Strategy TDNTupleAgt:   
How does it work for 2048?

## Literature

[[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)

## Nomenclature

|  |  |
| --- | --- |
| st | state at time t |
| pt ∈ [-1,+1] | player **to move** at time t |
| 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) |

(Currently we have not yet the afterstate logic of [Jaskowski16] integrated, but it is planned).

We describe the TDNTupleAgt-logic for both 1-player and 2-player games, to keep it general. For 1-player games pt is always +1.

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 we always refer to the same player.

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).

## 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 + pt\*V(st+1)   
      (For 1-player games, this simplifies to rt+1 + 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.

## 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 prob. ε a random action and 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 drawback 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](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).

## N-tuple score

**getScoreI()**, the n-tuple score function computes:

OLD target logic (NEWTARGET==false)

NEW target logic (NEWTARGET==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) is

OLD target logic (NEWTARGET==false)

NEW target logic (NEWTARGET==true)

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 σ’(st)=1 and the NEW target logic update rule will add δt/(mNS) to each Vi. Plugging this into the getScoreI-equation we see that the update rule with α=1 has consistently (for each number of n-tuples) the meaning: “Let V(st+1) become immediately the target T” :

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).)

## Update function in 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,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: Use java.util.LinkedList for storage within horizon: we store in this list h+1 objects of class EquivStates 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 🡪 much faster execution when LUTs are large!

## Debugging TDNTuple2Agt

* We realized that even with epsilon=0 frequent **random moves** occurred (??, which should not be!). The reason was wrong code around ‘progress’ and ‘randomSelect’ in TDNTuple2Agt::getNextAction (and likewise in TDNTupleAgt). – Now replaced with the simpler code randomSelect = (rand.nextDouble() < m\_epsilon).   
  No more random moves if m\_epsilon==0. Changed also in TDNTupleAgt.  
  The same code is used in TDAgent::getNextAction 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.   
TODO: Make a plot of the new ε-curves. If necessary, extend the ε-scheme to have beside the sigmoidal falloff a linear falloff (in the new logic) as well.

* 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 fundamental 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. Remedy: 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 it 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](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.
* 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(s)=0 is not wrong (!). Because Minimax as an exact agent has exactly this V(s)=0. That is we can reach a tie with EVERY initial move.
* 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.
* Think about: **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.
* TODO: Make a **tougher evaluator for Hex**: (a) one game from empty board (weight 50%) and (b) N games, when 1st player makes one of the N losing moves[[1]](#footnote-1) and the agent plays 2nd player. These N games weigh another 50%. Calculate the weighted sum of wins 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.
* TODO: implement afterstate logic
* TODO: member randomSelect may become part of Types.ACTIONS 🡪 simpler bookkeeping in train functions.

## Hex and TDNTupleAgt Debugging

There are two problems with Hex and TDNtupleAgt

* 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.[[2]](#footnote-2)
* 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.[[3]](#footnote-3) 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 when we use 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 for an **afterstate** s’ (so.advanceDeterministic) and in the next round, after

curBoard = nextBoard,

curBoard will be an afterstate as well. On the other hand, so in line 3 is the **state** s’’ (with random tile added) which we need 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. Have in trainNewTargetLogic the weight update only if curBoard!=null. This ensures the “if (t>0) …” part in [Jaskowski16].   
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), as desired.

The rest remains the same.

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

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 means more code changes, but the resulting code is better understandable and thus better maintainable. We use class NextState in trainAgent() and related functions.[[4]](#footnote-4)

### Some other items

* Possible bug in 2048: PlayAgent results and QuickEvaluation results seem different (PlayAgent gets less often tile 4096).
* In 2048 Quick Eval’s Average Score after 50 games is varying a lot. Might be due to low outliers? We tested whether a Median Score would be more stable. But it turns out, that it has similar fluctuations. The only way seems to be that we repeat Quick Eval (e.g. 10 times) and take the average of the Average Score (with a standard deviation reduced by a factor of sqrt(10)).
* Multitrain: add intermediate eval and print results to file.
* Heap space: When loading the big 44MB-agent (FIXEDNTUPLEMODE=2) and then starting the training anew, we get a heap space error, even with VM-argument -Xmx1024M. Look for memory leaks!

## Useful Parameter Settings

### 2048

TD-NTuple-2: eps=0, alpha=0.2🡪0.1, lambda=0, NO output sigmoid, USESYMMETRY, AFTERSTATE, FIXEDNTUPLEMODE=1 [Jaskowski2016, Fig. 3b], 10.000 games 🡪 Quick Eval Score approx. 34.000

TD-NTuple-2: eps=0, alpha=0.2🡪0.1, lambda=0, NO output sigmoid, USESYMMETRY, AFTERSTATE, FIXEDNTUPLEMODE=2 [Jaskowski2016, Fig. 3c], 40.000 games 🡪 Quick Eval Score approx. 50.000 (!), but mighty LUT: Even as ZIP, the agt.zip is 44 MB (!)

### 4x4 Hex

TD-NTuple-2: eps=1.3🡪0, alpha=0.5, lambda=0, output sigmoid, USESYMMETRY, random n-tuple 20\*5, 50.000 games, Quick Eval Mode 0 (not 2) 🡪 good Inspect initial board

1. 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-1)
2. 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-2)
3. 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-3)
4. 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 to the old version. [↑](#footnote-ref-4)