Towards Solving Kissing Number Problem with Reinforcement Learning and MCTS

Kai Yang
School of EECS, Peking University
yangkai@stu.pku.edu.cn

Yi Fang
School of EECS, Peking University
chnfy@stu.pku.edu.cn

Yichong Xia
School of EECS, Peking University
xiayc@stu.pku.edu.cn

Li Lin
School of EECS, Peking University
efsotr_l@stu.pku.edu.cn

Abstract

Reinforcement Learning has found widespread application across various domains. It has been successfully employed in searching and optimization problems, as well as in playing games and optimizing tasks such as matrix multiplication. In this project, we applied reinforcement learning and Monte Carlo Tree Search (MCTS) to address the kissing number problem in specific dimensions after discretization. To enhance our methodology, we meticulously designed the network architecture by incorporating a Transformer encoder alongside learnable special tokens for predicting policy distribution and value. Moreover, we introduced the innovative "looking one step ahead" technique to explore novel state-action pairs during the sample-based MCTS search. Furthermore, we attempted to leverage results obtained in low-dimensional space as starting points, thereby incorporating prior knowledge into the search process. Additionally, to boost efficiency, we utilized randomized data augmentation and parallelism. Through the implementation of these key strategies, we were able to achieve optimal results for the kissing number problem in dimensions $\dim = 3, 4, 5$. In the case of $\dim = 6$, our approach achieved 83.3% of the known optimal results.

1 Introduction

Deep learning has emerged as a powerful and transformative approach in the field of mathematics, revolutionizing problem-solving techniques across various domains. With its ability to learn from large datasets and extract complex patterns, deep learning has proven to be an invaluable tool for tackling intricate mathematical problems that were once considered computationally intractable. For instance, conventional optimization algorithms often rely on heuristics or simplifying assumptions, limiting their performance in solving highly nonlinear and non-convex optimization problems. However, deep learning techniques, such as deep reinforcement learning and evolutionary algorithms, have shown promise in addressing these challenges. Several work has applied deep reinforcement learning framework to solve some combinatorial optimization problems such as vehicle routing problem (VRP) [13]. By harnessing the capabilities of deep learning, we can pave the way for innovative solutions and deeper insights into mathematical problems, ultimately contributing to advancements in science, engineering, and beyond.

The kissing number problem is a fascinating mathematical conundrum that explores the maximum number of spheres that can simultaneously touch another central sphere in n dimensional Euclidean space. Formally, it seeks to determine the highest possible number of non-overlapping, congruent spheres that can be arranged around a central sphere such that each outer sphere is in contact with the

Term project for Machine Learning course in Peking University (2023 spring).

central sphere. Equivalently, we can ask *how many vectors can be arranged in n dimensional space, such that no two distinct vectors form an angle of strictly less than 60 degrees.* The kissing number problem becomes increasingly complex as the dimensionality increases, and while exact solutions are elusive in higher dimensions, significant progress has been made. For example, the lower bounds of the kissing number in $\dim = 10, 11, 14$ have been improved to 510, 592 and 1932 respectively [9].

However, the majority of existing attempts to solve this problem are based on mathematical derivation, with deep learning rarely being applied. In this project, our aim was to leverage machine learning techniques to address the kissing number problem. Initially, we employed traditional reinforcement learning techniques, but they proved to be unsuccessful unfortunately. However, by combining reinforcement learning with Monte Carlo Tree Search, similar to the approaches taken by AlphaGo and Alphatensor [15, 8], and conducting a search in the discretized space, we discovered that the model could achieve some optimal results. Building upon this framework, we meticulously **designed the network architecture based on Transformers**, incorporating learnable special tokens to predict the policy and value. Additionally, we introduced the technique of **"looking one step ahead" during the action selection process**, which yielded more practical and reasonable predicted values.

Although the aforementioned method can solve the problem in some simple cases, it suffers from efficiency issues. To address this concern, we propose a novel approach that **leverages results obtained in low-dimensional space** to achieve better results in higher-dimensional space more efficiently. In our method, given a feasible solution in a d-dimensional space with m vectors, we generate a feasible solution in a d'-dimensional (where $d' \geq d$) space with m vectors as well, serving as the starting point for the search process. This approach allows us to capitalize on the insights gained from lower-dimensional spaces and apply them effectively to higher-dimensional spaces.

To strike a balance between learning efficiency and memory consumption, we employ **randomized data augmentation** during the data collection phase. Rather than applying all possible permutations, we randomly select several permutations for each data pair. This strategy ensures a diverse range of training data while mitigating the computational burden. Furthermore, we incorporate **parallelism** during the search process to enhance efficiency. Since each simulation and search process operates independently and does not share the search tree, parallelization can be applied seamlessly, yielding significant efficiency improvements with minimal side-effects. By combining reinforcement learning with Monte Carlo Tree Search and integrating these optimization methods, our model outperforms all naive baselines significantly. Specifically, our model achieves known optimal results in dimensions $\dim = 3, 4, 5$, and attains 83.3% of the known optimal results in the case of $\dim = 6$.

The report is organized as follows. In Section 2, we introduce background on the Monte Carlo Tree Search, reinforcement learning and Transformer architecture. In Section 3, we present the naive approach using reinforcement learning directly, which totally failed in kissing number problem. In Section 4, we present the architecture that combines reinforcement learning and MCTS, together with "looking one step ahead" technique, utilizing results in low-dimensional space and other useful optimization. Experiments are presented in Section 5 to demonstrate the effectiveness of our RL+MCTS model. Related works are listed in Section 6 and several future directions are discussed in Section 7.

2 Preliminaries

2.1 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) [4, 2] is a decision-making algorithm commonly used in artificial intelligence, particularly in the field of game playing. It is a simulation-based search algorithm that efficiently explores the search space by employing random sampling and statistical analysis. MCTS has achieved remarkable success in various domains, including board games like Go [10], chess, and poker, as well as video games and other combinatorial optimization problems.

The basic idea behind MCTS is to build a tree representation of possible game states and make decisions by sampling and evaluating these states through repeated iterations. It consists of four main steps: selection, expansion, simulation, and backpropagation.

- Selection: Starting from the root node of the tree, the algorithm navigates down the tree by selecting child nodes based on some selection policy. Usually, it employs an upper

confidence bound formula, such as the Upper Confidence Bound for Trees (UCT), to balance exploration and exploitation.

- Expansion: Once a leaf node is reached (a node without children), the algorithm expands the tree by adding one or more child nodes, representing the possible actions from the current state.
- Simulation: The algorithm performs a playout or simulation from the newly added node, usually by randomly selecting moves until a terminal state is reached. The playouts are often quick and do not involve extensive computation.
- Backpropagation: The result of the simulation is backpropagated up the tree, updating the statistics of each node visited during the selection phase. This includes updating the number of visits and the accumulated rewards for each node.

By repeating these steps for a certain number of iterations or until a time limit is reached, MCTS gradually explores the search space and converges towards a good solution. The final decision is typically based on the statistics accumulated during the search, such as selecting the most visited child node or the action with the highest expected reward.

2.2 Reinforcement Learning

Reinforcement learning (RL) is a subfield of machine learning that focuses on training agents to make sequential decisions in an environment to maximize a notion of cumulative reward. Reinforcement learning provides a framework for developing intelligent systems that can learn and adapt through interaction with their environment. It has been successfully applied to various domains, including robotics, game playing, recommendation systems, and autonomous driving. In reinforcement learning, an agent interacts with an environment in a series of discrete time steps. At each time step, the agent observes the current state of the environment, takes an action, and receives feedback in the form of a reward and a new state. The goal of the agent is to learn a policy, which is a mapping from states to actions, that maximizes the long-term expected cumulative reward.

2.3 Transformer

The Transformer architecture is composed of stacked Transformer blocks [17, 5]. A Transformer block is a sequence-to-sequence mapping from $\mathbb{R}^{n\times d}$ to $\mathbb{R}^{n\times d}$, where n is the sequence length and d is the dimension of token embedding. A Transformer block consists of two layers: a self-attention layer followed by a feed-forward layer, with both layers having normalization (e.g., LayerNorm [1], RMSNorm [18]) and skip connections. For an input $\boldsymbol{X} \in \mathbb{R}^{n\times d}$, the self-attention layer and feed-forward layer are defined as follows:

$$\boldsymbol{A}^h(\boldsymbol{X}) = \mathrm{softmax}\left(\frac{\boldsymbol{X}\boldsymbol{W}_Q^h(\boldsymbol{X}\boldsymbol{W}_K^h)^\top}{\sqrt{d_H}}\right); \tag{1}$$

$$\mathrm{Attn}(\boldsymbol{X}) = \boldsymbol{X} + \sum_{h=1}^{H} \boldsymbol{A}^h(\boldsymbol{X})\boldsymbol{X}\boldsymbol{W}_V^h\boldsymbol{W}_O^h; \tag{2}$$

$$\mathrm{FFN}(\boldsymbol{X}) = \boldsymbol{X} + \mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W}_1)\boldsymbol{W}_2, \tag{3}$$

where $\boldsymbol{W}_O^h \in \mathbb{R}^{d_H\times d}$, $\boldsymbol{W}_Q^h, \boldsymbol{W}_K^h, \boldsymbol{W}_V^h \in \mathbb{R}^{d\times d_H}$, $\boldsymbol{W}_1 \in \mathbb{R}^{d\times r}, \boldsymbol{W}_2 \in \mathbb{R}^{r\times d}$. H is the number of attention heads, d_H is the dimension of each head, and r is the dimension of the hidden layer. $\boldsymbol{A}^h(\boldsymbol{X})$ is usually referred to as the *attention matrix*.

3 Naive Approach

In this section, we present our initial attempts, which ultimately failed to solve this specific problem. We aimed to model the kissing number problem as a one-player game and employ traditional reinforcement learning methods to learn an effective strategy. Each instance of the game starts with an empty vector list as the initial state. In each round, the player could choose a new vector of any direction. The environment adds the vector to the list only if it forms an angle of at least 60 degrees

with all the existing vectors in the list. The game ends after several rounds. Under these assumptions, the vector list always represents a feasible solution to the kissing number problem.

Our objective was to find a strategy for the player that maximized the size of the vector list at the end of the game. This objective lent itself naturally to a reinforcement learning problem formulation. The only requirement was to design an appropriate reward function for the environment. With that in place, we could utilize traditional reinforcement learning algorithms such as DDPG [12] and PPO [14].

Designing a suitable reward function is proved to be a technical challenge. Initially, we considered a direct reward function setting, where:

$$R(s,a) = \begin{cases} 1, & \text{if the vector chosen by player is added to the vector list} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

Furthermore, we attempted to enhance the reward function by incorporating the minimum angle formed between the new vector and the vectors in the vector list. Intuitively, we wanted to assign a larger reward if the new vector could be added to the vector list while maintaining a small minimum angle. Formally, our reward function was designed as follows:

$$R(s,a) = \begin{cases} 1 + f(\cos\theta), & 60^\circ \leq \theta \leq 180^\circ \\ 0, & 0^\circ \leq \theta < 60^\circ \end{cases} \tag{5}$$

Here, θ represents the minimum angle formed by the new vector and the vectors in the vector list, and $f(\cdot)$ is a monotonically increasing function. In our implementation, we explored various functions $f(\cdot)$, which differed in terms of bias, concavity, convexity, and scaling.

Nevertheless, even after extensive experimentation with various reinforcement learning algorithms and iterations of the reward function, our attempts failed to produce satisfactory results. The disappointing outcomes of this approach are clearly illustrated in Table 1, which reveals that the performance of our naive method is even inferior to that of random baselines.

4 Reinforcement Learning with MCTS

In this section, we present our final approach, which combines reinforcement learning with Monte Carlo tree search (MCTS). This method has achieved the best-known results in cases where $\dim = 3, 4, 5$. Here we present the implementation details of our approach.

4.1 Kissing Number Game

Although we can arrange vectors in $\mathbb{R}^d$ when considering the d-dimensional kissing number problem, which exists in a continuous space, we **discretize the problem** for our purposes. To achieve this, we limit each component of the d vectors to the set $\{x \mid x \in \mathbb{Z}, |x| \leq b\}$, where b is a constant. In other words, all the vectors are constrained to the subspace $\Omega_b^d = \{x \mid x \in \mathbb{Z}, |x| \leq b\}^d$. For this project, we have set the constant b to be either 1 or 2.

The Kissing Number Game is played as follows. The state of the d-dimensional kissing number game is represented by an $n \times d$ matrix, where n is greater than the known upper bound of the kissing number problem in d dimensions. Each row of the matrix corresponds to a vector, except for the zero vector, which indicates an empty row without a vector. The initial state of the game is a zero matrix, signifying that there are no vectors present. In each step t of the game, the player must select a vector from the space Ω_b^d, ensuring that it satisfies the constraints of the kissing number problem. The new state S_{t+1} is obtained by replacing the t-th row of S_t with the chosen vector. The game ends when there are no feasible vectors remaining in Ω_b^d, and the player's reward is determined by the number of vectors chosen.

4.2 Network Architecture

The deep neural network, denoted as $f_\theta(s) = (\pi, v)$ and parameterized by θ, is designed to process the current state s of the game and produce a probability distribution $\pi(\cdot \mid s)$ over actions and $v(\cdot \mid s)$ over returns, which represent the sum of future rewards. To achieve this, we employ Transformer

models for predicting π and v. This choice is motivated by the varying number of vectors in the inputs, while the matrix shape remains consistent. We treat the input as a variable-length sequence and utilize zero vectors as padding.

Our network architecture, as depicted in Figure 1, treats the state matrix as a sequence of vectors. To facilitate the prediction of policy distribution and value, we introduce special tokens known as the π-token and the v-token. This design choice mimics the `[CLS]` token in BERT [6] and Vision Transformer [7]. These tokens are added after the input has been projected into the embedding space. Once the tokens are incorporated, the input undergoes further processing through the Transformer encoder layers. Following the Transformer encoding process, two separate linear layers are employed to derive the output for the policy and value. During the training process, we update the value of the special tokens as well.

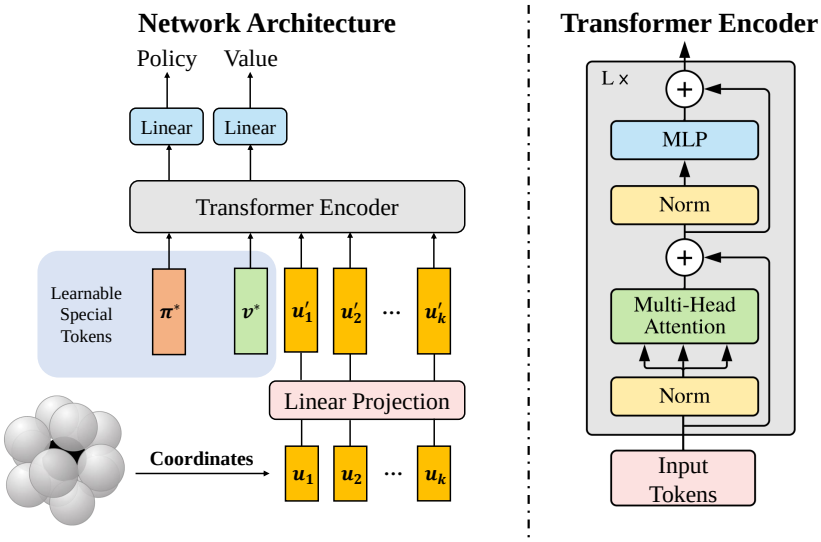

Figure 1: Network Architecture

Regarding the learning rate configuration, we utilize a warm-up process during the initial 10% of iterations, during which the learning rate linearly increases. After this warm-up phase, for the remaining 90% of iterations, the learning rate linearly decreases. This approach allows for a gradual adjustment of the learning rate during the early stages, followed by a more controlled descent in the later stages of training.

4.3 Sample-based MCTS Search

The sample-based Monte Carlo Tree Search (MCTS) utilized in this project closely resembles the approach described in the Alphatensor paper [8]. Like in the Alphatensor paper, the search tree comprises nodes that represent states and edges that represent actions. Each state-action pair (s, a) retains a collection of statistics, including the visit count $N(s, a)$, the action value $Q(s, a)$, and the empirical policy probability $\hat{\pi}(s, a)$.

Throughout the search process, the choice of each action is determined by maximizing the upper confidence bound (UCB) formula:

$$Q(s,a) + \hat{\pi}(s,a) \cdot \frac{\sqrt{\sum_i N(s,a_i)}}{1 + N(s,a)} \cdot \left[c_1 + \log\left(\frac{1 + c_2 + \sum_i N(s,a_i)}{c_2} \right) \right] \tag{6}$$

The exploration factor c_1, c_2 accounts for the influence of the empirical policy $\hat{\pi}(s, a)$ in relation to the action values $Q(s, a)$ as the nodes are visited more frequently. Notably, the second term of the UCB expression decreases as a node becomes more frequently visited. In our implementation, we have chosen $c_1 = 1.25$ and $c_2 = 10000$ to control the exploration factor.

In practical terms, if a state-action pair (s, a) has not been explored when calculating the UCB, we propose to **look one step ahead** for that particular pair and obtaining the value predicted by the

neural network. This adjustment results in $N(s, a)$ being set to 1, and $Q(s, a)$ simply represents the value predicted by the neural network. Compared to the implementation that sets $N(s, a) = 0$ and $Q(s, a) = 0$, this modified approach empirically provides a more accurate estimation of the UCB formula and performs better during training. Our experiments presented in Section 5.2 validated this intuition, and we observed a significant improvement in the training effect after applying this method.

4.4 Policy Improvement

After completing all the simulations, when the MCTS provides a promising glimpse into the near future, it becomes necessary to update the policy associated with the predicted nodes and return them for training purposes. In our project, we compute the improved policy as follows:

$$I\pi(s, a) = \frac{N^{1/temp}(s, a)}{\sum_i N^{1/temp}(s, a_i)} \tag{7}$$

Here, $temp$ is determined based on the round t. If $t < \mathrm{tempThreshold}$, then $temp$ is set to 1; otherwise, it is set to 0. When t is 0, we randomly select an action a from those that have been visited most frequently, and set the indicator variable $I\pi(s, a)$ to 1 for that action. For the remaining actions, we set $I\pi(s, a)$ to 0. In our implementation, we have chosen the value of $\mathrm{tempThreshold}$ as 15, which determines the point at which the temperature parameter switches from 1 to 0, indicating a transition in the exploration strategy.

During training, we directly utilize $I\pi$ as the action probability distribution and also as the target for the network policy π. This means that $I\pi$ serves as both the input distribution for selecting actions and the reference distribution for training the network policy.

4.5 Using Results in Low Dimension

It is widely recognized that MCTS can be time-consuming, especially when applied to high-dimensional spaces or larger action spaces. To mitigate this challenge, we have explored a strategy that **utilizes results obtained in low-dimensional space as starting points** for searching in high-dimensional spaces.

To generate an initial vectors putting matrix for the high-dimensional space, we leverage the results obtained in low-dimensional spaces. Suppose we have a feasible solution with m vectors in a d-dimensional space. We can create a feasible solution in a d'-dimensional space where $d' \geq d$, by simply considering the first d components of each vector as the solution in the d-dimensional space. This approach enables us to construct a matrix with m given vectors when attempting to solve the kissing number problem in the d'-dimensional space. In the context of the Kissing Number Game, this generated matrix serves as the initial state, replacing the empty matrix discussed in Section 4.1. By using the transformed matrix as the starting point, we can explore and optimize the solutions within the higher-dimensional space.

This approach has proven to significantly accelerate the searching process. In our experiments presented in Section 5.3, when we utilize the optimal result in $\dim = 4$ as a starting point for solving the kissing number problem in $\dim = 5$, focusing our search in the subspace Ω_2^5, it takes approximately 4 hours to identify the known optimal result. In contrast, if we attempt to solve the kissing number problem in $\dim = 5$ from scratch, searching in the subspace Ω_1^5, it takes approximately 3 days to converge to the known optimal result.

However, it is important to note that this approach does have some side-effects. In certain cases, it may inadvertently impede the search process, potentially leading to the discovery of sub-optimal results instead of the global optimum.

4.6 Parallelism and Data Augumentation for Efficiency

In practice, we leverage parallelism to enhance the efficiency of the MCTS simulation process. Since the MCTS search trees are independent across different simulation episodes, we can perform them simultaneously, taking advantage of parallel computing to expedite the overall process. Additionally, we have incorporated data augmentation techniques to augment the amount of data available for training. For each data instance, we **randomly select multiple rotations** and apply them to the data, generating additional data points. By applying data augmentation, we are able to diversify and

expand our dataset. This, in turn, facilitates more effective training of the neural network, improving its performance and generalization capabilities.

5 Experiments

In this section, we demonstrate the effectiveness of our RL+MCTS model (Section 4) by comparing with two naive baselines, naive RL (Section 3) and known results. We begin with detailing our model's training details and introducing baselines. Then, we present the full evaluation results for each setting, and we conclude with qualititave analysis, delving into the effectiveness of our final model.

RL+MCTS Training. During the simulation process, we simultaneously run a total of 8 episodes in each iteration. Once all episodes are completed, the data collected from the simulation process, along with the augmentation method, is utilized to train the neural network. To aid in the training of the Transformer encoder layer, we incorporate a warm-up process. In addition, we conduct an ablation study to evaluate the impact of the "looking one step ahead" technique mentioned in Section 4.3. Furthermore, we compare the performance of the model with and without incorporating results in low dimension, as explained in Section 4.5. More detailed information is provided in Appendix B.1.

Naive RL Training. We configure the game to have a total of 200 rounds and employ the PPO [14] algorithm to train naive reinforcement learning model. Both the actor and critic networks utilize a Transformer encoder layer, similar to the one described in Section 4.2. In the reward function mentioned in (5), we use $f(\cos\theta) = \exp(2(\cos\theta + 1))$. More detailed information is provided in Appendix B.2.

Baselines. We compare our approach to two naive baselines, each with its own running process described below:

- **Continuous Baseline:** In this baseline, we select a unit vector uniformly at random. We add the vector to the vector list only if it forms an angle of at least 60 degrees with all the existing vectors in the list. We repeat this process $1\mathrm{e}6$ times and then terminate.
- **Discretized Baseline:** In this baseline, we randomly select a vector from the set Ω_2^d (which represents all possible vectors in a discrete domain) in such a way that the chosen vector forms an angle of at least 60 degrees with all the existing vectors in the list. We continue this process until there are no feasible vectors left to choose from.

5.1 Results Found by Different Models

Table 1 presents the best results obtained by different models for the kissing number problem in $\dim = 3, 4, 5, 6$. In the case of $\dim = 5, 6$, the RL+MCTS+Knowledge approach utilizes the optimal results from the previous dimension to find the results in the current dimension. Both of our models presented here employ the "looking one step ahead" technique.

Table 1: Results found by different models

Model	dim = 3	dim = 4	dim = 5	dim = 6
Continuous Baseline	9	14	22	30
Discretized Baseline	8	12	17	26
Naive RL	8	11	15	17
RL+MCTS (Ours)	**12**	**24**	**40**	52
RL+MCTS+Knowledge (Ours)	-	-	**40**	**60**
Best Lower Bound	12	24	40	72
Best Upper Bound	12	24	44	78

From the table, we can observe that the naive reinforcement learning approach performs even worse than the naive baselines. It is noteworthy that the best results achieved by the naive reinforcement learning approach often occur during the first few training iterations, and the average total reward

tends to decrease as the training progresses. These observations suggest that the model easily produces worse results after more training iterations.

On the other hand, our approaches utilizing reinforcement learning and MCTS reach the known lower bound in $\dim = 3, 4, 5$. Additionally, when incorporating prior knowledge of optimal results in lower dimensions, the RL+MCTS+Knowledge approach achieves better results with fewer computational requirements. The detailed analysis of this advantage will be presented in Section 5.3.

5.2 Ablation Study for Looking One Step Ahead

In this section, we present a comparison of results between RL+MCTS with and without the "looking one step ahead" technique, aiming to demonstrate the effectiveness of this approach. We followed the training details described earlier and conducted 10 iterations for each case. The results of this comparison are presented in Table 2. Based on the table, we can conclude that incorporating the "looking one step ahead" technique significantly enhances performance and allows for easily achieving the best results in these cases. Furthermore, we observe that the results obtained by RL+MCTS without the technique do not improve even after training, indicating that the way it evaluates the UCB formula (6) is problematic.

Table 2: Results of RL+MCTS with/without Looking One Step Ahead

Model	dim = 3	dim = 4
RL+MCTS (with Looking One Step Ahead)	**12**	**24**
RL+MCTS (without Looking One Step Ahead)	10	12

5.3 Running Time Analysis

In this section, we present a comparison of running time between the RL+MCTS and RL+MCTS+Knowledge approaches when searching for optimal results in the case $\dim = 5$ for kissing number problem. The running time corresponding to the results shown in Table 1 is listed in Table 3.

Table 3: Running time of different models (dim = 5)

Model	Search Space	Running Time
RL+MCTS (Ours)	Ω_1^5	$\sim$ 3 days
RL+MCTS+Knowledge (Ours)	Ω_2^5	$\sim$ **4 hours**

From the table, we can conclude that even though the RL+MCTS+Knowledge approach utilizes a larger search space (Ω_2^5) compared to the RL+MCTS approach (Ω_1^5), the former requires less running time while obtaining better results. This observation highlights the significance of incorporating prior knowledge, such as the best results from lower-dimensional cases, in improving performance both in terms of results and time consumption.

These findings serve as inspiration for future attempts to solve the kissing number problem using a similar architecture. Introducing relevant prior knowledge, such as the optimal results in lower-dimensional cases, can significantly enhance the performance of the model in terms of both result quality and time efficiency.

6 Related Work

AlphaZero. AlphaZero [16] is a computer program developed by DeepMind to master the games of chess, shogi, and go. This algorithm uses an approach similar to AlphaGo Zero. AlphaZero is a more generalized variant of the AlphaGo Zero (AGZ) algorithm and is able to play shogi and chess as well as Go. Differences between AlphaZero (AZ) and AGZ include: AZ has hard-coded rules for setting search hyperparameters and the neural network is now updated continually. AlphaZero is trained by self-play reinforcement learning. It combines a neural network and Monte Carlo Tree Search in an elegant policy iteration framework to achieve stable learning.

AlphaTensor. AlphaTensor [8] is an artificial intelligence system developed by DeepMind that discovers novel, efficient, and provably correct algorithms for fundamental tasks such as matrix multiplication. It uses deep reinforcement learning to discover optimal algorithms for multiplication of arbitrary matrices. The algorithm discovery process for matrix multiplication can be formalized as low-rank decompositions of a specific three-dimensional (3D) tensor, called the matrix multiplication tensor. Researchers formulate the matrix multiplication algorithm discovery procedure (that is, the tensor decomposition problem) as a single-player game, called TensorGame. To solve TensorGame, researchers develop a DRL agent, AlphaTensor, which is based on AlphaZero. In 2021, AlphaTensor discovered a formula with 76 operations that is faster than the known methods for multiplying matrices. The technique of machine learning improves computational efficiency and could have far-reaching applications.

7 Discussions and Future Directions

In this project, we have employed the fundamental framework of AlphaGo and Alphatensor [15, 8] as a basis for designing a search process to tackle the kissing number problem. Our approach leverages reinforcement learning and sample-based Monte Carlo Tree Search (MCTS), while incorporating methods such as discretization, parallelism, and data augmentation tricks. To construct the network architecture, we utilize a Transformer encoder and introduce learnable special tokens. Building upon this framework, we have further optimized our approach to better suit the kissing number problem. Two key enhancements have been introduced. Firstly, we propose to look one step ahead during the searching process, enabling us to get more reasonable value predictions. Additionally, we have devised a straightforward technique to utilize results obtained in low-dimensional space. Both of these improvements have resulted in significant performance enhancements. Through a series of experiments, we have demonstrated the superiority of our approach over naive baselines. Notably, we have achieved optimal known results in dimensions $\dim = 3, 4, 5$.

There are several additional approaches that warrant consideration and application in our project. These approaches can further leverage the capabilities of reinforcement learning and Monte Carlo Tree Search (MCTS). Here are the proposed directions for future exploration:

- **Expanding the Search Space:** In order to discover more refined results, it is necessary to broaden our search space. Although we have achieved known optimal results in dimensions $\dim = 3, 4, 5$, it may be crucial to explore larger spaces to uncover non-trivial findings. Expanding the search space will likely require additional computational resources and innovative strategies.
- **Code Optimization:** To efficiently search for optimal results in larger spaces, it is essential to rewrite parts of the code to enhance its efficiency. By optimizing the code, we can ensure that computational resources are utilized effectively and that the search process can be scaled up accordingly.
- **Analysis of Upper Bounds:** Conducting an analysis of the upper bounds within the search space can provide valuable insights. Understanding the upper bounds can guide us in designing the search space more effectively, enabling us to focus on promising areas and potentially improve the search efficiency.
- **Equivariant Networks:** Considering the ideal symmetry in the input and output of the network, it may be beneficial to explore the application of equivariant networks [3]. By incorporating these networks, we can leverage their ability to handle symmetry and exploit the inherent structure of the problem more effectively.
- **Intelligent Utilization of Low-Dimensional Results:** Careful consideration should be given to the use of results obtained from lower dimensions. Improper utilization may lead to the degeneration of the search space, resulting in sub-optimal results. Exploring smarter ways to integrate information from low-dimensional spaces can help prevent such pitfalls.

These proposed approaches are left as future directions, as they have the potential to extract more value from reinforcement learning and MCTS techniques. By exploring these avenues, we aim to enhance our understanding of the problem and further improve the performance of our approach.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

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

[3] Taco S. Cohen and Max Welling. Group equivariant convolutional networks, 2016.

[4] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers, editors, *Computers and Games*, pages 72–83, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[7] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[8] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nat.*, 610(7930):47–53, 2022.

[9] Mikhail Ganzhinov. Highly symmetric lines, 2022.

[10] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175:1856–1875, 2011.

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

[12] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[13] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence V. Snyder, and Martin Takáč. Reinforcement learning for solving the vehicle routing problem, 2018.

[14] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[15] David Silver and Demis Hassabis Google DeepMind. Alphago: Mastering the ancient game of go with machine learning, 2016.

[16] 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. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 6000–6010, 2017.

[18] Biao Zhang and Rico Sennrich. Root mean square layer normalization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

A Solutions Found

In this section, we present the optimal solutions discovered by our model. We represent a feasible solution as a set of vectors, ensuring that any two distinct vectors form an angle of no less than 60 degrees.

A.1 Kissing Number in 3 Dimension

We found the placement of 12 vectors for kissing number in 3 dimension, which is the optimal result. The result is displayed below.

$$(-1,0,1),(-1,1,0),(0,1,1),(0,1,-1),(1,0,1),(1,1,0),$$
$$(0,-1,1),(1,0,-1),(1,-1,0),(-1,-1,0),(0,-1,-1),(-1,0,-1)$$

A.2 Kissing Number in 4 Dimension

We found the placement of 24 vectors for kissing number in 4 dimension, which is the optimal result. The result is displayed below.

$$(0,1,0,0),(-1,-1,-1,-1),(-1,-1,-1,1),(-1,-1,1,-1),(-1,-1,1,1),(-1,0,0,0),$$
$$(-1,1,-1,-1),(-1,1,-1,1),(-1,1,1,-1),(-1,1,1,1),(0,-1,0,0),(0,0,-1,0),$$
$$(0,0,0,-1),(0,0,0,1),(0,0,1,0),(1,-1,-1,-1),(1,-1,-1,1),(1,-1,1,-1),$$
$$(1,-1,1,1),(1,0,0,0),(1,1,-1,-1),(1,1,-1,1),(1,1,1,-1),(1,1,1,1)$$

A.3 Kissing Number in 5 Dimension

We found the placement of 40 vectors for kissing number in 5 dimension, which is the best result so far. The result is displayed below.

$$(1,1,0,0,0),(1,-1,0,0,0),(-1,1,0,0,0),(-1,-1,0,0,0),(1,0,1,0,0),$$
$$(1,0,-1,0,0),(-1,0,1,0,0),(-1,0,-1,0,0),(1,0,0,1,0),(1,0,0,-1,0),$$
$$(-1,0,0,1,0),(-1,0,0,-1,0),(0,1,1,0,0),(0,1,-1,0,0),(0,-1,1,0,0),$$
$$(0,-1,-1,0,0),(0,1,0,1,0),(0,1,0,-1,0),(0,-1,0,1,0),(0,-1,0,-1,0),$$
$$(0,0,1,1,0),(0,0,1,-1,0),(0,0,-1,1,0),(0,0,-1,-1,0),(-2,0,0,0,-2),$$
$$(-2,0,0,0,2),(0,-2,0,0,-2),(0,-2,0,0,2),(0,0,-2,0,-2),(0,0,-2,0,2),$$
$$(0,0,0,-2,-2),(0,0,0,-2,2),(0,0,0,1,-1),(0,0,0,1,1),(0,0,1,0,-1),$$
$$(0,0,1,0,1),(0,1,0,0,-1),(0,1,0,0,1),(1,0,0,0,-1),(1,0,0,0,1)$$

A.4 Kissing Number in 6 Dimension

We found the placement of 60 vectors for kissing number in 6 dimension, while the best result so far is 72. The result is displayed below.

$$(1,1,0,0,0,0),(1,-1,0,0,0,0),(-1,1,0,0,0,0),(-1,-1,0,0,0,0),(1,0,1,0,0,0),$$
$$(1,0,-1,0,0,0),(-1,0,1,0,0,0),(-1,0,-1,0,0,0),(1,0,0,1,0,0),(1,0,0,-1,0,0),$$
$$(-1,0,0,1,0,0),(-1,0,0,-1,0,0),(1,0,0,0,1,0),(1,0,0,0,-1,0),(-1,0,0,0,1,0),$$
$$(-1,0,0,0,-1,0),(0,1,1,0,0,0),(0,1,-1,0,0,0),(0,-1,1,0,0,0),(0,-1,-1,0,0,0),$$
$$(0,1,0,1,0,0),(0,1,0,-1,0,0),(0,-1,0,1,0,0),(0,-1,0,-1,0,0),(0,1,0,0,1,0),$$
$$(0,1,0,0,-1,0),(0,-1,0,0,1,0),(0,-1,0,0,-1,0),(0,0,1,1,0,0),(0,0,1,-1,0,0),$$
$$(0,0,-1,1,0,0),(0,0,-1,-1,0,0),(0,0,1,0,1,0),(0,0,1,0,-1,0),(0,0,-1,0,1,0),$$
$$(0,0,-1,0,-1,0),(0,0,0,1,1,0),(0,0,0,1,-1,0),(0,0,0,-1,1,0),(0,0,0,-1,-1,0),$$
$$(-2,0,0,0,0,-2),(-2,0,0,0,0,2),(0,-2,0,0,0,-2),(0,-2,0,0,0,2),(0,0,-2,0,0,-2),$$
$$(0,0,-2,0,0,2),(0,0,0,-2,0,-2),(0,0,0,-2,0,2),(0,0,0,0,-2,-2),(0,0,0,0,-2,2),$$
$$(0,0,0,0,1,-1),(0,0,0,0,1,1),(0,0,0,1,0,-1),(0,0,0,1,0,1),(0,0,1,0,0,-1),$$
$$(0,0,1,0,0,1),(0,1,0,0,0,-1),(0,1,0,0,0,1),(1,0,0,0,0,-1),(1,0,0,0,0,1)$$

B Training Details

B.1 Training Details for RL+MCTS

For the MCTS and data collection process, we perform 8 parallel searching episodes in each iteration. During the search, we simulate 30 times to obtain the action probability for a given state. Regarding data augmentation, we randomly select 10 permutations for the given data pair and apply these permutations for augmentation. In the UCB formula mentioned in (6), we fix c_1 as 1.25 and c_2 as 10000 to regulate the exploration factor. In the policy improvement process described in (7), we set $\mathrm{tempThreshold}$ to 15.

For Transformer architecture, the number of layers and the number of attention heads are set to 2 and 4, respectively. The hidden dimension is set to 128. We use Adam [11] as the optimizer, and set its hyperparameters ϵ to $1\mathrm{e}-8$ and (β_1, β_2) to $(0.9, 0.999)$. The peak learning rate is set to $1\mathrm{e}-3$ with a warm-up stage. After the warm-up stage, the learning rate decays linearly to zero. The batch size is set to 64. We set the dropout probability and weight decay to 0.0. All models are trained on a single GeForce GTX TITAN X.

B.2 Training Details for Naive RL

We use Transformer encoder for both actor and critic network. The number of layers and the number of attention heads are set to 2 and 8, respectively. The hidden dimension is set to 128. We use Adam [11] as the optimizer, and set its hyperparameters ϵ to $1\mathrm{e}-8$ and (β_1, β_2) to $(0.9, 0.999)$. The learning rate is set to $1\mathrm{e}-4$. The model is trained for 50K steps in total with the batch size as 2048. We set the dropout probability and weight decay to 0.0. All models are trained on a single GeForce GTX TITAN X.