consumption (dWArray, tcFactorArray), if we allow only TC-immediate.

Results are much better than with TD:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| λ=0.0 | **TC, 100k** training games | | | **TC, 200k** training games | | ~~400k training games~~ | |
| nPly | avg score | highest tile | moves/second | avg score | highest tile | ~~avg score~~ | ~~highest tile~~ |
| 0 | 122.000 ± 5.900 | 8192: 32/50 | 115.620 | 135.000 ± 8.900 | 16384: 1-5/50 | ~~113.000~~ | ~~8192: 31/50~~ |
| 1 | 125.000 | 8192: 32/50 | 56.800 | 125.000 | 16384: 2-6/50 | ~~120.000~~ | ~~16384: 1/50~~ |
| 3 | 169.000 ± 5.600 | 16384: 4/50 | 18.048 | 200.000 ± 8.600 | 16384: 12-14/50 | ~~170.000~~ | ~~16384: 4-6/50~~ |
| 5 | 191.000 ± 6.600 | 16384: 9/50 | 582 | 200.000 ± 8.400 | 16384: 12-13/50 | ~~196.000~~ | ~~16384: 14/50~~ |

The results are especially good for the early iterations (up to 50.000 – there we reach already a score of approx. 100.000), after that there is only weak improvement when we go from e.g. 100.000 to 200.000 training games.

TODO: might be that TC becomes in its current form less useful, if it runs for many training games. Might be that the TC-factors approach all zero when we run long enough 🡪 check the tcFactor curve for weights which are updated often. A cure can be that we introduce a forgetting term into TC-factor (forget towards TC\_INIT)

Surprisingly: When going from nPly=3 to nPly=5 in the case **TC, 200k**, there is NO further improvement (!) Why?

TODO: Check the timing, currently the program reports 500.000 moves/sec for nPly=1 (parallel eval) (?)

Training time is 1.5h for 100k

## TC on ConnectFour

A careful checking of TCL in GBG in comparison with TCL in paper [[Bagh14]](#Bagh14) revealed several points which were wrong in TCL-GBG and which are now fixed:

* fetching tcFactor did also update N and A 🡪 too early according to [[Bagh14]](#Bagh14). Now we first fetch tcFactor for LUT-update, then we accumulate N and A.
* Accumulation was done with error signal δ 🡪 now we have with switch tcAccRW in NTuple2.java the possibility to accumulate the recommended weight change RW = δ ei (the recommended choice), or to accumulate δ.
* the **TCL-EXP** scheme was missing: that is to use instead of tcFactor = N/A the exponential transfer function tcFactor = g(N/A) with g(x) = exp(β(x-1)), see [[Bagh14]](#Bagh14) and switch **tcEXP** and parameter **tcBeta** in NTuple2.java.
* OK: add these switches to NTParams and ParNT, remove TC factor type (we now use always “Immediate” in accordance with [Bagh14]) and remove TC interval (irrelevant if we use always “Immediate”)
* TODO: add optional decay parameter μ≤1 for N and A. A little problem: when to decay? Since weight activation is very sparse, a decay of every weight in every step is very time consuming and probably makes TCL ineffective. A better solution is to add the decay factor only to those steps where an accumulator is updated anyhow.
* OK: add parameter horizonCut to TDParams and ParTD
* modestly rising computation time: horizonCut=0.1: 2100 sec, horizonCut=0.01: 2900 sec (λ=0.5, horizon=7). For λ=0.6 and C=0.01 the horizon is 10 🡪 3500 sec
* The horizon varies as a function of λ and horizon cut C as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | λ | | | |
| C | 0.4 | 0.5 | 0.6 | 0.7 |
| 0.001 | 8 | 10 | 14 | 20 |
| 0.01 | 6 | 7 | 10 | 13 |
| 0.1 | 3 | 4 | 5 | 7 |

* OK: think about different eligibility trace variants as described in [[Thill14]](#Thill14): standard [et], resetting [res], replacing [rep], reset&replace [rr]. [[Thill14]](#Thill14) seems to indicate that they all are very similar for ConnectFour. GBG currently implements only standard [et] 🡪 now we have the options [et] and [reset].
* OK: track the results of the training games: is it after a while nearly always TIE? If so, does the agent stop learning then? – No, it is not always TIE, only about 10% TIEs, even after longer training time.

## Connect Four Eval: AlphaBeta and MCTS

First results:

Evaluation: evalMode=0: computeAgainstMCTS, numEpisodes=8, MCTS-iterations=1000, competeBoth. The opponent MCTS(1000) is a reasonable-playing, but not perfect-playing agent. If the trained agent were perfect, the competeBoth-result could reach 1.0 (winning all episodes against MCTS, whether as 1st or as 2nd player). If the opponent were perfect, the best competeBoth result would be 0.0 (agent can only win the episodes where he is 1st player).

Large fluctuations with only 8 episodes in evalMode=0 🡪 we need to average over 5-10 trainings and over 5 gameNum points to get stable results.

After some first results (OLD, see [appendix](#results_old_buggy)), we found a **bug** in XNTupleFuncs.**getBoardVector**: instead of setting bvec[n] to “3” (reachable-empty), we set wrongly board[i][j] to 3. As a consequence, state=3-weights were never activated in the old code (only 3% active weights). Now fixed, we see 7% active weights, the .agt.zip files are twice as big and we get better results (on first view) 🡪 we need to repeat all experiments from [appendix](#results_old_buggy)!!!

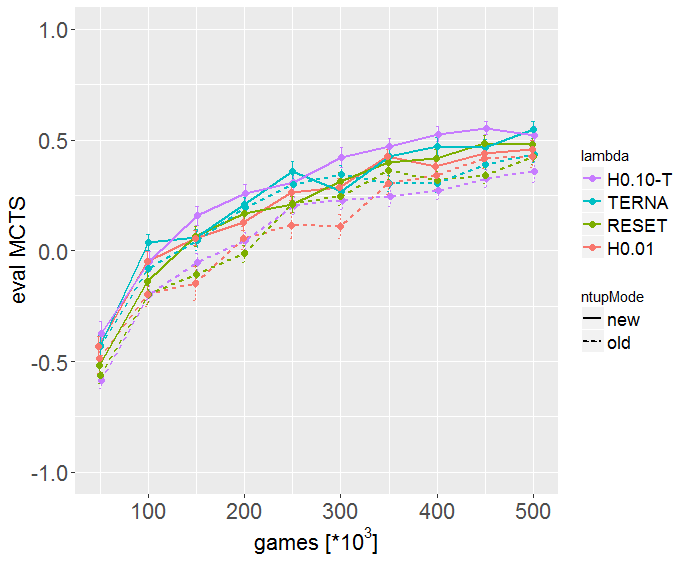
NEW, bug-fixed version: The overall settings (if not stated otherwise) are: USESYMMETRY=true, ChooseStart01=false, LearnFromRM=false, Reward=Score, Gamma=1.0, NORMALIZE=false, OutputSigmoid=true, MODE\_3P=2, fixed-n-tuple mode 1: 70 8-tuples, TC\_INIT=1e-4, TC-EXP with tcBeta=2.7, rec.weight-change accumulation. 500.000 training games, 10 runs.

1. TDNTal05-lam06-500k.agt.zip: TDNTuple2Agt **without TCL** produces only mediocre results: 500.000 training games for example has a Quick Evaluation result near 0.0.
2. [**TCL-EXP**-al20-lam05-500k-HOR001.agt.zip](file:///C:\Users\wolfgang\Documents\GitHub\GBG\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR001.agt.zip), with settings α=2.0, ε=0.1, λ=0.5, horizonCut=0.01, has **much better** results near **0.46** (the old version had 0.43). High ε, only 5% of the games are TIE.
3. [TCL-EXP-al20-lam05-500k-HOR001-TERNARY.agt.zip](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR001-RESET.agt.zip): same as 2., but [ternary](#TERNARY) update rule: evalQ = **0.55** (!!, the old version had 0.44).
4. [TCL-EXP-al20-lam05-500k-HOR001-RESET.agt.zip](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR001-RESET.agt.zip): same as 2., but elig-RESET: evalQ = **0.48** (the old version had 0.43).

We see that the [TERNARY](#TERNARY) update rule is much more important and beneficial in case of the new version (which has the [bug fix in getBoardVector](#bugGetBoardVector)). We activate thus TERNARY=true for the following experiments:

1. [TCL-EXP-al20-lam05-500k-HOR010-T.agt.zip](file:///C:\Users\wolfgang\Documents\GitHub\GBG\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR010-T.agt.zip): same as 2., but horizonCut=0.10: evalQ = **0.52**. (! the old version had 0.36, but without [TERNARY](#TERNARY))
2. [TCL-EXP-**al50**-lam05-500k-HOR001-TERNA.agt.zip](file:///C:\Users\wolfgang\Documents\GitHub\GBG\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR010.agt.zip): same as 2., but α=5.0: evalQ = **0.55**. (! the old version had 0.39, but without [TERNARY](#TERNARY))

The following plot confirms that the new version is nearly always better than the old one:



We have still the discrepancy that – although MCTS is an agent far from playing perfect – the evalQ-values with TCL are <0.55 and thus way below the eval-score of 0.95 that [Thill14] reports when testing against the perfect playing AlphaBetaAgent AB. It might however be that MCTS tests different qualities than AlphaBetaAgent (MCTS has more randomness 🡪 more diverse parts of the game tree search, while AB might follow more closely the same paths in the game tree and theses path are also those learned predominantly by TCL agent).

To test this, we ported MT’s AlphaBetaAgent to GBG. After some debugging (it is important not to use getScore, but getNextVTable(int[][] board, useSigmoid=true)), we got it working: the plausibility checks “always -1.0 eval for RandomAgent against AlphaBetaAgent” and “always -1.0 for every agent playing 2nd against AlphaBetaAgent playing 1st” are fulfilled.

Now we make all runs with **two evaluators**: quick eval = 0 (competBoth against MCTS) and train eval = 3 (single compete against AB). We see that eval\_AB reaches 0.95 and higher, while eval\_MCTS is not higher than 0.6: Thus we conclude that it is possible that a trained TCL can lose about 40% of the ‘both’ games against MCTS (and usually also around 30% of the ‘single’ games – those games that it theoretically can win since it plays 1st) AND at the same time can lose only 5% of the ‘single’ games it can win when playing against the perfect playing AB agent.

[We note in passing that the competeBoth eval score Sboth of AB would be – given that the single score is Ssingle=0.95 –

The competeBoth AB eval score Sboth ∈ [-1,0] is thus always ≤ 0. But for better comparison with [[Thill14]](#Thill14) we report here the single AB eval score Ssingle ∈ [-1,+1].]

The results from comparing both evaluators, MCTS and AB as well as [TERNARY vs. TD](#TERNARY) target:

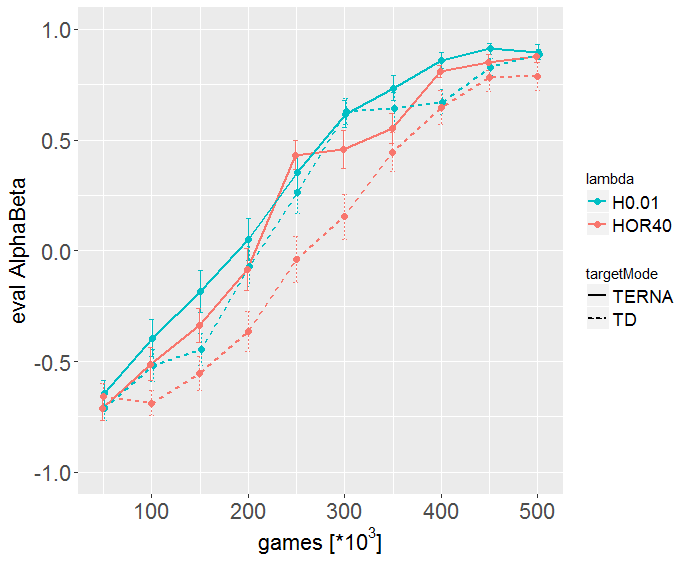
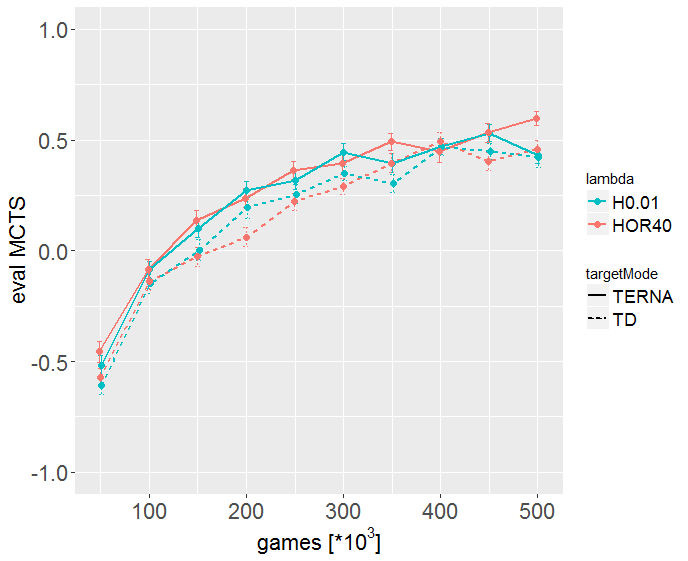


Figure 1: [multiTrainPlot-C4-TERNA.R]

Observations:

* We see that the AB evaluator generates smoother curves than MCTS.
* Most importantly, we can confirm that TDNTuple against AB eval reaches values around 0.9 (in the games that TDNT can win), while the same TDNT agents have only 0.5 competeBoth eval score against MCTS.
* [TERNARY](#TERNARY) is slightly better than TD target, but not much (at least not much for the better horizon H0.01)
* Surprisingly, H0.01 (the shorter horizon 7) is better than the longer horizon 40. It might be that a longer horizon 40 is of no advantage if we have 10% random moves where the long-ago eligibility traces are no longer useful to base the weight update on.

To make the comparison between AB and MCTS evaluator fully fair, we set the MCTS evaluator to **single compete**, that is TDNTuple plays always 1st, MCTS always 2nd, like it is for AB evaluator. Then we get the following curves:

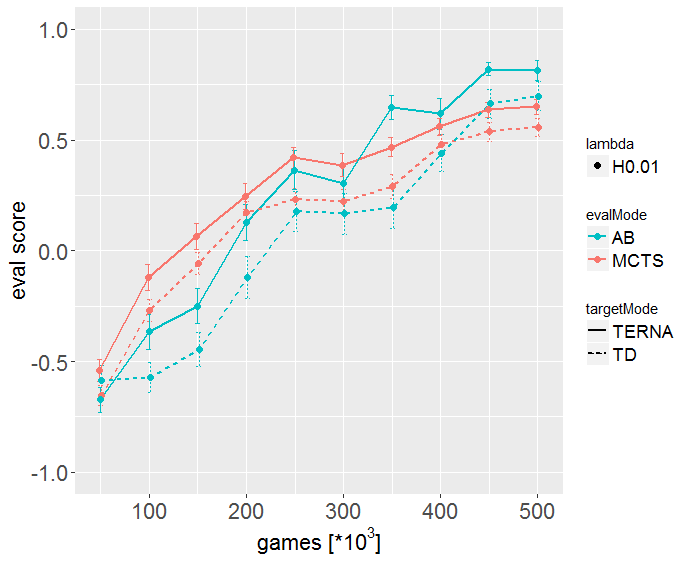


Figure 2: Single-Compete evaluators [multiTrainPlot-C4-single.R]

Observations:

* The score against MCTS is now a bit higher (around 0.6) than with compete both. This is understandable, TDNTuple has an advantage when playing 1st.
* TDNTuple gets in the end higher scores against AB than against MCTS. The score is 0.81 and 0.70 for AB vs. 0.65 and 0.55 for MCTS (TERNARY and TD target)
* The true gap when both evaluators run “single compete” is 0.81-0.65 = 0.16 (TERNARY) and 0.70-0.55 = 0.15 (TD), a bit smaller than in Figure 1.
* Nevertheless, the difference is remarkable and statistically significant.
* **Interpretation**: TDNTuple plays stronger against AB, because AB plays in a similar way as TDNTuple plays against itself. TDNTuple plays weaker against MCTS, because MCTS makes often weaker and ‘surprising’ moves, which lead however to another part of the game tree where TDNTuple has less / no learning experience and makes non-optimal moves which MCTS can then exploit.
* **Possible inference**: Would we get a stronger TDNTuple agent if we train it with some MCTS-opponent experience? We could for example replace the (totally) random moves by MCTS random moves, which induce variation, but in a more guided sense. – The downside: TDNTuple would no longer train solely by self-play but would rely to some extent on the ‘wisdom’ of the MCTS agent.
* The results for AB in Figure 2 (0.81 and 0.70) are considerably lower than they were above in Figure 1 (0.90 and 0.89), unclear why. This may be statistical fluctuations, but it is quite unlikely to have them that large.
* On the other hand, TERNARY is now in both cases significantly better than TD.

## Useful Parameter Settings

### 2048

TD-NTuple-2: ε =0, α =0.2🡪0.1, λ =0, NO output sigmoid, USESYMMETRY, AFTERSTATE, FIXEDNTUPLEMODE=1 [Jaskowski2016, Fig. 3b, 5 4-tuple], 10.000 games 🡪 Quick Eval Score approx. 34.000 (see figure in [Sec. Results](#ResultsAfterstate))

**Stand****ard settings** TD-NTuple-2: ε=0, α=0.2🡪0.1, λ=0, NO output sigmoid, USESYMMETRY, AFTERSTATE, FIXEDNTUPLEMODE=2 [Jaskowski2016, Fig. 3c, 4 6-tuple], 40.000 games (0.5h) 🡪 Quick Eval Score approx. 50.000 (!), but mighty LUT: Even as ZIP, the agt.zip is 44 MB (!, now deleted).

Training the same agent for 100.000 games (0.9h) 🡪 Quick Eval Score approx. **80.000** (!! **best so far**), but even larger ZIP: 69 MB (!). Highest tiles reached in 200 eval games are: 29% 8192, **75% 4096**, 89% 2048 (!). See [agents\2048\fixed 4 6-Tupels 100k TDNT2 afterState.xlsx](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\2048\fixed%204%206-Tupels%20100k%20TDNT2%20afterState.xlsx).

Training the same agent for 200.000 games (3.0h) 🡪 Quick Eval Score approx. **108.000** (!!), but even larger ZIP: 92 MB (!). Highest tiles reached in 200 eval games are: **48% 8192**, 46% 4096, 5% 2048 (!). See [agents\2048\fixed 4 6-Tupels 200k TDNT2 afterState.agt.zip](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\2048\fixed%204%206-Tupels%20200k%20TDNT2%20afterState.agt.zip) and [resources\R\_plotTools\multi-100k-200k-TDNT2-afterstate.png](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\resources\R_plotTools\multi-100k-200k-TDNT2-afterstate.png). And there seems still to be potential in the learning curve.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| training games | learning actions | our avg score | our highest tile | [Jaskowski16] score for TD(0.5), 1ply | [SzubertJaskowski14] score for TD(0.0), 1ply |
| 50.000 | 1.0e+8 | 68.000 |  | ≈50.000 | ≈40.000 |
| 100.000 | 2.6e+8 | 82.000 | 8192: 14/50 | ≈60.000 | ≈55.000 |
| 200.000 | 6.6e+8 | 108.000 | 8192: 24/50 | ≈80.000 | ≈62.000 |
| 300.000 | 11.23e8 | 116.000 | 8192: 22/50 | 100.000 | ≈64.000 |
| 400.000 | 16.02e8 | 113.000 | 8192: 31/50 |  |  |

The [Jaskowski16] results are read off from Fig. 6, at the very beginning of the curve (Jaskowski trains his nets for 1e10 learning actions, i.e. 100 times longer than our lowest result). [Jaskowski16] seems to use his Fig. 3c n-tuple set (33-42), which is in the text said to have four 6-tuples, but Fig. 3c shows **five** 6-tuples.

The [SzubertJaskowski14] results are read off from Fig. 11, they are for a slightly simpler n-tuple set {2 4-tuple, 2 6-tuple}.

We use the FIXEDNTUPLEMODE=2 n-tuple set with **four** (2 rectangular, 2 d-shaped) 6-tuples. Note that [Jaskowski2016, Fig. 3c] shows **five** 6-tuples (one more d-shaped 6-tuple), although [Jaskowski, Page 3 of 12] speaks about **four** 6-tuples.

These are our 1ply-results. [With ExpectimaxWrapper we get even higher results](#_Max-N_and_Expectimax-N). [With TC as well](#_TC_on_2048).

**TC settings**: TC-Immediate, TC-Init=10-4, α=1.0, ε=0, λ=0, NO output sigmoid, USESYMMETRY, AFTERSTATE, FIXEDNTUPLEMODE=2, 100.000 training games (2.0h).

Very large program during training (**4.0 GB!!**), needs Run Configuration -Xmx4096M. Large ZIP for saved agent [agents\2048\TC fixed 4 6-Tupels 100k\_V1.agt.zip](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\2048\TC%20fixed%204%206-Tupels%20100k_V1.agt.zip): 90.9 MB.   
Very good results: avg. score **121.000 ± 7.060**. Highest tile 8192 in 31/50 cases (**62%**).

**CAUTION**: Remember that ConfigGame.FIXEDNTUPLEMODE has to have the right value **in source code** when you want to retrain with similar results. Re-loading + re-train with wrong FIXEDNTUPLEMODE does NOT work. (Re-loading the agent and doing Play Game, Quick Eval or similar will however work).[[10]](#footnote-10)

(‘Choose Start 01’ and ‘Learn from RM’ are irrelevant here since we have no agent random moves.)

Note that [Jaskowski16] reports a 40% increase in score when changing α from 0.1 to 1.0.

Another thing to investigate: Is it true that our agent learns much faster than in [Jaskowski16] or [[SzubertJaskowski-CIG2014]](file:///C:\WUTemp\FH-MassenDaten\svnSoma\trunk\doc\CaseStudies.d\201314.d\CIG2014\MCTS.literature\2048\paper2048-SzubertJaskowski-CIG2014.pdf)? If so, why? (Most elements in TD-NTuple-2 are exactly as in [Jaskowski16]. But one thing different: we have perhaps the better update rule, where no index is updated twice in one round.)

### TicTacToe

TD-NTuple-2: ε=1.0🡪0, α=0.2, λ=0, output sigmoid, MODE\_3P=2, USESYMMETRY, random n-tuple 10\*6, 10.000 games, Quick Eval Mode 2 and Train Eval Mode 9 🡪 evalQ=0.0 after 2.000-3.000 games. [agents\TicTacToe\tdntuple2-oPar.agt.zip](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\TicTacToe\tdntuple2-oPar.agt.zip)

### 4x4 Hex

TD-NTuple-2: ε=1.0🡪0, α=0.5, λ=0, output sigmoid, USESYMMETRY, random n-tuple 20\*5, 100.000 games, Quick Eval Mode 0 (not 2) and Train Eval Mode 10 🡪 good Inspect initial board, evalQ=1.0, evalT=0.92. [agents\Hex\04\TDNTuple2\_3P-MODE2.agt.zip](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\Hex\04\TDNTuple2_3P-MODE2.agt.zip)

TD-NTuple-2: ε=1.0🡪0, α=0.99, λ=0, output sigmoid, USESYMMETRY, random n-tuple 20\*5, 100.000 games, Quick Eval Mode 0 (not 2) and Train Eval Mode 10 🡪 good Inspect initial board, evalQ=1.0, evalT=1.0.

Remember that we need in ‘Other pars’ the checkboxes ‘Choose Start 01’ checked and ‘Learn from RM’ NOT checked in order to get good results.

The ideal InspectV initial board has +1000 on the vertical diagonal and -1000 everywhere else. A good approximation to that is routinely found by the agents.

### 5x5 Hex

Quick Eval Mode 0 (not 2) and Train Eval Mode 10

TD-NTuple-2: ε=1.0🡪0, α=0.5, λ=0, output sigmoid, USESYMMETRY, random n-tuple 25\*6, 150.000 games 🡪 good Inspect initial board, evalQ=1.0, evalT=0.9.

TD-NTuple-2: ε=1.0🡪0, α=0.2, λ=0, output sigmoid, USESYMMETRY, random n-tuple 25\*6, 150.000 games, 🡪 good Inspect initial board, evalQ=1.0, evalT=0.9.

The InspectV initial board for α=0.2 (right) shows roughly the win-lose-pattern found by Hexy (left):

W

L W

L W L

L W W W

L W W W L

W W W L

L W L

W L

W

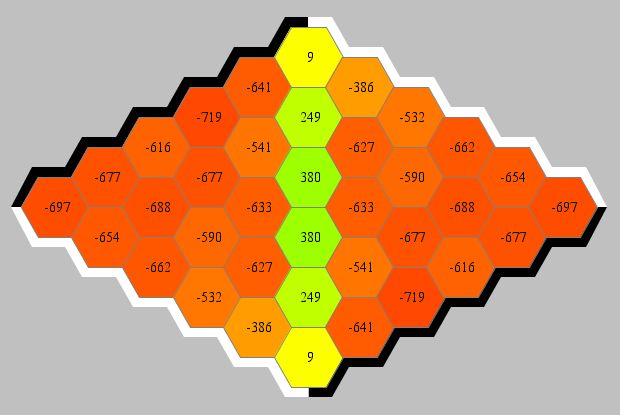
The diagram means: Black (1st player) loses (L) or wins (W) when he plays this first move.   
[It depends on the random factors of a run (which n-tuples and so forth), the InspectV initial board does not always mirror so nicely the true win-lose-pattern.]

Remember that we need in ‘Other pars’ the checkboxes ‘Choose Start 01’ checked and ‘Learn from RM’ NOT checked in order to get good results.

### 6x6 Hex

Quick Eval Mode 0 (not 2) and Train Eval Mode 10

TD-NTuple-2: ε=1.0🡪0, α=0.2, λ=0.5, output sigmoid, USESYMMETRY, random n-tuple 25\*6, 150.000 games, 🡪 good Inspect initial board, evalQ=1.0, evalT=0.9.

The InspectV initial board for α=0.2 (right) shows roughly the win-lose-pattern found by Hexy (left):

W

L W

L W L

L W W L

L W W W W

L W L L W L

W W W W L

L W W L

L W L

W L

W

The diagram means: Black (1st player) loses (L) or wins (W) when he plays this first move.   
[It depends on the random factors of a run (which n-tuples and so forth), the InspectV initial board does not always mirror so nicely the true win-lose-pattern.]

Remember that we need in ‘Other pars’ the checkboxes ‘Choose Start 01’ checked and ‘Learn from RM’ NOT checked in order to get good results.

TD-NTuple-2: ε=1.0🡪0.2, α=0.2, λ=0.5, output sigmoid, USESYMMETRY, random n-tuple **250\*4**, 150.000 games, 🡪 good Inspect initial board, evalQ=1.0, evalT=0.9.

### Connect Four

Standard game board with 7 columns, 6 rows. [It currently not possible to specify other game board size through C4Base.COLCOUNT and C4Base.ROWCOUNT, since several functions (those using bitboards) are specific to 7\*6.]

# Appendix

## Markus Thills’s CFour Software

The software CFour ([C:\Users\wolfgang\Documents\GitHub\Connect-Four\CFour](file:///C:\Users\wolfgang\Documents\GitHub\Connect-Four\CFour)) from M. Thill has still better results with TCL and elig than our CFour in GBG. Why?

In order to understand his algorithm precisely, we analyze the software

### Classes

class TDSAgent extends ConnectFour implements Agent, Progress, has elements  
 ValueFuncC4 m\_Net;

class ValueFuncC4 has elements  
 NTupleC4[][] nTuples;  
 WeightSubSet[][] weights;  
 LRCommon lrCommon;

class NTupleC4 extends WeightSubSet.

abstract class WeightSubSet has elements   
 float[] lut;  
 LRCommon lrCommon;

LearningRates lr;

Objects weights in class ValueFuncC4 are constructed in ValueFuncC4.initNTuples(). They can include bias weights as a special WeightSubSet, but usually they have not.

Interface LRCommon is a public inner interface of LearningRates.

Objects of abstract class LearningRates (package adaptableLearningRates) are constructed by the factory method WeightSubSet.createLearningRates(int lutLength). This returns in case of TCL a new object of class TemporalCoherence.

Class TemporalCoherence extends LearningRates.

### Rules and FAQs

UpdateParams u\_i.globalAlpha is member alpha from ValueFuncC4 (the global learning rate α).

TemporalCoherence.updateTables(u\_i) gets parameter mu = μinit from the Params tab, and this is usually 1.0. The recommended weight change is in case useRWC==true: ri = α⋅δ⋅ei. This is the same as in our GBG-software where we have ri = αm⋅δ⋅ei and the [scaled-down αm](#scaledAlpha) with (m= # n-tuples, Ns=# equiv states) corresponds to Thill’s global α=0.05. – A typical setting is mNs = 70⋅2 = 140, so we should use the GBG global α = 0.05⋅140 = 7.0 to have the same learning rate as Thill’s α=0.05. (Usually we have α = 2.0 or α = 5.0).

The global parameter “Use bias weights” is usually turned **off** in Thill’s software.

The function ValueFuncC4.**update(…)** is only called in case λ=0.0. If λ>0.0 then **updateWithEligTraces(u)** is used.

How do the mirror-symmetric states (equiv) come into play in updateWithEligTraces? – m\_Net.updateElig() is responsible for this: It constructs with   
 int i[] = getIndexSet(zobr, board, equiv);  
the index set for board and its mirror board equiv. Later, addGradToElig() is called for every i[k]

Is the difference “random move added to elig traces”, flagged with CAUTION below, important?

### Algorithms

How are the algorithms implemented in case TCL (with useRWC==true) and λ>0.0?

TDSAgent.trainNet() // train for 1 episode

getBestMove()

if (!randomMove || finished)

m\_Net.updateWeights()

if (!randomMove || !resetEligOnRandomMove)

m\_Net.updateElig()

else // i.e. if (randomMove && resetEligOnRandomMove)

m\_Net.resetElig()

CAUTION: The above logic means: If it is a random move AND we have standard [et] elig traces such that !resetEligOnRandomMove==true, then m\_Net.updateElig() will be called. That is, the random move is added to the eligibility traces. This is different to GBG.

ValueFuncC4.updateElig()  
 calculate gradient  
 calculate index vector for board n-tuples and mirror-board n-tuples  
 scale eligibility traces with γλ  
 add gradient to eligibility traces

The code around the calls addGradToElig (with eAdd and ‘sets all elements for the other player to zero’) is somewhat unclear 🡪 Ask MT

ValueFuncC4.updateWeights()  
 // normally tdPar.tclUseUpdateEpisodes==false  
 y = getValue(curBoard…); // the net’s value prediction  
 tg = (finished) ? reward : γ⋅getValue(nextBoard…);  
 u = new UpdateParams(…,alpha,delta,derivY,y);  
 updateWithEligTraces(u);  
 weights[0..1][0..numTuples-1].updateLUTwithElig(u);   
 WeightSubSet.update(u\_i) {  
 lut[u\_i.i] += dW;  
 lr.postWeightUpdateTask(u\_i);  
 }

postWeightUpdateTask calls updateTables(u\_i), and this updates the counters tcN[i] and tcA[i] according to RWC ri = α⋅δ⋅ei.

The activation function is y(x) = tanh(x) ∈ [-1,1]. Its derivative is y’(x)=1-y2(x). The activation function is realized with the call y = getValue(…) which in turn calls MYTANH, a faster tabularized version of Math.tanh. The steepness of tanh() is set to tdPar.sigOutputFac=1.0. The reward needs not to be scaled in the case of tanh, because it has the range [-1,1] as well.

GBG has also tanh() as sigmoid, if activated. CFour: tanh or MYTANH, is it different?

A **noticeable difference** is the line  
 tg = (finished) ? reward : γ⋅getValue(nextBoard…);  
which was our old update rule (we call it TERNARY), but now we have in GBG  
 tg = reward + γ⋅getValue(nextBoard…);  
with the extension that terminal states are trained to a value (expected future reward) of 0.0. Does this make any difference? 🡪 see debug switch **TER****NARY** in TDNTuple2Agt.

### AlphaBetaAgent

This agent has the perfect Minimax agent with opening book (Book.java, BookSum.java).

C4Game.initGame() (package gui) generates a standard AlphaBetaAgent alphaBetaStd and fetches for it the standard settings from OptionsMinimax winOptionsGTV. (The default settings are: useNormalBook=true, useDeepBookDist=true, maxSearchDepth=42, randomizeOnEqualMoves.) With this alphaBetaStd an object Evaluate eval = new Evaluate(alphaBetaStd) is constructed.

Now, whenever menu item “Quick Evaluation” is called, the following chain is executed:

C4Game.evaluate() 🡪 eval.getScoreStr(td,ab=alphaBetaStd,…) 🡪 eval.getScore(td,ab,…)

eval.getScore(…) constructs two objects Competition compX, compO with AlphaBetaAgent referee = alphaBetaStd and calls for both, compX and compO,

* if calc012==true: the member function Competition.compete(…) from different start boards.
* if calc50Games==true: 50x member function Competition.compete(…) from empty board. Agent referee will play randomly whenever it has the choice of two (or more) equal moves.

Each compete() call triggers the chain:

Competition.compete(…) 🡪 referee.getScore(board) (to get the expected winner) 🡪 then playing the game, where referee.getNextVTable() is called repeatedly.

Both, getScore() or getNextVTable(), look for the move with the best next value either in the opening books or they call AlphaBetaAgent() rootNode(true), which determines the value recursively.

**Value bar printing**: C4Game.printValueBar() 🡪 if (showGTV) realVals=getGTV() 🡪 AlphaBetaAgent GTVab.getNextVTable(c4.getBoard(),useSigmoid=true) … 🡪 valueBoard[0][i].setText((int)realVals[1]\*100.

The large score value (992, 1001, …) inside getNextVTable() are mapped with tanh to {-1,0,1} and appear in the value bar as {-100,0,100}.

### Questions to MT

Do we need all three books book.dat, bookDeep.dat, **bookDeepDist.dat**? 🡪 only **bookDeepDist.dat** really needed.

What is the more important eval-test: calc012 or **calc50Games**? – 50 games from empty board

[The code around the calls addGradToElig](#addGradToElig)

Why are there score values such large as -992, +992, +1001 in AlphaBetaAgent? – I assume, MT disentangles with this near and far losses. Since the numbers are big, a tanh() will map all of them to {-1,0,1}

## GBG Software: C4 Case

We write down the interplay and dependence of the GBG functions in the case of Connect-Four training, TCL algorithm (with useRWC==true) and λ>0.0, MODE\_3P==2:

### Rules and FAQs

We have UPDATE==**SINGLE\_UPDATE** and VER\_3P=true.

Since MODE\_3P==2, we have **nply=1** in getNextAction3

GBG: Does Z\_nply = actBest.getVBest() which is used as target, include the reward (for game over)? (the code in GBG is much too complicated around target, reward and value!?!) – Yes, it does: actBest.getVBest() returns the best value from g3\_Eval\_Nply, which is the score s­I+1 = ri+1 +γ Vi+1.

EligStates after game over? – Yes, updateWeightsNewTerminal() calls update(), and this constructs the EligStates object for the terminal state.

GBG: Are the EligStates right for 2-player games? In MT’s ValueFuncC4.updateElig() there is something said about the player and the gradient 🡪 To be checked!!

GBG: Think about better restructuring the software options in TDNTuple2Agt. Perhaps different classes derived from ScoreEval to structure the different g3-eval-branches better?

GBG: Extend StateObervation: if boolean hasOnlyFinalReward() returns true, then set TERNARY=true

### Algorithms

TDNTuple2Agt.trainAgent(so) // train for 1 episode, starting from ‘so’  
 while (gameNotOver) {

ACTIONS\_VT actBest = getNextAction2(so, true, true);  
 ns = new NextState(so,actBest);  
 reward=trainSingleUpdate\_3P(ns, actBest.getVBest(),…)  
 }

and, when game is over:

m\_Net.updateWeightsNewTerminal(…); // drag value of final state towards 0

getNextAction2(so,…) calls directly getNextAction3(so,refer=so,…)  
 for (all available actions acts) {  
 VTable[i] = g3\_Eval\_NPly(so,acts.get(i),refer,nply=1,…)  
 store (one of) the best action(s) with best score vBest in actBest  
 }  
 actBest = new Types.ACTIONS\_VT(actBest.toInt(), randomSelect, VTable, vBest);  
 return actBest; // actBest.getVBest() will return vBest

g3\_Eval\_Nply(so,act,refer=so,nply=1,…) calculates the score s­I+1 = ri+1 +γ Vi+1

if (randomSelect) return random number ∈ [0,1]  
 si+1=NewSO = so.advance(act);  
 return NewSO.getReward(so) + γ⋅getScore(NewSO,so) ;

If (TERNARY) return NewSO.isGameOver() ? NewSO.getReward(so) : γ⋅getScore(NewSO,so) ;

trainSingleUpdate\_3P(ns, target,…) (called from trainAgent) calculates:  
 reward = ns. getNextRewardCheckFinished(…);  
 if (!randomMove) m\_Net.updateWeightsNew(curBoard,…,target) {  
 calculate v\_old, delta=target-v\_old, e  
 update(curBoard,delta,e);  
 else   
 if (m\_elig==RESET) clearEligList();

NTuple2ValueFunc.update(curBoard,…) …

constructs the states equivalent to curBoard (including self), adds this object equiv of equivalent states (with deriv e) to the first place in LinkedList eList of EligStates objects. Elements at the end of eList are deleted if they are beyond horizon.  
Traversing eList from first to last, the EligStates are retrieved and their weights updated with factor λk, starting with k=0. The weight update is done by updating each n-tuple of the current player.

## Some results C4 with buggy version

OLD VERSION, not valid anymore (!)

The overall settings (if not stated otherwise) are: USESYMMETRY=true, ChooseStart01=false, LearnFromRM=false, Reward=Score, Gamma=1.0, NORMALIZE=false, OutputSigmoid=true, MODE\_3P=2, fixed-n-tuple mode 1: 70 8-tuples, TC\_INIT=1e-4, TC-EXP with tcBeta=2.7, rec.weight-change accumulation. 500.000 training games, 10 runs.

1. TDNTal05-lam06-500k.agt.zip: TDNTuple2Agt **without TCL** produces only mediocre results: 500.000 training games for example has a Quick Evaluation result near 0.0.
2. [**TCL-EXP**-al20-lam05-500k-HOR001.agt.zip](file:///C:\Users\wolfgang\Documents\GitHub\GBG\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR001.agt.zip), with settings α=2.0, ε=0.1, λ=0.5, horizonCut=0.01, has **much better** results near **0.43**. High ε, only 5% of the games are TIE. Compete-Both result (50 games) on **one** of the agents around **0.55**.
3. [TCL-EXP-al20-lam05-500k-HOR001-TERNARY.agt.zip](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR001-RESET.agt.zip): same as 2., but [ternary](#TERNARY) update rule: evalQ = **0.44**.
4. [TCL-EXP-al20-lam05-500k-HOR001-RESET.agt.zip](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR001-RESET.agt.zip): same as 2., but elig-RESET: evalQ = **0.43**.
5. [**TCL-EXP**-al20-lam05-500k-HOR010.agt.zip](file:///C:\Users\wolfgang\Documents\GitHub\GBG\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR010.agt.zip): same as 2., but horizonCut=0.10: evalQ = **0.36**.
6. [TCL-EXP-al20-lam05-500k-HOR001-noSYM.agt.zip](file:///C:\Users\wolfgang\AppData\Roaming\Microsoft\Word\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR001-noSYM.agt.zip): same as 2., but USESYMMETRY off: **much worse**, evalQ = **-0.09**.
7. [TCL-EXP-al50-lam05-500k-HOR001.agt.zip](file:///C:\Users\wolfgang\Documents\GitHub\GBG\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR001-RESET.agt.zip): same as 2., but with α=5.0: evalQ = **0.39**. This is inspired by Thill’s α=0.05 which is due to the scale-down in GBG equivalent to GBG α=7.0.
8. [TCL-EXP-al50-lam05-500k-HOR001.agt-DELTA.zip](file:///C:\Users\wolfgang\Documents\GitHub\GBG\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR001-RESET.agt.zip): same as 6., but with accumulation of error signal δ instead of RWC: evalQ = **0.39**.
9. [**TCL-EXP**-al10-lam06-500k-eps0025.agt.zip](file:///C:\Users\wolfgang\Documents\GitHub\GBG\agents\ConnectFour\TCL-EXP-al20-lam05-500k-HOR001.agt.zip), with settings Immediate, tcBeta=2.7, rec.weight-change accumulation, α=1.0, **ε=0.025**, λ=0.6, horizonCut=0.01, output sigmoid, has weaker results near evalQ = **-0.09**. Due to low ε, 50% of the games are TIE (but surprisingly only in the mid-phase of training, in the end the TIE rate is much lower, around 25%).

If we add for the agent in 2. (which had Compete Both score around 0.55) the option “Wrapper nPly=5”, then the Compete Both score increases to **0.7** (and computation is somewhat slower).

If we add to the O-agent (MCTS) the option “Wrapper nPly=1” we get a weird behavior: a) it takes a time, then all 50 game results appear at a sudden and b) the score for X jumps to 1.0 🡪 there must be **a bug with MCTS and Wrapper nPly**.

1. with tmin = 0 or 1 depending on whether we know the afterstate s’0 preceding s0 or not. [↑](#footnote-ref-1)
2. Which initial moves are losing moves? – For small boards up to 4x4 this can be calculated by Minimax, for larger boards this is not viable. But we can ask Hexy (or another strong Hex player) whether it can win as 2nd player after a certain initial move. [↑](#footnote-ref-2)
3. Of course this is inherently a flaw of the design: It is nicer if the information whether to use symmetry or whether to use sigmoid is stored only in one place. We changed it accordingly in TDNTupleAgt, TDNTuple2Agt, NTupleValueFunc, NTuple2ValueFunc. [↑](#footnote-ref-3)
4. A better solution would be to issue a warning when Minimax is attempted as Evaluator and the Hex board is too large. It should also be more transparent which the settings of the counterpart agent are. Perhaps with a new window showing the evaluator settings? [↑](#footnote-ref-4)
5. We tried to use class NextState in getNextAction() as well, but we found that this slows down training dramatically (40% - 70% slower!), so we stick in getNextAction() to the old version. [↑](#footnote-ref-5)
6. since it updates only once and ‘steals’ the target from the successor state of the other player’s value function [↑](#footnote-ref-6)
7. antagonistic reward function: R(st|p(0)) = – R(st|p(1)) for all st [↑](#footnote-ref-7)
8. reward is +1 for one player and -1 for all other players [↑](#footnote-ref-8)
9. This is [equivalent to the 2P-logic](#equivalence_2P). [↑](#footnote-ref-9)
10. Now changed to a safer design: Instead of ConfigGame.FIXEDNTUPLEMODE we have now m\_ntPar.fixedNtupleMode. – How to change fixedNtupleMode for an already trained agent? Start debugger, load agent, stop at a breakpoint, Right mouse - **Watch** on member m\_ntPar, change element fixedNtupleMode, save agent. [↑](#footnote-ref-10)