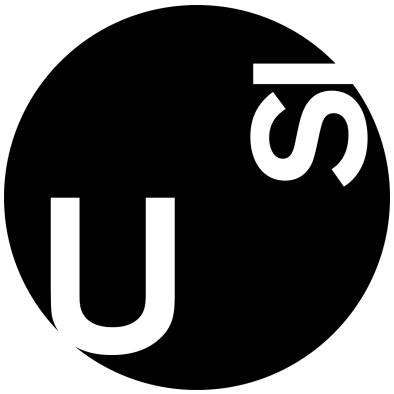


Application of Deep and Reinforcement Learning to Boundary Control Problems

Master Thesis

Zenin Easa Panthakkalakath

June 16, 2023

Advisors: Prof. Dr. Olaf Schenk

Co-Advisors: Dr. Juraj Kardoš

Faculty of Informatics, USI Lugano

Abstract

The goal of boundary control problem is, in essence, to find the optimal values for the boundaries such that the values for the enclosed domain are as close as possible to desired values while adhering to the predetermined limits for the values of the domain and boundaries. Many scientific problems, such as fluid dynamics problems involving drag reduction, temperature control with some desired flow pattern, etc., rely on optimal boundary control algorithms to maintain the values at a desired range. These forward solves are performed for multiple simulation timesteps; thus is a time-critical component of the application toolchain. Having a method that can solve the boundary control problem with fewer computational resources would improve the performance of the overall simulations.

Traditionally, the solution is obtained using nonlinear optimization methods, such as interior point, wherein the computational bottleneck is introduced by the large linear systems. Interior point methods use information from the Hessian matrix of the logarithmic barrier function and use Newton's method to iteratively arrive at a solution; wherein it requires calculating the inverse of the Hessian matrix, constraint matrix and Jacobian of constraint matrix to be found, which is often done using the conjugate gradient method. The computational complexity of this is quite high and there seems to be room for improvement here.

The objectives of this project are to explore the possibilities of using deep learning and reinforcement learning methods to solve boundary control problems, and, design experiments wherein such methods are implemented and evaluated in an attempt to see if these can rival existing solvers in terms of speed and accuracy. One such category of approaches arrived at is along the lines of policy gradient reinforcement learning method. This method utilizes the idea behind iterative optimization by treating the iterative optimization algorithm as a policy, and learning or improving this policy using policy gradient method. A method has been arrived at using this strategy that demonstrated slightly better accuracy than traditional interior point method based solvers, while the performance is calculated to be slightly worse. Another such category of approaches is along the lines of agent-based modeling with reinforcement learning, wherein the values at the boundaries and/or domain are controlled by agents. However, no method along this line has been able to produce any results that can remotely compare to the the traditional interior point method based solvers.

Overall, using deep learning and reinforcement learning to arrive at methods to solve boundary control problems have a lot of promise.

Acknowledgements

With heartfelt appreciation, I would like to use this opportunity to express my sincere gratitude to all those who provided me with unwavering support and guidance during this wonderful journey of working on this master's thesis.

Firstly, I would like to thank Prof. Olaf Schenk and Dr. Juraj Kardoš whose supervision not only helped me get through the project but also enabled me to enjoy the research process.

Additionally, I would like to express my heartfelt appreciation to Aditya Ramesh, a PhD student at USI. Conversations with him while going to play football helped in verifying and solidifying my understanding about some of the concepts in deep learning and reinforcement learning.

Furthermore, I would like to extend my special recognition to my former roommates, Arshjot Singh Khehra and Naga Venkata Sai Jitin Jami, who pursued their theses in the past two semesters, and my friends, Julien Markus Schmidt, Michal James Burgunder and Sebin Benny John, who were concurrently working on their theses as I. Our engaging conversations have significantly contributed, in varying degrees, to my successful completion of this project.

Contents

Chapter 1

Introduction

Solving optimal control problems faster has quite a lot of applications. For instance, there are various fluid dynamics related problems that involve drag reduction, temperature control, having some desired flow pattern, etc. that could be controlled by tweaking the parameters at the boundary. Using traditional methods is time-consuming; some simulations often take a day or days to complete. Improvement in performance could potentially reduce time consumption and hence allow researchers and engineers to have more opportunities to fine-tune their parameters.

The following sections provide information about the problem statement, provide information about some of the interesting relevant work that has been done in the past, and, define the objective of this study.

1.1 Elliptic control problems

Partial differential equations (PDE) are equations that describe relations between different partial derivatives of multivariate functions. The highest order of the partial derivatives that the PDE is composed of determines the order of a PDE. Consequently, a second-order PDE has the highest derivative term of order two. There are three types of bivariate second-order PDE, which are elliptic, hyperbolic and parabolic. These can be written in the general form

$$A\frac{\delta^2 f}{\delta x_1^2} + 2B\frac{\delta^2 f}{\delta x_1 \delta x_2} + C\frac{\delta^2 f}{\delta x_2^2} + D\frac{\delta f}{\delta x_1} + E\frac{\delta f}{\delta x_2} + Ff + G = 0$$

wherein A, B, C, D, E, F and G are functions of x_1 and x_2. If the value of the discriminant is less than zero, i.e.

$$B^2 - AC < 0$$

then the equation is an elliptic partial differential equation.

Optimal control theory is a branch of mathematical optimization dealing with deriving the optimal control for a dynamic system by optimizing an objective function, called cost function, which is a function of state and control variables. This is often described using a set of differential equations.

Elliptic control problems are optimal control problems involving elliptic equations describing the path of control variables.

1.2 Boundary control problems

Boundary control problems are a special category of elliptic control problems that involve controlling the values at the boundaries and involve control and state constraints. Figure 1.1 depicts the bounded domain and the boundaries. In brief, the objective of the boundary control problem is to find the optimal values for the boundaries such that the values for the bounded domain are as close as possible to certain desired values.

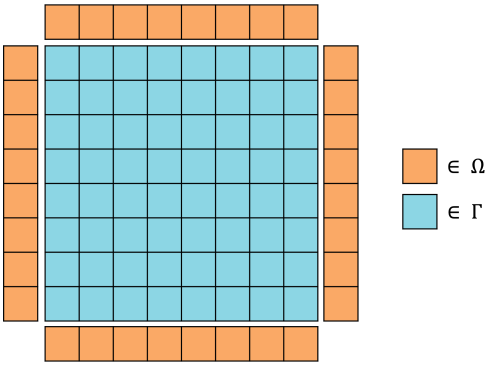
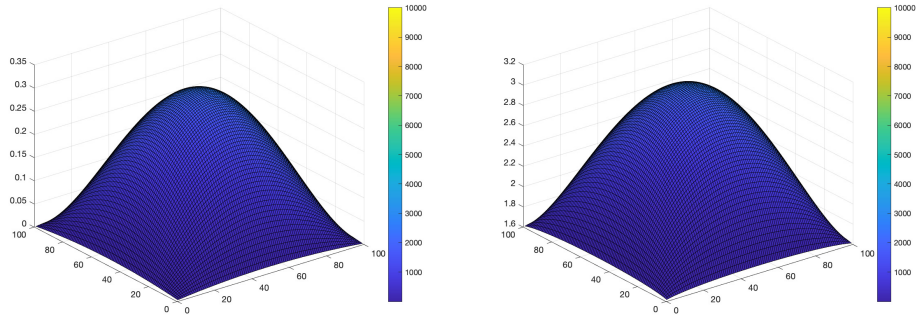


Figure 1.1: A diagram depicting the domain and the boundary

To visualize this problem better in our minds, let us view this as a temperature control problem wherein you have a large room and you can control the temperature at the boundaries using air-conditioners. You desire to have a particular temperature profile in the room; there is an area where people are sitting, there is a section where hot food is placed and there is a section where ice creams are kept. The task is to control the temperatures in the air-conditioners at the boundaries so that each area is at a temperature that is desirable; to preserve the temperature of the food, and the comfort of the people and to have ice cream that is not melting.

Indeed, this is a hypothetical example; there are better ways of doing this than just having to control temperatures at the boundary of the room; but let's just use it for the sake of visualization.

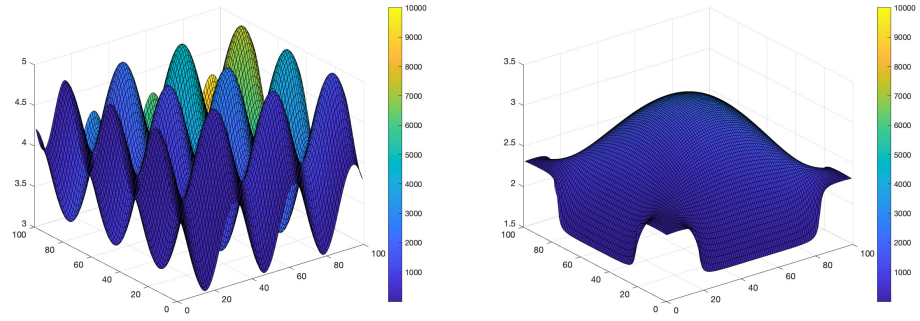
Now, if the desired profile for the domain is "not too complicated", it may be possible to achieve by controlling the boundary values. An example of this is shown in Figure 1.2. However, if the desired profile is a bit complicated,



(a) Desired **(b)** Optimal solution

Figure 1.2: When the desired domain profile is simple. Sourced from Maurer and Mittelmann[1].

then it may not be possible to achieve the same by controlling the boundary conditions. The most that can be done is to have a solution that is closest to the desired profile. An example of this is shown in Figure 1.3.



(a) Desired **(b)** Optimal solution

Figure 1.3: When the desired domain profile is complicated

Now, the question arises of how to quantify the closeness of the solution. This is done using a cost function, as shown below. Note that the objective is to minimize this cost function; in other words, the lower the value of the cost function, the closer is the solution to the desired profile.

$$F(y,u) = \frac{1}{2}\int_{\Omega}(y(x) - y_d(x))^2\,dx + \frac{\alpha}{2}\int_{\Gamma}(u(x) - u_d(x))^2\,dx$$

wherein Ω is the domain, Γ is the boundaries, y_d is the desired values for the domain, u_d is the desired values for the boundaries, y is the values for the domain, u is the values for the boundaries, α is a non-negative constant that determines how much weight must be given to the cost for the boundaries.

In the aforementioned equation, you could see the desired value for the boundaries. The term containing the same has a value α which is usually quite small (and often zero). This term is included to take into account the desired value of the boundaries. In our example to visualize in our minds, this can be viewed as the fact that keeping the air-conditioners off is more desirable as you can use less electricity.

In addition to the complications in the desired profile, there may be certain state constraints that restrict what the values within the domain and the boundaries can be. In our example to visualize in our minds, these can be viewed as the maximum and the minimum temperature that the materials that the room is made up of can withstand, and, the maximum and the minimum temperature that the air-conditioners can create, respectively.

1.3 Literature review

In this section, some of the interesting research works that were done previously that are relevant to the problem at hand are discussed briefly. These works are categorized under different subsections for improved readability.

1.3.1 Boundary control problems

The work done by Maurer and Mittelmann titled *Optimization Techniques for Solving Elliptic Control Problems with Control and State Constraints: Part 1. Boundary Control*[1] presents boundary control problems for semi-linear elliptic equations subject to control and state constraints. They approached the problem by transforming the control problem into a non-linear programming problem, and used interior point method to solve the same.

A recently published article by Lang and Schmitt titled *Exact Discrete Solutions of Boundary Control Problems for the 1D Heat Equation*[2] describes how they arrived at an analytical setting for optimal boundary control of one-dimensional heat equation, including Dirichlet and general Robin boundary conditions. They further use this to compare the numerical convergence orders for certain symplectic Runge-Kutta methods and Peer two-step methods of order four.

Indeed, there are boundary control problems of non-elliptic type. An article by Colli, Gilardi and Sprekels titled *A boundary control problem for the pure Cahn-Hilliard equation with dynamic boundary conditions*[3] discusses the use of Cahn-Hilliard equation as the governing equation, which is a fourth order PDE. Indeed, there are some attempts to solve Cahn-Hillard equations using physics inspired neural networks (PINNs) in the literature, one such work is done by Wight and Zhao titled *Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks*[4], however, this is not boundary control.

Another boundary control problem of non-elliptic type is described in an article written by Cai and Mamadou titled *Boundary Control of Nonlinear ODE/Wave PDE Systems With a Spatially Varying Propagation Speed*[5]. The work is regarding compensating the actuator delays governed by a first-order hyperbolic PDE coupled with a wave PDE dynamics with propagation speed that varies along the spatial dimension. Such a compensator is designed to be globally asymptotically stable with the stability being mathematically proven.

1.3.2 Deep and reinforcement learning in numerical computing

There has been quite a lot of work in attempting to use deep learning and reinforcement learning techniques to expedite numerical computing tasks through less computationally expensive deep learning and reinforcement learning based approximation in the past decade.

To begin with, quite a lot of scientific simulations and engineering analyses involve working with partial differential equations (PDE). Analytical solution for PDEs may not exist in all cases, and, hence, numerical methods are employed to solve for the same.

One of the early work in attempting use deep learning methods to solve PDEs was done by Lagaris, Likas and Fotiadis titled *Artificial neural networks for solving ordinary and partial differential equations*[6]. They attempted deep neural networks to solve initial value problems and boundary value problems and the results suggests to work well in the scenarios tested. Recent work by Jiang, Jiang and Yao titled *A neural network-based PDE solving algorithm with high precision*[7] discusses using residual network architecture that uses correction iteration which is inspired by traditional iterative methods. They claim that their method were able to solve PDEs with an error in the order of 10^{-7}, and it is less sensitive to problem size and as a result can work with large scale problems. They use Residual Network (ResNet) architectures, which involves the use of convolutional layers, in contrast to the usage of dense layers that were popular in the original papers, which enables having fewer trainable parameters, and hence, making it potentially easier to interpret the reason for certain behaviours by the network.

Researchers have also been working on using GNNs to expedite PDE solving. There is a work done by Brandstetter, Worrall and Welling titled *Message Passing Neural PDE Solvers*[8] where they attempt to solve one-dimensional partial differential equations using neural message passing, replacing all heuristically designed components in the computation graph with neural network based function approximators. In the paper titled *Beltrami Flow and Neural Diffusion on Graphs*[9] by Chamberlain, Rowbottom, Giovanni, Dong and Bronstein, the researchers attempt to solve a non-Euclidean diffusion PDE by using positional encoding derived from the graph along with node features.

When talking about numerical computing, a study on stability is quite important. A study by Kloberdanz, Kloberdanz and Le titled *DeepStability: a study of unstable numerical methods and their solutions in deep learning*[10] investigates which deep learning algorithms in popular libraries namely PyTorch and Tensorflow are numerically unstable, and does an analysis to identify the root cause and prescribe patches to these instabilities. This is done using a tool named DeepStability, which is proposed in the paper as well. The authors state that some of the instabilities detected were fixed by them and submitted and merged into these libraries. In the appendix of the paper, their idea is demonstrated with examples of how Softmax and LogSoftmax activation function implementation in these libraries had been unstable and how they modified the same to make it numerically stable.

When talking about using deep learning and reinforcement learning for numerical computing, the work done by DeepMind titled *Discovering novel algorithms with AlphaTensor* is clearly an unexpected one. In an article published in Nature by Fawzi et al. titled *Discovering faster matrix multiplication algorithms with reinforcement learning*[11] discusses how they used reinforcement learning to arrive at a matrix multiplication method that improves upon Strassen's algorithm, which was considered the most optimized for the past fifty years. The interesting thing about this approach is that the algorithms arrived at by the method is provably correct; the method literally arrived at a series of mathematical equations, and, those equations can be solved using Basic Linear Algebra Subprograms (BLAS) libraries, without any libraries for deep learning or reinforcement learning.

1.3.3 Deep and reinforcement learning in control and optimization

The article written by Williams titled *Simple statistical gradient-following algorithms for connectionist reinforcement learning*[12] talks about a class of associative reinforcement learning algorithms called REINFORCE algorithms that makes adjustments in their weights along the direction of the gradient of the expected reinforcement that helps with both immediate-reinforcement tasks, as well as delayed-reinforcement tasks. Backpropagation can be naturally

integrated into these methods. This bridges deep learning and reinforcement learning through policy optimization.

The article by Zhu titled *An optimal control view of adversarial machine learning*[13] shows how an adversarial machine learning method can be "translated" into an optimal control problem formulation. Although the paper discussed more on the adversarial machine learning aspects, it kind of provides quite a lot of information about how advancements in both the fields of control and adversarial machine learning could be used in conjunction.

The articled by Bello et al. titled *Neural Optimizer Search with Reinforcement Learning*[14] has an interesting approach towards finding an optimizer that can train neural networks; they apply language models to put together different functions to form an optimizer, and use reinforcement learning to find which such optimizers converge to the solution faster. This seems to be a clever way to come up with new and potentially better optimizers, or perhaps, ideas to come up with new optimizers.

In a study conducted by Henderson, Romoff and Pineau titled *Where did my optimum go?: An empirical analysis of gradient descent optimization in policy gradient methods*[15] different optimization algorithms like stochastic gradient descend with and without Nestrov momentum, RMSProp, AMSGrad, Adam, Adamax, AdaGrad, AdaDelta, ASGD and YFOptimizer on different policy gradient reinforcement learning benchmarks. The study concludes that the best overall performance was observed by Adam and RMSProp optimizers with small well-tuned learning rates.

At the time of writing this report, yet another paper related to arriving at an improved optimizer for neural network training named Sophia was published, written by Liu et al. titled *Sophia: A Scalable Stochastic Second-order Optimizer for Language Model Pre-training*[16]. Sophia appears to use only the gradient term and calculate momentum on it, it does not use second moment, which is quite similar to stochastic gradient descend with Nestrov momentum. The difference observed is that this method calculates some Hessian estimators every few steps, which are used to scale and the momentum term, following which the whole term is clipped to some upper and lower bounds.

In the work done by Bonny, Kashkash and Ahmed titled *An efficient deep reinforcement machine learning-based control reverse osmosis system for water desalination*[17] describes the approach to control the pressure at the transmembrane used in the reverse osmosis process for desalination, wherein the control is performed using deep deterministic policy-gradient (DDPG) algorithm, which was found to be quite effective.

A demonstration of using agent-based modelling with reinforcement learning to build energy system is discussed in the work by Shen et. al. titled *Multi-agent deep reinforcement learning optimization framework for building energy system*

with renewable energy[18]. The agent logic is modelled using duelling double deep Q-network while an additional value-decomposition network was used to encourage cooperation between multiple such agents. This approach has been shown to reduce the amount of energy wastage and reduce the duration wherein the heating systems were attaining uncomfortable values.

Soft Actor-Critic method described in the paper written by Haarnoja et. al. titled *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*[19] is a great way to combine Q-learning with policy gradient method. The method extends to work on continuous control tasks as well.

There are a few interesting papers that describe the use of graph neural networks in control. The work done by Chen et. al. titled *Graph neural network and reinforcement learning for multi-agent cooperative control of connected autonomous vehicles*[20] wherein connected autonomous vehicles that work with cooperative control by jointly planning how they act on the road. In a paper by Shen et. al. titled *A Graph Neural Network Approach for Scalable Wireless Power Control*[21], describe the use of graph neural networks in solving real-time wireless resource allocation. Furthermore, a work done by Meirom et. al. titled *Controlling graph dynamics with reinforcement learning and graph neural networks* [22] describes the attempt to control partially observable dynamic processes on a graph with exponential state space and combinatorial action space using a combination of graph neural network and policy optimization in reinforcement learning. Indeed, it is important to talk about PyTorch Geometric Temporal library which these works depend on or have contributed to directly or indirectly; this library has an academic publication written by Rozemberczki et. al. with the title *PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models*[23].

1.3.4 Other control and optimization problems and approaches

A preconditioned conjugate gradient method for solving finite-horizon linear-quadratic optimal control problems over two dimensional grids is discussed in the paper written by Zafar and Manchester titled *Structured linear quadratic control computations over 2D grids*[24]. This paper deals information varying along both spatial and temporal dimensions, and hence are arranged in a spatio-temporal graph structure with information exchanged between the nodes in a manner similar to message passing graph neural networks (however, the paper does not seem to be using any neural networks). The method appears to scale linearly in both spatial and temporal dimensions. A work using graph neural networks on the same problem can be seen in the article written by Gama and Sojoudi titled *Graph Neural Networks for Distributed Linear-Quadratic Control*[25].

In the paper by Tasseff et. al. titled *Exploring Benefits of Linear Solver Parallelism on Modern Nonlinear Optimization Applications*[26], a study on comparison of different linear solvers within IPOPT over different problem benchmarks to explore how much parallelism can benefit the overall performance. The problem sets include CUTEst collection, optimal power flow problems, boundary control problems and scalable 2-dimensional and 3-dimensional PDE-constrained problems.

The paper by Kardoš, Kourounis and Schenk titled *Two-Level Parallel Augmented Schur Complement Interior-Point Algorithms for the Solution of Security Constrained Optimal Power Flow Problems*[27] discusses interesting strategies employed to expedite solving for constrained optimal power flow problems using modified interior point methods, whereby performance for both single core execution as well as parallel or distributed execution are improved. Much of this performance boost is achieved by solving block-diagonal-bordered matrix by an augmented Schur Complement decomposition at two-levels, with further improvement in performance achieved by eliminating certain slack variables. This work appears to be a continuation of the work done by the same group of people along with Zimmerman titled *BELTISTOS: A robust interior point method for large-scale optimal power flow problems*[28], which describes a method that improves upon interior point optimization to cater to optimal power flow problems by exploiting how such problems are structured.

An investigation into how to use the standard interior-point method to exascale architectures for block-structured non-linear programming problems is discussed in the paper by Pacaud et. al. titled *Parallel Interior-Point Solver for Block-Structured Nonlinear Programs on SIMD/GPU Architectures*[29]. It begins with distributing computation on multiple processes using coarse parallelism, and then, make each of these processes use SIMD or GPU accelerator. They were able to demonstrate a fifty times speed-up in optimal power flow problems as compared to the aforementioned BELTISTOS method.

1.4 Objective

The key objective of this project is to use deep learning and reinforcement learning techniques to solve boundary control problems. This can be subdivided into two objectives:

- To identify areas where and how deep learning and reinforcement learning methods can be applied to solve boundary control problems.
- To design, apply and evaluate different deep learning and reinforcement learning methods.

Chapter 2

Interior Point Method

This chapter discusses the mathematical intuition behind some state-of-the-art methods used in optimization problems. Much of the content in this chapter is as per my understanding from certain sections in the papers published in the Journal of Computational and Applied Mathematics, written by Potra and Wright titled *Interior-point methods*[30], and, in the Bulletin of the American Mathematical Society, written by Wright titled *The interior-point revolution in optimization: history, recent developments, and lasting consequences*[31].

Interior-point methods (IPMs) are a class of powerful optimization algorithms that have proven to be capable of solving convex optimization problems in an extremely efficient manner. Unlike the methods that follow suit with the simplex method by traversing along the boundaries of the feasible region, IPMs approach the best solution by traversing through the interior of the feasible region. IPMs have been observed to perform quite well in solving boundary control problems.

Among the different subclasses of the IPMs, it appears to be that the primal-dual interior-point methods subclass are most successful for non-linear optimization. The idea behind this appears to be quite not too complicated to demonstrate, mathematically.

2.1 Mathematical idea behind primal-dual IPM

Let us start with the following optimization problem.

$$\begin{aligned} \text{Minimize} \quad & f(x) \\ \text{Subject to} \quad & x \in \mathbb{R} \\ & c_i(x) \geq 0 \text{ for } i = 1, \ldots, m \\ \text{where} \quad & f : \mathbb{R}^n \to \mathbb{R}, c_i : \mathbb{R}^n \to \mathbb{R} \end{aligned}$$

Barrier functions can be used to replace the inequality, thereby converting the inequality-constrained optimization problem to an unconstrained optimization problem. The following uses the Logarithmic barrier function.

$$B(x, \mu) = f(x) - \mu \sum_{i=1}^{m} \log(c_i(x))$$

where μ is a small positive number real number.

Let's calculate the gradient of the barrier function,

$$\begin{aligned} g_b(x, \mu) &= \nabla B(x, \mu) \\ &= \nabla f(x) - \mu \sum_{i=1}^{m} \frac{\nabla c_i(x)}{c_i(x)} \\ &= g(x) - \mu \sum_{i=1}^{m} \frac{J_i(x)}{c_i(x)} \end{aligned}$$

where $g(x) = \nabla f(x), J_i(x) = \nabla c_i(x)$.

Using Lagrange multiplier-inspired dual variable $\lambda \in \mathbb{R}^m$, the following is defined.

$$\begin{aligned} & \lambda_i c_i(x) = \mu \; \forall \; i = 1, \ldots, m \\ \implies & \mu - \lambda_i c_i(x) = 0 \\ \implies & \mu \mathbb{1} - diag(c(x))\lambda = 0 \end{aligned}$$

Applying the standard procedure for optimization, whereby setting the barrier gradient to zero to find the point of local optimum, and substituting the value of μ with that defined in accordance with Lagrange multiplier-inspired dual variable, the equation is transformed as follows.

$$\begin{aligned} & g(x) - \mu \sum_{i=1}^{m} \frac{J_i(x)}{c_i(x)} = 0 \\ \implies & g(x) - \sum_{i=1}^{m} \mu \frac{J_i(x)}{c_i(x)} = 0 \\ \implies & g(x) - \sum_{i=1}^{m} \lambda_i c_i(x) \frac{J_i(x)}{c_i(x)} = 0 \\ \implies & g(x) - \sum_{i=1}^{m} \lambda_i J_i(x) = 0 \\ \implies & g(x) - J(x)^T \lambda = 0 \end{aligned}$$

Now, the mission is to find the values of x and λ in the following equations.

$$g(x) - J(x)^T\lambda = 0$$
$$\mu\mathbb{1} - diag(c(x))\lambda = 0$$

This can be done by using root-finding algorithms; one such algorithm is the Newton-Raphson method.

Newton-Raphson method:
It is an iterative root-finding algorithm that works as follows:

$$x_n = x_{n-1} + \frac{f(x_{n-1})}{f'(x_{n-1})}$$

The update term in this can be written as,

$$p = x_n - x_{n-1} = \frac{f(x_{n-1})}{f'(x_{n-1})}$$

Which can be rearranged to get the following:

$$f'(x_{n-1})p = f(x_{n-1})$$

To use the Newton-Raphson method, the derivative of the equations being dealt with need to be found; with respect to x and with respect to λ. Simplifying this produce,

$$\frac{\partial(g(x) - J(x)^T\lambda)}{\partial x} = H(B(x,\mu))$$
$$\frac{\partial(g(x) - J(x)^T\lambda)}{\partial \lambda} = -J(x)^T$$

$$\frac{\mu\mathbb{1} - diag(c(x))\lambda}{\partial x} = -diag(\lambda)J(x)$$
$$\frac{\mu\mathbb{1} - diag(c(x))\lambda}{\partial \lambda} = -diag(c(x))$$

Instead of having individual update terms for each of these derivatives, a combined update term along x and λ direction could be used, which may be referred to as p_x and p_λ, respectively. The equation for the same is shown below.

$$H(B(x,\mu))p_x - J(x)^T p_\lambda = g(x) - J(x)^T\lambda$$
$$-diag(\lambda)J(x)p_x - diag(c(x))p_\lambda = \mu\mathbb{1} - diag(c(x)$$

Writing this in the matrix form would give,

$$\begin{bmatrix} H(B(x,\mu)) & -J(x)^T \\ -diag(\lambda)J(x) & -diag(c(x)) \end{bmatrix} \begin{bmatrix} p_x \\ p_\lambda \end{bmatrix} = \begin{bmatrix} g(x) - J(x)^T\lambda \\ \mu\mathbb{1} - diag(c(x)) \end{bmatrix}$$

Now, solve for p_x and p_λ. Once that is achieved, update x and λ as shown below.

$$x \leftarrow x - \alpha p_x$$
$$\lambda \leftarrow \lambda - \alpha p_\lambda$$

where α is a positive real number, which is kind of similar to the concept of learning rate in deep learning training.

2.2 Potential limitations in IPMs

By looking at the mathematical intuition behind IPMs, a few issues and limitations can be observed.

- It appears to be that IPMs are quite likely to converge to local minima, which need not necessarily be global minima. This also means that the method is sensitive to the initial points that are chosen from which it starts to perform the minimization procedure; it highly dictates which minima it reaches.
- The fact that each iteration involves computation of inverse, Hessian and Jacobian, interior point methods may indeed be computationally intensive. However, it has been observed that methods using IPMs converge faster, and hence fewer iterations are involved. Perhaps the methods that use the idea behind IPMs may have other tricks up their sleeves.

2.3 About IPOPT

IPOPT is the abbreviation for Interior Point OPTimizer (IPOPT), which is a fast and powerful tool that helps in solving large-scale nonlinear programming problems, which can work with even a million variables. IPOPT is an open-source software package released under Eclipse Public License (EPL) that is based on IPMs. The project is now maintained by Computational Infrastructure for Operations Research (COIN-OR).

The initial work and detailed information corresponding to this solver is available in the research paper written by Wächter and Biegler titled *On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming*[32].

IPOPT uses the primal-dual interior-point method, which we discussed in detail earlier. It utilises sparse matrix operations and can run with different libraries that implement Basic Linear Algebra Subprograms (BLAS) including, but not limited to, Intel Math Kernel Library and OpenBLAS. The usage of filter line-search in conjunction to IPM enables keeping a track of previously explored points in the solution space and helps avoid revisiting these points, thereby enabling to strike a better balance between exploration and exploitation.

IPOPT supports a variety of problem formulations, and it's prowess is showcased with a few examples in it's codebase. This includes a few examples that are related to solving boundary control problems with Dirichlet and Neumann boundary conditions, which is of great interest in this project.

Although IPOPT is primarily written in C++, there are interfaces that enables it to be used in MATLAB and Python as well. This enables researchers to use this tool to write highly performant code as well as perform easy experiments with interpreted languages.

Chapter 3

Deep and Reinforcement Learning

Deep learning and reinforcement learning techniques have found quite a lot of applications in the past decade. This chapter provides an overview about these techniques and introduce some of the terms that are required to better understand the proposed method.

3.1 Deep learning

Deep learning is a branch of machine learning that is concerned with algorithms involving multiple interconnected layers of nodes, often called "perceptrons", inspired by the biological structure and function of our brain cells. These nodes, along with algorithms to tweak their parameters, work together to learn from and generate output based on incoming data.

There is a fundamental theorem in the field of neural networks called the Universal Approximation Theorem, which states that any continuous function can be approximated by a feed-forward network that is sufficiently wide and deep. There are two cases of this theorem, which are *arbitrary width case* and *arbitrary depth case*. In *arbitrary width case*, the network can be just one layer deep and still approximate any function accurately, provided it is wide enough. However, in *arbitrary depth case*, there is a minimum width requirement for the network, beyond which a very deep network can approximate any function effectively.

Deep learning has been rapidly gaining traction and is being widely used across several domains today. It has revolutionized the field of computer vision by enabling the creation of highly accurate models for classification, object detection, and semantic segmentation tasks; enabling automation of analyzing visual information. Natural language processing is another field that deep learning methods have been enabling making tremendous progress; advancements in machine translation, sentiment analysis, text generation and

speech recognition can be directly attributed to recurrent and transformer-based neural networks. Sometimes, deep learning models act as "Gods" and decide what content users are to watch as well; they are now being extensively used in recommender systems and search algorithms.

Deep learning methods have found widespread adoption in the past decade and a half. This can be attributed to the availability of large datasets to train these networks, and, the increase in computational power. Furthermore, the availability of open-source libraries/frameworks like TensorFlow, Keras, PyTorch and various others have enabled the common man to access and create their own models. This democratization of deep learning methods has led to an expedition in the development of the field as well.

Now, the availability of computational resources has been increasing exponentially as well. The advancements in cloud computing have enabled distributed computing capabilities to train complex deep learning models on massive datasets at scale. With the help of cloud computing, deep learning models have also been empowered to perform real-time and streaming data analysis, enabling support for real-time fraud detection and predictive maintenance.

There are several different types of deep learning architectures that are being widely used. To tackle the problem at hand, architectures that can learn functions that take spatial and temporal information to arrive at the output may be the right choice. The following subsections cover the broad categories of such methods that are applied in this project; a lot of the information being inspired by the work done to summarize the same by Schmidhuber[33], LeCun et. al.[34] and Zhou et. al.[35].

3.1.1 Convolutional neural networks

A convolutional neural network (CNN) is a type of artificial neural network that is used to process data that has a spatial structure. It is predominantly used in various image processing-related applications including but not limited to object detection, and, image classification. It is also often used in speech recognition and static weather prediction.

A typical network of this type includes several layers that apply convolution operation to the input given to them. Such convolution layers can receive multi-dimensional data as input and output multi-dimensional data; it can be used for size invariant applications as well.

CNNs may also feature pooling layers, which are often used to create outputs that have smaller dimensions than the input. It can also have fully connected layers; however, this would require inputs and outputs that are fixed in size. Furthermore, different kinds of activation functions can be used with these networks; the most common being Rectified Linear Unit (ReLU) function.

CNNs have finite impulse response, which means that they can be replaced by strictly feed-forward neural networks by unrolling.

3.1.2 Recurrent neural networks

A recurrent neural network (RNN) is a type of artificial neural network that is used to process data that has a temporal structure. These networks have found applications in speech recognition, translation and text generation.

RNNs have some form of internal state memory that helps in accounting for information that they had encountered in the past.

There are several different kinds of layers that can be used in RNNs; Long short-term memory (LSTM) and Gated recurrent unit (GRU) are two of the most widely used layers. They are often horizontally and/or vertically stacked in order to tackle different problems.

RNNs have infinite impulse response, and, hence, can not be replaced by strictly feed-forward neural networks.

3.1.3 Graph neural networks

A graph neural network (GNN) is a type of artificial neural network that is used to process or learn from data that has some structure, that does not necessarily have to be grid-like. GNNs effectively capture the abstract idea behind neural networks in some sense, whereby enabling learning complex relationships similar to dense networks while also enabling the ability to have sparsification in a more elaborate manner than CNNs.

GNNs use message-passing mechanisms to propagate information through the graph. The nodes in the graph aggregate and update their representations based on the information from neighbouring nodes, enabling the network to learn complex representations.

Equivalent to CNNs and RNNs, GNNs have layers with similar properties that can extend to graphs, which are, Graph Convolutional, Graph Pooling, Graph Attention, Graph Attention Pooling and Graph LSTM/GRU layers. Additionally, GNNs feature complementary layers, Readout and Broadcast layers, which respectively collect all node representations in a graph to form a graph representation and send the graph representation to every node in the graph.

3.1.4 Spatio-temporal neural networks

A spatio-temporal neural network (STNN), also known as space-time neural network, is a type of artificial neural network that is used to process data that has both spatial and temporal structure. In essence, these networks

encapsulate CNNs and RNNs; however, the networks may feature graph-based layers as a replacement for or in addition to CNNs to handle spatial structures, and, attention-based mechanisms as a replacement for or in addition to RNNs to handle temporal structures. These are often used in video analysis, object tracking, pose detection and climate modelling.

3.2 Reinforcement learning

Reinforcement learning is a subset of machine learning that is based on providing positive and negative feedback for desired and undesired behaviours, respectively. It is an unsupervised way of learning, whereby the learning agent learns by interacting with the environment through trial and error. The feedback given to the learning agent is called a reward. The objective of the agent is to maximize the same.

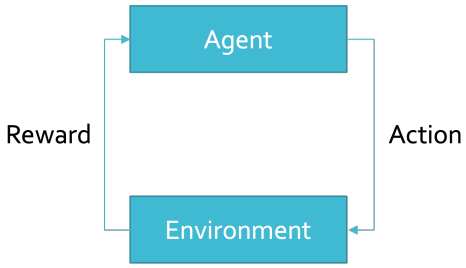


Figure 3.1: A high-level diagram depicting Reinforcement Learning

Unlike supervised learning methods that involve external entities with higher knowledge to provide examples, reinforcement learning learns by interacting with the environment and seeking a goal. From a high-level, a reinforcement learning problem can be modelled as a Markov Decision Process; which can be defined by a tuple (S, A, P_a, R_a), wherein

- S is the set of all states that an agent can be in.
- A is the set of all actions that an agent can take.
- $P_a(s, s')$ is the state transition probability, i.e. the probability that taking action $a \in A$ while in state $s \in S$ would lead to state $s' \in S$.
- $R_a(s, s')$ is the reward received after making the action $a \in A$ to transition from state $s \in S$ to state $s' \in S$.

However, in Reinforcement Learning, the probabilities and rewards used in the Markov Decision Process are generally unknown and need to be learned through experience.

Reinforcement learning methods are being rapidly adopted in several different fields today. These are being used in robotics and self-driving cars, where adapting to complex and dynamic environments is necessary. Often, these are tested virtually in agent-based modelling and scenario simulation environments where multiple agents using similar reinforcement learning methods interact with each other, and, over time, find more suitable and fine-tuned algorithms. Reinforcement learning methods have also found applications in the finance industry, where it is being used for portfolio management, algorithmic trading and risk assessment, in the energy section, where it is being used to optimize energy consumption and resource allocation, and in the game playing industry, where it has shown superhuman performances.

One key factor that has contributed to the wide adoption of deep learning methods has also contributed to the adoption of reinforcement learning methods - the availability of powerful computational resources. In fact, reinforcement learning methods are encapsulating deep learning methods to create powerful and adaptive decision-making systems. This integration has enabled the creation of sophisticated models that can perceive, understand and make decisions on unstructured data that may even have a huge number of dimensions.

In March 2016, Google's Deepmind came up with a reinforcement learning based program named AlphaGo[36], learned how to play the game named 'Go' and defeated Lee Sedol, the most highly rated player in the game at the time. The game has 250^{150} (about 10^{360}) possible moves, and simulating all these possible moves would be impossible even with an enormous universe-sized computer, as the total number of atoms in the universe is estimated by be between 10^{78} to 10^{82}. For a really long time, people thought that this game could only be mastered by a human, but reinforcement learning proved them wrong. This one instance had a profound impact on everyday people's perception of the capabilities and potential of reinforcement learning.

3.2.1 Terminologies and basic equations

Much of the content in this subsection is my understanding of the book *Reinforcement Learning, second edition: An Introduction*[37]. The following are some of the important terminologies in Reinforcement Learning, and equations to support the understanding of the same.

Policy

Policy function is, in essence, a mapping between an action and a state. This could be deterministic, whereby the function outputs either a 0 or a 1,

or probabilistic, whereby the function outputs a probabilistic distribution between 0 and 1.

$$\pi : S \times A \longrightarrow [0,1]$$
$$\pi(a,s) = Pr(a_t = a \mid s_t = s)$$

The objective of the Reinforcement Learning agent is to learn a policy that maximizes the expected cumulative return.

Return

Return is the accumulation of rewards over time. In general, it is defined as the sum of discounted rewards, as shown below.

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

where γ is the discount factor, $0 \leq \gamma < 1$, and, r_i is the reward obtained at the i^{th} step. Note that the discount factor helps in ensuring that solutions using fewer steps are more favourable.

State-value function

State-value function, sometimes simply referred to as value function, defines the value of being in a state while following a particular policy. Mathematically, it can be depicted as shown below.

$$\begin{aligned} V_\pi &: S \longrightarrow \mathbb{R} \\ V_\pi(s) &= E_\pi(G_t | s_t = s) \\ &= \sum_{a \in A} \pi(a|s) \sum_{s' \in S} [r(s,a) + P_a(s',s)\gamma V_\pi(s')] \end{aligned}$$

Action-value function

Action-value function, sometimes also referred to as the Q-function, is similar to the state-value function, but it also takes action into account. Mathematically, it can be depicted as shown below.

$$\begin{aligned} Q_\pi &: S \times A \longrightarrow \mathbb{R} \\ Q_\pi(s,a) &= E_\pi(G_t | s_t = s, a_t = a) \\ &= \sum_{s' \in S} [r(s,a) + P_a(s',s)\gamma V_\pi(s')] \end{aligned}$$

Bellman equations and Bellman optimality equations

The aforementioned action-value and state-value functions can be modified in such a way that the immediate reward and future reward terms can be separated. This gives us the Bellman equations.

$$\begin{aligned} V_\pi(s) &= \sum_{a \in A} \pi(a|s) \sum_{s' \in S} [r(s,a) + P_a(s',s)\gamma V_\pi(s')] \\ &= \sum_{a \in A} \pi(a|s) \left[r(s,a) + \gamma \sum_{s' \in s} P_a(s',s) V_\pi(s') \right] \end{aligned}$$

$$\begin{aligned} Q_\pi(s,a) &= \sum_{s' \in S} [r(s,a) + P_a(s',s)\gamma V_\pi(s')] \\ &= r(s,a) + \gamma \sum_{s' \in s} P_a(s',s) V_\pi(s') \\ &= r(s,a) + \gamma \sum_{s' \in s} P_a(s',s) \sum_{a' \in A} \pi(a'|s') Q_\pi(s',a') \end{aligned}$$

Given an optimal policy, i.e. a policy that is better than or at least as good as all other policies, the mathematical equations formed for state-value and action-value functions obtained using this policy from Bellman equations are called Bellman optimality equations. These are respectively denoted by V_* and Q_*.

$$V_*(s) = \max_{a \in A} \left[r(s,a) + \gamma \sum_{s' \in S} P(s',s,a) V_*(s') \right]$$

$$\begin{aligned} Q_*(s,a) &= r(s,a) + \gamma \sum_{s' \in S} P_a(s',s) V_*(s') \\ &= r(s,a) + \gamma \sum_{s' \in S} P_a(s',s) \max_{a' \in A} Q_*(s',a') \end{aligned}$$

3.2.2 Generalized policy iteration

There are two simultaneous interacting processes that work together in policy iteration. These are policy evaluation and policy improvement. Essentially, policy evaluation is performed to make the state-value function consistent with the current policy. Similarly, policy improvement is performed to make the policy choose the best possible (greedy) value with respect to the current state-value function. The idea is that if these two processes are repeated, then the policy and value functions converge to the optimal policy and optimal value functions. Figure 3.2 shows this in the form of a diagram.

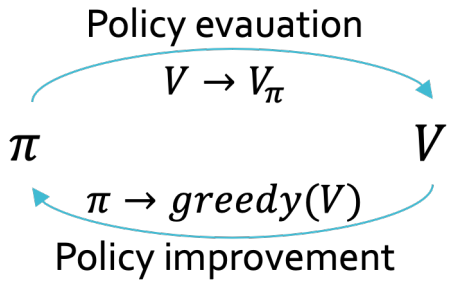


Figure 3.2: Depicting how generalized policy iteration works

3.2.3 Classification of reinforcement learning algorithms

Reinforcement learning methods can be classified into two, namely Model-based and Model-free methods.

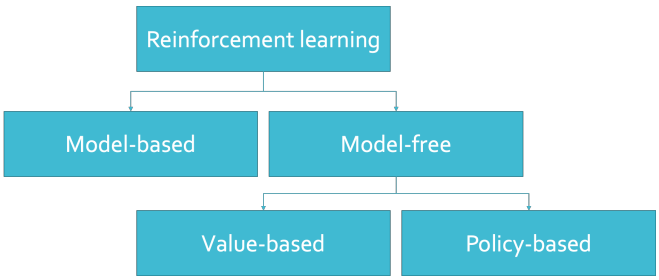


Figure 3.3: Classification of reinforcement learning algorithms

Model-based reinforcement learning methods rely on a known model of the environment; the agent explores that environment and learns it. These involve the explicit creation of an internal representation of the dynamics of the environment, which includes transition probabilities and rewards for different actions and states. Using this, model-based algorithms can simulate possible trajectories and predict expected outcomes. Generally, this kind of method would work in cases where accurate models are available and the environments are not very complicated; as the environmental complexity increases, the computational cost required for planning also increases.

Conversely, Model-free reinforcement learning methods do not rely on known models; rather, they learn the equivalent of a model from interactions with the environment. This is done by optimizing policies or estimating value functions without prior knowledge of the dynamics. By employing exploration in a trial-and-error manner, they update their policy or value estimates based on the rewards received from the environment. Thus, this kind of

method would work in cases where there exists no accurate model or when the environment is complicated. Model-free methods can be further classified into two, which are Value-based and Policy-based methods.

1. Value-based methods attempt to optimize the value function. The value function calculates the expected cumulative reward an agent can receive for a state or an action. Value-based methods are sometimes referred to as Q-learning methods, as in a lot of literature the action value function is represented by 'Q'. You can observe this in the equations in Section 3.2.1 as well.
2. Policy-based methods attempt to optimize the policy function. The policy function defines the agent's strategy to select actions at different states. In contrast to estimating the value function, these methods parameterize the policy and use optimization algorithms to update the policy parameters to maximize the expected cumulative reward.

Recently, there has been a trend towards combining both Value-based and Policy based Model-free reinforcement learning algorithms, known as actor-critic methods. In these, value function assumes the role of a critic while policy assumes the role of an actor. The critic estimates the value of actions done by or state achieved by the actor, thereby guiding the actor's behaviour. This enables efficient learning and exploration.

Additionally, another trend being observed is that both Policy-based and Value-based methods have been using deep learning methods to learn the optimal policies and values, respectively. Some implementations of the policy gradient method and Deep Q-learning follow this suite.

3.2.4 Policy gradient method

The policy gradient method is an example of a policy-based model-free reinforcement learning method. These methods attempt to directly optimize the policy parameters using gradient descent, wherein gradients are calculated with respect to the expected long-term cumulative reward. Gradient-based optimization enables policy gradient methods to handle both discrete and continuous action spaces.

Policy gradient methods have a few key advantages over other reinforcement learning methods. Firstly, they have the ability to handle continuous and high-dimensional action spaces as they directly optimize the policy without relying on the value function approximation. Secondly, they have the ability to handle stochastic policies which helps in scenarios that require exploration and often have uncertainties in the environment. Additionally, it is quite straightforward to incorporate prior knowledge and additional constraints into the learning model in policy gradient methods as all of this information can be a part of the policy. However, one key limitation of the policy

gradient method that arises due to the usage of gradient-based optimization procedures is that they typically require a larger number of samples to train on to achieve good performance.

It should be mentioned that AlphaGo was using a variation of the policy gradient method. This shows how powerful and intuitive this method is.

3.3 Optimization algorithms used in deep learning and reinforcement learning

Most of the popular optimization algorithms used in deep learning and reinforcement learning to minimise the loss or cost functions are based on gradient descent, also known as steepest descent, which is a first-order iterative optimization algorithm.

Gradient descent iteratively moves in the opposite direction of the calculated or approximated gradient of the function at the current point, eventually reaching the point of local optimum. Mathematically, if $f(\theta)$ is a function of a parameter θ, and, θ_n and θ_{n+1} corresponds to the value of the parameter at n^{th} and $(n+1)^{th}$ iterations respectively, then,

$$\theta_{n+1} = \theta_n - \gamma \nabla f(\theta_n) \tag{3.1}$$

The following are some of the frequently used methods that are based on gradient descent used in deep learning and reinforcement learning:

- **Stochastic Gradient Descend (SGD):**

  SGD[38] is the method in which you randomly sample a subset of the training data and proceed with gradient descend with it. However, recently, it has been observed that there are a few more parameters that are added to this method without actually introducing a new terminology for the same, which are to introduce momentum, acceleration (Nesterov momentum[39]) and dampening of the gradient. The algorithm for SGD is shown in Algorithm 1.

- **Root Mean Square Propagation (RMSProp):**

  RMSProp addresses the diminishing learning rate problem in SGD by maintaining the exponentially decaying average of squared gradients to normalize the updates. This method has been found effective in handling non-stationary objectives and has been observed to converge faster. The algorithm for RMSProp is shown in Algorithm 2.

  Interestingly, this method was not originally published in a scientific paper; rather it first appeared in a course in Coursera by Hinton titled *Neural Networks for Machine Learning*[40].

Algorithm 1 Stochastic gradient descend with momentum, acceleration and dampening

procedure SGD($\theta, f(\theta), \gamma, \lambda, \mu, \tau, a$)
  for n = 1 ... N **do**
    $g_n \leftarrow \nabla f(\theta_n)$ ▷ Calculate gradient
    $g_n \leftarrow g_n + \lambda\theta_{n-1}$ ▷ Weight decay based update
    if $\mu \neq 0$ **then**
      if $n > 1$ **then**
        $b_n \leftarrow \mu b_{n-1} + (1-\tau)g_n$ ▷ Momentum and dampening
      else
        $b_n \leftarrow g_n$
      end if
      if a is true **then**
        $g_n \leftarrow g_n + \mu b_n$ ▷ Accelerated gradient
      else
        $g_n \leftarrow b_n$
      end if
    end if
    $\theta_n \leftarrow \theta_{n-1} - \gamma g_n$ ▷ Scaled by learning rate
  end for
  return θ_N ▷ The final value of θ
end procedure

Algorithm 2 Root Mean Square Propagation

procedure RMSProp($\theta, f(\theta), \gamma, \alpha, \lambda, \mu$)
  for n = 1 ... N **do**
    $g_n \leftarrow \nabla f(\theta_n)$ ▷ Calculate gradient
    $g_n \leftarrow g_n + \lambda\theta_{n-1}$ ▷ Weight decay based update
    $v_n \leftarrow \alpha v_{n-1} + (1-\alpha)g_n^2$
    $\theta_n \leftarrow \theta_{n-1} - \gamma g_n / (\sqrt{v_n} + \varepsilon)$
  end for
  return θ_N ▷ The final value of θ
end procedure

- **Adaptive Gradient (AdaGrad):**

  AdaGrad[41] scales down the learning rate for frequently updated parameters and scales up the learning rate for infrequently updated parameters. This has been observed to be beneficial for handling sparse data. The algorithm for AdaGrad is shown in Algorithm 3.

Algorithm 3 Adaptive Gradient

procedure AdaGrad($\theta, f(\theta), \gamma, \alpha, \lambda, \mu, \eta$)
  for n = 1 ... N **do**
    $g_n \leftarrow \nabla f(\theta_n)$ ▷ Calculate gradient
    $g_n \leftarrow g_n + \lambda\theta_{n-1}$ ▷ Weight decay based update
    $\hat{\gamma} \leftarrow \gamma/(1 + (n-1)\eta)$ ▷ Learning rate scaling
    $state\_sum_n = state\_sum_{n-1} + g_n^2$
    $\theta_n \leftarrow \theta_{n-1} - \hat{\gamma} g_n / (\sqrt{state\_sum_n} + \varepsilon)$
  end for
  return θ_N ▷ The final value of θ
end procedure

- **AdaDelta:**

  AdaDelta[42] is a small modification on AdaGrad, wherein the aggressive monotonically decreasing learning rate is rescaled based on past gradients. The algorithm for AdaDelta is shown in Algorithm 4.

Algorithm 4 AdaDelta

procedure AdaDelta($\theta, f(\theta), \gamma, \alpha, \lambda, \rho$)
  for n = 1 ... N **do**
    $g_n \leftarrow \nabla f(\theta_n)$ ▷ Calculate gradient
    $g_n \leftarrow g_n + \lambda\theta_{n-1}$ ▷ Weight decay based update
    $v_n \leftarrow v_{n-1}\rho + g_n^2(1-\rho)$
    $\Delta x_n \leftarrow \frac{\sqrt{u_{n-1}+\varepsilon}}{\sqrt{v_{n-1}+\varepsilon}} g_n$
    $u_n \leftarrow u_{n-1}\rho + \Delta x_n^2(1-\rho)$
    $\theta_n \leftarrow \theta_{n-1} - \gamma\Delta x_n$
  end for
  return θ_N ▷ The final value of θ
end procedure

- **Adaptive Moment Estimation (Adam):**

  Adam[43] is arguably the most popular among all these methods. It combines the advantages of both AdaGrad and RMSProp by adapting the learning rate for each parameter based on the first and the second

moments of gradients. The algorithm for Adam is shown in Algorithm 5.

Algorithm 5 Adam

procedure ADAM($\theta, f(\theta), \gamma, \lambda, \beta_1, \beta_2$)
  for n = 1 ... N **do**
    $g_n \leftarrow \nabla f(\theta_n)$ ▷ Calculate gradient
    $g_n \leftarrow g_n + \lambda\theta_{n-1}$ ▷ Weight decay based update
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$ ▷ First moment
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ ▷ Second moment
    $\hat{m}_t \leftarrow m_t / (1-\beta_1)$
    $\hat{v}_t \leftarrow v_t / (1-\beta_2)$
    $\theta_n \leftarrow \theta_{n-1} - \gamma \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$
  end for
  return θ_N ▷ The final value of θ
end procedure

- **AdamW:**

  AdamW[44] is a modified version of Adam that incorporates weight decay regularization in order to prevent overfitting. This is done by adding an L2 regularization term to the weight update rule, which encourages smaller weights and better generalization. The algorithm for AdamW is shown in Algorithm 6.

Algorithm 6 AdamW

procedure ADAMW($\theta, f(\theta), \gamma, \lambda, \beta_1, \beta_2$)
  for n = 1 ... N **do**
    $g_n \leftarrow \nabla f(\theta_n)$ ▷ Calculate gradient
    $g_n \leftarrow g_n + \lambda\theta_{n-1}$ ▷ Weight decay based update
    $\theta_n \leftarrow \theta_{n-1} - \gamma\lambda\theta_{n-1}$
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$ ▷ First moment
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ ▷ Second moment
    $\hat{m}_t \leftarrow m_t / (1-\beta_1)$
    $\hat{v}_t \leftarrow v_t / (1-\beta_2)$
    $\theta_n \leftarrow \theta_{n-1} - \gamma \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$
  end for
  return θ_N ▷ The final value of θ
end procedure

Indeed, there are several algorithms that may or may not share similarities with the aforementioned methods. Some may be specific to certain cases,

while some are more general. Taking inspiration from these, it may be possible to come up with new and improved optimizers that could work better for certain specific scenarios.

3.4 Control and machine learning

There is a combined subfield of machine learning and control theory called Machine Learning Control (MLC), which deals with applying complex non-linear systems for control, wherein simple linear control theory may not work with reasonable accuracy.

In cases where there is a plethora of data, data-driven modelling control approaches could work. These approaches can leverage large datasets and train machine learning techniques that can uncover hidden patterns and relationships embedded in the data. Such an approach can work with non-linear and dynamic systems.

Additionally, reinforcement learning techniques are being used to create adaptive control algorithms that can dynamically adjust control parameters based on real-time feedback. Such a system can adapt to changes and uncertainties that can arise in the real-life scenario. For example, in machines that work in an assembly line, such uncertainties can occur due to the change in weather conditions, wear and tear of tools and workpieces, incorrect positioning of materials in a conveyor belt, etc. Autonomous vehicles are another example that combines control with machine learning that everyday people can relate to. These systems use the aforementioned adaptive control algorithms for decision-making based on real-time sensor inputs. This adaptability enables the creation of control systems that are, perhaps, more robust and versatile in dealing with unforeseeable scenarios.

It is evident that machine learning methods can be used to create or improvise optimization algorithms, especially using deep learning and reinforcement learning. Using these methods, controllers could be designed to be able to arrive at approximate control policies for complex systems. It is quite interesting that the terminologies used in control systems and reinforcement learning have a huge intersection. To be frank, even at an initial glance, one can observe that the structure and functioning of closed-loop control systems bear striking resemblance to reinforcement learning mechanisms.

3.5 About PyTorch

PyTorch[45] is one of the most popular deep learning frameworks, which was originally developed by Facebook's AI research lab (now called Meta AI), but now maintained by the Linux Foundation. PyTorch is an open-source

software package released under a modified BSD license, and is used by some top-notch companies like Tesla, Uber and Hugging Face.

One of the key strengths of PyTorch is that it is extremely flexible and easy to use at the same time. This enables easy experimentation to aid in attempting to create and experiment with new methods and algorithms. This allows PyTorch to be one of the most preferred libraries in AI research. Being popular has its merits; it has robust and active community support.

The primary interface for PyTorch is Python, which is extensively documented with examples and illustrations; however, there is a C++ interface, which is perhaps still being improved. It is quite easy to implement and debug code in Python; being an interpreted language, one can easily add a breakpoint anywhere and view intermediate values in Python.

PyTorch also has easy interface to enable hardware acceleration using GPUs in a very seamless manner. The code modification required for the same is quite minimal. Furthermore, it has support to enable distributed computing enabling the methods developed to be run in a cluster of computers.

Chapter 4

Design and Methodology

In this chapter, a brief overview regarding the preparations made to tackle the problem at hand. This includes information about the general setup and the approach taken.

4.1 High-level architecture

Discovering where to use deep learning and reinforcement learning methods is the first key step in this project. This involves architecting the approach towards solving the problem itself.

As a starting point, copious amount of existing iterative optimization algorithms were explored, like Jacobi iteration and gradient descent (and their variants), and observed certain features these have in common. It can be observed that each of these methods start with an initial guess value and then iteratively update this until they reach a solution that is assumed to be good enough, or, until a maximum number of iterations or when the solution values keep looping.

Now, it is to be noted that the boundary control problem is essentially an optimization problem, and, hence, can be modeled along the same line. The values for the boundaries need to be found and there exist a cost function that is based on the objective and the constraints.

Thinking along this direction, it becomes evident on where deep learning and reinforcement learning methods can be applied. Clearly, making an informed initial guess that takes the parameters of the problem into consideration can be done using a deep learning model, and, the optimizer used in updating the boundary values at each iteration could be treated as a policy, and, hence, can be approached using reinforcement learning techniques. You can see this depicted pictographically in Figure 4.1.

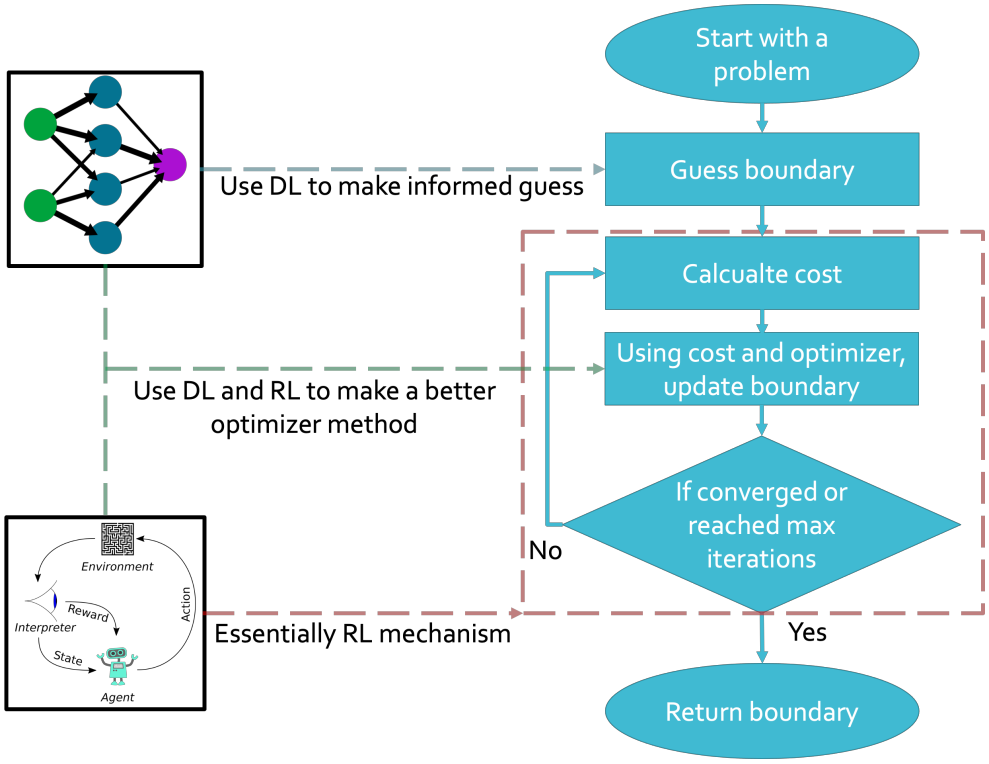


Figure 4.1: Iterative optimization based architecture with Deep Learning and Reinforcement Learning

4.2 Defining cost and reward

In a simplified mathematical sense, the boundary control problem can be written as shown below.

$$\begin{aligned} \underset{y,u}{\text{minimize}} \quad & \frac{1}{2}\int_{\Omega}(y(x)-y_d(x))^2\,dx + \frac{\alpha}{2}\int_{\Gamma}(u(x)-u_d(x))^2\,dx \\ \text{Subject to} \quad & \nabla^2 y = c,\ y_{min} < y < y_{max},\ u_{min} < u < u_{max} \end{aligned}$$

where Ω is the domain, Γ is the boundaries, y_d is the desired values for the domain, u_d is the desired values for the boundaries, y is the values for the domain, u is the values for the boundaries, α is a non-negative constant that determines how much weight must be given to the cost for the boundaries, c is a constant sourcing term, y_{min} and y_{max} are upper and lower bound of y, and, u_{min} and u_{max} are upper and lower bound of u.

In addition to the objective function (which is the cost function) that is to be minimized, there are additional constraints. Using a method that can output boundary values and the domain values are solved using a numerical PDE solver, then the constraint corresponding to the governing PDE would be

automatically satisfied. However, the constraints corresponding to the upper and lower bound satisfaction need to be explicitly handled, and a way to do that is to incorporate these into the cost function itself.

The modified cost can be defined as the sum of costs contribution by the objective function, constraint violation in the domain and constraint violation in the boundaries. Mathematically, this can be written and defined as:

$$
\begin{aligned}
F &= F_o + \beta F_v \\
F_o &= \frac{1}{2}\int_\Omega \left(y(x) - y_d(x)\right)^2 dx + \frac{\alpha}{2}\int_\Gamma \left(u(x) - u_d(x)\right)^2 dx \\
F_v &= \int_\Omega f_\Omega(x)dx + \int_\Gamma f_\Gamma(x)dx \\
f_\Omega(x) &= \begin{cases} (y(x) - y_{min})^2 & \text{if } y(x) \in (-\infty, y_{min}) \\ (y(x) - y_{max})^2 & \text{if } y(x) \in (y_{max}, \infty) \\ 0 & \text{otherwise} \end{cases} \\
f_\Gamma(x) &= \begin{cases} (u(x) - u_{min})^2 & \text{if } u(x) \in (-\infty, u_{min}) \\ (u(x) - u_{max})^2 & \text{if } u(x) \in (u_{max}, \infty) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

where β is the penalty factor for constraint violation, which should have a large positive value.

A quadratic cost for constraint violation was chosen because this would ensure that the function is continuous and differentiable for all real numbers. Logarithmic barrier functions were considered prior to choosing this; it was dropped because the cost would become infinite as it approaches the boundaries, and it would not defined for values beyond the boundaries, which would make it hard to train a neural network with.

For reinforcement learning, the reward could be modeled by a strictly monotonous decreasing function of the cost function. This means that when the agent tries to maximize the reward, it would essentially be minimizing the cost function. For simplicity, the reward function could simply be the negative of the cost function.

4.3 Data generation

In the work done by Maurer and Mittelmann[1], a total of eight different boundary control problems are discussed. Out of these, four problems have Dirichlet boundary conditions while the other four have Neumann boundary conditions. To scope the work done here, a decision has been made to work only on Dirichlet boundary conditions as it appears to be more straightforward to get started with. There are only four different problems

satisfying this condition in the paper mentioned here, which is quite few in number.

In order to train, test and validate a neural network, a lot more problems need to be made available. Training the network on only a few problems would increase the likelihood that it overfits; essentially acting like a sophisticated lookup table, which is undesirable. Therefore, a problem generator needs to be created that takes inspiration from the original set of problems, and it can create tens of thousand of problems.

4.3.1 Describing a problem

As evident from Section 4.2, the following are the parameters that describes a problem.

- *Domain size*: The size of the side of the square domain, denoted by N.
- *Alpha factor*: The penalty for violation of desired values at the boundary is scaled by this term, denoted by α.
- *Lower bound for domain*: The smallest value a cell in the domain can assume, denoted by y_{min}.
- *Upper bound for domain*: The largest value a cell in the domain can assume, denoted by y_{max}.
- *Lower bound for boundary*: The smallest value a cell in the boundary can assume, denoted by u_{min}.
- *Upper bound for boundary*: The largest value a cell in the boundary can assume, denoted by u_{max}.
- *Sourcing term*: The value for the sourcing term in the Poisson's equation, denoted by c.
- *Target profile equation*: The equation that describes the desired values in the domain. The values for the desired value of y, denoted by y_d, is generated using this equation.

There may be certain parameters that are not mentioned in the list above. These are kept constant for each and every problem being generated. For example, poisson's equation is used as the governing PDE in all problems. Another example would be that the desired values at the boundaries are set to zero for all problems.

4.3.2 Considerations for the generator

Taking a look at the original set of problems and experimenting different possibilities, the following are the intuitive considerations that needs to be followed to have a problem that may be considered realistic enough.

1. The target profile equations may contain only quadratic and sine squared terms. The coefficients along with these terms may be limited to randomly generated bounded integer values.
2. The upper and lower bound values that the boundaries and domains can take should take the target profile equation into account.
3. The domain size may be randomly generated bounded positive integers. Extremely small domain sizes would be unrealistic while large sizes would take a long time to solve to train the network.
4. A fixed value for *alpha* may be used.
5. Let us deal with constant sourcing term; it shall be randomly generated bounded negative integers.
6. Make sure that the IPOPT solution would have a cost less than a certain threshold value.

4.3.3 Steps taken to create the generator

To write a program that randomly generates equations based on the aforementioned considerations, a decision needs to be made on the range in which parameters and coefficients can vary, in order to create problems that are reasonable.

Domain size

It has been decided that the domain size value shall be randomly chosen to be an integer value between 10 and 100.

Alpha factor

The penalty for violating the boundary conditions should be kept low as it is not in our interest for this to be followed strictly. A value of 0.01 has been set for the same, and a decision has been made that this would be kept constant for though all the problems.

Target profile equations

The target profile equation has been created in such a way that it can have quadratic and sine squared terms in both the directions. The coefficients for each of the quadratic terms are integer values between -5 and 5. The frequency in the sine squared terms are set to be π times an integer between 1 and 5, while the phase angles are set to be π divided by random numbers between 1 and 6.

The following are some examples of target profile equations generated:

- "1*x2*x2+-1*x2+sin(2*pi*x1+pi/5)*sin(2*pi*x1+pi/5)"
- "0+sin(3*pi*x1+pi/6)*sin(3*pi*x1+pi/6)
  +sin(1*pi*x2+pi/1)*sin(1*pi*x2+pi/1)"
- "2*x2*x2+1*x2"

Bounds

The domain lower bound in all of the problems in the original set of problems were -10^{20}, which signifies negative infinity, which shall be used in the problems generated here as well. The rest of the bound values are chosen based on the target profile equation; by generating the desired domain profile by solving the target profile equation for the domain size. The maximum, minimum and median values in the desired domain are used.

The following are the specifications of the Bounds

- *Lower bound for domain* is set to -10^{20} for all problems.
- *Upper bound for domain* is a uniformly sampled random number between the median and the maximum values.
- *Lower bound for boundary* is the minimum value plus half of uniform random value generated between positive and negative difference between maximum and minimum values.
- *Upper bound for boundary* is the maximum value plus half of uniform random value generated between positive and negative difference between maximum and minimum values.

Sourcing term

It has been decided that the sourcing term shall be sampled from the set $\{0, -10, -20, -30, -40, -50\}$.

Additional filters

Once the problem is generated, there are two additional set of filters based on certain threshold that were incorporated to improve the quality of the equations generated. The first one is a threshold for the maximum and minimum values observed in the domain; a difference of less than 0.3 is discarded. The second one is a threshold on the cost that IPOPT predicts for the generated problem; a problem with a cost of more than 0.2 per cell is discarded.

4.4 Initial guess method

An initial guess method is a single-step method that takes the parameters of the problem as inputs and come up with a set of reasonable boundary values from which further optimization can be performed. An informed initial guess as close as possible to the best possible solution would enable us to have fewer iterations to get to the final solution. Theoretically, it is even possible that one could arrive at a method that could provide an answer that would not require further optimization.

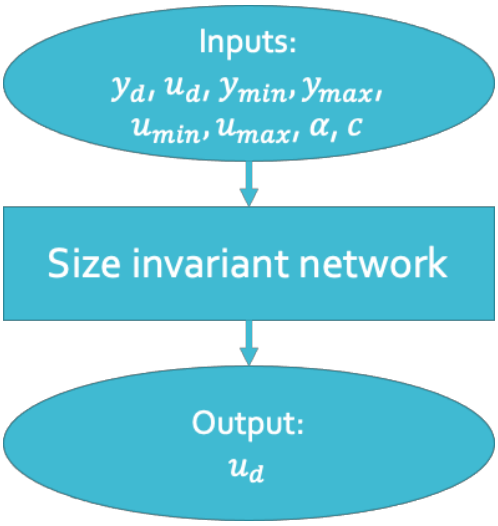


Figure 4.2: Initial guess network from a high-level

Since a problem would include desired profile of the domain, which is represented in a two dimensional array, and, the size of this two dimensional array changes from problem to problem, a deep learning method designed to tackle this should be able to work with spatial data and should be input and output size invariant. Convolution layers are capable of working with such input and output constraints. Variants of convolution layers, like graph convolution layer may also work.

Now, it is important to define a criterion that need to be satisfied in order to consider a method arrived at good enough to be an initial guess method. The following may be used for this purpose.

A method for informed guess can be considered good if the cost for the guessed boundary value is lower than the cost that would be encountered when the boundary values assume the mean of the desired domain values, the median of the desired domain values, and, the values at the edges of the desired domain values, in more than 50% of the cases; and the mean cost over all the problems should be lower than

all others.

It is to be noted that although the aforementioned criterion was arrived after a lot of thought and consideration, it should be treated as a suggestion and not a law.

4.5 Optimizer method

An optimizer method is multi-step method that takes the current boundary values and information related to the cost as input, and outputs updated boundary values. Ideally, it is desirable that the optimizer to converge to the best possible values with the fewest number of iterations.

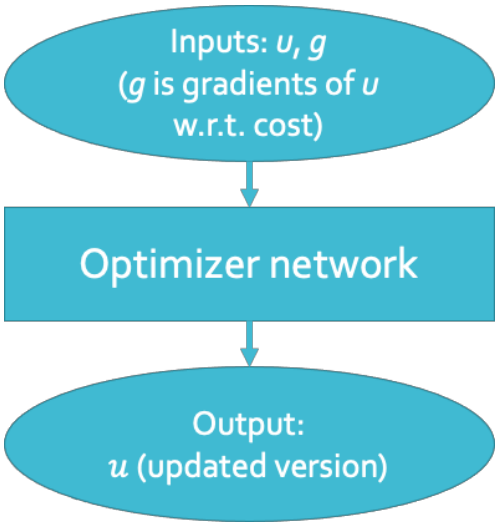


Figure 4.3: Optimizer network from a high-level

Since the optimizer method would take a set of boundary values and the corresponding cost information as input and attempt to iteratively improve it, it is desirable that a deep learning method designed to tackle this should be able to take both spatial and temporal information into consideration. To be precise, the output provided by the optimizer may effectively take the current and previous sets of boundary values and costs into account. Recurrent layers in combination with convolution layers would be capable of working with it. There are several strategies to implement spatio-temporal neural networks that could be used as inspirations.

The optimizer network shall be trained using existing methods used in the industry to train neural networks. Training the optimizer can be considered as policy improvement, and the whole process could be considered as a

version of policy gradient method. The reward would be the negative of the cost function, and the objective would be to maximize this reward.

The optimizer method shall be compared with an existing solver in the industry, named IPOPT.

Chapter 5

Experiments, Results and Analysis

This chapter contains information regarding the hardware and software specifications used, setting up the baselines and performing the experiments, and, the results and analysis. All the experimental code and associated results are accessible via our GitHub repository[46].

5.1 Experimental setup

5.1.1 Hardware and software setup

Hardware

The specification for the device used in training and validating all the neural network is provided in Table 5.1. Testing the models were done using several regular nodes in the university's ICS cluster with specifications listed in Table 5.2, which were used such that different tests were divided and executed on different nodes to get the results faster. The specification for the device used for performance evaluation is listed in Table 5.3, which is an old computer in which credentials for user with administrator privileges is available, and, performance analyzing tool named 'perf' runs smoothly.

Component	Specification
Hardware model	MacBook Pro 2019, 16-inch
Processor	2.6 GHz Intel Core i7
No. of cores	6
Graphics	AMD Radeon Pro 5300M 4 GB, Intel UHD Graphics 630 1536 MB
Memory	16 GB DDR4 @ 2667 MHz
OS	macOS 13.3.1

Table 5.1: Specification for the machine used for training, testing and validating neural networks

Component	Specification
Processor	Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
No. of cores	20
Memory	64GB DDR4 @ 2133MHz
OS	CentOS Linux 8 (Core)

Table 5.2: Specification for the nodes used from the HPC Cluster

Component	Specification
Hardware model	Lenovo ideapad 330-15ARR
Processor	AMD Ryzen 5 2500U
No. of cores	8
Graphics	AMD Radeon vega 8 graphics
Memory	8 GB (2× 4 GB DDR4 @ 2400 MHz)
OS	Ubuntu 22.04.02 LTS

Table 5.3: Specification of the machine used for performance evaluation

Software

The information of the software and libraries used in this project are listed in Table 5.4.

Software/library	Version
Python[47]	3.9.16
PyTorch[45]	2.0.0
Pandas[48]	1.3.5
Anaconda[49]	4.13.0
IPOPT[32]	3.14.12
MATLAB[50]	R2022b

Table 5.4: Versions of different software and libraries used

Note that the experiments were conducted in Python using PyTorch and other libraries, while much of the analysis were done in MATLAB. This not only helps in clearly separating the code used for the proposed method and analysis, but also that the default colours used in MATLAB for plots is aesthetically pleasing.

5.1.2 Setting up baselines

In order to set up a point of reference to compare and assess the effectiveness and accuracy of the method proposed, certain baselines need to be set up to

evaluate against. These baselines would also help in making decisions while conducting incremental experiments, and, would effectively act as a goal or a target that needs to be achieved.

Two different sets of baselines are created; one for the initial guess methods to target and the other for the optimizer method (coupled with initial guess methods) to target. Note that the second baseline is the baseline that determines how the entire method performs.

Baselines for initial guess

When given a problem, one can start with a set of boundary values and calculate for it's cost. This idea can be used to generate three different baselines by defining what what boundary values to start with. These are listed below.

1. Boundary values set to the mean of the desired domain values.
2. Boundary values set to the median of the desired domain values.
3. Boundary values set to the values at the edges of the desired domain values.

This is done for all 10000 problems that were generated, and, the information about the cost is stored in a CSV-file. This makes it easy to perform analysis later. However, in the decision-making and analysis, it might make more sense to use only the problems for which IPOPT found a feasible solution; the total number of problems for which IPOPT found a feasible solution is 5907.

Baselines for optimizer method

There are two sets of baselines used to determine how good the optimizer method created is.

1. *Comparison against solvers previously tried and tested to work for boundary control problems*.

   The state-of-the-art large-scale nonlinear optimization problem solver, IPOPT, shall be used to create this baseline. In IPOPT's codebase, there exist a few examples that shows how to solve boundary control problem.

   There are one set of CPP-file and HPP-file per example problems. These files contain information about the problem being solved, and it invokes the IPOPT engine from within this file. Although it is possible to create as many pairs of files to solve for the different generated problems, it is not a very efficient idea to do so.

Therefore, one single pair of CPP-file and HPP-file were created which can take all of the variations in a problem, like domain size, alpha, upper and lower bounds for domain and boundary, sourcing term, and, target profile equation, as command-line arguments. Handling every argument in this list is quite straightforward except for the target profile equation, the equation that is used to generate the desired values for the domain. The equation is in string format, and it needs to be evaluated at the C++ side. This was achieved using a mathematical expression parser written by Hamid Soltani[51], which is a modified version of a mathematical expression parser presented in the book written by Herbert Schildt titled *C++: The complete reference*[52].

2. *Comparison against optimizers used in deep learning*.

   Optimizers that are extensively used in deep learning like stochastic gradient descent (SGD) and adaptive moment estimation (Adam) are employed for this. The optimizer parameters are set in such a way that it can achieve as much overfitting as possible. A single-layer neural network with no inputs is used, which is essentially a layer with only the bias terms. The cost is calculated and hence the gradients, which is used to backpropagate; this is done several times until it converges to the optimal boundary values, or, it exceeds some maximum number of iterations.

If the proposed method beat both the baselines, then it would mean that people should consider switching to our method. If the method only beats the latter baseline, then, it would indicate that with more time and effort put into researching in this direction, one could potentially come up with better deep and reinforcement learning based methods that may enable the creation of a better solver.

It is to be noted that the former baseline method is computationally less expensive and hence it is easy to run and get results for all 10000 problems. However, the latter is computationally expensive and it needs to be run for several number of iterations before getting some meaningful results. This baseline shall only contain information for 400 problems, which are obtained after running for 100 optimizer steps.

5.1.3 Performing the experiments

Initial guess method

A neural network is designed based on intuition, with certain features being engineered into the network. As discussed in Section 4.4, the network is to take desired profile for the domain, upper and lower bounds for both domain and boundaries, and, sourcing term as inputs, and the network output would be the guessed boundary values.

The generated 10000 problems are divided into three; 80% of the problems are used for training, 10% of the problems are used for validation and 10% of the problems are used in testing. The best model for a given network design is chosen based on the lowest validation cost, while the overall best network design is chosen based on the lowest testing cost.

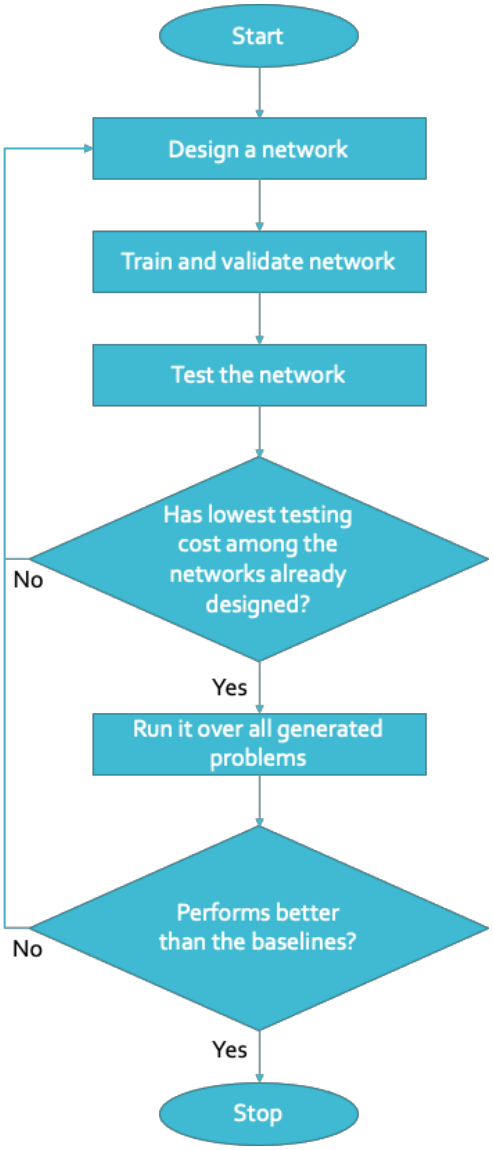


Figure 5.1: Experimental setup for initial guess network

The idea is to iteratively design, train, validate and test different networks until a network that performs better than the baselines is obtained. This

process is represented in Figure 5.1.

While designing the network, one must ensure that the input and output dimensions are variable, and, the output dimension is a function of input dimensions. If the target profile array is of size $N \times N$, then the output boundary value size would be $4 \times N$. It is intuitive that one could use convolution, graph convolution and different pooling layers in the network. Indeed, there may be other layers that could work with this constraint. A fully connected layer, a.k.a. linear layer, can not work with such a constraint as it requires the input and output dimensions to be known beforehand to work.

Optimizer method

An optimizer network can be arrived at by taking inspiration from several other existing and widely used optimizers; perhaps even incorporating the features of one or more optimization algorithms into the network being designed. As discussed in Section 4.5, the input to the optimizer network would be the boundary values calculated at the previous iteration and its gradient calculated with respect to the cost function, and, the output from the network would be the boundary values for this step.

Similar to that for the initial guess method, the generated set of problems is divided in the ratio 80:10:10 for training, validation and testing. The best model is selected in a similar manner. However, since this method takes longer to run, iterating over multiple network designs would be a tedious task. Therefore, only a few designs would be tried and tested, and this shall be done on an intuition basis.

Again similar to that for the initial guess method, one must understand that the input and output dimensions are related. The input would be the boundary values and their gradients, which are both of size $4 \times N$, and the output is the updated boundary values, which are of size $4 \times N$ as well.

Unlike the initial guess method that only deals with spatial information, optimizer network needs to deal with both spatial and temporal information. Hence, layers like convolution, graph convolution, recurrent based (like LSTM and GRU) and other layers that can work with spatial and temporal information shall be employed.

5.2 Results

By following the aforementioned procedure, the following are the network architecture and information related to their accuracy. Additionally, information on choosing to follow certain procedures that may seem unconventional are also discussed.

5.2.1 Initial guess method

Network architecture

It is possible to create a wide variety of networks that could be used for initial guesses. In fact, quite a few networks were designed and evaluated before arriving at the one discussed here. Figure 5.2 is the architecture that was arrived at after several preliminary experiments. However, it can not be stated that this is the best network that one can ever come up with; it's definitely far from it.

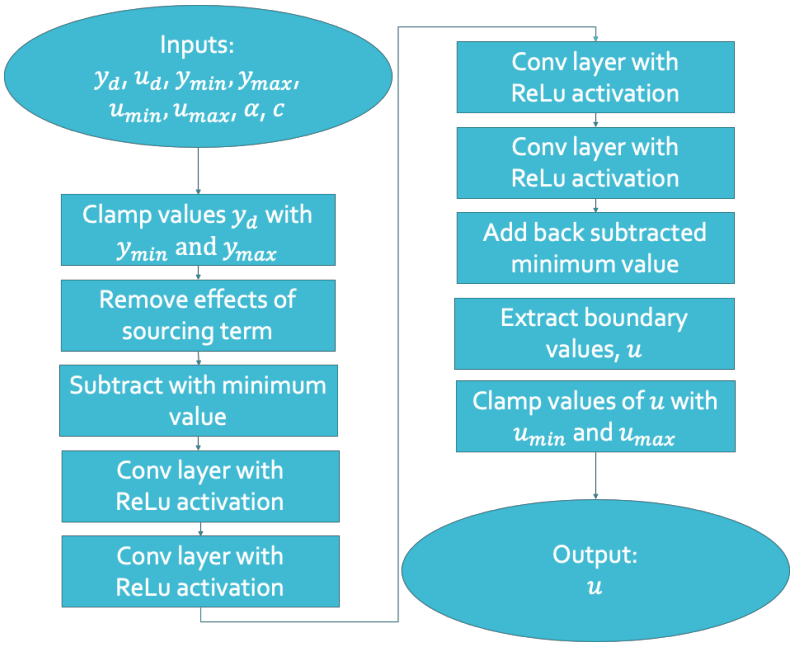


Figure 5.2: Initial guess network architecture

The network is designed to take the desired values for the domain, upper and lower bounds of the domain and boundaries, and, sourcing term as input. Starting with the desired values for the domain, the values are clamped to ensure that they fall within the upper and lower bounds of the domain. Next, the effects of the sourcing term are subtracted from the values. The details of this step are discussed in Section 5.2.1. Following this, in order to ensure that all the values in the array are greater than or equal to zero, the minimum value in the array is determined, and all the elements in the array are subtracted with this minimum value. The array is then passed through four convolution layers with rectified linear unit activations. Following this, the minimum value that was previously subtracted away is added back to the output. The values at the boundaries of this array are extracted and are

clamped to ensure that they fall within the upper and lower bounds of the boundary.

Removing the effect of the sourcing term

Upon conducting simple numerical analyses to see how the sourcing term in Laplace's equation affects the solution for different domain sizes and boundary values, two key relationships can be found.

- **Relationship between solutions for different sourcing term values**: If all the boundary values are set to zero, then it has been observed that for a given domain size, the numerical solution to Poisson's equation for different values follows a pattern, i.e. the solution for sourcing term assuming a value of -20 is twice that of when sourcing term assumes a value of -10. Similarly, the solution for -30, -40 and -50 are respectively three, four and five times that of -10. This has been experimentally verified for different domain sizes.
- **Relationship between solutions for random and zero boundary values**: For a fixed domain size, if A is the numerical solution for a given sourcing term with boundary values set to zero, B is the numerical solution for random boundary values with the sourcing term set to zero, and, C is the numerical solution for the same random boundary values and the given sourcing term, then it can be observed that,

  $$A + B = C \tag{5.1}$$

  This has been experimentally verified for different values of domain sizes and boundary values.

Putting these relationships together; using zero boundary values of dimensions $4 \times N$ and a constant sourcing term of -10, after forward solving for Poisson's equation, a matrix of size $N \times N$ is obtained as output. Such matrices can be created and stored for different values of N so as to reuse them. To get the corresponding value for other values of the sourcing term, simply multiplying the values in the stored matrix for the desired size with the ratio of the required sourcing term and -10 would suffice. This matrix can be subtracted from the solution to Poisson's equation with a constant sourcing term to get the solution to Poisson's equation without a sourcing term!

Quantitative results and preliminary analysis

A comparison is made between the results for the baselines and the best initial guess network filtered for the problems for which IPOPT found feasible solutions, and the summary of the same is made available in Table 5.5.

	Mean	Median	Edge	Network
Mean cost	4.5059	5.6269	87.1563	1.0591
Median cost	0.4422	0.4572	0.2010	0.2709
Lowest cost	0.0087	0.0091	0.000023215	0.0081
Highest cost	388.7509	1642.7	3478.2	125.8895

Table 5.5: Cost results for initial guess method as compared to the baselines

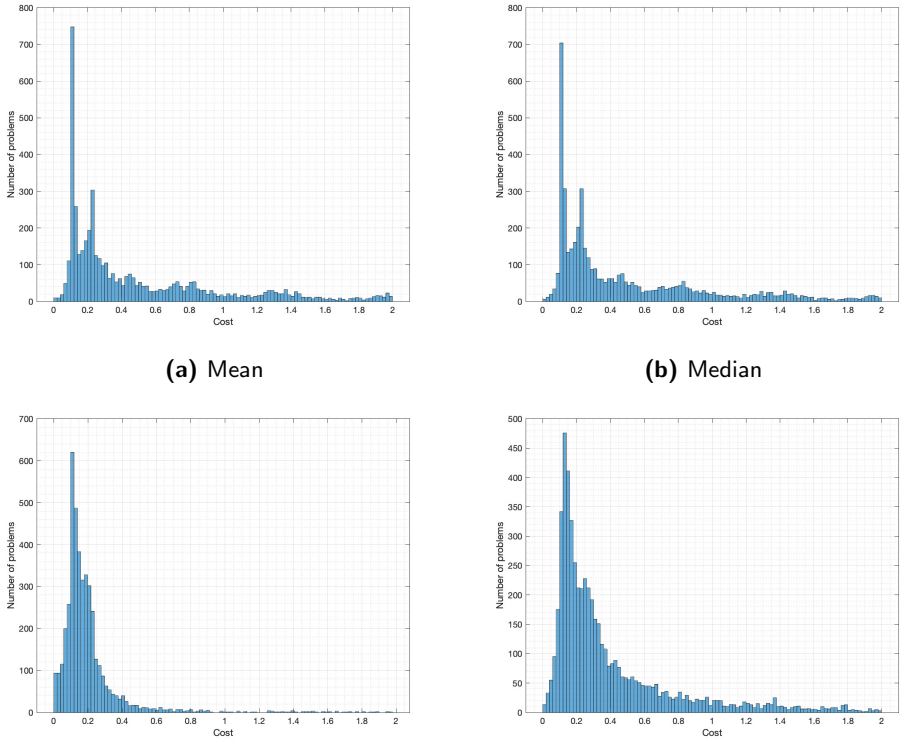
Taking the mean cost for all the problems for which IPOPT found optimal solutions, it is observed that when using the mean values of the desired domain values as the boundaries, was found to be the lowest among the baselines, with a value of 4.5059. The corresponding values for median as boundary values is 5.6269 and that for the edges of the desired domain values as boundary values is 87.1563. The same for the initial guess network is 1.0591. From this, it might be tempting to state that the edge values might be the worst possible values to use, and our network is significantly better. However, that is not necessarily the case, as the high penalty for boundary violation could have contributed to the same.

Taking the median cost for all the problems for which IPOPT found optimal solutions, it is observed that the mean, median and edge values as the boundaries would have a median cost of 0.4422, 0.4572 and 0.2010 respectively, while that for the network is 0.2709. In this scenario, it appears to be that the edge values seem to be the best possible values to use, in stark contrast to what the mean of all costs suggested! Our network gets the second spot in this.

Similar trends can be observed when you take the highest and lowest cost reported by the methods. Among the highest costs, the network seems to have the lowest value, whereas when it comes to the lowest cost, the network only has the second lowest value; the edge seems to beat the network here as well.

Figure 5.3 shows histograms for each of the baselines and our network wherein the number of problems that had cost within a certain interval is shown. To focus on the region with lower costs, the plots have been truncated at a cost value of 2. From these plots, it can be observed that all four methods have decently low costs for a lot of the problems. Additionally, it can be seen that the values corresponding to edge seem to have a few problems with significantly low costs (quite close to zero), which is much more than the others.

Perhaps a better way to visualize and compare would be by using a cumulative count as it is meaningful to superimpose the plots in this setting. A cumulative histogram would show the count of problems that have cost less than or equal to the value at any given point in the plot. Figure 5.4 shows



(a) Mean **(b)** Median

(c) Edge **(d)** Network

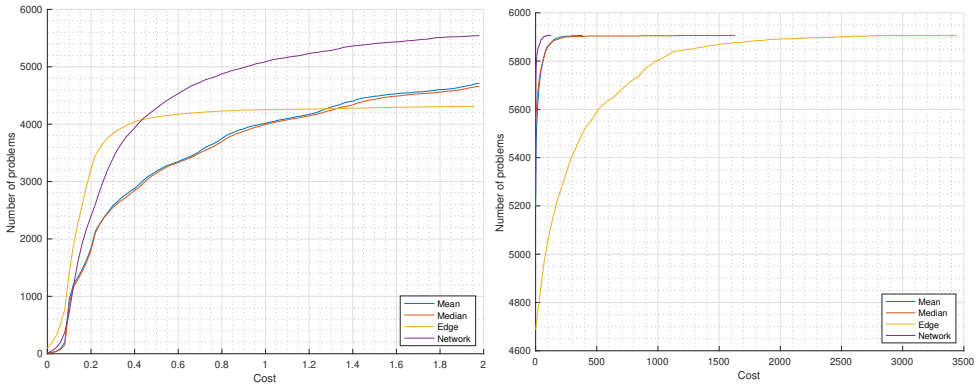
Figure 5.3: Counts represented in histograms for mean, median, edge and network

the same. Line plots are used instead of bars to make it easier to visualize information from multiple baselines and the proposed method.

It can be observed that there are about 4000 problems for which using edge values may be more advantageous than the other three. However, at the cost value mark of about 0.42, the tide changes; the network appears to have an upper hand from that point forward. From the sub-plot for larger cost values, it is evident that the edge method performs significantly worse than the other three methods at later points as it goes slowly into the regions with more and more cost.

Furthermore, if a head-to-head comparison between the cost for the network and the baselines is performed, it can be observed that the network beats mean, median and edge costs, respectively, 4152, 4201 and 2301 times out of the total 5907, which are about 70.29%, 71.12% and 38.95% of the cases.

Using the results at hand, it is hard to conclusively state that the method has beaten all the baselines. However, it has clearly beaten the baselines set by the means and medians of the desired profiles. The baseline set by the edges



(a) Trimmed till cost value of 2

(b) Larger cost values

Figure 5.4: Initial guess cumulative count

of the desired profiles is quite peculiar in that it is both the most accurate and the least accurate at the same time depending on how you look at it.

However, an argument can be made that using the edge values as a starting point is not very reliable as it could be the best and the worst at the same time; therefore, it is advisable to use the initial guess network, which has more consistency.

5.2.2 Optimizer method

Network architecture

It is possible to create a wide variety of networks that could be used for the optimizer method as well; Figure 5.5 shows the best network arrived at so far, but is unlikely to be the absolute best network possible in terms for both accuracy and computational cost.

The method is designed to take the boundary values and their gradient calculated using the cost function as input. The method encapsulates the Adam optimizer, RMSProp optimizer and a spatio-temporal network. The gradient is fed into these three as input, and each of these methods outputs some intermediary values, which are multiplied with three separate learning rates, and then subtracted from boundaries to get the updated boundaries.

The input to the spatio-temporal is feature engineered to be the concatenation of the gradients, the squared of the gradients and the cubed of the gradients. A temporal convolution layer has been manually defined, which is internally a convolution layer followed by LSTM. The inputs are passed through a couple of temporal convolution layers with rectified linear unit activations and finally through a convolution layer with rectified linear unit activation as well. This intermediary output is multiplied with the sign of the original

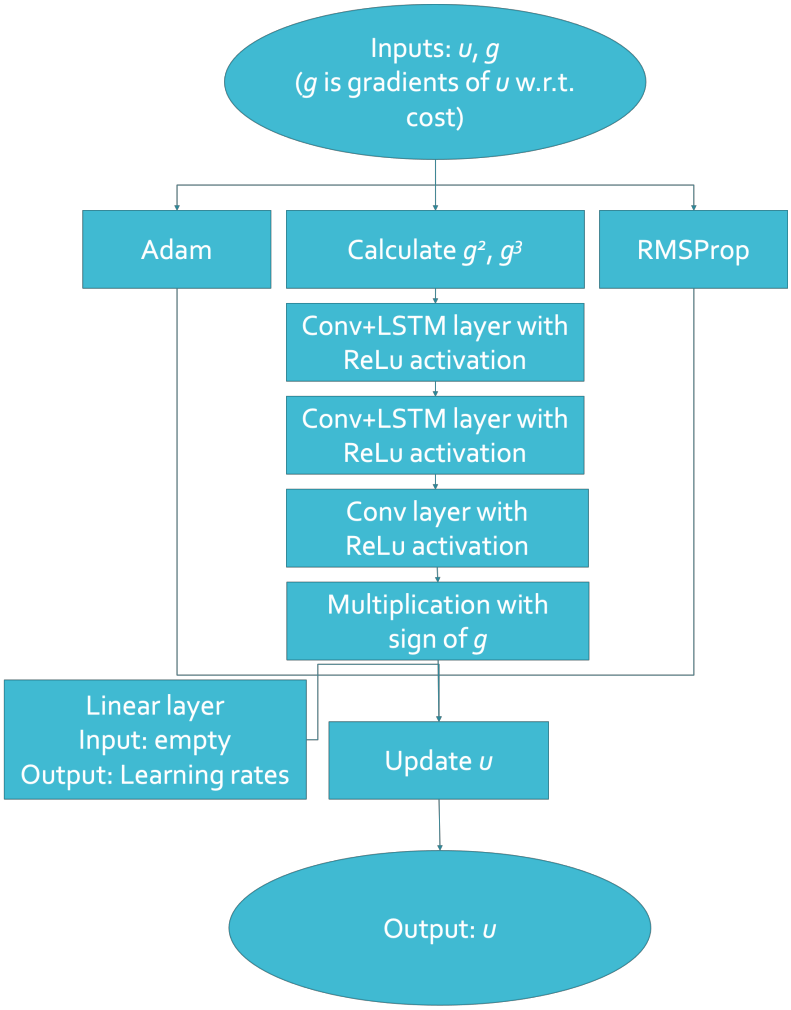


Figure 5.5: Optimizer network architecture

gradient values, as there is no way for rectified linear unit to output negative values.

It should also be noted that the learning rates used are trainable, and hence, the entire method can be considered as a giant network with some parts of it feature-engineered in accordance with Adam and RMSProp.

Why include Adam and RMSProp

There are two main reasons why the decision to include Adam and RMSProp along with the spatio-temporal network to create a grander optimization method.

1. Adam and RMSProp have been widely used in deep learning as opti-

mization algorithms to tune network weights. They have been demonstrated to work well in quite a lot of scenarios.

2. Having Adam and RMSProp appears to be guiding the spatio-temporal network to train. If they were not used, during the evaluation, the best values observed would be the first set of values (the output of the initial guess method), and hence, the network iterations would not be considered during backpropagation.

Indeed, the second issue could have been addressed by tweaking the training method. It is also true that either one of Adam or RMSProp could have been used for this and not both simultaneously. Furthermore, there are several other optimizers that are being used in research like Adaptive Gradient Algorithm (Adagrad), AdaDelta, Stochastic Gradient Descend (SGD), etc. This is just the decision that was made.

Quantitative results and preliminary analysis

Let us have a look at how the method fare against IPOPT and the baselines created by SGD and Adam with the 400 problems. Within this range, IPOPT has found feasible solutions for 252 problems, which shall be the ones that are used as baselines here. The summary of the results is made available in Table 5.6.

The optimizer network was originally trained for 8 iterations of optimizer steps. Therefore, it was initially tested with 8 iterations of optimizer steps as well. During this initial test, it was observed that IPOPT was doing way better than all of the others in all of the cost comparison matrices. The proposed network seemed to be doing about as good as Adam; a little better in terms of the mean, median and lowest cost, at the same time a little worse when you look at the highest cost. The network clearly performs better than SGD.

Nevertheless, it should be noted that the proposed optimizer network has only done 8 iterations in total, which is much smaller than the number of iterations done by IPOPT, SGD and Adam which are, respectively, 43, 100 and 100. At this point, it can be speculated that if the network had run for more iterations, it might as well have had a lower cost than what is reported here. Therefore, a decision was made to run the optimizer network again for 32 iterations in total.

Now, it can be observed that the network blows SGD and Adam out of the water in mean, median and lowest cost values. However, the highest cost observed in the network is a bit higher than that of Adam. Nonetheless, it looks like the network seems to be good enough to compete with IPOPT, as the mean cost is about equal, and the median and lowest costs are a bit lower than that of IPOPT.

	IPOPT	SGD	Adam	Network	
Mean cost	0.1223	8378.6	0.1808	0.1721	0.1223
Median cost	0.1252	0.3439	0.1672	0.1469	0.1181
Lowest cost	0.0097	0.0093	0.0164	0.0100	0.0034
Highest cost	0.1995	126160	0.5770	1.0894	0.6616
No. of iterations	43*	100	100	8	32

* The number of iterations for IPOPT is the rounded calculated mean value

Table 5.6: Cost results for optimizer method as compared to the baselines

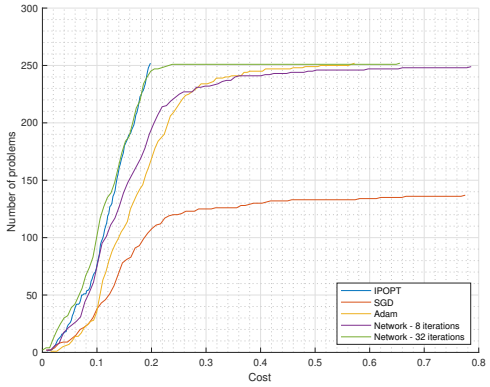


Figure 5.6: Optimizer network cumulative count

At this point, it seems like it might be beneficial to have a look at a cumulative histogram plot for the number of problems and the cost values; it would show the count of problems that have cost values less than or equal to the value at any given point in the plot. Figure 5.6 shows the same. Line plots are used instead of bars to make it easier to visualize information from multiple baselines and the proposed method.

Looking at the plot, it can be observed that there exist a very fierce competition between IPOPT and the optimizer network running 32 iterations; both of these intersect quite a few times. It can also be observed that the optimizer network running 8 iterations also seem to be performing well to a great degree. This indicates that a lot of the problems may have reached optimal values much before 32 iterations.

When a head-to-head comparison is done between the cost for the optimizer network and the baselines, it can be observed that the network beats IPOPT, SGD and Adam-based baselines 127, 226 and 244 times, respectively, out of 252 cases. This means that the network beats the respective baselines 50.40%, 89.68% and 96.83% of the times.

This indicates that the method arrived at is indeed quite good.

5.3 Detailed analysis

Now, it is time to perform a detailed analysis wherein the entire set of 10000 problems are used. The method shall be exclusively compared against IPOPT in this section.

5.3.1 Simple statistical analysis

As already discussed, out of the 10000 problems, IPOPT has been able to find feasible solutions in 5907 cases. In these cases, it can be observed that the cost predicted by the method is lower than that by IPOPT in 3011 cases, which is 50.97% of the cases.

However, this should be taken with a pinch of salt as a slightly more relaxed barrier function was used in the cost calculation. The maximum absolute constraint violation reported for a particular cell in the domain or boundary is 0.1033, whereas the mean absolute constraint violation is 8.3×10^{-4}, which may be regarded as quite small.

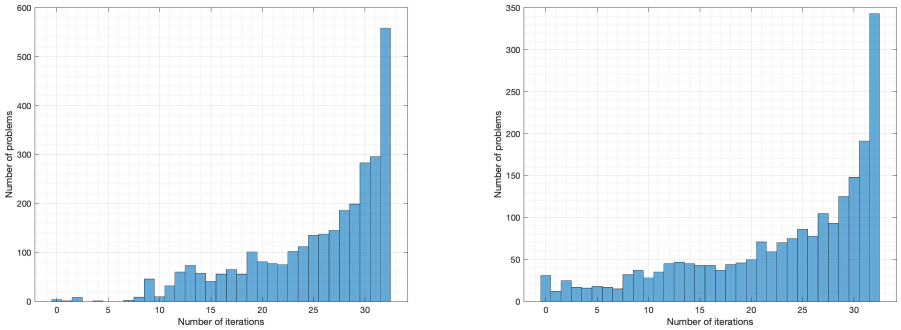
The total number of cases wherein the method was able to predict a feasible solution without any constraint violation was 2127, which is 36.01% of the cases. It can be speculated that this percentage could potentially go up if stricter constraint violation penalty factors were used in the cost calculation.

5.3.2 Iteration at which the network finds the best solution

Although the optimizer network has been running for a fixed 32 iteration, it should be noted that the method may have achieved its best possible results at an earlier iteration. Let us attempt to visualize at which iteration this happens with and without constraint violations for the problems where the network was able to achieve lower cost without any constraint violations, although the iteration at which the cost goes lower with some constraint violation is discussed. This has been done to make the visualization more meaningful.

Histograms depicting the same have been shown in Figure 5.7. In both cases, it can be observed that there exist a general trend that for a lot of the problems, the optimal values were found at later iterations. The median number of iterations to get the lowest cost is calculated to be 28 and 26 for the lowest cost and the lowest cost without constraint violation, respectively. Overall, this trend suggests that the optimizer is good at doing its job; it is able to iteratively reduce the cost.

Furthermore, the tremendous increase in the bar size at iteration number 32 suggests that for a lot of these problems, more optimal solutions could have been found if it had run for more iterations.

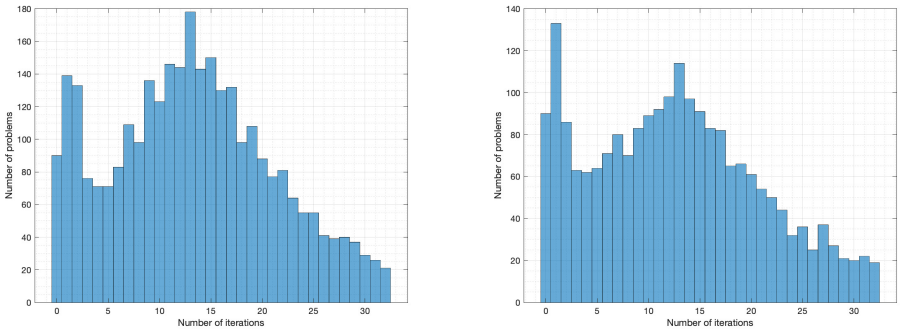


(a) With/without constraint violation **(b)** Without constraint violation

Figure 5.7: Histograms depicting the number of problems for which the method achieved the lowest cost at which iteration

5.3.3 Iteration at which the network beats IPOPT

Similar to the previous analysis, it should be interesting to have a look at the first iteration at which the network reported a lower cost than that of IPOPT. Additionally, it would be interesting to observe when are the first such instances where there are no constraint violation costs. This shall be only done on the problems where the network was able to achieve lower cost without any constraint violations. Even when talking about lowest cost obtained with constraint violation, the set of problems for which the method achieved zero constraint violation at some point shall be used.



(a) With/without constraint violation **(b)** Without constraint violation

Figure 5.8: Histograms depicting the number of problems for which the method first surpassed IPOPT's cost at which iteration

Histograms depicting this are shown in Figure 5.8. Two peaks can be observed in both of these histogram plots; one at the beginning, and, the other

right around the middle. The peak at the beginning, indeed, indicates the contribution of the initial guess network to the overall quality of the results, while that at the later point indicates the prowess of the optimizer network. The median number of iterations to beat IPOPT's cost value is calculated to be 13 and 12 iterations respectively for the case where there can be some constraint violation and the case where there can absolutely be no constraint violation.

5.3.4 Contribution of Adam, RMSProp and spatio-temporal parts

The learning rates for the difference calculated by Adam, RMSProp and the Spatio-temporal part of the optimizer are respectively 0.0223, 0.0221 and 0.0645, which means that the learning rates for the network is about three times as high as the learning rates assigned to Adam and RMSProp. Note that the learning rates where not assigned manually; rather these were learnt through backpropagation.

However, just because the learning rate is higher does not mean that the contribution of Spatio-temporal part would be higher during the update. This is because it is entirely possible that the differences calculated by the network could be lower than that calculated by Adam and/or RMSProp and hence there is a possibility that the contribution could be smaller.

The optimizer network was run over one hundred randomly chosen problems for 32 iterations. As the value to be updated by the network are the boundary values and it affectively has a dimension of $4 \times N$, in order to make it easier to compare, the contributions by each of the three terms are taken as the mean of absolute values of each of these $4 \times N$ elements. The average such contribution per optimizer iteration for Adam, RMSProp and the Spatio-temporal part are respectively 0.0151, 0.0133 and 0.0251. Clearly, the update contribution from the network is higher; almost as much as the sum of the other two.

This contribution information is collected for individual optimizer iterations along with the information on the size of the domain and the optimizer iteration at which this is recorded. Partial correlation between iteration number, domain size, contribution by Adam, contribution by RMSProp and contribution by the Spatio-temporal part was conducted.

There are some interesting negative correlation between the contributions of Adam and RMSProp with the number of iterations, with the partial correlation coefficient being -0.6017 and -0.3360. This indicates that the contribution by Adam and RMSProp reduces as the number of iterations increases, which is expected. Indeed, there is a negative correlation for the Spatio-temporal part with the number of iterations; however, the value is -0.0086, which is quite low. This is interesting to me as this may indicate that

the Spatio-temporal part may not slow down much after several iterations, which could indicate that it would not just converge to an optimum that it finds. However, this can not be stated with confidence only using the correlation information.

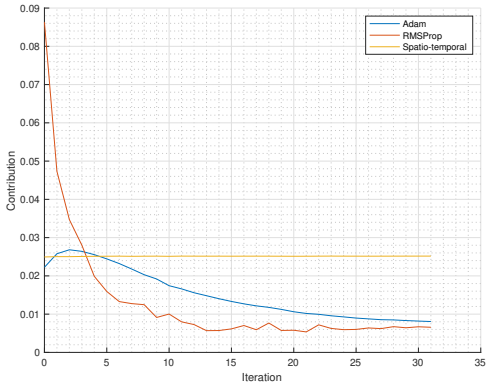


Figure 5.9: Comparing the contribution by Adam, RMSProp and Spatio-temporal part over different iteration

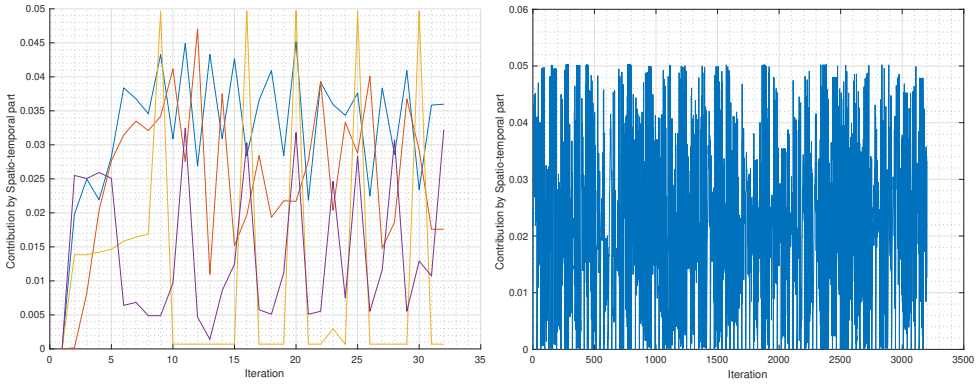
Figure 5.9 helps in visualizing this correlation. A heavy downward trend can be observed for the contribution by RMSProp over different iterations. The contribution by Adam does have a downward trend, however not as heavy as that of RMSProp. However, the trend by the Spatio-temporal part seems very peculiar; it does not seem to change at all. This is more evidence that the spatio-temporal part does not seem to be slowing down after several iterations. Perhaps this is something that requires more investigation.

Investigating Spatio-temporal part

This investigation is to know if the network is indeed doing something meaningful with the inputs to it, or whether it is just it's bias terms that is doing all the work here. If it is all about the bias terms, then for different inputs to the Spatio-temporal part would output more or less a constant value.

In the setup, a minor modification is made to store the value of the output of the Spatio-temporal part in the matrix form for the first iteration. Then, for every subsequent iteration, the output of the Spatio-temporal part at that iteration is subtracted with the stored value, following which the mean of the absolute values are calculated and logged.

Figure 5.10 helps in visualizing the information that was logged. From this, it can be observed that the contribution from the network is not constant. Rather, it does not follow any pattern. The variations observed are between



(a) For four different problems **(b)** For 3200 iterations from different problems

Figure 5.10: Contribution by Spatio-temporal part visualized

0 and 0.0503, whereas the mean value that was observed earlier is around 0.025, indicating that the variation is not an insignificant or meaningless fluctuation.

Although it is still quite interesting to see that the contribution by the Spatio-temporal network has a constant-ish mean absolute value, it does not mean in anyway that the network itself is giving us constant-ish value.

5.3.5 Comparative Analysis of initial guess network and edge values

Earlier, it was observed that that it is hard to conclude whether using the initial guess network or the edge values for the initial guess was more beneficial. Let us analyze the 3606 cases where using the edge values for initial guess were observed to be more beneficial and see if there is a pattern.

Since it is hard to manually have a look at each parameter and see if there is something that is contributing towards the accuracy, it seems logical to use some statistical method to help us identify the parameter that could be contributing to this.

One idea is to calculate partial correlation between the difference in cost between that of initial guess network and the edge values, and compare it with other variables related to the problem. These variables include the maximum, minimum, mean and median values in the desired profile for the domain, upper and lower bounds of the domain and boundary values, and, sourcing term of the PDE. Following the calculation, the observation is that the highest absolute partial correlation is with sourcing term, which is a value of 0.4593, which in itself is not a very significant value. However, the absolute partial correlation with every other variables are below 0.07, and

hence, it may be useful to attempt to observe how this information could be used to our advantage.

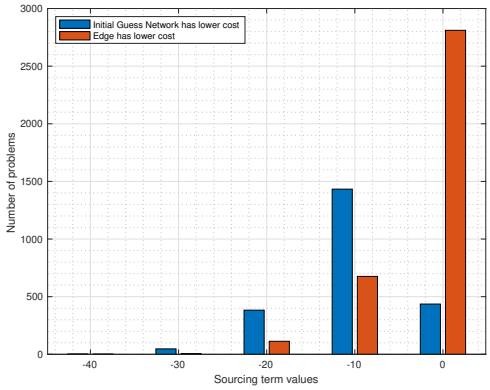


Figure 5.11: Number of problems for which initial guess network or edge values had lower cost for different sourcing term values

Let us attempt to visualize which of the two methods, the initial guess network values or the edge values, would win in a head-to-head comparison for different sourcing term values. The number of cases where one method has lower cost than the other and vice-versa can be compared easily. Figure 5.11 shows a bar graph helping visualize the same. From the graph, it can be observed that using the edge values might be more desirable when the sourcing value is 0. In every other cases, using the initial guess network would be more desirable.

This may indicate that one could come up with a better initial guess mechanism by simply adding a control statement that would let guess the initial values as the edge values if the sourcing term value is zero, and, would use the initial guess network in every other case. However, it is also likely that there exist some bias in the problem generator that has caused this.

5.3.6 Comparative Analysis of Optimizer Network and IPOPT

An analysis similar to the aforementioned was conducted to see if there are any patterns between the cases where IPOPT has better accuracy than the optimizer network. All absolute correlation coefficients after comparing with the same set of parameters were less than 0.07, and hence, it does not look like there is an easy explanation that could be given.

It may be that the results put forth by the optimizer network and IPOPT may be quite similar and the variation that is observed here is merely slight deviations. Let us make an attempt to directly compare the two methods.

A semi-transparent scatter plot with the cost by IPOPT in one axis and cost

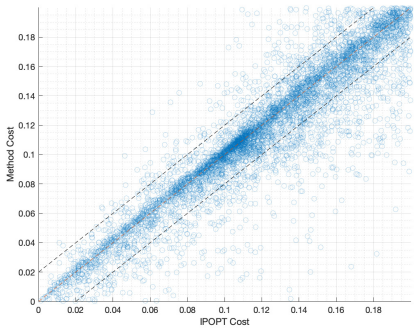
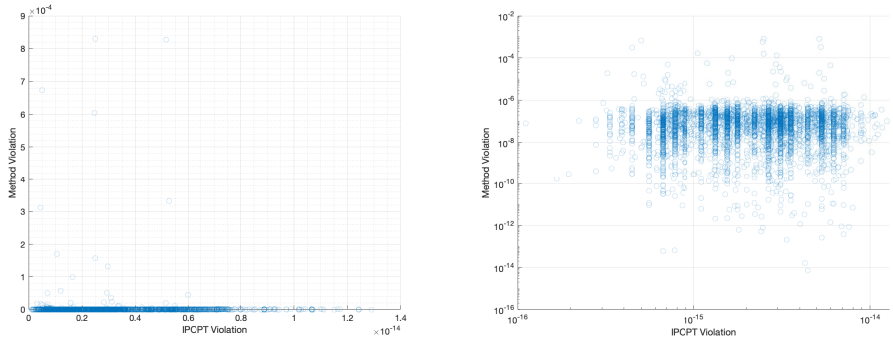


Figure 5.12: A scatter plot of cost by IPOPT and Optimizer network

by optimizer network in the other is shown in Figure 5.12. In the plot, a line of equality is drawn in orange color, which coincides with a lot of scatter plot points, indicating that the cost by both IPOPT and the optimizer network is quite similar. Two additional lines parallel to the line of equality has been constructed, which are at a distance of 0.0283 from each other and half that from the the line of equality. Note that these lines were constructed by adding X and Y intercepts of 0.02 to the line of equality. It can be observed that 4607 points are within these lines, which is about 77.99% of the points.

Now, let us make a similar scatter plot for the violations made by IPOPT and the optimizer network. Figure 5.13 shows the same. Two versions are shown,



(a) Standard axes **(b)** Log-log axes

Figure 5.13: Scatter plots of constraint violations by IPOPT and Optimizer network

one with data presented as is, and, the other one with the logarithmic axes. It can be observed that the violation is quite literally zero for a lot of the cases for the optimizer network, which is about 1581 cases, which, for obvious reasons, would not be visible in the logarithmic version of the plot. However,

there are a lot of cases wherein the violations made by the optimizer network is orders of magnitude larger than what is by IPOPT; much of the violation by IPOPT is in the range between 10^{-15} and 10^{-14}, whereas the same by the optimizer network is in the range between 10^{-8} and 10^{-6}. Indeed, the barrier functions used in the optimization task would have had a role in this results and changing the same and re-training the network may help in achieving smaller violations.

The differences between cost and violation could be visualized in a single plot as shown in Figure 5.14. Looking at this plot, it can be observed that quite a

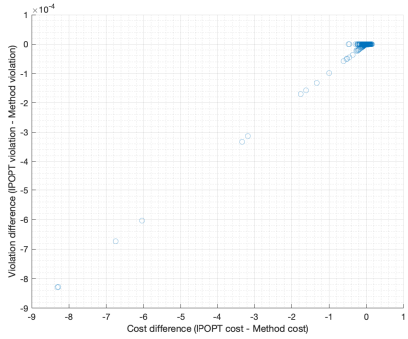


Figure 5.14: A scatter plot of cost and violation differences between IPOPT and optimizer network

lot of the instances that have close to zero cost difference and zero violation. Then, there are another group of points that appear to lie on a straight line, indicating that there is a linear relationship between the difference in violation with the difference in the cost, which in some sense, is expected as the violation in the boundary values do contribute towards the cost. A straight line passing through the origin would indicate that all the additional costs are contributed by such violations.

5.3.7 Performance evaluation and comparison

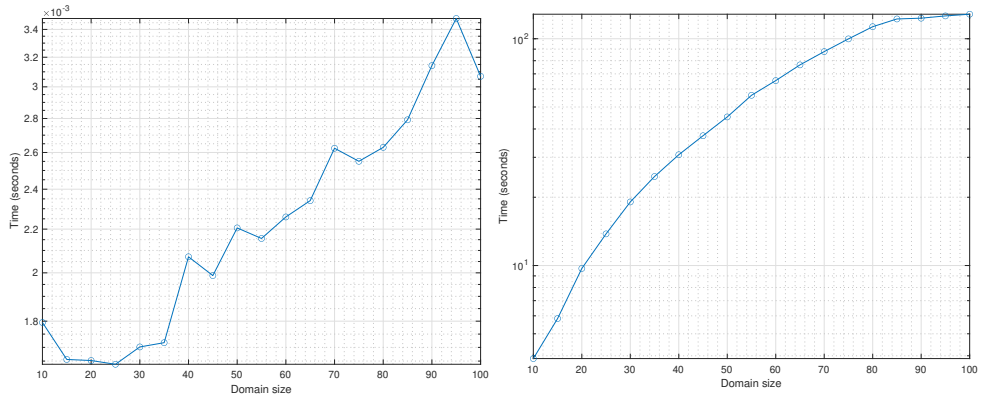
There are a couple of reasons for why it can be a bit tricky to compare the speed or performance of the proposed method with that of IPOPT. Primarily, both the proposed method and IPOPT take different number of iterations to solve different problems; on one hand, there are cases where the method achieves lower cost than IPOPT at very early iterations, while, on the other hand, there are cases where it does not achieve a lower cost than IPOPT even after the maximum number of iterations that the method has been tested for, which is 32. Secondarily, IPOPT is implemented in C++, while the proposed method is currently implemented in Python, which means that, based on

different analysis using different approaches to benchmark, the method could be 10 to a 100 times faster if it were implemented in C++ instead of Python.

In order to make a fair comparison between the two methods, it is important to take the aforementioned limitations seriously and device a way to compare the two by minimizing the affects of the same. The most logical way to do the same would be to compare the amount of time it takes for both the methods to finish 32 iterations for different domain sizes.

Timing

To begin with, let us have a look at the amount of time it takes to compute the initial guess network and optimizer network. This is done using Python's built-in module named 'time' and the results are summarized in Figure 5.15. It can be observed that the initial guess network takes in the order of 10^{-3}



(a) For initial guess network

(b) For optimizer network

Figure 5.15: Execution time for the initial guess network and 32 iterations of the optimizer network

seconds to execute which is negligible compared to the amount of time it takes to execute the optimizer network, which is in the order of 10^2 seconds. The trend of the optimizer network appears to be a lot smoother, which can be attributed to the fact that random fluctuations does not affect much for computations that run for a really long time.

The most straightforward way to compare the performance of two different methods would be to check the amount of time it takes to execute the two. However, in this case, it is more logical to compare the time taken by the proposed method and IPOPT along with some scaled values of the time taken by the proposed method with scaling factors of 0.1 and 0.01. This would provide a range for the amount of time it would take if the method was implemented optimally in C++ instead. This is shown in Figure 5.16.

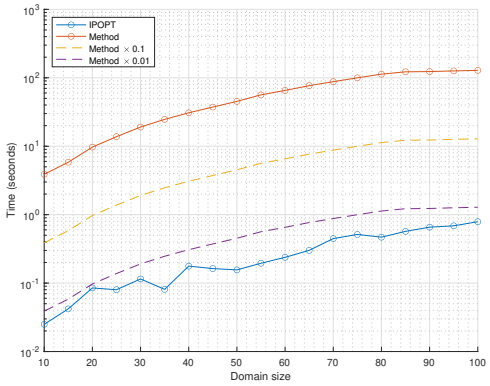


Figure 5.16: Comparing execution time for the proposed method with IPOPT

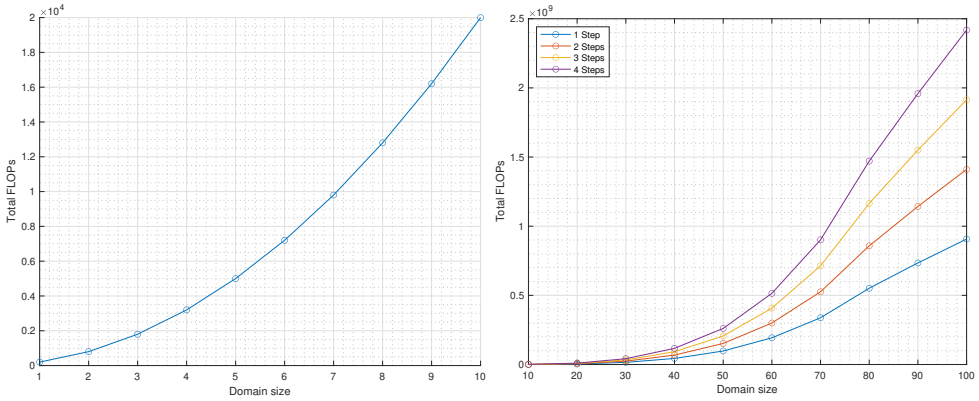
In the figure, it can be observed that the amount of time taken by the proposed method is significantly larger than the amount of time taken by IPOPT. Although it comes close, the scaled version of the time taken by the proposed method by a factor of 0.01 is still higher than the time taken by IPOPT. This indicates that, for the same number of iterations, the method would likely perform slower than IPOPT even if it were implemented in C++.

FLOPs

Perhaps a better analysis to compare the performance of the proposed method with that of IPOPT is to have a look at the number of floating point operations computed by the two methods. Ideally, this comparison would be independent of the programming language used or the machine settings. However, it should be noted that different computing hardware are designed to work better for different kinds of data structures or orientations, and hence, this would not give the full picture.

Firstly, let us have a look at the number of floating point operations involved in the computation for the initial guess network and the optimizer network. This is calculated using PyTorch's built-in profiler and the results are shown in Figure 5.17. A quadratic increase in the FLOPs is observed for increasing domain size for both the initial guess network and the optimizer network. However, it can be observed that the number of FLOPs for initial guess network is negligible in comparison to that for optimizer network; the former is in the order of 10^4 while the latter is in the order of 10^9.

Indeed, the number of floating point operations for different number of iterations is expected to be linear, which can be visualized better by plotting the number of FLOPs versus the number of iterations, which is shown in Figure 5.18. As it was observed that PyTorch's profiler is quite slow, and often crashes when attempted to work with larger iterations, it seems that



(a) For initial guess network **(b)** For optimizer network

Figure 5.17: FLOPs for the initial guess network and the optimizer network

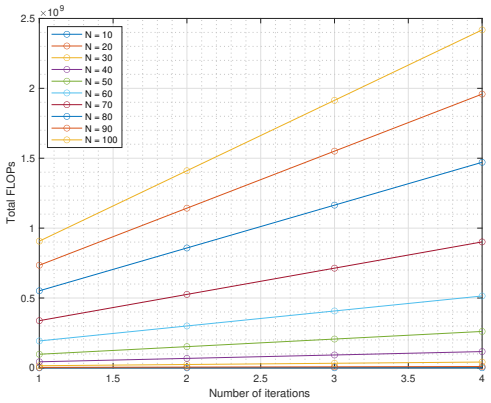


Figure 5.18: FLOPs versus iterations

the best way to calculate for 32 iterations would be to use this linearity to mathematically calculate the number of FLOPs for the same.

To calculate the number of floating point operations taken to solve problems in IPOPT, Linux's 'perf' tool was utilized. It was quite easy to set this up and it was able to provide with results pretty quickly for different problem sizes.

Let us compare the number of floating point operations for the network with that for IPOPT. Just like in the case for timing, this is done with both the proposed method and IPOPT running for 32 iterations for different domain sizes. Figure 5.19 shows the same. It can be observed that the proposed method has slightly more number of floating point operations than IPOPT for the same number of iterations, and hence it is expected to be slower if it is implemented optimally in C++ as is. This is quite consistent with what has been calculated for timing.

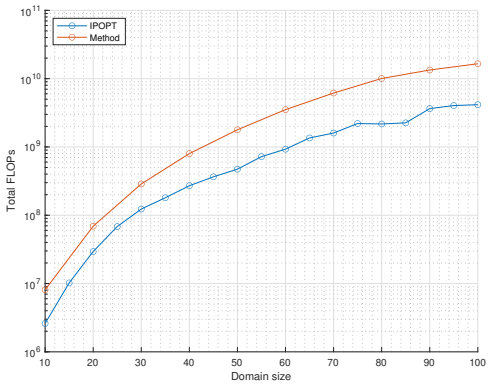


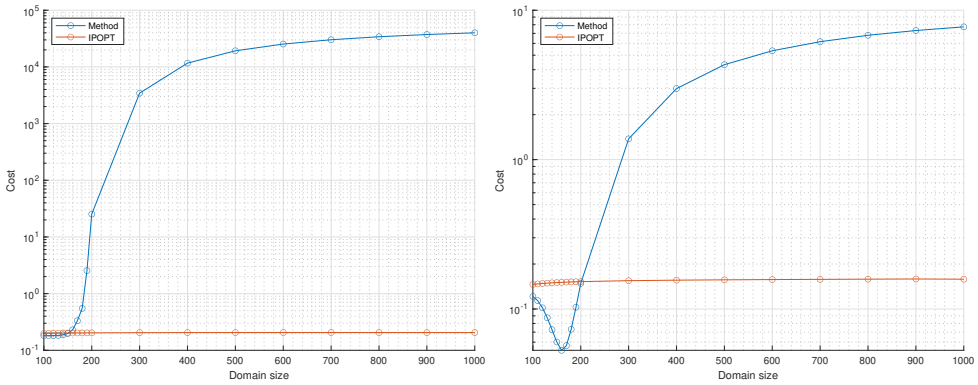
Figure 5.19: Comparing the number of FLOPs for the proposed method with that for IPOPT

5.3.8 Affect on accuracy for larger domain sizes

The method has been trained on data wherein the domain sizes vary between 10 and 100. Would the method still be able to arrive at good boundary values if the domain sizes are increased further?

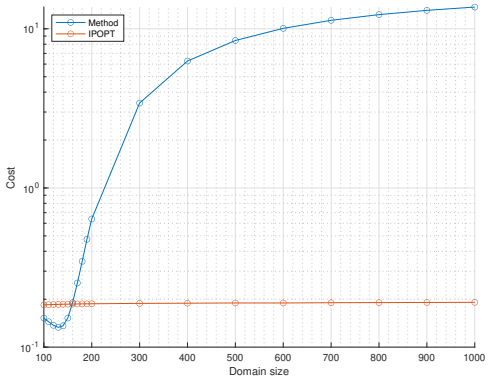
To answer this question, let us select a few questions and attempt to solve it for varying values of domain sizes. Starting with a value of 100, the domain size is increased by increments of 10 until it reaches 200. Afterward, the increments are changed to 100 until it reaches 1000.

In the figure, you can see that the method may work till about a domain size of 150. In some cases, it can be observed that the method starts to get lower cost as the domain size increases. However, the universal trend is that for even larger values of domain size, the method cost increases significantly.

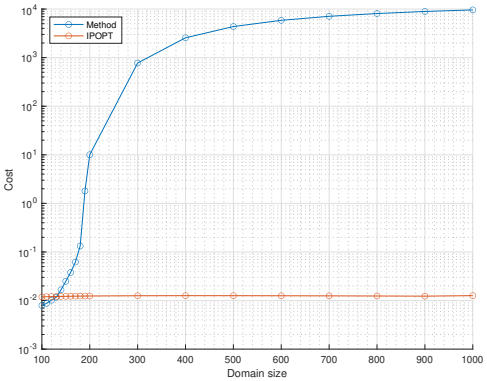


(a) Problem index 0

(b) Problem index 17



(c) Problem index 54



(d) Problem index 70

Figure 5.20: Cost for larger domain sizes

Chapter 6

Discussions

In this chapter, some of the potential limitations, flaws and incorrect assumptions and assessments in the method proposed, experiments conducted and results analysed are discussed. After these, another approach that was considered to solve boundary control problems with deep learning and reinforcement learning is mentioned as well.

6.1 Limitations

To begin with, one significant disadvantage of using deep learning based approaches is their lack of interpretability and explainability, especially in the manner in which it is used in this project. The proposed method is essentially a black box, and it would be hard to tell why it works in a certain way. Indeed, there may be two similar problems for which the network may work well for one and not quite well for the other, and it would not be easy to understand why that is the case.

Another limitation that needs to be taken into account is that the synthetic data generated to train the networks suffers from a lack of diversity and limited realism. This synthetic data may have also introduced biasses and mismatch with real world conditions, and hence, the method arrived at in this project is not likely to work well in most real world scenarios. The analysis with increasing the domain sizes has proven the same to some degree.

Furthermore, the manner in which the performance was compared between the proposed method and IPOPT for the solver speed may be flawed. Currently, the performance was evaluated based on the amount of time it takes for the solvers to perform 32 iterations. However, this does not account for the fact that a solver could converge to the optimal value way before that, and hence, even if the number of floating point operations or the amount of time taken per iteration is high, it does not mean that the solver is in effect

less performant than the other. The right way to compare the two would be to have the proposed method implemented in a faster lower lever language and have a head-to-head comparison of the amount of time each take for several different problems.

6.2 Another approach attempted

During the initial analysis phase, a few agent-based modelling with deep learning and reinforcement learning based approaches were looked into to solve boundary control problems. Such an approach seemed quite logical at the time considering that agents could control the values and be rewarded, and, this could very well be size invariant.

More details about these are provided below. Indeed, these approaches themselves may not have been abysmal; it is possible that the implementation may have had issues that could have contributed to their poor performance.

6.2.1 One agent per cell

The first approach was discretise the domain and the boundaries into different cells, and, each cell would be an agent. Each of these agents interact with each other and are rewarded based on how close they are to the desired profile, whether they satisfy the constraints and the governing PDE. The contribution to the reward for the governing PDE satisfaction would be kept quite high, followed by that for constraint satisfaction. Figure 6.1 helps in visualizing the setup.

The agent logic was modeled in accordance with my understanding on Multi-Agent Deep Deterministic Policy Gradient (MADDPG), which is proposed in a paper by Lowe et. al. titled *Multi-agent actor-critic for mixed cooperative-competitive environments*[53]. This method, in some sense, combines deep Q-learning with policy gradients.

However, it was observed that the networks were not improving even after several epochs.

6.2.2 One agent per boundary cell

After observing failure in the aforementioned method, an idea to reduce the reward complexity was considered. Instead of predicting for the domain cells, only the boundary cells are predicted using agent-based modelling. The domain values are solved by employing a PDE solver on the boundary values, and the cost and reward are calculated. Figure 6.2 helps in visualizing the setup.

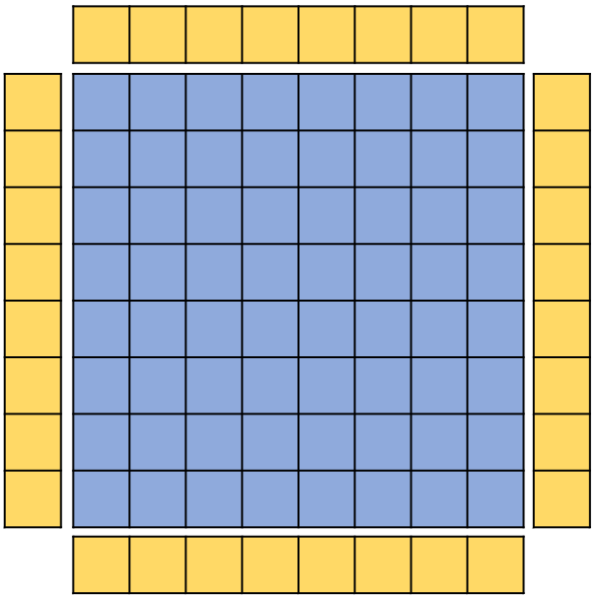


Figure 6.1: Each of these cells are individual agents

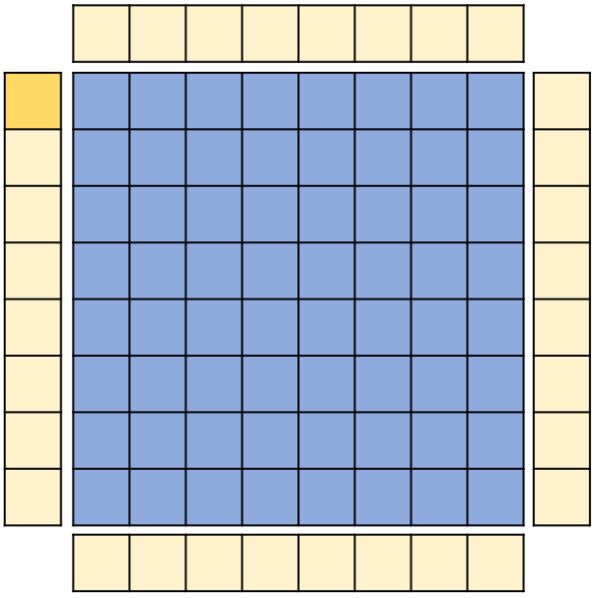


Figure 6.2: Each cells at the boundaries are individual agents. Highlighting one agent here.

Such a method would help in simplifying the reward function a bit; it would involve the contribution of the desired profile satisfaction and constraint violation only, and the contribution related to PDE satisfaction is automatically taken care of.

MADDPG based approach was taken to model the agent here as well, because the actions by each agent would affect the other agent and hence the agents need to cooperate with each other to achieve a higher reward. All the domain values are provided as an input to each agent.

However, this approach did not work, not even to a reasonable degree. The networks stopped learning after a few epochs and the values these converged to were not good enough.

6.2.3 One agent per pair of boundary cells

Both of the aforementioned approaches have a limitation that it is hard to define what additional inputs could go into the agents due to the way the agents are positioned. For simplicity sake, it is required that the agents are identical, and hence the overall information of the environment needs to be provided to each agents separately, along with the rewards for their actions. This means that, even if the method had worked, the amount of computation required to make it work would have been huge.

The idea here is to have boundary cells in the opposite sides be controlled by one agent. The input to the agent would be the values along the line that is in between these boundary cells, which can be attributed as it's field of view. Figure 6.3 helps in visualizing the same.

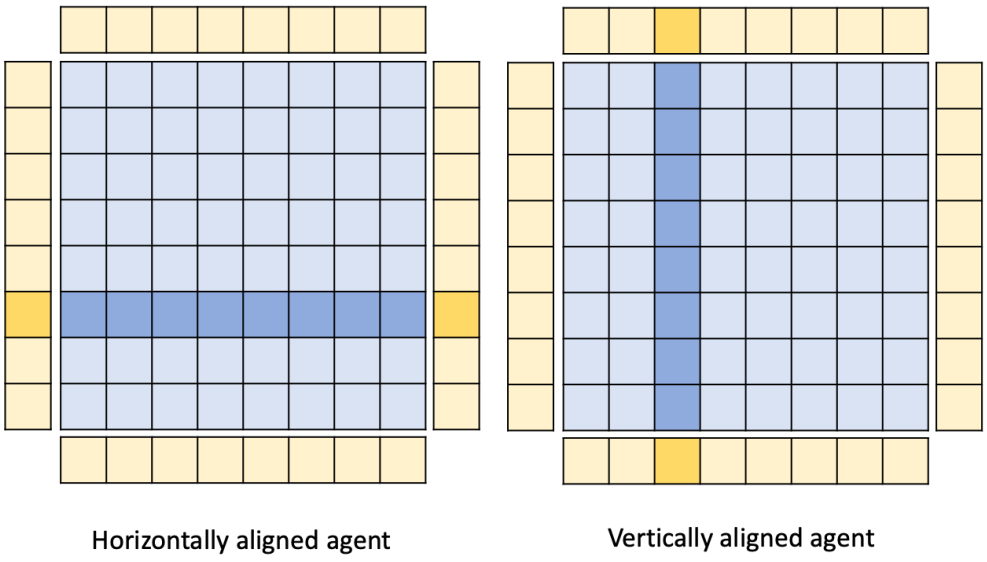


Figure 6.3: A pair of boundary cells form an agent. Highlighting an agent and it's field of view.

Indeed, to make this work for varying domain sizes, the field of view would need to be taken in by a size invariant network, or, it should be scaled to

meet the size that the network can handle.

Just like the other two methods, MADDPG based approach was taken to model the agent, as the agents need to learn to cooperate to maximize their rewards.

However, as innovative as this method appears to be, the accuracy observed was not very high in this case either.

Chapter 7

Conclusions

In this project, a study on identifying areas where and how deep learning and reinforcement learning methods can be applied to solve boundary control problems has been explored. Following this, an implementation of applying and evaluating deep learning and reinforcement learning strategies was performed.

In order to identify the areas where and how deep learning and reinforcement learning methods could be applied, inspirations were taken from existing iterative optimization algorithm, which predominantly relied a part that generated some initial guess value and another part that iteratively attempts to make this initial guess better and better over several iterations. These parts were respectively replaced with a deep learning network based on convolutional layers, and, with a policy gradient based reinforcement learning method that uses a deep learning network containing convolutional and LSTM layers within. The two parts are respectively referred to as the initial guess network and the optimizer network.

Through extensive experimentation and analysis, both initial guess and optimizer networks demonstrated to be effective. The initial guess network has been successful in outperforming other trivial methods of generating an initial guess value for the given problem to a great extend. The optimizer network put together with the initial guess network has been able to compete head-to-head and toe-to-toe with IPOPT, wherein it was able to arrive at lower cost than IPOPT in 50.97% of the cases.

However, it is to be noted that the method suffers from certain limitations and, hence, are avenue for further research and exploration. These are summarized below.

- The data for training, testing and validation were generated based on our understanding from a mathematical paper. Although about ten thousand problems were generated, it could be argued that these gener-

ated problems are still limited in the variations of different parameters, and hence, the method will likely fail to perform well beyond the set of parameters it has been trained and tested on. When it comes to real world problems in engineering, especially in fluid dynamics, the complexity of the problems could be much higher. Therefore, more such problems need to be generated with additional parameters observed in the real world conditions, and the network must be trained and tested on the same.

- For larger values of domain sizes, it is observed that the method performs poorly. An investigation needs to be conducted on whether the issue is caused by the initial guess method or by the optimizer method, or both. Based on that, formulate strategies that can be employed to help improve the approach.
- Each iteration of the optimizer method appears to be more computationally intensive than IPOPT. Research needs to be conducted to either reduce the computational load per iteration or reduce the overall number of iterations needed to solve problems, or both, without detrimentally affecting the accuracy. Some potential avenues for the same are:
    1. Faster PDE solvers: Currently, finite difference method was used to solve the governing PDE. Any improvement here could expedite the solver. Perhaps, neural network methods could be employed for this purpose.
    2. Faster gradient computation: Currently, PyTorch's autograd is used. It may be possible to calculate the gradient between the cost and the boundary values explicitly at a faster pace using other methods, or, develop a faster way to approximate the same.

Furthermore, agent-based modelling methods coupled with deep learning and reinforcement learning deserve further exploration. Intuitively, it is possible to use such methods to tackle boundary control problems. Successful implementation of these can lead to thinking about optimization problems in a new light.

Overall, approaching boundary control problems with deep learning and reinforcement learning has its merits. Further research may be conducted to create initial guess and optimizer networks that can obtain more accurate solutions with fewer computational resources, which could even work for optimization problems beyond just boundary control problems.

Bibliography

[1] Helmut Maurer and Hans D Mittelmann. Optimization techniques for solving elliptic control problems with control and state constraints: Part 1. boundary control. *Computational Optimization and Applications*, 16(1):29–55, 2000.

[2] Jens Lang and Bernhard A Schmitt. Exact discrete solutions of boundary control problems for the 1d heat equation. *Journal of Optimization Theory and Applications*, pages 1–13, 2023.

[3] Pierluigi Colli, Gianni Gilardi, and Jürgen Sprekels. A boundary control problem for the pure cahn–hilliard equation with dynamic boundary conditions. *Advances in Nonlinear Analysis*, 4(4):311–325, 2015.

[4] Colby L Wight and Jia Zhao. Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks. *arXiv preprint arXiv:2007.04542*, 2020.

[5] Xiushan Cai and Mamadou Diagne. Boundary control of nonlinear ode/wave pde systems with a spatially varying propagation speed. *IEEE Transactions on Automatic Control*, 66(9):4401–4408, 2021.

[6] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

[7] Zichao Jiang, Junyang Jiang, Qinghe Yao, and Gengchao Yang. A neural network-based pde solving algorithm with high precision. *Scientific Reports*, 13(1):4479, 2023.

[8] Johannes Brandstetter, Daniel Worrall, and Max Welling. Message passing neural pde solvers. *arXiv preprint arXiv:2202.03376*, 2022.

[9] Benjamin Chamberlain, James Rowbottom, Davide Eynard, Francesco Di Giovanni, Xiaowen Dong, and Michael Bronstein. Beltrami flow and neural diffusion on graphs. *Advances in Neural Information Processing Systems*, 34:1594–1609, 2021.

[10] Eliska Kloberdanz, Kyle G Kloberdanz, and Wei Le. Deepstability: a study of unstable numerical methods and their solutions in deep learning. In *Proceedings of the 44th International Conference on Software Engineering*, pages 586–597, 2022.

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

[12] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.

[13] Xiaojin Zhu. An optimal control view of adversarial machine learning. *arXiv preprint arXiv:1811.04422*, 2018.

[14] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Neural optimizer search with reinforcement learning. *CoRR*, abs/1709.07417, 2017.

[15] Peter Henderson, Joshua Romoff, and Joelle Pineau. Where did my optimum go?: An empirical analysis of gradient descent optimization in policy gradient methods. *arXiv preprint arXiv:1810.02525*, 2018.

[16] Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pretraining, 2023.

[17] Talal Bonny, Mariam Kashkash, and Farah Ahmed. An efficient deep reinforcement machine learning-based control reverse osmosis system for water desalination. *Desalination*, 522:115443, 2022.

[18] Rendong Shen, Shengyuan Zhong, Xin Wen, Qingsong An, Ruifan Zheng, Yang Li, and Jun Zhao. Multi-agent deep reinforcement learning optimization framework for building energy system with renewable energy. *Applied Energy*, 312:118724, 2022.

[19] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR, 10–15 Jul 2018.

[20] Sikai Chen, Jiqian Dong, Paul Ha, Yujie Li, and Samuel Labi. Graph neural network and reinforcement learning for multi-agent cooperative control of connected autonomous vehicles. *Computer-Aided Civil and Infrastructure Engineering*, 36(7):838–857, 2021.

[21] Yifei Shen, Yuanming Shi, Jun Zhang, and Khaled B. Letaief. A graph neural network approach for scalable wireless power control, 2019.

[22] Eli Meirom, Haggai Maron, Shie Mannor, and Gal Chechik. Controlling graph dynamics with reinforcement learning and graph neural networks, 2021.

[23] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, Guzman Lopez, Nicolas Collignon, and Rik Sarkar. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, pages 4564–4573, 2021.

[24] Armaghan Zafar and Ian R. Manchester. Structured linear quadratic control computations over 2d grids, 2023.

[25] Fernando Gama and Somayeh Sojoudi. Graph neural networks for distributed linear-quadratic control, 07 – 08 June 2021.

[26] Byron Tasseff, Carleton Coffrin, Andreas Wächter, and Carl Laird. Exploring benefits of linear solver parallelism on modern nonlinear optimization applications, 2019.

[27] Juraj Kardoš, Drosos Kourounis, and Olaf Schenk. Two-level parallel augmented schur complement interior-point algorithms for the solution of security constrained optimal power flow problems. *IEEE Transactions on Power Systems*, 35(2):1340–1350, 2020.

[28] Juraj Kardoš, Drosos Kourounis, Olaf Schenk, and Ray Zimmerman. Beltistos: A robust interior point method for large-scale optimal power flow problems. *Electric Power Systems Research*, 212:108613, 2022.

[29] François Pacaud, Michel Schanen, Sungho Shin, Daniel Adrian Maldonado, and Mihai Anitescu. Parallel interior-point solver for block-structured nonlinear programs on simd/gpu architectures. *arXiv preprint arXiv:2301.04869*, 2023.

[30] Florian A. Potra and Stephen J. Wright. Interior-point methods. *Journal of Computational and Applied Mathematics*, 124(1):281–302, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[31] Margaret Wright. The interior-point revolution in optimization: history, recent developments, and lasting consequences. *Bulletin of the American mathematical society*, 42(1):39–56, 2005.

[32] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106:25–57, 2006.

[33] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[34] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[35] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

[36] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[37] R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018.

[38] Herbert E. Robbins. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[39] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[40] Geoffrey Hinton. Neural networks for machine learning, 2018.

[41] Agnes Lydia and Sagayaraj Francis. Adagrad—an optimizer for stochastic gradient descent. *Int. J. Inf. Comput. Sci*, 6(5):566–568, 2019.

[42] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[43] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[44] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[46] Zenin Easa Panthakkalakath, Juraj Kardoš, and Olaf Schenk. Master thesis: Application of deep learning and reinforcement learning to boundary control problems. https://github.com/zenineasa/MasterThesis, 2023.

[47] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[48] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[49] Anaconda software distribution, 2023.

[50] The MathWorks Inc. Matlab version: 9.13.0 (r2022b), 2022.

[51] Hamid Soltani. Math expression parser in c++, 2016.

[52] Herbert Schildt. *C++: The complete reference*. McGraw-Hill, 2003.

[53] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

APPLICATION OF DEEP AND REINFORCEMENT LEARNING TO BOUNDARY CONTROL PROBLEMS

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
PANTHAKKALAKATH	ZENIN EASA

With my signature I confirm that

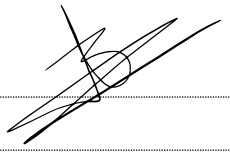
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

LUGANO, 02 JUNE 2023

Signature(s)



For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.