Combinatorial Optimisation on Graphs for Radio Astronomy

Aavash Subedi
Department of Physics and Astronomy
University of Manchester
aavashsubedi1@gmail.com
Student ID: 10638164

Work done in collaboration with Labeebah Islaam.

Abstract

This project explores the applications of combinatorial optimisers within a machine learning pipeline, providing a unique way to intersect the two fields to solve problems. We reproduce an example in the literature [1] using a Convolutional Neural Network and Dijkstra's algorithm for applications to the Warcraft II tileset. Our model accurately reproduces the shortest path for 77.3 ± 2.1 % of unseen samples compared to 96.0 ± 0.3 % documented in the reference paper. Although we cannot reproduce the performance, we demonstrate that including Dijkstra's algorithm greatly benefits the accuracy with pure machine learning-based models scoring 23.3 ± 0.3 % as reported in literature. Reproduction of this example serves as a proof of concept for applications to radio astronomy[2].

[1] M. Vlastelica, A. Paulus, V. Musil, G. Martius, and M. Rolínek, "Differentiation of blackbox combinatorial solvers," 2020

[2] The full code used for this project can be found in `https://github.com/aavashsubedi/masters_project`

Contents

1 Introduction

In recent years, Machine Learning has found massive success in the natural sciences. Dubbed the "fifth" paradigm of scientific discovery, many innovations are credited to deep learning; notable examples include GraphCast for weather forecasting, AlphaFold for protein folding, and many more [2; 3; 4]. Successes within astrophysics are also plentiful, with models assisting astronomers in analysing the petabytes of data from sky surveys across the entire electromagnetic spectrum [5]. This thesis explores a potential application of deep learning, intersecting these algorithms with traditional combinatorial optimisers for radio interferometry. Specifically, we are interested in combining these powerful techniques to meaningfully divide the SKA-Mid array into smaller heterogeneous subarrays to be used for multiple scientific tasks in parallel.

1.1 Radio Astronomy

Radio astronomy is concerned with the study of radio emissions from celestial bodies. Focusing on radio frequencies within the electromagnetic spectrum, this scientific tool provides a unique perspective of the cosmos, unobstructed by cosmic dust and gas [6]. Since Karl Jansky's seminal work in the 1930s, numerous discoveries have been made possible with this technique, ranging from detecting cosmic microwave background radiation to imagining black holes [7] [8]. However, generating these images is more complex and challenging than traditional optical telescopes. A large diffraction limit, which specifies the minimum angular separation a telescope can resolve, must be overcome, which is far greater than the limit of optical telescopes due to the enormous wavelength of radio waves. To achieve comparable signal-to-noise ratio and angular resolution, the diameter of a radio telescope must be much larger than its optical counterparts [6]. However, building larger and larger telescopes is not economically and technologically feasible; to overcome this challenge, astronomers turn to interferometry.

Interferometry combines the received signals from two or more antennae, creating a virtual telescope with a much larger collecting diameter than individual telescopes. Each pair in an interferometry array collects a part of the signal, with all pairs contributing to the final image [9]. Crucially, the distance or baseline b between a pair of telescopes, as measured in a right-handed coordinate system with respect to the Earth's rotational axis, plays a critical role in image synthesis. A longer baseline yields higher resolution signals that allow finer details to be seen, but at the expense of imaging diffuse long-range structures. Conversely, a shorter baseline better visualises larger objects but struggles with inspecting finer details [6]. Image synthesis is a careful balancing act, ensuring the final image contains the necessary details for each scientific application. This is notoriously difficult due to the combinatorial solution space of pairs that can be selected. The Earth's rotation transforms the baseline vector due to the coordinate system used, adding a dynamic element to an already complex problem. Overcoming this challenge can provide massive benefits due to the flexibility of interferometers; effective pair selections can allow scientists to adapt the device to study both the hot and the cold universe, peering into low-energy physical processes such as cosmic dust as well as extremely hot environments found in quasars [6]. By using machine learning and combinatorial optimisers, we aim to effectively sample the solution space and segment an entire interferometry array into smaller subarrays so that they may be used for various applications in parallel.

1.1.1 SKA

The Square Kilometer Array (SKA) is an upcoming interferometer with linked radio wave telescopes in Australia and South Africa. Upon completion, the SKA will be the largest interferometer ever built, achieving 50 times more sensitivity than existing telescopes. The SKA consists of two subarrays: SKA-Low, a collection of 131,072 telescopes operating in the frequency range of 50 MHz - 350MHz and SKA-Mid, which contains 197 large parabolic radio dishes working within the 350 Mhz - 15.4 GHz frequency range [10]. As the generated image depends on the light received, the properties of each antenna and the baseline vectors between pairs, we can effectively reduce the SKA-Mid array to a set of nodes and edges. Node features capture antennas and their properties, and the baselines behave as edge features in a fully connected graph. This transformation preserves the underlying and allows us to utilise the vast number of graph optimisation and graph-based machine learning techniques to tackle our problem. The large number of telescopes poses a problem due to the vast sample space. Selecting n subarrays from 197 satellites of the SKA-Mid array is combinatorial with

$\frac{(n+196)!}{197! \times (n-1)!}$ possible combinations. This number is further increased when we account for potential baseline pairings and the constraints imposed by physics, which are discussed in the section below. By applying graph-based combinatorial optimisers found in the literature, we can navigate the complex solution space and tackle the subarray problem.

We are interested in the SKA-Mid array within our project and will refer to it as SKA henceforth.

1.2 Combinatorial Optimisers

Combinatorial optimisers are highly versatile algorithms that work by finding the optimal solution of an objective function in a discrete and non-convex solution space [11]. Their applications span many domains, including logistics, route planning, supply chain optimisations, and even addressing scientific problems such as estimating reservoir water flow rates [12]. Infamous problems solved by these optimisers include the Travelling Salesman Problem (TSP), the Knapsack Problem, and the Job Scheduling Problem [13]. Many real-life scenarios can be distilled into a combinatorial problem, and applications of these algorithms to different domains have found tremendous success. We peer into the TSP problem and its application to highlight its versatility. Initially designed to find the optimal route that allows a salesman to travel across all cities without repetition, scientists have successfully employed this algorithm to optimise circuit design in quantum computers, enhancing their performance and efficiency [14]. The analysis of crystal structures obtained during X-ray crystallography has also benefitted from the TSP formulation[15]. These examples highlight the interests of academics and industry in innovating at the forefront of combinational optimisers and demonstrate their ability to solve real-world problems.

1.3 Machine Learning for Radio Astronomy

Machine learning for astronomy is a blossoming field that has found immense success due to the ever-increasing volume of generated data. The need for an automated, robust and effective method for data analysis has led to the increasing popularity of machine learning among astronomers [16]. Recent advances include traditional machine learning algorithms such as Support Vector Machines to search for variable stars [17], decision trees for separating galaxies and stars [18] and many others. Modern deep learning techniques, such as Geometric Deep Learning and attention-based transformers, have also enjoyed success in radio galaxy classification[19] [20]. Currently, deep learning techniques are being explored for inclusion in the SKA's data processing pipeline, ensuring that astronomers can readily reap its benefits when the telescope becomes fully operational [21] [22].

1.3.1 Machine Learning and Combinatorial Optimisers

The intersection of machine learning and combinatorial optimisers is of great interest to scientists due to its potential to revolutionise both fields. Research at the intersection can generally be divided into two categories: machine learning for combinatorial optimisers and combinatorial optimisers incorporated into a machine learning pipeline. The prior focus on creating faster and more efficient machine learning-based approximators of computationally expensive combinatorial optimisers to solve previously intractable problems [23]. Scientists have successfully used Reinforcement Learning and Graph Neural Networks to approximate and solve TSP, Shortest Path Problems, and many others [24; 25; 26]. Conversely, incorporating task-specific solvers and a gradient approximator, scientists have had massive success in problems where machine learning alone has struggled [1]. A notable example is found in the reference paper, where a machine learning model alone achieved 0% accuracy on a problem. In contrast, they report a near-perfect 99% accuracy on unseen data when a combinatorial optimiser is included.

Combining the scalability and generalisability of machine learning models with the theoretical guarantees of a combinatorial optimiser can vastly revolutionise both fields, paving the way for academia and industry to benefit from these advances [27]. It is important to note that there are limited implementations found in the astronomy workflow. By exploring the application of this new paradigm, we hope to open avenues for future work within astronomy and the broader natural sciences.

2 Background

2.1 Radio Interferometer

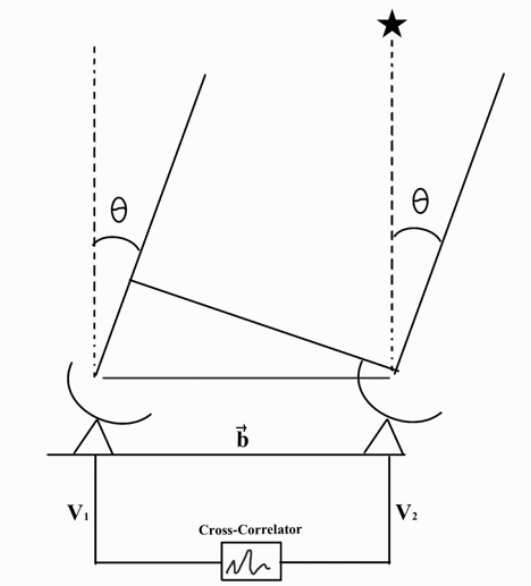


Figure 1: A pair of telescopes conducting interferometry. Signals from a far-field source arrive at an angle θ at the two receivers, and an interference pattern is produced. Measured electric fields are converted to voltages $V_{[1,2]}$ before being merged by a cross-correlator. The baseline vector b is measured in units of wavelengths of the received signal.

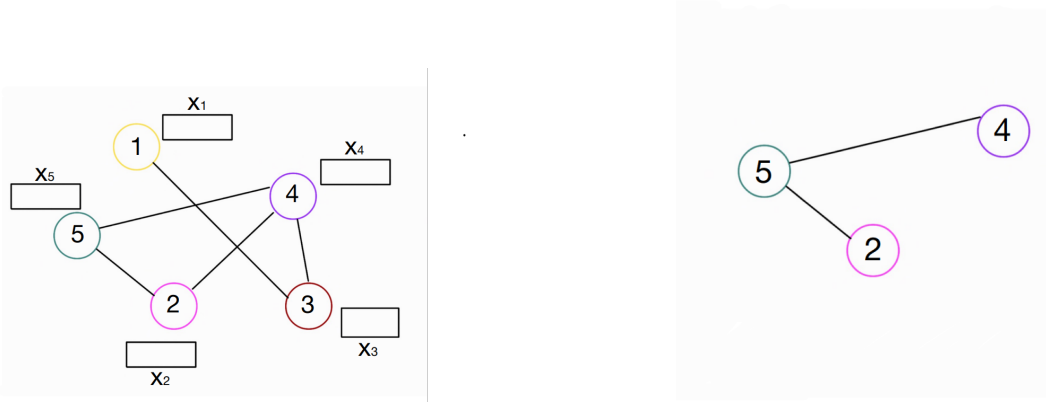
An interferometer is an array of telescopes working in unison to produce a complete image of a source. It contains many telescope pairs, all contributing to the final image via a process known as aperture synthesis [28]. To begin our discussion, we isolate and study one pair. As shown in Fig 1, each telescope pair is separated by a baseline vector b, which captures the displacement in units of wavelengths [6]. Signals arrive at the telescope with a phase difference $\delta \propto b$ due to the varying distance light must travel, producing an interference pattern. The antenna converts measured electric fields of the arriving signals into voltages, V_1, V_2, which are complex. These voltages are processed by a complex cross-correlator, which outputs a visibility measurement $V(x, y, z)$ [29]. Following this, a change of coordinates converts the baseline vector into the (u, v, w) coordinate system. Here, w captures the line of sight of the baseline, and u and v are the eastward and northward coordinates as measured via a right-handed coordinate system. Notably, this coordinate system depends on the hour angle, which measures the sun's angular displacement relative to the Earth's local meridian. This quantity and the coordinate system depend on the planet's rotation [6]. As such, even the same pair can produce different baselines depending on the time of day.

The visibility contains information regarding the amplitude and phase difference between the two signals [6]. Following the Cittert-Zernike theorem, astronomers can utilise the measured visibility and obtain the sky brightness of the source via an inverse Fourier transform [29][30][31]. Each baseline contributes one measurement to the visibility, $V(u, v)$; as such, extensive visibility coverage is desired. Interferometer arrays contain many telescopes N corresponding to $N(N-1)/2$ pairs and baselines to make this possible. The resolution of the final image we generate depends on the baselines we select, so it is essential to select these carefully. A long baseline contributes to a higher resolution, but concurrently, this can make diffuse extended objects indiscernible [6]. As such, a balance of short and long baselines is required. An example of this is to obtain the very first image of a blackhole, Very Long Interferometry was required, where baselines separated by up to thousands of kilometres were used [8]. Conversely, shorter baselines, which can resolve larger objects such as gas clouds and nebula, have been crucial in studying the Cosmic Microwave Background Radiation [7].

Due to the nature of interferometers, there are often more short baselines than long baselines. Homogeneous distribution of baselines is often desired in the image generation step to suppress

the over-representation of short baselines. This is because taking the Fourier transform of non-uniform baselines introduces grid artefacts in the final image, requiring complex and computationally expensive deconvolution algorithms to clean [9]. Additionally, due to the physical constraints of the telescope, we cannot wholly sample the visibility space, resulting in gaps in the generated image. Weighting each baseline before image synthesis ensures that all spatial scales are sampled uniformly and omits the computationally expensive deconvolution steps.

These dynamical factors prove challenging when tackling the task of segmenting the interferometer. Within our research question, each telescope may only belong to one array. As such, the baselines it provides can only contribute to one image. Hence, telescopes must be selected carefully for the visibility constraints introduced by the scientific problem while accounting for the requirements of other subarrays. Given the number of possible permutations of this selection, the sample space becomes huge. Using machine learning and combinatorial optimisers, we aim to tackle this issue. Additionally, we hope to organically generate the different baseline weightings required so that astronomers may use this setup to conduct research.



(a) (b)

Figure 2: a) A sample graph displaying the connections and node features x_i. b) Neighbourhood $\mathcal{N}_5$, for the node labelled 5.

2.2 What is a Graph?

To fully understand the graph formulation of the SKA, we must first define what a graph is. A graph, $G(V, E)$ as visualised in the left of Fig 2, is a data structure consisting of a finite set of vertices (nodes) $V = \{v_1, v_2, ..v_n\}$ and a set of edges E that convey the relationship between a pair of vertices, $E \in \{(i, j)|i, j \in V\}$ [32]. Graphs can be categorised into directed and undirected graphs, where directed edges restrict the direction of information flow. Each of these attributes, nodes, edges, and the graph can have features that encode information regarding the entity. Graph-wide features could be a particular property of the entire graph, e.g. the binding energy of a molecular graph. Node features within the same graph could be the type of atom it represents or the size, and edges could quantify the distance between the atoms. Due to requirements for graph weights and features, we extend the previous definition by incorporating a matrix of node features $X \in \{x_1, x_2, ..x_n\}$, and $E_f \in \{E_{f1}, E_{f2}, ..E_{fe}\}$, a set of edge features, where e is the number of edges. Notably, edge features that are singular numbers are known as edge weights. We arrive at a rich and highly informative graph $G(V, E, X, E_f)$. Node neighbourhoods $\mathcal{N}_i$ of node i is a powerful concept in graph theory and is used extensively in Graph Neural Networks. A neighbourhood refers to the set of nodes j that are connected to i via an edge, i.e. $\mathcal{N}_i = \{j : e_{ij} \in E\}$. The neighbourhood of node $i = 5$ can be seen on the right of Fig 2, and are the nodes $\{2, 4\}$.

2.3 Combinatorial Solvers

Combinatorial optimisers are algorithms that work by finding the optimal solution of an objective function in a discrete but ample solution space. However, the algorithms' discrete solutions are

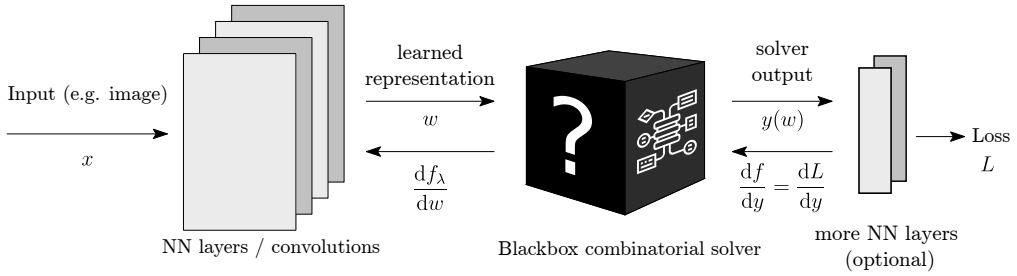


Figure 3: Pipeline for incorporating gradient approximators with machine learning. The arrows denote the direction of information flow, beginning at the model to the combinatorial optimiser + gradient approximator. Following this, an optional neural network layer can be included. This figure is taken from [1].

often non-differentiable, making it challenging to incorporate them into a machine learning pipeline, which requires gradients to operate. To remedy this, a gradient approximator is required. One such approximator is provided in Vlastelica et al. [1]. An illustration of the machine learning pipeline can be seen in Fig. 3 above. An interpolating function, see pseudocode below, generates approximate gradients from the optimiser output. This interpolator smooths the piecewise step function into a continuous-differentiable function to be used for backpropagation. The complete theoretical derivation of this approach can be seen in the literature [1].

2.3.1 Gradient Approximator

Algorithm 1 Forward and Backward Pass. This pseudocode is a modified version of the algorithm as found in [1]

function FORWARDPASS($\hat{o}$)
// $\hat{o}$: Output weights of model.
  $\hat{y} := \text{SOLVER}(\hat{o})$
  save $\hat{o}$ and $\hat{y}$ for backward pass
  return $\hat{y}$
end function

function BACKWARDPASS($^{\partial L}/_{\partial y}(\hat{y}), \lambda$)
// $^{\partial L}/_{\partial y}(\hat{y})$: Gradients from loss function
  load $\hat{w}$ and $\hat{y}$ from forward pass
  $o' := \hat{o} + \lambda \cdot {}^{\partial L}/_{\partial y}(\hat{y})$
// Calculate perturbed weights
  $y_\lambda := \text{SOLVER}(o')$
  return $\nabla_o f_\lambda(\hat{o}) := -\frac{1}{\lambda}\left[\hat{y} - y_\lambda\right]$
// Gradient is passed back to model for training
end function

The algorithm given in the reference paper heavily inspires the pseudocode given. Here, λ is a parameter that controls the level of smoothing applied to the function. A balance must be achieved to ensure we accurately represent the true function while smoothing small local perturbations during interpolation. Additionally, $\hat{o}$ is the output from the machine learning model, and y refers to the output from the combinatorial solver using either the output weights o or perturbed weights $\hat{o}$. The gradient obtained from the backward pass of the loss function is given by $^{\partial L}/_{\partial y}(\hat{y})$ and the gradients passed to the model are $\nabla_o f_\lambda(\hat{o})$. This algorithm represents the black box, as shown above in Fig. 3.

2.3.2 Dijkstra

Dijkstra's algorithm is a pathfinding algorithm that finds the shortest path between two nodes of a weighted graph, where edge weights represent the cost associated with traversing through them [33]. Minimising the summed edge weights for all edges in the path is equivalent to finding the shortest path. Commonly used for routing protocols and pathfinding, the algorithm requires an input graph G and a pair of starting and ending nodes (n_o, n_f). Iteratively, the shortest paths to each node in the path are found until we reach the final node. A pseudo-code inspired by literature is given below [34; 35].

Algorithm 2 Dijkstra's Algorithm for Shortest Path. Modified version of: [34]

function DIJKSTRA(Graph, n_i, n_f) // n_i: Starting node // n_f: Ending node
    $D[v] \leftarrow \infty$ for all v in Graph
    $D[n_i] \leftarrow 0$
    Mark all nodes as not visited
    while there are unvisited nodes **do**
        $u \leftarrow$ unvisited node with smallest $D[u]$
        Mark u as visited
        if $u == n_f$ **then**
            break
        end if
        for each neighbor v of u **do**
            $val \leftarrow D[u] + \text{length}(u, v)$
            if $val < D[v]$ **then**
                $D[v] \leftarrow val$
            end if
        end for
    end while
end function

Although there are no prior applications in the astrophysics setting, following the seminal paper on this domain, we propose first to investigate the shortest path problem for images. Specifically, we will pair an image-based convolutional neural network with Dijkstra's algorithm.

2.4 Machine Learning

Machine Learning works based on gradient descent, an optimisation algorithm based on iteratively updating a model's parameters. For the confines of our project, we are particularly interested in supervised and self-supervised learning, with the latter explored in the second half of our project. Supervised learning uses a known label for quantifying the error in the model's output, known as a loss. We can backpropagate the gradients using this loss, updating the models' parameters [36]. A significant challenge that models face is the "curse of dimensionality", where the number of data points is insufficient to successfully generalise a high-dimensional model's parameters [37]. Given that most current models contain many trainable parameters, with "bigger is better" being the current trend, we require techniques to mitigate the effects of this curse [38]. Parameter and data-efficient models can be generated by injecting inductive biases into the model architectures. These biases allow models to be tailored to specific applications, whether images, graphs or text.

We briefly introduce artificial neural networks and the gradient descent algorithm used for training. A more detailed study can be found in Chapter 2 of Goodfellow et al. [39].

2.4.1 Artifical Neural Networks

This section is heavily inspired by [39]. Artificial neural networks (ANN) are a computational approximation of neurons within our brains. We can consider a neural network to be a collection of neurons that combine to transform data x to a desired output y. This is done via a series of non-linear transformations of the input data with adjustable and learnable transformation parameters. We begin with a singular neuron. Mathematically, the neuron can be considered a function $f_i(X)$ that outputs a value y_i,

$$y_i = \sigma(W_i * x^T + b_i), \tag{1}$$

where W_i is the weight matrix of the neuron and σ is an activation function. By stacking these neurons in layers and feeding the output of each neuron forward, ANNs perform a parametric approximation of the true data-generating function.

Activation Functions To model complex and non-linear functions, an activation function σ is required. Without this activation function, we reduce a large, multi-layered neural network to a single

linear transformation, drastically restricting the ability to model non-linear relationships as found in the real world [40]. Introducing a non-linearity allows for more powerful models and has been mathematically shown to follow the Universal Approximation Theorem. This theorem states that any ANN with sufficient neurons, layers, and data can accurately model any desired function [41]. A ReLU activation function is a simple, computationally inexpensive non-linearity that has found immense success and adoption. It is defined as $ReLU(x) = max(0, x)$.

2.4.2 Training

To train a neural network is akin to finding the optimal parameters $\hat{\theta}$ that minimises a loss function, F_l. In supervised learning, a loss quantifies the error in a model's prediction, $L = F_l(\hat{y}, y)$, where $\hat{y}$ and y are a model's response and true label, respectively. Models are trained via gradient descent, where $\hat{\theta}$ is updated iteratively using the loss. An iteration of gradient descent can be seen in the equation below,

$$\theta_{new} = \theta_{old} - \alpha \cdot \nabla_\theta L, \tag{2}$$

where α is a hyperparameter known as the learning rate and θ_{old} is initialised based on the user's choice. A popular choice is random initialisation. Gradients are initially obtained with respect to the final layer and propagated backwards via the chain rule for differentiation [36]. Since the first implementation of gradient descent, advancements in the initial algorithm have been made to produce faster, more reliable and robust training algorithms.

It is important to note that throughout this project, we utilise a variation of gradient descent called Stochastic Gradient Descent (SGD). This algorithm introduces a momentum term, which adjusts and affects the learning rate, α. In traditional gradient descent, the iterative update term $\alpha \cdot \nabla_\theta L$ solely depends on the gradient at the current position in the loss landscape. However, by introducing a momentum term, the optimisation algorithm retains information regarding past gradients and moves accelerates towards the optimal solution[42]. This system of equations gives the updated SGD algorithm,

$$\begin{aligned} v_{\text{new}} &= \beta \cdot v_{\text{old}} + \nabla_\theta L \\ \theta_{\text{new}} &= \theta_{\text{old}} - \alpha \cdot v_{\text{new}}, \end{aligned} \tag{3}$$

where β is the momentum term, typically bounded between 0 and 1 and v_{old} is initialised to zero. We can liken the effect of SGD to a particle falling down a potential, where the dynamics of the particle depend not only on the potential at its current location and its associated acceleration but also on its current velocity. Since its inception, scientists have leveraged statistical and numerical methods to develop more stable and faster optimisation algorithms than SGD. Throughout this project, we utilise a variation of the SGD optimiser called Adam [43]. We refer readers to the literature for a detailed investigation of the optimiser landscape [44].

2.4.3 Key Terminology

Below is a non-exhaustive list of terminology associated with training machine learning models.
Batches Typically, models optimised via SGD are fed a smaller subset of the entire dataset. This is done due to computational constraints where loading the entire training dataset into memory is intractable. An estimate of the true stochastic gradients is obtained by averaging the gradients across multiple batches [39].
Epoch An epoch is the number of iterations required to see the entire dataset exactly once. For example, if there are 1000 examples and a batch size of 10, an epoch equals $1000/10 = 100$ iterations.
Overfitting Overfitting is the phenomenon where the model learns to exactly fit the training data instead of learning the underlying data-generating model. This is typically not desired as this leads to models performing poorly on unseen data or generalising.
Invariance & Equivariance Equivariance is a mathematical property of a function, where under the transformed input, the output transforms similarly. E.g. for an equivariant function $\phi(x)$, $\phi(Ax) = A'\phi(x)$. Here, A is a matrix representation of a transformation to which $\phi(x)$ is equivariant. For equivariant functions, $A = A^{'}$. If $\phi(x)$ was instead an invariant function, A' would be equivalent to the identity matrix. Equivariance and invariance to transformation are a desired property of models as they lead to more data and parameter-efficient models [45].

To produce more robust, parameter and data-efficient models, inductive biases are injected while producing the model architectures. These biases allow for models to be tailored to specific tasks, whether it be for images or graphs.

2.5 CNN

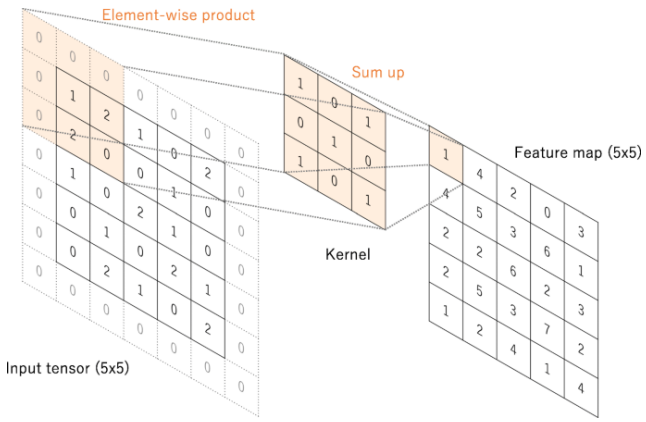


Figure 4: An example of a convolution kernel being applied over an image to obtain a feature map. Image is taken from: [46]

Convolutional Neural Networks (CNN) are a specialised form of neural networks designed to work on regular lattice-like structures such as images [39]. Pioneered in 1998 by Yann Lecun while working at Bell Labs, the algorithm has become a staple in modern machine learning workflows[47]. The power of a CNN lies in the convolution kernel and the cross-correlation operation. A filter or kernel is applied to an image, and cross-correlation is computed for each element it passes over, outputting a feature map that distils the contents of the image to a smaller dimension. Mathematically, this operation is the sum of the elementwise multiplication of the kernel matrix K and the elements of the image matrix I that the kernel is currently sliding over [39]. A visualisation of this operation can be seen in Fig 4 above. The mathematical cross-correlation operation of a two-dimensional kernel is as follows,

$$(I * K)_{ij} = \sum_{m} \sum_{n} I(i+m, j+n) \cdot K(m, n) \tag{4}$$

where K is a kernel matrix of shape $m \times n$, i and j are used to specify individual elements of I [39]. Typically, these learned convolution kernels behave like feature extractors, where the kernel is used to extract edges, textures, colours and the local spatial features found in images.

2.5.1 Weight Sharing

Weight sharing is a powerful notion in CNNs that allows for more parameter efficiency and mitigates the effects of the "curse of dimensionality". CNNs operate by learning a kernel's parameters instead of separate parameters for each pixel within the image. For example, for an input image of shape $i \times j$, the input layer of an ANN requires $i \times j$ parameters. However, the number of trained parameters for a convolutional layer is $k \times k$. As k is typically much smaller than the dimension of the image, we drastically reduce the number of parameters [47]. Importantly, using a kernel-based approach, the model is agnostic to feature translations as such CNNs display translational equivariance. Equivariance is the property of a function where transformation to the input also transforms the output similarly. This is a desired property of CNNs as it has been empirically shown that equivariant modules are often more robust and parameter and data efficient.

2.5.2 Pooling

Pooling layers are pivotal to a CNN as they allow models to downsample the computed feature maps and summarise the information stored in a region into a single scalar variable [39]. Common pooling operations include Max and Average pooling. Max pooling returns the maximum value stored over a region of the image, while average pooling returns the average. Like a convolution

kernel, a pooling kernel slides over the feature maps and extracts salient information. Instead of computing the cross-correlation, the selected pooling operation is applied. The receptive field of the pooling operator is a hyperparameter selected by the user, similar to the shape of convolution kernel k. Pooling allows representations to be invariant to small translations of the feature map. Additionally, by downsampling the output feature, model complexity is reduced, decreasing the computational overhead and the possibility of overfitting.

2.5.3 Hyperparameters

A hyperparameter is a feature of the model or training pipeline which must be selected before training. Unlike the learned parameters found in models, these features are not optimised by gradient descent.
Kernel Size The kernel size determines the shape of the convolution kernel, which allows models to have different receptive fields. Larger kernel sizes increase the receptive field and capture broader spatial patterns as each element of the feature map is a summarised statistic of a greater fraction of the image. We can think of this as a shorter baseline, where larger diffuse objects can be easily resolved. A smaller kernel would allow models to focus on finer localised image details [39]. Selecting the size of the kernel is a careful balancing act between desired feature maps and computational complexity.
Number of Layers The number of layers is a crucial hyperparameter in any machine learning workflow. CNNs have been shown to learn task-agnostic features within the shallower layers comprising simple edge, colour and texture detectors [48]. Deeper layers often extract more complex structures. Although more layers allow more complex patterns to be analysed, it also makes the model more prone to overfitting and increases the computational complexity.
Channels We have so far only considered a singular kernel coupled with a single-channel image. However, in typical implementations, this is often not the case. CNNs employ multiple kernels within each layer to extract various features where each channel can be considered a unique and independent convolution filter [39]. Increasing the number of kernels allows the model to learn more complex and varied features while increasing the number of learned parameters.

Finding the correct set of hyperparameters or "hyperparameter optimisation/tuning" is crucial for each task and can drastically alter a model's training dynamics and predictive power.

2.6 GNN

Graph Neural Networks (GNN) operate as learned transformations of a graph's properties. By passing an input graph through a series of GNN layers, desired feature transformations that minimise the loss are applied. Throughout this report, GNNs will be described within the "message-passing" framework, where we consider node features as messages and edges as message carriers. We can DIVIDE the operation of a GNN layer into three key steps, Message Passing, Aggregation and Update. This section is heavily inspired by [49] [50].

For convenience, we will begin by focusing on one node i, but the steps apply to all nodes in a graph.
Message Passing Using the neighbourhood $\mathcal{N}_i$ of node i, messages from nearby connected nodes are passed to the current node. Each node receives a set of messages $\{F_1(x_1), F_2(x_2), F_3(x_3), ..., F_n(x_n)\}$ where $F(x_j)$ is a learned function, transforming each message x_j from node j. Importantly, we are free to choose the parameterisation of F_j. It can be either an ANN or a non-linear transformation. The messages that are sent can simply be the node features. If edge weights or edge features are given, the messages will incorporate these before the passing stage. We can think of $F(x_j) = W_j \cdot x_j$

Aggregation The set of transformed messages received by node i are then combined or aggregated. The aggregating function is also a hyperparameter that can be selected, provided it does not break permutation invariance. Permutation invariance is a feature that means the ordering of nodes and feature labelling does not matter [51]. In the aggregation step, we can think of this as the order of aggregating nodes j, k, and then l, equivalent to any permutation of these nodes. Popular examples include Sum, Mean or Min/Max over all messages. Using an aggregating function A_G, the aggregated message $\bar{m}$ the node i receives is given by,

$$\bar{m_i} = A_G(\{F_j : j \in \mathcal{N}_i\}) \tag{5}$$

Update The combination of aggregated messages and the current node, edge and graph features is used to update the feature representations iteratively. The current node, edge and graph features can

be updated using the messages. The user is free to select the function used for the update step. The update yields a node representation h_i and is given by the equation below,

$$h_i = \sigma(U(H(x_i), \bar{m}_i)), \tag{6}$$

where U is an update function that takes as input the aggregated messages, $H(x_i)$ is a transformation of the current node features x_i, and σ is a non-linearity (e.g. ReLU) as discussed in the previous section. Often, U and H are distinct ANNs with learned parameters.

Combining the previous steps, we arrive at a unified update rule for a single node,

$$h_i^{new} = \sigma(U(h_i^{(old)}, A_G(F_j(x_j) : j \in \mathcal{N}_i)), \tag{7}$$

where the equation represents a single layer within a GNN. For example, using a sum aggregator and single layer ANNs as H and F_j, we arrive at an updated representation at step k,

$$h_i^k = \sigma(\mathbf{B}^k \cdot h_i^{k-1} + \sum_{j \in \mathcal{N}_i} \mathbf{W} \cdot h_j^{(k-1)}), \tag{8}$$

where **B** and **W** are both weight matrices associated with $H(x_i)$ and F_j respectively. Notably, h_i^0 is the input node representation x_i.

2.6.1 Graph Convolutional Network

Graph Convolutional Networks (GCN) is a generalisation of CNNs to arbitrary non-euclidean graph structures. Whereas the latter only works on grid-graph structures, GCNs are agnostic to the shape and topology of input graphs. GCNs follow the sample three-step principle outlined above but offer two key modifications contributing to their success.

Adjacency Matrix Previously, we discussed the functions of a GNN to a singular node and to extend these ideas to the entire graph, we introduce an adjacency matrix A, which encapsulates the neighbourhoods of all nodes in a graph. A is a sparse matrix, with each element $A_{ij} = 1$ if there exists an edge between nodes i and j. Or in other words, $A_{ij} = 1$ if $e_{ij} \in E$. Within a GCN, self-connections are added to ensure each node is aware of itself. In doing so, we arrive at a modified adjacency matrix, $\widetilde{A} = A + I$, where I is the identity matrix[52].

Degree Normalisation To prevent numerical instabilities, the authors introduce a degree-based normalisation for neighbourhood aggregation. The node degree matrix D is a diagonal matrix which counts the number of edges connected to the particular node. In other words, $D_{ii} = \sum_j A_{ij}$ and $\widetilde{D}_{ii} = \sum_j \widetilde{A}_{ij}$. Using this, the normalisation of the adjacency matrix is given by $\hat{A} = \widetilde{D}\widetilde{A}\widetilde{D}$.

Matrix Representation Converting the iterative update given in 8, we arrive at the equation,

$$H^{(k+1)} = f(H^k, A), \ = \sigma(AH^kW^k) \tag{9}$$

where H is a matrix generated by column-wise stacking of each node representation h_i, and f is a learnable function or an ANN. W^k is a weight matrix, akin to the weight matrices used in ANNs, for the k-th layer of the GNN.

Modification to Update Rule GCNs modify the update function given in equation 9 above by using the degree normalisation. The equation gives the combination of message passing and update steps,

$$H^{(k+1)} = \sigma(\hat{A}H^kW^k). \tag{10}$$

The benefits of GCNs arise from the fact that unlike the previous section, where W_j was localised to each node, all the graphs share the same weight matrix W [52]. By sharing these parameters, we drastically reduce the number of trainable parameters.

Channels Similar to its image-based counterpart, the CNN, GCNs contain channels as hyperparameters at each layer. These channels refer to the feature dimensions of the nodes in a graph. Essentially, the channel of the GCN at layer l is equivalent to the dimensions of the representation vector, h_i^l. Altering the channel size changes the complexity of features and relationships captured by the GCN [52].

Training our model is the same as learning the parameters of the weight matrix W^k such that the final layer H^{fin} produces the desired node representation. For us, this would produce the correct cost to traverse from the starting node to the initial node or the weightings applied to baselines. Alternatively, we can also perceive the final representation H^{fin} as the probability for the node existing in the shortest path or if it belongs to a particular subarray.

3 Shortest Path Problem

Due to the lack of prior work on combinational optimisers for radio astronomy, we attempt to reproduce the work of Vlastelica et al. [1] to demonstrate the effectiveness of combinatorial optimisers within the machine learning pipeline. We recreate the shortest path problem, which combines Dijkstra with a CNN to find the shortest path between a map's top left and bottom right corners. We can compare our implementation against the literature and set up the necessary tools to tackle the SKA subarray problem.

3.1 Data Procurement

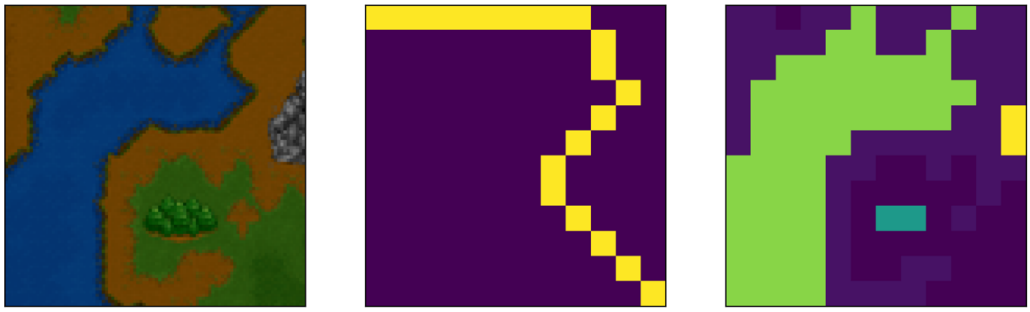


Figure 5: An example image from the dataset used. Left: Each input is a Warcraft terrain map of size 96×96. Middle: The true label for the shortest path for the image. Pixels are one hot encoded to 1/0 if they exist in the shortest path with label size 12×12. Right: Vertex weights of the image required to perfectly reproduce the label of size 12×12. Light green represents the water, dark green represents the woods, and yellow represents the mountains.

The dataset used is a collection of 12,000 randomly generated terrain maps from the Warcraft II game [53]. Data procurement proved challenging as the storage location of the researcher's custom dataset was incorrectly advertised in their GitHub repository. However, by doing a deep search of the research labs' public storage services, I was able to source the data. The dataset is split into 10000/1000/1000 training, validation and test sets. Each image is accompanied by a 12×12 image containing binary values 1/0 representing the shortest path. Fig 5 shows a terrain map and its associated label. The adjacency matrix required for Dijkstra's algorithm to output the true label is also included. Following the procurement, we follow the approach outlined in the paper by scaling the terrain maps between [0, 1]. We did not use data augmentations as the paper does not mention them. However, augmentations such as random reflections, translations, and rotations are widely used in literature as they allow models to generalise better and reduce overfitting [54]. Deviating from the paper, I also wrote a data loader that inherits from the PyTorch dataloader class [3][56]. A feature of the custom dataloader is that it converts the NumPy files into PyTorch tensors and is readily scalable to incorporate on-the-fly augmentation if desired.

3.2 Model

The model consists of a CNN-based machine learning block, a custom implementation of Djisktra's algorithm, and a gradient approximator layer. The model pipeline follows the framework in Fig 3. We choose to house these all under a torch.nn.Module class. We attempt to replicate the performance on the Warcraft dataset using the benchmark implementation of a CNN. Following the model implementation in the literature, the input data is passed onto a CNN layer with 64 channels and a kernel size 7. We pass the activation of this layer through a batch normalisation layer followed by a ReLU nonlinearity and a max pooling layer. Finally, this passes through the first layer of the ResNet-18 model before it is pooled, and a channel-wise mean is computed [57]. As Djisktra's algorithm cannot accept negative weights in the adjacency matrix, we take the absolute values of

[3] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "scikit-image: image processing in python," *PeerJ*, vol. 2, p. e453, jun 2014

the model output. A perfect model would reconstruct a matrix representing the cost of moving from one part of the terrain to another and generate the shortest path. Interestingly, we find that the implementation of Vlastelica et al. does not incorporate the pre-trained weights. Given that the first shallow layers learn task-agnostic features such as corners, edges and simple shapes, we were surprised that the authors decided not to benefit from this [48]. We omit the pre-trained weights to stay true to the reproduction and randomly initialise the model weights.

3.2.1 Implementing Dijkstra Algorithm

Due to a version mismatch in PyTorch, we deviate from the reference implementation as it inhibits a continuous gradient graph from forming. Dijkstra's algorithm is converted into a faux trainable layer with custom forward and backward propagation steps. The forward pass is the algorithm and outputs the shortest path from the top left to the bottom right corners. As we separate our gradient approximator from Dijkstra's algorithm, the backward pass of this function is simply the identity operator, backpropagating the approximated gradients. These gradients are used to update the CNN's weights. The implementation of the algorithm follows from the one found in the literature and was ported to the project by my colleague. Following this, I made the module applicable for PyTorch tensors as the original implementation used NumPy matrices. This is usually not a good idea because it can lead to issues with the gradient graph disconnecting. Additionally, the implementation found in the literature is slower as it requires more conversions to and from PyTorch tensors. A deep copy of the tensor is taken to convert between tensors format, which is computationally expensive. We removed these redundant steps using the same tensor and allowed Dijkstra to accept tensors.

3.2.2 Implementing Gradient Approximator

Our implementation of the gradient approximator is based on the torch.autograd.Function. [4]. We create a gradient approximator layer to easily add to the machine-learning pipeline given in Fig 3. The forward pass of the model does not apply any mathematical transformations but stores the output of the CNN model and the output of the Dijkstra algorithm for use in the backward pass. The backward pass follows the pseudo-code as given in the previous section. Using this approximation, we can pass gradients through the backward method of the approximator to the combinatorial optimiser and, ultimately, the trainable model.

3.3 Experimental Details

We must introduce two metrics to understand the training dynamics and evaluate our model.
Hamming Loss The Hamming distance is a measure of accuracy and a fraction of equivalent elements between two vectors u and v. The metric is the number of bits that need to be flipped $(1 \rightarrow 0, 0 \rightarrow 1)$ between the two vectors such that for all i, $u_i = v_i$.
Path Accuracy The path accuracy measures the model performance unaffected by degenerate shortest paths. The traversal cost is computed using an elementwise multiplication of the label image and the adjacency matrix. We compute the predicted cost from the model output, compare the two, and measure the accuracy.

Hyperparameter	Value
λ	20
Batch Size	70
Optimizer	Adam(learning rate=5e-4)
Epochs	50

Table 1: Model Hyperparameters for Combinatorial Resnet-18

We train the model for 50 epochs to minimise the Hamming loss between the predicted and the actual shortest path. The CNN model aims to generate the optimal adjacency matrix to reproduce the shortest path. A perfect model can accurately recognise each terrain within the image and predict travel costs from different map regions, such as green meadows, water, mountains, and hills. We explore two options for the learning rate: a flat one and a learning rate scheduler with decays. Following

[4] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," 2019

the implementation in the reference paper, we periodically drop the learning rate by dividing by ten at epochs 30 and 40. The hyperparameters of the training procedure, including the optimiser and interpolation parameter λ, are detailed in Table 1 above. All of the hyperparameters used are consistent with the reference paper.

While training, the batch-wise loss and accuracy are logged and uploaded to Weights and Biases so that we may understand the training dynamics of the model [5]. The model is evaluated every epoch on the validation set or every time it has seen the entire training set. We train our model with early stopping, where we manually stop the training if the validation loss has not decreased within the last 7 epochs, where 7 is an empirically selected hyperparameter. The model with the highest validation path accuracy is saved for inference and evaluation on the test split. We repeat this process 5 times with the same hyperparameter configuration but with different random seeds to mitigate the effects of statistical noise. Omitting this step would give us a false impression of the model's performance and provide us with the metric variance. Unfortunately, we could not perform a thorough hyperparameter sweep for our setup due to computational constraints. Ideally, we would like to use Bayesian optimisation for hyperparameter tuning using the sweeps implementation of Weights and Biases.

3.4 Results

	Training accuracy %	**Test accuracy %**
Vlastelica et al.	99.7 ± 0.0	96.0 ± 0.3
Vlastelica et al. (No Djikstra)	100 ± 0.0	23.0 ± 0.3
Warcraft CNN (scheduler)	85.4 ± 5.4	77.3 ± 2.1
Warcraft CNN (flat)	78.8 ± 8.3	75.1 ± 3.1

Table 2: Path Accuracy for shortest paths on Warcraft maps. Scheduler refers to a decreasing loss scheme [1].

As seen in Table 2, our model demonstrates a functional understanding of the task. We compare the test accuracies of the two learning rates to the "No Dijkstra" implementation, where the model is purely the CNN part. Our models significantly outperform this model, suggesting that including a combinatorial optimisation and recreation of the pipeline was a success. However, the noticeable 19% discrepancy between Warcraft CNN (scheduler) and the reference implementation highlights the need for further refinements. We also see no noticeable improvements between the two learning rates, with the results lying one standard deviation from each other. A criticism of the learning rate scheduler in the reference paper is the lack of clarity in the decay epochs selected. The learning rate is scaled by 1/10 at epochs 30 and 50. However, as there is a lack of empirical or theoretical reasoning given, we find this quite bizarre, as decay selection is uncommon in machine learning literature. We suspect this was selected empirically or tuned after multiple trials, but we can not confirm. To remedy this, we would be interested in exploring the training dynamics using the Cosine Annealing with Linear Warmup scheduler, a popular choice found in literature [60]. However, we could not explore this option due to time constraints and the added hyperparameters from the scheduler. Discrepancies between our and the reference implementation can arise for several reasons. Although care is taken to ensure the random initialisations are matched, statistical noise can contribute to poorer performance, compounded by the differences in versions of modules. Given that our model significantly outperforms the non-combinatorial version, we were sufficiently satisfied that the implementation pipeline was functional and worked as intended.

[5] L. Biewald, "Experiment tracking with weights and biases," 2020. Software available from wandb.com

4 Graph Representation of Warcraft Problem

As a stepping stone for tackling the SKA subarray problem, we reformulate the shortest path problem described in the previous section to work on graphs. We convert the image dataset to a graph dataset upon which we apply a GNN. As our problem setting is the same, we can repurpose most of the pipeline, functions and training methods used previously.

4.1 Data Procurement

An image can be interpreted as a grid graph, a special graph with regular spacing and structure akin to a square lattice. Each pixel can be considered a node, and edges connect neighbouring pixels. However, simply converting each image to a graph can lead to computational challenges with storing and processing these graphs. Given that the dimensions of our images are $[96, 96, 3]$ each, naively allocating a node to each pixel leads us to $96 \times 96 = 29216$ nodes. This poses a problem as the iterative message passing between many nodes can be computationally expensive [61] [62]. To overcome this challenge, we follow methods outlined in literature by obtaining superpixels [62]. Superpixels are a form of data aggregation for images, and combining the RGB values within the superpixel's boundaries can greatly reduce the number of nodes required.

4.1.1 Graph Conversion

Figure 6: An example image from the graph conversion used. The map is consistent with 5 Left: Raw Terrain map. Middle: The output of the SLIC algorithm. Right: Node centres and associated RAGs overlayed over the superpixels.

SLIC Using previous methods in graph-based image segmentation, we employ a Simple Linear Iterative Clustering (SLIC) algorithm based on an implementation found on sk-image [56]. Using this algorithm, we divide the RGB image into meaningful and homogeneous regions known as superpixels. This is based on a spatially localised K-Means algorithm where each pixel is assigned to a region that best matches its content. For our use case, we use the default parameters found in sk-image and take the mean of the channelwise mean of all the pixels assigned to the region, $\bar{R}\bar{G}\bar{B}$ as a form of superpixel aggregation. The image pre- and post-processing illustration is shown in the left and middle of Fig. 6. We can clearly see most of the details are captured by SLIC. Some finer details, such as the texture of stones and trees, are omitted, but these details should not affect performance. Additionally, we treat the location of each centroid as obtained by SLIC as a node with the average colour as its feature.

RAG To convert the centroids to a connected graph, we used the graph.rag_mean _colour function as found in sk-image [6][56]. Nodes are connected if they contain a boundary region between them. We can see the regional adjacency boundaries, region centroids and the original image on the right and left of Fig 6. We run the RAG algorithm with an infinite threshold, a hyperparameter that can be selected to connect all edges. Following this, we assign a unique label (increasing monotonically from zero onwards) to each region to identify whether a node participates in the shortest path. The

[6] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "scikit-image: image processing in python," *PeerJ*, vol. 2, p. e453, jun 2014

output of this process is a NetworkX [7] graph; hence, we can readily convert it to a PyTorch Geometric graph for later use.

4.1.2 Label Conversion

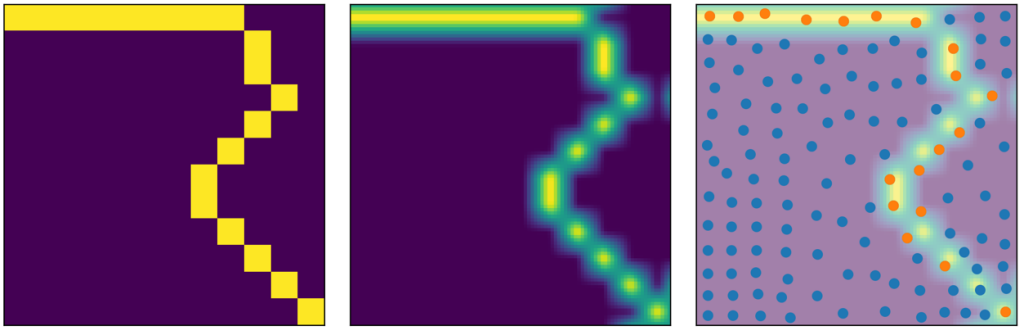


Figure 7: An example image from the label conversion of the map found in5 Left: Raw input label. Middle: Upscaled label image to 96×96, displays artefacts. Right: Centroids and node positions, with orange nodes denoting the path.

We also convert the true path into a graph structure, storing a boolean variable of whether a node in the input graph belongs to the shortest path, with a 1 denoting a positive presence. We resize the 12×12 image of the label (see previous section) into $96 \times 96 \times 1$ image using the scikit-image.transform.resize function. This was done so that we may overlap the locations of centroids with the label image. My colleague wrote a function that finds the b_p brightest pixels, where b_p is a hyperparameter tuned to 20%. Following this, euclidean distances are computed from the pixel coordinates to the centroids, and the nearest centroids are assigned to this pixel. We label this node as belonging to the path. To manually tune b, we visually inspect a set of randomly sampled images (20) and the corresponding label. 20% ensured we did not over-label the number of nodes in the path while generating a sensible path. We believe this method, alongside the graph conversion, introduces the most errors in our pipeline. As shown in the central image in Fig 7, this methodology introduces artefacts, with some pixels not following the binary $1/0$ labelling. Although we mitigate this effect by taking the top b brightest pixels, we suspect some of these artefacts trickle through. A proposed solution is using the centroid positions. These positions exist in the $[0-96, 0-96]$ range due to the resolution of our original image; rescaling these coordinates via a linear transformation to match the size of the label image could potentially reduce the artefacts. We suspect that our method for label generation is flawed as the output of Dijkstra might be able to find shorter paths than the one we have given as labels. Visually, we can see this in the rightmost image found in Fig 7. There are nodes we would assume are on the path in the bottom right, but the algorithm does not label it as such. Alternatively, we could also introduce an added constraint on label generation by restricting the number of nodes that can exist in a path, which may alleviate this problem.

4.1.3 Dataloader

After transforming the data into the correct format, I converted these images into a graph-based dataset. Inheriting from the "InMemDataset" class in PyTorch Geometric, I implement a custom version of a dataset [8]. The benefits of using this dataset class are that it is highly customisable and can contain the code necessary for processing the images within it. We store all the graphs within a split (train/val/test) as a collection of disconnected graphs instead of individual files for each image. By doing this, we optimise storage space with approximately 10x lower total file size. Lowering the size of the dataset also means we can load it entirely onto memory, significantly reducing the GPU wall time for loading and unloading objects to and from the random access memory. Upon processing

[7] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008

[8] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019

the graphs, we store the splits separately so that we may use them again without reprocessing. The processing process takes approximately 18 hours on an A100 GPU with 80 GB of VRAM. The dataloader is based on the default implementation found in PyTorch Geometric. Mimicking the hyperparameter selection in the previous section, we set the batch size to 70. This approach converts the image dataset to an iterable graph-based dataloader compatible with our previous pipeline and a GNN.

4.2 Model

Instead of using the custom Dijkstra algorithm found in the previous section, we use a pre-written function in the NetworkX library [9]. We pass the graph structure as input, and the output is the indices of nodes within the shortest path. We convert these indices into a one-hot encoded vector of size n where n is the number of nodes in the input graph. This output is compatible with the Hamming loss function and the rest of the pipeline. The model used is derived from the GCN class found in PyTorch Geometric. We elect to use 3 GCN layers with [128,128,128] channel dimensions. Following each GCN layer, we add a non-linearity, ReLU and batch normalisation. These layers are imported from the PyTorch Geometric library and used without modification [10]. The same model pipeline is used as defined in the previous section.

4.3 Experimental Details

The experimental setting is identical to the previous section, including the hyperparameters found in 1. An exception is the learning rate scheduler. We only trained our models using the flat scheduler. The validation set at the end of every epoch evaluates the model's performance, and weights were saved corresponding to the lowest validation loss. Contrary to the previous section, we could not compute the paths' accuracy. This is because we could not accurately generate the vertex weights, unlike the previous section where the true adjacency matrix was provided.

4.4 Result

Unfortunately, there is no accuracy metric associated with our results. As our model does not converge, see appendix A, reporting a metric is not meaningful. There are a range of issues that we suspect may be the cause. Data corruption is the most significant source of error within our pipeline. This is because we heavily approximate not only the position of nodes but also the label. If the label is wildly different from Djikstra's prediction, we would misidentify this as not being the shortest path, even if a valid path is found. This error would propagate backwards via gradient descent, leading to the model not training. Another potential issue is that we may need to sample more of the hyperparameter space. Given that each iteration of the model can take upwards of 18h+, the computational constraints mean we could not rigorously explore and finetune these hyperparameters. A debugging scenario we would be interested in exploring further is overfitting the model to a minimal number of samples. Theoretically, a sufficiently parameterised GNN can overfit and memorise a small sample. However, being unable to overfit does not necessarily point to issues with the code itself, as it may be due to the input graph and label generation.

5 Discussion

We successfully reproduced a pipeline for combining combinatorial optimisers and CNNs for the shortest path problem in the Warcraft dataset. Our path accuracy of 77.3 ± 2.1 suggests that injecting combinatorial optimisers to a machine learning pipeline can significantly improve results when compared to the 23.0 ± 0.3 accuracy reported by Vlastelica et al. for a purely machine learning-based pipeline [1]. However, we could only partially reproduce the results of Vlastelica et al. using identical hyperparameters and setups. This may be due to a mismatch in versions of PyTorch or due to several potential reasons that require further investigation. Reproducing the implementation found in the

[9] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008

[10] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019

paper also proved challenging. A reason for this is the drastic changes to PyTorch and the autograd engine since the implementation given by the authors was published. Attempts to reuse the code in the associated GitHub repository were difficult, with many functions being deprecated. Procuring the data was also challenging, with the location advertised by the authors being inaccessible. We were also surprised by the lack of empirical or theoretical analysis surrounding hyperparameter selection. Specifically, we were unsure how the learning rate scheduler was selected. We are unaware if the authors thoroughly searched the parameter space. Albeit not a direct criticism of the methods, the lack of examples outside of the computer vision space means the framework proposed by the authors is also untested in other domains, including implementations of graph-based problems. Porting the shortest path problem to a graph-based GNN proved unsuccessful. We offer two major hypotheses for its failure. Our graph representation of the terrain maps was unfaithful due to the numerous approximations we made during data procurement. The label generation needs to be more accurate due to the artefacts introduced. This leads to corrupt labels, which affect the gradients used for training, and, ultimately, the model fails to converge.

6 Conclusion

Using the pipeline we produced for the Warcraft shortest path problem, we are ready to tackle the SKA sub-array problem. Using the coordinates of each antenna in the SKA, we can generate a graph structure for training. A potential combinatorial optimiser of interest is spectral clustering, which segments the graph into subgraphs based on the local structures of the graph [65]. Randomly initialising the node features to two scalars, we can train a GNN model to correctly identify the baseline weightings associated with each node and the probabilistic classification of a node to a specific subarray. An avenue for further exploration is replacing the linear interpolation used in the gradient approximator. Inspired by recent works, we are interested in using the concrete distribution with learned parameters to mimic the piece-wise step function found in the discrete loss landscape [66]. Additionally, there is growing research in the applications of Reinforcement Learning for combinatorial problems, which could be a promising avenue of research [67]. GNN-based algorithmic reasoners are also an idea we would like to explore, as they have enjoyed immense success in approximating combinatorial optimisers. Using this method, we could omit the combinatorial optimiser entirely and bypass the problems associated with a discrete loss function [27].

References

[1] M. Vlastelica, A. Paulus, V. Musil, G. Martius, and M. Rolínek, "Differentiation of blackbox combinatorial solvers," 2020.

[2] C. Leng, Z. Tang, Y.-G. Zhou, Z. Tian, W.-Q. Huang, J. Liu, K. Li, and K. Li, "Fifth paradigm in science: A case study of an intelligence-driven material design," *Engineering*, vol. 24, pp. 126–137, 2023.

[3] J. Jumper, R. Evans, A. Pritzel, *et al.*, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, pp. 583–589, Aug. 2021.

[4] R. Lam, A. Sanchez-Gonzalez, M. Willson, *et al.*, "Graphcast: Learning skillful medium-range global weather forecasting," 2023.

[5] J. VanderPlas, A. J. Connolly, Z. Ivezic, and A. Gray, "Introduction to astroml: Machine learning for astrophysics," in *2012 Conference on Intelligent Data Understanding*, IEEE, Oct. 2012.

[6] A. R. Thompson, J. M. Moran, and G. W. Swenson, *Interferometry and synthesis in radio astronomy*. Springer Nature, 2017.

[7] C. L. Bennett, D. Larson, J. L. Weiland, *et al.*, "Nine-year Wilkinson Microwave Anisotropy Probe (WMAP) Observations: Final Maps and Results," , vol. 208, p. 20, Oct. 2013.

[8] Event Horizon Telescope Collaboration, "First M87 Event Horizon Telescope Results. II. Array and Instrumentation," , vol. 875, p. L2, Apr. 2019.

[9] S. Yatawatta, "Adaptive weighting in radio interferometric imaging," *Monthly Notices of the Royal Astronomical Society*, vol. 444, pp. 790–796, 08 2014.

[10] SKA Observatory, "Skao: Home of the square kilometre array," 2024. Accessed: January 2024.

[11] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency*. No. v. 1 in Algorithms and Combinatorics, Springer, 2003.

[12] A. Hobé, D. Vogler, M. P. Seybold, A. Ebigbo, R. R. Settgast, and M. O. Saar, "Estimating fluid flow rates through fracture networks using combinatorial optimization," *Advances in Water Resources*, vol. 122, pp. 85–97, 2018.

[13] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver, *Combinatorial Optimization*. USA: John Wiley & Sons, Inc., 1998.

[14] A. Paler, A. Zulehner, and R. Wille, "Nisq circuit compilation is the travelling salesman problem on a torus," *Quantum Science and Technology*, vol. 6, no. 2, p. 025016, 2021.

[15] R. G. Bland and D. F. Shallcross, "Large travelling salesman problems arising from experiments in x-ray crystallography: A preliminary report on computation," *Operations Research Letters*, vol. 8, no. 3, pp. 125–128, 1989.

[16] D. Baron, "Machine learning in astronomy: A practical overview," *arXiv preprint arXiv:1904.07248*, 2019.

[17] I. N. Pashchenko, K. V. Sokolovsky, and P. Gavras, "Machine learning search for variable stars," *Monthly Notices of the Royal Astronomical Society*, vol. 475, pp. 2326–2343, 12 2017.

[18] E. C. Vasconcellos, R. R. de Carvalho, R. R. Gal, F. L. LaBarbera, H. V. Capelato, H. Frago Campos Velho, M. Trevisan, and R. S. R. Ruiz, "Decision tree classifiers for star/galaxy separation," *The Astronomical Journal*, vol. 141, p. 189, May 2011.

[19] M. Bowles, A. M. M. Scaife, F. Porter, H. Tang, and D. J. Bastien, "Attention-gating for improved radio galaxy classification," *Monthly Notices of the Royal Astronomical Society*, vol. 501, p. 4579–4595, Dec. 2020.

[20] A. M. M. Scaife and F. Porter, "Fanaroff–riley classification of radio galaxies using group-equivariant convolutional neural networks," *Monthly Notices of the Royal Astronomical Society*, vol. 503, p. 2369–2379, Feb. 2021.

[21] S. S. Bhat, T. Prabu, B. Stappers, A. Ghalame, S. Saha, T. S. B. Sudarshan, and Z. Hosenie, "Investigation of a machine learning methodology for the ska pulsar search pipeline," *Journal of Astrophysics and Astronomy*, vol. 44, no. 1, 2023.

[22] S. Hassan, S. Andrianomena, and C. Doughty, "Constraining the astrophysics and cosmology from 21cm tomography using deep learning with the ska," *Monthly Notices of the Royal Astronomical Society*, vol. 494, p. 5761–5774, May 2020.

[23] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, no. 2, pp. 405–421, 2021.

[24] A. Bogyrbayeva, T. Yoon, H. Ko, S. Lim, H. Yun, and C. Kwon, "A deep reinforcement learning approach for solving the traveling salesman problem with drone," 2022.

[25] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[26] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell, "Neural execution of graph algorithms," 2020.

[27] P. Veličković and C. Blundell, "Neural algorithmic reasoning," *Patterns*, vol. 2, no. 7, 2021.

[28] B. Burke and F. Graham-Smith, *An Introduction to Radio Astronomy*. Cambridge University Press, 2010.

[29] R. Jennison, "A phase sensitive interferometer technique for the measurement of the fourier transforms of spatial brightness distributions of small angular extent," *Monthly Notices of the Royal Astronomical Society*, vol. 118, no. 3, pp. 276–284, 1958.

[30] F. Zernike, "The concept of degree of coherence and its application to optical problems," *Physica*, vol. 5, no. 8, pp. 785–795, 1938.

[31] P. H. van Cittert, "Die Wahrscheinliche Schwingungsverteilung in Einer von Einer Lichtquelle Direkt Oder Mittels Einer Linse Beleuchteten Ebene," *Physica*, vol. 1, pp. 201–210, Jan. 1934.

[32] E. Estrada, "Graph and network theory in physics," *arXiv preprint arXiv:1302.4378*, 2013.

[33] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pp. 287–290, 2022.

[34] N. Mamano, "Implementing dijkstra's algorithm." `https://nmamano.com/blog/dijkstra/dijkstra.html`, 2020. Accessed: 04/01/2024.

[35] D. Rachmawati and L. Gustin, "Analysis of dijkstra's algorithm and a* algorithm in shortest path problem," in *Journal of Physics: Conference Series*, vol. 1566, p. 012061, IOP Publishing, 2020.

[36] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, Oct. 1986.

[37] R. Bellman, R. Corporation, and K. M. R. Collection, *Dynamic Programming*. Rand Corporation research study, Princeton University Press, 1957.

[38] A. Brutzkus and A. Globerson, "Why do larger models generalize better? a theoretical perspective via the xor problem," 2019.

[39] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[40] D. Liang, D. A. Frederick, E. E. Lledo, N. Rosenfield, V. Berardi, E. Linstead, and U. Maoz, "Examining the utility of nonlinear machine learning approaches versus linear regression for predicting body image outcomes: The u.s. body project i," *Body Image*, vol. 41, pp. 32–45, 2022.

[41] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[42] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, no. 1, pp. 145–151, 1999.

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[44] S. Sun, Z. Cao, H. Zhu, and J. Zhao, "A survey of optimization methods from a machine learning perspective," 2019.

[45] T. Cohen and M. Welling, "Group equivariant convolutional networks," in *International conference on machine learning*, pp. 2990–2999, PMLR, 2016.

[46] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights into imaging*, vol. 9, pp. 611–629, 2018.

[47] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[48] J. Plested and T. Gedeon, "Deep transfer learning for image classification: a survey," 2022.

[49] A. Daigavane, B. Ravindran, and G. Aggarwal, "Understanding convolutions on graphs," *Distill*, 2021. https://distill.pub/2021/understanding-gnns.

[50] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko, "A gentle introduction to graph neural networks," *Distill*, 2021. https://distill.pub/2021/gnn-intro.

[51] C. Niu, Y. Song, J. Song, S. Zhao, A. Grover, and S. Ermon, "Permutation invariant graph generation via score-based generative modeling," 2020.

[52] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[53] J. Guyomarch, "Warcraft ii open-source map editor." `https://github.com/war2/war2edit`, 2017.

[54] S. C. Wong, A. Gatt, V. Stamatescu, and M. D. McDonnell, "Understanding data augmentation for classification: when to warp?," 2016.

[55] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "scikit-image: image processing in python," *PeerJ*, vol. 2, p. e453, jun 2014.

[56] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk, "Slic superpixels compared to state-of-the-art superpixel methods," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 11, pp. 2274–2282, 2012.

[57] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[58] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[59] L. Biewald, "Experiment tracking with weights and biases," 2020. Software available from wandb.com.

[60] A. Gotmare, N. S. Keskar, C. Xiong, and R. Socher, "A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation," *arXiv preprint arXiv:1810.13243*, 2018.

[61] F. Monti, D. Boscaini, J. Masci, E. Rodolà, J. Svoboda, and M. M. Bronstein, "Geometric deep learning on graphs and manifolds using mixture model cnns," 2016.

[62] P. H. C. Avelar, A. R. Tavares, T. L. T. da Silveira, C. R. Jung, and L. C. Lamb, "Superpixel image classification with graph attention networks," 2020.

[63] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

[64] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[65] F. M. Bianchi, D. Grattarola, and C. Alippi, "Spectral clustering with graph neural networks for graph pooling," 2020.

[66] C. J. Maddison, A. Mnih, and Y. W. Teh, "The concrete distribution: A continuous relaxation of discrete random variables," 2017.

[67] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, "Reinforcement learning for combinatorial optimization: A survey," 2020.

A Training Dynamics

We explore the training dynamics of each model via the use of the loss function. Visualising the loss landscape gives us valuable information about the performance of each model.

A.1 Training Curve for CNN Based Implementation

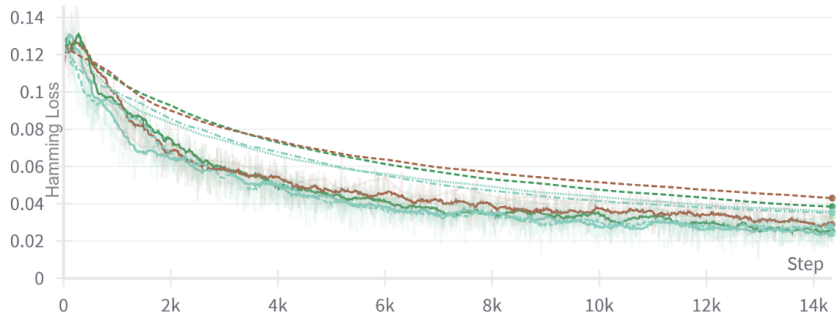


Figure 8: The training loss for the image-based implementation of the shortest path problem against current training iteration. Each of the curves represents different hyperparameters and training instances. Curves are smoothed for increased visibility.

As shown in Fig 8, the loss monotonically decreases for all models. This suggests that the model can identify and learn from features.

A.2 Training Curve for GNN Based Implementation

Figure 9: The training loss for the graph-based implementation of the shortest path problem against current training iteration. Each of the curves represents different hyperparameters and training instances. Curves are smoothed for increased visibility.

As shown in Fig 9, none of the hyperparameters yielded a model that displayed a decreasing loss. In contrast to the training curve found in 8, there is no decrease in the overall loss with respect to the number of parameter updates. This suggests the model is not learning any important features from the input graph, so our data generation procedure is flawed.