Linköping University | Department of Computer and Information Science
Bachelor Thesis, 18 ECTS | Cognitive Science
Spring term 2021 | LIU-IDA/KOGVET-G–21/031—SE

Exploring feasibility of reinforcement learning flight route planning

Axel Wickman

Tutor: Elisabeth Zetterholm
Examiner: Agneta Gulz

LINKÖPING UNIVERSITY

Abstract

This thesis explores and compares traditional and reinforcement learning (RL) methods of performing 2D flight path planning in 3D space. A wide overview of natural, classic, and learning approaches to planning is done in conjunction with a review of some general recurring problems and tradeoffs that appear within planning. This general background then serves as a basis for motivating different possible solutions for this specific problem. These solutions are implemented, together with a testbed in form of a parallelizable simulation environment. This environment makes use of random world generation and physics combined with an aerodynamical model. An A* planner, a local RL planner, and a global RL planner are developed and compared against each other in terms of performance, speed, and general behavior. An autopilot model is also trained and used both to measure flight feasibility and to constrain the planners to followable paths. All planners were partially successful, with the global planner exhibiting the highest overall performance. The RL planners were also found to be more reliable in terms of both speed and followability because of their ability to leave difficult decisions to the autopilot. From this it is concluded that machine learning in general, and reinforcement learning in particular, is a promising future avenue for solving the problem of flight route planning in dangerous environments.

Contents

1 Introduction **1**
- 1.1 Project . . . 2
- 1.2 Purpose . . . 3
- 1.3 Delimitations . . . 3
- 1.4 Background . . . 4
  - 1.4.1 Planning in nature . . . 4
  - 1.4.2 Classical approaches . . . 5
  - 1.4.3 Machine Learning approaches . . . 7

2 Method **8**

3 Implementation **9**
- 3.1 General development environment . . . 9
- 3.2 Program structure . . . 10
- 3.3 Environment . . . 12
  - 3.3.1 World generation . . . 12
  - 3.3.2 2D Map . . . 13
  - 3.3.3 Rigid-body physics and aerodynamics . . . 13
- 3.4 AI models . . . 15
  - 3.4.1 Autopilot . . . 15
  - 3.4.2 A* planner . . . 17
  - 3.4.3 Local DQN planner . . . 19
  - 3.4.4 Multi-Channel Memory Buffer planner . . . 20
- 3.5 Plan quality metric . . . 22

4 Results **23**
- 4.1 Performance . . . 23
- 4.2 Strategies . . . 26

5 Discussion **29**
- 5.1 Planning performance . . . 29
- 5.2 Future investigation . . . 30

6 Conclusion **32**

References **33**

Software references **34**

Division of labor **36**

List of Figures

1 *Physarum polycephalum* 5
2 Threading model 11
3 Generated world example 13
4 2D Map 14
5 Fighter jet's aerodynamic forces 15
6 Autopilot example 16
7 Multi-Channel Memory Buffer planner 21
8 Planners' static reward plots 23
9 Planners' pilot reward plots 24
10 Plots of initial distance versus execution time 26
11 A* planning example 27
12 Local DQN planning example 27
13 Multi-Channel Memory Buffer planning example 28
14 Multi-Channel Memory Buffer refining 28
15 Example of Local DQN training reward 30
16 Example of Multi-Channel Memory Buffer training reward 31

List of Tables

1 Performance of planners for different world sizes 25
2 Planner execution time for different world sizes 25

1 Introduction

The arrival of innovative machine learning (ML) techniques is presenting new methods for solving old problems. For problems such as planning, traditionally handcrafted algorithms are well suited to constrained and well-known domains, but are often hard to use within more complex environments and action spaces. One of the main reasons for this difficulty is that it is fundamentally hard to capture the complexity of the real world in a simulation, accurately enough for it to be used to make decisions that are efficacious when applied in the physical world. Small errors accumulate over time, making real-world agents slowly drift away from the intended trajectory. Adding complexity can help make the model more accurate, but this also increases the number of states that the system can be in, which in turn causes combinatorial explosions in many kinds of classical decision-making algorithms. The number of possible things that can happen simply becomes too many to be computable using practical limits on memory and execution time.

Artificial neural networks have helped present a solution to this. Because feed-forward neural networks have been mathematically proven to be universal function approximators (Csáji et al., 2001), this means that they all (given a sufficient number of neurons) have the capability of computing functions that mimic, for example, real-world systems. This capability is however only theoretical and is limited by our ability to pick out fitting parameters in the neural network. In other words, training the network, and doing inference with the network, are two separate problems.

Supervised learning methods train models by using a set of known inputs, feeding it through the neural networks, and comparing the computed output to a corresponding set of known desired outputs. This is possible since the whole inference step through the neural network is completely differentiable, which allows for changing the values of the parameters depending on their partial derivatives computed with respect to the difference between the output and the desired output. Calculating these gradients is called backward propagation, and using these to change the parameters to make the outputs converge, is called gradient descent. These kinds of supervised learning methods have known since the 1980's (Rumelhart et al., 1986), but saw widespread adoption and innovation in the 2010's as a result of increased parallel computing capabilities, abundant data, easier access through open source, and the realization of how effective it was at capturing the dynamics of complex systems in comparison with handcrafted methods.

One major shortcoming with this technique is however the need to know the right answer of which the models should produce during training. Supervised learning is for example suitable for classification (like machine vision and speech recognition), prediction (weather forecasts, failure prediction), and generative models (text and image generation). Within fields such as robotic control and game-playing, supervised learning falls short. This is because, even though we might know if the outputs of the model are desired or not, we do not know specifically what those outputs should be in order to be desirable. For example,

if one has a robotic arm and wants it to pick up an object, one knows that it is good if the output of the model leads to the object no longer being in contact with the ground (we can arrive at this conclusion by using simulation or a physical arm), but we do not know what specific movements should be executed to lead to this outcome. In this setting, one can instead use Reinforcement Learning (RL) algorithms. If supervised learning is like studying to learn the right answers for an exam, RL is more like teaching a dog to do tricks by giving treats as rewards. Fundamentally, the goal is to have an algorithm where outputs that are good are reinforced and become more likely to be produced in the future, and where undesirable outcomes are selected against. This can be done by using Q-learning, which has the model estimate the value of each action for every given state (OpenAI, 2018), or by using Policy Gradients (Peters, 2010), which learns to directly map between states and actions. Also needed is some mechanism of exploration. This can for example be done by every now and then doing something random, by adding noise to the chosen actions or simply by rewarding exploration itself.

Previously, many RL algorithms were limited to discrete state and action spaces. This limits their use to discrete problems, or forces the programmer to introduce assumptions about the world when discretizing continuous systems. In addition, the discrete representations are vulnerable to combinatorial explosions, which limits the practical size of the search space. However, by applying the mathematical theory behind Q-learning to neural networks, it has become possible to create agents which can generalize, and in turn, have the capability of acting in continuous state spaces. By using Policy Gradients, also continuous action spaces can be achieved.

Famously, Deepmind's Deep Q-Network (Mnih et al., 2013) could learn to outperform humans in many Atari games despite only having as input the pixels of the game. However, problems arose for games where the times between actions and subsequent rewards are too large. In general, long-term dependencies have presented a problem, and even though there has been progress within this field, RL still isn't the go-to method for problems such as planning for the future.

1.1 Project

This project was done on behalf of SAAB Aeronautics, and executed by Axel Wickman as a candidate thesis together Erik Örjehag at Dyno Robotics, during a 7-month period. How we divided the work between us can be found in Appendix A. The aim was to explore whether or not RL algorithms can be used as a viable method of solving the specific problem of pre-flight path planning in dangerous environments, as well as to compare the results with more classical methods of planning. Because the project should be unclassified, this meant we were unable to use previous work by SAAB, and needed to develop both the simulation environment and the algorithms from the ground up. This, in combination with the goal of the project being investigative, meant that there was relative freedom in the way to approach the project's aims.

The goal was to take inspiration from multiple sources when designing the algorithms. The planning methods make use of traditional grid search, as well as classical RL for neural networks. With the Multi-Channel Memory Buffer (MuMB), the aim was to integrate knowledge from within multiple fields of planning. The motivation behind using RL was to construct algorithms that are flexible enough to be applied also outside of this specific domain, that avoid making assumptions about what specific strategies should best be used, and that leverage the capabilities that these kinds of models have to generalize. For simplicity's sake, and to facilitate GPU (graphics card) computation, this method is also grid-based. By making this grid persistent and having it act locally, it helps to solve the problem of retaining information during the time that the plan is being refined, and also helps with explainability.

1.2 Purpose

This is an initial investigation that aims to explore the feasibility of using RL techniques for flight planning. In the process of doing this, I also aim to create a flexible framework that may be used by SAAB in the future for further exploration and development. These objectives mean that the research questions will be somewhat loosely formulated and general, as their purpose is to provide general guidance of appropriate future directions. To do this, I want to explore how well the algorithms do, where they falter, and their general behavior. I believe the following research questions treats these aims:

- How well can the implemented RL planning algorithms rival a hard-coded approach in terms of performance?
- How well do different planning algorithms function depending on search space size, in terms of execution time and plan quality?
- What strategies can be observed as being used by the different algorithms?

1.3 Delimitations

Much time was spent on developing the simulation environment, which left less time for developing and tuning the rest of the algorithms. The benefit of this is that the environment and general frameworks are constructed abstractly and efficiently enough to be used for future investigations. The aim wasn't to develop fully optimized, state-of-the-art algorithms, but rather to investigate the feasibility of different methods of solving the problem.

Algorithm-wise, I only focus on off-line planning (i.e. before flying starts), and planners that use grid-based input (although some can produce continuous output). Although some planners easily could be converted to produce 3D paths, I choose to have them all produce 2D paths (through 3D space) in order to keep all results comparable. Use is made of multi-threading and (single) GPU computation, but the on-policy planner has to run single-threaded with a batch size of one, for collecting the data.

1.4 Background

1.4.1 Planning in nature

In their book "*The cognitive psychology of planning*" (2005), Morris and Ward compiles bodies of knowledge about planning from multiple fields, and then draws parallels between the discoveries made. Much of the work of understanding planning in humans has been done by observing the behaviors of AI algorithms, and as a result, early models of human planning often contained notions of searching state spaces. This systematic approach is described as top-down planning, and involves very coordinated and deliberate thought processes. As opposed to this, there are bottom-up models, suggesting planning in humans is more opportunistic and that decisions are made as a result of conditions that happen to emerge, rather than planned in detail ahead of time. What seems clear is that both planning ahead and dealing with things as they come, have their place depending on the situation (pp. 37-38). Which strategy that humans tend to employ depends on multiple factors, such as if the problem is predicted to be easily solvable by perceptual means when it arrives, if there is a requirement to be able to verbally motivate the decisions made, and if there is an importance placed on speed or accuracy (pp. 41-42).

One of the main constraints that both humans and AI need to take into consideration, is that of the limitation of working memory. Bottom-up planning seems to involve lower-level brain functions, which doesn't create as much cognitive memory load as top-down planning does. This is for example evidenced by people being unable to verbalize during high workloads in tasks that require a high-level executive function (pp. 43-44). Knowing what work to do how and when is therefore vital in order to conserve both the resources of working memory, and time. This planning skill is learned, and the fact that there are individual differences in performance between different types of planning tasks, suggest that it involves many different brain function (p. 128). As with most executive functions, planning seems to involve large areas of the brain - with emphasis on the prefrontal cortex, which seems to play some role in sustaining information over time (pp. 192-193). This specialization of function therefore might lead one to believe that the process of planning is required to be highly centrally controlled. However, evidence exists to the contrary.

Physarum polycephalum (Figure 1), or Slime mold as it is usually known, is an acellular amoeba that is famous for its ability to make decentralized intelligent decisions despite not having any nervous system. The creature forages for food using protrusions (plasmodial tubes) extending with multiple concurrent heads into an unknown environment. This results in a dynamically evolving network of tubes that can be up to hundreds of square centimeters in size. These tubes first explore and then optimize to be the thickest along the shortest path. This gives the organism the ability to solve mazes (Nakagaki et al., 2000), optimize with respect to risk and food quality (Latty & Beekman, 2010), and perform speed-accuracy tradeoffs (Latty & Beekman, 2011). This last ability is important since the process of collecting information to base decisions on,

Figure 1: *Physarum polycephalum* growing on a tree trunk (Stoen, 2011).

is costly. This means that the organism likely would miss opportunities if it had a strategy that would search the environment too thoroughly, but on the other hand, would miss nutrients if it didn't search thoroughly enough. This kind of tradeoff is similar to the top-down/bottom-up tradeoffs we humans face in our decision-making. Just like humans, the disposition can be influenced by external conditions. We cannot directly tell *P. polycephalum* to think carefully, but it is possible to add stressors such as light and hunger. For example, for harder tasks, the mold will prefer to act quickly if there is a hazard present (Latty & Beekman, 2011, p. 544). A reasonable assumption would be that this kind of behavior comes from deep and complex internal mechanisms. However, while much of how *P. polycephalum's* functions remain unclear, research (Alim et al., 2017) has suggested that the internal rules which enable this kind of distributed decision making are in fact relatively simple, and that the complexity of behavior is an example of emergence.

1.4.2 Classical approaches

The field of classical pathfinding has existed since the 1950's and relates strongly to the field of computational search in general. Dijkstra's algorithm (Dijkstra, 1959) solves the problem of finding the shortest path between one node (i.e. state) and another, in a graph where the edges can have different weights (costs).

The algorithm initiates at the starting node and works its' way outwards in order to find the goal. Successively it updates the values of the connecting nodes along the way with the cumulative value of their shortest route to the start and does so by always choosing to next explore the node which has the smallest of these values. This results in an algorithm that is complete, which means that it will find a solution (if one exists) in a finite amount of time, and optimal, which means that it is guaranteed to find the solution with the shortest existing path.

A drawback of this approach is that is blind. The only heuristic it uses while conducting the search is that the shortest path so far is the one most worth exploring. Since the act of exploring the path, lengthens it, this can in many cases result in the head of the search moving across many different fronts, which makes the algorithm inefficient both in terms of execution time and memory usage. In addition, in order to use this algorithm in continuous environments (like the real world), it needs to be discretized. In most cases, this means turning the world into a grid. This makes it impossible for the algorithm to navigate at a more precise scale than the scale of the grid. Making the grid size smaller (or world bigger), increases the number of grid-points by the square in a 2D world and by the cube in a 3D world. Also, a state often consists of more than a location in space, and an action of more than a movement in this space. If the agent should be able to, for example, attack targets in the environment, this means that the computational complexity will increase vastly with both the increased number of possible actions, and with the states (which has to keep track of all the combinations of which targets have and haven't yet been destroyed).

Being relatively general, there has been a number of improvements to Dijkstra's algorithms, which take advantage of the known specifics of the problems in order to be more efficient. Likely the most famous of the grid-based algorithms is A* (Hart et al., 1968). This relies on a heuristic function that depends on the problem. This heuristic represents the estimated future cost to the goal from a given node, which helps to guide the search by allowing A* to pick the most promising state for exploration. The algorithm is always complete, and optimal as long as the heuristic never overestimates the actual cost to the goal. Often times it is sufficient to use a distance metric (such as Euclidean or Manhattan) as a heuristic. For other actions than spatial movement, what is and isn't a fitting heuristic however requires a deep understanding of the problem dynamics.

Sampling-based planners are another solution to motion planning. Here, a random sampling of the space is used as a first phase in order to build a connectivity graph, which can then be quickly queried for paths. If the random sampling happens close to the already explored state, this allows for sampling even in infinite domains. This method often provides better speed and less memory usage than search-based algorithms in large search spaces but comes at the tradeoff of completeness no longer being guaranteed within finite time, although a solution is guaranteed to become more probable with time (Elbanhawi & Simic, 2014, pp.2-3). This kind of algorithm is an example of the usefulness of randomness. Somewhat counter-intuitively, random algorithms can sometimes

outperform deterministic ones, owing to the fact that their internal processing, with many samples, converges to desirable results and thus aren't as likely to perform poorly due to worst-case scenarios (Kleinberg & Tardos, 2005, p. 708). For example, if a sampling planner unknowingly enters a dead end, it will still likely sample other places too, and thus be able to recover due to not putting all the eggs in one basket. Furthermore, random algorithms can often be advantageous because they may be simpler conceptually, easier to implement, and because they don't need to use as much memory (p. 708). The problems of getting stuck in local minima, the high computational cost, and (for some algorithms) having to craft problem-specific heuristics, still remains also for sampling-based planners.

1.4.3 Machine Learning approaches

Long-term memory has long been a challenge for many areas within ML. Even though some progress has been made in retaining information through recurrent mechanisms, such as LSTM's (Hochreiter & Schmidhuber, 1997), and by using attention mechanisms like Transformers (Vaswani et al., 2017), to filter large amounts of information, strategic planning using an end-to-end machine learning approach has proved more difficult. Early success was however found within ML approaches in combination with explicit traditional tree search algorithms. Maybe the most famous of these successes was that of AlphaGo (Silver et al., 2017), which learned to play to the game of Go better than humans, in spite of the very large search spaces. In this game, a neural network is trained using RL and tasked with predicting the value and probability of doing a given action depending on the current game state. This information is used by a Monte Carlo tree search algorithm, which can predict how good a move will be with consideration to the sequence of the likely future moves that will entail. This works by the search repeatedly taking a random combination of possible moves into the future, weighted by the networks appraisal, and then choosing the path which tended to give the best results overall. In essence, this gives the algorithm an imagination allowing it to look into the future. By choosing random paths and working probabilistically (similarly to sample-based planning), large state spaces could be understood well without being exhaustively searched.

However, this algorithm relies on hard-coded knowledge of the ways the system can dynamically develop. This problem was solved by integrating an old idea into a new method of planning. The Dyna-Q algorithm (Sutton, 1991) uses an external model, which takes as input a state and an action, and predicts the next state. This allows "hallucination" into the future, which oftentimes is cheaper than interacting with the actual environment. This can be turned into a planning algorithm by combining it with traditional search (Oh et al., 2017). By first letting the model encode the state to a latent representation, and then tasking the model with predicting the resulting latent state representation, a Monte Carlo tree search can be performed.

Gradually, the learning modules in the architectures of the planning algorithms, became more and more important. One problem that remained was

that the actual dynamics of planning still were hard-coded (albeit stochastic). The algorithms could learn to leverage the structure of the search by changing the latent representation, but they could not learn the appropriate structure of planning by themselves. The previous algorithms were also model-based in the sense that they were explicitly trained to understand the dynamics of the environment. However, computational capability and mechanisms that increase the capacity of neural networks, have since improved. As a result, there has been a shift back to model-free methods (Guez et al., 2019) where the mechanisms of planning are implicitly approximated within the model as a means to an end of solving the task, and is made possible by higher informational capacity. This allows the algorithms to develop more generalizable, and thus flexible models, which also have the capability of overtime iteratively improving the solution.

2 Method

The development of the program was done in milestones, after which we presented our progress to SAAB. For the first milestone, a development environment was chosen, created the world generation, implemented the rigid-body physics, and added 3D visualization with GUI and a top-down minimap. For the second one, the A* planner was added together with, aerodynamics, input argument parsing, an ML-model workflow, and implementation of an RL autopilot was begun. The implementation of the autopilot was finished during the last milestone, along with the implementation of the Local DQN planner, the MuMB planner, and the evaluation metrics.

On startup, the program generates a random world. This world consists of a heightmap terrain, no-fly zones, air defense systems, and a fighter jet. Either automatically or manually, a goal position is then selected. This queries the selected planner for a plan between the fighter jet's position and the current goal position. Once this plan is computed successfully, it is fed into the autopilot (if enabled). The purpose of having an autopilot is to give an indication of the plan's feasibility. The pilot will do its best to blindly follow the plan, and will do so either until the goal is reached, or the fighter jet crashes into the ground.

The overall performance metric that is used to compare the different planning algorithms, consists of a static evaluation of the plan, and the reward value acquired after each planning and flying episode. The reward is used both as a metric for performance analysis and as a signal to train the planning algorithms. The value is calculated by summing the autopilot's average ability to follow the flight plan, and other aspects of the planned route, such as flight time, amount of time in hazardous areas, and time performing of airspace violations.

3 Implementation

The following section details how the program described in the method was implemented. The first step was to setup an environment environment, which consists of choosing and installing all the necessary dependencies for general program development, ML-model execution, and rendering, as well as tools for version control and debugging. These choices then influence program structure and flow, which is discussed next. Since performance is of key concern, much effort is put on efficiency in terms of not wasting resources and parallel execution. In this so far relatively general framework, a specific simulation world then is then implemented. What this environment consists of, how it is randomly generated, and how it's physics are simulated, is explained. Lastly, the implementations of the intelligent algorithms are detailed. These are the autopilot and the different planners. The autopilot is an RL algorithm and thus functions similarly to some of the planners, but does not follow their implementation structure due to having a different functional role. This role is to provide insight into the followability of the plans generated, which it does together with the plan quality metric, explained last.

3.1 General development environment

To simplify the setup of development environments on multiple machines, use was made of multiple Docker containers (Version 18.09.1; Docker, 2021) based on Ubuntu (Version 20.04; Canonical LDT, 2020) with Nvidia CUDA (Version 11.1; Nvidia, 2021) and OpenGL (Version 3.2; Khronos Group, n.d.) support. Since Windows support was a priority, only dependencies with cross-platform capabilities were used. Because of the importance of computational performance in order to minimize training time and maximize the amount of data collection, the choice was made to use C++ (Version 17; ISO/IEC JTC1, n.d.) as the programming language. Being a compiled language that gives relatively low-level control, enabled us to write efficient programs. As opposed to Python, which is often used for ML applications, C++ offers significantly lower execution times (Tamimi, 2020), arguably richer support for Object-Oriented Programming, as well as multithreading support. CMake (Version 3.13; Kitware, 2018) was used for build automation. On Linux, compilation was done with GCC (Version 8.3.0; Richard Stallman, 2019), and on Windows with MSVC19 (Version 16.8; Microsoft, 2020). When using Docker, both compilation and program execution were done inside the container.

LibTorch, a C++ binding to the popular PyTorch (Version 1.8.1; Facebook's AI Research lab, 2020), was used as an ML framework. LibTorch brings with it a very large array of tensor operations, which can vastly be accelerated through the use of the library's GPU computation capabilities. This acceleration is made use of if the hardware is available, and as a fallback, the computation can instead be performed on the processor. Since I had previous experience with PyTorch, it was possible to speed up development by making use of the incompatible TorchScript-format. This allowed for first programming the models in Python

(Version 3.8.5; Python Software Foundation, n.d.) and then importing them into the C++ code during runtime. In order to visualize the progress of the training of the ML models, use was made of a C++ port (Version 570c297; RustingSword, 2020) of the Tensorboard client and added a Docker-container that launches the server (Version 2.2; Tzu-Wei Huang, 2020). This allows us to check in on training progress across the internet in real-time.

GLFW (Version 3.3; The GLFW Development Team, 2021) was used to instantiate and manage windows and OpenGL contexts. OpenGL is a cross-platform standard that allows efficiency for 2D and 3D graphics rendering using the GPU. It is managed from within C++, but also requires shaders, which are runtime-compiled programs written in GLSL and executed on the graphical device. I also made some direct use of the CUDA API functions in order to transfer data from OpenGL to LibTorch without the extra step of having to copy the data back and forth to and from RAM. The library NanoGUI (Version e9ec8a1; Wenzel Jakob, 2019) was chosen for creating GUI elements, because of it's OpenGL compatibility. JSON files were loaded using JSON for Modern C++ (Version 3.9.1; Niels Lohmann, 2021), and arguments were parsed using Argument Parser for Modern C++ (Version 1344889; Pranav Srinivas Kumar, 2021). Physics were simulated using ReactPhysics3D (Version c273e7c; Daniel Chappuis, 2020).

As for other external tools, git (Version 2.20.1; Junio Hamano, 2021) was used for version control, CLion (Version 2020.3; Jetbrains, 2021) and Visual Studio Code (Version 1.56; Microsoft, 2021) as IDE's, GDB for general debugging (Version 9.2; The GNU project, 2020), and Valgrind (Version 3.17.0; Julian Seward, 2021) for checking for memory leaks.

3.2 Program structure

The decision was made to use object-oriented programming, which means that the structure of the program is almost exclusively derived from classes that are instantiated into objects. This has the advantage of keeping the program highly structured and modular. By using abstraction and templating, I tried to keep interfaces and structure general, as to help with flexibility in case there is a need to further develop the program in the future.

One big structural challenge was to enable the program to function both in single-threaded headed mode and in multi-threaded headless mode. In headed mode, a single-window is launched and everything is visualized for the user. In headless mode, no visible windows are launched, and all program output only happens through the terminal (with the exception of Tensorboard logs, and saved models). Since rendering using OpenGL has to be done also in headless mode, it was needed to initialize OpenGL contexts regardless. The easiest available way of doing this was by using GLFW to ask the operating system to launch hidden windows. Given that it is not uncommon to use up to 1000 threads, and every thread needs its own context, this approach brings with it performance concerns and is a potential point of improvement for the future.

For efficiency's sake, all renderable objects in the scene have a function to

initialize rendering. This function only gets called in headed mode, meaning that loading 3D meshes and textures to RAM, and uploading this data from RAM to GPU, only is done if and when actually needed. This data is also handled statically, which means that all instances of a specific renderable object, share this data (and shaders), which minimizes memory usage. It is also possible to disable features that aren't needed for a given simulation, such as the map, air defense systems, or Tensorboard logging, by using program input arguments.

While some features can be toggled off and on in the GUI during runtime, full control of the program state is given during launch through use of the terminal. Settings such as the thread count, whether to restart on crash, how often the world should be randomized, what planner to use, and if training should be done, can all be set. Given a large number of input parameters, and the need to often repeatedly use the same inputs, configuration files were added. These are JSON files, and the data in them are treated as program arguments. It is possible to import configuration files recursively, with the shallower arguments overriding the deeper ones. This means that one can have one specific world configuration defining world size and terrain, and then have multiple dependant files that configure for training the pilot, training the planner, and doing evaluation, for this specific kind of world.

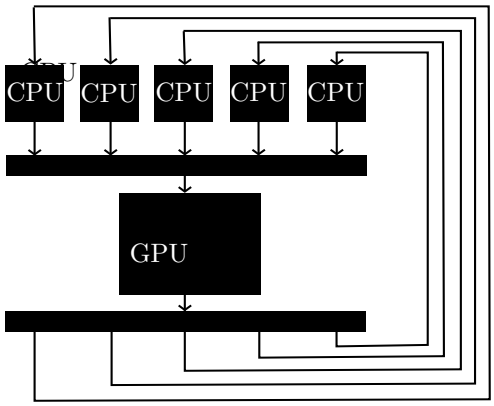


Figure 2: Threading model. Computation is done in parallel in threads on the processor (CPU). When needing to compute using LibTorch, all threads synchronize. The last one to arrive collects the data, issues the commands to the GPU, writes the result in all other threads, and then releases all the other threads.

Parallel processing can vastly speed up computation. Most computers have concurrent execution abilities in the processor, and in the GPU. The processor threads share RAM memory which each other, but cannot safely read and write to each other's memory without risking corruption. This problem is solved using thread synchronization. By using tools like *atomics* and *mutexes*, threads can perform safe data manipulation, and wait for each other to finish certain tasks. The disadvantage of synchronizing too much is that threads will a long

time spend time waiting, decreasing the efficiency of the program. One way to alleviate this is to minimize synchronization. Therefore all world generation, physics simulation, and rendering, happening separately in each thread with no interaction. Another way to increase efficiency is to increase the number of threads. Why this works is slightly counter-intuitive, given that a processor often only has a handful of cores. Yet, a greater amount of simulation seconds per real-world second is observed when using threads exceeding the core count. I believe this is because every thread takes a slightly different amount of time to execute, meaning that for 4 threads (on a 4-core system), if one is slightly slower than the others, they all have to wait for it during synchronization. If however there are 1000 threads, and one core is a bit slower, the remaining three cores can start computing the rest of the threads immediately without synchronizing.

As for GPU computation in LibTorch, data is processed most effectively by the processor issuing few, but large, workloads. In the threading model (Figure 2), all the threads are therefore concurrently ran on the processor, have all threads wait for each other to be done, let the last one to finish read data from itself's and the other thread's memory, tell LibTorch to perform GPU computation on this data, write the computed data into the common memory, release all the threads, and repeat. This loop happens for every AI update (generally about 15Hz simulation time).

3.3 Environment

3.3.1 World generation

World generation happens with the help of pseudorandom number generation. All generation is based on a seed, which is the value that the random number generator bases the returned values on. It is possible to regenerate a world identically by setting the seed to be the same as before. Since the world usually needs to be different on each generation, the seed is set to be equal to the current time.

The first step in generating the world is to get a random number from the number generator to seed a 2D Perlin noise generator. Perlin noise (Perlin, 1985) is a common technique used within computer graphics to create procedural random textures. Unlike pure noise, there is covariation within the pixels, resulting in a cloud-like image. By changing the frequency of this noise, and overlaying multiple weighted layers of it, the look of the texture can be customized according to the project needs. By interpreting the output value of the noise as a height value, a dynamic terrain is created. Instead of trying to mimic realistic mountains and valleys, I choose to construct more maze-like structures with a clear distinction between high and low altitudes. Perlin noise was also used to generate no-fly zones (allowed, not allowed) and territories (ally, enemy, disputed). In this case discretization is performed by thresholding the output values. This process generates a 2D grid, which is then projected into 3D space using scaling factors. This is results in worlds such as the one in Figure 3.

Air defense systems are scattered around the map and serve as hazards

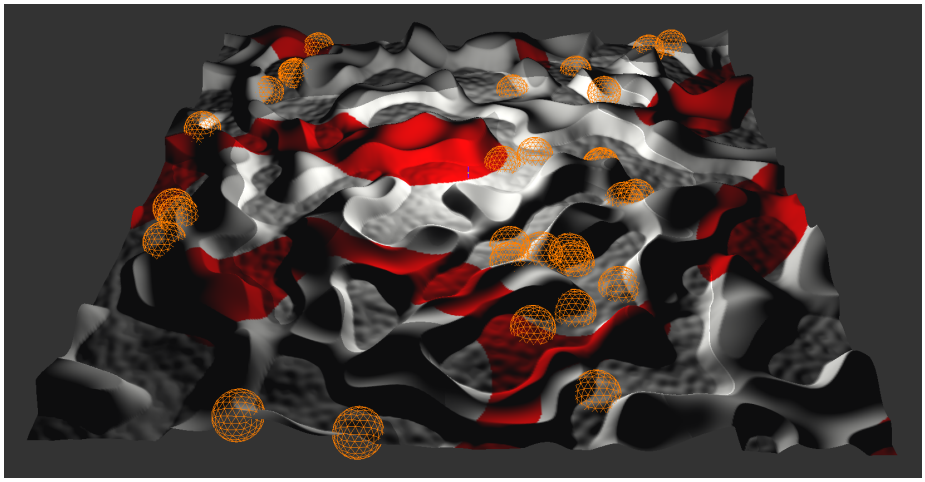


Figure 3: Example of generated world with size $500x500x10m^2$. Red areas are no-fly zones, and orange spheres represents reach of air defense systems. Territories aren't visualized, but causes air defense systems to group.

within a certain range. They generate most commonly on enemy territory, and sometimes on disputed territory, which makes them a bit more clustered. They all have a hard-coded reach distance of 150 meters.

3.3.2 2D Map

For the purposes of visualization and as input to the planning algorithms, it was decided to implement a top-down 2D map (Figure 4). This map is rendered in OpenGL and contains terrain height, no-fly zones, territories, air defense system positions, and the current plan. The map is always centered on the fighter jet and is rotated in such a way that the forward direction in 3D space becomes the up direction. The map is rendered every AI update, or when planning starts, and can be configured in multiple ways. For the purposes of simplifying computation for the neural network, it has a mode where different information is segregated into the different color channels (red, green, blue, alpha).

3.3.3 Rigid-body physics and aerodynamics

The physics system uses the library ReactPhysics3D. This library handles Newtonian simulation of objects, including forces, collision detection, collision handling, and ray-casting. Because the distances are big, not much effort was put into modeling accurate collision boxes. Instead, the fighter jet is encapsulated by a single bounding box. Conveniently, a height field primitive was already implemented, which could be used to enable collisions with the terrain. Air defense systems have no colliders.

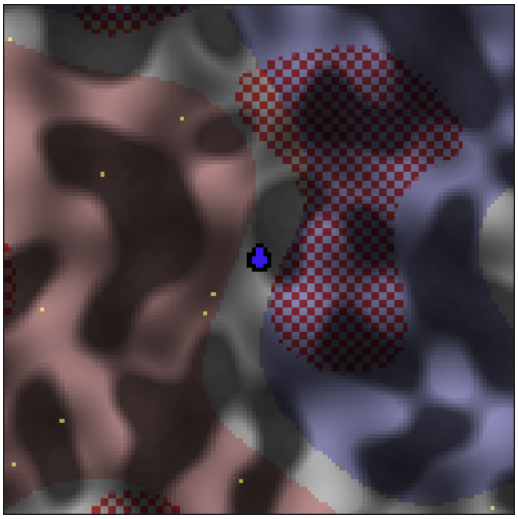


Figure 4: Rendered map with human readable colors. Center marker represents fighter jet and is always oriented upwards. The gray lightness corresponds to the height of the terrain. Yellow dots are air defense systems, and the red checkerboard pattern is no-fly zones. The blue transparent area is ally territory, red transparent is enemy territory, and uncolored disputed territory.

However, there is no native implemented support for aerodynamics in the library. This could instead be achieved by computing the forces of wings manually at every physics update and then telling ReactPhysics3D to apply the forces at the corresponding locations. This was done by defining a new type of physics primitive; an aerodynamic surface. These contain information about their chord and span, relative location and rotation, flap area percentage, skin friction, and stall angles. For the most part, the math behind this kind of simplified aerodynamics model consists of relatively standard and simple vector and quaternion operations. The most challenging variables to find are the coefficients of lift, drag, and torque. These vary during flight depending on the angle of attack and are often calculated experimentally. Since experimental determination was out of the scope of the project, I instead choose to use an approximation method developed by Khan and Nahon (2015). This method, originally used to model toy airplanes, makes use of thresholds and sinusoids to create functions that liken real coefficients in situations of normal flight, at low angle of attack, and at stall. The method also has support for making a fraction of the surface controllable. This enables us to manually (or automatically using the autopilot), control the plane exclusively by modifying the angles of the control surfaces. The airplane's wings are modeled using 5 aerodynamic surfaces (Figure 5).

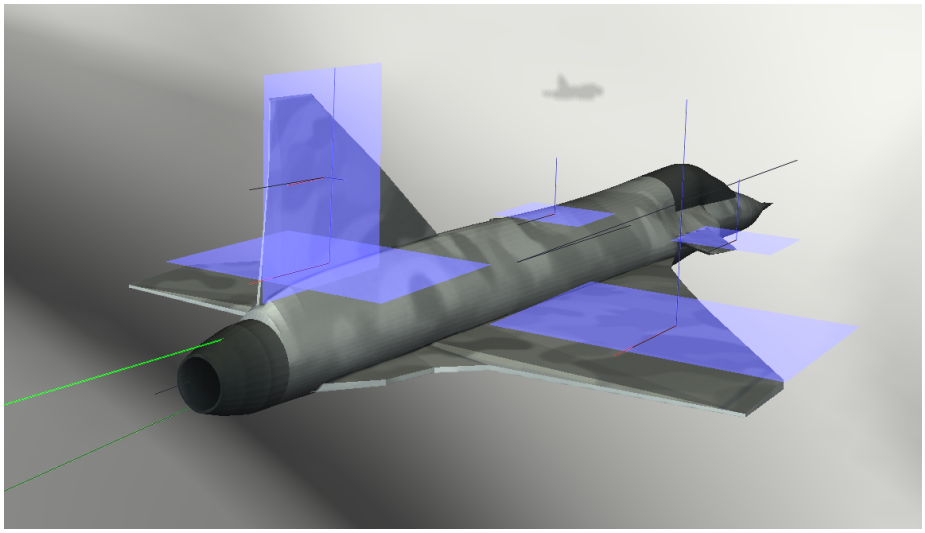


Figure 5: Fighter jet flying with aerodynamic forces visualized. The black lines are drawn from the center of mass forwards, and along the direction of movement (linear velocity). The black line coming from the engine represents current thrust force. The blue transparent areas are aerodynamic surfaces. Each of these have a red line representing drag force, a blur representing lift force, and a black representing air velocity.

3.4 AI models

The AI algorithms are all divided into three parts: the invoker, the reinforcement learner, and the model. The invoker is the algorithm that makes use of a specific AI model. These are the autopilot and the planners. The reinforcement learner is an algorithm that does thread synchronization, data accumulation, Tensorboard logging, and takes care of the training of the model (turning on and off gradient retention, setting optimizer, defining loss function, etc.). There are a couple of reinforcement learner implementations, which will be explained in detail later. The model is defined in PyTorch, converted to TorchScript and then imported during runtime. This defines the actual structure (a neural network), computation and learning parts of the algorithm. Ideally, all these three components would be interchangeable, but creating structure abstract enough to do this was deemed out of the project's scope.

3.4.1 Autopilot

The role of the autopilot is to constrain the planners to generate practically followable plans. It takes as input a current state, and produces a vector of bounded numbers corresponding to the thrust of the engine, and the angles of all flight control surfaces. It is trained using an off-policy, model-free and

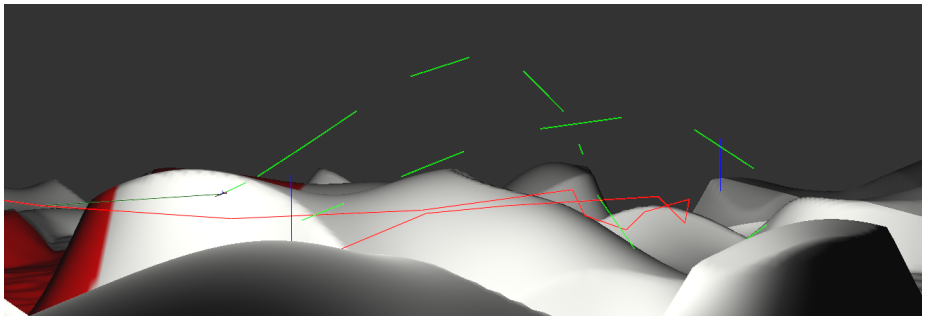


Figure 6: Autopilot steering fighter jet according to a plan generated by the random planner. The blue lines represents the start and the goal, the red line the plan, and the light green the already flown path. The dark green line is drawn between the fighter jet and the next waypoint to be reached.

continuous action-space RL algorithm called Normalized Advantage Function (NAF), invented by Gu et al. (2016). It is not the state of the art performance-wise, but served well as an initial RL algorithm to implement and develop a workflow around.

In short, the algorithm works similarly to Deep Q-learning (explained below), but with the constraint of having to train on a continuous action space. The structure of the reinforcement learner is here closely connected to that of the policy. Like many other RL algorithms, the value of a given action (a) in a certain state (s) is separated into two terms:

$$Q(s, a) = V(s) + A(s, a)$$

Where $V(s)$ can be seen as representing the average value of being in a certain state, and $A(s, a)$ being the advantage of a certain value compared to others. A trick in NAF is to define the advantage in such a way that it depends on the current policy, in addition to the given action (performed by other policy). This makes the only way to have $Q(s, a)$ converge towards the reward r, to have the policy converge to the optimal action.

In addition to this, two other techniques are used. For one, instead of training on data that has just been collected by the policy, state transitions $(s, a, r, s', a'$, where prime symbol represents the variable's subsequent value) are stored in a replay buffer. This memory is sampled randomly for every training step, which makes the algorithm off-policy (as opposed to on-policy). This has the benefit of being able to train using bigger batch sizes, as well as getting a greater diversity of samples for every training step. The second technique is called double Q-learning and means selecting the action, and evaluating the action, is done by two different, but often structurally identical, networks. Gradient descent is then performed on the evaluation model, after which the policy model has its parameters changed towards the other model's with a certain percentage. This causes the policy model to "lag behind", and helps with the problem state's

values otherwise tending to be overestimated.

The NAF is invoked by the pilot, which is responsible for gathering inputs, applying outputs, and returning a reward. The pilot starts to function after being given a plan. This plan consists of waypoints, which are locations in 3D space. Every time step, the pilot evaluates if a waypoint has been passed (if it is closer than 150 meters, or if the subsequent waypoint is closer) and if so moves the target waypoint forwards in the plan by one (as seen in Figure 6). The inputs to the model consist of the fighter jet's current speed, and 3-component vectors of upwards direction, angular velocity, linear velocity ($\vec{v}$), delta to next target waypoint ($\vec{\Delta}_{wp}$), and delta to the second next target waypoint. The last three of these are made relative to the current orientation through a change of basis operation. All outputs are constrained between -1 and 1 using a *tanh*-function. The reward function r is defined by the following:

$$a = \vec{\Delta}_{wp} \cdot \frac{\vec{v}}{\|\vec{v}\|}$$

$$r_{direction} = \begin{cases} a & a > 0 \\ 2a & a < 0 \end{cases}$$

$$r_{time} = -\log(t_{wp} + 1)$$

$$r = C_d\, r_{direction} + C_t\, r_{time} + C_p\, r_{passed} + C_c\, r_{crash}$$

Where t_{wp} is the time in seconds it took to pass the waypoint from the previous waypoint, r_{passed} represent whether a waypoint just has been passed (1) or not (0), and r_{crash} if fighter jet has crashed with the ground. The following coefficients to function well: $C_d = 1$, $C_t = 0.2$, $C_p = 1$, and $C_c = -2$.

Since the planners depend on the autopilot when training, the autopilot can't depend on plans generated by them to train. Thus, a random planner was implemented. This creates a path with a random offset, and a somewhat persistent but changing speed. The autopilot then tries to follow this plan until it crashes into the ground, in which case the simulation restarts.

3.4.2 A* planner

A* (Hart et al., 1968) was chosen as a traditional planning method to compare against. As mentioned earlier, it is grid-based and relies on a heuristic function to guide its search in an appropriate direction. It makes use of so-called dynamic programming, which means that it solves the problem of finding its way to the goal by progressively computing solutions to subproblems.

In this case the heuristic function $\hat{h}(n_c)$, where n_c is the current node, is defined as the 2D distance between the node and the goal. This estimates the cost of getting to the goal. The cost function, $g(n_p, n_c)$, is the accumulated cost of getting from the start the current node n_c, given the previous node n_p, and is defined as the sum of the $g_{value}(n_p)$, the distance between the nodes, the cost of

striking an air defense system if this was done, and the number of air defenses within reach times the cost of being near them.

Using these, the following function can be defined:

$$f(n_p, n_c) = g(n_p, n_c) + \hat{h}(n_c)$$

The update rule for the current node's g-value and f-value are as follow (where the prime symbol represents the subsequent value):

$$g'_{value}(n_c) = \begin{cases} g_{value}(n_c) & g(n_p, n_c) > g_{value}(n_c) \\ g(n_p, n_c) & g(n_p, n_c) < g_{value}(n_c) \end{cases}$$

$$f'_{value}(n_c) = \begin{cases} f_{value}(n_c) & g(n_p, n_c) > g_{value}(n_c) \\ g_{value}(n_p) + \hat{h}(n_c) & g(n_p, n_c) < g_{value}(n_c) \end{cases}$$

The first step is to add the starting node to an ordered queue called the *frontier*, which sorts values low-to-high depending on $f_{value}(n)$. Since the starting node doesn't have a previous node, let $g_{value}(n_{start}) = 0$, and $f_{value}(n_{start}) = \hat{h}(n_{start})$. In addition to this, the node is marked as being open. This constitutes the initialization, and is succeeded by the following loop:

Open the node with the lowest $f_{value}(n)$ in the *frontier*, and remove it from the set and unmark it as open. Check if it is the goal state, and if so, break the loop and reconstruct the path. Else, find all the node's valid neighbors. Loop through each neighbor, and update their $g'_{value}(n)$ and $f'_{value}(n)$, and if $g(n_p, n_c) < g_{value}(n_c)$, store the previous node as the origin, and add the node to *frontier* marked as open. This process continues for a maximum of iterations equal to the total number of world grid points.

It is when defining a node's neighbors that the possible actions of the fighter jet are defined. This includes 2D movement in a 3x3 (with the middle position excluded) square, and is only possible if the node is above ground and not in a no-fly zone. In addition, the agent can choose to attack any air defense system within reach, which results in it being registered as eliminated.

For efficiency's sake, the states have a hash function defined. This function expresses the state as a 64-bit hashmap (a list of zeros and ones), which can be used by the computer to easily sort and lookup states. Because of the size of the hashmap, a situation might appear where two different states are expressed the same. If this becomes the case, a function manually (and slowly) compares the states. The layout of the hashmap depends on the world size. The first bits must represent the possible locations and thus has a length of $\lfloor \log_2(W_x * W_z) + 1 \rfloor$), where W is a vector of the world size. The rest represents the combinations of alive and eliminated air defense systems.

The following would be a hashmap for a world size of $W_x = 500, W_z = 500$:

$$\underbrace{0000000000000000}_{\text{locations}}\underbrace{0000000000000000000000000000000000000000000000000}_{\text{air defense system statuses}}$$

3.4.3 Local DQN planner

Deep Q-Networks (Mnih et al., 2013), or DQN's, is a kind of RL algorithm applied to neural networks. They have the ability to learn from high dimensional input, and can by using convolutional layers, learn to act on visuals. This made makes DQN suitable for building a planner which can navigate using the map as input. Because I wanted to compare the difference of having only local knowledge versus having a global overview would have on performance, I choose this technique to create the local planner.

The idea behind DQN, and other Q-learning methods, is to create a function approximate that estimates the value of performing any possible action for a given state. It is then possible to perform the action which is predicted to give the highest future reward. This means that the method inherently only supports discrete action spaces. In addition to this, there is a certain percentage chance ($\epsilon(i)$, where i is current iteration) of picking a completely random action, which is the mechanism of exploration. The estimate of the action-value function, $Q(s, a)$, can be made to converge towards the optimal policy as $i \to \infty$, by updating it with a *value iteration* using the Bellman equation:

$$Q'(s, a) = \mathbb{E}\left[r + \gamma \; max_{a'} Q(s', a') | s, a\right]$$

Where r is the reward, γ the *discount factor*, and $\mathbb{E}$ the expectation operator. This means that the new estimate produced by the action-value function depends on its own estimate of the highest possible reward return given a certain action and state. In practice, however, this function is not updated directly but rather made to converge towards the new value. For neural networks, this is done much like a standard supervised problem, where a loss function is defined (Smooth L1), the model is backward propagated, and the parameters of the network updated using an optimizer (in this case RMSProp). A replay buffer is also used for training.

The input state to the algorithm consists of two tensors; a vector representing the relative coordinates towards the goal and of the distances to the air defense systems, and a zoomed-in, color segregated 2D map centered around the current position. This is fed into the model, which returns a vector representing 6 actions. The model feeds the vector through two fully connected networks with a hidden size of 32. The map is fed through three convolutional layers, and then two separate fully connected layers. These outputs are concatenated, and then put through three more fully connected layers which output values for all the actions. Action 1-5 representing various degrees of turning, and action 6 represents attacking the closest target. When planning, these actions are fed into a simple, virtual agent which can move around the map in 2D. This virtual agent returns the next state together with reward:

$$r_j^{distance} = 1/(0.001 d_j^{goal} + 1)$$

$$r_j^{threat} = \sum_{a \in ADS} max(0, exp(1 - d_j^a) - 1)$$

$$r_j^{attack} = \begin{cases} 1 & if\ attacking\ and\ in\ reach \\ -0.01 & if\ attacking\ and\ not\ in\ reach \end{cases}$$

$$r_j^{height} mean(m_j^{height})$$

$$r_j^{no-fly} = mean(m_j^{no-fly})$$

$$r_j^{total} = C_d r_j^{distance} + C_t r_j^{threat} + C_a r_j^{attack} + C_h r_j^{height} + C_n r_j^{no-fly}$$

Where d_j^{goal} is the distance to the goal, ADS are a set of all air defense systems, and d_j^a is the distance to a given air defense system. The variables m_j^{height} and m_j^{no-fly} represent matrices of the current map's height and no-fly zone values. The coefficents used are: $C_d = 1$, $C_t = -10$, $C_a = -10$, $C_h = -0.001$, and $C_n = 0.001$. In addition to this, at a set interval, the autopilot tries to fly the set plan and gives it's average reward as feedback to the reinforcement learner.

3.4.4 Multi-Channel Memory Buffer planner

To solve the problem with only having information about the local surroundings, as well as not being able to backtrack, an experimental global planner was also designed. The Multi-Channel Memory Buffer (MuMB) planner was inspired by the distributed, locally acting, intelligence of *P. polycephalum*. Working locally and adding recurrence, allows the algorithm to work iteratively as well as solves the problem of handling different sized input maps.

Proximal Policy Optimization (Schulman et al., 2017), or PPO, is a kind of policy-gradient RL algorithm developed by OpenAI and has been used in many domains to achieve state-of-the-art performance. Unlike DQN and NAF, it is on-policy, and this implementation therefore only supports single-threaded training without a long-term replay buffer. The core idea behind PPO is to define multiple training objectives that should be optimized and that weigh against each other. For one, there is a so-called critic, which is a neural network that tries to predict the discounted rewards given a current state ($\hat{r_i}$). Training happens in batch sizes of 128, and because the algorithm is on-policy, this means that this also becomes the frequency of the training steps. The true discounted rewards r_i are calculated at each training step and is calculated by backward iterating through the rewards and letting $r_i = \gamma r_{i+1}$, where i is the current step, and γ is the discount factor. The prediction of the critic is subtracted from the actual return, giving a value ($\hat{A}_i$), representing how much better the actor performed than expected. The actor is able to output continuous actions, and does this by having the model output a mean and standard deviation, which is then randomly sampled from a normal distribution.

The following loss function is then defined:

$$L_i^{VF} = \mathbb{E}[(r_i - \hat{r_i})^2]$$

$$p(a|s) = \frac{\pi_{current}(a|s)}{\pi_{old}(a|s)}$$

$$L_i^{CLIP} = \mathbb{E}[min(p(a|s)\hat{A}_i,\ clip(p(a|s), 1-\epsilon, 1+\epsilon)\hat{A}_i)]$$
$$L_i^{SUM} = L_i^{VF} - L_i^{CLIP}$$

Where $\pi_{current}(a|s)$ and $\pi_{old}(a|s)$ denotes the probabilities of performing the given action in the current policy, and for when the policy when the action was taken. The ϵ is a small value used to limit the maximum step size. The function *min* picks the lower of the parameters, and *clip* thresholds the first parameter between the second and third. Note that PPO typically also has an objective that maximizes entropy and thus encourages exploration, but that this was deemed unnecessary in this case. This loss function has the effect of the critic trying to predict the reward accurately, and the actor always trying to outperform this expectation.

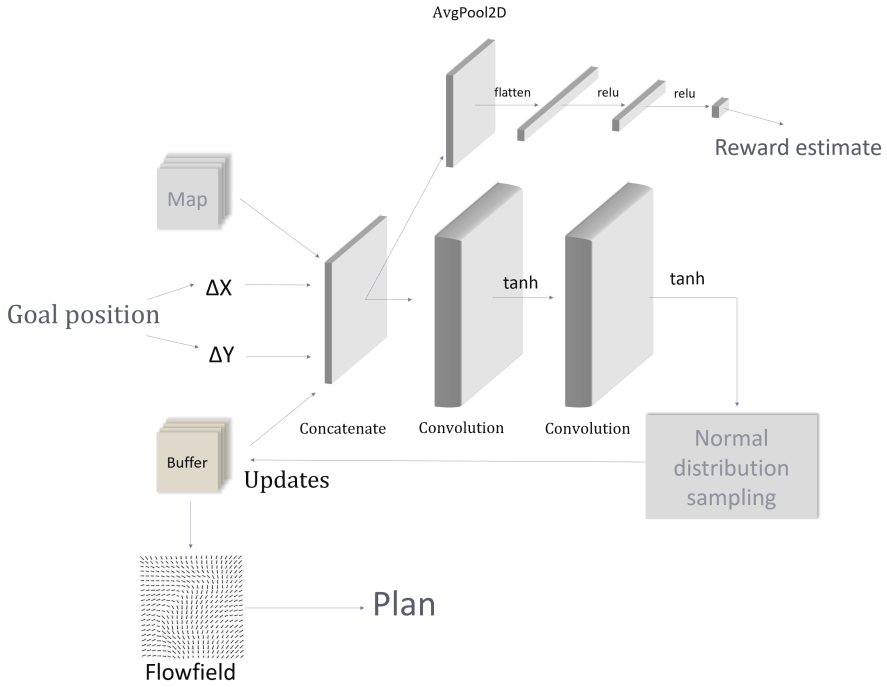


Figure 7: Diagram of Multi-Channel Memory Buffer model. The input to the network is the color segregated map channels, and two channels representing the component vector at each point towards the goal. This data is concatenated with the memory buffer, put though two convolutional layers, sampled from, and then used to update the memory buffer. This happens in a loop 10 times, after which the two first channels in the buffer are read as a flowfield in order to extract a plan. The critic (top part), tries to predict the resulting reward using a fully connected architecture.

The model used (Figure 7) is relatively simple. It relies on a memory buffer (B) which is of the same 2D size as the map. This buffer has 4 channels and servers as both the input and the output to the network. As input (s) it takes a map of the whole world (with color segregated), a normalized x and y vector of direction towards goal at every piled, and the buffer itself. This is ran through two convolutional layers (kernel size 3x3), with $tanh$ activation in between. The output of this network ($f(s)$) is used to update the memory buffer according to:

$$B' = \gamma B + (1 - \gamma) f(s)$$

Where γ is a number between 0 and 1. The value $\gamma = 0$ was found to work the best. This is then ran in a loop for 10 iterations. Normally, PPO samples randomly from all actions, which in this case would be a large amount of pixels. However, this would mainly result in noise across the image and thus wouldn't result in exploration due to there being no consistent policy changes being tested. Therefore, noise is added to the convolutional kernels at each inference step, and then reset. This will thereby affect the buffer in a more consistent manner, which was found to help with model convergence greatly.

At each step in the refining process (when training), a plan is extracted from the first two channels by interpreting each pixel as a vector, normalizing it, and scaling it with a factor. This plan then has a static reward extracted, which is fed into the PPO during the next step. For every n'th plan, the autopilot tries to fly it, and the average reward of the autopilot is given as feedback to the PPO.

3.5 Plan quality metric

Every plan can be statically evaluated, without actually having been flown, in order to get a reward quickly. This reward is used both for training, and as a metric for evaluating performance. The value is the mean of the individual reward for all waypoints r_j^{wp}, which are defined by the following equations:

$$r_j^{distance} = \left\| \vec{\Delta}_{j-1}^{wp} \right\| - \left\| \vec{\Delta}_j^{wp} \right\|$$

$$r_j^{nofly} = \begin{cases} 1 & if\ in\ no\ fly\ zone \\ 0 & if\ not\ in\ no\ fly\ zone \end{cases}$$

$$r_j^{ground} = min(wp_y - terrain(wp_x, wp_z) - 10, 0)$$

$$r_j^{wp} = C_d\, r_j^{distance} + C_n\, r_j^{nofly} + C_g\, r_j^{ground}$$

Where j is the waypoint's index, $\vec{\Delta}_j^{wp}$ is the waypoint's distance to the goal, wp_y is the waypoints y-coordinate in space, and $terrain(x, z)$ is the height of the terrain at a given 2D position. The constant were set to the following values: $C_d = 0.1$, $C_n = -0.5$, and $C_g = -0.2$.

4 Results

4.1 Performance

All planners were tested for world sizes of $375x375x10\ m^2$, and $500x500x10\ m^2$. The fighter jet always starts in the middle of the world, and a goal is a uniformly sampled random point. 3500 plans were collected for each planner, and their results were logged. Note that all planners can return *Not-a-Number* (NAN) static and pilot rewards when a plan failed to be generated. These values are excluded from the average and variance calculations. The reward and pilot reward mean, median, and percentiles were found and plotted in Figure 8 and 9, as well as listed in Table 1. Note that, because some values are excluded, the true performance may be lower by varying degrees.

Execution time was also measured for the whole planning process. This data was computed and plotted across the world sizes for every algorithm (Figure 10). For each planner, the data for both world sizes were also merged, and a Spearman's rank correlation coefficient was performed between execution time and the distance between the start and the goal. This gave $r_s = 0.246$ for A*, $r_s = 0.443$ for the Local DQN planner, and $r_s = 0.456$ for MuMB, with $p < 0.001$ and $N = 3499$ for all. Note that the first iteration was removed here since it is vastly slower than the coming ones due to caching not having been done yet.

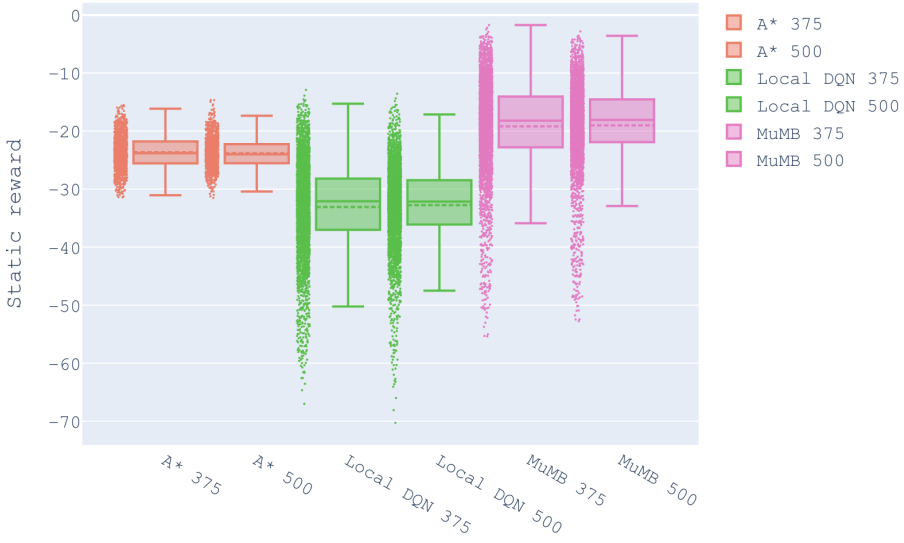


Figure 8: Box plot of static reward with NAN's filtered. Markers to left of boxes represents individual values.

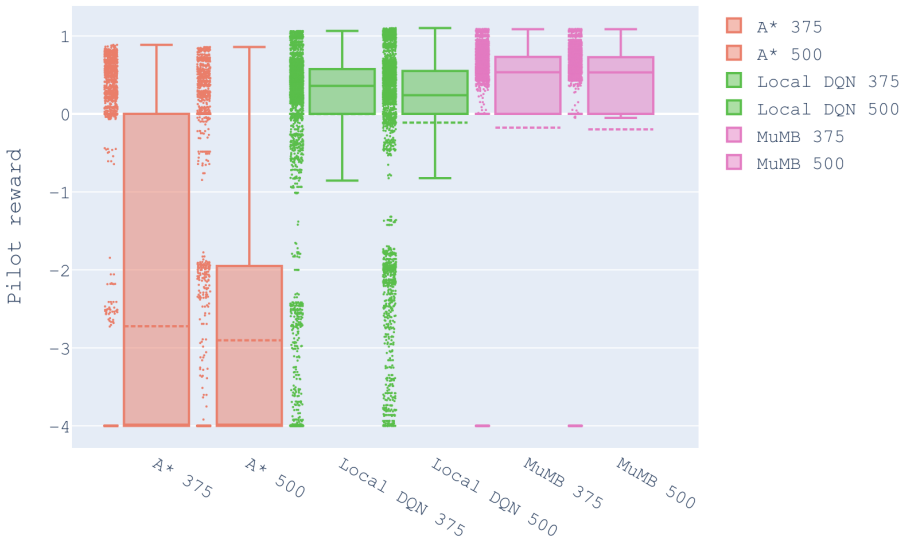


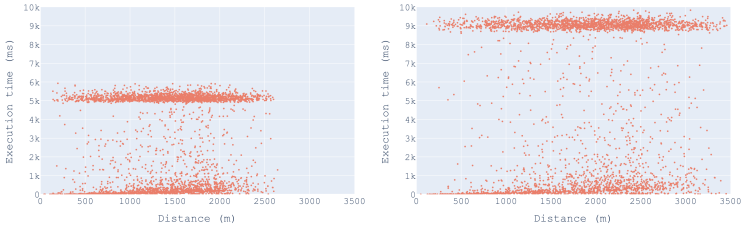
Figure 9: Box plot of pilot reward. Box plot of static reward with NAN's filtered. Markers to left of boxes represents individual values.

Planner performance						
Planner and world size	Mean static reward	Median static reward	Mean pilot reward	Median pilot reward	NAN pilot reward	NAN pilot reward
A* 375	-23.66	-23.83	0.25	0.32	66.4%	69.9%
A* 500	-23.81	-24.02	-0.14	0.01	68.1%	71.6%
L. DQN 375	-33.08	-32.10	0.11	0.38	0.0%	2.7%
L. DQN 500	-32.78	-32.15	-0.10	0.24	0.0%	0.4%
MuMB 375	-19.19	-18.22	0.58	0.58	0.0%	16.5%
MuMB 500	-19.01	-18.09	0.58	0.57	0.0%	16.9%

Table 1: Performance of planners for different world sizes.

Planner execution times		
Planner and world size	Mean execution time (ms)	Median execution time (ms)
A* 375	3092	4979
A* 500	5383	8789
L. DQN 375	98	112
L. DQN 500	125	127
MuMB 375	48	46
MuMB 500	69	67

Table 2: How long the planner took to generate a plan for different world sizes.

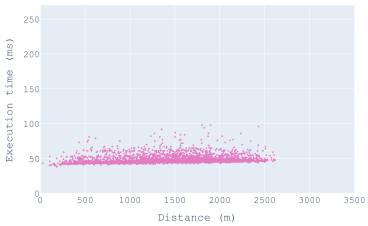


(a) A* 375x375

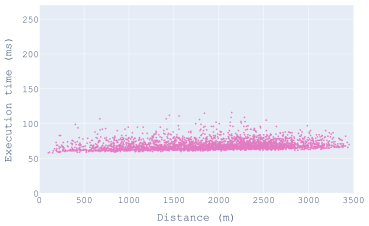
(b) A* 500x500

(c) Local DQN 375x375

(d) Local DQN 500x500



(e) MuMB 375x375



(f) MuMB 500x500

Figure 10: Box plots representing distance between the start and goal location versus the execution time in milliseconds for the whole planner execution time.

4.2 Strategies

As can be seen in Figure 11, the A* planner is strictly focused on getting to the goal, and will always behave optimally in relation to this objective. This leads to sharp angles and going along the edges of mountains, no-fly zones, and air defense system spheres. This planner's ability to attack is rarely used since there most often is another peaceful solution with a lower cost. One drawback with going so close to the mountains is also that it depends on the autopilot is able to follow the path exactly for long periods of time, which is something that it often has trouble with. This leads to a significantly lower performance for the pilot reward (Figure 9) due to crashing. Furthermore, the sharp angles of the plan do not follow a realistic flight trajectory, which might also be a contributing factor to this.

Figure 12 shows how the Local DQN planner is opportunistic. The planner rarely find a way all the way to the goal, but will often be able to move towards

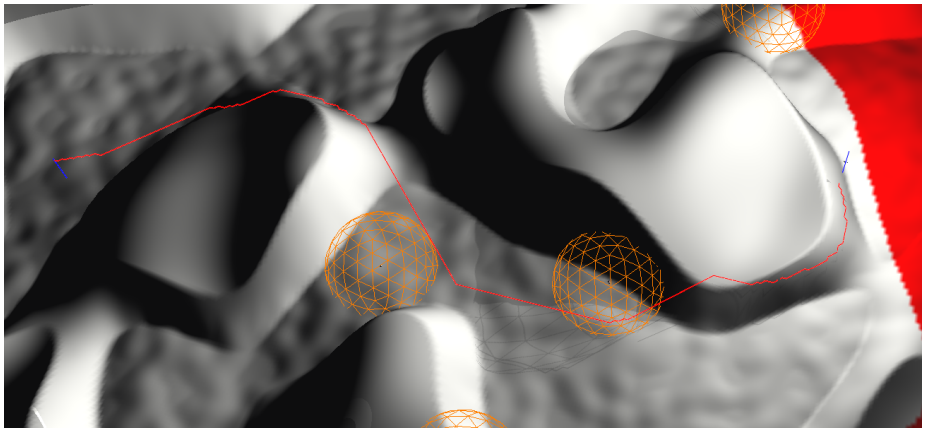


Figure 11: A* planner example. Tends to "hug" the edges of obstacles in order to minimize the distance to the goal.

it, taking into some account the obstacles in the way. That said, it is not rare for it to move through mountains, which sometimes enables to keep planning when not having a clear path forward. Running into dead-ends occurs frequently, which is either handled by the algorithm backtracking, or by planning in circles. The planned path is always smooth.

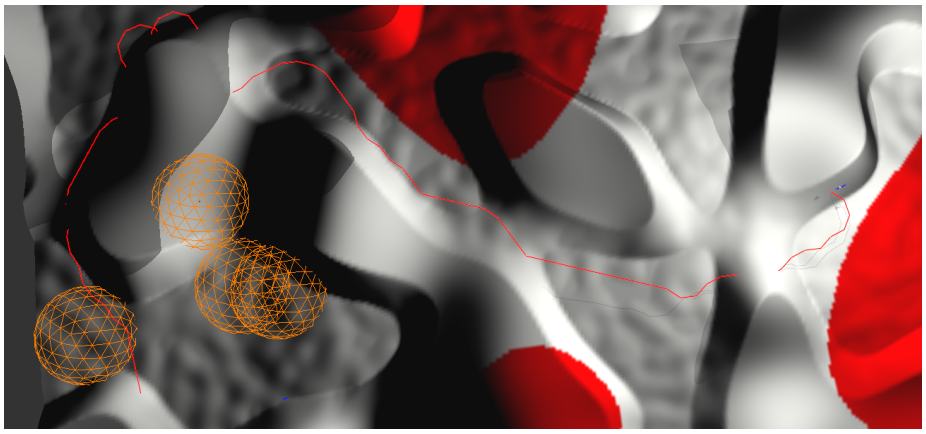


Figure 12: Local DQN planner example. The planner will often not go straight towards goal, although it often travels in the right general direction. It sometimes ignores obstacles. In left upper corner two loops are planned.

The MuMB planner (Figure 13) tends to disregard obstacles about as much as the Local DQN. However, unlike the DQN it tends to travels in somewhat straight lines. This behavior can also be seen by looking at Figure 14, which visualizes how the memory buffer changes for every iteration. This shows a color

shift centered around the goal, but also large swaths of uniform color, which is what results in the straight lines when interpreted as a flowfield. This more or less guarantees that the last waypoint of the plan will end up at the goal location. One can also observe distortions in the image, corresponding to the terrain below, which causes *some* reaction to obstacles.

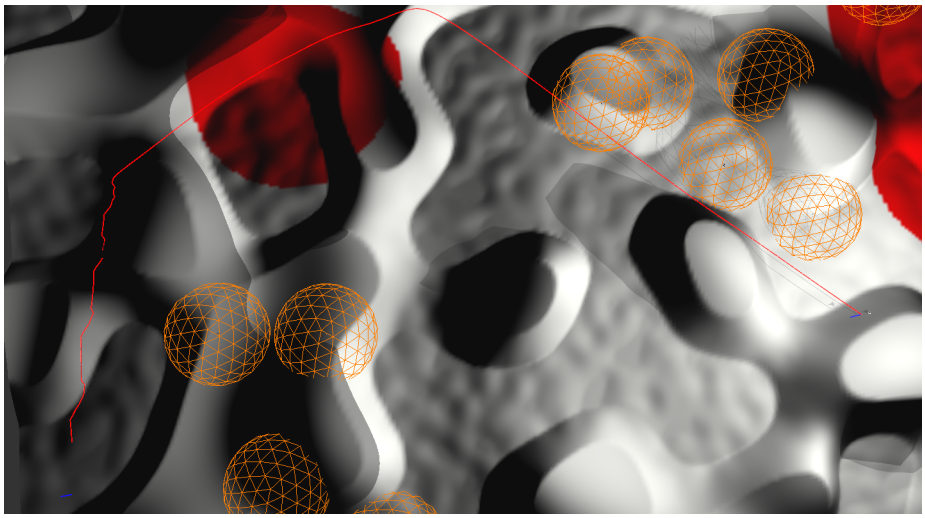


Figure 13: MuMB planner example. Will often travel in straight lines and exhibit sharp turns. It often ignores obstacles, but for some areas (left side), terrain is considered.

Figure 14: MuMB planner refining memory buffer iteratively. Red color channel represents x (east-west) component and green channel represents z (north-south) component of vector in plan extraction. Progressing left to right and top to bottom. Notice the oscillating behavior, and how every state is more refined than the second next.

5 Discussion

5.1 Planning performance

All planners sometimes fail at finding a valid path, and there is an overall wide spread in the success rate. The best planner is MuMB both in terms of average static reward, pilot reward (Table 1), and execution time (Table 2). The A* planner fails to generate plans a large majority of the time, and when it does it scores decently on the static reward, but poorly with the pilot reward. The Local DQN seems to perform more poorly than MuMB and A*, but is overall more reliable than A* when invalid plans are considered. As expected, both RL planners also do better at plan followability (Figure 9).

It should however be noted that reward may not necessarily correspond to what humans would deem as the best plans. It is also important to note that the specific terrain settings may have a significant effect on the planners' ability to find paths. For example, the A* planner might fail often because it is logically impossible for it to find a valid path through the terrain at the set flight altitude. As is suggested by Figure 10 *a)* and *b)*, A* may find a solution relatively quickly (although slower than the other planners) if one exists, but will stall and fail due to reaching the maximum iteration limit otherwise.

The planners all become slower when the world size increases. Although two sizes of worlds aren't enough to empirically confirm it, it can be expected that this value will at least increase linearly by the area of the world for all planners. However, as seen in Table 2, the relative increase in execution time with A* is more significant than for the others. A comparison can be made to how human working memory can be overloaded during planning for problems that require a high level of top-down computation. A* has a lot to keep in memory but is only able to explore one node at a time. It is also worse off than humans, and other neural networks, due to not having any ability to generalize in order to scale down the search space. It is also unable to act on heuristics. The Local DQN has the opposite problem. Due to its model structure, it can process the whole surrounding concurrently, but it cannot retain any information over time. This means that is has to understand as much as it can in the current moment, and then choose an action that is probable to lead it towards the goal. This makes it purely reactive to stressors such as no-fly zones, and means that it can be fully understood from a behaviorist perspective.

Much like the sampling-based path finding algorithms discussed earlier, Local DQN benefits from not being careful and betting on a specific path instead of spending compute resources on careful deliberation about the best way forward. This however comes at the risk of running into dead ends, which can significantly lengthen the planned path. In relation to the speed-accuracy-tradeoff, this suggests a planner which is heavily biased towards speed. This comes from the nature of the algorithm. Maybe as a method of increasing accuracy by "looking around", the planner will sometimes create one or two circles before continuing on. This is in contrast to MuMB, which almost always moves deliberately forward (although not always directly towards the goal) without getting stuck.

It is notable that both the Local DQN and MuMB gladly will plan through mountains, no-fly zones, and go close to air defense systems. This is likely a result of the algorithms not having converged enough during training to find a proper strategy, but it could also be seen as a form of bottom-up planning, where some decisions are left to be made later *in situ* by the autopilot.

5.2 Future investigation

All the RL algorithms were found to be relatively tricky to train, and as a result, there is a question of whether limitations in performance are a result of fundamental limitations of the model and overall method, or if there is left untapped potential due to bad convergence. As can be seen in Figure 15, catastrophic forgetting was an issue that relatively quickly arose. This was also a problem when training the autopilot, and likely arises when the replay buffer fills with only good policy, removing the necessary learning pressure exerted by failures. One way to counter this is to increase the capacity of the buffer, although this comes at the cost of RAM. Another possible solutions could be to implement a prioritized replay buffer (Schaul et al., 2016).

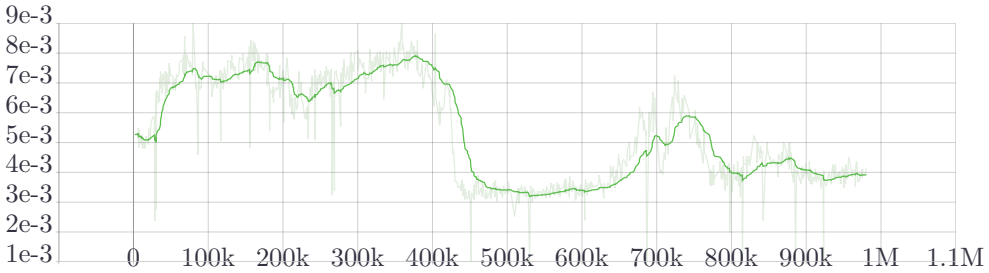


Figure 15: Local DQN training example from Tensorboard. X-axis represents training steps and is proportional with time trained, and y-axis the current reward (higher is better). The transparent line in the background is the original data, and the green line is with smoothing. The model quickly reaches maximum performance, and loss tends to be stable in the short term, but suffers from catastrophic forgetting in long therm (step 420k).

The on-policy PPO algorithm (training in 16) suffered less from catastrophic forgetting but was hard to make converge at all. Hyperparameter tuning and trying different implementation details (detailed by Engstrom et al., 2020), did help, and would likely be able to improve the convergence further. For the sake of simplicity, it would maybe be beneficial to switch all reinforcement learners to one algorithm. PPO is a good candidate for this because of its good converge, but does with this implementation not make use of the multi-threading capabilities.

One possible solution for all the learning algorithms could be to switch from dense rewards to sparse. Currently, multiple reward functions are used, all of which try to generate an estimate at every update. In some sense, this had

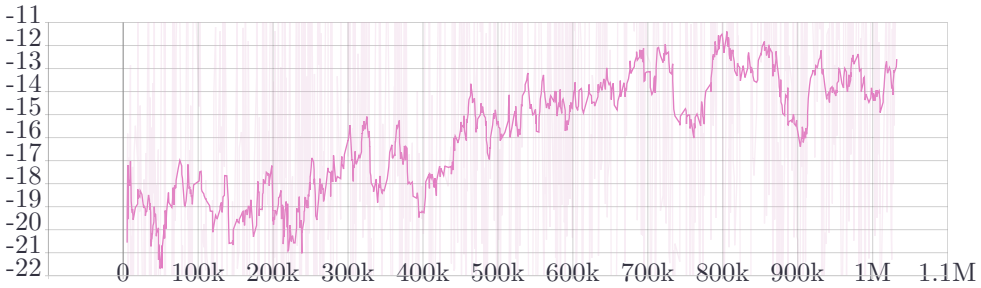


Figure 16: MuMB training training example from Tensorboard. X-axis represents training steps, and y-axis the current loss. As can be seen in transparent pink background line, the loss is very unstable, but tends to improve over time in a stable fashion until learning stagnates (pink line).

made it easy to fall into the trap of reward shaping, where complex reward functions are constructed to encourage a certain behavior. What often happens in these situations, and what has arguably happened here, is that undesired loophole behaviors develops instead. In this case, I observed planners ignoring mountains if punishment for this was low, and going in circles to not risk crashing into mountains if punishment was high. Sparse rewards would entail giving a reward signal only when reaching the goal, or crashing. This makes convergence harder to achieve, but also avoids rewarding the agent for "cheating".

Maybe the largest problem with the Local DQN is the inability to backtrack when running into dead ends. A promising way forward might be to convert it to a Value Prediction Network as done by Oh et al., 2017, and let a Monte-Carlo tree search perform the planning. The planner could also easily be converted to do produce output in 3D, which would make it more useful overall, and likely also increase plan followability.

The MuMB planner filled its purpose in terms of explainability and demonstrating that it works as a proof-of-concept. As can be seen in Figure 14, it does however not function completely as intended. Although the memory does refine, it oscillates between two distinct states, and there does not seem to be information spreading through the map. Although one could make the argument that these oscillations mirror the pulsing behavior of *P. polycephalum*, it likely evolves as a result of the limitations of the convolutional layers in the model. During development, I did experiment with allowing for more intelligent behavior by instead using a feed-forward network that strides across the buffer. This would likely allow for much more complexity in behavior, but so far I have been unable to make this kind of model converge. A suspected solution is to more effectively measure and reward exploration for this large action space. If this was solved, it would likely be possible to easily add 3D planning and attacking behavior to this planner as well.

6 Conclusion

The aim of this project was to investigate the feasibility of using RL algorithms for doing flight route planning. While neither of the implemented RL planners is unequivocally successful at this task, the algorithms could surpass the traditional A* planning algorithm in terms of both performance and execution time for a specific plan quality metric. The ability of the RL planners to adapt to an autopilot also makes them able to construct paths that are more feasible. The theoretical advantages of behind the learning algorithms in terms of being able to speed-accuracy-tradeoffs, choosing between top-down and bottom-up planning, and making use of distributed computing, are to some extent corroborated by the empirical results. That said, all algorithms are in their current form far from having the performance required in order to be used as real-world aids. While the field of flight route planning using modern methods hasn't been thoroughly investigated to my knowledge, much of the research done in other domains is relevant and could be used to considerably accelerate progress within also this field.

While it would be beneficial to further investigate other traditional algorithms such as sample-based planners, it is my opinion that RL is well worth exploring further for solving this task due to their flexibility, ability to perceptually generalize, and their ability to develop problem specific search strategies. Maybe most importantly of all, they have the ability to learn what should be planned beforehand and what can safely be decided by the pilot later. Hopefully, the simulation environment that was developed during this project also proves useful as a testbed for these future investigations.

References

Alim, K., Andrew, N., Pringle, A., & Brenner, M. P. (2017). Mechanism of signal propagation in physarum polycephalum. *Proceedings of the National Academy of Sciences*, *114*(20), 5136–5141. https://doi.org/10.1073/pnas.1618114114

Csáji, B. C. et al. (2001). Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, *24*(48), 7.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271. https://doi.org/10.1007/bf01386390

Elbanhawi, M., & Simic, M. (2014). Sampling-based robot motion planning: A review. *IEEE Access*, *2*, 56–77. https://doi.org/10.1109/ACCESS.2014.2302442

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., & Madry, A. (2020). Implementation matters in deep policy gradients: A case study on ppo and trpo.

Gu, S., Lillicrap, T., Sutskever, I., & Levine, S. (2016). Continuous deep q-learning with model-based acceleration.

Guez, A., Mirza, M., Gregor, K., Kabra, R., Racanière, S., Weber, T., Raposo, D., Santoro, A., Orseau, L., Eccles, T., Wayne, G., Silver, D., & Lillicrap, T. (2019). An investigation of model-free planning.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. https://doi.org/10.1109/TSSC.1968.300136

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780.

Khan, W., & Nahon, M. (2015). Real-time modeling of agile fixed-wing uav aerodynamics. *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, 1188–1195. https://doi.org/10.1109/ICUAS.2015.7152411

Kleinberg, J., & Tardos, E. (2005). *Algorithm design*. Addison-Wesley Longman Publishing Co., Inc.

Latty, T., & Beekman, M. (2010). Food quality and the risk of light exposure affect patch-choice decisions in the slime mold physarum polycephalum. *Ecology*, *91*(1), 22–27. https://doi.org/10.1890/09-0358.1

Latty, T., & Beekman, M. (2011). Speed-accuracy trade-offs during foraging decisions in the acellular slime mould physarum polycephalum. *Proceedings. Biological sciences / The Royal Society*, *278*, 539–45. https://doi.org/10.1098/rspb.2010.1624

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

Morris, R., & Ward, G. (2005). *The cognitive psychology of planning*. Psychology Press.

Nakagaki, T., Yamada, H., & Tóth, Á. (2000). Maze-solving by an amoeboid organism. *Nature*, *407*(6803), 470–470. https://doi.org/10.1038/35035159
Oh, J., Singh, S., & Lee, H. (2017). Value prediction network.
OpenAI. (2018). *Part 2: Kinds of RL Algorithms*. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html
Perlin, K. (1985). An image synthesizer. *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, 287–296. https://doi.org/10.1145/325334.325247
Peters, J. (2010). Policy gradient methods [revision #137199]. *Scholarpedia*, *5*(11), 3698. https://doi.org/10.4249/scholarpedia.3698
Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0
Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). Prioritized experience replay.
Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms.
Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., & et al. (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354–359. https://doi.org/10.1038/nature24270
Stoen, F. (2011). *Physarum polycephalum plasmodium*. https://commons.wikimedia.org/wiki/File:Physarum_polycephalum_plasmodium.jpg
Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bull.*, *2*(4), 160–163. https://doi.org/10.1145/122344.122377
Tamimi, N. (2020). How fast is c++ compared to python? - towards data science. https://towardsdatascience.com/how-fast-is-c-compared-to-python-978f18f474c7
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need.

Software references

Canonical LDT. (2020, April 23). *Ubuntu* (Version 20.04). https://ubuntu.com/
Daniel Chappuis. (2020, May 1). *Reactphysics3d* (Version c273e7c). https://www.reactphysics3d.com/
Docker. (2021, May 24). *Docker* (Version 18.09.1). https://www.docker.com/
Facebook's AI Research lab. (2020, April 25). *Pytorch* (Version 1.8.1). https://pytorch.org/
ISO/IEC JTC1, W., SC22. (n.d.). *C++* (Version 17). https://www.cplusplus.com/
Jetbrains. (2021, May 7). *Clion* (Version 2020.3). https://www.jetbrains.com/clion/
Julian Seward. (2021, April 19). *Valgrind* (Version 3.17.0). https://valgrind.org/

Junio Hamano. (2021, March 9). *Git* (Version 2.20.1). https://git-scm.com/
Khronos Group. (n.d.). *Ubuntu* (Version 3.2). https://opengl.org/
Kitware. (2018, November 20). *Cmake* (Version 3.13). https://cmake.org/
Microsoft. (2020, November 10). *Msvc19* (Version 16.8). docs.microsoft.com/en-us/cpp/
Microsoft. (2021, May). *Visual studio code* (Version 1.56). https://code.visualstudio.com/
Niels Lohmann. (2021, August 6). *Json for modern c++* (Version 3.9.1). https://github.com/nlohmann/json
Nvidia. (2021, May 24). *Cuda* (Version 11.1). https://developer.nvidia.com/cuda-zone
Pranav Srinivas Kumar. (2021, May 7). *Argument parser for modern c++* (Version 1344889). https://github.com/p-ranav/argparse
Python Software Foundation. (n.d.). *Python* (Version 3.8.5). https://www.python.org/
Richard Stallman. (2019, February 22). *Gcc* (Version 8.3.0). https://gcc.gnu.org/
RustingSword. (2020, November 17). *Tensorboard_logger* (Version 570c297). https://github.com/RustingSword/tensorboard_logger/
The GLFW Development Team. (2021, May 8). *Glfw* (Version 3.3). https://www.glfw.org/
The GNU project. (2020, May 23). *Gdb: The gnu project debugger* (Version 9.2). https://www.gnu.org/software/gdb/
Tzu-Wei Huang. (2020, May 3). *Tensorboardx* (Version 2.2). https://github.com/lanpa/tensorboardX/
Wenzel Jakob. (2019, September 22). *Nanogui* (Version e9ec8a1). https://github.com/wjakob/nanogui

Division of labor

- Program structure — Axel
- Development environment setup — Both
- Argument parsing — Axel
- Textual logging — Axel
- Threading — Axel
- Memory leak cleanup — Axel
- Perlin noise — Erik
- Heightfield generation — Erik
- No-fly zone generation — Erik
- Territory generation — Axel
- Air defense system generation — Axel
- World rendering — Erik
- OpenGL object management — Axel
- 3D modeling & texturing — Axel
- 2D Map — Axel
- Aerodynamics — Axel
- Tensorboard logging — Erik
- Autopilot — Axel
- Random planner — Axel
- A* planner — Erik
- DQN with motion primitives planner — Erik
- MuMB planner — Axel
- Data collection — Axel
- Data analysis — Axel