Improving Monte Carlo Tree Search with Memory in Go Project Report

Zeyi Wang

February 25, 2021

1 Summary

A previous work introduced Memory-Augmented Monte Carlo Tree Search (MMCTS) and showed incorporating a memory structure in Monte Carlo Tree Search (MCTS) could improve the search performance [1]. MMCTS was built on top of Fuego, a Go program that plays Go using Monte Carlo Tree Search. MMCTS also utilized a deep convolutional neural network (DCNN) to produce features for states [2]. Later on, stronger open-source Go programs such as *KataGo* were made available to researchers [3]. A natural question to ask is that, could MMCTS work on these stronger Go programs that have neural networks baked in?

In this project, we tried to answer that question. More specifically, we implemented a memory structure similar to one in MMCTS in *KataGo*. We called this new program *M-KataGo* and we evaluated its performance against the unmodified *KataGo*. Our experiments were not sufficient to show that *M-KataGo* is statistically better than *KataGo*. Here we report the designs we tried, the way we evaluated our algorithms, and lessons we learned from the experiments.

2 Methods

This project involved heavy experimentation and we did not obtain a "final form" of the algorithm. In the following section we will describe the methods we tried and the parameters we used. The combinations of all the methods and parameters described here define the search space of a class of *M-KataGo* algorithms. Unfortunately, we did not find a candidate algorithm in that space that consistently outperforms the original *KataGo*.

2.1 Feature Selections

We utilized two types of features to measure state similarities. The first one is board ownership map that indicates the predicted board ownership when the game ends. This feature is trained in *KataGo* as an auxiliary learning target and could be extracted from neural network query outputs when a MCTS is performed. The dimension of the feature depends on the board size. We usually use board sizes of 9x9 or 19x19, which corresponds to feature sizes of 81 and 361 respectively. The other feature we used was the representations of the states from the middle layers of the neural network. The size of this feature depends the layer it is extracted from, ranging from 2000 to 15000.

2.2 The Memory

2.3 Memory Instance

The memory is responsible for storing state information that could be later used to improve search statistics. When a new *M-KataGo* instance is created, a memory is created with these following parameters:

- *FeatureDim*, then dimension of the feature
- *MemorySize*, the size of the memory, usually $2000 \sim 20000$
- *NumNeighbors*, the number of nearest neighbors to select for querying, usually $4 \sim 64$
- *Aggregator*, the aggregator to use for aggregating candidates statistics, either `mean` or `softmax`

2.4 Memory Entry

An entry in the memory is a tuple of:

- *ID*, a unique identifier of the state
- *TouchCounter*, a counter to decide the oldest entry in the memory
- *FeatureVector*, a vector of floating point numbers that represents the state's features
- *Value*, the estimated value of the state
- *Visits*, the number of visits of the state

2.5 Memory API

The memory's API supports two types of operations: *Update* and *Query*. (In [1] there's also *Add*, but we treated as a special case of *Update*.)

Update takes an *ID* and update that entry's *FeatureVector*, *Value*, and *Visits*. If an entry with that *ID* is not found, then a new entry is added to the memory with the statistics. The entry being updated will have the largest *TouchCounter*. If the memory is full, then the oldest entry (the one with the smallest TouchCounter) be kicked from the memory.

Query takes a *FeatureVector* and returns *Value* and *Visits* aggregated from best candidates within the memory. First, a linear sweep in the memory is performed. The distances between the input *FeatureVector* and the entries are calculated and the top *NumNeighbors* candidates are selected. Then, based on the *Aggregator*, we either aggregate the statistics of the candidates by averaging them (`mean`) or proportiona to their *Visits* (`softmax`).

We tried to use a library for searching nearest neighbors (such as `Annoy`) to hold the entries. However, we found out that most of these libraries assumed data are relatively static (each update) and this is not the case with *M-KataGo* where thousands of states are inserted on-the-fly. As a result, we implemented our own nearest neighbor component that stores data in a hash-table and uses a linear search to find the nearest neighbors in the table. We found out that in practice a simpler linear search based memory outperforms libraries, and this is even more true when the memory size is small.

2.6 Memory Usage in the MCTS

To use the memory with MCTS, we first define a procedure called *BlendWithMemory*. This procedure takes a search node as an input, queries the memory, and returns a pair of blended statistics with the original node statistics. If the memory is not full yet, which means the search just started, the procedure returns the original statistics without modification. If the state is not in the memory, then the memory is updated with *Update* based on the state and its associated statistics.

We considered these two parameters for this procedure:

- *MemoryLambda*, the amount of blending from (0, 1), but usually within (0.1, 0.7). 0 means the memory is ignored, 1 means the original node is ignored and we only use the memory. Denoted as λ.

- *DiscountFactor*, the amount from (0, 1) that denotes the discounts of blending when the original node have more visits. Usually within (0.5, 0.99). A smaller number means we discount the blending more when the original node has more visits. Denoted as α.

Also, let v denotes the node's value and w denotes the node's weight (usually number of visits of the node). Then, let v_M denotes the value aggregated by the memory and w_M denotes the weight aggregated by the memory. Given a node, we blend the node's statistics with the memory using this formula:

$$\begin{aligned}\beta &= \lambda\alpha^{\frac{\log w * w}{w_M}} \\ v' &= (1-\beta)v + \beta v_M \\ w' &= (1-\beta)w + \beta w_M\end{aligned}$$

This is the general formula and we also tried a few variants of it. For example, the update to w' isn't necessary and only updating v' makes sense too.

There are two main places where we call this procedure in the MCTS. The first one is where the MCTS adds a new leaf node. In this case, we first follow the usual procedure of estimating the state's value by querying the value network. In *KataGo*, this value is directly used as the value of the new node with a visit count of 1. On the other hand, *M-KataGo* updates both the value and the visit count with *BlendWithMemory* then uses the updated statistics for the new node. The other place where we use this procedure is when the MCTS backups and the node statistics are recomputed. In *KataGo*, the new statistics of a node are recomputed based on the statistics of its children and the estimation of the value network. In addition to that, *M-KataGo* further updates those statistics with *BlendWithMemory* and now the recomputed statistics also depend on similar states in the memory. Notice that this is different from the method in [1] where parent nodes do not query the memory. In our experiments, the latency of querying the memory in all nodes was acceptable so we used parent queries in most of the experiments doing so. We also tried only updating the leaf nodes, but did not notice much difference.

3 Evaluations

KataGo was trained through self-playing so it has a built-in function of playing with itself. However, this built-in function does not handle different versions of the source code so we could not use it to compare *KataGo* and *M-KataGo*. As a result, we have use an external program that acts like a server and organizes games between two Go engines. More specifically, *KataGo* supports *Go Text Protocol* (GTP), which means it can take instructions and play games through this interface. We do not modify this part of *KataGo* that communicates through this interface so *M-KataGo* has the same functionalities.

We first used `gogui-twogtp`, a judge program that supports GTP, to orchestrate games between *KataGo* and *M-KataGo*. There were two major limitations to `gogui-twogtp` that made us decided to switch to a different program. The first problem is that it does not handle exceptions well. Depending on the type of the exceptions, `gogui-twogtp` would either suppress the message or log the message to the screen with other regular messages. This made tracing bugs with error messages difficult, especially with a large number of concurrent games where all games logs are dumped on the screen. The other problem is that it cannot be modified easily. `gogui-twogtp` is actually a sub-command of `gogui`, a toolkit that does much more than orchestrating games between two Go programs. In addition, it is written in Java and none of us know it well enough to make surgical changes.

Due to the limitations to `gogui-twogtp`, we switched to a different program called `gomill`. `gomill` is also a judge program that supports GTP, so it communicates with *KataGo* and *M-KataGo* instances the same way as `gogui-twogtp` does. `gomill` overcame the `gogui-twogtp`'s error handling problem by dumping logs of each competitors into separate files and never suppress useful error messages. Moreover, it is written in Python with minimum dependencies, so we could patch it easily when we need some extra functionalities.

There were two changes we made to customize `gomill`. We made the first change to cope with the problem of duplicate games. To reduce the number of duplicated games, we created a pool of games starting rules in which `gomill` will randomly select one for each pair of competitors. These rules included different Go rule sets ("chinese", "japanese", "tromp-taylor", e.t.c.) and different komi settings (5.5, 6.5, 7.5).

Our early experiments were done on local computers and Cirrus Cloud at the University of Alberta. We soon realized the experiments weren't fast enough and switched to using Compute Canada clusters. We evaluated *M-KataGo* by using the judge program mentioned above to play against *KataGo*. To test a specific version of *M-KataGo*, we spawn a job with 4 - 32 CPU cores and 1 GPU that runs 300 - 2000 games. The rule sets were either fixed or randomized by a customized judge program. The board size was one of 9x9, 13x13, 19x19. We also tried adjusting match specific parameters, such as the number of total node visits and the temperature of move selection. We always keep this type of parameters the same for both competitors. Once the games finish, we then calculate the 95% confidence interval for *M-KataGo*'s win rates in these games. Our goal were to find a version of *M-KataGo* for which the lower bound of its win rate confidence interval is above 55%. Unfortunately, the lower bounds we got usually lie between 45% and 50%, meaning either our methods were not in fact better or we did not obtain enough samples to show its statistical significance.

4 Conclusion

In this project we tried to implement MMCTS on top of *KataGo*. We experimented with various methods and parameter settings and evaluated each version of *M-KataGo* carefully. We did not find any version of *M-KataGo* that beats *KataGo* consistently. This means either our methods were not superior or the number of samples were not enough to show statistical significance. We suspect *KataGo* is not compatible with MMCTS for some reason but we couldn't isolate the reason since there was too much noise in the experiments. A possible direction for future study could be implementing MMCTS for a smaller game on top of a simpler MCTS. This could help understanding MMCTS better without involving the complexity of Go and integration with a massive C++ project like *KataGo*.

References

[1] C. Xiao and J. Mei, "Memory-augmented monte carlo tree search," p. 7.

[2] C. Clark and A. Storkey, "Teaching deep convolutional neural networks to play go," p. 9.

[3] D. J. Wu, "Accelerating self-play learning in go," p. 28.