Study of Learning to Search with MCTSnets

Merlin Laffitte
CentraleSupelec
merlin.laffitte@student.ecp.fr

Francois Meunier
CentraleSupelec
francois.meunier@student.ecp.fr

Horace Guy
CentraleSupelec
horace.guy@student.ecp.fr

Hai Xuan Xavier Tao
CentraleSupelec
hai-xuan.tao@student.ecp.fr

April 1, 2019

***Abstract*— This paper is a review of ”Learning to Search with MCTSnet”, published by DeepMind. We provide a detailed explanation of the neural network and a discussion around its implementation.**

I. Introduction

In reinforcement learning, Monte Carlo Tree Search (MCTS) has proven to be an efficient algorithm in a variety of games such as Chess and Go [4]. Early 2018, DeepMind's researchers proposed a new version of MCTS, called MCTSnet [2]. The aim of this neural network architecture is to learn how to perform a tree search, therefore removing the need of exploration rules. In order to do that, several small neural networks are combined in one which performs a tree search thanks to extensive flow control. The final net is differentiable almost everywhere, however some difficulties still arise during back propagation. We implemented the net using PyTorch and focused on two games: Sokoban, as per the initial article, and a simple game called MouseGame.

We will first explain the standard MCTS algorithm, then discuss the MCTSnet algorithm and finally some implementation details and results.

II. MCTS

The goal of planning is to find the optimal strategy that maximizes the total reward in an environment defined by a deterministic transition model $s = T(s, a)$ mapping each state and action to a successor state s, and a reward model $r(s, a)$, describing the goodness of each transition.

MCTS uses an empiric evaluation of the mean reward in each node to find the value of the node.

The model is based on the iteration of the Q function at each action:

$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{N(s,a)+1}(Rt - Q(s,a))$$

$$R_t = \sum_{t'=t}^{L-1} \gamma^{t'-t} t_{t'} + \gamma^{L-t} V(s_L)$$

The action a chosen to learn from is based on a simulation policy π that finds the optimum trade off between exploration and exploitation. N is simply the number of time a node is visited.

In this value based version, the evaluation function V can be of any form, learned from the data, or in a more traditional way a ”rollout” (or playout) simulation is done. For a fixed n integer, n actions are performed randomly. The final value of the leaf is the sum of all rewards. The overall algorithm is presented in Figure 2 while a tree approach is shown in Figure 1.

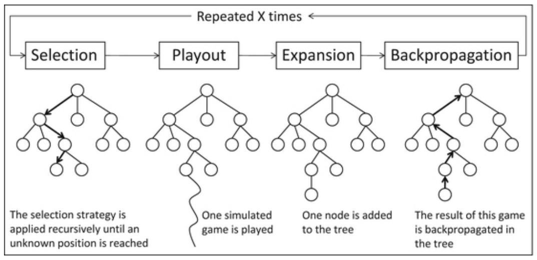
Repeated X times
Selection
Playout
Expansion
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One simulated game is played
One node is added to the tree
The result of this game is backpropagated in the tree

Fig. 1: *Tree representation of MCTS*

A parallel can be made between the two figures: (Selection) is the forward simulation until a leaf node is reached (2), (Playout and Expansion) is the evaluation of the leaf (3), (Backpropagation) is the back-up phase (4).

The final action selection is not shown in both figures. In multiplayer games we usually select the most visited node from the root, but other rules are possible for single player games (such as selecting the node with the best value). The simulation policy is also subject to discussion, while the most common approach is the UCT variation [1].

This algorithm, however powerful in practice, still leads several points to be set or studied for each specific problem. One can try several simulation policies and several leaf evaluation methods. The impact of the reward model should also be discussed, as the reward of each transition of the environment is often set empirically and can have a huge impact on the final search and other approaches proved to be better in some cases [3].

III. MCTSnet

The MCTSnet tries to tackle these difficulties by replacing each part of the algorithm with several neural networks. It also tries to store better statistics of each node: instead of a single value per node (and evaluation of the Q function), a vector of statistics is stored and updated. The algorithm is shown in Figure 3.

```
Algorithm 1: Value-Network Monte-Carlo Tree Search

1. Initialize simulation time $t = 0$ and current node $s_0 = s_A$.
2. Forward simulation from root state. Do until we reach a leaf node ($N(s_t) = 0$):
    (a) Sample action $a_t$ based on simulation policy,
        $a_t \sim \pi(a|s_t, \{N(s_t), N(s_t, a), Q(s_t, a)\})$,
    (b) the reward $r_t = r(s_t, a_t)$ and next state $s_{t+1} = T(s_t, a_t)$ are computed
    (c) Increment $t$.
3. Evaluate leaf node $s_L$ found at depth $L$.
    (a) Obtain value estimate $V(s_L)$,
    (b) Set $N(s_L) = 1$.
4. Back-up phase from leaf node $s_L$, for each $t < L$
    (a) Set $(s, a) = (s_t, a_t)$.
    (b) Update $Q(s, a)$ towards the Monte-Carlo return:
        $Q(s,a) \leftarrow Q(s,a) + \frac{1}{N(s,a)+1}(R_t - Q(s,a))$
        where $R_t = \sum_{t'=t}^{L-1} \gamma^{t'-t} r_{t'} + \gamma^{L-t} V(s_L)$
    (c) Update visit counts $N(s)$ and $N(s,a)$:
        $N(s) \leftarrow N(s) + 1, N(s,a) \leftarrow N(s,a) + 1$
```

Fig. 2: *MCTS Algorithm*

Differences between both algorithms are shown in Table I. If we keep this view it is obvious that both algorithms are really close. However; it is key to understand that the whole algorithm happens inside one neural network in the MCTSnet. This means that the computational graph changes at each iteration and therefore requires extensive flow control, which is provided by PyTorch and Tensorflow in there bare libraries but often not accessible when using wrappers such as Keras.

Precisely this is why the MCTSnet can be seen as a deep residual and recursive networks with shared parameters such as Figure 4. Each simulation is done using the original state and last simulation output (through the updated memory tree and statistics), while each small neural network described in Figure 3 are shared across residual blocks. Each recursive block is a simulation.

	MCTS	MCTSnet
Statistics	Q estimation	h state embeddings
Simulation policy	UCT formula	π neural network
Leaf value estimation	rollout	ϵ embedding initialization
Backup phase	Monte-Carlo return	β neural network on h
Action selection	Most visited node	ρ readout network

TABLE I: *Differences between MCTS and MCTSnet*

```
Algorithm 2: **MCTSnet**
For $m = 1 \dots M$, do simulation:

1. Initialize simulation time $t = 0$ and current node $s_0 = s_A$.
2. Forward simulation from root state. Do until we reach a leaf node ($N(s_t) = 0$):
    (a) Sample action $a_t$ based on *simulation policy*, $a_t \sim \pi(a|h_{s_t}; \theta_s)$,
    (b) Compute the reward $r_t = r(s_t, a_t)$ and next state $s_{t+1} = T(s_t, a_t)$.
    (c) Increment $t$.
3. Evaluate leaf node $s_L$ found at depth $L$.
    (a) Initialize node statistics using the embedding network: $h_{s_L} \leftarrow \epsilon(s_L; \theta_e)$
4. Back-up phase from leaf node $s_L$, for each $t < L$
    (a) Using the backup network $\beta$, update the node statistic as a function of its previous statistics and the statistic of its child:
            $h_{s_t} \leftarrow \beta(h_{s_t}, h_{s_{t+1}}, r_t, a_t; \theta_b)$

After $M$ simulations, readout network outputs a (real) action distribution from the root memory, $\rho(h_{s_A}; \theta_r)$.
```

Fig. 3: *MCTSnet Algorithm*

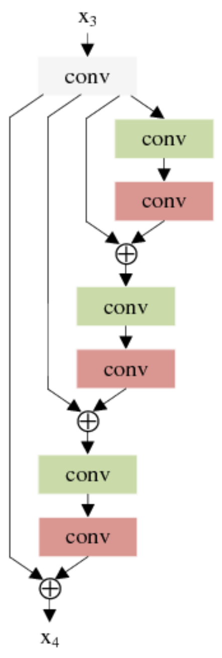

Fig. 4: *A deep residual and recursive network [5].*

IV. Learning

We will focus on supervised learning with a similar setup as the original article: the output of the MCTSnet is trained versus a ground-truth ”best action” that is found thanks to an oracle (let it be a regular MCTS with a high number of simulations and rollout, or any other method). The loss used is the cross-entropy.

One should note that the MCTSnet is not entirely differentiable. Specifically the logic behind the tree exploration (linked to the simulation policy π) is not differentiable as exploring the tree will require the call to the `argmax` function. However; the readout network will use the final state embeddings h, and therefore both networks ϵ (embeddings initialization) and ρ (back-up) are naturally differentiable and gradients can be computed in a straight-forward manner.

We did not investigate the approach proposed in the original article regarding the approximate gradient computations for the simulation policy and we will stick with a simple random simulation policy (as it is stated that it already gives good results, see Figure 5).

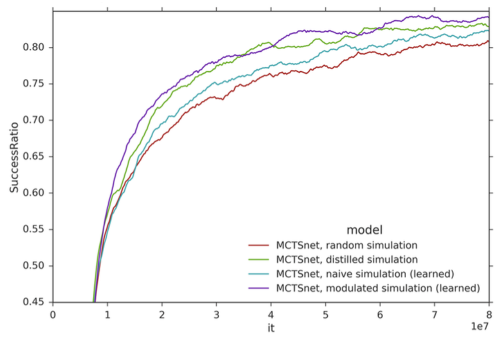

Fig. 5: *Official performance of the MCTSnet with different policies on Sokoban. As shown, the random policy already provides very good results.*

V. Implementation

We implemented the MCTSnet using PyTorch. Our code is freely available at `https://github.com/faameunier/MCTSnet`.

Several implementations of the memory tree were tested (`MCTSnet/memory/`). In our final approach we use a directed graph to store the visited states. This solution will work with a random policy but one should beware that with a deterministic policy this might end up in a closed loop and additional logic should be added. A simple tree is also implemented; however, it might lead to the same state having different embeddings and excessive RAM usage. The directed graph avoids this problem by storing in a separate variable the path followed by the current simulation.

Some basic logic has also been implemented regarding replanning (by cutting the tree or changing the graph root), but it might lead to an exception if the selected action was not visited. Therefore it should be used with caution.

Different implementations of all 4 small networks are provided (`MCTSnet/models/`), simple ones but also all versions mentioned in the original article. These do not rise any problems.

The final MCTSnet is implemented in `MCTSnet/models/MCTSnet.py`. This is where flow control is used and where the MCTSnet logic is implemented. As stated in section 3, we only tested random policies for the tree exploration.

We tried to stick with a simple interface for the environment, and followed as closely as possible the standard Gym methods (`https://gym.openai.com/`). Unfortunately, not all gym environments will work with the MCTSnet as some of the games cannot be copied with `copy.deepcopy`. Our implementation is versatile and one can easily set the input feature space, embedding size and number of actions. All intermediate layers are computed automatically.

Two games are implemented, first the Sokoban (specifically the same implementation as the original article, shown in Figure 6), and another game we like to call the MouseGame (Figure 7, this game was presented during the DeepLearning class at CentraleSupelec).

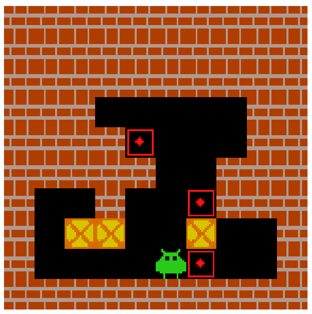

Fig. 6: *Example of Sokoban*

Sokoban solutions are computed by reverse engineering the game creation. This provides poor solution (around 200 steps on average, while a good solution takes on average 50 steps). Nevertheless we had to use this as a starting point as building an ideal solution with a MCTS took too much time to be practical.

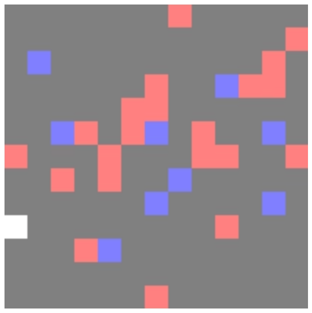

Fig. 7: *Example of MouseGame. The white mouse should eat all red cheese while avoiding the blue poison.*

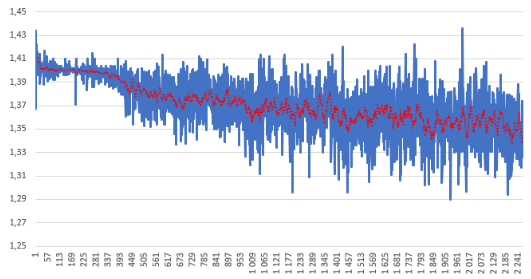

Fig. 8: *Evolution of the loss versus iterations for the mouse games. Unit of iterations is* 10^2 *iterations.*

MouseGame is solved using a MCTS UCT.

Trainers in `MCTSnet/trainer.py` provide an easy way to train, test, and play both games.

VI. Results

We were not able to reproduce the results of the original article with Sokoban. Training time is huge and takes 20 seconds for 100 iterations on a GTX 1070. Training to 5.0×10^6 iterations would take 11 days, which was not possible. Furthermore our oracle was noisy and we doubt that the MCTSnet would have converged with such ground truth.

For the MouseGame we pushed the learning up to 15 hours, and the loss is given in Figure 8. The loss is indeed going down but the variance is increasing. Even after 15 hours of training, the MCTSnet is not able to beat a random agent, which raises some concerns (a standard DQN would give good results in 6 minutes of training, while a MCTS takes 4 seconds to build a very good solution).

It is unclear if the algorithm just needs a lot more training time to perform well, if it requires double precision floating point computation or if they are just a lot of room for improvement in our code.

VII. Conclusion

To conclude we believe that the MCTSnet is a very elegant algorithm; however, it proves to be complicated to implement and unpractical to use. Both building the training set and training the model requires an awful lot of computational power, which may defeat the overall point of it: being faster and better than a standard MCTS. It is unfortunate that the original paper doesn't provide any code, as it would help replicate the claimed results.

References

[1] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon Mark Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1–43, 2012.

[2] Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets. *CoRR*, abs/1802.04697, 2018.

[3] Tom Pepels, Mandy J. W. Tak, Marc Lanctot, and Mark H. M. Winands. Quality-based rewards for monte-carlo tree search simulations. In *ECAI*, 2014.

[4] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[5] Ying Tai, Jian Yang, and Xiaoming Liu. Image super-resolution via deep recursive residual network. *2017 IEEE Conference on Computer Vision*

and Pattern Recognition (CVPR), pages 2790–2798, 2017.