Multi Agent Reinforcement Learning

Anna Dominic	Ridhika Agrawal	Saahil Jain
amd9200	rra10001	sbj7913

Siri Desiraju	Xingyu Wang	Yu Wang
scd4156	xw2628	yw6309

1 Problem Setup

The problem we aimed to solve in this project was to train a group of cops to catch a criminal by using reinforcement learning. This is a challenging task, as the criminal is programmed to dodge the cops, requiring the agents to work together and develop effective strategies to outmaneuver the criminal. We tested our approach in four different scenarios, involving teams of two and three cops, with shared or independent Q tables. We used Python and the Pygame library to implement our solution. The episodes ended either when the criminal was caught or after 200 steps had been taken.

One of the main reasons we chose this project was to gain insights into the effectiveness of multi-agent cooperation in reinforcement learning. While reinforcement learning is an active area of research, there are still many challenges associated with training agents to work together effectively in complex scenarios. By testing different approaches and analyzing the results, we hoped to contribute to the development of new techniques for multi-agent reinforcement learning.

2 Methodology

To train the cops using reinforcement learning, we used a Q table to make predictions about the optimal action to take in each state. The Q table was updated using the Q-learning algorithm, which is a common approach in reinforcement learning. The discrete states/observations that the agents saw were represented as rows of the Q table, while the various actions they could take (UP, DOWN, LEFT or RIGHT) were represented as columns. To reduce the size of the Q table, we provided partial visibility of the environment to the agents, including the relative positions of the criminal and other agents. We also limited the sight of the agents to reduce the grid space.
The Q-learning algorithm updates the expected reward for taking an action in a particular state by considering the maximum expected reward for the next state. The algorithm is given as follows:

Algorithm 1: Q-learning algorithm

```
Input: Environment
Output: Optimal policy
Initialize Q-table with random values;
for each episode do
    Initialize current state;
    while episode not finished do
        Choose action using epsilon-greedy policy;
        Take action and observe next state and reward;
        Update Q-value using the Bellman equation;
        Update current state;
    end
    Decay epsilon;
end
```

3 Background

Reinforcement learning is an exciting and rapidly developing field of research that has the potential to revolutionize many areas of application, including robotics and autonomous systems. By training agents to learn from their interactions with their environment, reinforcement learning provides a powerful way to enable intelligent decision-making in complex and dynamic scenarios. Multi-agent reinforcement learning, in particular, has gained increasing attention in recent years due to its potential applications in domains such as traffic control, robotics, and games. The key challenge in multi-agent reinforcement learning is to enable agents to learn to cooperate effectively to achieve a common goal while dealing with non-stationary environments and other agents' actions. In this project, we aimed to explore the potential of reinforcement learning for multi-agent cooperation in a challenging scenario involving a group of cops trying to catch a criminal. Our approach could have potential applications in a variety of domains, including law enforcement, military operations, and autonomous systems. Overall, our project contributes to the ongoing effort to develop new and effective techniques for multi-agent reinforcement learning in complex environments.

4 Initial Base Code

This code is an implementation of a Q-learning algorithm that allows a "cop" to catch a "thief" in a 2D grid world. The implementation uses various Python libraries, including numpy, PIL, cv2, matplotlib, pickle, and time. The libraries numpy, PIL, and cv2 are used to process and manipulate images in the program. Matplotlib is used to display visualizations of the simulation, and pickle is used to save and load the Q-table to/from a file. The time library is used to time the program's execution.

The program starts by initializing various parameters such as the size of the grid, the number of episodes, and the rewards and penalties associated with certain actions. The epsilon parameter is used to determine how much randomness should be introduced to the agent's decision-making process.

The program defines two classes: cop_class and thief_class. These classes define the behavior of the cop and the thief, respectively. The cop and the thief are randomly placed on the grid at the beginning of each episode. The relative_position function calculates the relative position of one object with respect to another object. The action function determines the movement direction of the cop based on its current state. The perform_action function updates the cop's position based on its chosen action. Finally, the update_table function updates the Q-table based on the cop's new position and the reward received for reaching that position.

The program also defines a start_q_table variable, which can be set to either None or a filename. If it is None, the Q-table is initialized to random values. If it is a filename, the Q-table is loaded from the file. Finally, the main loop of the program runs through a specified number of episodes, with the cop trying to catch the thief in each episode. During each episode, the cop updates its position based on the Q-table until it catches the thief or until the episode ends. After each episode, the epsilon value is decayed to reduce the amount of randomness in the cop's decision-making process over time. This initial code took 2.04s to run.

5 Python Optimization

To begin with, we performed some of the basic python optimization techniques we learned in class. First, we replaced all the global variables with local variables. In python, global variables are those defined outside a function class, but they can be accessed from anywhere in the code. On the other hand, local variables are defined inside a function and can only be accessed from the block it is defined in. One of the reasons we changed the code to exclude all global variables is because it improves overall performance of the code - accessing global variables is slower than local since Python has to search for the variable separately every time it is called. Since our program is complex, this can significantly slow down execution, and can easily be avoided by using local variables. Next, since we had a lot of variables we had initially set as global, this was using up memory since Python would have to store all the global variables while executing. Changing them to local variables helped us conserve memory. Another reason to make this change was that we could now treat the variables as parameters. We can do so by changing the default value when calling the functions in which it is defined as a local variable. On the other hand, if the variables are written as a global variable, in order to pass different values we would have to make changes to the code, which is overall clanky and unreadable.

The next major change we made to improve optimization was changing data structures i.e. changing data type from dictionaries to NumPy arrays. This led to a lot of rewriting of the code because of the difference in how we access values in the dictionaries vs. arrays. Making this change helped with reducing the run time in several ways. First, accessing

elements in a NumPy array through indexing is much faster than in dictionaries. NumPy arrays use a simple calculation to determine the block of memory that needs accessing, whereas dictionaries need a hash computation and an equality check, which takes more time. Next, since NumPy arrays are vectorized, performing arithmetic operations on them is much faster as opposed to performing the operation on each element in a dictionary. And finally, NumPy arrays use less memory since they store data in a contiguous block of memory and do not need extra metadata, which is needed for hash tables in dictionaries. This change also ultimately helps us with implementing Numba optimization.

We know that everything we put in a loop gets executed for every loop iteration. Initially, our code had embedded for-loops, which was very inefficient. To that end, we altered the code to remove embedded for-loops and replaced it so that finally we only had one for-loop in the entire program. Embedded for-loops are extremely bad in terms of time complexity, memory usage and code readability. Loop optimization works since it reduces the number of iterations that it takes to perform a task, ultimately improving run time.

Finally, we also altered the code to reduce function call overheads. Function call overhead is the cost associated with calling a function in a program. This is particularly expensive in python since it is a dynamic programming language. Further, a lot of the functions used in our program were written by us (as opposed to being built-in). Thus, altering the code to reduce function call overhead provides an even better opportunity for optimization. We did this by rewriting functions to avoid unnecessary function calls, using inline functions, and replacing them with built-in functions when we could.

Overall, these basic optimization techniques changed the run time from 2.04s to 1.29s, causing a 0.75 second improvement.

6 Cython Optimization

After implementing basic optimization techniques, we aimed to improve our Python code's performance by using Cython as a static compiler. Cython provides a powerful combination of Python and C, enabling seamless integration between Python code and C/C++ code for native performance. With Cython, we were able to easily add static type declarations to our Python code, resulting in this highly optimized C code. Other features in Cython include interacting efficiently with large datasets, such as multi-dimensional NumPy arrays, and leveraging the mature and widely-used CPython ecosystem. Additionally, the combined source code level debugging provided by Cython enabled us to effectively identify a
nd address any bugs in our Python, Cython, and C code.

After loading the Cython module into our environment, we import the Cython Jupyter extension "%load_ext cython". As a first try, the "%%Cython " magic is added before the definition of the two classes and our optimized version "train_cython". Internally, this cell magic compiles the cell into a standalone Cython module.

It is important to note that the effectiveness of using Cython as a static compiler can vary depending on the specific code and dataset being used. While we were able to implement static-type declarations, the overall improvement in performance was minimal. Despite our efforts to optimize the code further using Cython, we found that the performance gains were less significant than we had hoped. From the timeit module, the optimized Cython code is timed at a mean of 1.28s with a standard deviation of 31.1ms. Compared to the python baseline code timed at 1.29s.

By using "%%Cython -a", Cython can highlight lines of code with a background color that reflects its level of optimization. The darker the color, the less optimized the line is, and this is determined by the number of Python API calls on that line. Clicking on any line will reveal the corresponding generated C code. In this particular version, it seems that the code has not been fully optimized. With the annotation system Cython provided we found that only a few lines of code were optimized with C code, and the rest of the code just interacted with python. Despite the limited gains in our case, we were able to implement Cython optimization into our Python code and we decided to try an alternative optimization tool.

7 Numba Optimization

Numba is a powerful library capable of translating a subset of Python and NumPy code into high-performance machine code using the LLVM compiler infrastructure. By incorporating simple annotations, Python code with a focus on arrays and mathematical operations can achieve performance levels similar to C, C++, and Fortran, without the need to switch languages or Python interpreters. This leads to significant performance improvements, particularly in numerical and scientific computing applications. Reinforcement learning algorithms, which often involve numerous iterations and considerable computational expense, can greatly benefit from these optimizations.

To enhance performance, we have created two additional classes (‘cop_class_numba‘ and ‘thief_class_numba‘) utilizing Numba’s ‘jitclass‘ decorator. Accessing class attributes in Python can become a performance bottleneck, especially when used within loops. The ‘jitclass‘ decorator optimizes attribute access, resulting in more efficient code execution. We provide type information for all class attributes, allowing Numba to further optimize the generated code. All methods within a ‘jitclass‘ are compiled as nopython functions. Furthermore, data for ‘jitclass‘ instances are allocated on the heap as C-compatible structures, enabling any compiled function to access the underlying data directly, bypassing the interpreter.

We also employ the ‘@jit(nopython=True)‘ decorator for the train model. The ‘@jit‘ decorator instructs Numba to compile the function using its JIT compiler. By setting the ‘nopython=True‘ argument, the function is compiled in nopython mode, which entails that the entire function is transformed into machine code without any dependence on the Python interpreter. This approach typically yields the highest performance gains. If the compilation process encounters any issues, an exception will be raised.

The ‘%timeit‘ results demonstrate a substantial performance boost when utilizing the Numba-optimized version of the code. For the original Python version (‘train_python‘), the execution time is roughly 1.29 seconds, accompanied by a standard deviation of 274 milliseconds. In contrast, the Numba-optimized version (‘train_numba‘) exhibits an execution time of approximately 79.3 milliseconds and a standard deviation of merely 3 milliseconds. By comparing the execution times, we can observe that the Numba-optimized code is about 16 times faster than its original Python counterpart.

8 Results

Method	**Total Execution Time**	**Wall Time**
Starting Code	2.04 s	2.04 s
Python	1.29 s	58.7 µs
Cython	1.28 s	7.03 s
Numba	225 ms	799 ms

Table 1: Execution times for different optimization techniques

This table summarizes the execution time and wall time of four different methods used in the project. The original implementation without any optimization is referred to as “Starting Code”, while the optimized code in optimized Python methods. The other two methods are implementations using Cython and Numba, which significantly reduced the total execution time. The Cython implementation reduced the wall time to 7.03 seconds with a speed up of 1.01 times, while the Numba implementation further reduced the wall time to 799 ms with an overall speed up of 16 times from the original implementation. These results demonstrate the effectiveness of using optimization techniques like Cython and Numba to improve the performance of Python code.

9 Conclusion

In conclusion, the optimization techniques implemented in this project have significantly reduced the total execution time and wall time required to run the program. The starting code had a total execution time of 2.04 seconds, which was reduced to 1.29 seconds with the use of Python. The implementation of Cython further reduced the total execution time to 1.28s milliseconds, achieving a speedup of 1.01x compared to Python. Similarly, the use of Numba resulted in a total execution time of 225 milliseconds, achieving a significant speedup of 16x compared to Python.

The use of Cython and Numba in this project were instrumental in achieving the significant speedup of the program. Cython helped to optimize the Python code by converting it into C code and creating an executable that could run faster on the system. On the other hand, Numba used just-in-time compilation to optimize the Python code, resulting in faster execution. While both techniques were effective, Cython implementation could be further optimized by converting the Ptyhon code to C interpretable language. However, it is important to note that the implementation of Cython required significantly more wall time than the other techniques, which could be a drawback in certain scenarios.

Overall, the implementation of Cython and Numba provided a significant improvement in the runtime of the program, with Numba providing the best performance. These techniques can be applied in various scenarios where the speed of execution is crucial, especially in data science and scientific computing applications. By using these optimization techniques, programmers can significantly reduce the runtime of their programs and achieve faster results.