Batch Monte Carlo Tree Search

Tristan Cazenave

LAMSADE, Université Paris-Dauphine, PSL, CNRS
Tristan.Cazenave@dauphine.psl.eu

Abstract

Making inferences with a deep neural network on a batch of states is much faster with a GPU than making inferences on one state after another. We build on this property to propose Monte Carlo Tree Search algorithms using batched inferences. Instead of using either a search tree or a transposition table we propose to use both in the same algorithm. The transposition table contains the results of the inferences while the search tree contains the statistics of Monte Carlo Tree Search. We also propose to analyze multiple heuristics that improve the search: the μ FPU, the Virtual Mean, the Last Iteration and the Second Move heuristics. They are evaluated for the game of Go using a MobileNet neural network.

Introduction

Monte Carlo Tree Search (MCTS) using a combined policy and value network is used for complex two-player perfect information games such as the game of Go (Silver et al. 2018). MCTS is also used for many other games and problems (Browne et al. 2012). We propose multiple optimizations of MCTS in the context of its combination with deep neural networks. With current hardware such as GPU or TPU it is much faster to batch the inferences of a deep neural network rather than to perform them sequentially. We give in this paper MCTS algorithms that make inferences in batches and some heuristics to improve them. We evaluate the search algorithms for the game of Go.

The second section deals with existing work on MCTS for games. The third section presents our algorithms. The fourth section details experimental results.

Monte Carlo Tree Search

MCTS has its roots in computer Go (Coulom 2006). A theoretically well founded algorithm is UCT (Kocsis and Szepesvári 2006). Dealing with transpositions in UCT was addressed with the UCD algorithm (Saffidine, Cazenave, and Méhat 2011). The authors tested various ways to deal with transpositions and gave results for multiple games in the context of General Game Playing (GGP).

The GRAVE algorithm (Cazenave 2015) is successful in GGP. It uses a transposition table as the core of the tree search algorithm. Entries of the transposition table contain various kind of information such as the statistics on the moves as well as the generalized All Moves As First (AMAF) statistics. It does not use the UCB bandit anymore but an improvement of RAVE (Gelly and Silver 2011).

PUCT combines MCTS with neural networks (Silver et al. 2016). It is the current best algorithm for games such as Go (Silver et al. 2017) and Shogi (Silver et al. 2018). It was used in the AlphaGo program (Silver et al. 2016) as well as in its descendants AlphaGo Zero (Silver et al. 2017) and Alpha Zero (Silver et al. 2018).

The PUCT bandit is:

$V(s,a) = Q(s,a) + c \times P(s,a) \times \frac{\sqrt{N(s)}}{1+N(s,a)}$

Where $P(s,a)$ is the probability of move a to be the best moves in state s given by the policy head of the neural network, $N(s)$ is the total number of descents performed in state s and $N(s,a)$ is the number of descents for move a in state s.

Many researchers have replicated the Alpha Zero experiments and also use the PUCT algorithm (Tian et al. 2019; Pascutto 2017; Emslie 2019; Wu 2019; Cazenave et al. 2020).

Parallelization of MCTS

Three different ways to parallelize UCT were first proposed in 2007 (Cazenave and Jouandeau 2007). They were further renamed Root Parallelization, Leaf Parallelization and Tree Parallelization (Chaslot, Winands, and van Den Herik 2008; Cazenave and Jouandeau 2008). Root Parallelization simply performs multiple independent tree searches in parallel. Leaf Parallelization performs multiple playouts in parallel at each leaf. Tree Parallelization makes multiple threads descend a shared tree in parallel.

Virtual Loss

The virtual loss enables to make multiple descents of the tree in parallel when the results of the evaluations at the leaves are not yet known and the tree has not yet been updated with these results. It is used in Tree Parallelization. The principle is very simple since it consists in adding a predefined number of visits to the moves that are played during the tree descent.

The virtual loss is used in most of the Go programs including AlphaGo (Silver et al. 2016) and ELF (Tian et al. 2019).

A related algorithm is the Watch the Unobserved in UCT algorithm (Liu et al. 2018). It counts the number of rollouts that have been initiated but not yet completed, named unobserved samples and includes them in the UCT bandit. It gives good results on Atari games. It is quite different from our approach since it uses plain UCT and since the principle is to add unobserved samples to the number of simulations in the UCT bandit.

The virtual loss is also used in TDS-UCT (Yang, Aasawat, and Yoshizoe 2020) together with a message passing system that addresses load balancing. TDS-UCT was applied to molecular design with success.

Batched Inferences

Using batch forwards of the neural network to evaluate leaves of the search tree and find the associated priors given by the policy head is current practice in many game programs (Tian et al. 2019; Pascutto 2017; Emslie 2019; Wu 2019; Cazenave et al. 2020). Usually a set of leaves is generated using multiple tree descents. The diversity of the leaves is obtained thanks to the virtual loss. The neural network is run on a single batch of leaves and the results are incorporated into the search tree, backing up the evaluations up to the root. This algorithm is much worse than sequential PUCT when using the same number of evaluations. However as it makes much more evaluations than sequential PUCT in the same time it recovers when given the same time.

First Play Urgency

Vanilla UCT begins by exploring each arm once before using UCB. This behavior was improved with the First Play Urgency (FPU) (Wang and Gelly 2007). A large FPU value ensures the exploration of each move once before further exploitation of any previously visited move. A small FPU ensures earlier exploitation if the first simulations lead to an urgency larger than the FPU.

In more recent MCTS programs using playouts, FPU was replaced by RAVE (Gelly and Silver 2011) which uses the AMAF heuristic so as to order moves before switching gradually to UCT. RAVE was later improved with GRAVE which has good results in GGP (Browne et al. 2019; Sironi 2019).

In AlphaGo (Silver et al. 2016), the FPU was revived and is set to zero when the evaluations are between -1 and 1. We name this kind of FPU the constant FPU. It has deficiencies. When the FPU is too high, all possible moves are tried at a node before going further below in the state space and this slows down the search and makes it shallow. When the FPU is too low, the moves after the best move are only tried after many simulations and the search does not explore enough. When the constant is in the middle of the range of values as in AlphaGo, both deficiencies can occur, either when the average of the evaluations is below the constant or is greater than the constant.

In other programs such as ELF (Tian et al. 2019) the FPU is set to the best mean of the already explored moves. This is better than using a constant FPU since the best mean is related to the mean of the other moves. It encourages exploration.

The Batch MCTS Algorithm

In this section we describe the Batch MCTS algorithm and its refinements. We first explain how we deal with trees and transposition table. We then detail the associated heuristics: the μ FPU, the Virtual Mean, the Last Iteration and the Second Move.

Trees and Transposition Table

The principle of Batch MCTS is to simulate a sequential MCTS using batched evaluations. In order to do so it separates the states that have been evaluated from the search tree. The transposition table only contains the states that have been evaluated. The search tree is developed as in usual sequential MCTS. When the algorithm reaches a leaf it looks up the state in the transposition table. If it is present then it backpropagates the corresponding evaluation. If it is not in the transposition table it sends back the Unknown value. The algorithm has two options: developing the tree and building the batch. When it develops the tree it stops the search as soon as an Unknown value is returned. When it builds the batch it continues searching when an Unknown value is returned. In this case it adds the corresponding state to the next batch of states, and updates statistics in a different way than when developing the tree.

Batch MCTS increases the number of tree descents for a given budget of inferences compared to usual MCTS with a transposition table in place of a tree. An entry in the transposition table of Batch MCTS can serve as a leaf multiple times. Batch MCTS will redevelop a subtree multiple times with only a small increase in search time since the costly part of the algorithm is the evaluation of states and that previous common states evaluations are cached in the transposition table. Redeveloping common subtrees makes the statistics of the moves not biased by reaching some already developed states from a different path. In this case usual MCTS will go directly to a leaf of the subtree instead of redeveloping it.

Batch PUCT uses one transposition table and two trees. The transposition table records for each state that has been given to the neural network the evaluation of the state and the priors for the moves of the state. The first tree records the statistics required to calculate the PUCT bandit and the address of the children that have already been explored. The second tree is a copy of the first tree used to build the next batch of states that will be then given to the neural network.

Algorithm 1 gives the main PUCT search algorithm using a transposition table of evaluated states and the two trees. The GetBatch boolean is used to make the distinction between the first option to develop the tree and the second option to build the batch. The first tree is the main search tree while the second tree is only used to build the batch.

Algorithm 2 gives the usual way of updating the statistics used for the main tree. Algorithm 3 gives the update of the statistics for the second tree. It updates the statistics as usual when an evaluation is backpropagated, but it updates the statistics differently when an Unknown value is back-

propagated. In this case it can either use the usual virtual loss or the Virtual Mean that will be described later.

The main algorithm is the GetMove algorithm (algorithm 6). It calls the GetBatch algorithm (algorithm 4) that descends the second tree many times in order to fill the batch. It then makes inferences on the built batch and calls the PutBatch algorithm (algorithm 5) that put the results of the inferences in the transposition table and then updates the main tree. GetBatch, forward and PutBatch are called B times. In the end the GetMove algorithm returns the most simulated move of the main tree.

The μ FPU

The way we deal with the FPU is to set it to the average mean of the node (using the statistics of all the explored moves). We name this kind of FPU the μ FPU.

The Virtual Mean

Tree parallel MCTS uses a virtual loss to avoid exploring again and again the same moves when no new evaluation is available. We propose the Virtual Mean as an alternative to the virtual loss. The Virtual Mean increases the number of simulations of the move as in the virtual loss but it also adds the mean of the move times the virtual loss to the sum of the evaluations of the move in order to have more realistic statistics for the next descent.

Algorithm 3 gives the different ways of updating the statistics of a node. The vl variable is the number of virtual losses that are added to a move when it leads to an unknown leaf. A value greater than one will encourage more exploration and will avoid resampling again and again the same unknown leaf. The value is related to the maximum number of samples allowed in the GetBatch algorithm (the variable N in algorithm 4). A low value of N will miss evaluations and will not completely fill the batch. A large value of N will better fill the batch but will take more time to do it. Increasing vl enables to fill the batch with more states for the same value of N. However a too large value of vl can overlook some states and decrease the number of visited nodes in the main search tree. The res value is the evaluation returned by the tree descent, m is the move that has been tried in the descent, s is the state and t is the second tree.

When the Virtual Mean option is used, the sum of the evaluations of the move and the sum of the evaluations of the node are both increased by vl times the average evaluation of the move. It will give a better insight of the real average of the move after the backpropagation than using the virtual loss alone.

Algorithm 4 gives the main algorithm to build the batch. In order to present the algorithm simply we assume a copy of the main tree to treeBatch which is then used and modified in order to build the batch. A more elaborate implementation is to separate inside a node the statistics of the main tree and the statistics made during the building of the batch. A global stamp can be used to perform a lazy reinitialization of the batch statistics at each new batch build.

Algorithm 1: The BatchPUCT algorithm

```
1:  Function BatchPUCT (s, GetBatch)
2:    if isTerminal (s) then
3:      return Evaluation (s)
4:    end if
5:    if GetBatch then
6:      t ← treeBatch
7:    else
8:      t ← tree
9:    end if
10:   if s ∉ t then
11:     if s ∉ transposition table then
12:       if GetBatch then
13:         add s to the batch
14:       end if
15:       return Unknown
16:     else
17:       add s to t
18:       return value (s)
19:     end if
20:   end if
21:   bestScore ← −∞
22:   for m ∈ legal moves of s do
23:     μ ← FPU
24:     if t.p(s, m) > 0 then
25:       μ ← t.sum(s,m) / t.p(s,m)
26:     end if
27:     bandit ← μ + c × t.prior(s, m) × √t.p(s) / (1+t.p(s,m))
28:     if bandit > bestScore then
29:       bestScore ← bandit
30:       bestMove ← m
31:     end if
32:   end for
33:   s1 ← play (s, bestMove)
34:   res ← BatchPUCT (s1, GetBatch)
35:   if GetBatch then
36:     UpdateStatisticsGet (res, bestMove, s, t)
37:   else
38:     UpdateStatistics (res, bestMove, s, t)
39:   end if
40:   return res
```

Algorithm 2: The UpdateStatistics algorithm for the main tree

```
1: Function UpdateStatistics (res, m, s, t)
2:   if res ≠ Unknown then
3:     t.p(s, m) ← t.p(s, m) + 1
4:     t.sum(s, m) ← t.sum(s, m) + res
5:     t.p(s) ← t.p(s) + 1
6:     t.sum(s) ← t.sum(s) + res
7:   end if
```

Algorithm 3: The UpdateStatisticsGet algorithm

```
1: Function UpdateStatisticsGet (res, m, s, t)
2:    if res = Unknown then
3:       μ ← t.sum(s,m) / t.p(s,m)
4:       if VirtualLoss then
5:          t.p(s, m) ← t.p(s, m) + vl
6:          t.p(s) ← t.p(s) + vl
7:       else if VirtualMean then
8:          t.p(s, m) ← t.p(s, m) + vl
9:          t.sum(s, m) ← t.sum(s, m) + vl × μ
10:         t.p(s) ← t.p(s) + vl
11:         t.sum(s) ← t.sum(s) + vl × μ
12:      end if
13:   else
14:      t.p(s, m) ← t.p(s, m) + 1
15:      t.sum(s, m) ← t.sum(s, m) + res
16:      t.p(s) ← t.p(s) + 1
17:      t.sum(s) ← t.sum(s) + res
18:   end if
```

Algorithm 4: The GetBatch algorithm

```
1: Function GetBatch (s)
2:    treeBatch ← tree
3:    i ← 0
4:    while batch is not filled and i < N do
5:       BatchPUCT (s, True)
6:       i ← i + 1
7:    end while
```

Algorithm 5: The PutBatch algorithm

```
1: Function PutBatch (s, out)
2:    for o ∈ out do
3:       add o to the Transposition Table
4:    end for
5:    res ← BatchPUCT (s, False)
6:    while res ≠ Unknown do
7:       res ← BatchPUCT (s, False)
8:    end while
```

Algorithm 6: The GetMove algorithm

```
1: Function GetMove (s, B)
2:    for i ← 1 to B do
3:       GetBatch(s)
4:       out ← Forward (batch)
5:       PutBatch (s, out)
6:    end for
7:    t ← tree.node(s)
8:    return argmax_m(t.p(s, m))
```

The Last Iteration

At the end of the GetMove algorithm, many states are evaluated in the transposition table but have not been used in the tree. In order to gain more information it is possible to continue searching for unused state evaluations at the price of small inacurracies.

The principle is to call the BatchPUCT algorithm with GetBatch as True as long as the number of Unknown values sent back does not reach a threshold.

The descents that end with a state which is not in the transposition table do not change the statistics of the moves since they add the mean of the move using the Virtual Mean. The descents that end with an unused state of the transposition table modify the statistics of the moves and improve them as they include statistics on more states.

The Last Iteration algorithm is given in algorithm 7. The U variable is the number of visited unknown states before the algorithm stops.

Algorithm 7: The GetMoveLastIteration algorithm

```
1: Function GetMoveLastIteration (s, B)
2:    for i ← 1 to B do
3:       GetBatch(s)
4:       out ← Forward (batch)
5:       PutBatch (s, out)
6:    end for
7:    nbUnknown ← 0
8:    treeBatch ← tree
9:    while nbUnknown < U do
10:      res ← BatchPUCT (s, True)
11:      if res = Unknown then
12:         nbUnknown ← nbUnknown + 1
13:      end if
14:   end while
15:   t ← treeBatch.node(s)
16:   return argmax_m(t.p(s, m))
```

The Second Move Heuristic

Let n_1 be the number of playouts of the most simulated move at the root, n_2 the number of playouts of the second most simulated move, b the total budget and rb the remaining budget. If $n_1 > n_2 + rb$, it is useless to perform more playouts beginning with the most simulated move since the most simulated move cannot change with the remaining budget. When the most simulated move reaches this threshold it is more useful to completely allocate rb to the second most simulated move and to take as the best move the move with the best mean when all simulations are finished.

The modifications of the search algorithm that implement the Second Move heuristic are given in algorithm 8. Lines 33-39 modify the best move to try at the root when the most simulated move is unreachable. In this case the second most simulated move is preferred.

Algorithm 9 gives the modifications of the GetMove algorithm for using the Second Move heuristic. Lines 8-17 choose between the most simulated move and the second most simulated move according to their means.

Algorithm 8: The BatchSecond algorithm

Function BatchSecond $(s, GetBatch, budget, i, root)$
  if isTerminal (s) **then**
    return Evaluation (s)
  end if
  if $GetBatch$ **then**
    $t \leftarrow treeBatch$
  else
    $t \leftarrow tree$
  end if
  if $s \notin t$ **then**
    if $s \notin$ transposition table **then**
      if $GetBatch$ **then**
        add s to the batch
      end if
      return Unknown
    else
      add s to t
      return value (s)
    end if
  end if
  $bestScore \leftarrow -\infty$
  for $m \in$ legal moves of s **do**
    $\mu \leftarrow FPU$
    if $t.p(s, m) > 0$ **then**
      $\mu \leftarrow \frac{t.sum(s,m)}{t.p(s,m)}$
    end if
    $bandit \leftarrow \mu + c \times t.prior(s, m) \times \frac{\sqrt{1+t.p(s)}}{1+t.p(s,m)}$
    if $bandit > bestScore$ **then**
      $bestScore \leftarrow bandit$
      $bestMove \leftarrow m$
    end if
  end for
  if root **then**
    $b \leftarrow highestValue_m(t.p(s, m))$
    $b_1 \leftarrow secondHighestValue_m(t.p(s, m))$
    if $b \geq b_1 + budget - i$ **then**
      $bestMove \leftarrow secondBestMove_m(t.p(s, m))$
    end if
  end if
  $s_1 \leftarrow$ play $(s, bestMove)$
  $res \leftarrow$ BatchSecond $(s_1, GetBatch, budget, i, False)$
  if $GetBatch$ **then**
    UpdateStatisticsGet $(res, bestMove, s, t)$
  else
    UpdateStatistics $(res, bestMove, s, t)$
  end if
  return res

Algorithm 9: The GetMoveSecondHeuristic algorithm

Function GetMoveSecondHeuristic (s, B)
  $b \leftarrow size(batch)$
  for $i \leftarrow 0$ to B **do**
    GetBatchSecond $(s, B \times b, i \times b)$
    $out \leftarrow$ Forward $(batch)$
    PutBatchSecond $(out, B \times b, i \times b)$
  end for
  $t \leftarrow tree$
  $best \leftarrow bestMove_m(t.p(s, m))$
  $\mu \leftarrow \frac{t.sum(s,best)}{t.p(s,best)}$
  $secondBest \leftarrow secondBestMove_m(t.p(s, m))$
  $\mu_1 \leftarrow \frac{t.sum(s,secondBest)}{t.p(s,secondBest)}$
  if $\mu_1 > \mu$ **then**
    return $secondBest$
  else
    return $best$
  end if

Experimental Results

Experiments were performed using a MobileNet neural network which is an architecture well fitted for the game of Go (Cazenave 2021b,a). The network has 16 blocks, a trunk of 64 and 384 planes in the inverted residual. It has been trained on the Katago dataset containing games played at a superhuman level.

The μ FPU

We test the constant FPU and the best mean FPU against the μ FPU. Table 1 gives the average winrate over 400 games of the different FPU for different numbers of playouts. For example, the first cell means that the constant FPU wins 13.00% of its games against the μ FPU when the search algorithms both use 32 playouts per move.

Table 1: Playing 400 games with the constant and the best mean FPUs and different numbers of evaluations against the μ FPU, $\frac{\sigma}{\sqrt{n}} < 0.0250$.

FPU	32	64	128	256	512
constant	0.1300	0.1300	0.0475	0.0175	
best	0.3775	0.3450	0.3275	0.3150	0.2725

It is clear that the μ FPU is the best option. In the remainder of the experiments we use the μ FPU.

Trees and Transposition Table

We now experiment with using a plain tree associated to a transposition table. An entry in the transposition table only contains the evaluation and the prior. A node in the tree contains the children and the statistics of the state. We compare it to the PUCT algorithm with a transposition table that stores both the statistics, the evaluation and the priors. PUCT with a transposition table searches with a Directed Acyclic

Graph while its opponent develops a plain tree with transpositions only used to remember the evaluation of the states.

Table 2 gives the budget used by each algorithm (the number of forward of the neural network), the number of descents of the plain tree algorithm using this budget and the ratio of the number of descents divided by the number of forwards and the win rate of the plain tree algorithm. Both PUCT with a transposition table and the plain tree algorithm are called with a batch of size one. The PUCT with a transposition table algorithm makes exactly as many descents as forwards when the plain tree algorithm makes more descents than forwards. The ratio of the number of descents divided by the number of forwards increases with the budget. We can see that both algorithms have close performances with the plain tree algorithm getting slightly better with an increased budget.

Table 2: Playing 400 games with a tree, a transposition table and a batch of size 1 against PUCT with a transposition table with a batch of size 1, $\frac{\sigma}{\sqrt{n}} < 0.0250$

Budget	Descents	Ratio	Winrate
256	273.08	1.067	0.4800
1024	1 172.21	1.145	0.4875
4096	5 234.01	1.278	0.5275

The Virtual Mean

In order to compare the virtual loss and the Virtual Mean we make them play against the sequential algorithm. They both use batch MCTS. The results are given in Table 3. The first column is the penalty used, the second column is the value of vl the number of visits to add for the penalty used. The third column is the number of batches and the fourth column the size of the batches. The fifth column is the average number of nodes of the tree. The sixth column is the average of the number of useful inferences made per batch. The number of inferences made can be smaller than the batch size since the batch is not always fully filled at each call of the GetBatch algorithm. The last column is the win rate against the sequential algorithm using 64 batches of size 1. All experiments are made with the maximum number of descents $N = 500$. It is normal that the number of inferences per batch is smaller than the batch size since for example the first batch only contains one state because the priors of the root are not yet known.

The best result for the Virtual Loss is with $vl = 3$ when using 8 batches. It scores 20.75% against sequential PUCT with 64 state evaluations. The Virtual Mean with 8 batches has better results as it scores 31.00% with $vl = 3$ against the same opponent.

We also tested the virtual loss and the Virtual Mean for a greater number of batches. For 32 batches of size 32 (i.e. inferences on a little less than 1024 states) the best result for the virtual loss is with $vl = 2$ with an average of 157.69 nodes in the tree and a percentage of 79.00% of wins against sequential PUCT with 64 state evaluations. The Virtual Mean with $vl = 1$ and the same number of batches is much better: it has on average 612.02 nodes in the tree and a percentage of wins of 97.00% of its games.

In the remaining experiments we use the Virtual Mean.

Table 3: Playing 400 games with the different penalties (VL = Virtual Loss, VM = Virtual Mean) against sequential PUCT with 64 state evaluations. $\frac{\sigma}{\sqrt{n}} < 0.0250$

P	vl	B	Batch	Nodes	Inference	Winrate
VL	1	8	32	24.47	23.17	0.1300
VL	2	8	32	24.37	24.46	0.1525
VL	3	8	32	24.11	25.16	**0.2075**
VL	4	8	32	23.87	25.53	0.2025
VL	5	8	32	23.91	25.72	0.1600
VL	1	32	32	166.09	28.08	0.7725
VL	2	32	32	157.69	28.25	**0.7900**
VL	3	32	32	151.02	28.30	0.7800
VL	4	32	32	144.45	28.19	0.7550
VM	1	8	32	46.45	20.22	0.2625
VM	2	8	32	43.75	21.64	0.3025
VM	3	8	32	41.63	22.10	**0.3100**
VM	4	8	32	40.41	22.53	0.2400
VM	1	32	32	612.02	26.63	**0.9700**
VM	2	32	32	619.07	27.83	0.9675
VM	3	32	32	593.91	28.20	0.9500

The Last Iteration

Table 4 gives the result of using the Last Iteration heuristic with different values for U. The column vll contains the value of vl used for the Last Iteration. We can see that the win rates are much better when using 8 batches than for Table 3 even for a small U. A large virtual loss (vll) of 3 makes more descents but it is less accurate. Using a virtual loss of 1 is safer and gives similar results.

When using 32 batches against 512 states evaluations the win rate increases from 62.75% for $U = 0$ to 68.00% for $U = 40$.

The Second Move Heuristic

Table 5 gives the winrate for different budgets when playing PUCT with the second move heuristic against vanilla PUCT. We can see that the Second Move heuristic consistently improves sequential PUCT.

Ablation Study

The PUCT constant $c = 0.2$ that we used in the previous experiments was fit to the sequential PUCT on a DAG with 512 inferences. In order to test the various improvements we propose to fit again the c constant with all improvements set on. The results of games against sequential PUCT for different constants is given in Table 6. The $c = 0.5$ constant seems best and will be used in the ablation study.

Table 4: Playing 400 games with the Last Iteration algorithm against sequential PUCT with P state evaluations. $\frac{\sigma}{\sqrt{n}} < 0.0250$.

U	vl	vll	B	Batch	P	Nodes	Inference	Winrate
10	3	3	8	32	64	109.02	21.90	0.4975
10	3	1	8	32	64	75.09	22.04	0.4450
40	3	3	8	32	64	232.09	22.17	0.5100
40	3	1	8	32	64	129.90	22.02	0.5275
0	1	1	32	32	512	729.41	28.88	0.6275
40	1	3	32	32	512	962.84	28.81	0.6650
40	1	1	32	32	512	835.07	28.86	0.6800

Table 5: Playing 400 games with the second move heuristic used at the root of sequential PUCT against sequential PUCT. $\frac{\sigma}{\sqrt{n}} < 0.0250$.

Budget	Winrate
32	0.5925
64	0.6350
128	0.6425
256	0.5925
512	0.6250
1024	0.5600

Table 6: Result of different constants for 32 batches of size 32 against sequential PUCT with 256 state evaluations and a constant of 0.2. $\frac{\sigma}{\sqrt{n}} < 0.0250$.

c	Winrate
0.2	0.7575
0.3	0.7925
0.4	0.8100
0.5	0.8275
0.6	0.7975
0.7	0.7700
0.8	0.7550
1.6	0.5900

Table 7 is an ablation study. It gives the scores against sequential PUCT with 512 evaluations of the different algorithms using 32 batches with some heuristics removed.

Removing the Virtual Mean heuristic is done by replacing it with the virtual loss heuristic. However the virtual loss combined with the Last Iteration is catastrophic. So we also removed both the Virtual Mean and the Last Iteration heuristics in order to evaluate removing the Virtual Mean.

Removing the μ FPU was done replacing it by the best mean FPU. The Last Iteration uses $vll = 1$ and $U = 40$.

We can observe in Table 7 that all the heuristics contribute significantly to the strength of the algorithm. The Virtual Mean (VM) has the best increase in win rate, going from 29.50% for the virtual loss to 68.00% when replacing the virtual loss by the Virtual Mean. The Second Move heuristic (SM) also contributes to the strength of Batch MCTS. LI stands for the Last Iteration.

Table 7: Playing 400 games with the different heuristics using 32 batches of size 32 against sequential PUCT with 512 state evaluations. $\frac{\sigma}{\sqrt{n}} < 0.0250$.

μ FPU	VM	LI	SM	Winrate
y	y	y	y	0.6800
n	y	y	y	0.4775
y	n	y	y	0.0475
y	n	n	y	0.2950
y	y	n	y	0.6275
y	y	y	n	0.5750

Inference Speed

Table 8 gives the number of batches per second and the number of inferences per second for each batch size. Choosing batches of size 32 enables to make 26 times more inferences than batches of size 1 while keeping the number of useful inferences per batch high enough.

Table 8: Number of batches per second according to the size of the batch with Tensorflow and a RTX 2080 Ti.

Size	Batches per second	Inferences per second
1	38.20	38
2	36.60	73
4	36.44	146
8	33.31	267
16	32.92	527
32	31.10	995
64	26.00	1 664
128	18.32	2 345

Conclusion

We have proposed to use a tree for the statistics and a transposition table for the results of the inferences in the context of batched inferences for Monte Carlo Tree Search. We found that using the μ FPU is what works best in our framework. We also proposed the Virtual Mean instead of the Virtual Loss and found that it improves much Batch MCTS. The Last Iteration heuristic also improves the level of play when combined with the Virtual Mean. Finally the Second Move heuristic makes a good use of the remaining budget of inferences when the most simulated move cannot be replaced by other moves.

References

Browne, C.; Powley, E.; Whitehouse, D.; Lucas, S.; Cowling, P.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1): 1–43.

Browne, C.; Stephenson, M.; Piette, É.; and Soemers, D. J. 2019. A Practical Introduction to the Ludii General Game System. In *Advances in Computer Games*. Springer.

Cazenave, T. 2015. Generalized rapid action value estimation. In *24th International Joint Conference on Artificial Intelligence*, 754–760.

Cazenave, T. 2021a. Improving Model and Search for Computer Go. In *IEEE Conference on Games*.

Cazenave, T. 2021b. Mobile Networks for Computer Go. *IEEE Transactions on Games*.

Cazenave, T.; Chen, Y.-C.; Chen, G.-W.; Chen, S.-Y.; Chiu, X.-D.; Dehos, J.; Elsa, M.; Gong, Q.; Hu, H.; Khalidov, V.; Cheng-Ling, L.; Lin, H.-I.; Lin, Y.-J.; Martinet, X.; Mella, V.; Rapin, J.; Roziere, B.; Synnaeve, G.; Teytaud, F.; Teytaud, O.; Ye, S.-C.; Ye, Y.-J.; Yen, S.-J.; and Zagoruyko, S. 2020. Polygames: Improved Zero Learning. *ICGA Journal*, 42(4): 244–256.

Cazenave, T.; and Jouandeau, N. 2007. On the parallelization of UCT. In *proceedings of the Computer Games Workshop*, 93–101.

Cazenave, T.; and Jouandeau, N. 2008. A parallel Monte-Carlo tree search algorithm. In *International Conference on Computers and Games*, 72–80. Springer.

Chaslot, G. M.-B.; Winands, M. H.; and van Den Herik, H. J. 2008. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, 60–71. Springer.

Coulom, R. 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, 72–83.

Emslie, R. 2019. Galvanise Zero. https://github.com/richemslie/galvanise_zero.

Gelly, S.; and Silver, D. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artif. Intell.*, 175(11): 1856–1875.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, 282–293.

Liu, A.; Chen, J.; Yu, M.; Zhai, Y.; Zhou, X.; and Liu, J. 2018. Watch the unobserved: A simple approach to parallelizing monte carlo tree search. *arXiv preprint arXiv:1810.11755*.

Pascutto, G.-C. 2017. Leela Zero. https://github.com/leela-zero/leela-zero.

Saffidine, A.; Cazenave, T.; and Méhat, J. 2011. UCD: Upper Confidence Bound for Rooted Directed Acyclic Graphs. *Knowledge-Based Systems*, 34: 26–33.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587): 484–489.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T. P.; Simonyan, K.; and Hassabis, D. 2017. Mastering the game of go without human knowledge. *nature*, 550(7676): 354–359.

Sironi, C. F. 2019. *Monte-Carlo Tree Search for Artificial General Intelligence in Games*. Ph.D. thesis, Maastricht University.

Tian, Y.; Ma, J.; Gong, Q.; Sengupta, S.; Chen, Z.; Pinkerton, J.; and Zitnick, C. L. 2019. ELF OpenGo: An Analysis and Open Reimplementation of AlphaZero. *CoRR*, abs/1902.04522.

Wang, Y.; and Gelly, S. 2007. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *2007 IEEE Symposium on Computational Intelligence and Games*, 175–182. IEEE.

Wu, D. J. 2019. Accelerating Self-Play Learning in Go. *CoRR*, abs/1902.10565.

Yang, X.; Aasawat, T. K.; and Yoshizoe, K. 2020. Practical Massively Parallel Monte-Carlo Tree Search Applied to Molecular Design. *arXiv preprint arXiv:2006.10504*.