Imperial College London

MEng Individual Project Report

Imperial College London

Department of Computing

Meta-Learning Path Planning Networks

Author:
Zoltan Hanesz

Supervisor:
Dr. Ronald Clark

Second Marker:
Dr. Edward Johns

June 13, 2021

Abstract

Traditional path planning algorithms are characterised by long runtimes and additional constraints, such as requiring a discrete set of possible points to visit. Due to great need for fast and efficient path planning in realtime robotics, the focus of research in this field has recently shifted towards developing deep learning-based techniques, achieving significantly improved inference times at the cost of precision. This work aims to improve the predictive accuracy of the current state-of-the-art method in higher-dimensional spaces without the need for collecting additional training data. The main focal point of our approach is the application of meta-learning, a novel machine learning concept, which promises to increase the efficiency of the optimisation process while simultaneously enhancing performance. We show that our solution achieves consistent improvement on the baseline method and demonstrate the potential of meta-learning in the domain of path planning.

Acknowledgements

I would like to thank **Dr. Ronald Clark** and **Daniel Lenton** for our regular meetings, which resulted in many ideas that had a significant impact on this project's success.

I would also like to thank my friends and family, most notably **Zoltan**, **Angelika** and **Julia Hanesz** who have given me much support throughout all of my studies and during these last 15 months, which have been challenging both academically and in our personal lives.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Path planning, also known as the navigation problem, is a long-standing, well-established challenge and one of the core areas of research in the field of robotics and artificial intelligence, with the first notable solution being Dijkstra's path planning algorithm [15] in 1959. Accelerated by the development of self-driving cars, autonomous unmanned aerial vehicles (UAVs), mobile robots, or robotic manipulators, the need for path planning algorithms enabling spatial navigation in real-time scenarios has become even greater.

While extensive research has been conducted to tackle various aspects of this problem, traditional methods often fail in complex high-dimensional environments due to high memory requirements and computational time. Applying path planning methods in practice becomes even harder, since besides the underlying scene, the physical properties/limitations of the moving robot itself, the uncertainty or even partial lack of the image of the environment obtained via noisy sensor readings also need to be taken into account. One class of approaches capable of handling all of these constraints reasonably well are sampling-based algorithms (section 2.2.3), which sample random points in the environment and apply a number of heuristics to connect these points in order to build non-colliding paths between the start and goal points. These methods show promise, as multiple studies have demonstrated, that they can produce feasible paths relatively quickly even on harder problems, however they have also been proven to be rather heavy-tailed, so their ability to plan optimal routes within a reasonable time frame can be brought into question.

Due to the recent popularity of machine learning, powered by the massive increase in available computing power, deep learning methods (section 2.3) have been employed to tackle the path planning problem as well. Current research suggests that utilising deep learning to certain parts of the problem can hold the key to increasing the inference speed, while still generating cost-effective paths that can rival other state-of-the-art approaches. For instance, learning-based methods have already been applied to two-dimensional path planning problems [9], or in combination with other non-learning-based algorithms, even to higher-dimensional spaces with good success [11][16]. Moreover, the concept of meta-learning [17], also known as "learning to learn", aiming to create models that generate well to a large variety of tasks and can be quickly adapted to maximise performance on the individual tasks, shows promise to tackle the issues of long training times and lack of available data, which cripple many deep learning methods. These discoveries in the field of machine learning and path planning serve as the main motivation for this master's thesis.

1.1 Objectives

Inspired by recent promising results in deep learning-based path planning, the main objective of this thesis is to **apply meta-learning to increase the performance of path planning networks (PNet) introduced in [10]**. While the focal point of the original solution from [10] is training these networks in a supervised fashion via expert demonstrations, our goal is to achieve consistent improvement in the predicted paths without having to collect any additional ground truth data. We split the work required to fulfill this objective into the following steps:

1. **Train PNet** - To acquire the starting point for meta-learning as well as the baseline for evaluation, first we train our own path planning networks based on the implementation in [18].

2. **Unsupervised loss** - Find an unsupervised loss function and verify its ability to consistently enhance the performance of the networks obtained from 1. Here we focus mainly on the consistency, not so much on the extent of the improvements.

3. **Meta-learning** - Increase the responsiveness of the path planning networks to the unsupervised loss by training with a meta-learning pipeline customised to the problem at hand.

4. **Analysis & enhancements** - Evaluate various meta-learning methods to identify the one that is most suited to this problem and find ways of maximising the performance boost given by the unsupervised loss function.

1.2 Challenges

Overall, we faced many challenges related to the novel nature of our method, which required careful considerations, and there were also other hurdles encountered throughout the duration of this project. The most significant issues are listed below:

1. **Unstable optimisation process** - Perhaps one of the most characteristic and by far the most time-draining challenge of this project is the unstable optimisation process caused by essentially re-training pre-trained models to a slightly different objective and the instability of Model-Agnostic Meta-Learning itself. This hindered our progress as we had to find the perfect hyperparameters to ensure that our method reliably improves over time.

2. **Long training times** - In addition to being forced to reduce the speed of learning due to stability issues, we also utilise computationally expensive operations in our solution, causing long training times of up to 2 days. This fact considerably inhibited our ability to iterate on our approach.

3. **Evaluation metrics** - Due to the lack of a widely accepted unified benchmark for path planning algorithms, we had to come up with our own metrics that best captured the performance of our method and of the baseline we aimed to improve on. Evaluating collisions proved particularly tricky since there can be a vast difference in quality between two paths that both collide with the environment depending on where and how many collisions occur.

4. **Fine margins** - Since in some cases only marginal improvements are achieved on simpler problems, we had to perform thorough analysis across all available metrics to ensure our approach is valid, and thus we can move on to harder problems, expecting larger improvements.

1.3 Contributions

The contributions of this thesis can be summaries in 4 key points:

1. **PNet refinement** - We refine PNet by addressing its weakness of getting feedback only on the predicted path segments during training using an unsupervised loss function providing path-level response and show consistent increase in performance.

2. **Meta-learning methods comparison** - By comparing three well-known meta-learning techniques commonly associated with each other, we critically evaluate the important differences between them and provide reasoning why some are not suited for our problem or any similar task.

3. **Meta-learning PNet refinement** - Considered as the main focal point of this thesis, we demonstrate the benefit of applying meta-learning to maximise PNet's sensitivity to the unsupervised loss, resulting in shorter paths with less collisions.

4. **More informed refinement** - We highlight the most significant limiting factors of our solution and address the issue of the rigid mathematical formulation of the unsupervised loss by extending it with additional information obtained during the meta-learning optimisation process.

1.4 Ethical Considerations

Taking a critical look at any potential ethical issues relating to this project, we can say for certain that all data used to train our model and for subsequent benchmarks in the evaluation phase are synthetically generated environments and paths, therefore no living beings (i.e humans or animals) are involved and no personal data is being processed. Furthermore, as all the used data and methods are created by us or under the freely distributable MIT license, we do not have any evident legal issues to consider.

The most notable areas where ethical consequences of our research need to be brought into question are dual-use and misuse. It is a fact that autonomous vehicles are of great interest of many militaries around the world. The possible use cases of such vehicles cover a wide range of the spectrum, from unharmful cargo carriers through spying drones to autonomous fighter jets. While it is important to note, that path planning itself does not directly constitute towards any human lives being changed due to military intervention, it provides the tools for these vehicles to operate more efficiently, which can make a big impact on how they will be deployed in the future. As in the current days, it is relatively simple to obtain certain devices even via regular legal channels, the possibility of criminal misuse of similar types of machines is also something to be considered. Taking all of these factors into consideration, we emphasize that our model is not intended for any of the uses described above.

Chapter 2

Background

In this chapter, first we introduce the main concepts which serve as the base building blocks of our solution and are necessary to understand this work. Here we also briefly introduce other topics used in existing methods closely related to ours for clarity. Specifically, we provide foundational knowledge on various deep learning techniques and signed distance functions. Next, we break down existing traditional path planning methods based on the taxonomy by Yang et al. [8] to give context on where our solution fits in across the entire landscape of path planning algorithms. As this study focuses mainly on non-neural-network-based methods, we also discuss these in further detail separately in section 2.3. Lastly, we present the most relevant meta-learning approaches to our work in section 2.4, highlighting their characteristics which point to their potential to produce good results in the domain of path planning.

2.1 Foundational Knowledge

2.1.1 Deep Learning

Neural Networks

Virtually every computer program contains functions, which given some input parameters, perform operations on the input data, and return some output. A simple example would be a search function that returns elements from a collection of data matching the inputted search parameters. In scenarios like this, when there is clear correspondence between the input parameters and the function's objective (e.g. string and integer comparisons are trivial), algorithms can be devised by programmers to find these correspondences and solve problems with maximum certainty which can then be translated to computer programs. A significantly more complex example would be an image classification problem, where from a raw input image of a cat or a dog, the desired function $f(x) = y$ needs to be able to differentiate between the two. In these cases, traditionally programmers are forced to come up with creative methods of approximating f (e.g. by extracting geometrical features of the image). Inspired by the way humans learn [19], neural networks and deep learning aim to tackle these types of problems by providing the computer with a large number of expected input-output pairs, enabling it to learn to estimate the output of f.

More formally, neural networks can be described as functions $f'_\theta(x) = \hat{y}$, parameterised by parameters θ, which are iteratively refined using the provided input-output pairs during so-called *training*, to minimise the differences between the predicted outputs $\hat{y}$ and true outputs y. The fundamental components of every neural network are artificial neurons. As depicted in fig. 2.1, the neuron N processes the incoming signals $x_1, x_2, x_3, x_4 \in x$ using the network parameters $\theta_1, \theta_2, \theta_3, \theta_4, b \in \theta$ and produces the neuron output by passing the signal from N through a non-linear activation function φ to allow estimating non-linear functions. Neurons are traditionally stacked vertically into network layers enabling each neuron to extract different information from the provided input signal.

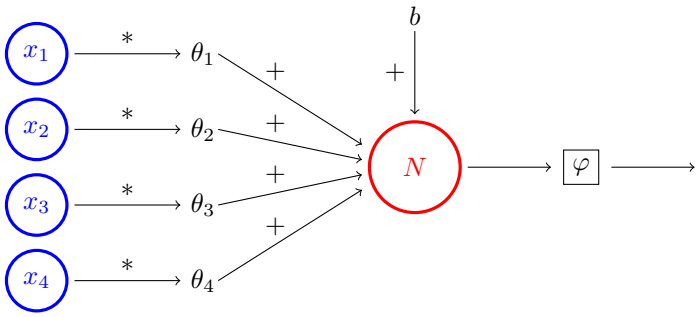


Figure 2.1: Diagram of an artificial neuron with 4 input signals x_1, x_2, x_3x_4, weights $\theta_1, \theta_2, \theta_3, \theta_4, b$ and activation function φ.

Deep Neural Networks

As the complexity of the task at hand grows, network layers are often connected horizontally to form Deep Neural Networks [20]. The idea here is that in order to accurately approximate f using just one layer, a large number of parameters θ might be required, but by viewing network layers as separate functions, f'_θ can be expressed as a composition of **multiple hidden-layers**. In essence, each layer can learn to extract different features from the input signal and by passing the output of one layer to the other, a more expressive, better approximation of f can be achieved often with less parameters used. In the end, the output of the last hidden layer is combined in the final output layer as seen in eq. (2.1), with hidden layers $h^{(1)}_{\theta_1}, h^{(2)}_{\theta_2}$ and output layer o_{θ_o}.

$$f'_\theta = o_{\theta_o}(h^{(1)}_{\theta_1} \circ h^{(2)}_{\theta_2}) \tag{2.1}$$

Results of research in recent years have shown that building deeper networks does provide a clear performance benefit, and due to their modular design, many network architectures have been developed to tackle a large variety of tasks. Alongside traditional **Feed-Forward Networks**, such as the one depicted in fig. 2.2, the two most notable are **Convolutional Neural Networks(CNNs)** and **Recurrent Neural Networks(RNNs)**.

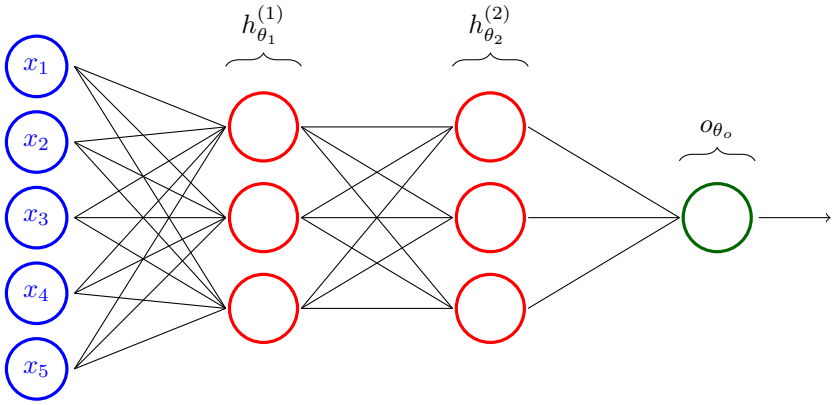


Figure 2.2: Simple Deep Neural Network with two fully connected hidden layers $h^{(1)}_{\theta_1}, h^{(2)}_{\theta_2}$ and output layer o_{θ_o}.

Much like traditional methods that utilise convolutional kernels to extract key features of a large possibly noisy input, such as the Sobel filter used for edge-detection [21], **CNNs** [22] learn the weights of many different convolutional kernels to extract the most important information for the given task. In a deep CNN, each hidden layer outputs feature maps as a result of applying the learned kernels to the input, and with the use of pooling layers (e.g. max-pooling retains only the largest value within the kernel window), the signal is reduced to a significantly smaller size before the output layer produces the final output. **RNNs** [23] are another very important class of deep learning methods. Their main characteristic is that the information flow is not described as

feed-forward, but cyclic, meaning that the input to the network is dependent on the output of the network in the previous iteration. This property makes them particularly useful when modelling problems with sequentially dependent data, such as machine translation and robot control.

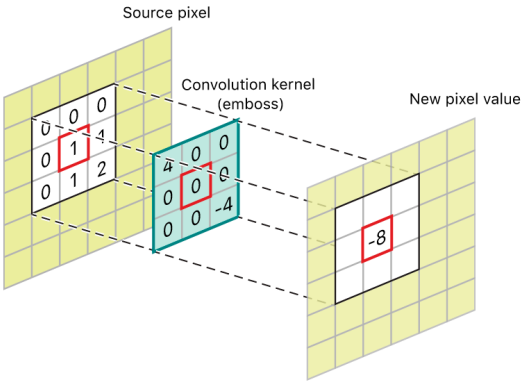


Figure 2.3: Applying a convolutional kernel with pre-defined weights to blur the input image [1]. CNNs learn the weights of kernels to maximise performance on the task.

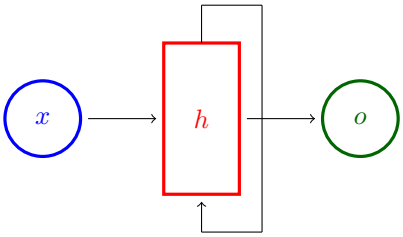


Figure 2.4: The output h_{t+1} of the hidden layer h at time step $t+1$ depends on the input data x as well as the output h_t at time step t.

Training a Neural Network

So far, we have briefly touched on the fact that neural networks are conditioned on parameters θ, which are iteratively refined during training to optimise the model's performance on the task. Throughout this process, the network predictions $f'_\theta(x) = \hat{y}$ are evaluated via a chosen loss function $\mathcal{L}$ that captures how well f'_θ approximates the true function f. A method called **backpropagation** can then be performed by taking the gradient $\nabla_\theta \mathcal{L}(f'_\theta(x))$, updating θ in the direction of the gradient in order to minimise future losses. It is important to note that the choice of $\mathcal{L}$ is critical, since it has to be differentiable with respect to θ to compute $\nabla_\theta \mathcal{L}(f'_\theta(x))$, and its minima must guide θ to a good approximation $f'_\theta \approx f$ i.e the loss must correspond to the task. Due to limitations caused by the large size of the training datasets, it is not feasible to compute the loss on the whole dataset, so often a method called **Stochastic Gradient Descent (SGD)** [24] or its batch variant, Mini-Batch SGD is used, which randomly chooses a single or a smaller number of training samples from the training dataset X to compute the loss and thus approximate the true gradient. By picking the right batch size, Mini-Batch SGD provides a good trade-off between computational performance and accuracy of the update direction, reducing potential input noise and improving the convergence rate as a result. In practice, other hyperparameters can be tuned to achieve good convergence, like the learning rate α, which regulates the size of the update step taken in the direction of the gradient. Optionally the gradient itself can be reduced by clipping its maximum value or normalising by the magnitude of its l2-norm. Putting everything together, we get the update equation eq. (2.2).

$$\theta = \theta - \alpha * \sum_{i=0}^{n} \nabla_\theta \mathcal{L}(f'_\theta(x_i)) \qquad x_1, ..., x_n \subset X \tag{2.2}$$

In contrast with traditional algorithm-based methods, where the solution is usually based on an analytical solution to the task, when it comes to deep learning, a higher emphasis is placed on the network architecture and on the method of learning. Below we discuss the three most common learning paradigms in deep learning.

Learning paradigms

In today's world, where large amounts of data are collected everywhere, **supervised learning** is a very powerful technique. As the name suggests, the learning process is supervised by comparing the network prediction with the desired output, often taking the mean squared error of the two as training loss. The presence of ground truth data makes it suitable for classification tasks or even

complex regression tasks which might not be otherwise easily mathematically tractable. The most challenging aspects of this method are obtaining sufficient amount of data to make them useful and processing the data so that the network can achieve best results.

Unsupervised learning provides an interesting alternative to the data-hungry supervised learning. Here the feedback to the network during training is granted in the form of a loss function that is conditioned only on the input data and network output, so there is no requirement to obtain ground truth data. Unsupervised loss functions can be much more sophisticated as their minima has to correspond to the task at hand. While having a clear advantage in terms of data requirements, this method is mainly used to identify patterns (e.g. clustering, compression) and not widely utilised for complex tasks due to its reliance on hand-crafted loss functions, which may not represent the desired behaviour accurately

For problems where we have an agent that can perform certain actions in a environment and the goal is to devise a policy to determine the best actions to take in a given state, it might be more practical to develop the policy from past experience. Based on this principle, **reinforcement learning (RL)** techniques are traditionally trained by having an agent interact with the environment based on the policy π, which is refined over time to maximise the reward received after taking an action. For large state-action spaces, deep RL utilises a neural network to learn the optimal policy via various Policy Gradient Algorithms. Despite not having requirements for any input-output data, these methods may not be suitable in some scenarios, as they require the environment to be readily available during training and the time required to sufficiently explore the space is also a clear disadvantage.

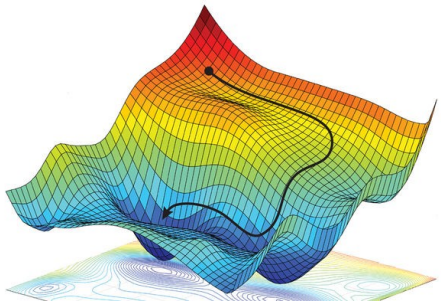


Figure 2.5: [2] Unsupervised and supervised learning utilise gradient descent [3], an optimisation algorithm to find a local minimum of a differentiable function. Various measures need to be taken to ensure the algorithm isn't stuck in suboptimal local minima.

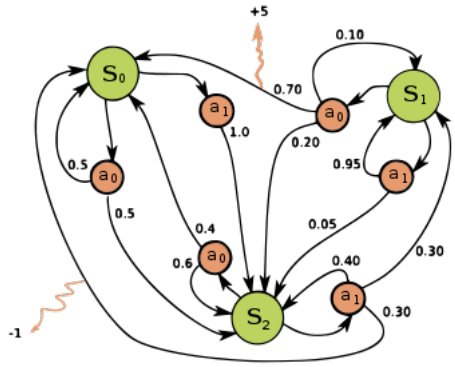


Figure 2.6: [4] Reinforcement learning models the underlying task as a Markov Decision Process with states $S_0, S_1, S_2... \in S$, actions $a_0, a_1... \in A$ and seeks to find the best policy i.e distribution of probabilities of taking a particular action in a given state that maximises the reward R.

2.1.2 Signed Distance Functions

The **signed distance function (SDF)** provides a way of expressing the relative distance of points from the objects in a given environment. More formally, given a set Ω with boundary $\delta\Omega$ in a metric space M with distance metric d (e.g Euclidean space with Euclidean distance), the function returns the distance of an arbitrary point $x \in M$ from the boundary of Ω. Depending on whether the point is inside or outside the boundary, the assigned value is negative or positive, which leads us to the general formulation of the SDF:

$$sdf(x) = \begin{cases} d(x, \delta\Omega), & \text{if } x \notin \Omega \\ -d(x, \delta\Omega), & \text{if } x \in \Omega \end{cases} \tag{2.3}$$

Thanks to its formulation, it is very convenient for describing shapes of objects and has been used in the field of computer graphics for real-time volumetric rendering and in computer vision

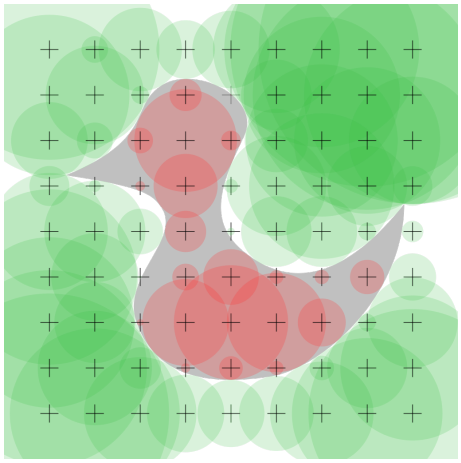


Figure 2.7: Example of a signed distance function used to represent the the shape of a duck. These fields are commonly stored as raster images to enable real-time rendering.

[25]. Another great property is that if Ω is a subset of the Euclidean space, it is differentiable in most cases, making it suitable for gradient-based learning. This characteristic combined with the ability to accurately represent obstacles in the environment make it a great candidate for use in gradient-based path planning problems to provide feedback during training. We also make extensive use of the SDF in our solution, albeit with slight modifications given by the nature of our input data, further discussed in section 3.1. Due to technical limitations, in some cases it might be more efficient to train a neural network to predict SDF values using neural networks instead of computing them as described in [26], which we will also explore further in section 3.4.

2.2 Path planning algorithms breakdown

According to [8], path planning algorithms can be broken down into 4 different categories as follows:

- Node-based algorithms
- Mathematical model-based algorithms
- Sampling-based algorithms
- Bioinspired algorithms

2.2.1 Node-based algorithms

The main characteristic of these algorithms is that they all explore through a set of points in a graph to find the optimal path, hence why they are sometimes also referred to as discrete optimal planning [27]. Note that this requires that the underlying scene is broken down into a fixed number of nodes (e.g. split the input image into fixed-sized cells), and the weights of arches connecting the nodes (i.e. connections between adjacent cells) are computed.

One of the oldest and most well-known node-based algorithms is **Dijkstra's** shortest path algorithm [15], which computes the optimal path between a specific source node and all the other nodes in the graph as seen in the pseudocode in algorithm 1. Intuitively, we can see that the number of nodes visited is much higher than the overall length of the optimal path due to the breadth-first nature of the algorithm. In contrast, Best-First-Search, which plans a path by traversing the graph in a greedy fashion, where every next step is determined based on some heuristic, clearly is less likely to find an optimal path, but no move is wasted as they are all included in the final path.

A* [28] combines the main idea behind Dijkstra's (favoring nodes that are close to the starting point) and Best-First-Search (favoring nodes that are close to the goal) by employing a heuristic estimate. It does so via an estimated path cost $h(n)$ in combination with $g(n)$, the path cost from the starting node to the current node n. These are used to determine the node with the lowest $f(n) = g(n)+h(n)$ so it can be visited first. This enables A* to have better performance on average than Dijkstra's because it allows skipping certain nodes with a worse estimated path cost as seen

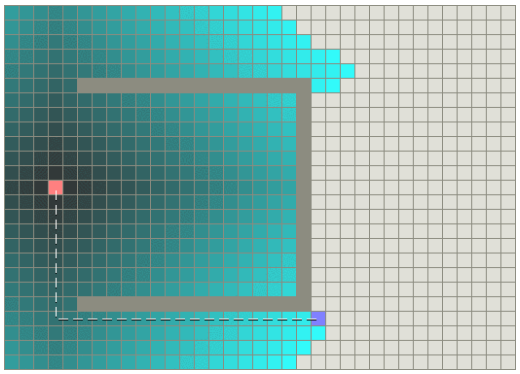
Algorithm 1 Dijkstra's path planning algorithm

```
1: procedure DIJKSTRA(G,W : weighted graph; start : node; finish : node)
2:     tree[start] = true
3:     for x in adj[start] do:
4:         fringe[x] = true; parent[x] = start; distance[x] = W[start, x]        ▷ Initialisation
5:     while not tree[finish] and fringe nonempty do:
6:         f = GetMin()                          ▷ Obtain fringe node with minimal distance[f]
7:         fringe[f] = false; tree[f] = true              ▷ Move f to the tree from fringe nodes.
8:         for y in adj[f] do
9:             if not tree[y] then
10:                if fringe[y] then
11:                    SeenUpdate()    ▷ Update distance and parent arrays if shorter path found.
12:                else
13:                    UnseenUpdate()    ▷ Add y to fringe nodes, update its distance and parent.
       return distance, parent
```

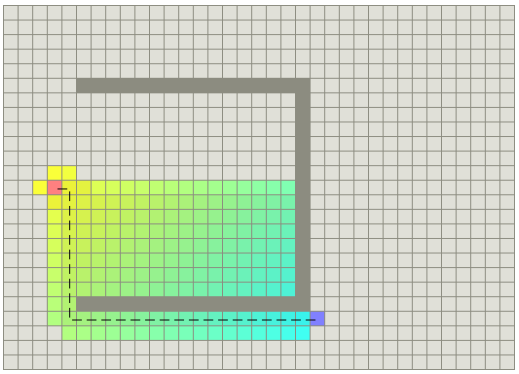
in the comparison in fig. 2.8 (i.e. there is no point visiting a node which is known to lead to a less optimal path). Note that the choice of the heuristic significantly impacts the rate of improvement compared to Dijkstra's algorithm, and it also must be admissible such that $h(n) \leq h^*(n)$, where $h^*(n)$ is the optimal path cost (i.e. the heuristic does not overestimate the effort to reach the goal) to ensure that the optimal path is found.

Over the years, some alternative methods have emerged based on A*, such as LPA*[29] or Focused D*[30], which improve the performance of A* in certain settings. However, similarly to A*, they all rely on the fact that the scene is decomposed into a graph with a discrete set of nodes, so every change in the environment requires that the graph is rebuilt (from potentially noisy sensor readings), which significantly hinders the usability of these algorithms in practice. Considering real scenarios, such as a robot in a complex high dimensional space, the computational complexity of node-based algorithms is not sufficient.

Figure 2.8: Comparing the number of nodes visited in Dijkstra's shortest path algorithm and A* as found in [5]. The start and goal are represented with pink and purple squares respectively.



(a) **Dijkstra's shortest path algorithm:** No notion of path cost estimation, so all neighbouring nodes are visited resulting in potentially unnecessary visits.



(b) **A* algorithm:** Path cost estimation heuristic allows skipping visits of nodes that cannot be parts of the optimal path.

2.2.2 Mathematical model-based algorithms

The most widely used model-based algorithms fall under linear programming-based and optimal control-based methods. Both of these model the environment (kinematic constraints) and the system (dynamic constraints), and use the acquired constraints to bound the cost function in order to minimise it and therefore find the optimal path.

Linear-programming methods, such as MILP [6] formulate the path planning problem of minimising a cost function bound by linear equality and inequality constraints. The cost functions

to be optimised in these scenarios can for example be the path length or fuel consumption. These methods are very expressive as the constraints can include various physical limitations on the movement of the robot as well as restrictions enforced by the environment itself, such as obstacles [31]. This is a major upside when considering the use of these algorithms in real-life scenarios compared to node-based approaches, where incorporating the dynamics of the robot can lead to an overly complex problem. **Optimal control-based approaches** formulate the path planning problem in the form of differential equations. Similarly to linear-programming methods, these can efficiently represent a large variety of constraints.

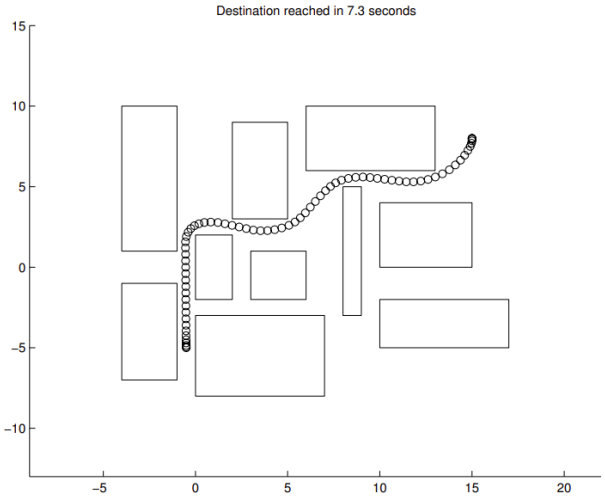


Figure 2.9: Fuel-optimal shortest path of unmanned vehicle in a 2-D environment presented in [6].

Due to their great expressiveness, these approaches can be considered for real-time scenarios, but they often suffer greatly from poor inference speed for large problems. While there have been some approaches to reduce the inference time (e.g. by breaking down the problem into subproblems [32]), these still fall behind other sampling-based methods discussed in section 2.2.3.

2.2.3 Sampling-based algorithms

This class of algorithms utilises random sampling of points in a pre-defined obstacle-free space to build a path or a network of nodes containing paths from the starting point to the goal. Due to the efficiency of sampling random points, these algorithms can find feasible paths relatively efficiently even in high-dimensional continuous spaces. Yang et al.[8] further divide these algorithms into active (can achieve the best feasible path all on its own) and passive (generate a road network containing multiple feasible paths and a separate node-based algorithm is used to discover the optimal path).

Rapidly exploring random trees (RRT)

This method, first proposed by LaValle [33] in 1997 can swiftly discover a feasible path between two points x_{init} and x_{goal}, given an obstacle space X_{obs}, obstacle-free space X_{free} and maximum step size γ. The algorithm builds a tree by performing the following steps repeatedly:

1. Randomly sample a point x_{rand} in X_{free}.
2. Find the nearest tree node x_{near} to x_{rand}.
3. Attempt to connect x_{rand} and x_{near}. If the distance between these two nodes is greater than γ, create a new node x_{new} along the vector (x_{near}, x_{rand}) starting from x_{near}. If $x_{new} \in X_{free}$, connect x_{new} and x_{near}, otherwise do not connect any nodes.

While RRT does a fine job of discovering a feasible path between two nodes, a few improvements have been developed to tackle its main shortcomings. Dynamic Domain RRT (DDRRT) [7] tackles

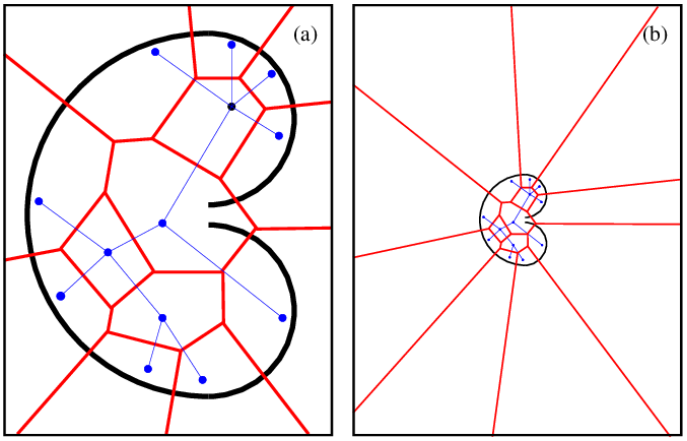
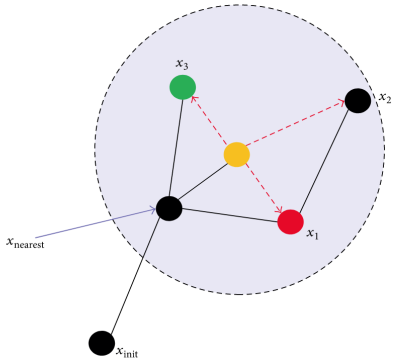


Figure 2.10: Illustration of the bug-trap problem and the tree built by RRT with its corresponding Voronoi regions addressed in [7]. Here we can see that escaping the trap becomes much harder for RRT once the sampling area is increased because the likelihood of the sampling point falling inside the trap is much smaller.

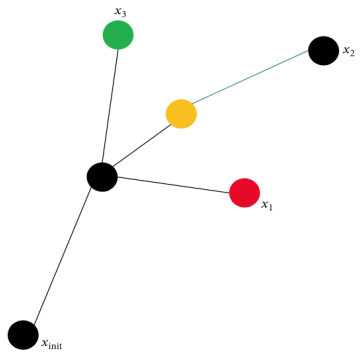
the issue of Voronoi bias of RRT when sampling. DDRRT introduces spheres around the nodes of the graph, and it only attempts to connect those randomly sampled points to the tree which lie within the sphere of the corresponding nearest node, ensuring fast exploration.

An even bigger disadvantage of RRT is that it has been proven to be not asymptotically optimal [34]. For this exact reason, RRG and its tree version, RRT* [35] were developed. RRG differs from the original RRT in the connection phase, as in RRG the new node x_{new} (i.e. the one being added to the graph) is not only connected to the nearest node, but to every other node within a certain radius. By connecting more nodes at the same time, a complex network is generated, which is proven to be asymptotically optimal. Similarly to PRM [36], this method produces a network of nodes, therefore it requires a secondary node-based path planning algorithm to find the optimal path, but research suggests that RRG performs better in a general setting than PRM [37]. RRT* applies a similar logic during the connection phase, with the only difference being that in order to maintain a tree structure, only those nodes are connected to x_{new}, where the path going through x_{new} would prove to be better than their current path (so-called pruning phase).

Figure 2.11: Demonstration of the RRT* pruning phase in the work of Yang et al. [8].



(a) **Connection phase:** Similarly as in RRG, besides $x_{nearest}$, all the nodes within a certain radius are considered for connections with x_{new} .



(b) **Pruning phase:** Maintain the tree structure by pruning away those edges, which would lead to higher cost paths.

Compared to the path planning methods mentioned previously, RRT* tackles all main issues of node-based algorithms in high-dimensional spaces, since due to sampling, there is no need to extract a separate graph representation of the underlying scene, and it can find feasible paths very quickly. However, it has also been shown that RRT* requires an infinite amount of time to find the optimal solution [38] and suffers from memory issues as it continues to explore the entire space

[39].

2.2.4 Bioinspired algorithms

Bioinspired algorithms aim to imitate the way humans or other natural beings think or behave. Their goal is to build systems, which can learn to perform tasks just as a human would. These techniques are further split into Evolutionary algorithms and Neural Network-based algorithms. The basic steps of an evolutionary algorithm for path planning were proposed by Jia et al. [40]. In the initialisation phase, the algorithm is provided with a number of feasible paths which serve as the first generation. Feasible paths can be efficiently generated using RRT as in [41]. This is followed by a cycle of evaluation, parent selection, mutation phases, where the paths are continuously evaluated, the best performing paths are kept, and the information between them is shared to produce the next generation. The best individuals remaining at the end of this cycle are then decoded and returned as optimal paths.

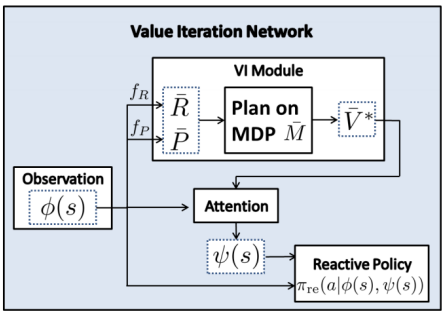
2.3 Neural network-based methods

With the rapid rise in popularity of deep learning in recent years, several research initiatives have been taken to apply deep learning approaches to the problem of path planning. These utilise a variety techniques, such as reinforcement learning [42, 43, 44, 45], supervised learning [11, 46] or imitation learning [47, 48] to provide an alternative to more traditional path planning algorithms discussed in section 2.2, with main focus on use in real-time scenarios in high-dimensional environments. In the following chapter, we present some of the main methods developed in this domain and aim to point out their shortcomings, which our method improves on.

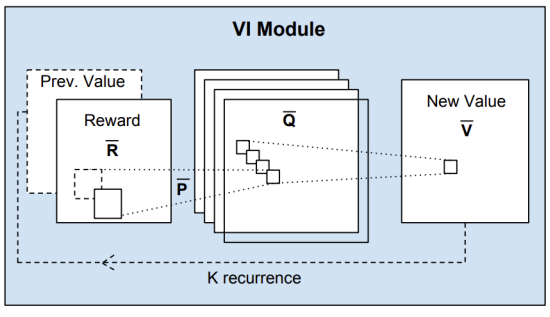
2.3.1 Value Iteration Networks (VINs)

VINs, introduced by Tamar et al. [9], build on the concept of traditional value iteration in reinforcement learning, where an RL agent learns the value of each state in the problem MDP in order to identify the actions with the lowest cost in a given state to produce the optimal path. To deal with situations where the underlying MDP and perhaps even the reward function is unknown, [9] introduces a differentiable value iteration module inside the planning network, which learns some other MDP different from the actual one but that provides useful plans for the actual problem. This is crucial because at no point do VINs require any knowledge about the true reward maps or value maps, we simply hope that by performing actions based on the VI module, the neural network learns a useful MDP. This method certainly shows promise as it generalises well on various tasks within a grid world as shown in [9], but it is currently unclear how it performs in higher-dimensional environments.

Figure 2.12: Overview of the Value Iteration Network model architecture presented in [9]



(a) The VIN architecture as a whole in an RL setting. The VI module predictions and real-world observations are used to decide a policy. f_R and f_P are learned as part of the policy learning process to facilitate the connection between $\overline{M}$ (MDP of the VI module) and M (true MDP).



(b) The inner VI module following a NN architecture. Each VI iteration is seen as passing the previous value function and reward function through a convolution and max-pooling layer. Thus, each convolution layer layer corresponds to the Q-function for a specific action.

2.3.2 Imitation Learning

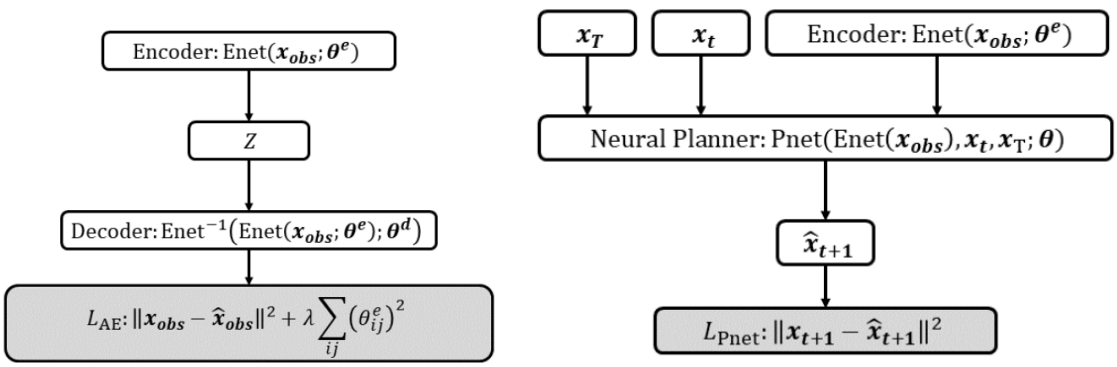
When it comes to path planning problems, the optimal path is often heavily influenced by the implicit structure of the environment (i.e. configuration of objects/obstacles in the environment). Most current state-of-the-art methods don't exploit this property as they aim to sample the entire action space, and therefore they often struggle to converge in a reasonable time and are susceptible to falling into poor local minima. Imitation learning-based solutions were introduced to tackle this issue by training neural networks using expert demonstrations, giving the model additional information about certain areas which might be more beneficial to explore than others. This also makes them suitable to combine with sampling-based algorithms, which can benefit greatly from improved sampling heuristics. One such approach found in [47] uses deep neural networks trained via expert demonstrations to adaptively sample the configuration space for sampling-based algorithms. The biggest drawback of imitation learning-based techniques is that they rely on the presence of expert demonstrations, making them difficult to train for many real-world robotics scenarios. Additionally, these demonstrations are produced via interactions with the environment, which may not be readily available and can bias the training dataset to contain few datapoints on the areas around the edges of obstacles as shown in fig. 2.15.

2.3.3 MPNet

Motion Planning Network (MPNet), introduced by Qureshi et al. [10, 11] uses two neural networks to solve the path planning problem. The first network is an **encoder (ENet)**, which embeds the obstacles provided as point clouds into latent space. This makes it readily applicable in robotics, where the image of the environment is usually obtained via numerous sensor readings frequently represented as point clouds. The second model is a **planning network (PNet)**, which learns to predict path segments using the obstacle point encodings produced by ENet.

ENet can be trained in an end-to-end fashion with PNet or as a contractive autoencoder [49] in an encoder-decoder architecture, where a reconstructive loss is used to learn a robust and invariant feature space required for planning and genalisation to unseen workspaces. PNet is trained via supervised learning as a feedforward neural network. Given the obstacle representation, start and goal points, PNet predicts a point representing the next position of the robot, therefore the training objective is to minimise the mean-squared-errors between the predicted points and the points contained in the ground truth path. These can be provided as an expert demonstration by a human or generated by other highly accurate methods, such as RRT*.

Figure 2.13: Overview of the training setup of both networks in the offline phase as proposed in [10].



(a) The encoder network (ENet) in the encoder-decoder architecture. The reconstructive loss has 2 components: mean-squared-error of obstacle point reconstructions; a penalising component based on the size of the encoder parameters θ^e.

(b) PNet takes the current point x_t, the end goal x_T and the obstacle encoding provided by ENet to predict the next step $\hat{x}_{t+1}$ in the path. The mean-squared error between this point and the point predicted by the oracle is used as training loss.

The actual online path planning process of MPNet utilises the trained networks in combination

with a non-model-based approach. Here, the outputs of ENet and PNet are used repeatedly in a bidirectional fashion to produce a path between two points in multiple steps:

1. Global planning: Create a path by marching from the start and goal points in both directions using the predictions provided by PNet, connecting points, which do not collide with the environment.

2. Neural replanning: Uses Pnet recursively on the non-connectable consecutive states produced in the global planning phase.

3. Steering: Connect the new points generated in the replanning phase to form a fully connected path. This may need to be performed multiple times depending on the success of neural replanning.

4. Lazy states contraction: Prunes out redundant states making the path more efficient.

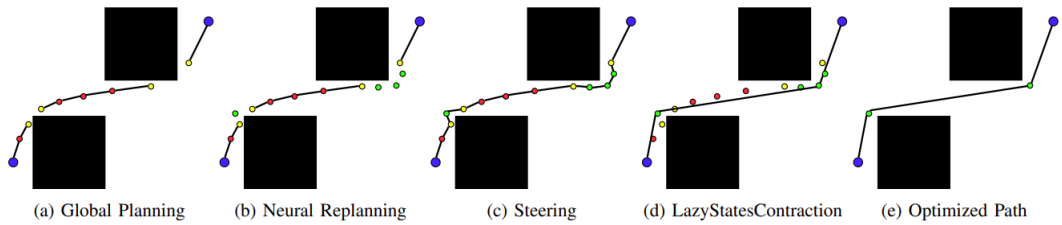


Figure 2.14: Visualisation of each planning step depicted in the original paper [11]

In terms of performance, in the original paper [10], MPNet was shown to outperform state-of-the-art sampling-based algorithms in unseen 3D environments and 7 DOF robot manipulators, providing a much better inference speed while maintaining near-optimal path costs. Its two main weaknesses are a long training time (around 15 hours) and its supervised learning nature.

2.3.4 Other methods

OracleNet introduced by Bency et al. [16] proposes an algorithm utilising recurrent neural networks constructed of stacked LSTM layers [50]. Similarly to MPNet, the final path is produced via bidirectional path planning, where the trained network outputs the individual path steps. Its biggest disadvantage compared to MPNet is that no obstacle information is used during training, therefore it needs to be retrained for every new scene, limiting its use to largely static spaces (e.g. car factories, airport terminals).

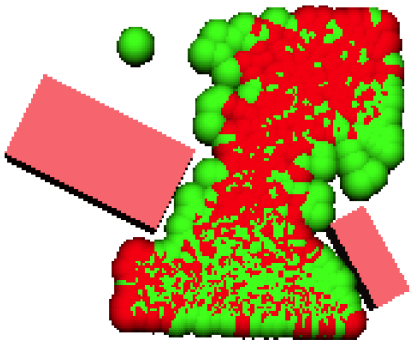


Figure 2.15: Jurgenson et al. [12] visualising the data distribution on edges of obstacles. Green spheres were generated by an RL agent during training, while red spheres are imitation learning targets produced by a path planner network. Green spheres are dominant near obstacle boundaries pointing to a lack of data that imitation learning agents have on the obstacles.

The work of Jurgenson et al. [12] highlights an interesting issue of imitation learning and supervised learning-based algorithms. According to [12], the fact that these methods use expert demonstrations to learn optimal paths is troublesome in environments with tight passages. In these situations, the expert demonstrations strictly avoid hitting any obstacles, which leads to an underrepresentation of obstacle boundaries in the dataset, so there is no guarantee that the model will truly learn to avoid obstacles. To solve this, in [12] a deep deterministic policy gradient algorithm [51] is used, training a model via interaction with the environment. While this solution tackles the proposed issue, interacting with the environment for so long is not realistic in many path planning scenarios.

2.4 Meta-learning

Meta-learning [52, 53, 54] is a concept, which takes a novel approach to tackling traditional machine learning problems. It draws from the natural ability of humans to adapt and quickly learn new tasks as they progress with age. The idea here is that at a young age, we learn a variety of basic concepts and skills, and this experience allows us to later master even harder tasks in a relatively short amount of time. The goal of Meta-learning is to build machine learning models with this exact property. Upon success, this idea could significantly improve on training times and the amount of data required, both being a major disadvantage of current state-of-the-art deep learning-based planners. In this chapter, we introduce the most promising solutions published recently in this domain relevant to our path planning optimisation problem.

2.4.1 MAML

Model-Agnostic Meta-Learning (MAML) [13] is a fairly general optimisation algorithm applicable on any model trained via gradient descent. In [13], its adaptability was demonstrated on multiple different architectures, including classification, regression, and policy gradient reinforcement learning. In contrast with popular meta-learning methods [55, 56, 57], its objective isn't to learn the update function or learning rule, so the number of learned parameters is not increased in the process. Additionally, it doesn't constrain the model architecture, as it is in the case of a Siamese network [58].

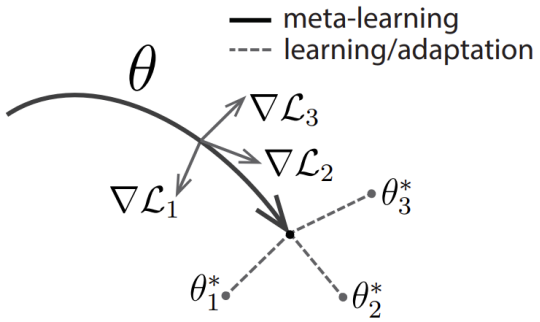


Figure 2.16: Diagram of MAML found in [13] depicting the main learning objective θ and its ability to quickly adapt to new tasks with only a few gradient steps.

Its main objective can be summarised from a feature learning standpoint as constructing an internal representation broadly applicable to many tasks and with the ability to produce good results via a few or even just a single gradient step. From a dynamic systems standpoint, it can be seen as maximising the sensitivity of the loss functions of new tasks with respect to the model parameters, to ensure that small changes to the parameters can lead to big improvements on these new tasks. The only assumptions made are that the model is parameterised by a vector θ, and the loss function is smooth enough in θ that gradient-based learning techniques can be applied.

The training process itself is characterised by a nested loop structure, where both the outer and inner loop use batches of data to perform updates to a different set of parameters. More specifically, the inner loop performs the **adaptation** of the model parameters θ to a specific task $\mathcal{T}_i$ sampled from a distribution of tasks $p(\mathcal{T})$ to produce the parameter vector θ_i' via a gradient

update from θ. The outer loop then performs a gradient update to θ using the gradients of the losses of all the tasks $\mathcal{T}_i$ parameterised by their respective θ_i'. This outer loop update is called the **meta-update**. The overall optimisation process can then be summarised as the following minimisation problem:

$$\min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(\theta_i') = \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(\theta - \alpha \Delta_\theta \mathcal{L}_{\mathcal{T}_i}(\theta))$$

With this objective in mind, the model parameters are updated via stochastic gradient descent as can be seen in the pseudocode of MAML in algorithm 2. This way of updating θ involves computing a gradient through a gradient, therefore it requires an additional backward pass, which hinders its performance somewhat and is later addressed by an alternative method in 2.4.2.

Algorithm 2 Model-Agnostic Meta-Learning [13]

procedure MAML($p(\mathcal{T})$: distribution over tasks; α, β : step size hyperparameters)
  Randomly initialise θ
  while not done **do**:
    Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
    for all $\mathcal{T}_i$ **do**
      Evaluate $\Delta_\theta \mathcal{L}_{\mathcal{T}_i}(\theta)$ with respect to K examples
      $\theta_i^{'} = \theta - \alpha \Delta_\theta \mathcal{L}_{\mathcal{T}_i}(\theta)$ ▷ Compute parameters for new tasks using gradient descent
    $\theta \leftarrow \theta - \beta \Delta_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(\theta_i^{'})$ ▷ Update the model parameters using gradient descent

Overall, MAML proves to be a good starting point for many meta-learning implementations due to its versatility and simplicity. [13] shows that in many settings, it can either rival or outperform other approaches of its kind and considering that using a reinforcement learning model, it actually outperforms traditional RL methods for 2D navigation and locomotion tasks in terms of required gradient steps, its application in path planning appears promising.

2.4.2 First-order meta-learners

As mentioned in 2.4.1, MAML's parameter updates are computed via a second-order derivative of gradients, increasing its computational complexity significantly. First-order MAML (FOMAML) solves this issue by ommiting the second derivatives. The reason why this is possible can be derived by examining the gradient g_{MAML} as part of the $\theta = \theta - \beta g_{MAML}$ outer loop gradient update:

$$\begin{aligned}
g_{MAML} &= \Delta_\theta \mathcal{L}^{(1)}(\theta_k) \\
&= \Delta_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot (\Delta_{\theta_{k-1}} \theta_k) \cdot ... \cdot (\Delta_{\theta_0} \theta_1) \cdot (\Delta_\theta \theta_0) \qquad \text{by applying chain rule} \\
&= \Delta_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \Pi_{i=1}^{k} (\Delta_{\theta_{i-1}} \theta_i) \cdot I \\
&= \Delta_{\theta_k} \mathcal{L}^{(1)}(\theta_k) \cdot \Pi_{i=1}^{k} I - \alpha \Delta_{\theta_{i-1}} (\Delta_\theta \mathcal{L}^{(0)} \theta_{i-1})
\end{aligned}$$

Thus by ignoring the second derivative term we get:

$$g_{MAML} = \Delta_{\theta_k} \mathcal{L}^{(1)}(\theta_k)$$

This approximation of the original method lead to a 33% improvement in computational speed and performed almost identically on the MiniImageNet dataset in a few-shot classification setting [13].

Another remarkably simple yet effective algorithm is Reptile, developed by Nichol et al. [59]. The biggest difference here is that instead of performing a gradient descent update in the outer loop, the model parameters are simply moved towards the new parameters. The parameter update is defined as: $\theta \leftarrow \theta + \beta \frac{1}{n} \sum_{i=1}^{n} (\theta_i' - \theta_i)$, where θ_i' represent the new parameters trained via multiple SGD steps in the inner loop. In [59], Nichol et al. provided a theoretical explanation why this method works by approximating it with a Taylor series, but due to the lack of successful experiments provided, its practical usefulness remains to be seen.

2.4.3 Meta-SGD

While all the methods mentioned previously learn the parameters of a model to serve as initialization for adapting to a specific task, Meta-SGD [14] publishd by Li et al. promises higher capacity, by "learning to learn" the learner update direction and learning rate as well in a single supervised learning or reinforcement learning process. The formulated minimisation problem is similar to MAML and takes the following form:

$$\min_{\theta,\alpha} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{meta(\mathcal{T}_i)}(\theta_i') = \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{meta(\mathcal{T}_i)}(\theta - \alpha \circ \Delta \mathcal{L}_{adapt(\mathcal{T}_i)}(\theta))$$

Here $\mathcal{L}_{meta}$ and $\mathcal{L}_{adapt}$ represent the loss functions used during the meta-update and adaptation updates respectively and $\circ$ is the element-wise multiplication operator. This objective is differentiable with respect to both θ and α as well, which means that it is solvable with a similar gradient-based method as MAML. In [14], Li et al. demonstrate that Meta-SGD performs marginally better on the same benchmarks that MAML [13] was evaluated on, so it could prove to be useful for our use-case as well.

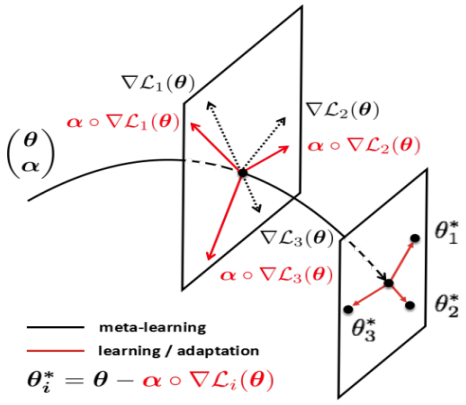


Figure 2.17: Diagram of Meta-SGD [14] illustrating its two-level learning process. Gradual learning is performed across various tasks to train the meta-learner model parameterised by (θ, α). Similarly to MAML, this meta-learner can perform rapid adaptations to specific tasks, with the main difference being the learned α, which promises to further accelerate the adaptation process.

2.5 Summary

Our effort to research the background and existing solutions in the domain of path planning can be summarised in the following points:

- Traditional path planning algorithms focus mainly on finding optimal paths and do not place a high emphasis on inference speed, making them difficult to use in practice, as their performance is hindered by increasing the complexity of the environment in general. Sampling-based solutions seem to offer some flexibility when it comes to the path optimality/computational speed trade-off, but in complex environments, even these require time in order of 10s.
- Neural network-based solutions provide a great alternative to the traditional methods. Their inference speed and ability to learn to predict paths in high-dimensional spaces is particularly attractive for practical applications. Their biggest downside is that they often require a large number of expert demonstrations, take a long time to train [11] and might require re-training for different environments [16].
- Meta-learning is a relatively new subfield of machine learning, which promises to solve the main issues outlined above. It does so by training models which can generalise to new tasks quickly with only a few required training steps. Current research on this topic shows that there is merit in applying meta-learning to the problem of path planning.

Chapter 3

Approach

In the following chapter, we present our solution on using meta-learning to refine and improve the path planning neural networks (PNet) presented in [10]. We start off by introducing a loss function, which utilises a signed distance function (SDF) compatible with the PNet dataset, and show that it corresponds well to the task (i.e. finding the shortest collision-free path) and thus can be used for deep learning-based path-planning. We follow up by presenting our formulation of applying Model-Agnostic Meta-Learning (MAML) to PNet. Lastly, we showcase a number of improvements/modifications to the initial solution.

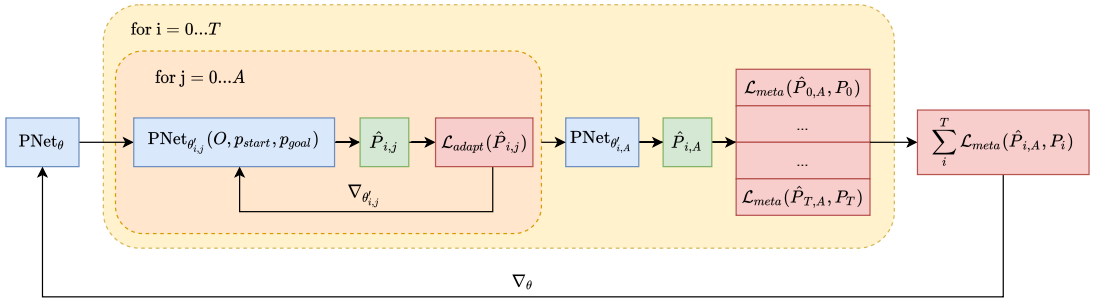


Figure 3.1: Diagram of our MAML meta-learning pipeline. PNet is adapted to T environments for A number of adaptation steps per environment using $\mathcal{L}_{adapt}$, yielding networks PNet$_{\theta'_{i,A}}$ where $i \in [0, T]$. The weights θ of PNet$_\theta$ are then updated via the sum of $\mathcal{L}_{meta}$ taken from each of the adapted networks.

3.1 Refining MPNet via unsupervised loss

One of PNet's main disadvantages is the dependence on expert demonstrations or paths generated via other methods (e.g. RRT, RRT*) as it's trained in a supervised fashion. While this is not catastrophic for standard PNet at test time, our meta-learned PNet will be trained to rapidly adapt to a given task, therefore it will have to perform some kind of learning at test time. This adaptation cannot be reasonably performed via supervised learning, especially in environments unseen during training, where we may not have any ground truth information available. For instance, in a practical scenario, where a robot is required to navigate in a disaster-struck area, it must be able to adapt its planner network to the new environment just from the scene point-clouds obtained via lidar or other sensor measurements. To accommodate for this requirement, we look for a way to define a function that allows for training/refining PNet in an unsupervised fashion.

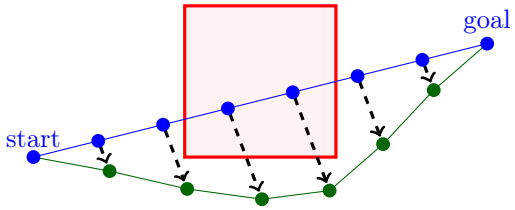


Figure 3.2: The collision feedback term $c(x)$ guides the network parameters θ so that the predicted paths are further away from the obstacles by minimising the SDF-based loss. Paths generated after a couple of gradient steps (green) avoid collisions much better compared to the initial prediction (blue).

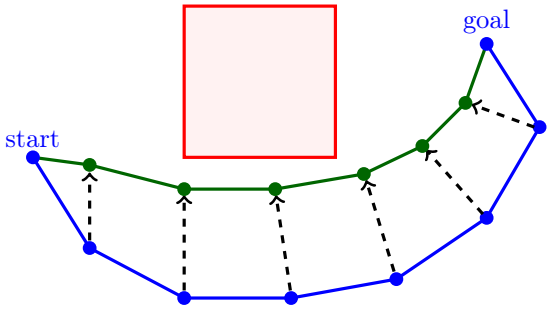


Figure 3.3: The path length term $l(x)$ promotes parameter updates to θ so that the predicted paths have minimal length. It is useful for exploiting possible shortcuts and restricting the amount of unnecessary collision avoidance cause by $c(x)$.

We divide PNet's task into two main objectives: find a collision-free path; find the path with minimal length. Based on this assumption, a loss function can be defined as a sum of two terms, each of which provides feedback on one of the two main objectives. In this report, the notation used for the path length feedback and collision feedback is $l(x)$, $c(x)$ respectively, where x is the input containing only information about the scene and the predicted path. Figure 3.2 and fig. 3.3 demonstrate the intuition behind these two terms and how they can contribute to improving on both objectives. The path length feedback can be easily computed as the sum of Euclidean distances between intermediate points in the path as given in eq. (3.1) where $p_1 ..., p_n \in P$ is the set of path points. Deriving $c(x)$ requires more careful consideration and is discussed in detail in section 3.1.2.

$$l(x) = l(P) = \sum_{i=0}^{n-1} ||p_i - p_{i+1}||_2 \tag{3.1}$$

3.1.1 Path predictions using PNet

As described in section 2.3.3, PNet is trained to predict the next point in the path from a given location and uses a combination of network outputs and other heuristics to produce full paths. Since we intend to apply our unsupervised loss function on the path-level, the paths are obtained by directly concatenating the points predicted by PNet in an iterative fashion. We realise that dismissing the heuristics used in the MPNet paper [10] might hinder the quality of the paths, but these would increase the inference time and would make batch predictions impossible, both of these factors being crucial later during meta-learning. To guarantee completeness, if the points outputted by PNet don't end up being close enough to the goal in a certain number of prediction iterations, we append the goal point as the last path point. In practice, paths are generated in batches and optionally padded with their respective goal points in case the goal was reached in a smaller number of steps. The whole procedure is outlined in pseudocode algorithm 3.

3.1.2 Collision feedback term derivation

The criteria that this term needs to satisfy is that the higher the collision percentage of the path (i.e. percentage of the path colliding with the environment), the higher the cost assigned to it. This can be achieved via an SDF-like formulation, where we consider every point in the point-cloud to be a separate object and assign the cost based on the negative distance of each path point from the closest object. Note that in order to compute this loss uniformly across the entire path and to eliminate potential uneven distribution due to padding during batch predictions, we apply polyharmonic interpolation [60] to the predicted path points and sample new points equidistantly instead of using the PNet predictions directly. In addition to evenly distributed path points, we use the sigmoid function to normalise each negative distance term and to prevent assigning

Algorithm 3 Batch path generation using PNet

procedure GENERATEPATHS(PNet: planner network, P: paths to be generated)
  $\text{paths}_i \leftarrow [\text{P}_{i,\ \text{start}}]$, i = 0, 1...length(P) ▷ Initialise path start points
  $\text{goal\_reached}_i \leftarrow$ false, i = 0, 1...length(P) ▷ Set goal not reached for every path
  curr_step ← 0
  while not all(goal_reached) and curr_step < max_steps **do**:
    curr_step ← curr_step + 1
    $\text{pnet\_input}_i \leftarrow [\text{P}_{i,\ \text{scene\_encoding}}, \text{P}_{i,\ \text{start}}, \text{P}_{i,\ \text{goal}}]$, i = 0, 1...length(P)
    point_predictions ← PNet(pnet_input)
    for i in range length(P) **do**
      if not goal_reached_i **then**
        $\text{paths}_i \leftarrow$ append $\text{point\_predictions}_i$
      else
        $\text{paths}_i \leftarrow$ append $\text{P}_{i,\ \text{goal}}$ ▷ Pad paths with the goal point
      $\text{goal\_reached}_i = ||\text{point\_predictions}_i - \text{P}_{i,\ \text{goal}}||_2 < \epsilon$
  for i in range length(P) **do** ▷ Append goal point to to incomplete paths
    if not goal_reached_i **then**
      $\text{paths}_i \leftarrow$ append $\text{P}_{i,\ \text{goal}}$

unnecessarily large loss values to points, which are sufficiently far from obstacles. Equation (3.2) describes the full formulation of $c(x) = c(P, O)$, where $p_n \in P$ denotes n-th point in the set of resampled path points and $o_m \in O$ denotes the m-th point in the obstacle point-cloud O. From the heatmap in fig. 3.4 we can see that $c(P, O)$ does a good job of describing the obstacles in the environment.

$$c(P, O) = \sum_{p_n \in P} \text{sigmoid}(- \min_{o_m \in O} ||p_n - o_m||_2) \tag{3.2}$$

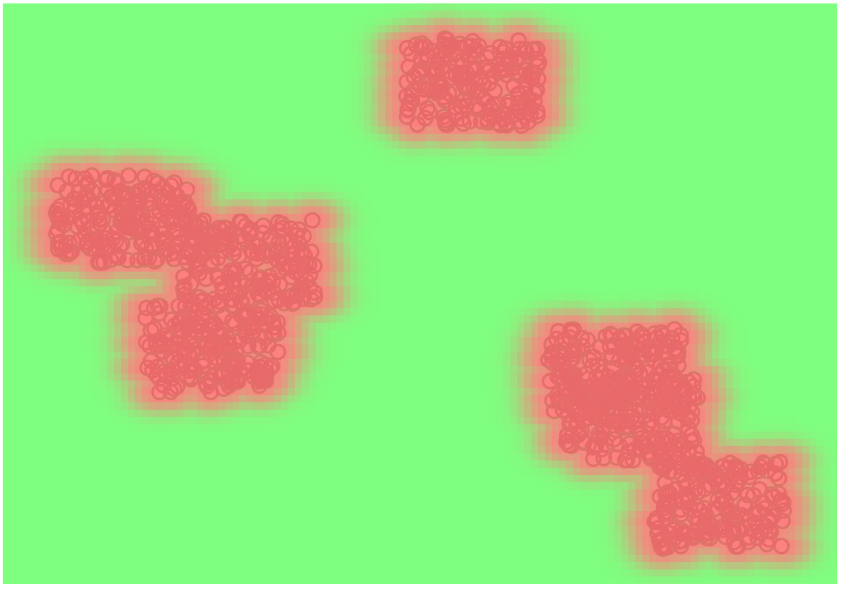


Figure 3.4: Visualisation of the values of the collision term computed for each point in the scene i.e. $\text{sigmoid}(- \min_{o_m \in O} ||p - o_m||_2), \forall p \in E$, where E is the set of all points in the environment. The areas near the obstacle point clouds visibly have a much higher cost associated to them, pushing any path containing points in the red regions away from the obstacles.

3.1.3 Loss function formulation

Finally, $l(x)$ and $c(x)$ can be combined into one loss function, which will be used in the MAML inner loop during training and for adaptation at test time. A weighting factor ρ is also introduced to balance the two different terms. The final formulation is given in eq. (3.3). Quantifiable evidence of the overall effectiveness of this function to refine the weights θ of PNet is further discussed in section 4.2.

$$f(x) = f(P, O) = l(P) + \rho * c(P, O) \tag{3.3}$$

$$= \sum_{i=0}^{n-1} ||p_i - p_{i+1}||_2 + \rho * \sum_{p_n \in P} \text{sigmoid}(- \min_{o_m \in O} ||p_n - o_m||_2) \tag{3.4}$$

3.2 Applying MAML to MPNet

After establishing the loss function used to adapt the planner network at test time, we build on it by placing it in a training pipeline in accordance with the one described by the pseudocode in algorithm 2. A summary of how this general framework is applied to our specific task can be found below, along with the description of some aspects in which we deviate from the original MAML [13] formulation.

3.2.1 Planner network architecture and initialisation

Since our objective is to improve on the predictive power of PNet, we use the same network architecture for ENet (environment encoder network) and PNet, as provided by the authors of MPNet in the open-source implementation [18]. While in [13] the weights θ are initialised randomly, considering the already long training time of PNet combined with the added time complexity of computing the unsupervised loss and using entire paths for training, we have decided to initialise θ with pre-trained PNet weights obtained via the same training setup as found in [18] and depicted in fig. 2.13.

3.2.2 MAML training setup

To draw a parallel with the general formulation from algorithm 2, the environments in which planning is performed are interpreted as the set of tasks $\mathcal{T}$ and a path between two points is considered as one training example for a given task. When it comes to the choice of loss functions, we deviate from the standard approach by introducing separate losses $\mathcal{L}_{adapt}$ for the adaptation steps and $\mathcal{L}_{meta}$ for the meta-update instead of using the same for both. The unsupervised loss described in section 3.1 is used as $\mathcal{L}_{adapt}$ and the supervised mean-squared-error loss is put in place as $\mathcal{L}_{meta}$ to help in more complex planning scenarios, where $\mathcal{L}_{adapt}$ comes up short (discussed in section 3.4). Note that $\mathcal{L}_{meta}$ is only used during training, where we allow leveraging ground truth data. Furthermore, since $\mathcal{L}_{adapt}$ is unsupervised, the same set of data can be used for the adaptation steps as for the meta-update. This also enables adapting PNet to an environment via the same start/goal points as the ones we wish to predict after the refinement at test time. Specific training parameters and implementation details are later listed in chapter 4.

3.3 Alternative methods and improvements

As noted in section 2.4.1, backpropagating through the entire computational graph when computing the gradients of $\mathcal{L}_{meta}$ with respect to the planner network parameters θ can be very computationally intensive, allowing only a small number of adaptation steps to be undertaken during training. To mitigate this problem with little loss of information, we tried two approaches.

First-order MAML, introduced alongside MAML by Finn et al. [13] approximates the gradient ∇_θ in the meta-update by only taking the gradient with respect to the modified parameters θ', removing the memory constraints as no variables have to be tracked between individual adaptation steps. In practice, this means storing $\nabla_{\theta'_i}$ for each task after the final adaptation step and taking the sum or mean of these gradients to update the parameters θ after each task mini-batch

Algorithm 4 Applying MAML to path planning using PNet

procedure PATHPLANNINGMAML(α, β: step size hyperparameters)
  Initialise θ with pre-trained PNet weights
  while not done **do**
    $\mathcal{T}_i \leftarrow \mathrm{P_i}$, i = 0...batch_size ▷ Select batch of environments with paths to generate
    for all $\mathcal{T}_i$ **do**
      for number of adaptation steps **do**
        Initialise adapted network parameters θ'_i with θ
        Evaluate $\nabla_{\theta'_i}\mathcal{L}_{adapt}(\theta'_i, \mathrm{P_i})$ by predicting paths in $\mathrm{P_i}$
        $\theta'_i, \leftarrow \theta'_i - \alpha\nabla_{\theta'_i}\mathcal{L}_{adapt}(\theta'_i, \mathrm{P_i})$ ▷ Update θ'_i on every adaptation step iteration
    $\theta \leftarrow \theta - \beta\nabla_\theta \sum_{\mathcal{T}_i \in \mathcal{T}} \mathcal{L}_{meta,\mathcal{T}_i}(\theta'_i, \mathrm{P_i})$ ▷ Perform meta-update on θ via supervised loss

as shown in eq. (3.5). This way, a similar normalisation behaviour is achieved as taking the sum of meta-losses $\mathcal{L}_{meta}$ during the regular MAML setup.

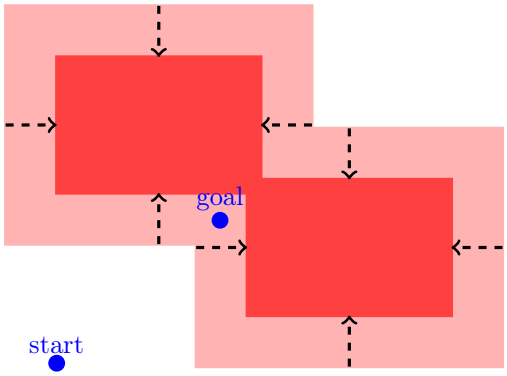
$$\theta = \theta - \beta \sum_{\mathcal{T}_i \in \mathcal{T}} \nabla_{\theta'_i}\mathcal{L}_{meta}(\theta'_i) \tag{3.5}$$

Reptile is another alternative presented by Nichol et al. [59], where instead of performing the meta-update as an additional gradient step using $\mathcal{L}_{meta}$, the planner weights θ are directly pulled towards the adapted planner weights θ' as described in section 2.4.2. However, the absence of $\mathcal{L}_{meta}$ might make it unsuitable for our use-case because if we want to maintain the constraint of using the unsupervised loss function at test time, we will have to fully rely on it during training as well. This key characteristic of Reptile is later critically evaluated in chapter 4.

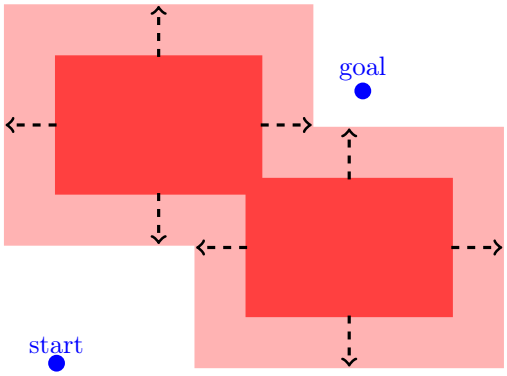
3.4 Unsupervised loss shortcomings

While the loss functions used during the adaptation period and for the meta-update are different, fundamentally they both provide feedback on the same objective i.e. finding the shortest collision-free path. In case of the supervised mean-squared-error, the accuracy of the feedback for a given path prediction depends only on how close the expert demonstrations are to the optimal solution for that path planning problem.

Figure 3.5: Comparing two different path planning problems, where the environment and the start point are the same, but setting the obstacle avoidance weighting based on the goal could be beneficial.



(a) When the goal point is close to the obstacles, but still reachable without any collisions, we would want to reduce the weighting for obstacle avoidance as taking a straight line would be most beneficial.



(b) When the goal point is behind a thin border/obstacle, obstacle avoidance should be increased to avoid taking shortcuts leading to collisions.

The same cannot be said about the unsupervised loss introduced in section 3.1. One thing that can have a significant impact on its output is the weighting factor ρ, which provides the balance

between terms $l(x)$, $c(x)$ i.e. encouraging the network to find shortcuts or to avoid obstacles. One way of finding a good value for ρ is by performing a hyperparameter search, running multiple experiments with different ρ in order to determine the one that leads to best performance. Albeit simple and reliable, this method can be very time-consuming as it can take a fair number of attempts to find an acceptable value. Furthermore, by applying this weighting after $c(x)$ has been computed, each path point is assigned the same ρ. Such lack of flexibility can be an issue in some scenarios, for example when the goal point is close to an obstacle, and we could benefit from reducing the amount of obstacle avoidance in close proximity of the goal. This leads us to the second issue with this initial formulation, which is that the collision term $c(x)$ does not depend on the goal point, meaning that the assigned cost only reflects the short-term effort to avoid obstacles as opposed to providing feedback on the overall objective. Figure 3.5 presents some examples where the initial unsupervised function is simply not sufficient. Based on this reasoning, we attempted two modifications to the initial approach.

3.4.1 Learning a collision residual

As outlined above, having a fixed weight applied after calculating $c(x)$ is suboptimal for multiple reasons, so we seek to find a way to determine a value to complement the collision term of each individual path point, obtaining a more powerful representation. Specifically, we ommit the use of ρ and introduce a residual term $R_{p_n}, \forall p_n \in P$, creating a modified collision term as follows:

$$c_R(x) = c_R(P, O) = \sum_{p_n \in P} \text{sigmoid}(-\min_{o_m \in O} ||p_n - o_m||_2) + R_{p_n} \tag{3.6}$$

$$\mathcal{L}_{adapt} = l(x) + c_R(x) \tag{3.7}$$

To obtain R_{p_n}, we utilise a second neural network further denotes as **RNet**. The inputs to RNet contain the scene embeddings obtained via ENet [10], the point p_n to which the predicted value is assigned to as well as the goal point p_{goal}. The two input points are provided via positional encodings similar to the ones used in [61, 62]. Here the role of the positional encodings is not to include a notion of order as in the transformer architecture [63], but to map continuous input coordinates into a higher dimensional space to help approximate a higher frequency function. The parameters of RNet are initialised randomly, and updates are performed during training by backpropagating through the supervised $\mathcal{L}_{meta}$, meaning that the network is optimised with the ground truth paths in mind. Note that the output of RNet is only applied when computing $\mathcal{L}_{adapt}$, but the adapted weights θ' depend on RNet, maintaining the connection in the computational graph when computing the gradient $\nabla_\phi \mathcal{L}_{meta}$ during the meta-update as shown in the modified pipeline in fig. 3.6.

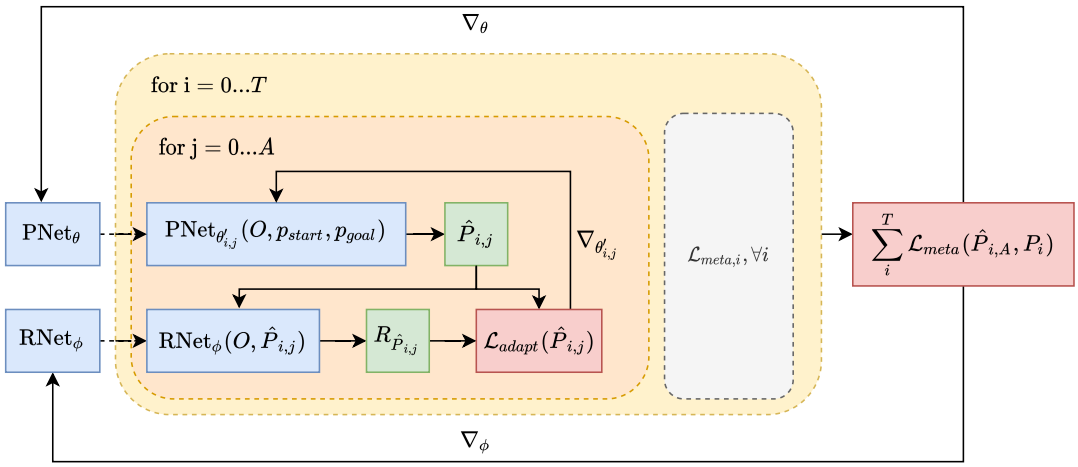


Figure 3.6: Diagram of the updated MAML pipeline with the inclusion of RNet. Here $\mathcal{L}_{adapt}$ is computed using the path predictions $\hat{P}_{i,j}$ as well as the residual terms $R_{\hat{P}_{i,j}}$. These operations make the updated parameters θ' dependent on RNet's weights ϕ, allowing to simultenously backpropagate through $\mathcal{L}_{meta}$ with respect to θ and ϕ during the meta-update.

3.4.2 Learning the collision term

Motivated by the idea of embedding additional information about the expert demonstrations into the unsupervised adaptation period, in our second modification, we replace the $c(x)/c_R(x)$ computation in its entirety with a neural network further denoted as **CNet**. This leads to a new unsupervised loss function formulation in eq. (3.8).

$$\mathcal{L}_{adapt}(P, O, \text{CNet}) = l(P) + \sum_{p_n \in P} \text{CNet}(p_n, p_{goal}, O) \tag{3.8}$$

This not only improves the time complexity of computing $\mathcal{L}_{adapt}$ due to the absence of computing minimum distances across all path and obstacle points, but also allows us to treat CNet as a special SDF optimised for our path planning meta-learning setup. Compared to the residual solution in section 3.4.1, the predicted collision terms $\text{CNet}(x)$ directly correspond to $c(x)$, making them much easier to interpret and draw comparisons. More importantly, this relation means that we can pre-train CNet to predict $c(x)$ before even starting the MAML optimisation process (offline), enabling a much faster convergence rate compared to RNet. Integrating CNet into the pipeline is even simpler than the RNet formulation, as the entire collision term is replaced by a trainable module. Parameter updates are performed in the same way as for RNet, by backpropagating the gradients through $\mathcal{L}_{meta}$.

CNet offline pre-training

The objective here is to pre-train CNet to approximate the output of $c(x)$, which will then be refined in the meta-learning phase. Similarly to RNet, we also include the goal point in the network input to allow predicting different values depending on the specific path. Getting the dataset for this training procedure is simply a matter of generating obstacle encodings via ENet and obtaining paths to compute the ground truth labels $c(x)$. Another great advantage of doing this offline is that we are not constrained to using a relatively small number of different environments, and the provided paths do not have to be efficient collision-free paths as it is during MAML training. To avoid overfitting to only a few environments and prevent bad generalisation performance caused by uneven distribution of path data (paths for training PNet always go around obstacles, but we need to predict collision terms for colliding paths as well), we generate our own dataset by uniformly sampling random start and goal points in several thousand environments. Specific network and training hyperparameters along with examples demonstrating CNet's ability to approximate $c(x)$ are later included in chapter 4.

3.5 Notes on the optimisation process

3.5.1 Path-level performance constraints

Possibly the biggest disadvantage of our method in terms of runtime compared to MPNet is having a path-level approach in the optimisation process. We speculate that providing feedback on the path as a whole should lead to better generalisation across various environments/destinations as opposed to teaching the network to mimic specific points. After all, merely having the predicted points as close as possible to expert demonstrations does not guarantee the collision-free property, especially when navigating tightly around obstacles. However, working with entire paths when only having a network that predicts intermediate points does come with some disadvantages. Below we list two main drawbacks encountered during the implementation of such an optimisation process.

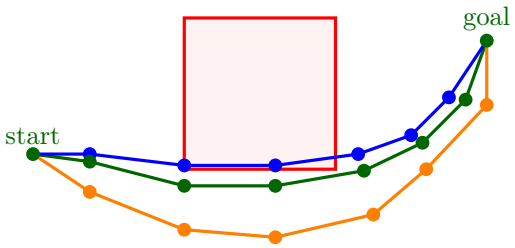


Figure 3.7: The L_2 loss of the blue path points with respect to the expert demonstration (green) is visibly better compared to the orange path. Due to the nature of path planning, the best solutions often narrowly avoid obstacles, so simply aiming to get as close as possible to the ideal path does not always lead to a collision-free outcome.

Path interpolation requirement

As already mentioned previously in section 3.1.2, to compute $c(x)$ and thus $\mathcal{L}_{adapt}$, it is crucial to perform interpolation and resample the path equidistantly. This also applies to the RNet and CNet pipelines too since the concept stays the same, they just provide a more fine-tuned modification to the initial solution. Additionally, the same has to be done when computing $\mathcal{L}_{meta}$ as well for the reasons outlined in the paragraph above and also because we need to ensure that the parameter updates promoted by $\mathcal{L}_{adapt}$ and $\mathcal{L}_{meta}$ are somewhat related to each other, otherwise we will be continuously overwriting the progress made in the adaptation step during the meta-update vice-versa. Despite our attempts to parallelise computation as much as possible, this requirement often negatively affected training times, forcing us to reduce the amount of training data and hindering our ability to iterate on our methods quickly.

Unsupervised loss computational complexity

Examining the definition of the initial unsupervised loss $\mathcal{L}_{adapt}$ and the modifications introduced later, it is perhaps easy to identify certain aspects of it that can incur significant performance penalties. Besides the path interpolation described above, computing the distance between each path point and every obstacle is especially costly in scenarios like ours, where the scene is represented by a point-cloud of many obstacles. The CNet formulation remedies this to some degree, but computing its output and the gradient computation during backpropagation also comes with a significant degree of computational complexity. This issue directly affects the adaptation step time, limiting the number of adaptation iterations performed during training.

3.5.2 Training instability

Based on our experience and backed up by existing research [64], training with a MAML or MAML-like pipeline can be quite unstable and very sensitive to small changes in hyperparameters. In this section, we describe our main efforts to stabilise and aid the convergence of the optimisation process.

Multi-Step Loss

Presented in the work of Antoniou et al. [64], multi-step loss (MSL) is an easily applicable method shown to significantly increase the stability of some MAML pipelines. Motivated by a similar idea as regular batching described above, it modifies the original MAML algorithm by taking the meta-loss $\mathcal{L}_{meta}^{(i)}$ at each adaptation step i instead of taking it only after the last one. The meta-update is then performed on the weighted sum of these losses as seen in algorithm 5, where the weights are gradually annealed over time, to ensure that eventually the adaptation loss taken after the last step is the only significant contributing term. We implement MSL for FOMAML as well, where the difference is taking the weighted sum of the gradients $\nabla_{\theta'}$, since the adapted networks with parameters θ' are discarded every time we finish the adaptations for a given task i.e. environment.

Algorithm 5 MAML for PNet using Multi-Step-Loss

procedure PATHPLANNINGMAML(α, β: step size hyperparameters)
  Initialise θ with pre-trained PNet weights
  while not done **do**
    $\mathcal{T}_{\mathrm{i}} \leftarrow \mathrm{P_i}$, i = 0...batch_size ▷ Select batch of environments with paths to generate
    for all $\mathcal{T}_{\mathrm{i}}$ **do**
      for j $\leftarrow$ 0...A **do** ▷ Perform A iterations of adaptation steps
        Update θ_i' using $\nabla_{\theta_i'} \mathcal{L}_{adapt}(\theta_i')$ ▷ Perform adaptation step update
        Evaluate and store $\mathcal{L}^{(j)}_{meta,\mathcal{T}_{\mathrm{i}}}(\theta_i')$
    $\mathcal{L}_{meta,\mathcal{T}_{\mathrm{i}}} \leftarrow \sum_{j=0}^{A} w_j \mathcal{L}^{(j)}_{meta,\mathcal{T}_{\mathrm{i}}}$, i = 0...batch_size ▷ Weighted sum of losses at each step
    Reduce weights $w_0 ... w_{A-1}$, increase w_A ▷ Anneal the MSL weights
    $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_{\mathrm{i}} \in \mathcal{T}} \mathcal{L}_{meta,\mathcal{T}_{\mathrm{i}}}$ ▷ Perform meta-update on θ via MSL

Batching

Producing multiple network predictions in batches is a commonly used technique to increase inference speed via parallelised computation and to reduce the variance between network outputs, ensuring a smoother learning curve. During training, the losses computed for each training example are averaged within a given batch, reducing the effect of any outliers at the cost of increased memory requirements. In general, the best practice is to use the largest batch size possible to maximise its benefits. As it is such a widely-used method, machine learning libraries support a wide array of operations, including batch network predictions as well as mathematical operations on batches of data, which were particularly useful when implementing our unsupervised adaptation loss. Another type of batching that we utilise in the context of meta-learning is related to performing adaptations on multiple tasks during training and performing the meta-update by taking the sum or mean of the losses across these tasks as described in algorithm 4.

Gradient Clipping

Clipping gradients by their norm is a good way of avoiding cases where performing a parameter update with an irregularly large gradient would cause the whole optimisation process to diverge. It works by scaling the gradients to ensure their L_2 norm does not exceed a certain threshold provided as a hyperparameter during training. In our experience, we have found gradient clipping to be particularly useful when working with pre-trained networks such as MPNet and CNet. This could be caused by the fact that these networks were pre-trained on slightly different objectives or by the general instability of MAML in some scenarios.

3.6 Summary

In this chapter, we introduce all the building blocks that make up our final solution:

- In section 3.1 we present an unsupervised loss function, which uses the weighted sum of two terms to provide feedback on path correctness. While this section does not include quantifiable proof of its effectiveness, fig. 3.2, fig. 3.3 and fig. 3.4 give a good visual intuition to what these two terms represent and are backed up by quantifiable results later in section 4.2.
- Section 3.2 places the unsupervised loss into a MAML meta-learning scenario by defining every planning environment as a separate task and each path planning problem as a training example within the task. The objective here is to refine the pre-trained PNet weights θ in order to maximise PNet's positive response to the unsupervised loss $\mathcal{L}_{adapt}$. The optimisation process is performed using $\mathcal{L}_{adapt}$ for the adaptation step updates and the supervised L_2 loss $\mathcal{L}_{meta}$ for the meta-update, meaning that the L_2 loss is the meta-objective i.e. what we aim to maximise after adaptations.
- Section 3.3 and section 3.4 describes some noteworthy alternatives/modifications to our initial meta-learning formulation. Specifically, section 3.3 present key modifiations to MAML, which could significantly improve training performance and section 3.4 provides alternatives to $\mathcal{L}_{adapt}$, that could boost our model's ability to optimise for the meta-objective.

- Lastly, section 3.5 gives insight on some of the challenges encountered during the optimisation process, which were unique or rather tricky compared to other deep learning-based path planning solutions.

Chapter 4

Experiments & Evaluation

Possibly due to their varying applicability in different scenarios, there is no unified benchmark for evaluating path planning algorithms. Additionally, at the time of writing, no attempts to apply meta-learning to this problem have been published, therefore the evaluation of our model is not straightforward. Given that our approach aims to improve on MPNet's planning module (PNet), which is trained to minimise the L_2 loss from the provided expert demonstrations, but this may not be an accurate representation of finding the shortest non-colliding path (demonstrated in fig. 3.7), during evaluation we emphasize the following metrics: **path length**, **collision rate**, L_2 **loss**.

Following the general structure of chapter 3, we seek to answer the questions below regarding our approach:

- Does the unsupervised loss function described in section 3.1 have the capacity to refine PNet and improve its performance?
- Can MAML or other similar methods (FOMAML, Reptile) be applied to maximise the potential of the unsupervised loss, training PNet to be sensitive to rapid refinements via a few gradients steps?
- Does embedding additional information from the expert demonstration into the unsupervised loss computation (section 3.4) improve performance?

4.1 Evaluation Preliminaries

4.1.1 Neural network pre-training

ENet

ENet is trained for 400 epochs in an unsupervised fashion as a contractive autoencoder (described in fig. 2.13a). It contains 4 fully-connected hidden layers for encoding the obstacle point-clouds, reducing their dimensionality by a factor of 100 (i.e. 2D environments are encoded from 2800 to 28 and 3D environments from 6000 to 60). To decode the signal during training, 4 additional hidden layers with ReLU activations are put in place with layer sizes identical to the first 4, but in reverse order. Note that ENet is not being refined during our experiments, so we only train it once for both the 2D and 3D setting and store the learned encodings as part of our experiment dataset.

PNet

PNet is pre-trained for 500 epochs on all 100 training environments with 4000 paths each (process described in fig. 2.13b). Its architecture is comprised of 10 hidden layers with ReLU activations, where the input and output dimensions vary depending on the type of environment (2D or 3D) it is operating in. Throughout our experiments, we initialise PNet with the weights obtained via the training procedure above and maintain the same architecture. It also serves as a baseline during evaluation.

4.1.2 Data Characteristics

2D Environments

We train and evaluate our models on MPNet's **Simple2D** dataset included in the open-source implementation [18]. This dataset contains 7 square-shaped obstacles of equal size, each covering 6.25% of the environment's space and obstacles may overlap, reducing the overall coverage. As the name suggests, the path planning problems in this 2D setting are relatively simple, so it is mainly used to validate the soundness of our approach without expecting large improvements compared to PNet, which already does quite well on these toy problems. In total, 30000 different point-clouds with 1400 points each are provided to train ENet. To train/evaluate PNet, our dataset includes 100 environments with 4600 (4000 for training, 100 for validation, 500 for testing) and 10 environments with 600 (unseen during training; 100 for validation, 500 for testing) path expert demonstrations.

3D Environments

To verify our approach in more complex environments, we make use of the **Complex3D** dataset also provided in [18]. This contains 10 rectangular objects of various sizes represented as point-clouds of 2000 points per environment. The amount of data and respective splits are the same as in the case of Simple2D. The main difference in terms of the implementation is a different input size to the neural networks, which needs to reflect the increase in point dimensions, otherwise our method is not constrained by the dimensionality of the problem.

4.1.3 Obtaining evaluation metrics

Path length

Since all of our experiments are performed in 2D and 3D euclidean spaces, the **path length** can be computed as the euclidean distance between each intermediate points $\hat{p}_1 ... \hat{p}_n \in \hat{P}$, where $\hat{P}$ is the set of points produced by the planner network forming a path:

$$l(\hat{P}) = \sum_{i=0}^{n-1} ||\hat{p}_i - \hat{p}_{i+1}||_2 \tag{4.1}$$

L_2 loss

For the reasons outlined in section 3.5.1, to compute the L_2 **loss**, we require that the points forming the path are distributed equidistantly. This needs to be done both for the predicted paths $\hat{P}$ and the expert demonstrations P as these are obtained via RRT*, thus not evenly distributed. We denote the sets of equidistantly distributed path points $p_{eq,0}, ..., p_{eq,n} \in P_{eq}$ and $\hat{p}_{eq,0}, ... \hat{p}_{eq,n} \in \hat{P}_{eq}$ respectively. P_{eq} and $\hat{P}_{eq}$ are obtained using a Tensorflow library function [60], allowing us to approximate each path as a polyharmonic spline, which can then be evaluated at evenly spaced query points. Following these operations, the L_2 **loss** is computed as follows:

$$L_2(\hat{P}_{eq}, P_{eq}) = \sum_{i=0}^{n} ||\hat{p}_{eq,i} - p_{eq,i}||_2 \tag{4.2}$$

Collision-rate

When detecting collisions, we must consider two important factors: the datasets used throughout all our experiments provide environments in the form of point-clouds with no volume; the evaluated paths are represented as a discrete set of points. Keeping these two limitations in mind, we detect collisions by resampling each path P to a sufficiently large number of equidistant points P_{eq} and evaluating for each $p_{eq,0}, ..., p_{eq,n} \in P_{eq}$ if they are within range ϵ of any obstacle point $o_0, ..., o_m \in O$ as seen in eq. (4.3). Conceptually, one can think of it as drawing spheres with a certain radius ϵ around each obstacle point and checking whether the path points lie inside the sphere's volume.

$$\text{colliding}(p) = \begin{cases} 1 & \text{if } \min_{o_m \in O} ||p - o_m||_2 <= \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

By examining the obstacle point-clouds in the 2D and 3D dataset, the average minimum distance between two obstacle points is ≈ 0.04 and ≈ 0.6 respectively, so we set $\epsilon_{2D} = 0.04/2 = 0.02$ and $\epsilon_{3D} = 0.6/2 = 0.3$ to approximate the volumes represented by the point-clouds. We set the number of path points for P_{eq} to be $n = 1000$, providing sufficient amount of detail to accurately represent **collision percentage** on the path level i.e percentage of points $p_{eq,0}, ..., p_{eq,n} \in P_{eq}$ colliding with the environment as seen in eq. (4.4). We define **collision rate** in eq. (4.6) as the percentage of evaluated paths $P_{eq,0}, ..., P_{eq,k}$ exceeding a certain threshold of collision percentage γ. By observing collision-rates at different γ we can evaluate approaches, where we suspect that a large number of paths only slightly collides with the environment.

$$\text{collisionPercentage}(P_{eq}) = \frac{1}{n} \sum_{p_n \in P_{eq}} \text{colliding}(p_n) \tag{4.4}$$

$$\text{collidingPath}(P_{eq}) = \begin{cases} 1 & \text{if collisionPercentage}(P_{eq}) >= \gamma \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

$$\text{collisionRate}(P_{eq,0}, ..., P_{eq,k}) = \frac{1}{k} \sum_{i=0}^{k} \text{collidingPath}(P_{eq,i}) \tag{4.6}$$

4.2 Refining PNet with unsupervised loss

In line with the timeline of how we developed our approach, before jumping into a full meta-learning solution, we first validate that the unsupervised loss function introduced in section 3.1 can be effectively used to refine PNet. In essence, in these experiments we evaluate the adaptation period of the MAML pipeline in isolation (corresponding to lines 6-9 in algorithm 4), by computing $\mathcal{L}_{adapt}$ and performing gradient updates to PNet for multiple iterations on each environment individually. To thoroughly examine the capabilities of $\mathcal{L}_{adapt}$, we record the metrics from section 4.1.3 after a different number of adaptation steps A and for various weights ρ. When it comes to ρ, we are curious to find out whether having one of the two terms $l(x)$, $c(x)$ weighted higher significantly impacts the performance. For values of A, we are interested to see how much of an effect does 1 gradient update have and if further updates can provide a significant improvement.

4.2.1 Experiment setup

Each experiment is initialised with pre-trained PNet weights, which also serves as the baseline when comparing the obtained metrics. Likewise as in [10], environments seen and unseen during the PNet pre-training are evaluated separately, with 500 paths per each 100 seen and 10 unseen environments. Note that none of the paths used for evaluation have been used during the PNet pre-training (i.e. not even the paths in the 100 seen environments have been seen during pre-training). Each adaptation parameter update is performed with a learning rate $\alpha = 0.0001$ and the gradients are clipped by norm at clipnorm = 25.

4.2.2 Results

By looking at the **path length** comparisons in table 4.1 and table 4.2, as expected, it is clear that a larger ρ causes the paths to be longer due to the higher priority placed on $c(x)$. Overall, we observe that taking multiple adaptation steps is certainly beneficial, but the rate of improvement is reduced significantly after 5 steps. These results signify that with appropriate weighting, the $l(x)$ term has the ability to provide meaningful feedback to the neural network to promote the prediction of shorter paths.

Path Length Simple2D	Seen Environments			Unseen Environments		
	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$
PNet w/ 1 adapt. step	4.731	4.741	4.754	4.840	4.847	4.865
PNet w/ 5 adapt. steps	**4.698**	4.744	4.808	4.752	4.783	4.871
PNet w/ 10 adapt. steps	4.700	4.764	4.880	**4.735**	4.778	4.893
PNet		4.747			4.866	
RRT*		4.671			4.654	

Table 4.1: Evaluating path length on the **Simple2D** dataset for various weighting constants $\rho \in \{0.5, 1.0, 2.0\}$ and number of adaptation steps $A \in \{1, 5, 10\}$.

Path Length Complex3D	Seen Environments			Unseen Environments		
	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$
PNet w/ 1 adapt. step	5.247	5.248	5.250	5.374	5.376	5.468
PNet w/ 5 adapt. steps	5.227	5.229	5.236	5.329	5.332	5.338
PNet w/ 10 adapt. steps	**5.222**	5.225	5.239	**5.298**	5.300	5.312
PNet		5.255			5.387	
RRT*		5.184			5.197	

Table 4.2: Evaluating path length on the **Complex3D** dataset for various weighting constants $\rho \in \{0.5, 1.0, 2.0\}$ and number of adaptation steps $A \in \{1, 5, 10\}$.

To validate $\mathcal{L}_{adapt}$ from the collision avoidance perspective, we capture the **collision rate** across various collision percentages γ for both datasets in section 4.4.2 and combine these results with the L_2 **loss** in table 4.3 and table 4.4. In an ideal scenario, both of these metrics should indicate the same outcomes, but as we have discussed previously, this is not entirely the case. While in most cases, refinements via $\mathcal{L}_{adapt}$ lead to a better L_2 loss and collision rate as well, various values of ρ seem to affect the two metrics differently. According to our measurements in table 4.3 and table 4.4, the L_2 loss benefits from lower values of ρ, whereas the collision rate measured after 5 adaptation steps as depicted in section 4.4.2 is significantly improved when increasing ρ. This kind of discrepancy between our true objective (i.e. minimise collision rate) and the meta-objective (i.e. minimise L_2 loss) becomes even more significant during meta-learning experiments. Lastly, we keep an eye on the fact that $\mathcal{L}_{adapt}$ appears more effective on seen environments, most likely because these have more "almost collision-free paths" (i.e. colliding paths with low collision percentage), which can be corrected by smaller adjustments as opposed to unseen environments, where PNet performs considerably worse.

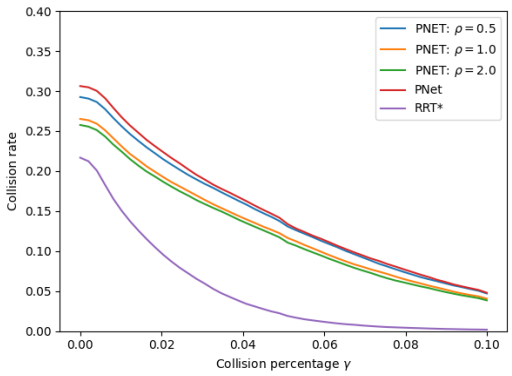
L_2 **loss Simple2D**	Seen Environments			Unseen Environments		
	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$
PNet w/ 1 adapt. step	1.312	1.351	1.431	3.210	3.253	3.412
PNet w/ 5 adapt. steps	**1.202**	1.374	1.753	**2.596**	2.776	3.489
PNet w/ 10 adapt. steps	1.237	1.510	2.335	2.686	2.805	3.749
PNet		1.367			3.427	

Table 4.3: Evaluating the L_2 loss on the **Simple2D** dataset for various weighting constants $\rho \in \{0.5, 1.0, 2.0\}$ and number of adaptation steps $A \in \{1, 5, 10\}$.

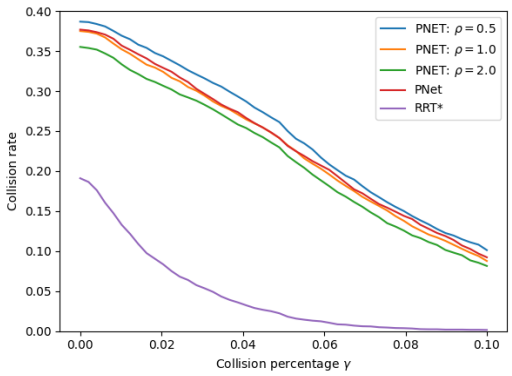
L_2 **loss** **Complex3D**	Seen Environments			Unseen Environments		
	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$
PNet w/ 1 adapt. step	1.177	1.177	1.184	2.204	2.206	2.216
PNet w/ 5 adapt. steps	1.103	1.104	1.120	1.911	1.914	1.954
PNet w/ 10 adapt. steps	**1.086**	**1.086**	1.118	**1.753**	1.760	1.817
PNet		1.250			2.324	

Table 4.4: Evaluating the L_2 loss on the **Complex3D** dataset for various weighting constants $\rho \in \{0.5, 1.0, 2.0\}$ and number of adaptation steps $A \in \{1, 5, 10\}$.

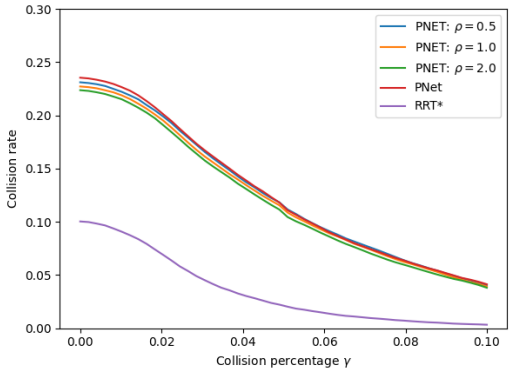
Figure 4.1: The 4 plots below depict the collision rate obtained after **5 adaptation steps** for various thresholds γ. At each γ, the path is labeled collision-free if its collision percentage is below γ, allowing us to consider "almost collision-free" paths as well with up to 10% collision percentage. $\mathcal{L}_{adapt}$ consistently improves the collision rate for the seen environments for all ρ, whereas unseen environments benefit from higher ρ emphasizing the collision term $c(x)$.



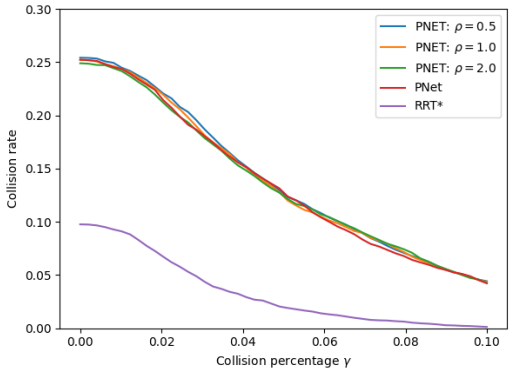
(a) Simple2D: Seen during pre-training



(b) Simple2D: Unseen during pre-trainin



(c) Complex3D: Seen during pre-training



(d) Complex3D: Unseen during pre-training

4.3 Meta-Learning to improve PNet

After showing that the unsupervised $\mathcal{L}_{adapt}$ provides meaningful feedback for adapting PNet to a given environment, in this section we evaluate whether the three meta-learning approaches described in section 3.2 and section 3.3 can be successfully used to increase PNet's sensitivity to $\mathcal{L}_{adapt}$. As MAML, FOMAML and Reptile differ in some key aspects, first we train three separate models with the same parameters using each of these methods to see which one is most suitable for our application. The most promising method is then chosen to meta-learn models with various weighting factors ρ for comparisons with the baseline PNet, Adapted PNet (i.e. PNet w/ adapt.

steps) obtained in section 4.2 and also RRT* where relevant, to provide a more comprehensive overview.

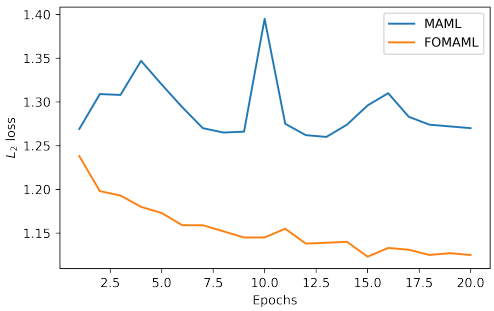
4.3.1 Experiment setup

In all the experiments below, meta-learning is performed only on the 100 environments seen during PNet pre-training, with 800 paths per environment, as opposed to the 4000 training samples available due to performance constraints explained in section 3.5.1. Comparisons between the three models regarding their learning progress are made by recording a validation loss after each epoch. During validation, we perform 5 adaptation steps for every environment (including both 100 seen and 10 unseen) using 100 paths each and report the L_2 loss. This gives us a good understanding, whether the performance on the meta-ojective is improving over time. To evaluate the best performing method across various ρ, the same 500 paths per environment are used as in section 4.2. The learning rates during meta-learning and evaluation are always set to $\alpha = 0.0001$, $\beta = 0.0001$, the gradients are clipped by norm with clipnom $= 25.0$, the number of adaptation steps is $A = 5$.

4.3.2 Results

When comparing which method is best suited for our problem, we observe vastly different results. In the past, Reptile has shown promise in meta-learning scenarios where the equivalent MAML formulation would have $\mathcal{L}_{adapt}$ and $\mathcal{L}_{meta}$ as the same function, therefore the use of $\mathcal{L}_{meta}$ can be mitigated by essentially pulling the network weights in the direction of the adapted parameters θ' as described in section 2.4.2. Since in our case $\mathcal{L}_{adapt}$ is a less informed unsupervised loss function and gives weaker feedback on the overall path planning problem than the expert demonstrations contained within $\mathcal{L}_{meta}$, the Reptile meta-update guided by only $\mathcal{L}_{adapt}$ gradually deteriorates the overall performance as depicted in fig. 4.8b, therefore we will not consider Reptile in follow-up experiments. In terms of their approach, MAML and FOMAML are much more closely related, as they mostly share the same procedures, except for the meta-update. In the FOMAML meta-gradient derivation in section 2.4.2 we notice that MAML includes several second-order gradient terms influenced by $\mathcal{L}_{adapt}$, whereas FOMAML omits these. Looking at fig. 4.8a, we speculate that the unstable learning progress of MAML can be attributed to these second-order terms, making FOMAML the clear winner of this comparison.

(a) MAML vs FOMAML validation loss



(b) Reptile validation loss

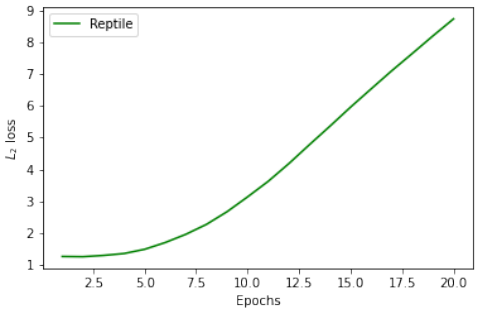


Figure 4.2: Validation losses measured during the training of MAML, FOMAML and Reptile. FOMAML is clearly the most suitable method for our use-case. MAML suffers from great instability, most likely due to noise in the second derivative terms coming from the $\mathcal{L}_{meta}$. Reptile is fundemantally unsuitable because of the absence of $\mathcal{L}_{meta}$ during training and the updates promoted by $\mathcal{L}_{adapt}$ pull it away from the objective over-time.

After establishing FOMAML as the best performing method in the first half of this experiment, we train separate models with various weights $\rho \in \{0.5, 1.0, 2.0\}$ on both the Simple2D and Complex3D datasets. Here, we are interested in not only whether the FOMAML approach outperforms pre-trained PNet, but we also want to see how it compares to Adapted PNet from section 4.2 i.e. if the FOMAML model became more sensitive to $\mathcal{L}_{adapt}$. From table 4.5, our meta-learned method consistently produces a better path length than Adapted PNet, indicating that the responsiveness

to the $l(x)$ term in $\mathcal{L}_{adapt}$ has indeed been increased. More importantly, the fact that the L_2 loss has also been substantially improved across the board is a sign of our meta-learning approach being successful, since this metric is the meta-objective of the optimisation process. Unfortunately, this isn't fully reflected in the collision rate. By examining fig. 4.3, we can see that in the seen environments, our model produces a significant number of paths with collision percentages in the 6-10% range, but overall the amount of collisions (0% collision percentage) is higher than in the case of Adapted PNet. We suspect that because this problem is rather simple, RRT* manages to explore the scene very well using its sampling-based approach, outputting ground truth paths that are very close to obstacle edges, leading to inconsistency between the L_2 loss and collision rate. However, the marginal improvement of collision rate in unseen environments, where PNet performs considerably worse leaving more room for refinement, does indicate our model's ability to reduce collisions.

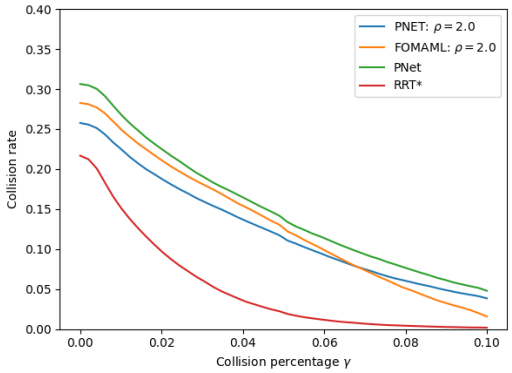
Path Length **Simple2D**	Seen Environments			Unseen Environments		
	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$
PNet w/ 5 adapt. steps	4.698	4.744	4.808	4.752	4.783	4.871
FOMAML w/ 5 adapt. steps	**4.675**	**4.678**	**4.684**	**4.707**	**4.718**	**4.751**
PNet		4.746			4.866	
RRT*		4.671			4.654	

Table 4.5: The meta-learned FOMAML model clearly outperforms PNet, Adapted PNet and gets close to the ground truth RRT* path length, especially on the seen environments in **Simple2D**.

L_2 **loss** **Simple2D**	Seen Environments			Unseen Environments		
	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$
PNet w/ 5 adapt. steps	1.202	1.374	1.753	2.596	2.776	3.489
FOMAML w/ 5 adapt. steps	**1.071**	**1.079**	**1.107**	**2.471**	**2.436**	**2.578**
PNet		1.367			3.427	

Table 4.6: The L_2 loss is reduced across the board in **Simple2D**, with most significant relative improvements achieved for larger ρ.

(a) Simple2D: Seen during pre-training



(b) Simple2D: Unseen during pre-training

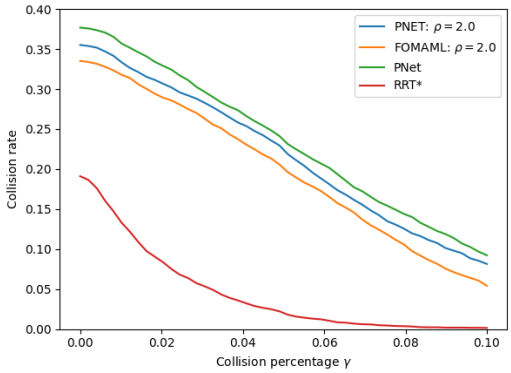


Figure 4.3: Collision rate comparison between RRT*, PNet, Adapted PNet with $\rho = 2.0$ (best overall collision rate from section 4.4.2) and FOMAML with $\rho = 2.0$ on the **Simple2D** dataset. FOMAML struggles to improve the collision rate in seen environments despite a better L_2 loss, but boosts the performance slightly in unseen environments.

Looking at the results obtained on the Complex3D dataset, the path length and L_2 metrics

in table 4.7, table 4.8 are consistent with the 2D case, demonstrating that FOMAML indeed improves on Adapted PNet. The significant reduction in the L_2 loss on unseen environments is particularly promising, as it shows that even despite only using 20% of the available dataset for meta-learning, our model doesn't overfit and considerably outperforms Adapted PNet. The collision rates plotted in fig. 4.5 support our hypothesis from the 2D dataset evaluation regarding the suboptimal ground truth data, since here we see a consistent decrease in collisions for both seen and unseen environments, and the slopes of the curves plotted across different collision percentages are also much more similar. Lastly, we note that while still maintaining the overall tendencies, the differences in results achieved for various weighting factors ρ have been noticeably reduced, suggesting that we are getting close to exhausting the potential of the current unsupervised loss formulation.

Path Length **Complex3D**	Seen Environments			Unseen Environments		
	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$
PNet w/ 5 adapt. steps	5.227	5.229	5.236	5.330	5.332	5.338
FOMAML w/ 5 adapt. steps	**5.193**	**5.196**	**5.206**	**5.257**	**5.258**	**5.268**
PNet		5.255			5.387	
RRT*		5.184			5.197	

Table 4.7: The path length is also reduced in **Complex3D**, getting quite close to the ground truth RRT* path length in seen environments.

L_2 **loss** **Complex3D**	Seen Environments			Unseen Environments		
	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$
PNet w/ 5 adapt. steps	1.103	1.104	1.120	1.911	1.914	1.954
FOMAML w/ 5 adapt. steps	**0.871**	**0.878**	**0.911**	**1.460**	**1.446**	**1.480**
PNet		1.250			2.324	

Table 4.8: The L_2 loss is significantly improved in **Complex3D**, especially in the unseen environments. Interestingly, the difference between various values of ρ is reduced after meta-learning.

(a) Simple2D: Seen during pre-training

(b) Simple2D: Unseen during pre-trainin

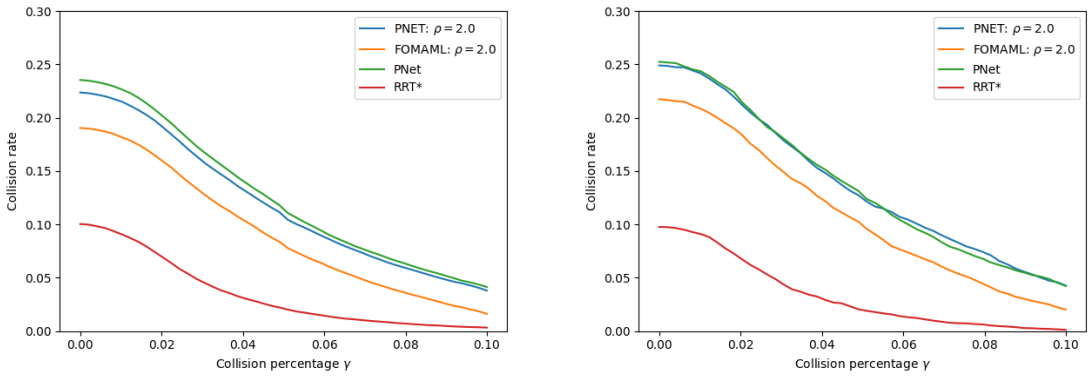


Figure 4.4: Collision rate comparison between RRT*, PNet, Adapted PNet with $\rho = 2.0$ and FOMAML with $\rho = 2.0$ on the **Complex3D** dataset. Our model achieves noticeable reduction in collision rate in all environments.

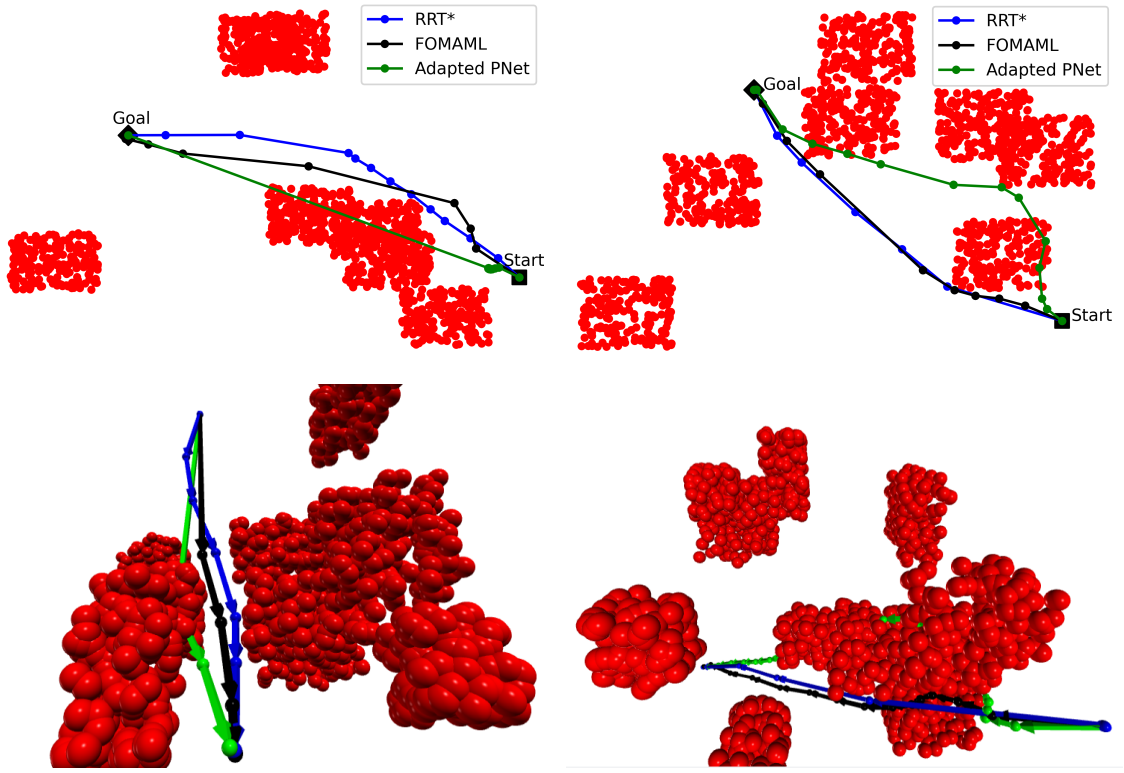


Figure 4.5: In both 2D and 3D environments, the paths generated by FOMAML (black) after adaptation via $\mathcal{L}_{adapt}$ maintain a lower average L_2 loss to the ground truth paths (blue) than the paths generated by Adapted PNet (green), resulting in better paths overall.

4.3.3 Limiting factors

Besides the **inflexible unsupervised** $\mathcal{L}_{adapt}$ function to which we dedicate section 3.4 and section 4.4 of this report, there are two other important limitations of our solution, which lead to the most common failure cases depicted in fig. 4.6, fig. 4.7.

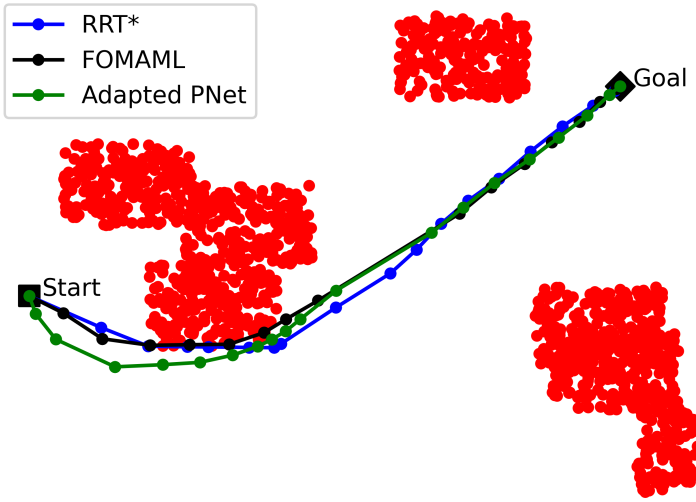


Figure 4.6: One of the most common failure cases, especially on the Simple2D dataset is when despite our model doing a better job of optimising for the training meta-objective i.e. L_2 loss, the predicted paths still collide with the environment. On the other hand, Adapted Pnet with the same number of adaptation steps (5 steps) manages to avoid collisions by keeping further away from the ground truth path.

One type of common errors is caused by the issue encountered with the Simple2D dataset,

where the **ground truth paths** generated by the RRT* algorithm are simply **too close to the obstacle edges**, and our model fails to produce collision-free paths even though it is doing a good job of minimising the L_2 loss. The second problem is related to the **instability of the training setup**, forcing us to set fairly low learning rates and frequently apply gradient clipping, resulting in a **slow learning process**. When looking at the paths generated by our network, there are many instances, where we can see by comparing to Adapted PNet, that the response to $\mathcal{L}_{adapt}$ has definitely increased, however not to the extent that it would fully correct the path.

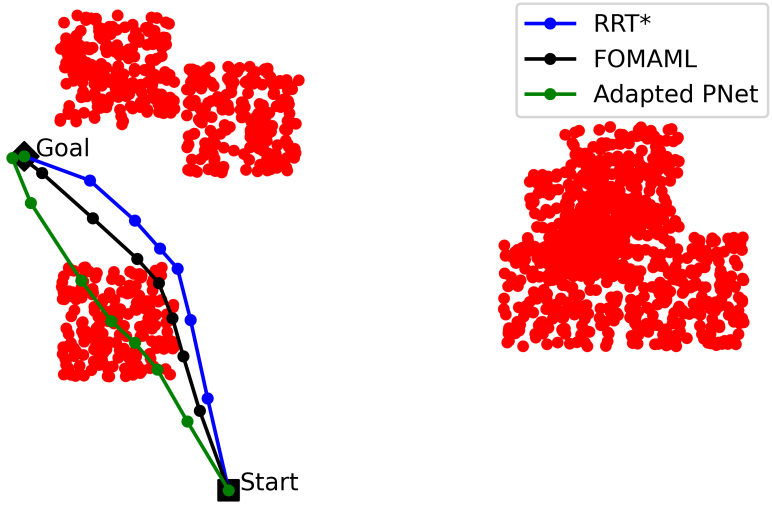


Figure 4.7: Due to the small learning rates enforced by the unstable optimisation process, a number of colliding paths a very close to being fully corrected. In this image we can see how much more sensitive FOMAML is to adaptations as opposed to Adapted PNet, yet the path is still not entirely fixed.

4.4 Alternative formulations

In these experiments, we implement the two modifications to the initial unsupervised $\mathcal{L}_{adapt}$ described in section 3.4, observe what the networks taking part in the new $\mathcal{L}_{adapt}$ computation learn and compare their performance to our initial approach.

4.4.1 Experiment setup

For both RNet and CNet, we use the same neural network architecture with 5 hidden layers and ReLU activations. The inputs to both networks are composed of the scene encodings generated by ENet and the positional encodings of the start and goal points, using 5 sine and 5 cosine encodings for each input signal. Prior to meta-learning, the only difference between RNet and CNet is that CNet is pre-trained to predict $c(x)$ on 5000 environments from the Simple2D dataset as it will fully replace the computation of the collision avoidance term in the modified $\mathcal{L}_{adapt}$. The optimisation process is performed for 20 epochs, same as the FOMAML experiments with fixed ρ. Considering that the methods evaluated in this section are not the main contribution of this master's thesis and due to time constraints, these solutions are only attempted as proof-of-concept, therefore only trained on the Simple2D dataset.

4.4.2 Results

From the results displayed in table 4.9, table 4.10 and fig. 4.8, we can see that the CNet architecture overall performs slightly better than RNet and the FOMAML experiments with fixed ρ, but in general, the differences are not very significant. While this does not necessarily mean that the presented modifications cannot provide some benefits, it is difficult to draw definitive conclusions from such a limited amount of experiments. What particularly stands out is that the CNet architecture, which fully replaces the collision term calculation during meta-learning, is still able to outperform Adapted PNet when it comes to the L_2 loss. This indicates that CNet might be

learning a collision term representation that helps further minimise the L_2 loss, possibly leading to a better end result than FOMAML with a fixed weight if trained for more epochs, allowing CNet to learn more.

Path Length Simple2D	Seen Environments			Unseen Environments		
	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$
PNet w/ 5 adapt. steps	4.698	4.744	4.808	4.752	4.783	4.871
FOMAML w/ 5 adapt. steps	4.675	4.678	4.684	**4.707**	4.718	4.751
FOMAML w/ RNET		4.681			4.754	
FOMAML w/ CNET		**4.673**			4.729	

Table 4.9: Path length comparison of RNet, CNet and FOMAML with various fixed weights ρ. No considerable improvements are observed, but neither RNet nor CNet fall behind.

L_2 **loss Simple2D**	Seen Environments			Unseen Environments		
	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$	$\rho = 0.5$	$\rho = 1.0$	$\rho = 2.0$
PNet w/ 5 adapt. steps	1.202	1.374	1.753	2.596	2.776	3.489
FOMAML w/ 5 adapt. steps	1.071	1.079	1.107	2.471	2.436	2.578
FOMAML w/ RNET		1.094			2.53	
FOMAML w/ CNET		**1.063**			**2.4**	

Table 4.10: CNet is showing signs of being superior to RNet with improved L_2 loss in seen and unseen environments, despite fully relying on CNet predictions for the collision term $c(x)$.

(a) Simple2D: Seen during pre-training

(b) Simple2D: Unseen during pre-training

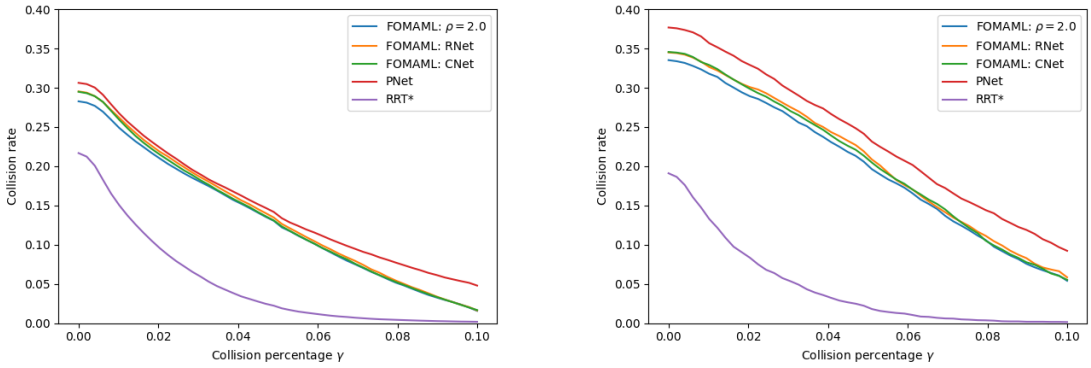


Figure 4.8: In both seen and unseen environments, RNet and CNet show similar collision-rate tendencies to FOMAML with fixed weights with very marginal differences.

The slope of the learning curves in fig. 4.9 supports our hypothesis that the CNet architecture is worth pursuing further as it looks quite likely that with a longer training time, it could surpass FOMAML with fixed ρ. Another interesting finding depicted in fig. 4.10 is that based on the updates from $\mathcal{L}_{meta}$, CNet seems to learn that the ideal paths tend to be focused in close proximity of the obstacles, which further indicates that this representation could provide extra benefits. Unfortunately, due to the time constraints of this project and already long training times (2 days for 20 epochs), additional explorations of this method are out of the scope of this work.

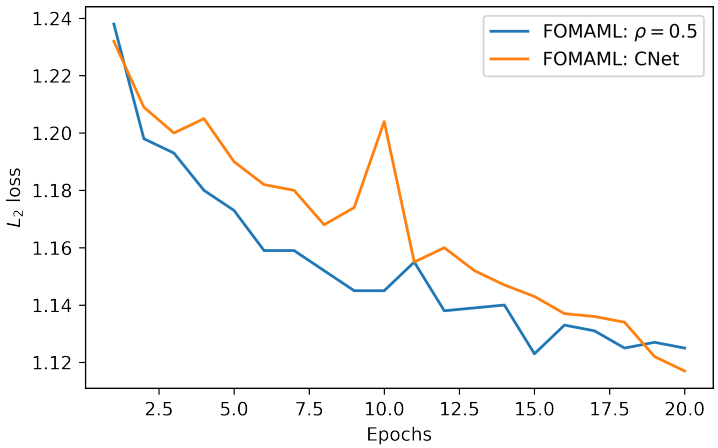


Figure 4.9: The slope of the validation loss curve of the CNet architecture suggests that it might lead to a noticeable improvement compared to FOMAML with fixed ρ given more training time.

(a) Pre-trained CNet heatmap before meta-learning

(b) CNet heatmap after meta-learning

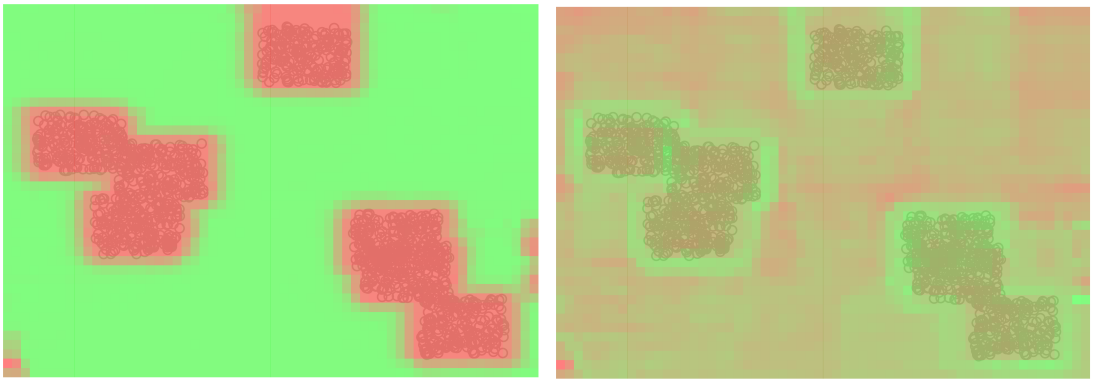


Figure 4.10: Heatmaps depicting the values predicted by CNet before and after meta-learning. Thanks to the information from ground truth paths, CNet learns to prefer paths that go narrowly around the obstacles.

4.5 Summary

At the beginning of this chapter we outlined three main questions that we wished to answer during the evaluation of our approach. By performing relevant experiments in 2D/3D environments and collecting meaningful metrics defined in section 4.1, we arrived at the following conclusions:

- In section 4.2 we showed that by refining PNet using a set of paths, the unsupervised loss $\mathcal{L}_{adapt}$ does provide sufficient feedback to improve PNet's performance on the selected paths. While the improvements are not very significant, they are consistently maintained over both tested datasets, indicating that it can be relied upon during meta-learning.

- In section 4.3, first we evaluate which method out of MAML, First-order MAML (FOMAML), Reptile is most suitable for our approach. After FOMAML emerges as the clear winner of the comparison, we train a number of models with varying parameters and show that our model responds significantly better to refinements by $\mathcal{L}_{adapt}$ as the non-meta-learned PNet model. Lastly, we list what we consider to be the three most important limiting factors of our method and provide examples of common failure cases associated with them.

- While the modified versions of $\mathcal{L}_{adapt}$ evaluated in section 4.4 did not lead to an immediate performance boost, we do think that the idea of embedding information from the ground truth into the unsupervised loss warrants further exploration based on our findings.

Chapter 5

Conclusion

This work is motivated by prior research in path planning neural networks (PNet) [10], which provides a highly applicable solution for robot path planning utilising point-cloud information about the environment, possibly obtained via lidar measurements in practice. Our contribution enhances the predictive capabilities of these networks in three steps:

1. We address one of PNet's [10] disadvantages of being trained in a supervised fashion to predict individual path segments instead of outputting entire paths by showing that its performance can be enhanced by further training with an unsupervised loss function providing feedback on the path-level.
2. As the main focal point of this master's thesis, we apply three common meta-learning approaches and demonstrate their ability to maximise the performance improvements achieved by the unsupervised loss function.
3. Identifying the strict mathematical formulation of the unsupervised loss as a possible bottleneck, additional methods are explored to fully utilise the capacity of our meta-learning pipeline.

To validate the proposed approach, we define three key metrics reported at each of the stages listed above, tracking whether the obtained results match our expectations. We also provide qualitative comparisons between the attempted meta-learning techniques and state hypotheses on why one was clearly the most suitable for this task. Lastly, a discussion on the main limiting factors of our solution is included that could be addressed in future work.

Overall, we consider the outcomes of this work to be generally positive. We have succeeded in our main goal of demonstrating that meta-learning can be successfully applied in this domain, showing consistent improvement in most scenarios. Although the performance increase is marginal at times and still quite far from current state-of-the-art sampling-based path planning algorithms characterised by much longer inference times, we believe to have provided sufficient evidence to support further pursuit of this optimisation technique.

5.1 Future Work

Given that both scientific works serving as our inspiration were presented fairly recently, in 2017 [13] and 2019 [10] respectively, there are still plenty of unexplored pathways waiting to be addressed by the scientific community. Following our initial results, we attempted some alternatives but couldn't fully follow-through on them due to the project's limited timeline.

Different unsupervised loss formulation

Given the main focus of this project, the unsupervised loss function was only assessed in terms of its potential to be used in meta-learning, therefore other alternatives can be considered to maximise the refinement effectiveness. As our training pipeline does not place significant constraints on the optimisation process as long as a similar planning network to PNet is used, learning-based modifications such as the ones described in section 3.4 or other techniques could be utilised with minor adjustments.

Obtain better training data

As described in section 4.3, our model is negatively impacted when trained with expert demonstrations that are very close to the obstacles because it's minimising the L_2 loss which does not directly guarantee less collisions. By generating new data in a way that the ground truth paths would maintain a certain minimum distance from the obstacles, we could increase the correspondence between the objective of minimising the L_2 loss, reducing the overall collision rate as well.

Stabilise meta-learning

In some sense, our approach fits the definition of a transfer learning optimisation process [65] [66], as PNet is pre-trained on the MPNet objective and is being re-trained to a slightly different meta-objective. Perhaps due to the difference in these objectives, we limit the learning rates in our implementation to prevent unstable learning, which could potentially be improved using Differential Adaptive Learning rates [67], meta-learned learning rates [14] or other methods.

Robot manipulator control

Considering the level of success in higher dimensional 3D spaces, we expect a similar level of improvement on even harder problems, such as a 6 degree-of-freedom robot manipulator control task. Given that the authors of MPNet have already experimented with such a task [68], this can be considered a straightforward extension to our current implementation.

Enforcing path-level constraints

Thanks to the path-level response during the optimisation process, the current solution is suitable for motion planning tasks, where the outputted path is constrained by physical limitations. For instance, if certain joints of a robot manipulator do not have a full range of motion, this can be reflected in the unsupervised loss computation.

Bibliography

[1] Apple Inc. Blurring an image. URL https://developer.apple.com/documentation/accelerate/blurring_an_image.

[2] Kamil Krzyk. Coding deep learning for beginners — linear regression (part 3): Training with gradient descent. URL https://towardsdatascience.com/coding-deep-learning-for-beginners-linear-regression-gradient-descent-fcd5e0fc077d.

[3] Haskell B Curry. The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261, 1944.

[4] wikipedia.org. Cmarkov decision process. URL https://en.wikipedia.org/wiki/Markov_decision_process.

[5] Amit Patel. Introduction to a* from amit's thoughts on pathfinding. URL http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[6] Tom Schouwenaars, Bart De Moor, Eric Feron, and Jonathan How. Mixed integer programming for multi-vehicle path planning. pages 2603–2608, 2001.

[7] A. Yershova, Leonard Jaillet, Thierry Simeon, and LaValle. Dynamic-domain rrts: Efficient exploration by controlling the sampling domain. pages 3856 – 3861, 05 2005. doi: 10.1109/ROBOT.2005.1570709.

[8] Liang Yang, Juntong Qi, Dalei Song, Jizhong Xiao, Jianda Han, and Yong Xia. Survey of robot 3d path planning algorithms. *Journal of Control Science and Engineering*, Jul 2016. URL https://www.hindawi.com/journals/jcse/2016/7426913/.

[9] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. *Advances in neural information processing systems*, 29:2154–2162, 2016.

[10] Ahmed H Qureshi, Anthony Simeonov, Mayur J Bency, and Michael C Yip. Motion planning networks. *arXiv preprint arXiv:1806.05767*, 2018.

[11] A. H. Qureshi, Y. Miao, A. Simeonov, and M. C. Yip. Motion planning networks: Bridging the gap between learning-based and classical motion planners. *IEEE Transactions on Robotics*, pages 1–19, 2020. doi: 10.1109/TRO.2020.3006716.

[12] Tom Jurgenson and Aviv Tamar. Harnessing reinforcement learning for neural motion planning. *CoRR*, abs/1906.00214, 2019. URL http://arxiv.org/abs/1906.00214.

[13] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

[14] Zhenguo Li, Fengwei Zhou, Fei Chen, and Hang Li. Meta-sgd: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835*, 2017.

[15] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[16] M. J. Bency, A. H. Qureshi, and M. C. Yip. Neural path planning: Fixed time, near-optimal path generation via oracle imitation. pages 3965–3972, 2019. doi: 10.1109/IROS40897.2019.8968089.

[17] Joaquin Vanschoren. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548*, 2018.

[18] Ahmed Qureshi. Motion planning networks open-source implementation, 2018. URL https://github.com/ahq1993/MPNet.

[19] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[20] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org.

[21] Nick Kanopoulos, Nagesh Vasanthavada, and Robert L Baker. Design of an image edge detection filter using the sobel operator. *IEEE Journal of solid-state circuits*, 23(2):358–367, 1988.

[22] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[23] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[24] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[25] Erik Bylow, Jürgen Sturm, Christian Kerl, Fredrik Kahl, and Daniel Cremers. Real-time camera tracking and 3d reconstruction using signed distance functions. In *Robotics: Science and Systems*, volume 2, page 2, 2013.

[26] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. 2019.

[27] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006. doi: 10.1017/CBO9780511546877.

[28] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/tssc.1968.300136. URL https://doi.org/10.1109/tssc.1968.300136.

[29] S. Koenig and M. Likhachev. Improved fast replanning for robot navigation in unknown terrain. 1:968–975 vol.1, 2002. doi: 10.1109/ROBOT.2002.1013481.

[30] Anthony Stentz. The focussed d* algorithm for real-time replanning. pages 1652–1659, 1995.

[31] Liying Yang, Juntong Qi, and Jianda Han. Path planning methods for mobile robots with linear programming. *Proceedings of 2012 International Conference on Modelling, Identification and Control, ICMIC 2012*, pages 641–646, 01 2012.

[32] C. S. Ma and R. H. Miller. Milp optimal path planning for real-time applications. pages 6 pp.–, 2006. doi: 10.1109/ACC.2006.1657504.

[33] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[34] Sertac Karaman and Emilio Frazzoli. Optimal kinodynamic motion planning using incremental sampling-based methods. pages 7681–7687, 2010.

[35] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *CoRR*, abs/1105.1186, 2011. URL http://arxiv.org/abs/1105.1186.

[36] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[37] Rahul Kala. Rapidly exploring random graphs: motion planning of multiple mobile robots. *Advanced Robotics*, 27(14):1113–1122, 2013.

[38] Sertac Karaman and Emilio Frazzoli. Sampling-based motion planning with deterministic mu-calculus specifications. *Proceedings of the IEEE Conference on Decision and Control*, pages 2222–2229, 12 2009. doi: 10.1109/CDC.2009.5400278.

[39] O. Adiyatov and H. A. Varol. Rapidly-exploring random tree based memory efficient motion planning. pages 354–359, 2013. doi: 10.1109/ICMA.2013.6617944.

[40] Dong Jia and Juris Vagners. Parallel evolutionary algorithms for uav path planning. page 6230, 2004.

[41] Gorkem Erinc and Stefano Carpin. A genetic algorithm for nonholonomic motion planning. pages 1843–1849, 2007.

[42] Jing Xin, Huan Zhao, Ding Liu, and Minqi Li. Application of deep reinforcement learning in mobile robot path planning. In *2017 Chinese Automation Congress (CAC)*, pages 7112–7116. IEEE, 2017.

[43] Linhai Xie, Sen Wang, Andrew Markham, and Niki Trigoni. Towards monocular vision based obstacle avoidance through deep reinforcement learning. *arXiv preprint arXiv:1706.09829*, 2017.

[44] Leonid Butyrev, Thorsten Edelhäußer, and Christopher Mutschler. Deep reinforcement learning for motion planning of mobile robots. *arXiv preprint arXiv:1912.09260*, 2019.

[45] Jingwei Zhang, Jost Tobias Springenberg, Joschka Boedecker, and Wolfram Burgard. Deep reinforcement learning with successor features for navigation across similar environments. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2371–2378. IEEE, 2017.

[46] S Aditya Gautam and Nilmani Verma. Path planning for unmanned aerial vehicle based on genetic algorithm artificial neural network in 3d. In *2014 International Conference on Data Mining and Intelligent Computing (ICDMIC)*, pages 1–5, 2014. doi: 10.1109/ICDMIC.2014.6954257.

[47] Brian Ichter, James Harrison, and Marco Pavone. Learning sampling distributions for robot motion planning. *CoRR*, abs/1709.05448, 2017. URL http://arxiv.org/abs/1709.05448.

[48] Sanjiban Choudhury, Mohak Bhardwaj, Sankalp Arora, Ashish Kapoor, Gireeja Ranade, Sebastian Scherer, and Debadeepta Dey. Data-driven planning via imitation learning. *The International Journal of Robotics Research*, 37(13-14):1632–1672, 2018.

[49] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *ICML*, pages 833–840, 2011. URL https://icml.cc/2011/papers/455_icmlpaper.pdf.

[50] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018. URL http://arxiv.org/abs/1808.03314.

[51] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[52] Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

[53] Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012.

[54] Devang K Naik and Richard J Mammone. Meta-neural networks that learn by learning. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 1, pages 437–442. IEEE, 1992.

[55] Yoshua Bengio, Samy Bengio, and Jocelyn Cloutier. *Learning a synaptic learning rule*. Citeseer, 1990.

[56] Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

[57] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. 2016.

[58] Gregory Koch. Siamese neural networks for one-shot image recognition. 2015.

[59] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *CoRR*, abs/1803.02999, 2018. URL http://arxiv.org/abs/1803.02999.

[60] Tensorflow polyharmonic interpolation. URL https://www.tensorflow.org/addons/api_docs/python/tfa/image/interpolate_spline.

[61] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European Conference on Computer Vision*, pages 405–421. Springer, 2020.

[62] Ellen D Zhong, Tristan Bepler, Joseph H Davis, and Bonnie Berger. Reconstructing continuous distributions of 3d protein structure from cryo-em images. *arXiv preprint arXiv:1909.05215*, 2019.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[64] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your maml, 2019.

[65] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global, 2010.

[66] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[67] Saeid Iranmanesh, M Amin Mahdavi, et al. A diffesrential adaptive learning rate method for back-propagation neural networks. *World Academy of Science, Engineering and Technology*, 50(1):285–288, 2009.

[68] Anthony Simeonov. Mpnet implementation on baxter robot manipulator, 2019. URL https://github.com/anthonysimeonov/baxter_mpnet_ompl_docker.