Accelerating Neural MCTS Algorithms using Neural Sub-Net Structures

Prashank Kadam
Vesta Corporation
Portland, Oregon, United States
prashank.kadam@vesta.io

Ruiyang Xu
Northeastern University
Boston, United States
xu.r@northeastern.edu

Karl Lieberherr
Northeastern University
Boston, United States
k.lieberherr@northeastern.edu

ABSTRACT

Neural MCTS algorithms are a combination of Deep Neural Networks and Monte Carlo Tree Search (MCTS) and have successfully trained Reinforcement Learning agents in a tabula-rasa way. These algorithms have been able to find near-optimal strategies through self-play for different problems. However, these algorithms have significant drawbacks; they take a long time to converge, which requires high computational power and electrical energy. It also becomes difficult for researchers without cutting-edge hardware to pursue Neural MCTS research. We propose Step-MCTS, a novel algorithm that creates subnet structures within the complete network, each of which simulates a tree that provides a lookahead for exploration. A Step function is used to switch between the subnet structures. We show how state-of-the-art Neural MCTS algorithms can be extended to Step-MCTS and evaluate their performances. Algorithms extended to Step-MCTS show up to 2.1x decrease in the training times and achieve a faster convergence rate compared to the other widely used algorithms in the Neural MCTS domain.

KEYWORDS

Competitive MARL, Adaptive Learning, Growing Neural Networks

1 INTRODUCTION

In recent years, there has been considerable progress in beating the human benchmarks and achieving superhuman capabilities on games like Chess, Go, Shogi, etc. The main reason for this breakthrough is the accessibility of a larger amount of computational resources and the development of machine learning in these domains. The invention of algorithms like AlphaZero [28], and MuZero [26] has led to computers beating humans in games like Go which was a farfetched idea some years ago. In general, these algorithms are called Neural Monte Carlo Tree Search (MCTS) algorithms as they use a combination of Deep Neural Networks (DNNs) and MCTS to find near-optimal strategies to play these games. Along with this, these algorithms can learn optimal strategies in a tabula-rasa fashion essentially without any prior knowledge except the game rules.

Although these algorithms perform exceptionally well on such complex games, they have some significant drawbacks; These algorithms take incredibly long periods to converge. A user cannot run these algorithms on average computer hardware for complex games; they need state-of-the-art graphics processors to run in parallel for training. This is an important reason why many researchers are unwilling to pursue Neural MCTS research.

In general, it has been observed that larger networks perform better at state estimation but training these networks is computationally very expensive, whereas smaller networks can be trained much faster, but the evaluations are not very accurate.

As part of this paper, we have developed Step MCTS, an algorithm that can be extended to other existing Neural MCTS algorithms to converge much faster on the same hardware configuration in considerably less training time. We achieve this by creating subnet structures within the complete network and using the smaller subnets for faster estimations. We further use these estimations to provide a look ahead to the larger subnets, enabling the complete network to train much faster than standard training methodologies. The idea here is that we start with the most basic possible network configuration, eg., a network with a single input, hidden, and output layer. We start training the agent with a Neural MCTS algorithm (ex. AlphaZero, MuZero, etc.) using this network. A tree is generated using this network, and priorities are given to each of the states visited in the tree. The states with the highest priorities are then stored in the memory. After each training iteration, a step function is called, which decides whether to "switch" or not. Switching here means adding another layer to the network; if the current configuration of the network is good enough to handle the complexity of the problem it is solving, the step function will output a "not switch" action. This means we continue using the same configuration.

On the other hand, if the network is lagging too much in finding better strategies and requires additional resources, the step function will output a "switch" action. In this case, we switch, i.e., add another layer to the network, initialize its parameters and continue training. After the switch event, all the crucial states stored in the memory while training the previous subnet are assigned higher priors in the value function so that the tree generated by the current subnet puts more emphasis on these states for exploration and finding better strategies. This is done because the previous subnet has already tried to find the best possible strategies using its resources, so we need to use this information to form better strategies. The training is thus continued further until a fixed number of iterations or convergence.

We extend our Step-MCTS to one of the most widely used Neural MCTS algorithms, AlphaZero, and evaluate our improvements over the vanilla and advanced versions of this algorithm for different games like Connect-4, Othello and Chess. Our experiments show that Step-MCTS shows up to 2.1x improvements in the training times while having a better rate of improvement during training compared to vanilla versions of these algorithms and some of their special configurations.

Proc. of the Adaptive and Learning Agents Workshop (ALA 2023), Cruz, Hayes, Wang, Yates (eds.), May 9-10, 2023, Online, https://ala2023.github.io/. 2023.

2 BACKGROUND

2.1 Neural MCTS

Monte Carlo Tree Search (MCTS) [5] [17] has been used widely for solving combinatorial problems. It has gained much success in recent times by combining with deep neural networks for value estimations. This concept of Neural MCTS was proposed independently in Expert Iteration [1], and AlphaZero [28]. AlphaZero uses a single neural network as the policy and value approximator. During each learning iteration, it carries out multiple rounds of self-plays. Several MCTS simulations are run to estimate a policy at each state during each self-play. This policy is then sampled to pick a move and continue. The game's outcomes are propagated to each state in its trajectory. All these trajectories are stored in a replay buffer which is later used to train the network.

The MCTS runs for a fixed number of simulations during self-play to generate an empirical policy. Each of these simulations passes through 4 phases:

(1) SELECT: At the beginning of each iteration, the algorithm selects a path from the root (current game state) to a leaf (either a terminal state or an unvisited state) according to an upper confidence boundary (UCB) algorithm [24]. Specifically, suppose the root is s_0. The UCB determines a serial of states $\{s_0, s_1, ..., s_l\}$ by the following process:

$$a_i = \text{argmax}_a \left[Q(s_i, a) + c\pi_\theta(s_i, a) \frac{\sqrt{\sum_{a'} N(s_i, a')}}{N(s_i, a) + 1} \right] \qquad (1)$$
$$s_{i+1} = \text{move}(s_i, a_i)$$

Here Q represents the Q-value of the state-action pair, N is the number of times a state-action was observed and π_θ is the policy learned by the network. Selecting simulation actions using Eq.1 is equivalent to optimizing the empirical policy [11]

$$\hat{\pi}(s, a) = \frac{1 + N(s, a)}{|A| + \sum_{a'} N(s, a')} \qquad (2)$$

where $|A|$ is the size of the current action space, which approximates the solution of a regularized policy optimization problem, which can be stated as follows:

$$\pi^* = \text{argmax}_\pi \left[Q^T(s, \cdot)\pi(s, \cdot) - \lambda KL[\pi_\theta(s, \cdot), \pi(s, \cdot)] \right]$$
$$\lambda = \frac{\sqrt{\sum_{a'} N(s_i, a')}}{|A| + \sum_{a'} N(s, a')} \qquad (3)$$

MCTS simulation will optimize the output policy to maximize the action value output while minimizing the change to the policy network as long as the value output of the network is accurate.

(2) EXPAND: If a selected phase ends at a previously unvisited state s_l, it will be expanded completely and marked as visited. All the child nodes will be considered as leaf nodes during the next iteration.

(3) ROLL-OUT: Every child of the expanded leaf node carries out a roll-out. The algorithm will use the network to estimate the result of the game. This value is then backpropagated to the previous states to move into the next phase.

(4) BACKUP: The statistics for each node in the selected states $\{s_0, s_1, ..., s_l\}$ are updated by the algorithm from the given iteration. To illustrate this process, suppose the selected states and corresponding actions are

$$\{(s_0, a_0), (s_1, a_1), ...(s_{l-1}, a_{l-1}), (s_l, \_)\}$$

Let $V_\theta(s_i)$ be the estimated value for child s_i. The Q-value should be updated such that it equals the average cumulative reward over each access of the underlying states i.e., $Q(s, a) = \frac{\sum_{i=1}^{N(s,a)} \sum_t r_t^i}{N(s,a)}$. To rewrite this updating rule in an iterative form, for each (s_t, a_t) pair, we have:

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$$
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{V_\theta(s_r) - Q(s_t, a_t)}{N(s_t, a_t)} \qquad (4)$$

This process will be carried out for all of the roll-out outcomes from the last phase.

The algorithm returns the empirical policy $\hat{\pi}(s)$ for the current state s after the given number of iterations has been reached. The action is then sampled from the $\hat{\pi}(s)$ after the MCTS simulation, and the game continues.

3 PREVIOUS WORK

Monte Carlo Tree Search [5] [17] has always been poor in performance when compared to other tree search techniques like minimax in the domain of tactical games. MCTS proves to be more beneficial as the state space of the increases and running minimax on such large trees is not computationally feasible. If the exploration and the exploitation is efficiently balanced, MCTS would converge asymptotically [17]. Initially, the maximum backpropagation technique was introduced for updating the node values in the trees [5]. Here we propagate the maximum value instead of the average value so that after a certain point the search algorithm will consider the node to have converged. Maximum backpropagation is still being successfully used in works like BRUE [7] for probabilistic planning and as Bellman backups for Dynamic Programming [15]. The performance of MCTS has also been enhanced using prior knowledge, initial applications showed success in the games of Computer Go [8] and Breakthrough [21]. In this case the values were assigned higher priorities based on some offline learnt technique and then these values were used for simulating trees. Another way of injecting prior knowledge is by using progressive bias during selection [3]. Another technique developed for enhancing the performance of MCTS is using minimax-style backups and heuristic values and replacing them with actual rollouts [23]. Using heuristic evaluations to terminate games like Lines of Action and Amazons early has been a great success [32], [16], [20]. More recently, MCTS has been used with heuristics using implicit minimax backups [19]. Another way to improve the performance of MCTS by using proven wins or losses as additional information in MCTS, MCTS Solver [31] is a prime example of this technique. Finally, there came algorithms like AlphaGo [27] and AlphaZero [28] which used neural networks to estimate the value of the node being evaluated. These techniques have proven to be very successful and are known as neural MCTS algorithms. Recently, warm start techniques have been used along with MCTS to accelerate the process of training [33], [30]. In these

cases, various enhancements like RAVE [9], adaptive starts [29] and Q-value injections [12] have been used to optimize neural MCTS training.

Our approach builds on neural MCTS. It is based on using subnet structures within the neural network to learn basic strategies and give a lookahead for more complex structures by transferring the gained information and the parameters to these complex structures. Although there has been work done in training neural networks using subnets [13], in the neural MCTS domain, there has not been any work done on optimizing the training using subnets. The closest approach to our idea that we could find is the MPV-MCTS [18], it uses two separate networks (smaller and larger) to generate lookahead. Also, there is no concept of switching, within each iteration of training, the smaller and the larger network generated trees are assigned a fix number of simulations. Smaller network tree generally has a higher number of simulations than the larger one.

In our work, we are trying to adapt the algorithm to the complexity of the problem that it is trying to solve by providing it with additional resources. The idea of adaptive computation has been previously explored in the literature. More recently, algorithms like ACT [10] have been helpful in training RNNs to decide how many computational steps should be taken between taking in the input and emitting the output, but this technique has biased gradient estimates. Another algorithm that has been recently developed is called PonderNets [2], which learns the number of computational steps required to achieve a balance between prediction accuracy and computational cost. PonderNets use a step function to determine the time step to stop the execution of a network. Most of the previously used techniques for dynamic resource allocation used Reinforcement Learning [4]. In our work, we also use a step function but this step function is used to inform the network to switch between various configurations of the subnet starting from the lowest possible configuration going all the way up to the full scale network.

4 STEP MCTS

4.1 Subnet

In this section, we discuss the Step MCTS algorithm in detail. Let us call our complete network f, which has N hidden layers. Note that a complete network, in this case, means the upper bound to the number of computational resources available. Thus, N is the highest number of hidden layers which we could add to the network given computational hardware. In theory, N could be as large as possible, but we fix it to some finite value in practice. We define a subnet as a configuration of f that has an input layer, output layer, and n hidden layers such that $n \in [1, N]$. Thus, a subnet with one hidden layer would be the most basic configuration available to the algorithm. Let's call it f_1. We start our training with f_1 and keep on increasing the number of hidden layers as required by the network up to f_{N-1} at which point we would have used up all our resources.

Each of these subnet configurations simulates its tree. Let us call the tree simulated by f_1 as T_1 and so on until $T_{(N-1)}$. Starting from f_1, we train in mini-batches, and after each iteration, the step function tells us whether we move to f_2. If the switch is made, f_2 will help

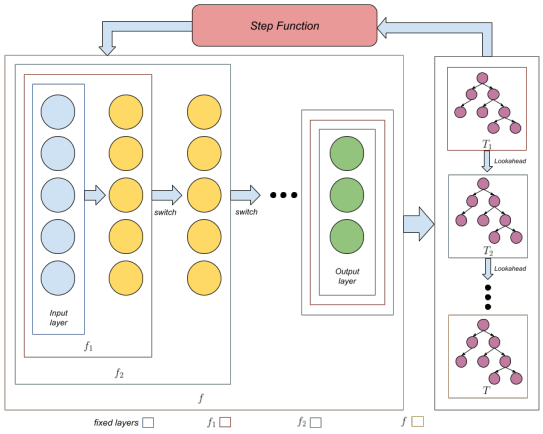

Figure 1: Step-MCTS: Here, we can see how each of the subnets look. f_1 is a subnet with 1 hidden layer, f_2 has 2 hidden layers and this goes on to f_{N-1}. So we begin with f_1, switch to f_2 and so on. Also, we see how each of the trees provides a lookahead to the subsequent tree generated by the next subnet and the step function outputs the switching action. Lookahead here means the priority states from the storage buffer

roll out T_2, which will now use all the important states/observations explored by T_1, thus saving a considerable time on exploration.

When training the subnet, at the end of each iteration of training, the subnet is provided training examples of the form (s_t, π_t, z_t). π_t is an estimate of the policy from state s_t, $z_t \in [-1, 1]$ is the final outcome of the game from the perspective of the player at s_t. All the subnets are then trained to minimise the following loss function (excluding regularisation terms):

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \pi_t . \log(p_\theta(s_t))$$

here p_θ and v_θ are the policy and values functions respectively.

4.2 Step Function

The step function is applied after each iteration of training to decide whether to switch to the following subnet configuration or not. Depending on the requirement, the step function could have any underlying configuration (Threshold based, MLP, RNN, etc.). In this paper we have shown two different configurations - Threshold based and RNN based.

4.2.1 *Threshold based.*

In a threshold based step function, we simply set up a threshold value $\lambda_{threshold}$ where $\lambda_{threshold} \in [0, 1]$, for the proportion of selfplays that should be won during an iteration of training to not make a switch action. If the ratio of the number of selfplays won during an iteration of training gets below the threshold value, then we make a switch to a larger subnet configuration. The subnet would be prone to switch just after making a switching move, to avoid this we introduce a fixed number of iterations that a subnet configuration has to train before taking a switch action.

4.2.2 *Recurrent Neural Network based.*

We have also implemented an RNN-based threshold function. In this case, an RNN is used to predict the switching action (whether to switch or not). In our experiments, we found that the RNN-based step-function takes a lot more time to train and performs as good the threshold-based step function. Hence for the rest of this paper, we will be concentrating on the results of the threshold-based step function. The details of the RNN-based step function can be found in the appendix.

4.3 Lookahead

Each of the subnets simulates a tree T_i; thus, the total number of trees in the complete network will be $T_i \in [T_1, .., T_{N-1}]$. Starting from T_1, each tree simulates faster than the next one. We take advantage of the faster simulations of the initial network subsets to find the states/observations that are most commonly visited. These states/observations are the most likely ones to be explored further by the larger subnet configurations. Thus, rather than waiting for the larger subnets first to discover these states/observations and explore them, we prioritize these states in the value functions of the larger subnets so that these states could be explored immediately, saving time on computation.

To achieve this, we maintain a buffer through the training that stores the values of the states/observations that have been previously explored in a prioritized manner. When the step function returns the "switch" state, before starting the training for the next subnet, the value function of the next subnet is updated with the values from the buffer. The states/observations with higher priorities are assigned higher values, and the value increments are decreased proportionately. Algorithm 1 describes the Step MCTS in detail.

Algorithm 1 Step MCTS

$s \leftarrow$ Initial State/Observation
$f_i \leftarrow f_1$
$T_i \leftarrow T_1$
$m \leftarrow$ Self-plays per iteration
for Each Iteration **do**
  if $S(n) \leftarrow$ RNN based **then**
    for $k \in [1, m]$ **do**
      $\lambda_k \leftarrow$ Sampled from the p_k
    end for
    if $\frac{1}{N}\sum_{k=1}^{K} \lambda_k < p_T$ **then**
      $s_{leaf} \leftarrow$ Select leaf from T_i using lookahead
      if No lookahead is available **then**
        $s_{leaf} \leftarrow$ Select unevaluated leaf from T_i
      end if
      $(p, v) \leftarrow f_i(s_{leaf})$
      Update $(T_i, s_{leaf}, (p, v), p_k)$
    else if $\lambda_k \geq p_T$ **then**
      $f_i \leftarrow f_{i+1}$
      $T_i \leftarrow T_{i+1}$
      $s_{leaf} \leftarrow$ Select leaf from T_i using lookahead
      if No lookahead is available **then**
        $s_{leaf} \leftarrow$ Select unevaluated leaf from T_i
      end if
      $(p, v) \leftarrow f_i(s_{leaf})$
      Update $(T_i, s_{leaf}, (p, v), p_k)$
    end if
  else if $S(n) \leftarrow$ Threshold based **then**
    if $\lambda_{threshold} < n(z_{won})$ **then**
      $s_{leaf} \leftarrow$ Select leaf from T_i using lookahead
      if No lookahead is available **then**
        $s_{leaf} \leftarrow$ Select unevaluated leaf from T_i
      end if
      $(p, v) \leftarrow f_i(s_{leaf})$
      Update $(T_i, s_{leaf}, (p, v), p_k)$
    else if $\lambda_{threshold} > n(z_{won})$ **then**
      $f_i \leftarrow f_{i+1}$
      $T_i \leftarrow T_{i+1}$
      $s_{leaf} \leftarrow$ Select leaf from T_i using lookahead
      if No lookahead is available **then**
        $s_{leaf} \leftarrow$ Select unevaluated leaf from T_i
      end if
      $(p, v) \leftarrow f_i(s_{leaf})$
      Update $(T_i, s_{leaf}, (p, v), p_k)$
    end if
  end if
end for

5 EXPERIMENTS

5.1 Experimental Setup

All the AlphaZero and MuZero based experiments in this paper are performed on an Apple M1 8-core CPU with 8GB of LPDDR4X-4266 MHz SDRAM and an integrated 8-core GPU with 2.6 teraflops of throughput. For our experiments, we will be using a 6-layered convolutional neural network with four convolutional layers and two fully connected layers as our default complete network. Two additional convolutional layers are added to the complete network configuration, only for Chess and Breakout, due to high state space and computational complexity. A complete network is nothing but the upper limit of the resources available to us; thus, if we start from 1 convolutional layer, Step-MCTS can make up to 3 switches during the whole training since we have four convolutional layers in the complete network. We did this to make the comparisons with other algorithms fair as other algorithms would not change the network configurations during training dynamically. Each convolutional layer has a filter of size (3,3). We are keeping the filter size the same throughout the network for simplicity. Note that our algorithm can be used with any configuration of a deep neural network (e.g., MLP, ResNet). The architecture will have to be formed accordingly. Each iteration consists of 100 self-plays, and we use a batch size of 64 for each iteration of training; the optimizer used is Adam, and the regularization parameter is initiated with $1e^{-4}$. For each iteration, the temperature parameter is set to 0 for the first 25 self-plays to explore initially. Later on, we set it to 1. The loss functions used by Step-MCTS for training the main network are the same as its vanilla Neural MCTS version.

The threshold based step function has the threshold value set to 0.8, which means that in each iteration of training, if the agent is not able to win more than 80% of the selfplays, we make a switch.

We came up with this value after extensive experimentation with the RNN based step function.

The RNN based step function is a 2-layered RNN. We used the same batch size and optimizer as the main network. We keep the configurations of the other two algorithms the same as Step MCTS. All the hyperparameters are tuned to the same values. In the case of the network architecture, the depth of vanilla algorithms is kept the same as that of our complete network. For MPV-MCTS, a more extensive network is designed the same as AlphaZero, and the smaller network is a four-layered ConvNet with two convolutional and two fully connected layers. The number of simulations of the smaller tree is set to 1.5 times that of the enormous tree. All the other hyperparameters are kept the same for fair evaluations.

Step-MCTS is implemented in our AlphaZero experiments is we use an initial Policy-Value network (subnet) having only a single convolutional layer. After each training iteration, the self-play trajectories generated are stored in the replay buffer. These trajectories are then used to train the subnet and the step function. If the step function output is higher than the set threshold (we used 0.8 for our experiments), we take a snapshot of the current subnet. We add a new layer to the network, so now we have two convolutional layers, then we load the snapshot and initialize the weights of the new layer.

Along with this, the states that are most commonly visited are stored in the replay buffer in a separate table. The values of these states are then initiated as per their priorities, i.e., the most commonly visited states are given higher initial values so that they are explored more by the MCTS. The new tree created by the next subnet is rolled out based on these values. The training thus continues until the set number of iterations are completed.

5.2 Training Step MCTS

The idea is to use the smaller networks to find the states that are most commonly visited so that we can assign higher values to those states in the policy. The values are not uniformly incremented for all the previously visited states, hence we have priorities. The states visited the most number of times will have the highest priority and the priorities decrease as we go to the less visited states. The values of the states with high priorities are increased by a higher factor in the value function and this factor keeps decreasing as we go to lower priorities. At each step during a self-play, we get a tuple (s,π,z). s is the current state of the board, π is the policy from which we sample a move and z is the outcome of the game. If we find that s is a high-priority state, we increment the value of s in π by some factor (ex. 5%) and save it to the tuple. This is how the values of priority states are inflated. Now, all these inflated, as well as non-inflated (which do not have set priorities) tuples, will be stored in the replay buffer as a part of self-play data where they will be used to train the network in the next training iteration. From an implementation standpoint, we have two buffers, replay buffer will store all the self-play data which will be later on used for training (just like AlphaZero) and can have $5x10^5$ games. The second buffer (let's call it the priority buffer) will have information about states and priorities. When we reach a certain state (board configuration) during self-play, we hash the current configuration of the board (which is a matrix) and save it as a key to a hash table in the priority buffer with value as count set to 1, if the hash already exists, we increment its value by 1. Thus at the end of self-plays for one iteration of training, we have counts of all the states visited and their counts. During the next iteration of training, when we arrive at a certain state, we do a lookup in the hash table to see if the state is present and what's the priority, if it is present, we update the value in the policy accordingly. Also for games with larger states spaces, we just keep the top 10^5 states (based on count) in the buffer so that we do not overflow the buffer.

Training stops (network switches) once the output is sufficiently inaccurate. Step MCTS essentially starts from scratch and asks for more resources as its performance decreases, and the Step function decides when to make that request.

A step function could be any configuration of a neural network. It takes an input x and gives an output $_t$, which is essentially the probability of switching and a hidden state h_t to which the function is applied recurrently for K steps. K refers to the number of self-plays that take place per iteration of training. x is the embedding for each episode of self-play within an iteration of training, and $\lambda \in [0, 1]$ is the probability of switching the network. The switch is made after the value of this probability crosses a certain threshold p_T. After each iteration of training the main network, the step function is called K times recurrently. If $\frac{1}{N}\sum_{t=1}^{K}\lambda_t > p_T$, the main network switches to a bigger configuration immediately. The value of K is limited by the number of self-plays that we configure in each training iteration. We suggest using a higher number of self-plays per iteration in the case of higher state-space games. Having a finite K also helps us normalize the probability distribution, which is a challenge in the case of PonderNets. Care is taken that the network does start switching immediately at the start of training or on a switching action.

The loss function is optimized based on the total weighted loss of the main network in the previous training step. A difference of 1 time-step is maintained during the training between training the main network and the step function. During the initial training or after a switch event occurs, the network will immediately tend to switch to a higher configuration since the total weighted loss is relatively high in the beginning. In these cases, care should be taken that the switch does not occur instantly. For our experiments, we do not allow the network to switch for the first two iterations after training or switching events for smaller games like Connect-4 and Othello. In the case of larger games like Chess, we stop the switching for the first five iterations.

In case of threshold based step function, the switch is made simply if the number of games won is below the set threshold for a given iteration of training. We simple add a new hidden layer to the network and continue training.

The main network is trained the same way as any vanilla Neural MCTS algorithm like AlphaZero. After every switching event, a snapshot of the current network is taken, a new network is initialized using an additional layer, and the snapshot is loaded back. The weights of the additional layer are initialized using the He [14] initialization method, and the training is then resumed.

A memory buffer is used to prioritize the states that have been most commonly visited. Before resuming the training after a switch

event, we initialize the value function with the priors that we calculated using the data from the buffer for each of the states.

6 EVALUATIONS

Step-MCTS is a generic approach that can be extended to various Neural MCTS algorithms. Here we will show how we have extended the most commonly used Neural MCTS algorithms like AlphaZero [28] to Step MCTS. We also evaluate our extended Step-MCTS algorithms for different games. To evaluate our new algorithm, we will be using the 6x6 Connect-4, 8x8 Othello and Chess. We choose these games because they are sufficiently complex to concretely show that our algorithm performs much better than other Neural-MCTS configurations (AlphaZero [28], and MPV-MCTS [18]). Our evaluations are based on the performance ratings of the algorithms and the training times. The performance evaluation metrics that we have used are Elo score [6], and Alpharank [22]. We will calculate these metrics over the training and compare them for different algorithms that we are evaluating.

6.1 AlphaZero

After extensively training Step MCTS with the RNN based step function, we found that it uses a switching strategy where it moves to the next subnet configuration if the network wins less than roughly 80 % of the selfplays in the given iteration of training. The performance of both Threshold based and RNN based step functions is nearly the same when we use this value for the threshold. Along with this we save a large chunck of training time when we do not use the RNN. As you will see in the evaluations done below for different two player games, even the switching action approximately takes place at the same training step in both our step functions. Hence, after this point in this paper, we will be referring to the configuration using the threshold based step function when we refer to the Step MCTS model. Table 1 describes the performance of Step-MCTS for each of the evaluated games.

6.1.1 *Connect-4.*

The first game that we evaluated our algorithm on was 6x6 Connect-4. This game has the lowest state space of all the games we have evaluated in this paper. The performance scores and the training time comparisons obtained can be seen in Figure 7. The figure shows that Step-MCTS converges much faster than vanilla AlphaZero and MPV-MCTS; this is because the Connect-4 game is less complex, and the algorithm can devise a smart winning strategy using smaller subnets, later further optimizing over it using larger subnets.

6.1.2 *Othello.*

The next game that we have evaluated is the 8x8 Othello game. This game has a significantly larger state space as compared to Connect-4. The performance and the training time comparisons can be seen in Figure 3. Here too, the Step-MCTS algorithm converges faster than vanilla AlphaZero or MPV-MCTS. Vanilla AlphaZero took 148 hours to complete 60 iterations of training, and MPV-MCTS took 127 hours, while Step-MCTS was able to do this in 102 hours and achieve faster convergence.

6.1.3 *Chess.*

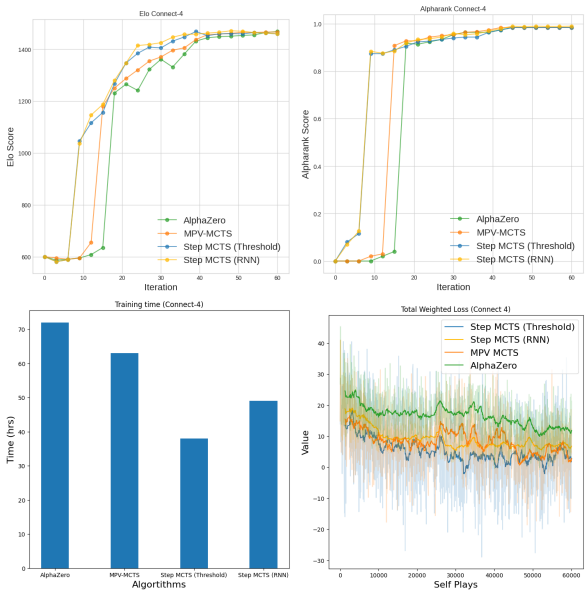

Figure 2: Elo (top-left), Alpharank (top-right), total training time (bottom-left) and total weighted loss (bottom-right) of different algorithms for Connect-4 evaluations

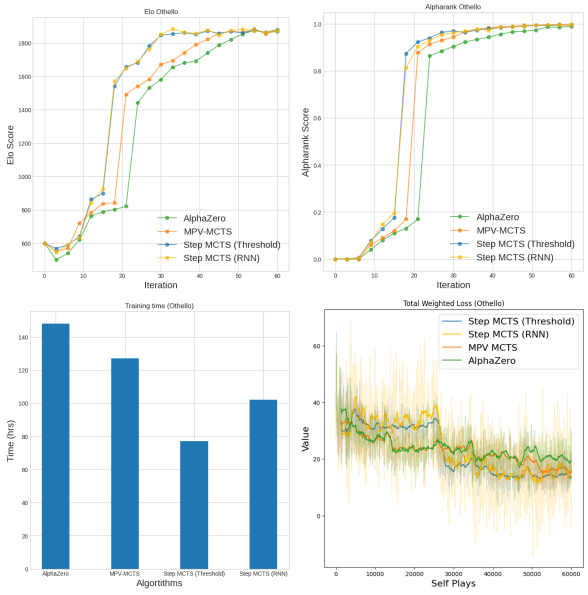

Figure 3: Elo (top-left), Alpharank (top-right), total training time (bottom-left) and total weighted loss (bottom-right) of different algorithms for Othello evaluations

The last game that we evaluated using AlphaZero is Chess which is computationally intensive to train. Two additional convolutional layers were added, now we have six convolutional layers and two fully connected layers in the complete network. While training Step-MCTS for Chess, we observed that the network quickly made

switches in the initial iterations of training to reach the complete network configuration. This was due to the inability of a straightforward network configuration (1 or 2 layers) to handle a complex game like Chess. Hence, we started the training with three convolutional layers. We also kept a default number of iterations that a subnet configuration must complete before taking the switch action. We chose this value as 5 in our Chess experiments. The total time taken by AlphaZero and MPV-MCTS for 300 iterations was 488 and 456 hours, respectively, while Step-MCTS only took 384 hours.

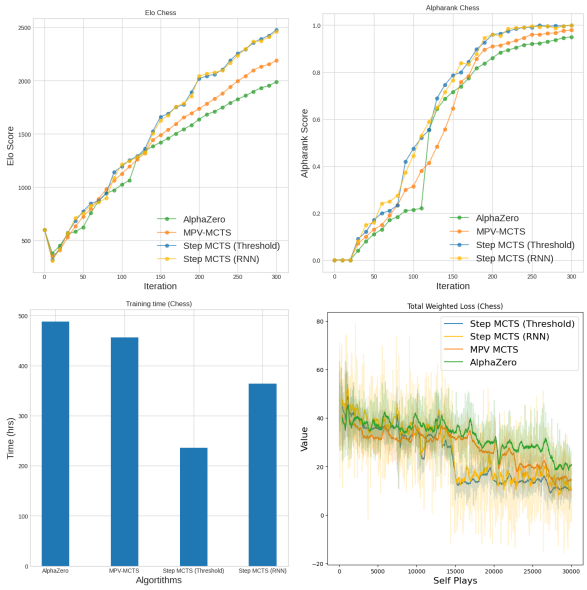

Figure 4: Elo (top-left), Alpharank (top-right), total training time (bottom-left) and total weighted loss (bottom-right) of different algorithms for Chess evaluations

7 CONCLUSION

In this paper, we presented Step-MCTS, a novel algorithm that uses subnet structures within the complete network to generate lookahead that helps in faster training of Neural MCTS algorithms on modest hardware. We extended our algorithm to one of the most commonly used Neural MCTS algorithms, AlphaZero, and showed that using Step-MCTS trains 2 times faster on average over all the games that we have evaluated. Note that along with the improvements with training time, we also saw a faster rate of improvement in the Elo and the AlphaRank scores compared to Vanilla forms of those algorithms. A 2x improvement in training time also means that we are able to save roughly 50% of the electrical energy that would be required on other state of the art models, which is a considerable amount as these algorithms take a very long time to train. Along with this, we believe that an algorithm like Step-MCTS would help facilitate more researchers in Neural MCTS research, which is currently restricted largely to organizations with state-of-the-art hardware. As a future scope of this research, we would like the step function also to predict what kind of layer should be added to the network after the switch event (convolutional, residual, dropout, fully connected, etc.) which, for this paper, we had only considered convolutional layers.

Step MCTS (Threshold)								
Game	Iterations	Self-plays/Iter	Simulations	Switching Iterations	Total Time	Improvement over AZ	Elo Score	AR Score
Connect-4	60	100	25	[4,7,26]	38 hrs	1.89x	1462	0.987
Othello	60	100	50	[3,12,24]	77 hrs	1.92x	1848	0.994
Chess	300	100	50	[7,74,148]	236 hrs	2.06x	2482	0.991

Table 1: Step MCTS training metrics

REFERENCES

[1] Thomas Anthony, Zheng Tian, and David Barber. 2017. Thinking Fast and Slow with Deep Learning and Tree Search. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 5366–5376.

[2] Andrea Banino, Jan Balaguer, and C. Blundell. 2021. PonderNet: Learning to Ponder. *ArXiv* abs/2107.05407 (2021).

[3] Guillaume Chaslot, Mark Winands, H. Herik, Jos Uiterwijk, and Bruno Bouzy. 2008. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation* 04 (11 2008), 343–357. https://doi.org/10.1142/S1793005708001094

[4] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. 2017. Hierarchical Multiscale Recurrent Neural Networks. In *ICLR*. arXiv:1609.01704 http://arxiv.org/abs/1609.01704

[5] Rémi Coulom. 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search, Vol. 4630. https://doi.org/10.1007/978-3-540-75538-8_7

[6] Arpad E Elo. 1978. The rating of chessplayers, past and present. (1978).

[7] Zohar Feldman and Carmel Domshlak. 2013. Monte-Carlo Planning: Theoretically Fast Convergence Meets Practical Efficiency. *Uncertainty in Artificial Intelligence - Proceedings of the 29th Conference, UAI 2013*.

[8] Sylvain Gelly and David Silver. 2007. Combining Online and Offline Knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning* (Corvalis, Oregon, USA) *(ICML '07)*. Association for Computing Machinery, New York, NY, USA, 273–280. https://doi.org/10.1145/1273496.1273531

[9] Sylvain Gelly and David Silver. 2007. Combining Online and Offline Knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning* (Corvalis, Oregon, USA) *(ICML '07)*. Association for Computing Machinery, New York, NY, USA, 273–280. https://doi.org/10.1145/1273496.1273531

[10] Alex Graves. 2016. Adaptive Computation Time for Recurrent Neural Networks. *ArXiv* abs/1603.08983 (2016).

[11] Jean-Bastien Grill, Florent Altché, Yunhao Tang, T. Hubert, Michal Valko, Ioannis Antonoglou, and Rémi Munos. 2020. Monte-Carlo Tree Search as Regularized Policy Optimization. *ArXiv* abs/2007.12509 (2020).

[12] Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Tobias Pfaff, Theophane Weber, Lars Buesing, and Peter W. Battaglia. 2019. Combining Q-Learning and Search with Amortized Value Estimates. https://doi.org/10.48550/ARXIV.1912.02807

[13] Marton Havasi, Rodolphe Jenatton, Stanislav Fort, Jeremiah Zhe Liu, Jasper Snoek, Balaji Lakshminarayanan, Andrew M. Dai, and Dustin Tran. 2020. Training independent subnetworks for robust prediction. *CoRR* abs/2010.06610 (2020). arXiv:2010.06610 https://arxiv.org/abs/2010.06610

[14] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *2015 IEEE International Conference on Computer Vision (ICCV)* (2015), 1026–1034.

[15] Thomas Keller and Malte Helmert. 2013. Trial-Based Heuristic Tree Search for Finite Horizon MDPs. In *Proceedings of the Twenty-Third International Conference on International Conference on Automated Planning and Scheduling* (Rome, Italy) *(ICAPS'13)*. AAAI Press, 135–143.

[16] Julien Kloetzer. 2010. Monte-Carlo techniques : applications to the game of the Amazons.

[17] Levente Kocsis and Csaba Szepesvári. 2006. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning* (Berlin, Germany) *(ECML'06)*. Springer-Verlag, Berlin, Heidelberg, 282–293. https://doi.org/10.1007/11871842_29

[18] Li-Cheng Lan, Wei Li, Ting-Han Wei, and I-Chen Wu. 2019. Multiple Policy Value Monte Carlo Tree Search. In *IJCAI*.

[19] Marc Lanctot, Mark H. M. Winands, Tom Pepels, and Nathan R. Sturtevant. 2014. Monte Carlo Tree Search with heuristic evaluations using implicit minimax backups. In *2014 IEEE Conference on Computational Intelligence and Games*. 1–8. https://doi.org/10.1109/CIG.2014.6932903

[20] Richard J. Lorentz. 2008. Amazons Discover Monte-Carlo. In *Proceedings of the 6th International Conference on Computers and Games* (Beijing, China) *(CG '08)*. Springer-Verlag, Berlin, Heidelberg, 13–24. https://doi.org/10.1007/978-3-540-87608-3_2

[21] Richard J. Lorentz and Therese Horey. 2013. Programming Breakthrough. In *Computers and Games*.

[22] Shayegan Omidshafiei, Christos Papadimitriou, Georgios Piliouras, Karl Tuyls, Mark Rowland, Jean-Baptiste Lespiau, Wojciech M. Czarnecki, Marc Lanctot, Julien Perolat, and Remi Munos. 2019. α-Rank: Multi-Agent Evaluation by Evolution. *Nature* 1038 (Jul 2019).

[23] Raghuram Ramanujan and Bart Selman. 2011. Trade-Offs in Sampling-Based Adversarial Planning.

[24] C. D. Rosin. 2011. Multi-armed bandits with episode context. *Ann. Math. Artif. Intell.* 61, 203–230.

[25] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning internal representations by error propagation.

[26] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. 2019. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. http://arxiv.org/abs/1911.08265 cite arxiv:1911.08265.

[27] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George 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. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (01 2016), 484–489. https://doi.org/10.1038/nature16961

[28] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144. https://doi.org/10.1126/science.aar6404 arXiv:https://www.science.org/doi/pdf/10.1126/science.aar6404

[29] Hui Wang, Mike Preuss, and Aske Plaat. 2020. Warm-Start AlphaZero Self-play Search Enhancements. In *Parallel Problem Solving from Nature – PPSN XVI*, Thomas Bäck, Mike Preuss, André Deutz, Hao Wang, Carola Doerr, Michael Emmerich, and Heike Trautmann (Eds.). Springer International Publishing, Cham, 528–542.

[30] Hui Wang, Mike Preuss, and Aske Plaat. 2021. Adaptive Warm-Start MCTS in AlphaZero-Like Deep Reinforcement Learning. In *PRICAI 2021: Trends in Artificial Intelligence*, Duc Nghia Pham, Thanaruk Theeramunkong, Guido Governatori, and Fenrong Liu (Eds.). Springer International Publishing, Cham, 60–71.

[31] Mark Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. 2008. Monte-Carlo Tree Search Solver. 25–36. https://doi.org/10.1007/978-3-540-87608-3_3

[32] Mark Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. 2010. Monte Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games* 2 (12 2010), 239 – 250. https://doi.org/10.1109/TCIAIG.2010.2061050

[33] Ruiyang Xu, Prashank Kadam, and Karl Lieberherr. 2021. First-Order Problem Solving through Neural MCTS based Reinforcement Learning. https://doi.org/10.48550/ARXIV.2101.04167

A APPENDIX

A.1 RNN-based Threshold Function

RNN [25] is applied K times recurrently where K is equivalent to the batch size. Note that K will be the same for a particular subnet, and the step function will be applied for each mini-batch for $k \in \{1...K\}$. After the step function is applied for each step, we get a scalar probability λ_k based on which we decide whether to switch the network or not. To represent the Markov process, we can simply define a Bernoulli random variable $X_n = Ber(\lambda_k)$ whose properties can be given as follows:

$$\text{Prob. Mass Func.} : P(X = 1) = \lambda_k; P(X = 0) = (1 - \lambda_k)$$

$$\text{Expection} = E[X] = \lambda_k$$

$$\text{Variance} = Var(X) = \lambda_k(1 - \lambda_k)$$

The two states that this variable can take are $X_k = 0$ (continue), which means we do not need to switch the subnet in the current iteration and we can continue with the same subnet and $X_k = 1$ (switch), which means that we need to now switch to the next subnet f_{i+1}. Here we can set the transition probability as follows:

$$P(X_k = 1|X_{k-1} = 0) = \lambda_k \leq k \leq K$$

We will depend on the value of λ_k to decide the optimal value of k, after which we can switch the subnet. Once we get $X_k = 1$, the process is terminated, and the training immediately continues using the next subnet. The probability distribution p_n as the generalization of a geometric distribution as can be given as:

$$p_k = \lambda_k \prod_{j=1}^{k-1} (1 - \lambda_j)$$

This can be treated as a valid probability distribution if we consider K sufficiently large. While predicting, we simply sample from this distribution to find if the network should continue with the same subset or switch to the next one. If after K time steps, $\frac{1}{N} \sum_{k=1}^{K} \lambda_k > p_T$, we make a switch. Here, p_T is a threshold probability above which we make the switch and below which we do not.

Inorder to train the step function, we use the following loss function:

$$L = \sum_{k=1}^{K} p_k \mathcal{L}(\text{TWE}_{f\,i,i\in[1,N]}) + \beta KL(p_k||p_G(\lambda_p))$$

The first part of the equation is called the Reconstruction loss (L_{rec}), and the second part is called the Regularization loss (L_{reg}). Here $\mathcal{L}(\text{TWE}_{f\,i,i\in[1,N]})$ is the Total Weighted Error (TWE) of the current subnet structure $f_{i,i\in[1,N]}$. The geometric prior distribution $p_G(\lambda_p)$ on the switching policy can be defined using the hyperparameter λ_p. The Regularization loss is the KL divergence between the switching probability p_k and the prior, a geometric distribution parameterized by λ_p. The purpose of regularization loss is to provide a non-zero probability to all the possible steps during the recurrent application of the step function. A similar loss function has been used in PonderNets [2] where the objective was to decide when to stop the training, whereas in our cases, the objective is when to switch the subnet. The reconstruction loss in the case of PonderNets is cross-entropy, whereas we use the TWE of the current subnet structure.

A.2 Loss Plots

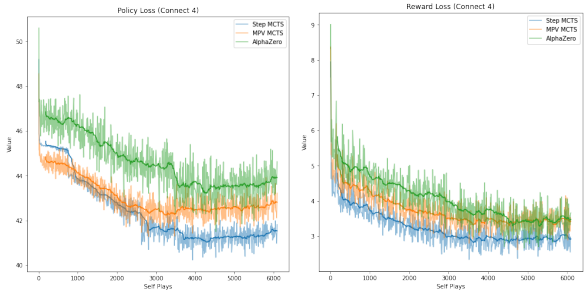

Figure 5: Policy Loss (Left) and Reward Loss (Right) Plots for Connect-4. Note that for Step MCTS (Threshold), the plot shows a drop at around the 8th step, our first switch happened at 7th step.

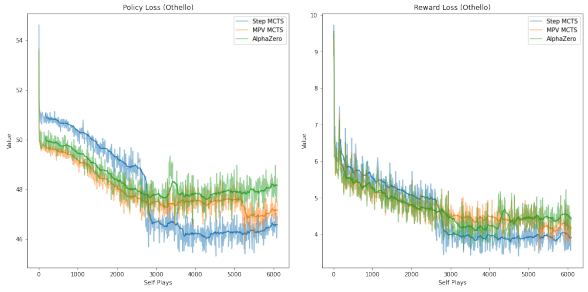

Figure 6: Policy Loss (Left) and Reward Loss (Right) Plots for Othello. Note that for Step MCTS (Threshold), the plot shows a drop at around the 25th step, our second switch happened at 24th step.

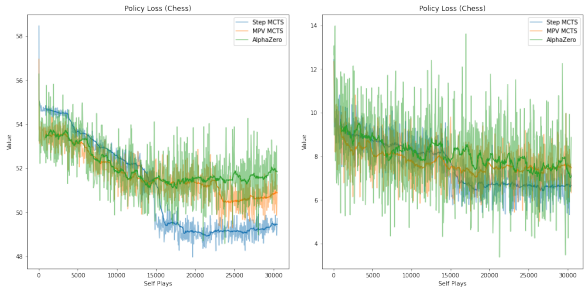

Figure 7: Policy Loss (Left) and Reward Loss (Right) Plots for Chess. Note that for Step MCTS (Threshold), the plot shows a drop at around the 150th step, our third switch happened at 148th step.