An Intent-based Neural Monte Carlo Tree Search Framework for Synthesis of Printed Circuit Boards

by

Arpan Kaphle

B.S Physics and Computer Science and Engineering, Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

© Massachusetts Institute of Technology 2022. All rights reserved.
An Intent-based Neural Monte Carlo Tree Search Framework for Synthesis of Printed Circuit Boards

by

Arpan Kaphle

Submitted to the Department of Electrical Engineering and Computer Science on May 6, 2022, in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

PCB Synthesis is a difficult joint optimization problem that has eluded automation in Electronic Design Automation (EDA) industry until now. Past approaches to create algorithms that intelligently learn to solve these problems have not yet widely been seen. Cadence Design Systems, affiliated with the author, works on solving such problems. This paper proposes the usage of Monte Carlo Tree Search (MCTS) augmented to improve search in order to self-generate datasets, culminating in a process called LFS (Learning Feedback System). This process allows using past data to accelerate MCTS with deep RL models on new or similar board configurations. Datasets are utilized with forms of dataset-based Reinforcement Learning (RL) algorithms, known as 'Offline' and 'Off-Policy' algorithms to solve this problem in a useful and simplified scope. The problem scope starts when other approaches have left a design with constraint violations. This paper baselines with both an algorithmically improved version of MCTS and that further accelerated with PPO, a purely online non demonstrator-based Deep RL algorithm. The results find that MCTS allows for smooth self-generation of datasets, a process inspired by AlphaGo Zero. Adding on to that, we find that off-policy and expert-based RL algorithms such as Adversarial Inverse Reinforcement Learning (AIRL) and Generative Adversarial Imitation Learning (GAIL) can significantly utilize the generated dataset to improve solving the board over time, and do far better when compared to the baseline trained for the same training amount once proper tuning is done. We also find that the complexity of the problem related to the performance of the baseline. Regarding our exploration of offline CQL within this MCTS-connected environment, we find that performance was not up to par, but that it was still able to generalize reasonable actions. We found that all approaches can be tuned to further accelerate MCTS’s decision making and help it prune better for larger state spaces upon comparison of overall actions per episode. The results indicate that amongst the methods tried, the neural accelerated MCTS feedback loop proposed seems to promisingly perform the best with expert-based RL methods.
Thesis Supervisor: Cathy Wu
Title: Assistant Professor

Thesis Supervisor: Taylor Hogan
Title: Distinguished Engineer

Thesis Supervisor: Luke Roberto
Title: Principal Software Engineer
Acknowledgments

Even with COVID going on, I was graciously supported by my family - my father Shiva Kaphle, my mother Gita Kaphle, and my younger brother Aaron Kaphle. I am very grateful to them in ways words can’t express. Their effort and work as immigrants to this country has given my brother and I the chance to work hard and find our own path forward. I’m also grateful to everyone on my team at Cadence, especially Luke Roberto, Jack Murphy, Shang Li, Dominik Martinez, and Taylor Hogan. Luke was an excellent mentor and guide throughout the process. I can confidently say that he was probably one of the most supportive and inspiring mentors I’ve ever had - especially considering the pandemic time during which I have done this research. Jack was super helpful, not just with work but also about navigating the 6A fellowship as a whole. He was always supportive and helped me get compute resources to train my machine learning models. For all of that, I’m incredibly grateful. Shang and Dom were a great help to uncover details about the system I was working on. Taylor was incredibly helpful to assist me in getting additional time and extensions given COVID affected the timeline of this research. Everyone else on the team was also super friendly and helpful.

Even before getting into MIT, I was incredibly lucky to have had such great teachers and friends throughout my life. In high school specifically, I am very thankful to Mr. Newton, Mrs. Ashmead, Mr. Gonzalez, and Mr. McLoda, as well as all my other teachers and peers that helped me get a great education.

To MIT, I’m grateful for the many years I was able to learn and grow here, and to the many friends and experiences I was able to have along the way. I’m thankful to 6A for providing such a wonderful program and to Kathy for helping me get everything in order while in the program. I’m also very grateful to Professor Cathy Wu for being such an awesome supervisor and the rest of her team Combinatorial Optimization team (Zee, Sirui, and Vicky) who taught me so much more about this field. I am super grateful for all those Friday sessions where I learned about the awesome projects everyone was working on, and for getting a chance to hear and talk
to the team! Finally, I’m very thankful to a few of my friends, Victor Phares and Robert Vunabandi, who were with me in a journey of understanding RL far before this research via a reading group we formed dedicated to it. It was a strange period in the world for everyone but with the support I received from all the people above, I was able to put in good effort on this thesis and learn a lot, and for that I am incredibly lucky and thankful.
# Contents

1 Introduction 19

1.1 Modernizing PCB Synthesis ............................................. 20
1.2 Our Approach ............................................................. 22
  1.2.1 The Dataset of Experiences ......................................... 22
  1.2.2 The Learning Feedback System (LFS) .............................. 23
1.3 Approaching this Read .................................................. 23

2 Related Work 25

2.1 Inspiration from AlphaZero .............................................. 25
2.2 Google's Nature Paper - Digital Integrated Chip Design (Digital IC) . 27
2.3 Related PCB Machine Learning Methods ............................... 29

3 Background 31

3.1 Modeling the Problem .................................................... 31
3.2 MCTS .............................................................................. 32
  3.2.1 Vanilla MCTS ............................................................ 33
  3.2.2 Non Machine Learning Augmentations ............................ 35
  3.2.3 Machine Learning Augmentations in Brief ......................... 38
3.3 Reinforcement Learning ................................................... 39
3.4 Deep RL ......................................................................... 40
3.5 Off-Policy and Demonstrator-Based Algorithms ...................... 42
3.6 Purely Offline RL ............................................................ 43
4 Our Work and Additions

4.1 MCTS Adaptation and Algorithmic Augmentations - Java

4.1.1 Generated Features

4.2 Machine Learning - Python

4.2.1 Offline Algorithms Adaptation

4.2.2 Fine Tuning

4.2.3 Online Off Policy and Expert Based Algorithms Adaptation

4.3 The Learning Feedback System (LFS)

5 The Results

5.1 Initial MCTS Results

5.2 Augmented MCTS Vanilla Results

5.3 Augmented MCTS with PPO - Baseline Results

5.4 Augmented MCTS with Offline RL Results

5.5 Augmented MCTS with Online Off-Policy Expert DRL Algorithm - GAIL

5.6 Augmented MCTS with Online Off-Policy Expert DRL Algorithm - AIRL

5.7 The Result of the Winners Strategy

5.8 The Effect of Increasing Dataset Size

5.9 The Effect of Increasing Model Training Epochs and Fine Tuning (Deep RL Training Iterations)

5.10 The Core Result - LFS

5.11 Aggregate Results

5.12 Discussion

5.13 Future Improvements

6 Conclusions

A Tables
List of Figures

1-1  This is an example of a modern PCB being built at Cadence Design Systems. It is an amazing marvel of humanity and you’ll find cousins of if you peek into any compute-capable machine near you! . . . . . . 21

3-1  This shows how MCTS, as it expands and grows its mini sub-tree, explores the better part of the game tree while looking at fraction of all the nodes. This, along with the modifiability and policy improving capabilities of MCTS, is why MCTS is such a powerful algorithm. . . 34

4-1  These are the parameters available on the MCTS Engine, which cover anything from strategy type to tuning simple parameters and choosing actions, amongst many others. . . . . . . . . . . . . . . . . . . . . . . . . . . 49

4-2  This is an example of this internal profiling tool we created to measure performance - it is important to note this particular visualization is not used in our results, but can be generated by the tooling. Our results use the CSV generated data sent to python to allow better analysis - such as through Seaborn in Python as shown in Figure 4-3. However, the gist is that this tool lets us measure relative performances over code regions - like a profiler specified for this MCTS engine - and is how we can figure out the average number of iterations per episode or moves per episode and so on. . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4-3 An example of using the data-set generation suite to save the data and then the python statistical analysis suite to re-draw the performance results more aesthetically. Our results in the next chapter heavily use this strategy. 54

4-4 This is the full column setup at present for the dataset that the MCTS simulation generates. It is highly modifiable and can thus provide robust data to any models. At present, this stores the MDP in the form of $S, S', A, R, T$ and additional data. The simulator can truly output any range of required data from the PCB Environment in Java. Further future work, such as placing this in databases for CI/CD, etc... is wholly possible but was not needed for this research. 54

4-5 All models are dockerized on the cloud, with the left being in Java - MCTS which does the selection, expansion, simulation, backpropagation, and a forward environment runner to allow on policy training. They exist in separate processes within the container to allow simultaneous training and utilization. The dataset of the game is then automatically collected and aggregated via an amplification strategy, and split to be prepared for the winners strategy through a user called script. Then the dataset is used to train online and offline deep RL models in Python, which may utilize the forward environment runner if online (via gRPC IPC). Finally, the deep RL models are fed back into MCTS, which then uses them instead of a random policy during simulation. This is the overall feedback loop generated for this research. PPO, while it can be placed in this loop, is a baseline because it does not fundamentally use the data from the past iteration in a future iteration (i.e. it cannot use this loop idea since it doesn’t rely on the dataset - and thus is a good benchmark against models that do). 56
4-6 This is a simplistic example shown of how CQL may be trained (offline deep RL) via both the amplification and winners strategy before an expensive 'actual' test. Note that you can substitute any of the other experimental models after the training step. The other models only required expert offline data so they did not require this amplification strategy applied to them - they just used all the winning paths for many games as their expert dataset. The Winners Strategy applies to all models for quick evaluation.

4-7 At a very high level, we essentially built a MCTS engine on the Java side, alongside a forward environment runner. On the Python side, which can be run on cloud machines, we have all offline, on-line, and on-line off-policy demonstrator based algorithms and models. The central point is how the dataset and models are built - the Java code-base builds the dataset automatically via MCTS, and the Python side contains pre processors and deep RL model generation.

5-1 Utilizing my board builder suite discussed in Chapter 4, this three dimensional two layer board was built as the main combinatorial board to run all these tests with. It is called 'BoardSeven', and has two layers, 3 nets - each of varying size vias (see the colors - grey, yellow, and green). Each color represents a net - a group of points that need to be connected through wire-traces. Each net is separate from the others. There are L-shaped blockages (the circular, dark black objects), and the combinatorial space requires looking deep because it can't be solved directly since there can be many overlaps.

5-2 We can see a clear improvement in the number of iterations per episode. This is not surprising, as we have increased thread counts in MCTS and modified selection equations, amongst other changes mentioned in the previous chapter.
5-3 All the MCTS + PPO variants base-lined to Augmented MCTS with Vanilla MCTS for comparison. This is a log graph, and the PPO variants show interesting results - with one stand out result for 'Board Seven'.

5-4 The PPO Models show potential if more training (top) was done after fine tuning entropy. The exploration (bottom) converges to stopping - which might change with more time. However, if we see other models perform better at the same epoch counts, especially if they use PPO within their objectives, that may indicate they are improved models for this PCB space when applied with the LFS research concept utilizing MCTS. We will see an example like that with GAIL/AIRL further in this chapter.

5-5 All the MCTS accelerated CQL variants base-lined to Augmented MCTS. The results are not immediately positive, but the ability to generalize was seen for this PCB space with this MCTS+CQL concept generated for this research. However it was not a strong result for this paper.

5-6 The ability to generalize to new states, as we will see this model became capable of, was only possible using the Amplification Strategy. This is the tensor-board result of training loss, first pre-Amplification strategy, then after. Considering the nature of offline deep RL, and the limited dataset size, this was an interesting outcome. Note that the loss range is wildly different after the strategy was applied, improving by a factor of $10^8$.

5-7 The trends are already visible for MCTS+GAIL in changing epochs for training, changing dataset sizes, and the LFS. Since GAIL uses PPO as its generator (the same PPO that runs on its own as the baseline) within the overall optimization, it also provides an interesting comparison to PPO on its own.
5-8 The trends are already visible for MCTS+AIRL in changing epochs for training, changing dataset sizes, and the LFS. Since AIRL uses PPO as its generator (the same PPO that runs on its own as the baseline) within the overall optimization, it also provides an interesting comparison to PPO on its own. Compared to GAIL, we notice similar performance but a better lower bound with AIRL. . . . . . . . . . . . 84

5-9 As we can see, the correlation between these models and the performance discussed around this chapter is that the models that have an accuracy between 5-50 percent generalize better. Note that we have re-normalized 55 percent to be 100 percent. Some interesting suggestions raised by others indicate this may be a potential way to test entropy parameter generation for the models that are tested. That might be something to explore in the future. . . . . . . . . . . . . 86

5-10 We notice that for AIRL, GAIL or CQL, changing dataset size, given our limited dataset sizes, did not have much positive effect. . . . . . . 87

5-11 We notice that PPO gets much worse with more epochs and fine tuning, whereas AIRL and GAIL improve by halving the decisions taken or more. CQL does not change positively due to its 'over-fitting' on dopamine-hits of rewards. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

5-12 We notice the LFS strategy explored in this paper was most promising for the Online Off-policy Expert Based deep RL models - with AIRL performing the best, and GAIL close behind. They achieve similar performance to PPO 50 epochs, while showing potential to continue improving with more training and further feedback loops with more man power and a CI/CD system applied. Notice CQL does not have a round 2 due to the over-fit of round one making it infeasible - thus we just cap its performance. . . . . . . . . . . . . . . . . . . . . . . . . . . 91

5-13 This is the LFS result showcased with just GAIL and AIRL, the most promising models used within the MCTS+Expert DRL concept introduced in this paper for optimizing PCB synthesis. . . . . . . . . . . 93
These are all the model variations tested, which have been discussed in detail above within their respective sections. This is in log scale.

These are all the model variations shown in non-log form to indicate the large difference in decision making accomplished from the start of this research to the end. All of these results were crafted based on the core MCTS Engine software generated in this research.

Only models after Augmented MCTS are directly comparable due to the algorithmic equivalencies. This shows the change from that augmented form alone in non log fashion to show the deep RL improvement from the methodology proposed in this paper.

These are the models that made the best decisions - or took, on average, the least number of iterations (which are proportional to actions). These are only comparable after Augmented MCTS. Even so, considering we start from scratch with the Vanilla MCTS engine for this research, the improvement is quite large on a practical scale!
List of Tables

5.1 Overall Summary of All Results . . . . . . . . . . . . . . . . . . . . . . 98
5.2 The Winners Strategy Results across the Models . . . . . . . . . . . 99
A.1 List of Approaches Taken in this Paper . . . . . . . . . . . . . . . . 111
Chapter 1

Introduction

The world of Printed Circuit Boards (PCBs) is a world that led to the revolution of our present day. Almost overlooked by their abstractions, they provide the basis for the recent Informatics revolution. People often do not think of PCBs, but rather think about their outcomes. Whether it is the most recent Graphics Processing Unit (GPU) from a big name company, or a new computer system (commonly referred to as a 'gaming rig'), people are very fond of the abstractions PCBs bring - exciting games, self driving cars, and the Microsoft Excel sheet that calculates their yearly taxes\textsuperscript{1} amongst an infinitude of possibilities.

Going down levels of abstraction, all PCBs allow is one of the most essential objects of all - an electron - to be a medium to carry information around. They give life to circuits - or rather, they \textbf{are} the very circuits that run our world. All logical signals travel around circuits and run transistor operations which abstract into logic gates - and those snowball into the wonderful world of computers we are all fond of today. So can PCBs run Crysis\textsuperscript{2}? The answer is a resounding 'yes' - or rather more profoundly, Crysis can exist because of the PCBs. PCBs are the most compact circuits created by mankind, which allow nanoscale level operations to work. They are the circulatory system of the computer age. And precisely, for this reason, PCBs

\textsuperscript{1}One of these is not like the others.

\textsuperscript{2}This was a common joke for Computer Hardware enthusiasts - Crysis was a very \textbf{great} looking game created in 2007 that was far ahead of its time in graphics quality. Thus, people often used it as a benchmarks for their systems, and the joke has stuck around since!
are a technology worth improving and investing in. To do this is the aim of this research.

1.1 Modernizing PCB Synthesis

PCBs are generated by a very tedious process. Much as children are taught that a circuit is completed with some power applied to conducting, insulated wires, which let the electron flow through other objects (i.e. LEDs are common), and then go to 'ground' (back to the conventionally negative portion of the battery), designing PCBs requires connecting a incredibly complex circuit that essentially fit that rule. All parts that need to be powered are powered (networks of connections), there are no disconnects anywhere, and the parts are all grouped together in ways that optimize various properties, such as conductivity or resistance amongst a multitude of others. To make it more concrete, this is a constraint satisfaction problem. There are many components in a complex structure, and there is a large combinatorial space to optimize given these constraints. The goal is to properly synthesize all pieces - connect every port, ensure the thermal conditions are proper, signal quality is acceptable, etc... - so the final board is capable of working. Often times, a PCB circuit, if imagined on the same scale as the child’s light bulb circuit, would span anywhere from a room to an entire city or state for the more complex boards. Thus, making this all work in a nano-structure is of the utmost importance, and doing so in a modern way is pivotal to ensure the technology can keep up with the changing world. See Figure 1-1 for an example.

General modern approaches to PCB design often involves large software suites that perform similarly to how mechanical engineers create mechanical components. There is a CAD structure where a PCB can be designed, and multitudes of tools that help users create all the intricate electronic engineering to build a PCB. From understanding complex geometries, to the physical behavior or properties of the final board, these tools give engineers tons of power of a computer. All the powers of copy and paste aside, there are even some methods that utilize algorithms to automate
A Modern Complex PCB

Figure 1-1: This is an example of a modern PCB being built at Cadence Design Systems. It is an amazing marvel of humanity and you’ll find cousins of if you peek into any compute-capable machine near you!

some processes. For example, these software suites may contain the ability to quickly ‘draw out’ the wires for a small net, or place components in simple configurations. However, there has been no designed algorithm thus far that can entirely understand and generate the PCB Board in large part or whole. Even the tools above require a human to have an idea of the overall design - i.e. ensure the circuit actually is design-able. Unlike other circuit domains and their automation tools, the PCB domain has not seen modern tools that can keep up with the complexity of constraints needed to satisfy a complete synthesis - or even a partial synthesis that was unable to complete to the end. Similar systems, such as chip design, are much simpler in nature because chips are more uniform in design and only require robust placement to ensure optimality. PCBs, the circulatory and nervous system for those chips, must account for a wide array of actions, and have a much larger problem space to combinatorially solve within. In general, for optimization, pure deep neural network automation without much nuance tends to make more mistakes, and methods such as brute force or solution trees that eventually solve the problem are computationally intractable. A million year solution is not very useful for something that needs to be shipped next month. As far as we know, no other method exists that can simultaneously synthesize boards better and is well automated and understands at a more fundamental level what it means to ‘synthesize a board’. Our solution,
which aims to teach a model to learn the intent of solving a board, will be a step in that direction. Built to initially solve boards that are slightly partially complete, this model is able to learn the concept of board synthesis and could lead to eventually a path for solving the entire board. We define 'intent' and 'concept' as being able to 'look far ahead' and know how to properly combinatorially combine the various actions and pieces in a way that results in a fully connected and working board with properties optimized.

1.2 Our Approach

We design a system that utilizes Monte Carlo Tree Search (MCTS) with various augmentations combined with offline and online off-policy and on-policy-expert methods and apply them to PCB Synthesis as a novel attempt on this type of board synthesis. Note that we will use the words 'expert' and 'demonstrator' interchangeably throughout the paper. That is because the demonstration data provided to the expert-based models may truly be perfect - or may be generally good, with the real important details hidden amongst common features from multiple decent-but-imperfect demonstrators.

1.2.1 The Dataset of Experiences

The solution we have designed is a system that is able to take advantage of the concept of using a dataset of experiences. Recent developments in RL have started generating models with the ability to understand 'intent' via distilling useful policies from offline or expert experience [7][10]. We utilize such models for our problem space of PCBs, a novel approach in this space as far as we are aware. We build a dataset of PCB designs as Markov chains, known as trajectories from the MCTS algorithm. Then utilizing the dataset, we apply them to algorithms, particularly AIRL, GAIL, PPO, and CQL. These methods help learn the intent behind these demonstrations, with AIRL and CQL hypothesized to do better than others. A plethora of pre-processing tools were generated within Python in order to properly
parse the Java MCTS based output dataset to feed into these deep RL algorithms. Fine tuning and evaluation strategies were generated as testing RL models are not a straightforward process - especially when the final results should be better than the demonstrators that generated the model. While heavily detailed in Chapter 4, an introductory summary of the development can be found in Appendix B.2 and B.3.

1.2.2 The Learning Feedback System (LFS)

The general approach proposed in this research creates a feedback system that will allow for both the generation of demonstration data as well as the generation and improvement of models. This is because we employ a well known policy improvement algorithm in the reinforcement learning space, MCTS. Given the size of the state space, this algorithm helps efficiently solve the problem and generate a dataset as it runs. The output dataset we then utilize to train our deep reinforcement learning offline and expert-based models, which can then be used to assist the MCTS algorithm in the next run. This leads to a feedback loop that will end in both MCTS, and the models it creates, improving. It is similar to other approaches in different fields such as the game of Go. We also generate an inter-process communication (IPC) system for on-policy online training for PPO - which is involved in some expert based models - so that the Python model can be trained through a Java environment.

1.3 Approaching this Read

This thesis is split up as shown in the table of contents, but here we will list the sections with a brief description for readers who wish to focus on a particular aspect.

1. Introduction - An overview on the importance of this field and work.

2. Related Work - While none have solved this exact problem, there are some parallels.

3. Background - Academic description of MCTS and its augmentations, Online Deep RL (AIRM, GAIL, PPO) and Offline Deep RL (CQL)
4. Our Work - Describes the final system built for this paper and what was used from the background and the feedback loop concept introduced in this paper.

5. Performance and Baselines - Describes results of the system with the differing deep RL models used - especially in relation to data size, training differences, and feedback capability.

6. Conclusion - Discusses the work as a whole and steps forward.

This research has shown that exploring these combinatorially large spaces of PCB generation can certainly benefit from utilizing MCTS with augmentations. Then, after further accelerating MCTS with various Deep RL models especially online-off-policy demonstrator based and offline models, we can see that when compared to on-policy online models like PPO, the expert based models perform better at higher training counts. Offline models did not perform as well but still appeared to show promise in generalizing - their weakness might be explained by the small number of data collected in the time frame of this research. More work is certainly required - improved ability to represent a PCB as a graph rather than an image might substantially improve predictions. The very difficult ability to automatically add on additional actions to models, as well as improved fine tuning and generality are all important details that need to be sorted out. Perhaps newer models and techniques will bring that to life in the future. However, this research clearly shows this sort of approach is definitely the right path forward in synthesizing PCB boards as we move to the future.
Chapter 2

Related Work

It is true that while no example of this approach can be found (to the best of our knowledge) with applications to Printed Circuit Boards, our approach was inspired by other great work in recent time. There are some similar papers which need to be discussed in detail and acknowledged for their contributions to the field. We begin with discussing AlphaZero, an inspiration for this paper and exactly what we gleaned from it. Then we move on to Google’s recent Nature paper on Digital Integrated Chip Design (Digital IC), and compare it to our present problem. Finally, we look at other attempts that also work in this PCB space but differ in method and scope.

2.1 Inspiration from AlphaZero

AlphaGo Zero is a fairly famous recent adventure in the RL space. The utilization of MCTS with multi-head policy and value networks allowed the generation of neural networks that play the game of Go with superhuman ability [5]. Similar to our research here, humans did not provide input in the training loop. It utilizes MCTS - a known policy improvement operator - to improve neural networks, of which there are two - a value and policy head. Then it runs multiple games and improves the networks through self play within MCTS until the networks are capable of working on their own with high accuracy. This is done in a highly parallel and scaled setting.

Their work initially used human demonstrator data [4]. However, in their next
iteration, known as AlphaGo Zero (or AlphaZero), they used the policy head to predict probabilities for next actions based on the MCTS algorithms node values of visitations and average reward [5]. They also trained the value head based on the node frequency values that MCTS produced - where these frequencies acted as probabilities. This is an approach known as 'soft labeling' where all possible paths are considered and trained directly as probabilities to the policy network. Thus, utilizing their policy and value heads, the MCTS algorithm ran with these networks as their decision makers (rather than heuristics or the random actions MCTS conducts - more on MCTS in Chapter 3). It guaranteed a proper exploration/exploitation given the structure of MCTS, even if the models were initially random. It would only train the models after an episode finished. It also avoided the rollout step of MCTS via using the value network, which avoided a high compute step - even if it may have different slightly in accuracy.

Our work is heavily based on the 'no human in the loop' model with MCTS. Our field is far separate, and the models we utilize are far different. For example, we utilize offline RL DQN (CQL) [11] models and expert based intent learning on-line off-policy RL algorithms [7]. We do not train a value head but instead utilize demonstrator based models that predict reward with objectives that naturally improve upon the exploitation/exploration problem and properly decouple states from dynamics. We also use the full MCTS tree in dataset form to generate 'hard labels' (our experts) and a ring buffer for a fully offline method - CQL. We do not stop the rollout as the rollout provides the solution MDP. We also do not have an adversarial scenario, just a constraint optimization for a single player. We hope that in this different field, our similar but differing approach will allow even better natural improvement of the models to learn how to fix and solve the boards they are trained for. It is possible that we could have explored DAgger (BC - Behavioral Cloning) or this sort of AlphaZero like system (at a simpler scale, given time and manpower resources of this research) as another benchmark, and that is left for future research. We chose to benchmark with PPO for this current work.
2.2 Google’s Nature Paper - Digital Integrated Chip Design (Digital IC)

Google’s paper is from an adjacent field - chip design. Chips are the processors at the heart of PCBs - but the combinatorial space of PCBs are far larger than that in Digital IC. PCBs are three dimensionally layered, with non homogeneous components that route and exist across the aforementioned dimensions. Digital IC does not route outside of 90 degree angles, which makes routing a simpler problem. The chips also contain macros which are very uniform components (essentially always different sized squares and rectangles). They also do not contain three dimensionally layered components - even if the wires can cross layers, there will not be components in other layers. Thus training a pure deep RL algorithm (i.e. PPO) properly long enough will be effective. This is possible since their problem scope is much smaller and their simulators run very fast, achieving agent-environment interactions over thousands of iterations. Their constraint space and reward functions are also simpler and only proxy the routing, so utilizing PPO (in a complex and heavily configured/thought out way) is enough to get results. Google’s Digital IC paper has conducted this work and they were able to be fantastically successful. They are a great beacon to show that this type of work is the right approach in this field [13].

To further cement the difference, let us first discuss optimization - it’s true that the point of this approach is to optimize PCB synthesis (in the use case of fixing a partially completed board). This deep RL accelerated MCTS system, in this unique expert/offline RL experimentation form, is merely an optimization tool. Given that, one may be questioning the purpose of this work - why not apply it to digital IC? It is clear it can be - though with a huge amount of auxiliary code modifications that might take quite a bit of development time. The entire codebase was built around PCBs. However, the optimizer underneath is logically capable of exploring anything with a combinatorial tree or space. The importance is what configuration space and constraints the optimizer was built for. Consider any game tree - for example Chess and Go. Chess may be more complex than Go with more constraints
(rules), but Go has **far more possible states**. Thus when directly comparing overall complexity, the optimization space of Go is far larger to master. Chips from digital IC and PCBs are similar in this way - except PCBs not only have a larger configuration space **but also more constraints**. Thus optimizing PCBs requires an optimizer design that is better suited to handle the more complex problem. And yes - **it can also tackle Chips** - but the intended design was generated under the larger optimization space assumption. Even if the example problems explored in this paper do not hit on the complexity, the model design certainly was based on it! The idea is that if the design introduced in this paper were scaled up, it would be able to work in this more complex combinatorial space of PCBs better (but with enough auxiliary code changes - simple but potentially time consuming development changes, not on the main algorithm - would work for Chips as well).

**This research aims to combine and create this feedback loop** under the assumption that PCBs require **more generality and ability to work in higher sized game trees**. At the very least, under that assumption, the goal is to show this optimization strategy proposed in the paper, if scaled up, has the potential to be **used for PCB synthesis** and can showcase improved generalization capabilities to warrant further work.

Outside of differing scope, discussing only the technology built, their approaches are quite different when comparing to the work done in this paper. They do not utilize MCTS as a improvement operator for their models. They also do not utilize AIRL and GAIL, two recent robust expert and entropy based off-policy deep RL models that have promise with broken/changing dynamics, which matches the structure of our problem scope. This theoretically may indicate better generality, which we explore in this paper. Due to the aforementioned wider combinatorial PCB space, the approach pursued in our current paper is properly different from that taken by this Chip Placement paper to meet the higher combinatorial space. This current paper will benchmark the system against a PPO network (on its own) to showcase the difference with our approach. In our case it's just to compare with the other deep RL models (which conveniently also include PPO in their generator process), such
as the previously mentioned AIRL and GAIL models. We also utilize an offline RL model - particularly CQL - which is yet another avenue of exploration that Google’s paper does not. So not only is the domain itself different - both semantically and combinatorially - but our approaches are quite varied. However, once more, Google’s chip paper is a great sign that this sort of approach for this style of work is the right one.

2.3 Related PCB Machine Learning Methods

There has been work done by others within the PCB space at Cadence. They explore other avenues of this problem space. We will refer to three here - the first goes over utilizing a neural network to predict PCB routability [14]. This is a very useful predictive power that may give insight into how a PCB may be fully connected. It’s completely adjacent to our work but could actually be used within our work to retrieve reward signals, for example. The second goes over utilizing PPO to solve a simplified PCB problem - it’s very similar to our work, except it is not utilizing offline nor expert based models, nor MCTS. It also uses a toy domain - we use a toy problem but in the actual domain of classified PCB technology used within the partner company for this research. It can be thought of as a direct adaptation of the google paper to the PCB domain [1]. The third paper attempts to utilize a slightly adjacent field within learning - genetic optimization - and applies this to the routing and via strategy [17]. This work can help form the inner actions that our system uses, but is focused on a piece of the puzzle. Our work can utilize this work to build a larger system that can solve/fix boards generally. Our aim is primarily to ‘fix’ boards but with an optimizer that could theoretically be expanded to entirely solve it.

No other work as far as we are aware has been identified to be exactly like this one applied in the PCB space - a MCTS accelerated offline and expert based online deep learning system. In fact, as far as we are aware, this is the only research into a dataset oriented feedback loop utilizing MCTS and expert based online and offline
deep reinforcement learning methods like AIRL, GAIL, and CQL for any purpose. However, these other similar works certainly show inspiring vantage points on why this methodology is worth pursuing.

[^1]: Plenty of methods exist that combine deep RL with MCTS - but none have used AIRL, GAIL, or CQL with MCTS in this data-set based feedback loop generated in this paper (LFS) as far as a deep search online can indicate. However, it’s clear that MCTS plus Deep RL, in general, are heavily efficient together and lots of recent literature and research indicates this.
Chapter 3

Background

This paper utilizes recent advances in Deep Reinforcement Learning, Offline and Off-Policy Learning, and MCTS. Thus we aim to provide a survey of much of that material in this chapter. The discussion will center around MCTS, an online deep RL benchmark, the various off-policy and demonstrator algorithms, as well as the offline algorithm that we use for the main set of experiments.

3.1 Modeling the Problem

All the methods that we utilize in various combinations during this thesis follow a general mathematical formulation - that of the Markov decision process (MDP). An MDP is a graph of events occurring through a number of time steps. The graph is a sequence of actions taken within some environment, with feedback presented per step from the environment, such as reward. In some deep RL models, the MDP formulation involves a transition probability - the underlying transition network of the environment at hand. In other problems (for example model-free RL problems), we are able to marginalize out the transition probability due to high sample count and focus more on the state and action pair and do not see all future states (hence the model is focused on the distribution of the (A|S) random variable). Thus while there is some underlying MDP with probabilities for transition, the model generating that MDP is unaware of those transitions but learns them with enough sampling.

31
Simply put, an MDP can be defined as a 5 vector of \((S,A,R,S',T)\)\(^1\) where each element is itself a tensor relating to the trajectory formed by the MDP.

A few other parameters exist as well, namely \(\gamma, S_0\), etc... - also known as the discount factor, the initial state, etc... - they can be thought of as implicitly being formed or used within some MDPs. \(S\) refers to the states/observations (the former is for full observance, the latter is for partially viewed observance) along the trajectory. It is a tensor that is as long as the trajectory, and per each index contains a tensor related to the observation - i.e a multi-channeled image, or a simple vector.

\(A\) is a tensor that contains per each \(i\)'th index, the action performed at the \(i\)'th state. \(R\) is the reward tensor, which contains at every \(i\)'th state, the immediate reward received for taking action \(A_i\) at state \(S_i\). \(S'\), which may be seen as redundant (but in some parts of the dataset where the buffer is not in order, may be important) is merely the state reached after taking \(A_i\) from \(S_i\). The discount factor matters for problems with horizon where each step may have some underlying probability of complete failure that prevents the system from taking any more future states. It is more commonly used to think of overall horizon, however - the number of timesteps in the future until afterwards the agent does not care about receiving reward. Other miscellaneous parameters, such as the start state, or any other numerous creative additions, may help define initial conditions for the system as well as any unique changes.

### 3.2 MCTS

MCTS is a very natural algorithm developed to help explore enormous combinatorial action spaces. The vanilla version of this algorithm is essentially improved tree search with random sampling \([3]\). What we and many others have done is utilize ML models to change its tree search and improve sampling to be better than normal through industry-specific priors or modern deep network architectures.

\(^1\)Yes, this acronym does contain a subset that matches the name of a certain sneaky virus that has been around recently!
3.2.1 Vanilla MCTS

This is the initial algorithm, formed in 2006 by Kocsis and Szepesvari - and Coulom - in application for the board game of Go [6]. It was a natural breakthrough where even considering enormous action spaces - like that of many games - it could allow retrieving a tractable solution with exploration and exploitation parameters. On its own, it accelerated algorithmic Go adversaries by some margin, but more recent works rely on augmentations to the vanilla form which we will discuss later. MCTS has four main phases:

1. **Selection** - Utilizing the special 'tree policy' algorithms for exploration or exploitation, known as UCB (Upper Confidence Bounds), the algorithm chooses a leaf node in the presently expanded tree (aka the game tree that’s been explored outside of simulations - a tiny subset of an enormous tree). UCT is UCB applied to trees. It gives a node a particular score - a value, similar to that of a discounted value in RL since it comes from looking deep into the tree. This value is as follows: \( v = \arg\max_{a \in A(s)}(Q(s, a) + C \cdot \sqrt{\ln[N(s)]/N(s, a)}) \). Essentially, a particular 'node choice' - or an 'action' out of possible next actions (possible nodes to reach) gets a value. That value is the sum of the average value of that next node plus a quantity that compares the visitation number of that particular node and from the node being visited from (aka the present state). \( Q(s, a) \) is that average value - from state \( s \), going to node \( s' \) via action \( a \) had in the past after rollouts and backpropagation resulted in some average value \( (V/N(s, a)) \) where \( V \) is the sum of all past values). The second 'exploitation' quantity will allow a lower score if the relative ratio of the current state visitation \( (N(s)) \) and the next state \( (N(s, a)) \) are similar. If we have never gone to the next state, we will get an infinite value for the UCT score so we will definitely follow it. The idea is that over time, even in an enormous game tree, this will be able to start 'choosing' good paths [6].

2. **Expansion** - Selection keeps occurring until we get to a leaf of the mini-tree we physically have (i.e. the subset of the huge game tree). The first node that has
yet unexplored nodes (visitation of zero), or is empty and needs to be expanded or terminal already, causes further change. If it is terminal, then we ended up with a quick solution - this directly back-propagates. If it is a previously visited node but not terminal, we expand, and simulate [6].

3. **Simulation** - Also known as ‘rollouts’, this is the heart of the algorithm. In vanilla, this is **random search** down a special subset of the enormous game tree - either to termination or to a heuristic depth. This is the *monte carlo* part as it is random search [6].

4. **Backpropagation** - This step uses the final node reached in simulation to update the node from the node that *started* the rollout and above. This is what gives MCTS the ability to ‘look ahead’ even in an enormous state space [6]. This is done by adding the final value of this rolled-out node far down in a part of the game tree that will be forgotten about to that node above and higher. This allows getting a feel for what this ‘side of the game tree’ weighs. This allows for asymmetric tree growth, which is how MCTS finds the ‘better portion’ of the larger game tree without fully exploring it. See Figure 3-1 [6] for an example.

**The Asymmetry of MCTS**

![The Asymmetry of MCTS](image)

Figure 3-1: This shows how MCTS, as it expands and grows its mini sub-tree, explores the **better** part of the game tree while looking at fraction of all the nodes. This, along with the modifiability and policy improving capabilities of MCTS, is why MCTS is such a powerful algorithm.

Doing all 4 of the above steps is considered **one iteration** of the algorithm. The
full algorithm is just running as many iterations as possible under a time limit. This is the think time, after which we make what will be defined in this paper as a move. The algorithm repeats these moves until the final state is reached (terminal state) where the problem is solved – we may refer to this as an episode.

**Algorithm 1** Monte Carlo Tree Search

Require: $GAME_{NODE}$ exists

Require: $Various_{ConfigsSet}$

while $GAME_{NODE} \neq Terminal$ do
  $T_{max}$ exists
  $I_{max}$ exists
  $VariousConfig$ set
  $i \leftarrow 0$ \hspace{1cm} $\triangleright$ Iteration
  $T \leftarrow 0$ \hspace{1cm} $\triangleright$ Time in MS per Iteration
  $N \leftarrow GAME_{NODE}$ \hspace{1cm} $\triangleright$ Present Root Node = Game Main State
  while $T \leq T_{max}$ and $i \leq I_{max}$ do
    RunProcedureSelection()
    RunProcedureExpansion()
    RunProcedureRollout()
    RunProcedureBackProp() \hspace{1cm} $\triangleright$ N keeps updating and expanding
  end while
  $GAME_{NODE} \leftarrow N$ \hspace{1cm} $\triangleright$ A ’move’ occurs and the main state is updated
end while

That is the essence of vanilla MCTS. While a practical starting point, there are tons of augmentations we can take. Let us focus on quickly listing the theoretical ones we’ve considered, and in future chapters, we will describe the ones that were implemented.

### 3.2.2 Non Machine Learning Augmentations

The power of MCTS becomes apparent when it is customized for unique scenarios. Every aspect of this algorithm can be augmented or tuned. Each move taken is done after a set number of iterations or time-limit. Depending on the time available, this can be highly modified or changed. Once many moves have occurred to a terminal point (i.e. a PCB is fixed), we have finished an episode. High iterations, high time limits will raise the time per move. Low time limits may not allow high iterations to
work but may allow making more moves quicker (albeit with less in-depth explored trees - 'less think time' if you will). Each piece of the iteration can be modified - and for our use case, a dataset of MDPs can be collected easily via backtracking the various paths the algorithm explores.

It is worth noting that for benchmark purposes, we have chosen to allow maximal time per move. This effectively means when we benchmark algorithms later, they are purely being compared on their ability to make better overall decisions on an iteration by iteration basis. In other words, we give it enough time such that any noticeable differences will come from making better decisions in the iteration amount they have. This makes iteration ∝ actions for all models that have equal MCTS based augmentations. This is a very important point that should be remembered when the results are shown in Chapter 5. This only works, however, when the algorithmic acceleration is equivalent. This is another important idea that will be discussed more in the results, but it simply ensures equivalent comparison outside the changing deep neural network models.

With that said, let me briefly describe the various augmentations and fine tuning one can conduct in MCTS without machine learning models mixed in.

1. Selection - The UCB algorithm described above can be modified. **UCB-Tuned** is a more popular one with better practical guarantees. There are many such forms of these algorithms, such as Bayes UCT, EXP3, heuristic values for nodes to improve weighing, "opening books" heuristics to change the node value, rapid action value estimation (RAVE), All Moves As Fast (AMAF), etc... they all attempt to use some sort of heuristic or number to modify the UCB equations [3][6].

2. Expansion - The default strategy is to just pick the first unexpanded node, and expand it. There can be many ways to do this that can modify speed. Pruning actions via heuristics, or storing actions or not storing them based on network usage can all impact the performance and space that the model takes up. Generating the system modularly allows easily swapping in strategies [3].
Note that we describe these additional configurations since in my case the model was in a state machine that was initially configured over RPC network calls.

3. Simulation - The most heuristic laden one, this core piece will benefit greatly from action pruning. It has a default random strategy, so putting the ability to easily swap strategies maybe be useful. Pruning with domain knowledge is a possibility. Various heuristics, such as rollout depth, thread count (leaf parallelization), etc. can also be utilized to speed up the model. Some may attempt to add decaying rewards, some may group a set of moves into one large 'super-move'. Good system design can make this a fun fine tuning location to swap and test various changes.

4. Back-propagation - this one can best be improved by changing how the number is aggregated. For example, if parallel leaf parallelization is done for simulation, one may wish to average the values from all leaves, rather than keep the mode of the results or just one. This can lead to improved node accuracy. The numerical value can be modified as long as this strategy is kept modular.

5. General Changes 1 - If the game is 'single player' (i.e. like our problem), one can benefit very widely from pruning the entire tree outside the chosen move. If this is done move by move, this can save a ton of space. Also, move by move, reusing the sub-tree that was already expanded also can save time - as the next move starts with an expanded tree. Normally, since MCTS was made for two player games, the algorithm re-sets the tree to a single node after every move - but in single player games it does not need to do this. Our PCB optimization is a single player game since it is just looking for a solution with no adversary. Collecting a dataset allows flexibility with this even further as the data can be retained and saved to deep storage without affecting in-memory program usage - and the data can be safely pruned each move without loss of useful data for later cases.

6. General Changes 2 - Early Termination is yet another interesting use case
which can be more useful for single player games (and still useful at finishing moves for multi-player games). Whether in simulation rollout or not, there should be a way to bring think time 'rollouts' (i.e. fractional pieces before a move that contain many thought out future paths) out to the world if and only if it results in a successful win. This of course can be done in adversarial games as well, but it is a necessity in this problem. This must be done while accounting for parallel rollouts if that augmentation is added. Beyond these, there are many more non deep learning augmentations that can be applied, which can be found in recent survey papers [6].

3.2.3 Machine Learning Augmentations in Brief

The augmentations discussed above (action pruning/heuristic values) - naturally fall in line with machine learning. If the dataset is gathered, or a forward on-policy training environment is generated, then deep RL models can be trained that can eventually be fed back into MCTS. This is the ultimate goal of this paper - a strategy called "LFS", or the Learning Feedback System. In the expansion phase, for example, we could have a neural net predict the action to take. In the multi-threaded rollouts, a network could predict the best actions each time. This is essentially the ultimate 'history heuristic' as it learns from past data/data about the environment. Reinforcement Learning algorithms work really well here, as can normal value/policy heads. There are many ways to do this - some cleverly 'bootstrap-raise' MCTS, such as in AlphaGo, where the old policy is used with the UCB strategy of MCTS, but the training of the policy is only done after a full game. Others, such as in this work for example, could use each dataset generated game with one policy to train another offline/expert-based policy, then start a feedback with said offline/expert-based policy until the final system improves.

There are many ways to bring ML algorithms into the mix - but the idea is that MCTS can utilize them to further improve itself and these models in the next step [6][3]. This research, as far as we know, is the first to mix MCTS with Offline and Online Off-Policy Expert based deep RL algorithms in this data-set.
feedback loop - certainly within the space of PCB problems. However, it is heavily inspired by the MCTS and value-head/policy-head combination used by DeepMind for their work on Go.

### 3.3 Reinforcement Learning

The mathematical foundation for RL is akin to MCTS and has been discussed above when MDPs were discussed. One addition is that the hope with RL is to maximize a particular objective - the sum of discounted future rewards, also known as the value function. This core principle is at the heart of all of RL and Deep RL - any modifications and strategies always relate to this optimization in some way or another.

\[
\text{max } V(s_0) = \text{max } \sum_t (\gamma_t \ast R(s_t, a_t))
\]

This essentially means that we want to, at any current state, \(S_0\), go to the future and reap the highest rewards over time - accounting for the discounting rate of return. Note that this equation is simplistic - adding in transition dynamics, probability of neural net models, etc... will complexify this greatly - however for a general view, the concept is clear with this equation. In general, there are descriptions like 'value' or 'q value' - these are just distribution shifted variants of this over-arching idea of maximizing sum of total rewards. Value assumes that you’re able to see all possible actions and where you end up next (i.e. you have the transition dynamics and thus things can be computed in terms of states alone). The 'q value' assumes you can’t see the transition dynamics (so it is state, action based) - meaning it is model-free (no transition dynamics visible) - but over time you marginalize the transition dynamics out with enough samples. Notice the sum of \(q(s, a)\) values for state \(s\) and all possible \(a\) is equivalent to the \(v(s)\).

You might have noticed MCTS has similarities to RL - this is due to the underlying MDP structure, and MCTS can be seen as a 'type' of RL algorithm. However, RL has a much deeper history, which we quickly dive into now. Initially, RL was
research on making the best decisions over an horizon, with exploration and exploitation being the most pivotal concept [8]. The field relied heavily on theories of Markov Chains and MDPs, and the probability theory involved. The first algorithms were full-backup value and policy iteration algorithms. Using dynamic programming, and a transition dynamics model (where it was assumed the environment had some underlying known transition probabilities), a good path and policy could be uncovered to take an ideal set of steps forward to optimize a larger term objective. These methods are called model-based because the agent is able to see all future potential states and can work directly with the underlying transition dynamics. Notice that MCTS is effectively a model-based method. Model-Free versions essentially marginalize out any underlying transition probabilities by increasing the number of samples. This led to algorithms like Q Learning or TD Lambda [8]. Dissatisfied with simple model-based algorithms, further improvements such as prioritized sweeping and dynaQ were implemented to take advantage of past experiences to ‘guess’ model transition dynamics while reducing the number of samples required to learn how to solve a problem. It wasn’t long before machine learning started to enter the fray with deep learning, but other aspects - such as partition-based learning or partial-observation-based learning (when the full state was hidden - aka hidden states) also came about [8].

3.4 Deep RL

Jumping to deep RL just means adding in neural network models. A model that could predict the action per any state in hopes of maximizing the long term value certainly seems like the natural step to take [8][10]. The slightly modified optimization equation became as follows:

\[ \theta^* = \arg \max_{\theta} E_{\tau \sim \rho(\tau)} \left[ \sum_t (\gamma_t \cdot R(s_t, a_t)) \right] \]

\[ \text{Do not dismay, these techniques too have seen deep learning voraciously swallow them - as everything eventually is nowadays! For example, for hidden states, recurrent neural networks and further attention based transformers can be helpful [10].} \]
All that's saying is that now we want the neural network, $\theta$, which provides for any random value trajectory ($\tau$) of possible trajectories it can create (from the distribution of random variable $T$ for all possible trajectories the current $\theta$ can traverse), $\tau \leftarrow p_\theta(T)$, a maximal reward when traversing all those possibilities. The time-steps in the summation refer to the steps taken in any of those possible trajectories. It will be the ideal policy network, capable of reaping the highest reward from any point of view or angle of the network. This equation will stretch, twist, and be heavily modified\textsuperscript{3}, but is the core of most of the deep RL work.

A first method, in the sub RL-field of Behavior Cloning (BC), called DAgger(Dataset Aggregation) let a network take a whole bunch of random on-policy actions. On Policy means the model that ultimately undergoes gradient descent optimization is also the one that initially made the forward decisions\textsuperscript{4}. For DAgger, upon training, a human would edit each 'wrong action' the network made with the human assisted 'right' action, and re-train the net on this 'updated' dataset. This led to an improving network. Other attempts revolved around PID(Proportional, Integral, Derivative)-like correcting, and inclusion for continuous actions via Gaussian mixture models and other similar methods [10][8].

DAgger was deemed too costly as it required human intervention - though there is methodology to replace the human relabeling with MCTS which may be warranted an exploration in a future paper for the current research. Thus an autonomous on-policy method, commonly called REINFORCE, was born. It is the back bone of most on-policy deep RL algorithms (also commonly called policy-gradient). It involves gathering random paths, than automatically summing the paths for their rewards and going in the direction of highest expected reward (it is expected because the neural networks make probability-defined choices - either from the underlying transition dynamics or the neural network outputs). This objective is successful, and over time, leads to an improved model with no human input. Further improvements

\textsuperscript{3}Like a good Pizza

\textsuperscript{4}Without jargon, the same 'being' experiences the world (random actions), goes home, learns from it during sleep (training). Then repeats. But in the case of DAgger, 'external actors' modify its 'memories' so it learns better in training.
to the back-propagation style and improved gradient searching via a trust policy optimization region will lead this to more advanced policy-gradient methods. PPO is a modern version of these algorithms, with an improved objective function that clips the KL-Divergence (entropy difference between distributions). It is a very effective algorithm but needs to always ‘see new data’ on its own as it is on-policy. In some cases, Trust Region Policy Optimization (TRPO) might be preferred to PPO. Our research will see PPO used as the baseline, but future research could compare with TRPO for this PCB problem in this MCTS-based LFS setting. PPO - as we saw in this research - appears to get results with fine tuning that matches the complexity of the environment it is solving with. It also requires extensive tuning.

Off-Policy methods arose when imagining various models working together - such as A2C (advantage actor critic) which instead had another algorithm ‘explore’ and yet another predict the value. This lead to model-free, online, off-policy deep RL networks, such as DQNs, where the underlying transition distribution was marginalized out. This allowed for ring buffer storage of data - and data could come from another model so as long as it was from the same environment. Some models were bootstrapped, like DQN - this meant they essentially improved themselves using older distributions of themselves lagged by a certain amount of time-steps to avoid distributional drift. Since the older models are essentially a bit different from the present model, it is effectively like an off-policy second model with the exception that it is distributionally related within acceptable tolerance.

Now, these earlier methods have spun off into many more - such as deep model-based RL, meta learning, curriculum learning, etc... - but we’ll focus on how the DQN got involved in offline learning, and how some demonstrator-based off-policy algorithms arose, as that’s what we use in our paper.

### 3.5 Off-Policy and Demonstrator-Based Algorithms

While the whole on-policy and off-policy approaches were being developed above, another field focusing on imitation learning was taking shape. A particular paper, En-
**tropy based Inverse Reinforcement Learning** [2] showcased that one could use demonstrations and build an algorithm that could favor commonly re-used features of high value in those demonstrations via entropy. Entropy ensured information outside the demonstrations were as random as possible. This allows for a more ‘natural’ improvement - especially as future data came in to fill in the present unknowns - with no bias or over-assumption. This could all be done without rewards known at the beginning. This method was successful - but unfortunately not for high dimensional spaces.

Luckily, with advances in deep learning, these ideas re-emerged with tractability in high state spaces. These methods were good but not robust - such as guided cost learning. A method called Generative Adversarial Imitation Learning used generative adversarial networks (GANs) within the process and had great success. This worked with an on-policy generator - i.e. PPO, TRPO, REINFORCE, or similar - that would create policies that aimed to be like the demonstrators; a discriminator network would attempt to reject the generators and accept the fed-in demonstrators. This would ultimately lead to a powerful on-policy generator. However, it was shown with GAIL that due to reliance on a connected state action reward, the transition dynamics distributions were baked into the goal reward state, which would be non-robust against changes. Thus, Adversarial Inverse Reinforcement Learning (AIRL) was created. This allowed decoupling the goal from the dynamics - which allowed for a more robust and general model than possible before [7][10]. This is particularly promising for our research as the idea of decoupling transitions maybe useful for robust PCB synthesis. More is discussed in Appendix B.1.

### 3.6 Purely Offline RL

Alongside all the developments above, further ideas were explored for those who *just love storing data*[^5] One of these larger and more recent ideas is offline RL. As discussed above, on-policy is when the model explores and collects data on its own - data it uses to train itself in the future - like **REINFORCE**. Off-Policy

[^5]: Which really is every company in the world it seems nowadays!
allows data to be collected by another model - any model as long as the environment is consistent - and then the model wanting to be trained can utilize this data and maybe also enter a simulation (like normal DQNs or AIRL/GAIL). This can go as far as bootstrapping after a certain amount of time has passed to avoid distribution shift - such as is done in DQNs [10]. In effect, the 'off-policy' algorithms are offline in the period where they do not get any new input data or enter live simulations.

Further continuing these ideas of distribution shift and off-policy queues of data, a methodology of offline RL was created. It involved never collecting new data again - or at least it was designed to work without it. New data could just be used to fine tune in the future. The idea was that with the old existing data, a model could be generated that was capable of functioning. It was almost a pure 'supervised learning' for reinforcement learning agents - and they never needed access to the environment. Most models rely on statistical tricks to avoid over-estimating values and to ensure the distributions do not shift away from the true underlying environment. You can essentially imagine this as attempting to teach the nuances of simulation with data lying around from the past. Theoretically speaking, a simulation is just a MDP - and if a model were to be transported back in time after a simulation, all changes undone, and told to ‘train from the MDP’ data of the future simulation in exactly the same way as the simulation, learning should be equivalent. This is the idea behind offline RL. We choose to explore it because it might be beneficial to know how it performs in our use case, as this means this model could be beneficial over time as more and more data is aggregated.

One particularly famous and recent Offline RL models is CQL, or Conservative Q Learning [11][9]. Based on DQNs, a common model-free deep RL network discussed above, these models ensure that theoretically, under some fine tuned parameter, it is possible to under-estimate Q values in purely offline training. The largest hurdle of offline RL, as mentioned prior, is attempting to make a model with only a fixed amount of information. The distributional shifts in the models predictive power over

---

6Humans love taking things to the extreme! Though of course this has very practical cases where tons of data exists.
the environment as they train are often over-optimistic - CQL wrangles that in via modifying the bellman error objective with a Q-Value regularizer. While requiring heavy tuning, this has shown to be theoretically powerful. This gives a great foundation for teams with tons of data and no easy simulator - a working deep RL model can be formed from that alone. The implementation of the CQL code-base is quite more in-depth than the pure theory of their paper, and required a deep analysis. A summary of that can be found in Algorithm 2. Custom additions modify the input to the training loop, and generate a procedure to save the model once the training is done in accordance with my heavily differing use-case (theirs was built in Python for Atari games while ours is for a classified java-based PCB system).
Algorithm 2 CQL Implementation Deep Dive

Require: \( \beta_{data} \) exists ➤ PreProcessed Data for Ring Q Buffer in Suffix-Based Forms

Require: DQNAgentModel exists ➤ Various types - Multi-Head/Network/etc...

Require: DQNAgentTuning exists ➤ InnerMLP, Update Horizon, Batch Size, etc...

Require: \( NumBuffer \leftarrow N_0 \)

Require: \( DQNAgentModel \leftarrow \beta([\beta_1, \beta_2, \text{etc.}]) \) ➤ \( N_0 \) Parallel Loaded suffix-data

\( I_{max} \leftarrow N_1 \)

\( I_{cur} \leftarrow 0 \)

while \( I_{cur} \leq I_{max} \) do

\( T_{max} \leftarrow N_2 \)

\( T_{cur} \leftarrow 0 \)

\( \beta[\beta_1, \beta_2, \text{etc.}] \leftarrow P_{uniform}(B_{all}) \) ➤ Re-sampled parallel load in data each iteration

while \( T_{cur} \leq T_{max} \) do

if \( \text{TrainConfigTrue} \) then

\( BatchSize \leftarrow N_3 \) ➤ Batch Size set

\( \beta_{sample} \leftarrow P_{uniform}(\beta[\beta_1, \beta_2, \text{etc.}]) \) ➤ Uniform suffix sample

SampleBatchSizeFrom\( \beta_{sample}() \)

This is a Complex step - MDP and \( V(S_0) \) are found with batch size length. Proper sampling and bounds checking done.

TrainAgentWithCQLRegularization() ➤ Train network from batch

end if

if \( \text{LogConfigTrue} \) then

Log Tensorboard Data

else

Skip ➤ It’s assumed all variables are properly incremented

end if

end while

end while

SaveModel()
Chapter 4

Our Work and Additions

In this section, we will describe the augmentations applied to MCTS for our use case, and why those choices were made. We will also describe the overall system generated in this research. It should be noted the reason for the split codebase in Python and Java was due to the origin of this research. This research was done with Cadence, and Cadence IP - such as the PCB environment - was in Java. The deep learning work required using very modern research which was mostly in Python and unrealistic to convert into Java in this time-frame of the research. Thus an IPC system reflected any required state to the Python side to work with modern deep RL tools in a time-sensible manner.

4.1 MCTS Adaptation and Algorithmic Augmentations - Java

A large part of the first half of this project was spent generating a highly customised and configurable MCTS Engine that would work within a distributed environment. Many of the algorithmic augmentations mentioned in Chapter 3 are included, as are various other architecture choices that helped design a modular system ready for change. If anything is unclear below, please refer to Chapter 3 for its background.
4.1.1 Generated Features

See Figure 4-1 on the configuration file of the MCTS Engine built from the ground up for this research (I have redacted some internal naming). As you can see, there are quite a lot of designed features. Each software suite or state machine took a significant amount of time to develop and acted as its own application. They are described below and their configuration nuance can be found in Figure ??.

1. **Board Generation Suite** - A suite of Board Generation utilities were created that allowed easy test case building of PCB environments. This allowed ease of debugging since different PCBs could be compared against each other. It is also how we generate the test cases we use to run our method against.

2. **Board Breaker Suite** - A debug utility that allowed breaking a board in order to use it as a metric upon fixing. A **Verb-Noun action scheme** was generated to specify for the breaking (such as 'Break Vias'). This **Verb-Noun scheme** is modular and is the same scheme used in action generation for the MDP.

3. **Action Suite** - Any MDP based system will require actions. A **Verb-Noun setup** was generated that was highly modular and easily addable-to. Gaussian angular actions, move vias between layers, route actions, via-push actions, etc. were implemented. *Note that not all of these were used since this research did not explore variable action sizes for the model.* This may require further research into programs that generate programs to adapt, or into RNNs that heavily use the input to generate variable outputs. Either way, the ability to keep adding additional actions was highly considered in these board mutation verb-noun action suite. Ideas to handle variable actions were generated - such as a 'direct access array' of possible total actions - but that was deemed too poor in terms of action size. Another alternative could be to rely on existing optimizers, bringing them in as actions and working under a delauney triangle format across the board. This would allow using those mini-optimizers (i.e.
MCTS Engine Configuration Developed

```json
{
    "designDir": "UsedBoardsRules/",
    "designName": "",
    "designRoot": "",
    "boardChoice": 7,
    "breakMethod": "remove",
    "breakType": "WIREs",
    "actMethodType": [
        "move", "schedconlayers",
        "route", "wires",
        "angle_n", "vias"
    ],
    "amountToBreak": 0,
    "numIterers": 100,
    "timeLimit": 10000000000,
    "selectionType": "UCB1Tuned",
    "expansionType": "network",
    "expansionPolicy": "firstavail",
    "simulationType": "network",
    "simulationPolicy": "random",
    "simulParallel": "leaf_normal",
    "saveSimulTraj": false,
    "rolloutDepth": 40,
    "routeBias": 30,
    "numberofLeafThreads": 1,
    "storeWholeExpMap": false,
    "mainDebugDir": "boardImages/",
    "debugInitDir": "InitCond/",
    "debugMDPVisualize": "MDPDir/",
    "debugDetailedDir": "DebugDetailed/",
    "trajectoryDir": "Trajectory/",
    "dataconfigvisual": "visualize",
    "repeatRunsTest": true,
    "repeatCombinatorial": false,
    "repeatRangeRollouts": [100, 110, 20],
    "repeatRangeRouteBias": [30, 40, 20],
    "repeatRangeThreadsLeafPar": [1, 2, 2],
    "repeatRangeTestIter": [300, 1000, 250],
    "repeatRangeTestTime": [10000, 50000, 12000],
    "multiRunPerfDrive": "TestingMCTSspeed/",
    "RepeatNonTest": false,
    "NumRunsToDo": 1
}
```

Figure 4-1: These are the parameters available on the MCTS Engine, which cover anything from strategy type to tuning simple parameters and choosing actions, amongst many others.
those developed by others on my team) in this larger, more global optimization, which may allow easily merging this system to handle variable actions. The system is highly modular to do so but this was not explored further and is something good for future research.

4. Network Target Suite, Main Runner State Machine, and Environment Runner State Machine - This research was built and conducted within a system of various interconnected distributed services following a very specified state machine. This was built in full as a prototype, with services passing along data and revealing final results via gRPC for network and IPC. Thus, it was initially designed for modular network target additions. That ultimately made it easier to add machine learning models. The Environment Runner State Machine is slightly separate and is used as a server in Java for outside machine learning models to train on-policy with. It is not connected to MCTS but rather is a standalone ‘forward action runner’ for direct model usage. This gives environments in Python’s OpenAI Gym - for example - to have access to the true Java environment of the classified PCB software (but would work for any optimizer object environment as long as the object is re-pointed to and the actions re-formed).

5. MCTS Algorithm Suite - This suite is highly configurable. The ability to modify iteration count, time count, rollout depth heuristics, and route bias/action bias is just a start. It also contains easy-integration with the Action Suite, and highly modular interfaces for Selection, Expansion, Simulation, and Back-propagation (for example easily switch Simulation to use various deep RL models instead of random). This is a substantially modifiable suite with plenty of fine tuning capabilities. Selection gives choices between UCB and UCBTuned. Expansion allows swapping strategies modularly - but uses the choose first strategy and chooses to save and prune the expandable actions every move. This means we fully take advantage of the single player nature and re-use the sub-tree while saving space by pruning and saving backtracked
paths to disk. Simulation is highly modular as it uses the various different ML models and heuristic values. It is also multi-threaded and conducts parallel leaf rollouts and the back-propagation properly modifies to aggregate for the various different threads via averaging. Simulation also contains the ability to early terminate properly baked in - including nuances with dataset generation and parallel rollouts.

6. **Algorithm Details** - To create the values for the heuristics mentioned above, we had to set some expectations for the problem. For example, the **problem horizon** is expected to be on the order of how many moves are theoretically required to achieve complete synthesis of a partially complete board. In our problem, with our test case, we expect that to be on the order of 5 to 100 moves. In fact, improvement of the horizon is a category that we show in the results. Iterations and moves to episode completion are proportional. We theoretically allow it to solve for infinite horizon, but we expect a solution in this order of magnitude horizon, and this is observed. Do to our action branch size of 97 (for the problem we use for the results in this paper), we do not expect many levels of the tree to be larger than height 2 or 3 - this is discussed further in the **Amplification Strategy**. In some cases, some actions are invalid, but we always expect roughly around 97 actions. However, the depth of the **early-terminated** winner pulled from within the leaf-parallel rollouts is expected to be proportional to the horizon length mentioned prior.

7. **Debug Suite - Visual and Text** These are a versatile set of tools that allow various levels of text and image logging given urgency. These are toggle-able and can end up generating pictures of *every step in every thread* or display no logs at all. These help follow **every move made by our designed MCTS plus deep RL combination algorithm**. Heatmaps and color coded changes are shown - they can easily help find memory issues and bugs. It allows configurable directory creation for these various logged results. This may be worthy of

---

1 They actually helped uncover a pretty major one related to a history of changes not matching up with the final PCB that was formed in my work!
building into a GUI framework due to its versatility and exploration of MCTS - but that was not necessary for this research. This also has the option to generate files in hierarchies (nested) or separately - this is useful for the performance testing suite. The ability to do 'multi-runs' is also included, which allows mass producing the dataset. This may also lead to multi-objective optimization - though that was not explored.

8. **Performance Testing Suite** - Essentially an internal self-generated profiler uniquely specified for the MCTS Engine built for this research, this tooling lets us see how our experimental algorithms change overall decision making ability. It will benchmark every iteration, move, episode - even sub iterations such as expansions, simulations, and selection - and at the end produce data in chart and csv format. For optimization, the ability to use a range-like object (min, max, skip) in configurations allow fine tuning the MCTS algorithm. This was used to go from vanilla MCTS to an augmented form as it allows checking performance for fine tuning. Beyond fine tuning the augmented algorithm, this tooling allows us to see - with all else equal - how deep RL acceleration of MCTS changes the number of iterations taken to fix the partially complete PCB board across various stages of the algorithm. It is fairly robust and easy to add on another testing criteria. **It ensures that per any episode, it has 'probes' in regions of the code connecting to these specific chunks of the MCTS algorithm.** This means it always collects data - but if multi-episode performance is chosen, it is able to generate aggregate data for all those runs - which is useful to compare augmentations and their effects with more statistical reliability. See Figure 4-2 for an example of the probes results for a single game. The generation of a csv dataset allows re-displaying the data utilizing other tools - such as Python. The limitation of seamless data analysis in Java\(^2\) is the reason behind this choice.

9. **Data-set Generation Suite** - This is a highly configurable code-base which

\(^2\)Java is a bit unflattering when compared to Python's data analysis with clunkier visuals. But I'll forever cherish Java with all its quirks and objects - and you should too!
Performance Probe Results Example

Figure 4-2: This is an example of this internal profiling tool we created to measure performance - it is important to note this particular visualization is not used in our results, but can be generated by the tooling. Our results use the CSV generated data sent to python to allow better analysis - such as through Seaborn in Python as shown in Figure 4-3. However, the gist is that this tool lets us measure relative performances over code regions - like a profiler specified for this MCTS engine - and is how we can figure out the average number of iterations per episode or moves per episode and so on.

allows generating and storing CSVs of all MDPs generated as MCTS runs. All (S, A, R, S', T, etc...) data is stored in various usable forms. It handles early termination by properly combining past data with the live MDP. All MDPs are stored to a local on-storage CSV and flushed from in-memory each move. Backtracking is conducted to generate all these paths prior to flushing. Flushing requires consistent usage of Breadth First Search (BFS). It properly handles edge cases caused by some-times early termination and parallel rollouts. Tons of scripts exist outside the MCTS codebase which parse this enormous set of data and pull it into one large aggregate file (with bias-potential if we want to repeat the number of times a 'winning' path is placed in the database due to low length MDPs taking up most space (which some models cannot train from)). There is also an option to keep every single MDP in its own separate file, which depending on the offline DRL model, might be more useful. Here is an example of a MCTS-generated dataset, shown in Figure 4-4.

We have now described how the MCTS Engine was built - it is the cornerstone for this work. The above section also described added functionality and various methods and micro-services required in the design - such as training learning models across
Performance Data Saved and Redrawn by Python Statistical Analysis
(LOG SCALE)

Figure 4-3: An example of using the data-set generation suite to save the data and then the python statistical analysis suite to re-draw the performance results more aesthetically. Our results in the next chapter heavily use this strategy.

The Dataset Generated

Figure 4-4: This is the full column setup at present for the dataset that the MCTS simulation generates. It is highly modifiable and can thus provide robust data to any models. At present, this stores the MDP in the form of $S, S', A, R, T$ and additional data. The simulator can truly output any range of required data from the PCB Environment in Java. Further future work, such as placing this in databases for CI/CD, etc... is wholly possible but was not needed for this research.
processes in various programming languages.

## 4.2 Machine Learning - Python

This is a split codebase - all the machine learning work was done in Python for reasons pertaining to Cadence IP as mentioned prior. The central shared dataset directories are used to store data and models - but the technology itself is split between two languages. There are general scripts shared within the Python side outside of the Online and Offline algorithms. To see an overview of the entire feedback system - and in general the architecture in more specific terms (a general one will be displayed at the end of this chapter), see Figure 4-5

1. **PreProcessor** - A rather general preprocessor was built. This had the ability to clean any data from the Java side - either in live gRPC based IPC training or in offline dataset processing to feed into CQL. This pre processor handled all image manipulation, down-scaling and channeling (pcb layer-stacking - which was an idea that seemed to impact model learning positively for three dimensional PCBs) as well as base64 conversion operations and everything else necessary to get data from the Java side to fuse in with the Python models. All tensor manipulations, type conversions, and ring buffer data formation were done here. It was made pretty robust to allow easy changes but given the nature of preprocessing, it was not as heavily designed as its Java-MCTS engine counterpart.

2. **Heavy Scripting** - A ton of scripts were generated to setup, run and install any codebase as well as train any model. They also helped generate winners, which was a concept inspired from good offline RL practices (will be explained further below). These scripts also helped dockerize this work so that models could be trained remotely fairly easily and safely in sand-boxed containers. This helped save time, and with further development, could be fit into a CI/CD system.

3. **Data Analysis** - Considering the enormous amount of data coming in from
The Artificial Intelligence Architecture (MCTS and Deep RL Models Feedback Loop)

Figure 4-5: All models are dockerized on the cloud, with the **left** being in Java - MCTS which does the selection, expansion, simulation, backpropagation, and a forward environment runner to allow on policy training. They exist in separate processes within the container to allow simultaneous training and utilization. The dataset of the game is then automatically collected and aggregated via an *amplification strategy*, and split to be prepared for the *winners strategy* through a user called script.

Then the dataset is used to train online and offline deep RL models in Python, which may utilize the forward environment runner if online (via gRPC IPC). Finally, the deep RL models are fed back into MCTS, which then uses them instead of a random policy during simulation. This is the overall feedback loop generated for this research.

PPO, while it can be placed in this loop, is a baseline because it does not fundamentally use the data from the past iteration in a future iteration (i.e. it cannot use this loop idea since it doesn’t rely on the dataset - and thus is a good benchmark against models that **do**).
the performance probes - whether from vanilla MCTS or with all the added augmentations and machine learning models - we needed a way to analyze data to showcase patterns emerging from my experiments. This data analysis and charting suite was built to chart data specifically related to this project. It heavily relies on the dataset - both for the MDPs and the performance - generated on the MCTS-Engine Java side. It is able to do things such as replaying movies of a run from a dataset (to see the agents working in this system!), to analyzing the statistical properties and generating charts. Most importantly, it is able to feed MDPs properly into the deep RL models.

Outside of that, modifications were model specific.

### 4.2.1 Offline Algorithms Adaptation

For offline RL, we mainly focused on CQL, and its internal variants. A lot of time was spent basically re-adapting the codebase to understand what it did and use it for our PCB and MCTS research purposes. The system of sampling buffer data was understood, and then utilized with the preprocessed data to train PCB specific models. Considering that these were offline RL algorithms, the output model should be smarter than the MCTS data that trained it. In other words, this is not a simple evaluation like that you’d find in typical supervised learning. Conducting quick tests would not be so simple, and running actual live runs to test the model may end up wasting expensive time. Thus, to come up with a way to quickly test the model (and other RL models), a strategy was devised - one which we name the Winners Strategy. This was generated for CQL but ended up being usable for all models. It will be discussed below. Once a model exists, we generated a Python gRPC server to host these models so MCTS or anyone else could call the model in a distributed inter process manner - or over the internet if required. This was made to be thread-safe - multiple calls are ensured to hit an immutable set of code in its own scope when arriving at the server. This will occur since rollouts in the MCTS-Engine undergo leaf parallelization as discussed earlier. Locking is not necessary since all
leaf processes will call one function in differing scopes that share no underlying data. Possible improvements and parallelizations of the model server may further speed this process up and could be considered for the future.

The Winners Strategy

As stated above, Offline algorithms (and RL algorithms in general with a reference dataset that is only semi optimal) provide difficulty in finding time-inexpensive methods to test them before subjugating them to more time-intensive tests [11]. There are easy techniques which we incorporated, such as seeing the Q value and the advantage value during the training and testing phases (as well as during usage). But this isn’t that useful and time consuming to evaluate. So what is an ideal way to ‘quickly test’ this sort of model? Well, the idea we had was as follows - if MCTS produces a semi-optimal dataset, and we train a model on it, the model should pick up on cues that are most common (highly common features). Then, if we had just leave out a small set of data from the original sample before training, and take the winning one or two steps from each chunk of data in the sample, then it is highly likely that a learned model would also take these same steps on a never before seen state from the same environment - or at least certainly take the second step (final state). This was because those were all winning steps. This idea was generated in order to evaluate the models. This strategy may not be the ideal way of testing or evaluating models, but is a concept that was a part of the evaluation. As we will explore in the results section, this idea seemed to indicate models were ‘learning’ because it was able to be accurate even on these never before seen data test points that were close to winners. Let us define accurate here as choosing a sensible move far above the $1/N$ ratio, where $N$ is the number of actions available. The Q values should also showcase reasonable intent that this was picked, and that was also observed. It is important to note this idea worked (with respect to its main target - CQL) only due to the fine tuning below - particularly the proportional amplification of winning paths while still allowing it the freedom to see any horizon 2 or greater random path by relying on MCTS’s value already functioning as the value - i.e. a discounted sum of rewards.
Note that **we actually take the ‘final two’**, and thus if the model truly learned, it may choose the optimal action in the pre-final move (i.e. do what MCTS did not do), and we also seem to observe this. This means MCTS may have chosen, for example, **move 3 and 4**, where 4 won the game. Since MCTS is always improving, but not maximally optimal, it might have gotten away with making move ‘4’ instead of move ‘3’ in the second to last move.

A CQL model that learned the underlying state space may then make a move that doesn’t agree with MCTS, but is thus even more optimal than MCTS in the move prior to the winning move. Thus we may expect an accuracy of 50 percent or so depending on getting the final state right, and the pre-final state as something else. Therefore, in this PCB space, this strategy does seem to be a good way to identify if the offline model performs well on *never before seen similar states*. However, when applying this model to the **whole swath of variants experimented on this this research**, as we will show in the **results section**, we seemed to find the models that **had an accuracy between 5-45 percent** on this generated ‘Winners Strategy’ when compared to the **semi-optimal** validation/test MCTS dataset ended up being the ones that would be the **most intelligent**. It appears that the ones that got the ‘expected result’ on this test actually ended up choosing a good move - i.e. **route** out of **97 possible moves** - but in actual run time, would **route** far too often.

It is easy to call this over fitting, and it may be true further fine tuning could improve this, but the moves chosen **are good moves**. We cannot call this ‘over fitting’ in the traditional supervised learning sense because these models can learn from the environment or from data coming from the environment. These **are** moves that increase reward - and often quickly. They just aren’t what will win the overall game. They usually have the ‘largest spike’ in reward. So, once results are considered, practically, it appears this **winners strategy** was a form of potentially figuring out if a deep RL model was ‘over-learning’ **good** actions while missing out on actions that helped **solve the overall objective in less** decisions. The combination with the ‘last-2-rows’ from validation/test sets of MCTS - which is semi-optimal outside of the last move - may have lead to this practical result. A more exploratory model may
yet still try other moves when routing is the best move, but still route a lot of time.

Given the practical nature of this ‘winners strategy’ idea, some unexplored suggestions from advisors and peers\footnote{Huge thanks to Zee Han for these creative ideas!} after the time-frame of development suggest possibly using this strategy in finding exploration parameters for deep RL models. This might be something to explore in the future. A start to exploring this idea is shown in the results on Chapter 5. We reweigh the winners strategy results by the maximum result to better indicate the idea described above where it may be a quick ‘evaluation tool’ to determine improved performance. It is also in a form that may be more suitable to creating exploration parameters. A diagram of both the **Winners Strategy** and the **Amplification Strategy** is shown in Figure 4-6. Note that the amplification strategy was used purely to improve the offline deep RL results, and while the generated Winners Strategy was initially for quickly testing offline deep RL, it ended up being usable for all models.

### 4.2.2 Fine Tuning

Since CQL utilizes DQN at the heart, fine tuning revolved around ensuring that the bootstrap $\theta'$ (aka $\phi$) network was lagging behind the main model to ensure the distribution shift was happening at a slow enough rate to not mess with predictive capabilities. Looking up various anecdotal sources, as well as general academic insight from mentors, it appears many others in differing problems have found success with a significant fraction of the total training steps \cite{10}. Thus, fine tuning here set on the target network updating in the thousands of step range, since we used 15000 training steps per iteration (epoch), and 20 epochs (we also attempt a much larger set of training steps (40,000 steps over 100 epochs) once on a better machine but the chosen lag of $\theta'$ was left the same as it would still be a significant fraction). This was chosen due to time limits of our experimentation - with more time, it could be trained for far longer. Considering the dataset already contains ‘value’ from looking deep into the tree (in our MCTS use case), another parameter, the update horizon was set to just 1. Since many MDPs from non-winning paths were less than length
Figure 4-6: This is a simplistic example shown of how CQL may be trained (offline deep RL) via both the amplification and winners strategy before an expensive 'actual' test. Note that you can substitute any of the other experimental models after the training step. The other models only required expert offline data so they did not require this amplification strategy applied to them - they just used all the winning paths for many games as their expert dataset. The Winners Strategy applies to all models for quick evaluation.
3 (due to MCTS’s exponential action branching), computing the discounted value for longer than two horizon steps in each sample during training would disqualify many MDPs. Quite a few are likely still disqualified, but an idea to amplify the winners dataset when aggregating data from MCTS likely offsets this issue so that the model is able to learn a good amount from the winning path as well as any non-winners it can latch to. This was done by scripts that pulled the Java-dataset outputs and allowed aggregation - a quick addition to amplify the winners easily added this feature. The risk would be that without amplification, batch sizes may also never hit the winning path as it randomly samples from the more numerous non-winners. All other parameters, such as the gradient step optimizer, train step size of 4 (skip 4 before training/sampling), batch size of 32 per train step from the buffers sampled at the start of the epoch/iteration, underlying convolutional network parameters other than image tensor manipulations and stacking, etc... were left unchanged. More could definitely be done with the fine tuning to further squeeze out better models. Due to how similar all the generated test boards look, the CNN used to input our state to the deep RL process might also be getting more confused then it might otherwise.

An attempt with artificially different images (stock photos) substituted for the 'states’ appeared to find robust paths, for example (i.e. images that were markedly different from each other). Even so, this set of fine tuning (and the general set mentioned prior with the preprocessor that used ideas such as PCB image stacking), while not as thorough as it could have been, definitely seems to improve results for this problem scope markedly with CQL. Without it, the offline model would produce results that would continually take bad actions. As we explore in the next Chapter, this offline model now at least 'over-fits’ on a reasonable action - even if it is one that is only 'good’ and not the 'best'.

Quick Aside on the Amplification Strategy

Upon initial training of offline models, quickly tested with the winners strategy mentioned above, the model was choosing nonsense moves and scoring a zero. This was due to the nuance of how the CQL model batch sampled and used the MCTS
output MDP data. A particular PCB synthesis could have thousands of mini MDPs of length 2-3, and would have one winning MDP of far longer length. A methodology was designed to treat each MDP as a buffer or to aggregate all MDPs from a game into one large buffer and ‘checkpoint’ it per MDP. Testing indicated the latter worked better - but the issue was that most of the MDPs were of length 2-3. Thus, the model, while containing the reasonable fine tuned update horizon of 1⁴ - which let more paths be trainable⁵ - also meant that the model was less likely to see the winning path and truly pick up on the underlying state space (winners and non-winners both help identify the underlying space).

Thus, this idea for amplification was proposed. A percentage of the total number of MDPs produced by a game - set to fifteen percent - was chosen, and then the winning path was amplified by that number. Thus the final aggregate dataset would contain all paths - non-winning and winning in amplified form. So if a particular solution had 1400 non-winning MDPs, and 1 winner, we would end up with 221 winners (copied amplifications) and 1400 non winning paths. Since the winning path is always far longer than the losing paths, even if sub-optimal (but always in the right direction due to the guarantee of MCTS), this meant that overall the pick of the underlying state space - both winners and non-winners - would be reasonable for the model upon training and picking samples with a uniform distribution. After attempting this strategy, the winners strategy proposed above seemed to get far better results for CQL. From choosing completely random actions, it started choosing a reasonable action for an unseen state. Thus this strategy seemed to have favorably impacted the experiment with offline models in this problem space. Further fine tuning to find an optimal split could be conducted, and the chosen 15 percent may not have been ideal. This can be left for future research where this system is in a more automated CI/CD state.

⁴Considering MCTS generates values from rollouts - which already looks deep into the tree - there was less need for a larger update horizon to compute the discounted sum of rewards.
⁵Larger update horizons would disqualify paths that ‘terminated’ before the update horizon length was passed.
4.2.3 Online Off Policy and Expert Based Algorithms Adaptation

Just like CQL above, once we have a trained AIRL, GAIL, or PPO model, we can put it up in a thread-safe server for access by others. To actually train it in our system though was the interesting part. We had to generate a few things.

1. **Java On-Policy gRPC Environment Server** - This was discussed earlier with the Java software. It is a gRPC server that allows acting like an OpenAI Gym environment in Java - where it has access to the classified PCB Software Suite. Python can then use gRPC as a client to then ping the java environment server and train a deep RL model.

2. **OpenAI and Stable Baselines Environment** - On top of the proto gRPC files and communication between sandboxed docker containers, we also required a working OpenAI/ Stable Baselines environment in Python which could communicate with the models and with the Java environment runner. This was generated and connected with the gRPC IPC service mentioned above.

3. **Fine Tuning** - As with any other model, fine tuning was necessary. For example, AIRL doesn’t like differing sizes of demonstrator trajectories and differing sizes of real trajectories. It expects them to be the same size to avoid issues. The number of training epochs is usually the most important, and was set to high values - and in fact is a part of our result space (i.e. the impact of training steps to the feedback loop explored in this project is shown in our results section). Taking more steps is more likely going to impact the final performance the most because the model is able to explore moving around the environment.

**Different models have different objective functions** - which means PPO used on its own may perform worse than PPO within AIRL (as the generator), for example. These results are heavily explored in the results chapter. PPO, GAIL, and AIRL all went through the same process because they were all from the same library - stable baselines 3 (or some variant of it - in our case we use
Berkeley’s Human Compatible AI Imitation library [16]). Thanks to help from my team, we were able to spend a bit more time fine tuning this - with modifications in entropy parameters for exploration, and gamma for the horizon. Certainly more could be done in the way of fine tuning to squeeze performance out of this model as will be discussed in the next chapter. However, our fine tuning focus definitely did show some improvement and promising results for some of these online-methodology based models!

With that out of the way, we had PPO, GAIL, and AIRL ready and working within this PCB System. Thus, all that’s left is to showcase the final feedback system and the results!

### 4.3 The Learning Feedback System (LFS)

This is the heart of the concept created for this paper. Everything above leads to this system. Of course we have the default different fine tuned MCTS + Deep RL Expert/Offline result set as well, since that unique combination was also explored in this paper. (i.e. the effect of changing dataset size, or the effect of changing training iterations for the deep RL model before plugging it back into MCTS) since they were interesting in their own right. But the pieces for the LFS have been set. We know that MCTS is a policy-improvement operator - as shown in the examples discussed above, such as Go and Atari, it can help teach other models about the problem. This setup is a bit different but similar to other feedback loops utilized in similar literature.

The idea in this is that we collect a lot of data from the MCTS system for a particular board class (namely one 'type' of PCB, which may contain various variations of damage). This data is used to train a model, entirely offline, or on-line off-policy expert based, separately. Then, in the next iteration of training, MCTS no longer uses random exploration - it instead robustly calls these specifically chosen models trained on past data they generated (which may also have had an on-policy simulation - such as in PPO, AIRL’s and GAIL’s case - in the forward environment). The new updated dataset is then used to train the next set of models again and so on.
until we reach a high-fidelity set of models which can aid MCTS to reach the target faster and faster (a synthesized PCB that works!). This happens while these models are a part of MCTS’s rollout, instead of on their own. This allows them to continue to benefit from the MCTS algorithm.

The idea with this is that MCTS can act as the center-point and the models can 'give back' after being trained by it in the past (and later by a previous generation of itself plus MCTS), and this allows a feedback system that can utilize offline models and expert generation models. This sort of feedback is unique as far as we are aware in application within this PCB field (and this specific combination w/ AIRL/GAIL/CQL in MCTS is unique anywhere as far as we know), and the results are promising and will be explored in the next Chapter.

Here is the final architecture, shown at a very high level, or what was built for this research - see Figure 4-7. It goes well when also visualized with the architecture mentioned earlier in this chapter - see Figure 4-5.

---

6Stay tuned! Or I guess in this case, stay 'reading'?
The Full High Level System Architecture of this Research

Figure 4-7: At a very high level, we essentially built a MCTS engine on the Java side, alongside a forward environment runner. On the Python side, which can be run on cloud machines, we have all offline, on-line, and on-line off-policy demonstrator based algorithms and models. The central point is how the dataset and models are built - the Java codebase builds the dataset automatically via MCTS, and the Python side contains pre-processors and deep RL model generation.
Chapter 5

The Results

These results will be showcased starting from vanilla MCTS, to augmented MCTS, to finally MCTS accelerated with neural networks trained either with data from MCTS or via the forward environment runner in Java. The details of the architecture and build up to these results was described in Chapter 4 - here we mostly describe the results. If any details feel sparse, please refer to that earlier section to know what technology was developed for this research that allowed these results. We generated a particularly challenging board class (it is called "BoardSeven" - see Figure 5-1) to run these tests on. It is challenging in that it requires some horizon of thinking ahead to solve. In the final test setup, this board has 97 actions available per decision, and while a few may not be allowed due to constraint violations, it is expected to always be similar due to the openness of the board. You may notice this board has 2 layers. This is to showcase it is truly 'three dimensional'. Even if the current example problem used in this paper had reduced-actions (the 97 mentioned above) to avoid variable actions due to the scope, the optimizers can work in the three dimensional layering. We had issues with variable size actions which we address in future work. However, we did test on other simple boards (just a pin or two and a single blockade) that would not vary once more layers got involved, and the optimizer properly works across layers.

Note that this software (and the results of this research) built can work with *any* such board, one at a time. That’s why we will focus on the results for this particular
board - to give a rough fair view of the improved decision making over a potential use case. This test showcases high combinatorial spaces as the solution can be found from an enormous game-tree of moving objects, routing them, and un-routing them, amongst the other actions developed.

These results are shown in terms of iterations per episode - which is proportional to the number of decisions made for all models after Vanilla MCTS. We only show Vanilla MCTS’s results across some graphs to showcase the overall performance change from the initial policy improvement operator that was developed earlier in this research. It took a lot of development time to generate that initial software from scratch, and was the intended first milestone for Cadence. That is why it has a place in the results. The graphs are displayed in non-log form for the most part (though there may be log graphs as a supplement, especially when Vanilla MCTS is displayed). At the end, we will display aggregate data for all experimental versions in a few forms, and will specifically showcase the result with the LFS described in this project for applicable experiments. All experiments were tested with an order of 10 episodes/solutions each - mainly because a board synthesis can take quite a bit of compute time. A 95 percent confidence line is showcased where applicable across those 10 runs. The goal is to show that far better decisions were clearly made - and I believe this goal was certainly met, even accounting for variance amongst the runs.

A few experiments, specifically those without confidence bars, took too long and we only got one run of data. Some models, where after enough time clearly showed they would follow trends for these longer runs, were capped with the time of our longest run already captured. This is because they 'clutch' on to the MCTS algorithm and all decide to do a good move - route - even when they shouldn’t - therefore taking similar amounts of decision making steps to synthesize the board. This had to be done given the constraints of time left to conduct the experiments but the performance of the models choosing the 'route' action made it clear that the overall proportional quantity of decisions made would be similar. All testing was done on the same cloud machine through docker containers,
communicating to each other via IPC in the same data communication pipeline. The results, in terms of MCTS iterations, heavily focus on the algorithm’s decision making capabilities rather than the performance of the machines they are tested on, so we can be confident in our results.

5.1 Initial MCTS Results

Once the vanilla MCTS was formed and software tested, it was put to the performance test with the performance suite generated for this research. Utilizing the verb-noun based actions, simple boards, and even more complex boards were able to eventually be synthesized. The larger the pin count, the larger the time it would take. For BoardSeven, when just performance tested with the vanilla MCTS codebase, we achieved an average iteration count of $\mu = 8800$. This was already promising to see MCTS being applied successfully to synthesize PCBs, even if it is taking quite a lot of actions. This is expected due to the random nature of the rollout of vanilla MCTS.

5.2 Augmented MCTS Vanilla Results

After vanilla MCTS, significant time was spent to engineer additional configurations to MCTS’s algorithm with regards to this PCB problem. This was described in the last chapter. Given the setting described at the start of the this chapter, this data indicates that the changing performance is truly a result of these experimental tests, which gives us intuition into if the augmentations have improved the model. The results are shown in Figure 5-2. Augmented MCTS drops the $\mu = 8800$ iterations taken over an episode by Vanilla MCTS to under 1000 iterations per episode on average. It is important to note that Augmented MCTS is the baseline for the rest of the experiments (along with Augmented MCTS accelerated with PPO for the deep learning baseline) since further acceleration is done with the same settings as Augmented MCTS. Vanilla MCTS will be left in to show the improvements from
Figure 5-1: Utilizing my board builder suite discussed in Chapter 4, this three dimensional two layer board was built as the main combinatorial board to run all these tests with. It is called 'BoardSeven', and has two layers, 3 nets - each of varying size vias (see the colors - grey, yellow, and green). Each color represents a net - a group of points that need to be connected through wire-traces. Each net is separate from the others. There are L-shaped blockages (the circular, dark black objects), and the combinatorial space requires looking deep because it can’t be solved directly since there can be many overlaps.
the initial experiment in some charts - even if it is not the acceleration baseline due to the reasons mentioned earlier.

We can clearly see that augmenting vanilla MCTS causes an enormous shift in performance. However it is very important to remember that both variants take random actions in their rollout steps. In reality, these two variants are not comparable for proportional actions. This is because an iteration in accelerated MCTS has multi-threaded rollout, so many more actions are taken per iteration. However, it is a clear indication of the overall performance improvement that just these algorithmic and heavily studied MCTS augmentations can bring about for this PCB problem. From now, I will remove Vanilla MCTS from the comparisons for the most part (except a few comparisons such as with the deep RL benchmarks). Building that first software suite from scratch for Vanilla MCTS was the initial goal of this research and led to the rest of the experiments being viable, so it was an important milestone for this work. However, the results after Augmented MCTS are comparable since they have equal augmentations applied. We will briefly re-show Vanilla MCTS in the full picture of all the experiments done later towards the end to show the magnitude of change this research has seen across all designed aspects.

5.3 Augmented MCTS with PPO - Baseline Results

Once the augmented version was generated and fine tuned (i.e. we used the performance tools to find an optimal augmented MCTS parameter set), the next part of our research was to build a deep RL baseline for this experiment - augmented MCTS combined with PPO. PPO is a good baseline algorithm for many reasons. It does not use any data stored in any off-policy or off-line format, and purely learns from simulating. It is a stark contrast to AIRL and GAIL, which both use off-policy expert data and simulations. It is also the opposite of CQL, an offline RL algorithm that uses only off policy data with no simulations. However, interestingly enough, it is the generator used by both AIRL and GAIL. It provides us a starting point to really see if decision making improves. We will showcase these results with all
Figure 5-2: We can see a clear improvement in the number of iterations per episode. This is not surprising, as we have increased thread counts in MCTS and modified selection equations, amongst other changes mentioned in the previous chapter.
variations of our experiment. Modifying the epoch (known as 'iteration' but using a different name to avoid confusion) quantity will be discussed in later results. However, our baselines will be the 50 and 100 epoch trained models. You can see the results in Figure 5-3.

These results indicate that practically, **PPO with MCTS in this system generated for this research performs best with less training steps involved in training it rather than more (after a certain extent).** This result might be a quirk that connects to how 'difficult' it found the board (i.e. a different board complexity could shift these results - however, we may expect these shifts should be relative when re-comparing all these experimental variants to new boards). We see that **training 50 epochs** allowed the PPO model to synthesize what took **Augmented MCTS** around 1000 iterations per episode in under 100 iterations. When trained for 100 epochs, the models started to take 'good' moves that would not help them synthesize the board (but would provide quick 'dopamine hit' increments of reward). Thus it performs with around 400 iterations per episode in the MCTS+DRL system when plugged back in.

The outlier performance of the 50 epoch PPO model is again likely due to the model 'sticking less' to the input environment (the entire combinatorial space of BoardSeven). This is more of a limitation, potentially, of PPO rather than some incredible result of this algorithm. For example, **later we will compare PPO used inside another generative model as the generator** and see that increasing iterations does not affect the performance in the same pattern observed here. Thus, the ability to compare how well the algorithm does as it is exposed to more epochs of training in the environment may also help understand the changing ability of the models.

It is further important to discuss the training and exploration capability of PPO in this MCTS + PPO concept. Since PPO is used by GAIL/AIRL, seeing how PPO on its own will train given changes may give insight into the improvement of the other models. We see in Figure 5-4 that increasing iterations reduced the loss, but the fine tuned parameters affect the exploration. It is possible that further fine tuning
The MCTS + Deep RL Baseline with PPO

Figure 5-3: All the MCTS + PPO variants base-lined to Augmented MCTS with Vanilla MCTS for comparison. This is a log graph, and the PPO variants show interesting results - with one stand out result for 'Board Seven'.
the exploration parameters, such as entropy parameters, would continue to improve PPO. The fine tuning may require *more* exploration as the final 'blue' line does not reach as low of a loss as others. However, as we will see in AIRL/GAIL (since they use PPO as a generator!), the high iteration and entropy changes actually have value (even though the generator is also PPO). Thus this could be seen as a limitation of PPO on its own but also means these results could improve with more fine tuning. However, it potentially gives us a clear indicator that the objective functions of these other deep RL models applied in the research concept presented in this paper has improved ability.

**PPO Variants Training Loss and Exploration**

![Figure 5-4: The PPO Models show potential if more training (top) was done after fine tuning entropy. The exploration (bottom) converges to stopping - which might change with more time. However, if we see other models perform better at the same epoch counts, especially if they use PPO within their objectives, that may indicate they are improved models for this PCB space when applied with the LFS research concept utilizing MCTS. We will see an example like that with GAIL/AIRL further in this chapter.](image.png)
5.4 Augmented MCTS with Offline RL Results

As discussed, the core experiment versions after MCTS, Augmented MCTS, and Augmented MCTS synthesized with PPO are how Augmented MCTS will synthesize with Offline RL and Expert Based Off Policy RL deep learning models. We will first showcase the outcome of the Offline deep RL experiments. We determined that Offline RL was a promising approach. It is important to remember that because of the scope of this project and the time available, an enormous dataset of many synthesized boards were not generated. Offline RL performs well with more data of the state space. Our chosen offline algorithm was CQL for reasons discussed in earlier chapters. Thus our results may not be indicative of potential results of this algorithm were far more data applied to it. We describe the overall results in Figure 5-5. Further down this chapter, we will re-contextualize these results. It is important to remember that this approach may only get better with more data/time. For now, the most important details to take away are that CQL did not perform much better than Augmented MCTS, taking between 800 to 1200 iterations. However, it was able to learn to generalize to a new, never before seen PCB state and make a reasonable choice - to route. Unfortunately it got fixated on a single action - essentially over-fitting on it. We can use the term 'over-fit' as this deep RL model is purely attempting to understand the environment with off policy ring buffer data and no simulation. Once more, the nuances of the environment may be lost since we do not have all that much data due to the time frame of this research.

An interesting result can be shown for the Amplification strategy discussed in Chapter 4, generated for CQL. We notice that upon applying it, the training loss started to actually converge rather than diverge after 100,000 steps. This shows that the Amplification Strategy was at least a promising first step. Later, the Winners Strategy result showcases the improved generalization capability, which was only possible after the Amplification Strategy was conceived and implemented. Here, in Figure 5-6 is the Huber training loss before and after the amplification strategy was applied. It is important to note that the range is far different. The first, pre-
The MCTS + All CQL Offline DRL Variants

Figure 5-5: All the MCTS accelerated CQL variants base-lined to Augmented MCTS. The results are not immediately positive, but the ability to generalize was seen for this PCB space with this MCTS+CQL concept generated for this research. However it was not a strong result for this paper.
amplification, explodes to **hundreds of millions** for the loss. The second stays below **2**. There is heavy variance, but the result is clearly far different and the second model seems to learn from the data better.

**CQL - Amplification Strategy Improvement**

![Graph showing training loss before and after Amplification Strategy](image)

Figure 5-6: The ability to generalize to new states, as we will see this model became capable of, was only possible using the **Amplification Strategy**. This is the tensor-board result of training loss, first **pre-Amplification strategy**, then after. Considering the nature of offline deep RL, and the limited dataset size, this was an interesting outcome. Note that the loss range is **wildly different** after the strategy was applied, improving by a factor of $10^8$.

Other than that, the effect of increasing epochs of training, adding in more data (but at these small quantities), and applying the LFS were not able to improve the results beyond Augmented MCTS alone (i.e. random actions). It is important to note that **MCTS+CQL does not take random actions** - but rather it fixates on
the 'route' action as mentioned before. So it is different from Augmented MCTS but the ultimate decision quantity ends up being similar.

5.5 Augmented MCTS with Online Off-Policy Expert DRL Algorithm - GAIL

The first model we showcase results for is GAIL, the precursor to AIRL. The most important results are that increasing epoch count and fine tuning the PPO exploration parameter seems to have a positive effect when combined with the LFS. Unfortunately, the dataset change did not seem to have much impact, most likely again due to the limited data available given the time frame of the experiment for experts. Finally, the LFS strategy (mixed with fine tuning as described earlier) seems to have a positive effect upon two cycles of the feedback loop, more than halving the number of iterations required. The results are quickly shown in Figure 5-8.

It is important to remember that PPO is used in the generator (i.e. the actual policy decision maker) for GAIL. This automatically indicates that PPO - under the GAIL optimization function - has far better performance with equal epochs for simulation when compared to PPO alone. In terms of numbers alone, we see that GAIL varies, but can reach less than 300 iterations for an entire episode.

5.6 Augmented MCTS with Online Off-Policy Expert DRL Algorithm - AIRL

Since AIRL is the continuation of GAIL, we may expect it to perform better and generalize better on the dataset. We see this potentially true, but our results seem to indicate similar results between AIRL and GAIL. Our 'best' AIRL variant did outperform the best GAIL model, and was closely comparable to the best PPO variant. When comparing PPO with the same number of iterations, AIRL shows its objective allows its own generator PPO to perform far better in this MCTS
Figure 5-7: The trends are already visible for MCTS+GAIL in changing epochs for training, changing dataset sizes, and the LFS. Since GAIL uses PPO as its generator (the same PPO that runs on its own as the baseline) within the overall optimization, it also provides an interesting comparison to PPO on its own.
This may suggest that AIRL, in this concept, is truly better able to generalize given its objective function. Given that these are deep RL algorithms that explore the environment we would expect this to be the case as the actual state space is so enormous that exploring it all isn’t possible. In terms of numbers, we see this can reach between 100 to 200 iterations per episode on average. This is heavily close to PPO trained for 50 epochs - but these AIRL models train their own PPO generators for 400 epochs.

Thus, the fact that more training did not garner worse performance (with the SAME inner algorithm used in a different objective function) is an indicator that AIRL may respond better to further data, more training, and more fine tuning. Thus, similarly with GAIL (but with slightly better performance in its best version), these models appear to hold more promise than PPO (our DRL baseline) when used in this MCTS+Expert DRL model framework introduced in this research. They can also match the outlier performance of PPO’s 50 epoch variation within low constant multiples.

Regarding the overall effects, increasing data did not have much impact (again likely due to the small number of data we captured given the time for this research). Increasing training, unlike PPO or CQL, but similar to GAIL, improves the model, from many hundreds of iterations per episode to under 200 iterations. And the model seems to possibly have promise with LFS. It certainly didn’t get much worse, taking only about 50 more iterations. Given more statistical testing, which was not possible in the time-frame of this research, this feedback loop may yield more powerful results. The statistical testing that was completed indicates this shows promise. Since GAIL was able to perform well, this is highly promising since GAIL is a similar version of this AIRL algorithm with slightly less theoretical generalization.
The MCTS + AIRL All Variants

Figure 5-8: The trends are already visible for MCTS+AIRL in changing epochs for training, changing dataset sizes, and the LFS. Since AIRL uses PPO as its generator (the same PPO that runs on its own as the baseline) within the overall optimization, it also provides an interesting comparison to PPO on its own. Compared to GAIL, we notice similar performance but a better lower bound with AIRL.
5.7 The Result of the Winners Strategy

As discussed with the Amplification strategy, CQL was making very poor decisions until that strategy was applied. Considering that RL algorithms are not supervised, it is tough to know if they’re performing well in a supervised like fashion. Ultimately, even if CQL learns from MCTS, it can’t use MCTS dataset to ‘check’ the performance. This can apply to more DRL algorithms. Thus, this strategy was generated and described in the earlier chapter. Here, we will just showcase the result. The strategy does seem to correlate with performance - almost like an over-fitting’ vs under-fitting’ check. The results here can be compared with the models that perform generally better than others - and the pattern emerges that the models that get between 10-50 accuracy seem to generalize the best. The ones that get the expected 50-55 accuracy appear to ‘over-fit’, and the ones at 0 appear to under-fit. The results are shown in Figure 5-9 in re-scaled form for reasons discussed in the last chapter. At the very least, this can show us that some models are learning the ’right things’ - such as CQL on never before seen states - even if the ultimate performance of the model is poor. This provides indication that the model may perform better with more data, and may lend reason to do further experimentation.

5.8 The Effect of Increasing Dataset Size

We will now start to show aggregate results. Thus you will now also see baselines to MCTS+PPO with 50 and 100 iterations. See Figure 5-10 to understand this discussion. As we saw above, the effect of increasing dataset size did not have much of a result when comparing to our MCTS and MCTS+PPO baselines. We notice that increasing dataset size alone was incapable of improving the performance for any section. This may be caused because the amount of data, even if collected throughout the research period, was not enough. It is wholly possible that all these results could change with far more data. It is also important to note that other changes, such as epoch count, do improve performance. Thus, with more
The Result of the Winners Strategy - Practical Potential Deep RL 'Fit' Check

Figure 5-9: As we can see, the correlation between these models and the performance discussed around this chapter is that the models that have an accuracy between 5-50 percent generalize better. Note that we have re-normalized 55 percent to be 100 percent. Some interesting suggestions raised by others indicate this may be a potential way to test entropy parameter generation for the models that are tested. That might be something to explore in the future.
time, more generated data, and tests utilizing increased data with other parameters, we may see better results. However, for our experiments that were conducted, there was not much performance change associated with the dataset sizes.

The Effect of Dataset Size on Variants

Figure 5-10: We notice that for Airl, Gail or CQL, changing dataset size, given our limited dataset sizes, did not have much positive effect.
5.9 The Effect of Increasing Model Training Epochs and Fine Tuning (Deep RL Training Iterations)

This is not to be confused with MCTS Iterations which are the core performance metric in this research. We have thus been calling these 'deep RL iterations' as 'epochs' - one full loop of training steps. We are able to have PPO as part of the results, since we have more than just the 50 and 100 epoch baselines. The results discussed in this section are shown in [5-11].

We notice that PPO gets worse with more epochs of training, at one point taking even more iterations than Augmented MCTS with its random actions. The same issue as CQL is seen - which is strange considering PPO is an environment exploring on policy model. However, the moves PPO 'over-fits' (in quotes since this is a deep RL model exploring an environment) on is route, a move that can quickly increase reward even though it can’t synthesize the PCB on its own. Thus, this may be a limit to PPO, and its potentially less capable generalization. However, it also means that the epoch count that results in maximum performance may ‘match’ the input space’s complexity to some degree. And as discussed earlier, let us consider that PPO is used within GAIL and AIRL as their on policy generator. We can clearly see their results for high epoch counts are far better than PPO alone - even though they also use PPO within. This may indicate AIRL and GAIL in this MCTS + Expert DRL concept truly is a better algorithm that has potential to improve with more training and further feedback loops. Certainly, it would indicate those are worth another look into with far more fine tuning and compute resources allocated to them.

Both GAIL and AIRL respond incredibly positively to higher iterations and improved exploration fine tuning. Considering the time scope of the research, this could not be explored further, but what is shown is highly promising. They both appear to halve in the number of decisions taken per epoch (proportionally with the iteration count). They become comparable to the outlier PPO performance discussed prior. And unlike PPO alone, even with the usage of PPO within their objective formula, they have potential to continue improving far more with more fine tuning.
and training.

CQL does not show any improvement, a bottleneck likely attributed to the **small amount of data**. The applied **Amplification strategy**, while helpful to at least understand that ‘route’ is a good move over taking random moves, was not able to help CQL much more. Once more this is likely caused by limited data given the scope of the experiments.

**The Effect of Fine Tuning and Increasing Epoch Count for Models**

![Bar chart showing the effect of fine tuning and increasing epoch count for models.](image)

**Figure 5-11:** We notice that PPO gets much worse with more epochs and fine tuning, whereas AIRL and GAIL improve by halving the decisions taken or more. CQL does not change positively due to its ‘over-fitting’ on dopamine-hits of rewards.
5.10 The Core Result - LFS

As discussed quite a bit above and in earlier chapters, the core idea for this research was to generate a feedback loop that could allow - given far more man power and compute resources akin to DeepMind or other corporations whose core product is deep RL - a way to train models that continually improve the more they act. Our results, due to the time frame, only allowed testing two loops of this feedback loop. Each loop uses the data generated by the prior loop, and the first loop uses the data generated by random MCTS. Thus any changes or improvements would only be caused by the dataset output of the prior MCTS+DeepRL model. Thus we can be confident that any further changes we see relate to the improved or worsened output of the previous feedback loop. The results are shown in Figure 5-12.

Compared to our PPO baselines, we notice that CQL does not perform so well. This is likely due to the lack of data as has been discussed. Again, the use case of offline DRL is when there is tons of data present. The results we had regarding its ability to 'route' is promising - and over time, more data will surely be collected. Thus it may prove to be a powerful supplement to other on policy algorithms in the future, maybe in some ensemble of results. However, for now, CQL does not appear to respond to the LFS concept and was unable to get through a cycle (or would take too long) due to the 'over-fitting'.

Our most promising results are with AIRL and GAIL used within this research generated concept. GAIL shows that when training on data by a past version of an expert based deep RL model, it was able to improve in its second iteration, by more than halving (combined with some other fine tuning changes). AIRL stayed in a similar area of decision making performance, getting slightly worse by 50 iterations. However, this indicates that LFS is promising, and further fine tuning, improved dataset sizes, and computation applied to this already existing research may result in improved LFS optimization. One thing to note is the variance is quite high - and this can only be improved with further experimentation, which would require more time. These results were all relatively generated with approximately $n = 10$ tests.
The Effect of the LFS

Figure 5-12: We notice the LFS strategy explored in this paper was most promising for the Online Off-policy Expert Based deep RL models - with AIRL performing the best, and GAIL close behind. They achieve similar performance to PPO 50 epochs, while showing potential to continue improving with more training and further feedback loops with more man power and a CI/CD system applied. Notice CQL does not have a round 2 due to the over-fit of round one making it infeasible - thus we just cap its performance.
per each model (where possible), a limitation by the time available for this research. However, these results are promising, even with the tests that were able to be done. The results for AIRL and GAIL with LFS only is shown in Figure 5-13 for direct comparison.

5.11 Aggregate Results

Thus, as we have seen all the unique details of all the experimental versions, we now showcase the aggregate performance for all model variations tested in Figure 5-14. We also showcase the actual non-log improvements to showcase the overall difference from the initial Vanilla MCTS in Figure 5-15. There is also a post augmented MCTS comparison just to compare all deep RL accelerated and algorithmically tuned MCTS against just purely algorithmically tuned MCTS in Figure 5-16.

We show models that already have promising performance in terms of decision making in Figure 5-17.

It is clear that this approach is able to improve the optimization capability for the PCB space, and that the LFS seems to allow high epoch training to result in potentially improving results.

Here is a table, Table 5.1, that describes the overall results for all the models in terms of iterations per episode. This allows easily comparing the changing iterations, changing dataset size, and the LFS loop number. We also include moves per episode. This describes when a model is a baseline, as well as if it is capped to save time given the time constraints of this research. Some results, like R2 of CQL for LFS, are explicitly indicated to be capped and were not shown in the graphs earlier since R1 was unable to produce satisfactory results.

We show another table, Table 5.2, that describes the result of the Winners Strategy.
Figure 5-13: This is the LFS result showcased with just GAIL and AIRL, the most promising models used within the MCTS+Expert DRL concept introduced in this paper for optimizing PCB synthesis.
Figure 5-14: These are all the model variations tested, which have been discussed in detail above within their respective sections. This is in log scale.
Figure 5-15: These are all the model variations shown in non-log form to indicate the large difference in decision making accomplished from the start of this research to the end. All of these results were crafted based on the core MCTS Engine software generated in this research.
Aggregate Results - All Tested Post Augmented MCTS - Non Log

Figure 5-16: Only models after Augmented MCTS are directly comparable due to the algorithmic equivalencies. This shows the change from that augmented form alone in non log fashion to show the deep RL improvement from the methodology proposed in this paper.
The Already Promising Results from this Research

Figure 5-17: These are the models that made the best *decisions* - or took, on average, the least number of iterations (which are proportional to actions). These are only comparable *after* Augmented MCTS. Even so, considering we start from scratch with the Vanilla MCTS engine for this research, the improvement is quite large on a practical scale!
Table 5.1: Overall Summary of All Results

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Iter/Episode</th>
<th>Move/Episode</th>
<th>Dataset Size</th>
<th>LFS Loop</th>
<th>Epoch Train Amount</th>
<th>Further Fine-tuning?</th>
<th>IsBaseline</th>
<th>IsCapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vanilla MCTS</td>
<td>8800</td>
<td>85</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>AugMCTS</td>
<td>1000</td>
<td>4.7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>PPO 50</td>
<td>80</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>PPO 100</td>
<td>450</td>
<td>2.5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>PPO 400</td>
<td>700</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>PPO 600KL</td>
<td>600</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>100x2048</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>AIRL 1500</td>
<td>1000</td>
<td>4.8</td>
<td>15</td>
<td>-</td>
<td>100x2048</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>AIRL 5000CL</td>
<td>1000</td>
<td>1.2</td>
<td>15</td>
<td>2</td>
<td>100x2048</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>AIRL 15000KL</td>
<td>600</td>
<td>1.5</td>
<td>15</td>
<td>2</td>
<td>100x2048</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>GAIL 6000G</td>
<td>100</td>
<td>1.4</td>
<td>15</td>
<td>2</td>
<td>250x2048</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>GAIL 15100</td>
<td>1000</td>
<td>4.8</td>
<td>60</td>
<td>-</td>
<td>100x2048</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>CQL 154015K</td>
<td>600</td>
<td>4</td>
<td>30</td>
<td>-</td>
<td>40x15000</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>CQL 304015K</td>
<td>600</td>
<td>4</td>
<td>30</td>
<td>-</td>
<td>40x15000</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

5.12 Discussion

The results are promising. With the fine tuning and systems generated, MCTS and all the tested augmented forms were able to make increasingly better decisions on the test board. Further can be done - improved fine tuning, improved performance testing, the ability to handle varying action sizes - but this shows that MCTS in this space can certainly work. The effective reduction - from around 10000 iterations to 1000 by just algorithmically improving a random MCTS approach - is large, a 10x improvement in decisions made as well as in time to synthesize. Then using deep RL - particularly Offline and Online Off-Policy Expert based DRL models bench-marked with a purely online DRL model, we were able to further reduce the now comparable decision making ability - on average - by around 5-10 times to within 70 to 300 iterations per episode. Since we focus on proportional values to decisions made, this is an effective true improvement in the ability of the PCB fixing optimizer generated for this research. The MCTS + Offline/Expert Based DRL concepts in the approach created for this experiment clearly showcases improved decision making performance.

PPO on its own was able to get an outlier performance - but that may just be due to how it performs relative to the input space. In general, PPO was our baseline, but showcased that, while improving slightly on Augmented MCTS outside the outlier, it would get worse with more training and fine tuning. However, even further research could get more out of this model. It is important to note why we call the
Table 5.2: The Winners Strategy Results across the Models

<table>
<thead>
<tr>
<th>Model Type</th>
<th>WinStrat%</th>
<th>WinStrat% MaxNorm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vanilla MCTS</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AugMCTS</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PPO 50</td>
<td>37%</td>
<td>66%</td>
</tr>
<tr>
<td>PPO 100</td>
<td>53%</td>
<td>95%</td>
</tr>
<tr>
<td>PPO 400</td>
<td>53%</td>
<td>95%</td>
</tr>
<tr>
<td>PPO 400EG</td>
<td>53%</td>
<td>95%</td>
</tr>
<tr>
<td>AIRL 15100R1</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>AIRL 15400R1</td>
<td>21%</td>
<td>38%</td>
</tr>
<tr>
<td>AIRL 15400R2</td>
<td>5%</td>
<td>9%</td>
</tr>
<tr>
<td>GAIL 15100R1</td>
<td>53%</td>
<td>95%</td>
</tr>
<tr>
<td>GAIL 15250R2</td>
<td>16%</td>
<td>28%</td>
</tr>
<tr>
<td>GAIL 60100</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>CQL 154015K</td>
<td>56%</td>
<td>100%</td>
</tr>
<tr>
<td>CQL 304015K</td>
<td>56%</td>
<td>100%</td>
</tr>
<tr>
<td>CQL 1510040KR1</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>CQL1510040KR2</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>
performance with 50 epochs an outlier - other models that use the exact same PPO model within their objective formulations - i.e. GAIL and AIRL - perform far better at higher epoch ranges. Thus, PPO on its own shows weaker generality compared to AIRL and GAIL, but performs better with far less training likely due to how it approaches the input space. The more it learns, its clear that it starts to 'over-fit' (in a RL sense of the word) in a way that AIRL/GAIL avoid while using PPO within them.

To compare with the baseline on-policy PPO algorithm, we were able to then test AIRL and GAIL, two demonstrator based off-policy on-line deep RL models. We had expected AIRL to outperform GAIL and PPO within the same number of epochs, because of their internal architecture. Our results matched this initial expectation for the lower bound and for higher epoch cases. The 'smartest' AIRL experiment out performed the 'smartest' GAIL experiment ('smarter' in the sense of numbers of proportional decisions made per episode). High epoch trained AIRL experiments far outperformed their PPO counterparts at that epoch range. However, both performed very well compared to Augmented MCTS, offering on average a 2-4 times improvement, and at best nearly a 7-9 times improvement. We found that increasing the number of experts (dataset size) did not have an effect on both. This is likely due to the small amount of data generated for this research given the cost per generation and the time scope of the project. We found the LFS strategy to potentially be successful as GAIL improved significantly and AIRL hovers around an already 'low' region. It is entirely possible that more compute and manpower as well as a continuous iterative/continuous deployment system upon the processes generated for this research could lead to the LFS - especially with AIRL and GAIL - to make an extremely powerful feedback operator that could continually learn the more it sees for any particular board. It holds very promising results for the few cycles we were able to generate during this research time frame.

Lastly, moving on on to the offline only models, we discovered that our winners strategy seems to be an efficient way to evaluate the models to some degree. It might be biased since they all are 'near-winning' states, but they certainly give a
good degree of information that the model is able to predict the right move on a never before seen state even with 97 total actions available on the BoardSeven class. This also seems to showcase the model is able to out-perform the semi-optimal MCTS step right before the winning step. When we applied CQL to MCTS for the main feedback concept of this paper, we unfortunately did not see an improvement. The effect of increasing data, increasing epochs, and the LFS lead to slow results that were not promising, and some results had to be capped due to the over-fit nature of the moves. The moves chosen were always 'good' moves (even if non optimal) however, and there is certainly promise. Offline RL is highly useful under context of enormous data available, and that wasn’t true for this experiment. This test does showcase the model is able to learn and pick up information without ever seeing a simulator for the PCB space, and it may lend reason (especially with the Winners Strategy result) to further test\textsuperscript{1}.

Lastly, fixating upon just the LFS, we did not find promising results with CQL (though that is not true for its potential ability to choose good actions under limited data - that was promising). However, with AIRL and GAIL, our expert based off-policy online models, we have results that - at high epoch counts - seemed to indicate that the feedback loop has promise. Both hover around exceptional (relative to Augmented MCTS) decision making capabilities, and improve past their PPO equal epoch trained counterparts - even with PPO being a part of their GAN structure.

Thus, with all this accomplished, the initial desired outcome was achieved. Augmented MCTS accelerated with demonstrator based on-line algorithms such as AIRL and GAIL perform well and can increase ability to synthesize PCBs better. We found that PPO alone was good enough to improve on augmented MCTS (even with our limited fine tuning and less computation/manpower than that used to produce AlphaGo, for example). Base-lined to PPO, with similar epoch counts, we found GAIL and AIRL did better with AIRL beating out GAIL for the lowest decisions recorded

\textsuperscript{1}Companies always keep getting more data right? So this deep RL class should only continue to increase in popularity!
on average to finish an episode (synthesize a PCB). We found the offline model, CQL, was not capable of accelerated feedback loop improvement at these small dataset sizes but was capable of generalizing to ’good’ actions. It also didn’t react much to changing dataset sizes or iteration counts - most likely again because the size of the overall dataset was small given the timeframe of this research and the cost per generating the data for a synthesis. We also found that for AIRL and GAIL increasing the our dataset size/experts did not improve the performance by much - again likely due to the small dataset size. However, increasing epoch count heavily improved performance, unlike for our baseline PPO, the latter of which seemed to achieve results ’clinging on’ to the input space complexity. Finally, we found that AIRL and GAIL showed potential to be capable of accelerated feedback loop improvement. All these results certainly indicate these deep RL models are worth pursuing in this PCB space and showcase that they are truly able to result in increased performance on average - even accounting for high variance.

5.13 Future Improvements

The project was initially side tracked due to the requirement of building a mock system environment within which MCTS could run, due to the random timing of my entrance into the team (the team hadn’t yet built the official system). Thus a lot of the initial time upfront was spent generating a fully working mock system, and then making MCTS and all of its augmentations as a highly configurable engine on top of it. This mock system ended up performing as expected and solved the problem with MCTS augmented and was capable of doing what was initially desired. However, it is still possible that a true system - once built - could reduce latency, and increase performance far past what this research worked with. The full potential of the MCTS engine generated and augmented in this work may not have been fully visible. This was combated by measuring performance in terms of ’decision quantity’ - but latency and low level performance improvements would turn decision quantity directly into time to solve improvements. Thus, combining this work with an optimized final
system may yield in far better results even with the exact same MCTS engine. Luckily, the codebase built for this research was heavily generated with modularity in mind so this should be possible or at least easy to extend from. Further improvements within my MCTS engine could also likely be done from a performance engineering standpoint. Anything ranging from further parallelization, more machines to help train the models, more machines to generate the dataset in parallel, and so on could be applied to the existing codebase and existing system to continue to maximize performance and speed of training and results (i.e. applying resources similar to that applied by DeepMind for their AlphaGo model training [5]). In fact, to further test the potential of this research, I think a system of that scale should be applied to run a CI/CD version of this LFS tested in this research to see its performance after many millions of runs.

Moving on to the machine learning side (the latter half of my project), a good amount of time was spent on working with CQL and figuring out how to pre process for all these various libraries and tools. Some models, after learning to pre process, were straightforward to use (such as Stable Baseline3’s AIRL, PPO, and GAIL) once the underlying theory was fully mapped to this project. For others, a deep dive was required into the system to implement major chunks for personal adaptation (namely CQL), on top of the already time consuming pre processing. This already limited time to fine tune these models. At a few points during this work, I lost access to essential machines and services due to logistical errors that understandably can arise in such strange global times. To finally top it off, during the final month of the project, I, along with my family, got hit with the COVID virus which caused a week or two of hardship. Luckily, after an extension that I am grateful for, these results were able to be gathered alongside starting the final semester of school. Thus, while the results have explored the promising preliminary results, and continued adding to them with more clarity, certainly more fine tuning is possible. Even so, the fine tuning that was achieved in the available time did clearly have a positive impact on the results. The additional results gathered in the final month of research continued to add on to the promise of this research as shown in this chapter.
There are certainly more ideas that have not been pursued in this paper. One is the idea of changing action sizes. A tough concept for machine learning models which train to predict a certain range of discrete outputs (action sizes), this might have unique solutions in the future. An idea to make this like a direct-access-array and pre-fix pieces of actions to the maximum limit would result in too large of an action space. My mentor for this research, Luke Roberto, suggested possibly utilizing NeRF (Neural Radiance Fields) in some way - I think this sort of research is exciting and worth a look into for the future. Besides being able to work with variable action sizes, another possible addition is to use graph networks - which almost seem to naturally represent PCBs - instead of images. A brief foray into this and discussions with my mentor revealed this might be an entire project on its own. Also, while we explored CQL for the offline model, and ARL, GAIL and PPO for online models, there have been some new offline demonstrator based models which could also be looked at and implemented. Another discussion raised by lab-mates indicate behavioral cloning may prove to be a more effective benchmark than PPO, so that may be worth exploring. In essence, there is still a lot more that can be attempted.

Certain ideas from this paper may yet expand to other fields - and ideas from other fields can expand into the work here. Any way the problem is diced, this paper solves an optimization problem. And as mentioned in prior chapters, changing the environment and adding auxiliary development allows optimization ideas to be reused to a variety of novel problems. One interesting such examples is the field of **Vehicle Routing** - an optimization problem seen commonly in applications like Uber or various other counterparts. This specific problem is known as the **Vehicle Routing Problem, or VRP**. At a very high level, the goal is to find optimal routes for multiple vehicles - essentially multiple traveling salesman problems (TSPs) - with added constraints. In fact, with just one vehicle, it becomes the TSP with constraints. Thus, it is a hard problem. It is very important to recognize there are parallels between the PCB environment and the VRP problem space. If we generalize what PCBs are, we can see it is equivalent to optimizing multiple TSPs with constraints that optimize electrical properties amongst others. This is because
we wish for our different nets to all connect optimally across the three dimensional combinatorial space of a PCB - and to do this for all different nets in an optimal way. The constraints may differ but optimizers can change how constraints are approached. A recent paper in the VRP space has found a way to speed up the VRP problem for large numbers of cities [12]. This can be seen as directly related to PCBs with large numbers of components and pins, such as motherboards. The goal for the VRP is to ensure all the different vehicles optimally route with constraints - and for PCBs the goal is to optimally route different nets optimally given constraints. The way this recent paper achieves this for VRP is to iteratively and automatically split the VRP space into subproblems, and delegate a subsolver to each subproblem. Through considering spatial locality, they also reduce the number of subproblems to a linear count. If this sort of approach were to be applied in the PCB space, it may provide interesting speed ups and improvements that parallels what it has already proven to provide in the VRP space. Thus, this is yet another avenue that is definitely worth exploring - especially in applicability to our PCB space. Unfortunately, this was not able to be done in the present work given the time constraints and I think it is worth exploring in the future.

However, given what was accomplished, we have a system that is capable of optimizing and fixing boards with the examples attempted - which was exactly what was desired. Augmented MCTS, heavily designed in this research, was able to achieve the initial desired results in reasonable amounts of actions. The combined system as it currently stands, in both Java and Python, with MCTS and various Deep RL networks - truly indicate that this sort of feedback-system can solve this combinatorial problem in faster and improved ways. At the very least, this shows that this is the right direction to take, and with the improvements mentioned above, that might be made even more apparent than it already has been in this paper. The system is all here and built from this project, as are a lot of results that showcase the value of this work. However, a lot more can continue to be done.
Chapter 6

Conclusions

In the ever-important world of PCBs, we described how we designed a system that is capable of fixing up a partially complete PCB through MCTS and Augmented MCTS (and fix simple toy boards from scratch). Then, through experimentation with Deep RL, we worked with an Offline DRL Model (CQL), as well as on-line, off-policy demonstrator-based DRL Models, AIRL and GAIL. We also base-lined with PPO, an on-line, on-policy DRL Model as a benchmark. These models were all used to accelerate MCTS, and not on their own. From comparing all of them, we expected that AIRL would outperform GAIL and PPO and learn the ‘intent’ of synthesizing the board - which we’d want to see through improved demonstrator samples increasing performance by reducing the decision quantity of the MCTS Solver after a few runs - in a feedback loop. We also would expect the offline DRL model, CQL, to do the same, if the quantity of data is sufficient. We were able to do this research via generation of a Python-Java system, where Java contained a highly robust and modularity-focused built-from-scratch MCTS engine and all other software suites related to it - alongside the classified Cadence PCB environment. Python held all the machine learning, and gRPC was used in client-server pairs between the two sides for many processes and data passing.

We discovered that this approach can solve what we were looking to do from an augmented MCTS level within a reasonable decision count for a simple board (∼ 1000 action proportional iterations). With additional deep reinforcement learning methods,
we saw the baseline PPO on-policy model could further improve performance, with an outlier performance ($\sim 80$ iterations) that in context is not as interesting as it may look. With the same limited fine tuning, we saw on-line off-policy demonstrator based models like GAIL improve upon PPO, and AIRL improve upon GAIL and PPO, under equal epoch conditions. Both AIRL and GAIL improve under more epochs (unlike PPO), and show potential under the LFS, which is why they are the most **promising result** from this research. With high epochs (above 200), they can achieve consistent iteration count of around $\sim 100 - 200$ iterations. They effectively show promise on our hypothesis about their generality since they appear to improve the more you train - unlike PPO on its own. This is further reinforced by the fact that the models utilize PPO as their generator - the exact same inner model - but get superior results. However they do not seem to improve with more dataset count. Finally, we saw the offline CQL model was able to generalize well to a 'good' action (thanks to strategies developed to train and test it, such as the Amplification and Winners strategies) but was unable to get great results over the LFS system nor from changing dataset size or epoch count. A potential reason for increasing dataset size not improving performance is due to the count - the dataset size is small relative to the action space of the entire 'solve-tree' of the PCB. More data might help improve AIRL, GAIL, and CQL, and that was not explored due to time constraints in this research.

The overall research - especially AIRL and GAIL - lead to signs that the LFS may hold promise over far more iterations than tested. This is a good sign that the direction this research has walked in for this field is the right one. Given the pandemic and some understandable logistical errs of various kinds during the research, some time was lost that could likely have led to improved fine tuning that could lead this this already working system squeezing out even more performance. With more optimizations in performance, as well as newer actions and strategies, this sort of approach could be improved far further, and I think is a great next direction for the upcoming future. From variable action sizes and improved fine tuning or newer deep RL models, more manpower and a CI/CD system built mimicking this research
proposed LFS, to an improved feedback system, the utilization of graph networks for PCBs, or even applying other optimizers in similar optimization spaces like VRP (even if wildly different fields), there's plenty more research to be done that could add on to the already promising results shown in this paper.
Appendix A

Tables

Table A.1: List of Approaches Taken in this Paper

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCTS Vanilla</td>
<td>Non Deep RL</td>
</tr>
<tr>
<td>MCTS Augmented</td>
<td>Non Deep RL</td>
</tr>
<tr>
<td>MCTS Augmented + PPO</td>
<td>Deep RL On-Policy</td>
</tr>
<tr>
<td>MCTS Augmented + CQL</td>
<td>Deep RL Offline</td>
</tr>
<tr>
<td>MCTS Augmented + GAIL</td>
<td>Deep RL Off-Policy Expert</td>
</tr>
<tr>
<td>MCTS Augmented + AIRL</td>
<td>Deep RL Off-Policy Expert</td>
</tr>
</tbody>
</table>
Appendix B

Figures and Additional Details
B.1 AIRL’s difference from GAIL

AIRL and GAIL rely on the same GAN formulation. However, GAIL has rewards that are heavily shaped by dynamics. This means that the actions taken are incorporated into the reward function. There is a concept called rewards shaping that is a large part of reinforcement learning. This is the general form of reward shaping:

\[ \hat{r}(s, a, s') = r(s, a, s') + \gamma \Phi(s') - \Phi(s) \]

This is just saying that the true reward is related to the predicted reward by some function of the difference between the next state and the present state and allows the optimal policy that generates these rewards to be invariant. AIRL is able to decouple the reward function by removing this reward shaping reliance on the dynamics (aka the actions taken to form the trajectory). Without this decoupling, a change in transition actually breaks the policy in-variance that the typical reward shaping equation provides. Thus, even with similar end goals, modified start positions and middle-states would confuse the model’s process. Decoupling this removes the reliance on the dynamics, which then allows the goal to be achieved even from varying start positions.

\[ Q^*_{r,T}(s, a) = Q^*_r(s, a) + f(s) \]

This format, where \( r'(s, a, s') \) is disentangled from the dynamics, means that for all dynamics possible (all possible trajectories, T), the optimal Q function relates to the reward with regards to state only. Thus, these state only rewards can be disentangled and are more robust to changing states. It is important to note that the true reward function may actually be smoother with actions involved - while still being disentangled - because the true underlying reward function is more related to state and action rewards. AIRL does not account for this, as it always assumes the underlying decoupled reward is only state parameterized. If a state-action reward were decoupled, it could still work amongst varying dynamics (due to the decoupling).
while accounting for the true underlying reward space. This was argued by a paper for Variational AIRL[15]. For the purposes of our use case, however, utilizing AIRL alone was sufficient to experiment with, as PCBs have a general final state (‘routed’ or ’solved’), and can be broken in various ways and start points. The state only reward function, and therefore the more robust opportunity to reach the same end goal from various start points, was an alluring feature of AIRL alone. Further experimentation may be desirable on this topic within the field of PCBs, however.

B.2 MCTS Augmentations and our Use Case - A Quick Summary

This paper modifies MCTS with various augmentations in recent literature, taking into account that we are playing a single player game - a tough constraint optimization with no adversary. This ensures that termination and state finding are improved, and performance is also improved with parallel searching. Given our system situation and use case, the MCTS will be running within a network based environment of distributed machines, and our solution also works around our use case. The system was built to be highly configurable and with many tools from the ground up. From debugging utilities at every step of the algorithm (visualizers, debug loggers), to the ability to easily add and modify actions and the various steps of the algorithm, there’s plenty to tune. There are performance testing suites that generate performance evaluations, as well as a ton of other fine tune-able heuristics. The dataset generation was highly featurized as it generates Markov Decision Processes, and there are abilities to run multiple solves of the board. Additional features exist to ‘break’ PCBs on purpose to further stress test the system - and a suite of 'board generator utilities’ was created to easily make examples to use. Since this ended up being a highly distributed system, with many pieces communicating over IPC/RPC, heavily configurable network targets were also generated.
B.3 IPC between Java and Python - A Quick Summary

The background of the system meant that the PCB environment was in Java - thus MCTS - with all of its features - was generated in Java. In order to train deep RL models in Python, we utilize gRPC to host Python and Java client and server pairs (one of each for both) such that we could train on-line models (Java Server, Python Client) and use any existing models from within MCTS (Java Client, Python Server).
Bibliography


