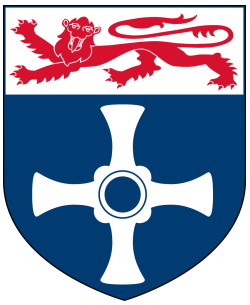
Neural Architecture Search Across Expanded and Infinite Spaces

Efficient Design of Optimal Neural Networks



Robert Joao Geada

Supervisors: A.S. McGough
D. Prangle

School of Computing
Newcastle University

This dissertation is submitted for the degree of
Doctor of Philosophy

September 2022

Abstract

Neural networks are incredibly powerful tools for a variety of different situations. However, their development can be difficult, time-consuming, and expensive. To address this, the field of Neural Architecture Search (NAS) seeks to provide automated algorithms to produce optimal network designs. However, recent criticism has been levied towards these algorithms regarding their performance compared to a purely random search strategy. Additionally, these algorithms themselves require a significant amount of configuration, limiting their ability to ease the costs of network design. To examine these criticisms BonsaiNet is presented, a NAS algorithm that operates over a significantly broadened search space that is a superset to those used by other leading NAS algorithms. This broadened search space lowers the average quality of random networks in the space, while preserving high-quality networks to be potentially discovered by NAS. Indeed, BonsaiNet still produces networks competitive with the state-of-the-art, indicating that random search is only competitive with NAS in over-constrained search spaces. Furthermore, BonsaiNet employs a *large-cell* design pattern which eliminates the need to specify the count or types of each individual cell in the model, thus significantly reducing the necessary configuration. To further examine the random search and configuration concerns, SpiderNet is presented, a NAS algorithm that dynamically evolves from a minimal initial state within an infinitely large search space. This transfers the burden of determining network size and macro-level connectivity patterns of the networks from the user to the algorithm, drastically reducing the amount of configuration necessary to provide good results. Indeed, despite the infinite search space and minimal configuration, SpiderNet produces highly competitive models. Furthermore, it consistently produces more time and parameter efficient models than random search, indicating two new dimensions by which NAS can have an advantage over random search. As such, BonsaiNet and SpiderNet demonstrate that random search's comparable ability to NAS is an illusion produced by both an over-constrained search space and a disregard for time and parameter efficiencies. Additionally, both algorithms provide a strong proof-of-concept towards a minimal-configuration NAS algorithm.

This thesis is dedicated to my family; my Dad for being a lifetime source of scientific inspiration and full-time sanity checker, my Mum for always providing unconditional love and support, and my sister who would never let me live it down if she got her PhD before me.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Robert Joao Geada
September 2022

Acknowledgements

I would foremost like to acknowledge the contributions of my supervisors to this work; they were ever-present to focus my thoughts into concrete ideas. Their guidance was crucial in shaping this work's over-arching aims, helping me develop a cohesive argument from a mountain of experimental results as well as identify the directions that new experiments needed to take. Second, I would like to thank Red Hat, for providing the funding by which this project came to fruition. Red Hat offered me a flexible support structure and treated my PhD research with equal respect to my work responsibilities, for which I am forever grateful. Finally, I would like to acknowledge Paul Watson, Jennifer Wood, and the rest of the CDT family for creating the supportive and inquisitive environment that encouraged me to push myself to the fullest.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Neural networks are immensely powerful. They power self-driving cars (Karpathy, 2021), can outperform doctors at spotting cancer (Walsh, 2020), are the best chess players in history (Silver et al., 2018), and can write articles indistinguishable from those of human writers (Brown et al., 2020). They can become experts in these tasks without requiring their designers to have any expertise themselves; AlphaZero's neural networks are the greatest chess players in history yet need only the basic rules of the game as inputs in their path to dominance (Silver et al., 2018). Neural networks also excel at finding subtle patterns within massive datasets, ones so nuanced and at such scale as to be imperceptible to human eyes, and can use them to discover better ways of solving problems. In theory, all of these talents are out-of-the-box features of neural networks, and as such make neural networks massively appealing as solutions to a huge variety of different problem domains.

In practice, however, the performance of these networks are greatly affected by their configurations; that is, everything from how they are trained to their internal wiring schemes drastically affect a network's abilities. Tuning these configurations is a difficult process on a number of levels. First, and perhaps most challenging, is that neural networks are 'black boxes', devices whose internal workings are so complex as to be functionally impossible to comprehend in any meaningful way. That means the effects of any configuration change are unlikely to be entirely predictable, and thus must be experimentally tested. While a general intuition can be built up from years of experience, this can often be pretty domain-specific and not particularly generalizable, at least in my own (painful) experience. As such, the design directions that should be taken when trying to improve a network are often complete mysteries, and thus design is mostly a trial-and-error process.

The second problem, and one that directly compounds the first, is that evaluating the performance of a neural network takes a lot of time and compute power, both of which are expensive. For reference, a total of 3,999 individual neural networks were evaluated and recorded as part of this work, comprising a grand total of 451 days, 6 minutes, and 23 seconds of graphics processing unit (GPU) time. Of that, 228 days were taken by just the 100 networks most rigorously evaluated, an average of 55 hours each. Prohibitive time costs aside, for those without access to a dedicated GPU this work is either fundamentally impossible or at least very expensive: renting an Azure

cloud instance of sufficient power for the same duration would cost over £20,000 (Microsoft Azure, 2021). Even with access to a personal GPU it is not particularly cheap, considering that the GPU itself can cost over £1,500 and the cumulative energy cost to run it for 451 days would be around £1,700, given current average electricity costs (UK Department for Business, Energy, and Industrial Strategy, 2021). All of this means that the amount of time that can be spent playing with different network configurations is limited for the average user.

Finally, the performance that is possible for any particular problem is often an unknown quantity. If a network can perform some arbitrary task at 75% accuracy, is that the best possible performance? Would another configuration score 80%? Without spending a huge amount of time (and therefore money) exploring different candidate networks and configurations, this is an unknown quantity, and another factor that makes designing and tuning neural networks very difficult. For example, suppose that same example network can perform that arbitrary task at 75% accuracy, and every change tested so far has only been detrimental to performance, is that network already at peak possible performance or were they just bad changes?

All of this makes the promised performance and capability of neural networks feel a bit like a siren song; you're lured in by tales of their unbelievable, superhuman abilities, but instead drown in years of architecture tuning and bills from your cloud provider. Unfortunately, this also means that the amazing power of neural networks is often wielded by those with near limitless financial and temporal resources to throw at network design; the four examples of incredible networks given in the first paragraph are owned by Tesla, Google, DeepMind, and Microsoft, respectively. Indeed, this thesis will explore neural networks and algorithms published by these companies that require over 60 *years* of compute time to achieve their results.

In order to democratize the potential of neural networks, to allow the average user to define the state-of-the-art, the barrier to entry must be lowered. One way to do this is by eliminating the need for configurational tuning as best as possible, by discovering means of automatically designing optimal neural networks. Research towards this goal constitutes the field of Neural Architecture Search (NAS), which has seen a lot of success in the last few years: a NAS generated network can rival (if not entirely out-compete) the best human designed networks, while only needing a few GPU-hours of design time. Despite this success, there are a few concerns that constitute the specific focus of this work, the details of which are described in the following sections.

1.1 Deconstricted Search Spaces: BonsaiNet

In general, a NAS algorithm comprises of two parts: a search space and a search algorithm. The former is the set of all possible networks that the algorithm can design, while the latter is the means by which the algorithm selects a network from that space. A recent concern in the field is that a random search algorithm can perform equally well to state-of-the-art algorithms when operating over the same search space; that is, a randomly picked network from the search

space[1] is often just as good as one that is 'intelligently' designed by a NAS algorithm. At face value, this appears to indicate that existing NAS algorithms are poor, considering that they are extraordinarily expensive means of selecting a network that is apparently no better than one pulled out of a hat.

However, there is a potential explanation for this phenomenon, and that revolves around potential over-restriction of the network search spaces. Often, NAS algorithms are evaluated by how well the networks they produce perform over a set of standard benchmark tasks. However, there is an extensive body of literature describing how to design excellent models for these exact benchmarks. How many of these best-practices have contaminated the search spaces that NAS algorithms operate over? If the search space is exclusively full of good networks because of this contamination, then regardless of whether a network is selected intelligently or randomly from that set it will be good. Therefore, in such a space, a flawless NAS algorithm will be indistinguishable from random search.

The Bonsai search space was designed to explore this idea. The Bonsai search space is drastically larger and less constrained than any seen in the literature discussed in this work, while remaining a superset of existing search spaces. Amongst this set of less-constrained networks there is greater potential for bad networks to exist, while still preserving all the good networks that were present in the original constrained spaces. In terms of network quality, networks in the less-constrained Bonsai search space will be of lower average quality than in the constrained space, but with higher variance. This will have the effect of lowering the quality of any randomly selected network from the space, creating a bigger difference between average networks and good ones. This difference should then make it obvious if a NAS algorithm is truly selecting good networks. Additionally, this presents a much more realistic starting emulation of NAS' real world application to novel problems, wherein the research required to specifically tailor the search space does not exist.

However, the constrained search spaces of many NAS algorithms are not simply an oversight or accidental bias, but rather a design feature to avoid hardware limitations. As such, most existing algorithms in the literature would be unable to operate in the Bonsai search space. To examine this BonsaiNet was built, a search algorithm designed from the ground up to operate in the much larger, less-constrained search space. BonsaiNet is a supernet-based NAS algorithm that builds models progressively, wherein models are broken down into contiguous groups of cells called *sections*. Searching occurs via differentiable pruners along its network edges which let the model select any arbitrary mixture of operations as opposed to the traditional single-path operational mixture frameworks seen in algorithms like DARTS (Liu et al., 2018). Since searching and pruning occur as the same operation, the act of searching for a model section implicitly creates room for subsequent sections to be appended to the model. The growing and pruning process repeats until the model is fully grown, at which point the model is fully searched. This process allows unique per-cell architectures, which creates significantly greater network heterogeneity as compared to the cellular duplication approach common in NAS literature. .

[1]Throughout this thesis and through much of NAS work, random search often refers to the concept of replacing every 'intelligent' decision made by a NAS algorithm with a random uniform sampling over the available choices.

BonsaiNet is then evaluated over a variety of different conditions to determine whether the over-constriction hypothesis is correct. This work on the Bonsai search space and BonsaiNet was presented as a paper at CVPR-NAS 2020 (Geada et al., 2020).

1.2 Minimizing Configuration: SpiderNet

In the process of experimenting on BonsaiNet, as well as in the general examination of other NAS algorithms, a particularly troubling trend became apparent: rather than alleviate the need for manually tuning network designs, BonsaiNet and other NAS algorithms were guilty of merely shifting that tuning burden to the configuration of the NAS algorithm itself. While these algorithms excelled at producing good networks within a search space, choosing an appropriate search space was implicitly a configuration decision that these algorithms could not make. Specifically, those NAS algorithms were capable of making micro-level design choices like individual operational choices at certain points in the model, but macro-level design choices like overall size, number of internal connections, and complexity of the connectivity were fixed across the entire search space. All of those macro-level decisions were still the user's responsibility to design. For example, the DARTS (Liu et al., 2018) algorithm uses a graph structure to represent its networks, and can select which operation should exist along certain edges of the graph. However, it does not determine the edge count, node count, or node degree of the graph itself, instead leaving those decisions to the user. This is a much smaller responsibility than designing a network entirely from scratch, but is still broad enough that exploring every option is entirely unfeasible. How could NAS claim to be a comprehensive means of automatically building networks, if the user still needs to design the overarching frameworks that define the shape of those networks? As a reductive analogy, NAS algorithms were claiming to be artists, but were instead merely crayoning in the coloring books we have designed for them.

To explore these concerns a search algorithm that provided its own search space was designed, one that could dynamically expand or constrict the space as it saw fit. This work became SpiderNet, an algorithm that uses a variety of heuristics to guide its traversal through an infinite search space. This infinite space is rarely seen in literature, as most NAS algorithms instead operate within a finite search space bounded by the macro-level design decisions provided by the user. To avoid this configuration, SpiderNet starts from minimal initial conditions before it mutates itself into more complicated structures, which minimizes the amount of manual tuning of the space required. All that is required is to specify the conditions of the initial trivial seed, which is an implicitly smaller set of decisions due to its small size. After designing SpiderNet, a variety of experiments were conducted to evaluate various search-guiding heuristics, as well as how these heuristics compare against a randomly-guided search.

1.3 Contributions

This thesis has produced the following papers and contributions:

- **BonsaiNet:** A NAS algorithm that successfully operates over a significantly expanded search space and demonstrates NAS' ability to exceed random search in such spaces, published as a workshop paper in CVPR-NAS 2020 (Geada et al., 2020). A follow-up paper detailing subsequent results and further analysis is planned upon completion of this thesis.
- **SpiderNet:** A minimally-configured NAS algorithm that operates over an infinite search space, and demonstrates NAS' ability to outperform random search in a number of different, previously unconsidered metrics. A condensed version of the SpiderNet chapter of this thesis (Chapter 5) was published as a workshop paper in CVPR-NAS 2022 (Geada and McGough, 2022).

Additionally, the following research outcomes were produced. While these are not directly related to the work within this thesis, the experience gained from this work is what made them possible.

- **CVPR-NAS 2021 Unseen Data Competition:** A NAS competition that evaluated how well NAS algorithms perform on entirely unseen datasets and tasks. I led the design, management, and public presentation of the competition. Entries came from 71 teams from around the world, and the competition was eventually won by Samsung Research (Samsung R&D Institute China, 2021).
- **CVPR-NAS 2022:** From the success of the 2021 competition, our team was chosen to organize the upcoming 2022 CVPR-NAS workshop. The second annual Unseen Data Competition will be held here, and again I will be writing the evaluation scripts and designing the competition datasets.

1.4 Access

All code described in this thesis is available at `https://github.com/RobGeada`, specifically:

- **BonsaiNet:** `https://github.com/RobGeada/bonsai-net`
- **SpiderNet:** `https://github.com/RobGeada/spidernet`
  and `https://github.com/RobGeada/spidernet-NTKLRC`

1.5 Outline

This work will first explore the fundamental background of neural networks in Chapter 2, such as to provide the foundation that the rest of the work will stand upon. This will cover their mathematics, the practical resources and techniques used to extract performance out of them, the advanced components that go into more specialized network design, and the datasets and tasks used to benchmark their abilities. Next, Chapter 3 will explore the existing literature in both state-of-the-art network design and Neural Architecture Search, before discussing some missing pieces that warrant further examination. These lead to Chapters 4 and 5 which discuss BonsaiNet

and SpiderNet respectively, specifically, their objectives, design, and evaluatory experiments. Finally, Chapter 6 summarizes the results and discusses their ramifications for the field as a whole.

Chapter 2

Background

2.1 Introduction

In this chapter the basic foundation of computer vision models will be presented, starting from the basics of neural networks in Section 2.2. From there, the general techniques used to train neural networks will be covered in Sections 2.3 through 2.5, along with more specific tools that can be used to extract higher performance. With the foundations of simple networks and network training laid, Sections 2.6 through 2.7 discuss the more complex, computer vision-specific operations, as well as how they combine to form convolutional neural networks. Additionally, the benchmark datasets used commonly in computer vision research are introduced in Section 2.8. Finally, Section 2.9 describes the software and hardware necessary to work with these kinds of neural networks. This all serves to provide the conceptual background required to discuss the more specific literature that appears in the subsequent chapter.

2.2 Neural Networks

2.2.1 Neuron

The absolute foundation of neural networks is the artificial neuron. In the simplest case, an artificial neuron takes a vector of inputs $[x_0, x_1, \ldots, x_n]$ and computes:

$$y = f\left(\sum_{i=0}^{n} \theta_i x_i\right) \tag{2.1}$$

where $\vec{\theta}$ is a vector of size n that controls the weighting of each input to the neuron and f is referred to as the *activation function* (Anthony, 2001). Also common is the addition of a bias b, which allows the neuron to represent a y-intercept within the internal sum:

$$y = f\left(b + \sum_{i=0}^{n} \theta_i x_i\right) \tag{2.2}$$

This activation function f varies depending on the desired properties of the neuron, but is typically a nonlinear function that maps $\mathbb{R} \to \mathbb{R}$ or some subset of $\mathbb{R}$ (more on activation functions in Section 2.6.2). The real magic of the artificial neuron is the weight vector $\vec{\theta}$ and the bias b, the values of which can adapt or 'learn' to better serve the needs of the neuron. These weights are referred to as the *learnable parameters* of the neuron, and the process of learning these weights is referred to as *training* which will be explored in more depth shortly. The more inputs to a neuron, the more learnable parameters it needs to have, which comes at the expense of computational cost and training complexity.

2.2.2 Feedforward Networks

If m neurons are stacked in parallel such that all receive the same n inputs $[x_0, x_1, \ldots, x_n]$, their joint output can be expressed as:

$$\vec{y} = f(\theta\vec{x} + \vec{b}) \tag{2.3}$$

$$\begin{pmatrix} y_1 \\ y_2 \\ \ldots \\ y_m \end{pmatrix} = f\left(\begin{pmatrix} \theta_{11} & \theta_{12} & \ldots & \theta_{1n} \\ \theta_{21} & \theta_{22} & \ldots & \theta_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ \theta_{m1} & \theta_{m2} & \ldots & \theta_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_m \end{pmatrix} \right), \tag{2.4}$$

where the activation function f is applied elementwise to the vector. This formulation is called a fully-connected layer, which for n inputs and m neurons has an mxn matrix of learnable weights and m learnable biases. By composing these layers together into larger *feedforward networks*, such that the outputs of one layer are sent to the inputs of subsequent layers, the relatively simple function approximations of a single neuron can be built into much more complex functions. For such a stacking of L layers, the network output is:

$$\begin{aligned} \vec{y_0} &= f_0(\theta_0\vec{x} + \vec{b_0}) \\ \vec{y_1} &= f_1(\theta_1\vec{y_0} + \vec{b_1}) \\ &\ldots \\ \vec{y_L} &= f_L(\theta_L\vec{y_{L-1}} + \vec{b_L}) \end{aligned}$$

In fact, with enough neurons per layer, these *feedforward networks* become Universal Approximators (Stinchcombe and White, 1989), that is, capable of approximating any arbitrary function that maps $\mathbb{R}^{n_{in}}$ to $\mathbb{R}^{n_{out}}$, where n_{in} is the number of network inputs and n_{out} is the number of network outputs. However, the larger the network, the more computational power and time necessary to calculate the network output $\vec{y}$ for some input $\vec{x}$. Additionally, the number of learnable parameters is $\sum_{l=1}^{L}(m_l * m_{l-1} + m_l)$, where m_l is the number of neurons in layer l and m_0 is the number of network inputs n_{in}.

With this, the simplest forms of neural networks have been outlined. Before discussing any more advanced network design concepts, it is first necessary to explore how these networks are actually trained.

2.3 Supervised Learning

As shown in Section 2.2.2, neural networks are essentially a function $f(x,\theta)$ that, given some input x and its learnable parameters θ, approximates some desired function F. The question is, how should θ be learned such that the desired function is accurately modeled? In situations where samples of the mapping are available, that is, both a set of inputs $\mathbf{X}$ and corresponding *labels* $\mathbf{Y}$ such that:

$$F(x) = y \quad \forall x, y \in \mathbf{X}, \mathbf{Y} \tag{2.5}$$

supervised learning can be performed to learn the parameters. This is the process of learning some θ such that $f(x,\theta) \approx y$ for all $x, y \in \mathbf{X}, \mathbf{Y}$, performed by minimizing a *loss function*.

2.3.1 Loss Functions

Loss functions are functions that measure the distance between the ground-truth label y and the model output $f(x,\theta)$, the latter quantity named the *prediction* of the model. Often, these are measured over the entire set of input values $\mathbf{X}$, producing a single value that summarizes the general distance between all of the ground truth values and all of the predictions. For models that are attempting to learn a regression, a mapping from some input $x \in \mathbf{X}$ to some arbitrary number or numbers, common loss functions are based around common distance metrics such as mean square error:

$$L_{mse}(\mathbf{Y},\ f(\mathbf{X},\theta)) = \frac{1}{|\mathbf{X}|} \sum_{x,y \in \mathbf{X},\mathbf{Y}} (y - f(x,\theta))^2 \tag{2.6}$$

In classification tasks, where the desired label is some integer or integers that represents a discrete classification of the input, the model output is often formatted as a probability vector $\vec{y}$, where y_i represents the model's belief that the input belongs to the ith class. In this instance, the goal is to maximize $p(y|x,\theta)$, which can be accomplished by performing maximum likelihood estimation. This means a logical choice for loss function is *negative log likelihood* (Goodfellow et al., 2016):

$$L_{nll}(\mathbf{Y},\ f(\mathbf{X},\theta)) = -\frac{1}{|\mathbf{X}|} \sum_{x,y \in \mathbf{X},\mathbf{Y}} \log p(y|x,\theta) \tag{2.7}$$

Negative log likelihood is often referred to in literature as *cross-entropy loss*, and this convention is used within this thesis as well.

Notice in both of these cases $L = 0$ represents the optimal condition where the predictions exactly match the ground truth labels. As such, the training process is commonly represented as the following optimization:

$$\text{argmin}_\theta L(\mathbf{Y},\ f(\mathbf{X},\theta)) \tag{2.8}$$

Thus, the loss function should map $\mathbf{Y}$ and $f(\mathbf{X},\theta)$ into $[0,\infty)$, where 0 marks the optimal result. While the above are just examples of some common loss functions for two common types of network, any function L that maps the desired ground truth outputs and model predictions into $[0,\infty)$ could in theory be used as a loss function.

2.4 Stochastic Gradient Descent

The problem of optimizing the parameters of a neural net is notoriously tricky. Neural nets can have upwards of a billion parameters in extreme cases (Huang et al., 2018), meaning that optimizing said parameters to minimize error is a multi-billion dimensional optimization problem, which is mathematically intractable. Additionally, networks of any meaningful scale are expensive in time and computation to evaluate, which limits the number of samples that may be drawn from the parameter space. To approach this problem, stochastic sampling methods can be used. The most prominent technique for such optimization is *gradient descent* (Goodfellow et al., 2016). Gradient descent is used to minimize loss functions, and does so through iterative minimization. This is done by stepping the model parameters θ in the direction that most decreases L:

$$\Delta\theta_t = -\eta\nabla L(\theta) \tag{2.9}$$

where η is referred to as learning rate. To translate this gradient into meaningful updates of each parameter, the partial derivative of the loss with respect to each parameter is computed by traversing the model backwards. This allows the use of the chain-rule to quickly compute these partial derivatives, meaning that each parameter need only be aware of the gradient of its output. This process of propagating the errors back up the model is referred to as *backpropagation*, and is the mathematics at the very heart of network training.

During each gradient update step performed in gradient descent, $L(\theta)$ can be computed in a variety of different ways and contexts. If the desire is to minimize loss over the entirety of the training dataset at once, the loss can be taken as the average of the loss of each datapoint in the training data. The step that the model will then take in gradient descent would be one that minimizes the average loss over the entire data. However, this would be tremendously expensive to compute in terms of memory; the gradients of each parameter with respect to each input would need to be stored. Considering a reasonable training dataset for a neural network is on the order of tens of thousands of data points at the very least, and model parameters are usually on the order of hundreds of thousands if not millions, this would be prohibitively expensive. On

the other hand, gradient updates could be performed for each datapoint, computing $L(\theta)$ as the loss for each specific point. The disadvantage with this single-datapoint approach is that many neural network computational libraries can make use of massive parallelization when processing multiple inputs, and backpropagating inputs one at a time through the model is the least efficient use of that capability. Additionally, tuning the entirety of the model parameters to suit a single input is likely heavy-handed as specifically tailoring the model to a single input will likely be detrimental to the performance over all the rest. As a compromise, $L(\theta)$ can be computed from a small batch of the input dataset, then the model updated to minimize loss over that specific batch. This means that each gradient update will be performed with respect to a variety of inputs, while still being able to take advantage of parallelization. This process is referred to as *minibatch stochastic gradient descent* (Goodfellow et al., 2016). However, in the literature it is common to see minibatch SGD referred to as simply stochastic gradient descent, and this thesis will use the same convention.

This process can be thought of as navigating some multi-dimensional parameter landscape, with the position in this landscape given by the current coordinates θ, and the 'height' at that position given by $L(\theta)$. To find the lowest point in that landscape, a step of size η in the direction $-\nabla L(\theta)$, that is, opposite from the direction of steepest gradient, could potentially move downwards and thus decrease the loss. Of course, the gradient at the current position might radically change over the span of the step of size η, thus the next step through this unseen territory might end up worsening the loss. Smaller values of η minimize this risk, as they minimize the unknown space stepped over, but result in needing to take more steps (and thus increasing the number of expensive samples to perform). Additionally, smaller values of η increase the likelihood of getting stuck in local minima; if the step is not big enough to step up and out of a local minima then the model might end up 'circling the drain' forever. Larger values of η can more quickly cover a broader expanse of the parameter landscape and step out of small local minima. However, this also means that each step travels through more unseen terrain, which could potentially step through areas of significant gradient change and into significantly worse positions. The process of conducting an effective stochastic gradient descent is thus balancing and moderating this η to get the best of both the large and small η regimes. A common way of achieving this is through learning rate annealing, wherein the descent process is started with a large η value, such as to explore a large swath of the loss landscape, but this learning rate is then slowly lowered over the course of training such as to introduce finer and finer tuning steps as the model settles into its final minimum (Goodfellow et al., 2016).

Stochastic gradient descent, by the very nature of being a stochastic algorithm working on a tremendously complex loss landscape, is sensitive to initial conditions, algorithmic parameters, and idiosyncrasies in the loss geometry. There is a variety of techniques that can be applied to mitigate these factors and improve the stability and performance of SGD; chiefly, momentum and weight decay.

2.4.1 Momentum

Momentum appends a term into the SGD function:

$$\Delta\theta_t = -\eta\nabla L(\theta_t) + p\Delta\theta_{t-1} \tag{2.10}$$

This term adds some fraction p of the previous step into the current step, literally adding a momentum to the movements through the parameter landscape. This speeds up the movement on extended descents, as well as allows for rolling up and out of local minima at smaller learning rates than would be possible otherwise.

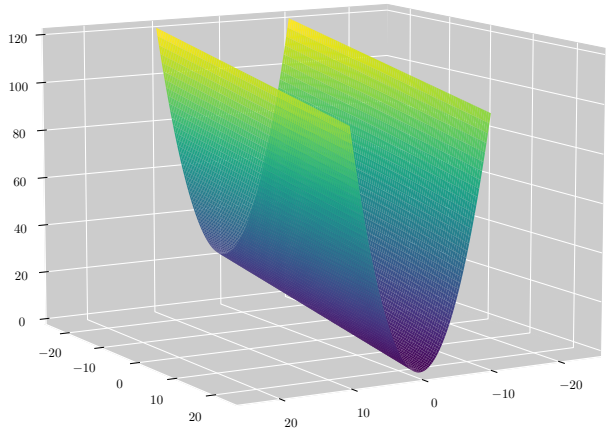


Fig. 2.1 A long and narrow valley as described by Qian (1999).

In one of the first exploratory papers of the concept, Qian (1999) describes a particular geometric case of a "long and narrow valley," (see Figure 2.1) which slopes gradually down along the long axis of the valley. Along the floor of the valley, the gradient points along the short axis of the valley, that is, up the valley walls, as this is the direction of greatest slope. This means a standard SGD algorithm will oscillate roughly in line with the short axis of the valley, slowly traversing down the sloping long axis. The component of the loss update along the long axis will remain roughly constant throughout the process, always pointing slightly downward along the long axis. The short axis component will invert at each step, as the algorithm steps up and returns down each wall, meaning it will zig-zag across the walls of the valley while only making incremental progress downwards. If a momentum term is added, some fraction of the last step persists into the current step, therefore dampening any oscillating motion between steps while magnifying consistent motion. Figure 2.2 shows the difference in descent paths between SGD with and without momentum.

This accelerating factor and oscillatory dampening in strange geometries is exactly the benefit of momentum, and since loss landscapes often feature incredibly complex geometries, momentum is a crucial tool to best perform efficient gradient descent.

2.4.2 Weight Decay

Weight decay (Krogh and Hertz, 1992) adds a L2 regularization to the loss function:

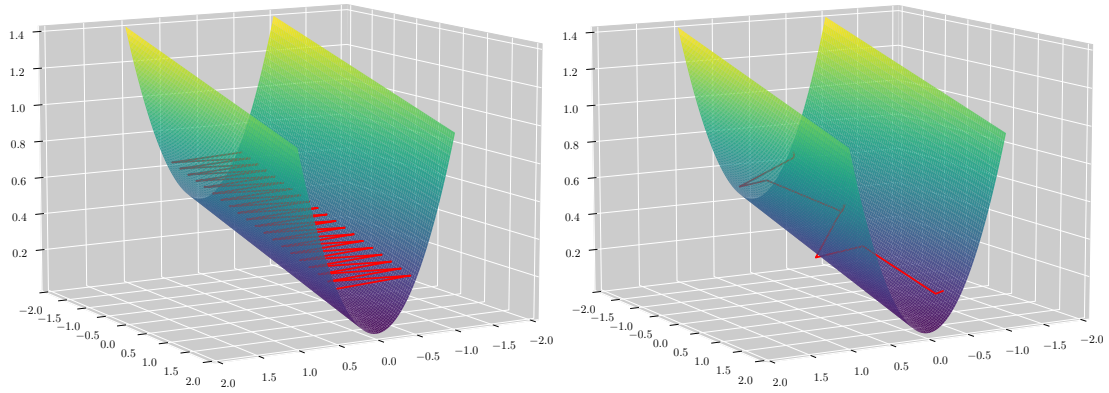


Fig. 2.2 Starting at the point $x = 0.5, y = -2$ these figures show the path taken by SGD as it attempts to descend to $f(x,y) = 0$. At left, SGD is performed without momentum, whereas the right uses a momentum value of 0.9.

$$L(\theta) = \sum_i (L(\theta_i)) + \frac{1}{2}\lambda \sum_i \theta_i^2 \tag{2.11}$$

thus penalizing large values of $\vec{\theta}$ by some factor λ. The effect of this is to both reduce the number of possible 'good' places in the loss landscape to traverse to, as ones further from the origin will be more heavily penalized, while also preventing model overfitting to some degree; the model cannot travel to extreme lengths to contort itself to the training data, instead being obliged to focus on more general solutions in the more localized loss landscape.

2.4.3 Other Optimizers

Stochastic gradient descent is just one of many such optimizers that perform efficient searches of the loss landscape to find optimal model parameter values. Another highly popular choice is the Adam algorithm (Kingma and Ba, 2015), which uses running means and variances of the loss gradient to inform the direction and magnitudes of weight updates. However, the performance of various optimizer algorithms is highly dependent on both their application and choice of specific tuning parameters, which means that the most crucial differentiating factor in their performances is often just how familiar the user is with the algorithm's particular nuances (Goodfellow et al., 2016). Much of the literature that most directly pertains to this thesis makes heavy use of SGD over any other algorithm, and as such is the exclusive choice for optimizers in this work. For more information about other optimizers, see Goodfellow et al. (2016).

2.5 Neural Network Strategies

2.5.1 Validating Model Fitting

While stochastic gradient descent and backpropagation are effective tools to ensure that neural models learn some θ such that $f(\mathbf{X}, \theta) \approx \mathbf{Y}$, there are a couple of pitfalls that this learning

process can fall into. First is *underfitting*, where the model is unable to accurately approximate the desired function and its predictions; the model simply does not know the material well enough to succeed. This can be caused by a number of factors, usually insufficient training time or insufficient model complexity. The former occurs when the model has not seen enough samples of x, y pairs (where $x \in X$, $y \in \vec{y}$) to learn the nuances of the desired function. An easy analogy for this circumstance is not studying enough for an exam; the model has simply not spent enough time reviewing the exam material to perform well. The latter occurs when the model's θ is not large enough to model the desired function; remember the conclusion of the Universal Approximation Theorem from 2.2.2 only applies to sufficiently large models. An analogy that can be drawn is trying to fit a polynomial regression $y = \sum_{i=0}^{n} \beta_i x^i$. If there are $[x, y]$ samples $[0,0]$, $[1,1]$, $[2,4]$, $[3,9]$, $[4,16]$, it is clear that the function being modeled is x^2. However, if the polynomial regression model is constrained to $n = 1$, no amount of fitting will adequately model the desired function. This is exemplary of the second main cause of underfitting, insufficient model complexity.

On the other hand is *overfitting*. This is when the model has managed to essentially 'memorize' the training data, simply memorizing that $f(x_1, \theta)$ and $f(x_2, \theta)$ should equal y_1 and y_2 without actually learning *why*. This happens due to two main causes; insufficient amounts of training data or too much model complexity. The earlier polynomial regression example can easily demonstrate the former cause. In this case, if there are only three samples $[-1,1]$, $[0,0]$, $[1,1]$, there are an infinite range of polynomials that would fit this data perfectly; $y = x^n$ for any even n, for example. Despite all of these functions having perfect accuracy over the sample data, it cannot be definitively stated that they have modeled the desired function; there are simply not enough data samples to have any inkling as to the true function. The latter case occurs when the model is large enough that it has enough flexibility that it can essentially just act as a lookup table; input 1 should return output 1, input 2 should return output 2, etc.

With these two potential pitfalls, it is crucial to understand how to diagnose when a model has fallen into one. The answer is through the use of data splits. In the simplest form, data samples are divided into two sets, a *training set* and a *test set*. Only data from this training split is used when training the model. Two metrics can then determine how the model is faring; training set and test set performance. Since the exact metric to measure 'performance' can vary depending on the particular task at hand, good and bad performance will be referred to qualitatively rather than quantitatively for these examples. Training set performance can reveal how well the model is able to emulate the training samples, and determine whether it has sufficient flexibility to model the function. If the training set performance is satisfactory, it should at be least capable of the desired modelling. Then, the test set performance can be investigated to get a measure of how well the model fares on new data from the same distribution. This provides insight on how much of the model's training performance is learning the desired function or is simply memorization. Good test set performance means the model is accurately predicting the correct outputs for new inputs, despite having never seen these datapoints before. This implies that it is in fact modelling the desired function (or at least one that produces the desired outputs for both the training and

test inputs). Good train performance but poor test performance implies that the model has simply memorized the train datapoints but has no knowledge of the true underlying function to use on novel datapoints. The possible permutations are listed in Table 2.1:

	Good Test Performance	Poor Test Performance
Good Train Performance	The model is well-fit	Overfit
Poor Train Performance	Rare	Underfit

Table 2.1 The various permutations of the two metrics. The lower left state (poor train, good test performance) is unusual, and perhaps indicates that the train and test sets come from different data distributions and are thus not particularly useful for diagnosing model fit.

With these tools it is now possible to identify when a model is moving towards one of these training pitfalls; however, what can be done to avoid them? This question has produced an immensely rich vein of research, particularly in the early 2010s as model size began to skyrocket and overfitting issues became more widespread. In particular, strategies within both data preprocessing and the training policies are particularly useful in avoiding both issues.

2.5.2 Data Preprocessing

Data preprocessing refers to everything done to the data to before passing it into the model. The most common processing done to input data is to standardize and normalize it for the model. Data standardization refers to the process of ensuring each input tensor to the model is of the same shape, that is, having equal size along each of the dimensions. For the types of models discussed here, they are either only capable of operating over fixed size inputs or perform better when that is the case. In the case of image data, typically dimensions to which most of the inputs roughly conform are picked, and then input images are either scaled or cropped to ensure that they all have uniform size (Krizhevsky et al., 2012). Normalization on the other hand is simply ensuring that the values of each of the input datapoints come from roughly the same distribution. Typically, each input to a model is normalized by subtracting the dataset mean μ_X and scaling by the dataset standard deviation σ_X:

$$x' = \frac{x - \mu_X}{\sigma_X} \quad \forall x \in X. \tag{2.12}$$

This ensures that the input dataset as a whole has a mean of 0 and a standard deviation of 1. As seen in Figure 2.8, most common neural network nonlinearities are most interesting between $[-1, 1]$. Significantly outside these bounds, the nonlinearities are all mostly linear, defeating the purpose of using the nonlinearity in the first place. Normalizing the input data ensures that it lies in this interesting region, thus maximizing the power that the network can provide.

Another data strategy that can be performed to maximize the value of the input dataset is *augmentation*. This is the process of transforming and modifying the input data such as to increase the total number of datapoints that are available. Typically, this involves looking for a

set of transformations T of the input data that are *label invariant*, i.e., do not modify the output labels:

$$f(x) = f(t(x)) = y \quad \forall t \in T. \tag{2.13}$$

These sets of transformations vary on a per-dataset, per-task basis, as the definition of label-invariance varies depending on the type of input data and the exact mechanism of labelling. Figure 2.3 provides an example of this concept. This is particularly common in image data, with augmentations like horizontal or vertical flipping, rotations, random croppings, and color jittering used fairly frequently. The power of augmentations is that for even the simplest transformations like flipping, the amount of input data is doubled. This can be very helpful in scenarios with small datasets which can run into overfitting issues, where the larger augmented dataset can make it harder for the model to memorize the inputs.

Task: Object Classification

(a) $f(x) = \texttt{car}$

(b) $f(t(x)) = \texttt{car}$

Task: Letter Classification

(c) $f(x) = \texttt{b}$

(d) $f(t(x)) = \texttt{d}$

Fig. 2.3 Horizontal flipping augmentation applied to two different image classification tasks. In the first, horizontal flipping does not change the label of the image; a mirror image of a car is still a car. In the second, horizontal flipping changes the label of the image from `b` to `d`. This is an excellent example of why augmentation policies need to be chosen on a per-dataset, per-task basis; despite both datasets being image classification tasks, this augmentation policy is only valid for the first.

Furthermore, augmentation can be used to describe or reinforce desirable properties of the mapping that the network is trying to learn. For example, if it is known that the desired mapping is commutative, an augmentation that randomly shuffles the order of values within the inputs can be applied. This will directly demonstrate to the model that the order of values within the input does not matter to the final labelling, which forces it to also produce a commutative mapping

if it wants to be performant. In the case of image data, encouraging translational invariance (regardless of where an object appears in a picture, it does not change the properties of said object) can be helpful. Doing this entails grabbing a random subcropping of each image and then padding them back to the desired dimension. This will shift the contents of the image by however much the center of the random cropping differs from the center of the original image, thus demonstrating the desired translational invariance. Augmentation, by both increasing the amount of data that the model is exposed to and enforcing useful strategies that the model can apply to better learn the data, can help with both under and over-fitting.

While the previous two strategies have been mainly focused on making the tasks easier for the model to learn, data preprocessing can also be used to apply *regularization*. Regularization is essentially a limitation that is applied to constrict the valid solutions to the problem, in such a way that the model is guided to a more suitable or better performing answer. In terms of data preprocessing, a very common regularization that is applied particularly to image data is Cutout, introduced by DeVries and Taylor (2017). Cutout simulates object occlusion, by cutting out a square section from a random location of the input image, as demonstrated in Figure 2.4.

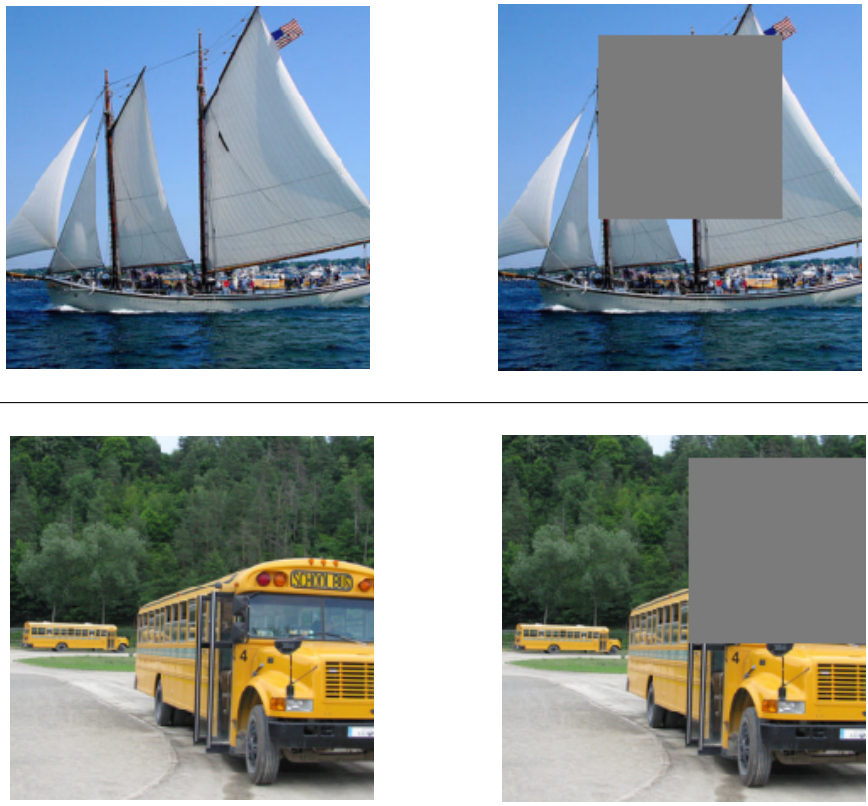


Fig. 2.4 Cutout applied to different images from the ImageNet dataset. The cutout area is typically replaced with the mean color from the dataset to avoid imbuing artificial color information into the data.

Cutout is applied probabilistically to each input image, meaning that every time that image is passed through a model it may potentially occur. If it does, the location at which it occurs will also be randomly chosen. This makes it very hard for the model to memorize the input images, as particular locations that the model might otherwise latch on to cannot be reliably trusted to appear in the image. For example, in the schoolbus image in Figure 2.4 a model might latch on

to the light pattern and words “SCHOOL BUS” on the top of the bus, and use these to memorize that this image is a school bus. However, if this section is cutout, that memorization is no longer possible and the model is forced to look at other ways to correctly identify this image. In this sense the model is forced to learn more about the content and context of the images that it is processing, not just memorize the positions of key objects. The idea is that the former method of learning is much more generalizable than the latter, and should therefore help the model avoid overfitting.

Since the data constitutes the entirety of the model’s knowledge of the problem and thus its potential to learn, it is critical that data strategies like normalization, augmentation, and regularization are used to ensure that the teaching material is as powerful as possible.

2.5.3 Training Policies

With the potential of the data fully-extracted (ideally), the focus shifts to how the learnings from the data are applied to the model. The methodology of how the model is trained and the strategy therein is called the training policy.

The biggest decision when it comes to training a model is setting the parameters of the training regime itself, called the *hyperparameters* of the model. These are not in and of themselves trainable parameters of the model, but play a crucial role in the values those parameters take on. While there are essentially infinite possible classes of hyperparameters depending on the model’s specific needs and how training is implemented, the most ubiquitous and influential are *training epochs*, *learning rate*, and *batch size*. Training epochs refer to the total number of times the entirety of the training data is passed through the model. This is necessary due to the iterative nature of gradient descent, as each step can only bring the model slightly closer to an optima. A single step improves the model, but many steps are needed to satisfactorily traverse the loss landscape. The choice of this is often entirely dataset and model dependent, and typically is tuned through experimentation or grid searching. In general, large epoch numbers bring good performance at the cost of compute time, while low epoch numbers bring the opposite, and as such finding a balance is crucial.

Learning rate is identical to the learning rate described in Section 2.4, and simply indicates the size of the step taken by the gradient descent algorithm. Choice of learning rate is crucial for the same reasons as outlined in that section, as it determines the level of detail to which the algorithm can navigate the loss landscape. In this sense, overly large learning rates can cause underfitting, as the model will step over finer dips and valleys in the loss landscape and thus only be able to find coarse minima. However, a learning rate that is too small will cause training to take a long time, as many steps are needed to get anywhere meaningful. An effective strategy to avoid making this choice altogether is called *learning rate annealing*. This is the process of adjusting the learning rate over the course of training, such as to get benefits from both sides of the learning rate spectrum. Typically, such strategies start with a large learning rate, to bring the model quickly from initialization to some decent coarse minima, and then reduce the learning rate to allow for further fine optimizations. Early strategies were often as simple as dividing

the learning rate by a constant after a fixed number of epochs. A strategy like this however tends to complicate hyperparameter choice; now the initial learning rate, the divisor, and interval between decisions have to be chosen. Each of these has the potential to drastically affect the final performance of the model, so rather than removing the single high-stakes choice of learning rate, two more choices are added. An annealing strategy that I find elegantly avoids this issue is *Cosine Annealing*, introduced by Loshchilov and Hutter (2016). Here, the learning rate η at some epoch t is brought from some maximum η_{max} to a minimum η_{min} over the course of T epochs (PyTorch, 2021):

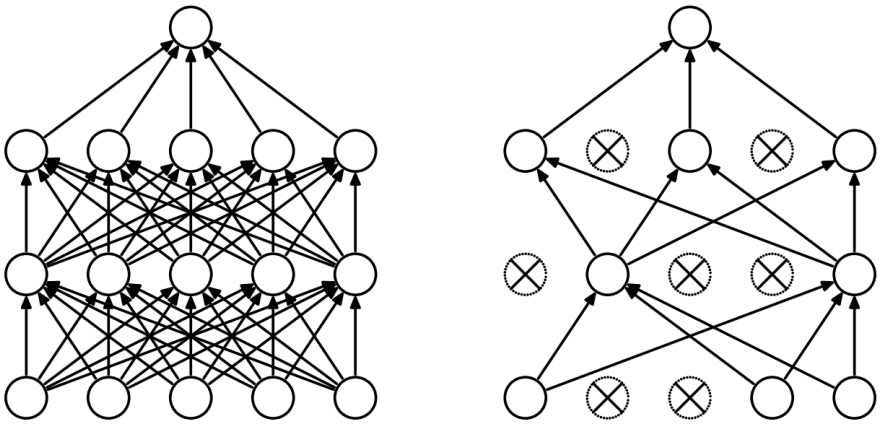
$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{t}{T}\pi\right)\right) \tag{2.14}$$

This way the learning rate is gradually and smoothly swept between the upper and lower bound over the entirety of training. Typically, η_{min} is chosen to be 0, and T is equal to the total number of training epochs of the model. This means the only thing to choose is η_{max}, and as such cosine annealing grants the benefits of learning rate annealing without adding any extra hyperparameter selections to the model. Again, for more information about adaptive learning rates, see Goodfellow et al. (2016).

Finally, batch size refers to the number of datapoints that are passed simultaneously to the model during training. This turns stochastic gradient descent into batch gradient descent: rather than adjust model parameters after each individual training data point, model parameters are updated according to the average loss gradients of all the training points within a specific batch. Too small a batch size can result in the model making gradient updates according to idiosyncratic information within the batch that may not generalize well, whereas too large a batch size can cause gradient updates to be so general as to contain little meaningful information (Smith and Le, 2017). Meanwhile, small batch sizes are cheap in both computational cost and memory allocation as less concurrent data needs to be passed through the model, but many such batches are needed to perform a full pass over the training data. Large batches can pass over the training data faster, but are conversely more expensive to compute. As such, a balance between the two extremes is crucial to find.

Training strategies can also aim to apply regularization to the weights of the model in some way, such as to enforce solutions that are unlikely to be overfit. The first of these is a concept called *dropout*, introduced by Srivastava et al. (2014). Here, a random selection of connections within the model are removed during each forward pass, essentially sampling a random subnet of the model as shown in Figure 2.5.

In every forward pass of the model, its exact configuration will be different, with the idea being that this reduces the complexity of the model in an unpredictable way. This complexity is precisely the tool the model can exploit to 'memorize' the training data, by using its immense number of connections to fit to subtle details or even just noise in the training data. When the relation between these connections cannot be guaranteed to be consistent, the model cannot rely on pure brute force to learn the mapping. Instead, Srivastava et al. note that each neuron in the model must instead learn to be collaborative with a random sample of its fellow neurons, and



(a) Standard Neural Net (b) After applying dropout.

Fig. 2.5 Dropout's effect on the connectivity of a neural net, as shown in Srivastava et al. (2014). The exact selection of edges to remove, and thus the exact difference between (a) and (b) changes every forward pass.

cannot form the intricate codependencies necessary to memorize the training data. As such, each neuron is forced to robustly extract useful features without reliance on other neurons. When compared to networks trained without dropout, Srivastava et al. find that its use drastically reduces overfitting of the model, and produces significant increases to test performance. It may seem counterintuitive that destroying a model's ability to use intricate relations between neurons to learn highly nuanced mappings would benefit test performance, but remember that overfitting often comes from models fitting to noise within the training data that does not exist in the test data. By preventing the model from 'cheating' in training, it is forced to learn actual insight into the task at hand, and this insight is generalizable to new inputs. In practice, dropout is typically performed on fully connected layers in networks, with a fixed dropout probability between 0.3 and 0.5 during training and 0.0 during testing.

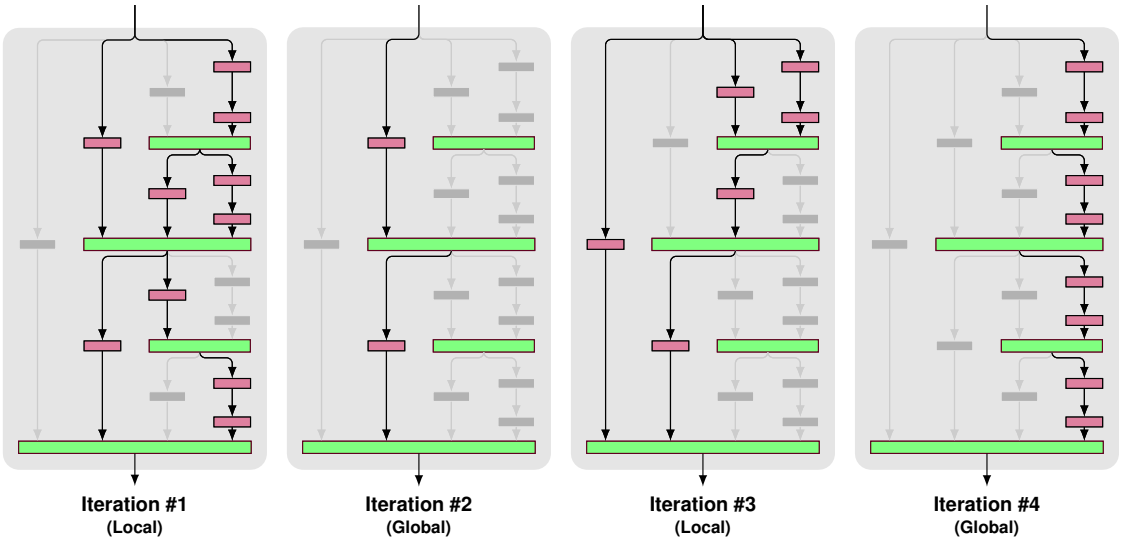


Fig. 2.6 Local and global Drop-Path's effects on the connectivity of a neural net, as shown in Larsson et al. (2016).

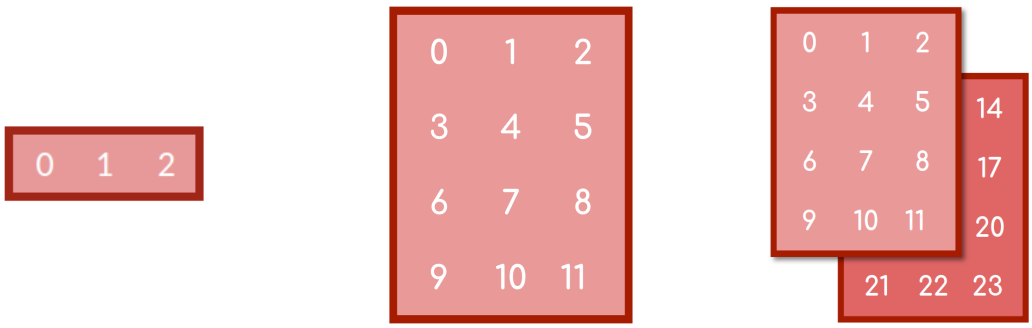
This regularization-via-removal concept can be extended to network architectures as a whole, and drop out entire pathways through the model. This process is referred to as Drop-Path (Larsson et al., 2016), and can operate over two different scales: local or global. Local Drop-Path looks granularly at individual layers of the network architecture, specifically at the various inputs to any given layer. These inputs are then randomly dropped during training, while still ensuring that at least one path always remains to the input. Global Drop-Path instead looks at each possible path from the network input to output, and randomly samples a single one at each training step. Figure 2.6 shows graphically the effects of both local and global Drop-Path. Together, they ensure that the paths to particular layers and the various paths through the model are all independently capable of producing good results, improving a model's strength and robustness.

2.6 Beyond Feedforward Networks

Fully-connected layers can be tremendously powerful in large numbers and can learn to approximate any function that might be needed. However, by complementing these layers with other types of operations their power can be greatly expanded. Furthermore, directly inserting layers that have useful mathematical properties will reduce computational cost and time, since it bypasses the network training needed to approximate those functions organically. In general, all such operations are formulated as tensor operations, typically performed using linear algebraic functions which allows for fast parallel processing (see Section 2.9 for more details).

2.6.1 Tensor Operations

A tensor, at least for the purposes of this work, is simply the n-dimensional generalization of vectors and matrices. An order-1 tensor is a vector, an order-2 is a matrix, and so on, and the data that flows through deep learning models are represented in this form.



(a) Order-1 tensor (vector) (b) Order-2 tensor (matrix) (c) Order-3 tensor

Fig. 2.7 Depictions of tensors of various orders.

A tensor's shape refers to its size in each of its dimensions; so the tensors in Figure 2.7 are of shapes (3), $(4,3)$ and $(2,4,3)$, respectively. Tensor operations manipulate these tensors, performing linear algebra transformations over their contents. While the normal set of linear

algebra operations like additions and inner products are used fairly commonly in neural networks, there are more domain-specific and specialized ones that are also very common.

2.6.2 Nonlinearities

The simplest form of tensor operation is the nonlinearity or activation function, a nonlinear function that often maps $\mathbb{R}$ into some subset of $\mathbb{R}$. These nonlinearities allow for more complex functions to be modelled, thus turning the purely linear combinations produced by neurons into Universal Approximators (Stinchcombe and White, 1989).

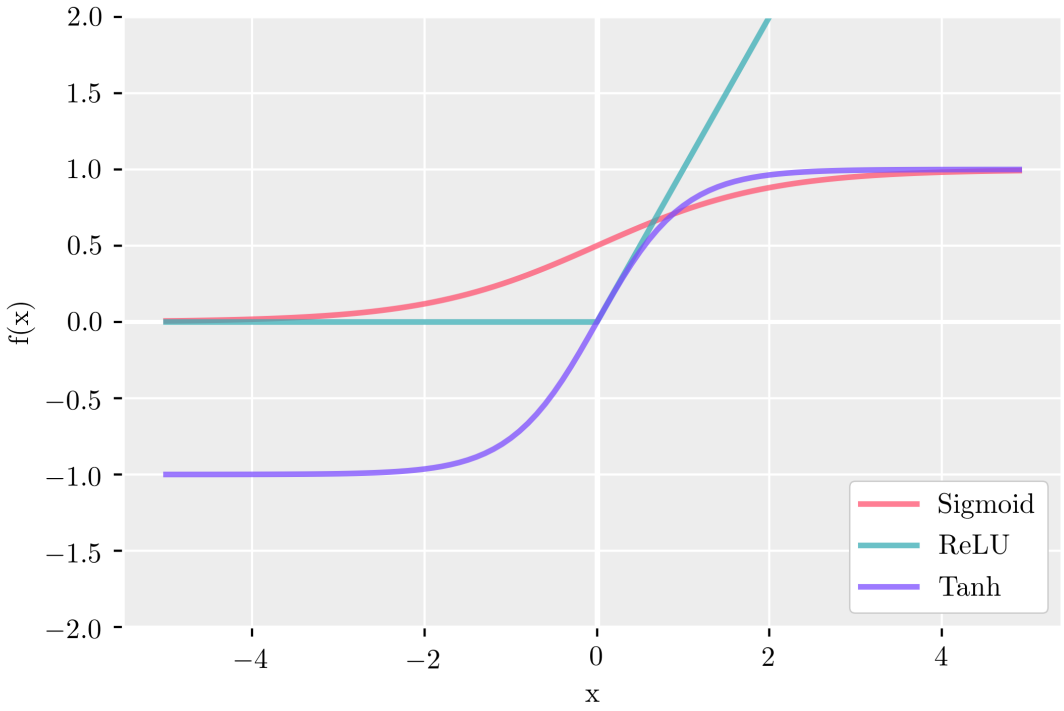


Fig. 2.8 The three most common tensor nonlinearities.

$$\text{Sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{2.15}$$

$$\text{ReLU}(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{2.16}$$

$$\text{Tanh(x)} = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{2.17}$$

The three most common types of nonlinearity used are the sigmoid (Equation 2.15), the rectified linear unit (ReLU, Equation 2.16), or the hyperbolic tangent (tanh, Equation 2.17). Sigmoids and tanhs are closely mathematically related and often serve similar purposes within neural nets, and their ability to map $\mathbb{R}$ into a very restricted space serves well to prevent a neural net phenomenon

called gradient explosion. Gradient explosion can occur during network training, where the learnable weights of the network are updated proportionally to their error gradient (more on this later). In cases where this error gradient is very large, it can create a positive feedback loop, and rapidly send the weights and outputs of a function to positive or negative infinity. With the tanh and sigmoid, this is prevented because $|f(x)| \ll |x|$ for the majority of x values, which clamps down the feedback loop and ensures that this runaway effect never occurs.

However, this exact behavior causes another issue to arise, and that is the dying gradient problem. In cases where the sigmoid and tanh function are receiving values of such that $|x| \gg 0$ (called saturation), the derivatives of these nonlinearities with respect to these extreme inputs are zero. This in turn means the error gradient and thus the proportional weight update are also zero, meaning the network cannot update the neuron weights and the neuron is stuck at this saturated value. This causes poor model training results, and tends to limit the depth of networks that make use of sigmoid or tanh activations (Maas et al., 2013).

ReLUs seek to remedy this issue by eliminating positive saturation, allowing positive values to flow unaltered through the activation. This is more biologically similar to organic neurons which exhibit the same thresholding behavior as the ReLU, which is where the ReLU equation draws inspiration (Glorot et al., 2011). While Glorot et al. note that negative saturation is hard-coded into ReLUs and is more extreme than in either sigmoid or tanh activations (i.e., the gradient for all $x < 0$ is always 0), the original authors note that it does not have particularly adverse effects on model training performance. They theorize that this is due to the counter-balancing effects of other parallel unsaturated neurons, which are more prevalent in ReLU networks due to their single direction of saturation. However, since ReLUs do not throttle positive inputs, gradient explosion can be a significant problem. Despite this, ReLUs are typically the activation of choice in modern models, helped by their relatively minimal computational cost compared to others.

2.6.3 Batch Normalization

To remedy both dying and exploding gradients, Ioffe and Szegedy (2015) introduced batch normalization. Batch normalization simply ensures that each batch of tensors that pass through the operation are first normalized to mean 0 and variance 1, before a learned scaling γ and shifting β is applied:

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \qquad \text{(Normalize according to batch mean } \mu_B \text{ and variance } \sigma_B^2\text{)} \quad (2.18)$$

$$y_i = \gamma \hat{x_i} + \beta \qquad \text{(Scale by } \gamma \text{ and shift by } \beta\text{, values set by network's particular needs)} \quad (2.19)$$

The mean and variance are computed per-batch during training, while the running averages of these statistics are used to estimate the population mean and variance during inference. Performing this normalization on a per-batch basis as opposed to globally is more computationally efficient and allows for the influence of the normalization to be differentiable, meaning that it

can be seamlessly incorporated into backpropagation without modification. The second step of the batch-normalization function, the scale and shift, is in place so that the original 'intent' of the operation can be maintained; consider a sigmoid operation to understand why this is necessary. Around zero, it is more or less a linear mapping. In large positive or negative regions, it is a hard mapping to 1 or -1. Were all inputs to a sigmoid function normalized to mean 0 and variance 1, the sigmoid is rendered mostly useless. As such, the output of the batch-normalization can be shifted or scaled such that the model can align it with the desired input distribution of the subsequent operation.

Allowing the model to set a desired region in which each input distribution should lie in mitigates a crucial issue known as covariate shift. This occurs when the distributions of input to some learned function changes over the course of training. Imagine some deep layer f_n of the network, whose input x_n is some combination and composition of all the previous operations in the network:

$$x_n = f_{n-1}\left(f_{n-2}\left(f_{n-3}\left(\dots f_1\left(x_1\right)\right)\right)\right) \tag{2.20}$$

As the network learns, the output distributions of each of the layer operations $f_{n-1}, f_{n-2}, \dots, f_1$ will change. This is a compounding effect, in that each change to the output distribution of an early layer will effect the input distributions of later layers, thus influencing their output distributions in turn, and so on. This means that these later layers will have to keep adapting their parameters for these changing inputs, having to accommodate constant fluctuations in input distribution. By enforcing a consistent distribution that will always lie in some desired location, batch normalization allows operations to spend the entire training process specializing to their specific task in the network, and should therefore converge must faster than operations that had to constantly adjust for changing input distributions. Ioffe and Szegedy found this to be exactly the case; models with batch-normalization operations converged around five times faster than models without.

2.6.4 Convolutions

When working with higher dimensional input data such as images, it can get very computationally expensive to use pure feedforward networks. For a very small input 32x32 image with three color dimensions, the corresponding tensor has a shape of (3, 32, 32): a 3072 dimensional input to the model. Simply adding a single feedforward layer of comparable dimension requires on the order of 10 million differentiable parameters. Furthermore, there is a structural facet to the input data; in images, information is likely to be highly locally correlated both spatially and colorwise. This means the properties of regions are likely to be more interesting than individual points, and want to perform operations that can meaningfully extract this spatial information: learning how these local regions interact is more useful than the relation between remote regions. While feedforward layers could in theory learn this spatial dependence, it would require a lot of serendipitous coordination. Each neuron in a fully-connected layer receives input from all previous neurons, so

a neuron trying to focus locally would need to set a majority of its input weights to 0 to achieve this behavior. Another consideration when working with images is translational invariance; under many circumstances, the ground-truth function $f'(x)$ of an image does not change if the contents of the image change their position. For example, if the ground-truth function is to identify the animal within an image, an image of a dog is still an image of a dog regardless of where it appears within the borders of the image.

While fully-connected layers could in theory learn all of these desirable behaviors given enough time and favorable training material, it makes a lot more sense to directly use an operation that comes with all of them built in, and this operation is the convolution. Tensor convolutions are operations that filter some n-dimensional input tensor $\mathbf{A}$ by some kernel $\mathbf{K}$:

$$(\mathbf{A} * *\mathbf{K})_{x_1,\ldots,x_n} = \sum_{i_1 \in \mathbb{Z}} \ldots \sum_{i_n \in \mathbb{Z}} \mathbf{K}(i_1,\ldots,i_n)\mathbf{A}(x_1 - i_1,\ldots,x_n - i_n) \tag{2.21}$$

This equation slides the kernel $\mathbf{K}$ over every element of the input signal $\mathbf{A}$. At each position, it takes the Frobenius inner product[1] of the kernel and the overlapped region of the input.

In practice, these convolutions are almost exclusively used in two-dimensional cases wherein $\mathbf{K}$ is only defined over some finite range $[-m,m]$ and $\mathbf{A}$ over some range $[-n,n]$. Outside the range $[-n,n]$, the signal is 'zero-padded' setting $A(x_1,x_2,\ldots,x_n) = 0$ if any x_i is outside the range $[-m,m]$, which serves in the case wherein $\mathbf{A}$ is indexed in an undefined region. In this two-dimensional, finite case, the equation becomes:

$$(\mathbf{A} * *\mathbf{K})_{x_1,x_2} = \sum_{i_1=-m}^{m} \sum_{i_2=-m}^{m} \mathbf{K}(i_1,i_2)\mathbf{A}(x_1 - i_1, x_2 - i_2) \tag{2.22}$$

For example:

$$\text{If } \mathbf{A} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \quad \text{and} \quad \mathbf{K} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \text{ then:} \tag{2.23}$$

[1] The Frobenius inner product is an adaptation of the vector dot product tailored for use with matrices; it is computed as the sum of the element-wise product of two matrices: $\langle \mathbf{A},\mathbf{B} \rangle_F = \sum_i \sum_j \mathbf{A}_{ij}\mathbf{B}_{ij}$

$$(\mathbf{A} * *\mathbf{K})_{1,1} = \begin{matrix} 0 & 1 & \\ 3 & 4 & \\ & & \end{matrix} = \left\langle \begin{matrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{matrix}, \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 3 & 4 \end{matrix} \right\rangle_F = 4 \tag{2.24}$$

$$(\mathbf{A} * *\mathbf{K})_{1,2} = \begin{matrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{matrix} = \left\langle \begin{matrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{matrix}, \begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 3 & 4 & 5 \end{matrix} \right\rangle_F = 6 \tag{2.25}$$

$$(\mathbf{A} * *\mathbf{K})_{1,3} = \begin{matrix} 0 & 1 & 2 \\ 3 & 3 & 4 \\ 6 & 7 & 8 \end{matrix} = \left\langle \begin{matrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{matrix}, \begin{matrix} 0 & 0 & 0 \\ 1 & 2 & 0 \\ 4 & 5 & 0 \end{matrix} \right\rangle_F = 6 \tag{2.26}$$

and so on, such that the final output is:

$$\mathbf{A} * *\mathbf{K} = \begin{matrix} 4 & 6 & 6 \\ 10 & 16 & 14 \\ 10 & 18 & 12 \end{matrix} \tag{2.27}$$

These kernels filter the input data, and as such can highlight or suppress certain features of the data depending on the contents of the kernel. Working with some examples will demonstrate how this can be useful. Since the predominant use case for these convolutional kernels in the field is for image processing, images will be used as input data in these examples. A color image can be thought of as a stack of three two-dimensional matrices, one matrix (or channel) for the red values at each point, one for blue, and one for green. Then, the kernel will be applied to each channel of the image (essentially performing three separate convolutions) and then the final output will demonstrate what the kernel has extracted or suppressed. Figure 2.9 looks at two 3×3 kernels, and shows the results produced by convolving images with these kernels:

The first kernel $\mathbf{K}_{blur}$ will return a weighted sum of each of the nine points overlapped by the kernel; this is equivalent to the average pixel value of the nine points. The end effect of this is that each pixel in the output image is the average value of the 3x3 region centered at that point in the input image, producing a blurring effect.

The second kernel $\mathbf{K}_{edge}$ extracts edges in the input image. This is because in regions with uniform color information (and thus edgeless regions), the strongly weighted center point will get counterbalanced by the negatively weighted outer points. The only way for this to return nonzero is if some of the outer points have strongly different information than the center point, which would occur at color boundaries. This therefore means the output image is nonzero only along such boundaries, thus extracting edges.

While these two kernels are hand-designed to work specifically for images (and to produce good looking results for this example), convolutional layers within models can discover and learn

Original Images:

$$\mathbf{K}_{blur} = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Fig. 2.9 Blurring kernel

$$\mathbf{K}_{edge} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Fig. 2.10 Edge detection kernel

kernels that best extract relevant information from the data passing through their particular layer within the network. There are a few different types of the convolutional layers, which all serve different purposes within models. The most common type applies multi-dimensional kernels of shape $(N, M, k_1, .., k_n)$, where N refers to the number of output channels, M the number of input channels, and $k_1, \ldots, k_n$ to the size of the kernel in each spatial dimension. The kernels given above in Figure 2.9 would be of shape $(3,3,3,3)$, as they were each 3×3 kernels that received and outputted the three color channels in the image. The general equation for this class of convolutional layers is:

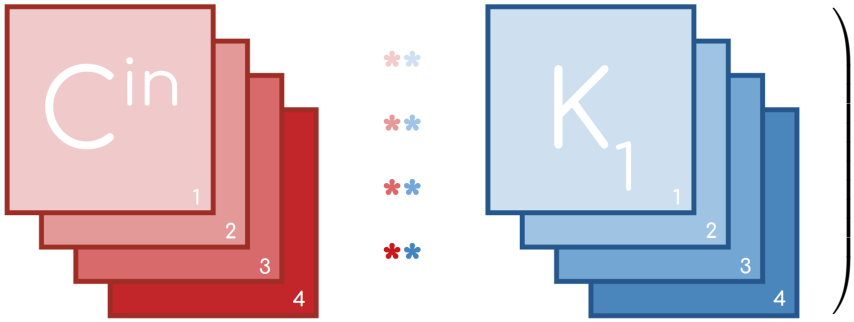
$$\mathbf{C}_n^{out} = \sum_{m=1}^{M} (\mathbf{C}_m^{in} * * \mathbf{K}_{nm}) \tag{2.28}$$

Here, $\mathbf{K}_{nm}$ refers to the $(N, M, k_1, .., k_n)$ kernel tensor sliced at the nth output dimension and mth input dimension. For example, if there are four input channels and two output channels are desired, the kernel tensor would be of shape $(2,4,k_1, .., k_n)$, and the first output channel would

be calculated as follows:

$$\mathbf{C}_1^{out} = \sum_{m=1}^{4} (\mathbf{C}_m^{in} * * \mathbf{K}_{1m}) \tag{2.29}$$

$$\mathbf{C}_1^{out} = \sum \left(\right. \tag{2.30}$$

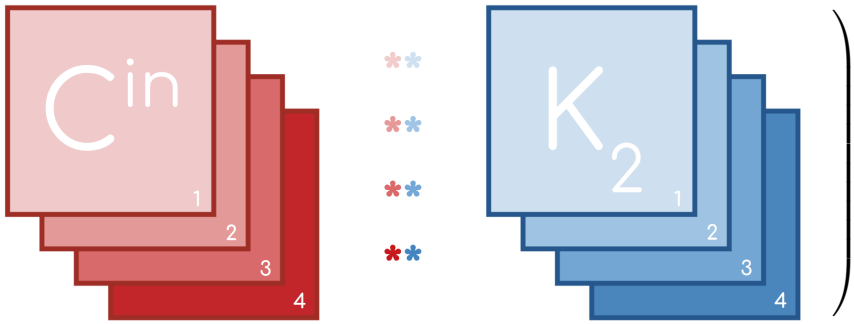


$$\left. \right) \tag{2.31}$$

and the second output channel:

$$\mathbf{C}_2^{out} = \sum_{m=1}^{4} (\mathbf{C}_m^{in} * * \mathbf{K}_{2m}) \tag{2.32}$$

$$\mathbf{C}_2^{out} = \sum \left(\right. \tag{2.33}$$



$$\left. \right)$$

A subtype of this vanilla $k \times k$ convolution is the 1×1 convolution or the Network-in-Network, introduced in a 2013 paper of the same name (Lin et al., 2013). These convolutions involve 1×1 convolutional kernels, which means the convolutional equation simplifies to:

$$\mathbf{C}_n^{out} = (\mathbf{A} * * \mathbf{K})_{x_1,x_2} \tag{2.34}$$

$$= k\mathbf{A}(x_1, x_2) \tag{2.35}$$

$$= \sum_{m=1}^{4} (k_{n,m} \mathbf{C}_m^{in}) \tag{2.36}$$

where $k_{n,m}$ is the single value that arises when the nth 1×1 kernel $\mathbf{K}_n$ is sliced along channel dimension m. In essence, this means that each 1×1 convolutional kernel is performing a weighted sum of the values of the input channel. The output is thus N weighted sums of the input, meaning the 1x1 convolution is the equivalent of a M-N fully connected layer, hence the name Network-in-Network.

2.6.5 Poolings

While convolutions are an excellent means of efficiently extracting information about local patterns within a tensor, they are still relatively expensive to train; a $k_1 * k_2$ convolutional kernel with N input channels and M output channels has $k_1 * k_2 * N * M$ parameters to learn. In circumstances where it is more important to aggregate local patterns than synthesise new information about these patterns, some interesting kernels can be 'pre-baked' to perform these aggregations with no need for training. Such aggregations are called *pooling* operations, and they simply aggregate local information in some preset way. The two most common pooling operations in literature are by far Max Pooling and Average Pooling. The former is a kernel that simply returns the max value found within the kernel, while the latter operates identically to the blurring kernel discussed in Figure 2.9, returning the mean value of the values within the kernel.

$$\text{Max Pool}_{3x3}\left(\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}\right) = \max(0,1,2,3,4,5,6,7,8) = 8 \tag{2.37}$$

$$\text{Average Pool}_{3x3}\left(\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}\right) = \text{mean}(0,1,2,3,4,5,6,7,8) = 4 \tag{2.38}$$

Commonly, pooling operations are used to downsample images, that is, shrink them down to some smaller height and width. This is done by changing the operation's *stride* s to some value greater than 1. Stride sets the step size of the kernel as it slides over the input image, which has the effect of sampling the result of the pooling operation at every sth location. This in turn reduces the output size of the operation by a factor of s. For example, a stride=2 pooling applied to a 50×50 pixel image would produce a 25×25 output.

2.7 Convolutional Neural Networks

By taking the operations described above and combining them in various connectivity patterns, models can be designed that can take advantage of the variety of desirable mathematics that these operations hold. These models are referred to as *Convolutional Neural Networks* or CNNs. Typically, the convolutional neural network gets its name from the operation that most commonly makes up the majority of its internal transformations, but there is nothing preventing a CNN from having a majority of pooling operations or non-linearities.

2.8 Datasets

The two most important things that shape the design of deep learning models are the data and task to which the model is intended to be applied. The data heavily affect the internal mathematics

of the model, that is, determining which kind of tensor operations are most applicable to this domain of data. Image data, for example, are often paired with convolutional operations because they are capable of providing significant performance very efficiently. Meanwhile, the task the model is intended to perform also affects the model connectivity and the architecture, particularly the structure of the later stages and output of the model. Due to this co-mingling influence of data and task, it is helpful to wrap them together into a single conceptual unit. Henceforth, a deep-learning 'problem' refers to this conceptual unit, the joint data and task pair that is being worked with.

However, while building models for a specific, individualized problem is useful to those individuals grappling with said problem, it may be of limited applicability in other domains. As such, in order to create truly generalizable approaches that are appropriate for a wide variety of applications, it is imperative to work with problems that are proxies for an entire domain of problems. The ideal proxy problem is one that is truly generalizable, wherein a model that performs well on the proxy will perform well in any problem within the proxied domain. These proxy problems come to represent the entire data domain, where showing performance on one of these proxy problems is a shorthand for claiming equal performance on any such problem.

2.8.1 MNIST

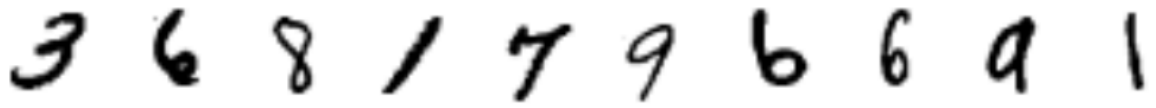


Fig. 2.11 Some sample images from the MNIST dataset, as presented in the original 1998 paper.

One the earliest such sets of proxy problems for visual classification was the MNIST handwritten digit classification problem, introduced in LeCun et al. (1998). This dataset is based on two datasets of binary images of handwritten digits published by the American National Institute of Standards and Technology, (NIST) one taken from a sample of census workers and the other high school students. Each image of a handwritten digit was paired with its corresponding label from 0 to 9, meaning it is a 10 class classification problem. LeCun et al. combined the two datasets into a distinct train and test set, such that both types of writer appeared in both sets but no individual writer overlapped the two sets. This left them with 60,000 training images and 10,000 test images, each 28x28 pixels in size. 10 such images are shown in Figure 2.11. They dubbed this dataset the Modified NIST dataset, or MNIST for short, and provided some examples of convolutional neural network performance over this dataset. While the largest feasible convolutional neural networks in 1998 are orders of magnitude smaller than the ones used today, their networks managed to achieve up to 99.3% accuracy on the test set.

MNIST is a very small dataset, due to both being monochromatic and having a compact image size, and has well-distinguished classes. While these attributes make it very portable and quick to work with, it means the task is relatively simple. This is evidenced by the remarkable

performance LeCun et al. managed to find with the rudimentary networks of 1998. This fact has caused MNIST to fall out of prominence as a computer vision proxy problem; there is very little to prove in scoring very highly on MNIST. If a 99.3% was possible in 1998, it is not particularly impressive to do better in 2021.

2.8.2 CIFAR

Fig. 2.12 An image from each class of the CIFAR-10 dataset.

In the name of introducing a dataset involving "real", full color images, Krizhevsky (2009) introduced the CIFAR family of classification problems. Krizhevsky argued that full color, real images, that is, ones that are actual photos of real world objects, demonstrate unique data correlations. Specifically, pixels are strongly correlated to nearby pixels and weakly correlated to distant ones, and same-channel correlations between neighboring pixels are much stronger than cross-channel correlations. Furthermore, Krizhevsky comments on the lack of useful labels on existing datasets, specifically referring to the MIT Tiny Images dataset (Torralba et al., 2008).

To address this, a 60,000 image subset of the Tiny Images dataset was hand-labelled into 10 classes, and another 60,000 image subset into 100 classes. Each image is 3×32×32, with the first dimension representing the three color channels of the RGB image. These two subsets were named the CIFAR-10 and CIFAR-100 datasets, after the Canadian Institute For Advanced Research. Ten CIFAR-10 images are shown in Figure 2.12, one from each class. Of the two datasets, CIFAR-10 is by far the most popular, due to its conciseness; it is easy to visually distinguish, remember, and program the 10 available classes as compared to the more numerous and nuanced classes of CIFAR-100.

As a result, CIFAR-10 is ubiquitous within the field of computer vision, with the original paper having around 10,000 citations on Google Scholar and the search term "CIFAR-10" yielding 360,000 Google search results as of March 2021, and as such is one of the most common benchmarks researchers use to compare their models against competitors. As of March 2021, state-of-the-art results on CIFAR-10 tend to range from 97% to 99% classification accuracy, with the record score of 99.0% being held by GPipe (Huang et al., 2018). While accuracy is generally the most common and considered metric for CIFAR models, parameter count serves as an additional orthogonal direction of research. Both directions have been explored in recent papers, with papers like DARTS (Liu et al., 2018) highlighting their technique's relatively low 3.3 million parameters as more portable and compact than the competition, while GPipe's ability to scale up to a massive 83.9 billion parameters is promoted as unprecedented and unparalleled. Both directions are important, in that compactness and portability are tremendously useful in

bringing the power of deep learning to a wider range of devices, while sheer size is often the key to unlocking higher model performance (Tan and Le, 2019).

2.8.3 ImageNet

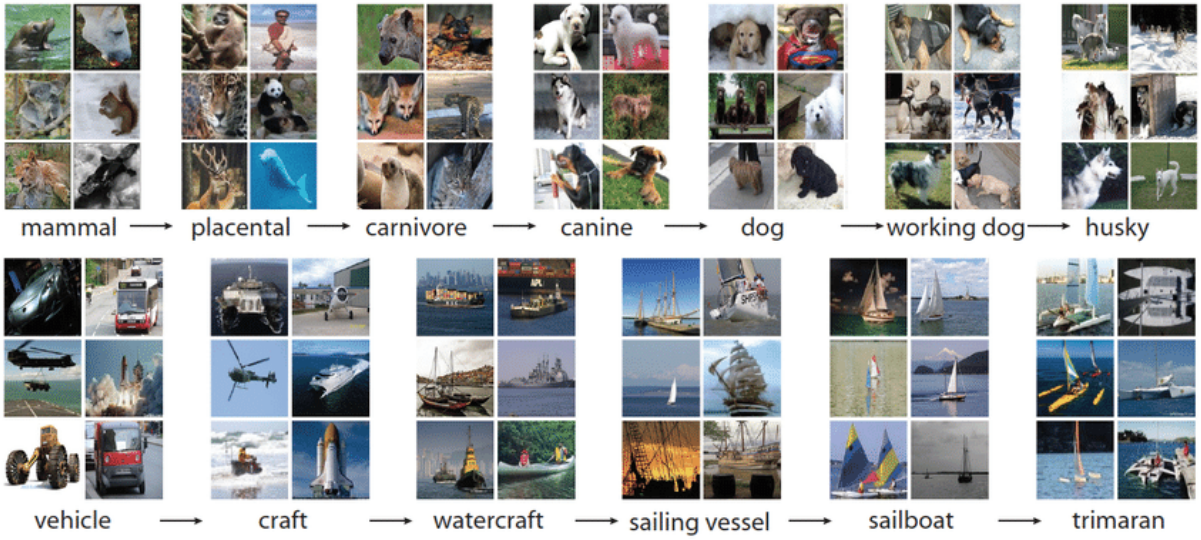


Fig. 2.13 Some sample images within the ImageNet dataset (Ye, 2018).

While CIFAR-10's appeal lies in its simplicity and comprehensibility at the expense of broad generalizability, ImageNet's appeal is the exact opposite. The ImageNet dataset (Deng et al., 2009; Russakovsky et al., 2014) is a collection of 3.2 million images, released in 2009 and based on the WordNet semantic hierarchy. This semantic hierarchy organizes words into a tree, wherein each word in the tree could also be meaningfully described by every word in its antecedent nodes. For example, the traversal to the word 'computer' is:

$$\texttt{artifact} \rightarrow \texttt{instrument} \rightarrow \texttt{device} \rightarrow \texttt{machine} \rightarrow \texttt{computer}$$

while the traversal to the word 'skateboard' is:

$$\texttt{artifact} \rightarrow \texttt{instrument} \rightarrow \texttt{conveyance} \rightarrow \texttt{vehicle} \rightarrow \texttt{wheeled vehicle} \rightarrow \texttt{skateboard}$$

For more more traversals with accompanying images, see Figure 2.13. The images within the ImageNet data are labelled in this manner, which means the label of each image conveys not only the contents of the image but its place within the overarching semantic tree as well. There are 21,841 distinct labels, and as such ImageNet contains an extraordinary catalogue of objects by which to evaluate the capabilities of a computer vision model. Models trained on the entire 14 million image ImageNet dataset would generalize extraordinarily well to other image classification problems due to its implicit comprehensiveness; the model has likely already seen whatever it is that the new problem would ask it to classify.

There is a variety of tasks based upon this dataset, but the most common one in the particular field of deep learning architecture research is the aforementioned ImageNet Large Scale Visual

Recognition Challenge or ILSVRC, specifically the image classification challenge. This consists of a non-overlapping subset of 1000 classes within the full ImageNet data, with around 1000 images per class, for a total data size of 1,281,167 images. The objective is to classify these images, and often both top-1 and top-5 accuracy are given due to the subtler nature of the class distinctions. The record for this problem is 84.3% top-1 and 97% top-5 accuracy, also held by GPipe. Again, parameter count is recorded as well, with ongoing research into pushing the boundaries in both directions.

2.9 Software and Hardware

With the basic theoretical concepts of deep learning outlined, the practical aspects necessary to perform it can be explored.

2.9.1 GPUs and CUDA

The predominant means of performing deep learning at scale is through the use of Graphics Processing Units (GPUs). A GPU is designed to perform massively parallel linear algebra computation for the purposes of digital rendering, however, these exact computations can be leveraged to rapidly acceralate deep learning (Krizhevsky et al., 2012) as compared to a purely CPU-based computation system. CUDA (Compute Unified Device Architecture) is a parallel computation library released in 2006 by NVidia, specifically designed to make the compute capabilities of modern GPUs accessible to user programs. For the purposes of this thesis, CUDA serves as a means of performing very large linear algebra operations very rapidly, such as those used ubiquitously throughout neural networks such as Equation 2.1 or 2.28.

The main limitation to the scale of these operations comes in the form of the GPU's video memory, or VRAM. The inputs, outputs, and intermediate values of any calculation performed on the GPU must be stored in VRAM, which places a limit on the possible parallelization and input size of any calculation. Modern, commercially-available GPUs have on the order of 8 GB to 24 GB of VRAM, which corresponds to between 2 and 6 billion 32-bit floats at maximum that can be allocated at once. While this seems massive, a normal batch of image inputs to a convolutional neural network can have on the order of 200,000 floats, which means there is only space to perform around 10,000-30,000 manipulations of this input. Considering that some neural networks can have upwards of 100 million differentiable parameters, each of which could potentially require a full allocation of another 200,000 floats, the available VRAM space can be consumed very quickly. Management of the available VRAM space is one of the chief design considerations of neural networks, such as to ensure that this space is used as efficiently and effectively as possible.

2.9.2 Torch

The most common way to use CUDA operations is to use a library that interfaces with them, at least in the field of machine learning. This allows easy access to the benefits of CUDA's parallelization ability without the need to work directly with CUDA's GPU APIs in C. The significant majority of published deep learning papers that release their codebases are written in Python, and the most common Python libraries for interfacing with CUDA are PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016). While for the most part the two are very similar in capability, PyTorch's organizational style is particularly clear and elegant, particularly its highly object-oriented approach to model construction. As such, the entirety of the deep learning work in this thesis is written in Python and heavily utilizes PyTorch.

At its core, PyTorch is a tensor processing library that uses CUDA operations to drastically speed up computation time. It also provides a library named `torch.nn`, which builds a neural network construction framework on top of the tensor functionality. PyTorch structures neural models as computation graphs, and provides inference and back-propagation functionality through traversals of this graph. Model inference involves traversing this graph from input to output, and is called the `forward` function of the model. Backpropagation traverses the graph in the opposite direction, automatically computing the gradients of each parameter with respect to the output loss along the way via the chain rule. The backwards pass is aptly named the `backward` function of the model. The primary advantage of using such a library is automatic computation of these partial gradients for a suite of operations that come bundled into the library, which allows users the flexibility of combining multiple components together into new and interesting compositions without worrying about computing their derivational properties.

When writing these compositions, an important thing to be cognizant of is the function trace of the various PyTorch operations being used. Each function within PyTorch is often comprised of a variety of underlying C or CUDA functions, the complexities and number of which vary depending on the mathematical properties of the operation and whether the operation is being run in the forwards or backwards direction. Some operations are deceptive in this regards, with things like tensor indexing (accessing the *n*th item of a tensor) being very quick and efficient in the forwards direction, but painfully slow and expensive in the backwards direction. This means that code optimization in PyTorch takes on a new and unfamiliar dimension; optimizations must be effective in both the forward pass and in the often less tractable gradient calculations within the backwards pass.

One final idiosyncrasy to note is the problem of stuck tensors in the case of VRAM out-of-memory errors. PyTorch automatically and intelligently clears tensors out of VRAM when they are no longer in use, dynamically freeing memory to be reallocated later. However, this intelligent garbage collection breaks down when tensor allocation fails due to insufficient available VRAM; this failure can cause all tensors that currently exist on the GPU to become stuck there, unable to be freed unless the entire Python kernel that originally allocated them is killed. This can be disastrous in cases where these out-of-memory errors crop up late in model training, as this means the model and its weights must be deleted in order to clear the stuck tensors.

2.10 Conclusion

This chapter consitutes the necessary foundations of deep learning and computer vision to understand this work. For more information on what has been covered thus far, Goodfellow et al. (2016) is an excellent reference textbook. Now, the specific world of state-of-the-art computer vision models can be explored, as can the neural architecture search techniques that strive to automatically generate them.

Chapter 3

Literature Review

3.1 Introduction

Having explored common computer vision datasets and covered the fundamentals, design, and usage of neural networks, moving on to more advanced material is now possible. In this chapter, the *taxonomy* of network design will first be covered, defining what constitutes a neural architecture and why it is such a broad problem. Next, the history of convolutional neural network design will be covered, investigating the landmark results and principles that have been uncovered in the 10 years since convolutional neural networks first demonstrated their immense computer vision potential. From this foundation of the cutting-edge of human design, this thesis will then look at algorithmic network design, examining the different approaches that have been tried to automatically design networks more complex and more powerful than anything seen before. However, this power comes with significant caveats and shortcomings, which are then explored in the third section of this chapter. Here, the flaws and fundamental scientific unsoundness of a subset of these algorithms is discussed, as well as the lessons that can be learned from these flaws.

3.2 On Neural Architectures

When defining more complex networks such as CNNs or RNNs, the operations listed in Chapter2 such as convolutions or poolings can be put together in any order, and the data can be defining to flow through these components in any number of ways. A good analogy is that of a circuit; we have various components (resistors, capacitors, inductors, diodes, transistors) that are be connected by wires, and finding novel combinations and connectivities of those components has constituted the majority of human technical discovery in the last century or so. Neural architectures are very similar; we have a huge gamut of components (each with a variety of possible specifications, like kernel size or output dimensionality), each of which can be chosen an arbitrary number of times, and then these components can all be linked up in an infinite number of ways. One way to conceptualize this infinite design space is that of graphs as in

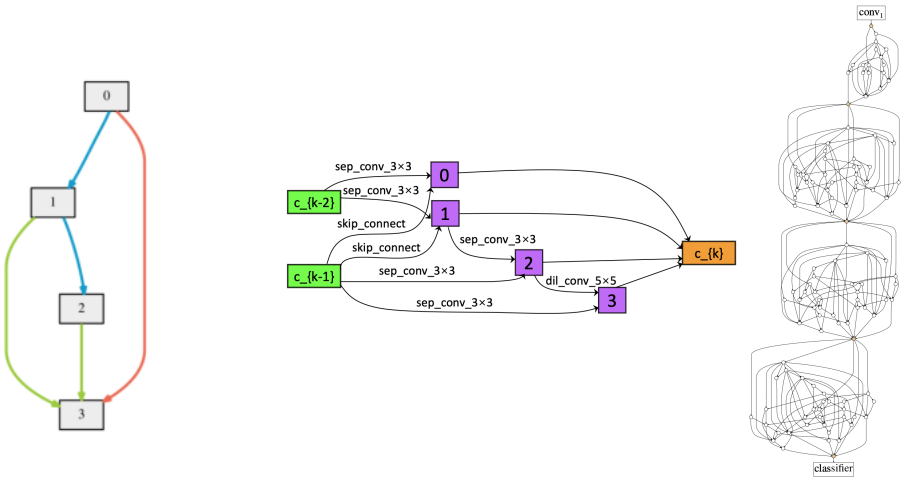


Fig. 3.1 Various graph representation of architectures, from Liu et al. (2018); Xie et al. (2020); Xu et al. (2020).

Figure 3.1, where nodes represent data aggregation (i.e., summation or concatenation) and edges represent data modification via operations like convolutions or poolings.

In this representation, any arbitrary graph can be converted into a neural architecture simply by defining which aggregations happen at each node and which operations lie along each edge. The problem of architecture design stems from the fact that the particular choice of graph, of operation-edge assignment, of nodal aggregation, all affect the final efficiency and performance of the network on the chosen task. As such, finding a good design is crucial to finding effective models. However, picking a good model in this infinite design space of all possible graphs can feel like finding a needle in an infinite haystack, and as such it is common to limit the the network design space into a *search space*: the set of all possible models that are eligible to be designed by a particular algorithm or search strategy. The search space places constraints on the possible models that may be designed, for example, that graphs must be acyclic or that only one edge can connect any two nodes. Meanwhile, textitsearch algorithms are methods that pick a specific architecture from within the set defined by a search space.

Since network design is such a crucial factor in its efficacy, the history of deep learning advances is one of design advances. This chapter will explore both the manual design leaps that brought the field to where it is today, as well as the automated design methods that seek to find needles within the infinite haystack for us.

3.3 Manually Designed Convolutional Neural Nets

3.3.1 AlexNet

Prior to early 2010s, convolutional neural networks were mostly a computational novelty. They had shown promising results on small problems such as MNIST (LeCun et al., 1989), but were so computationally expensive to train that they were infeasible to use for larger problems like CIFAR-10 or ImageNet. However, in 2012 Krizhevsky et al. published “ImageNet Classification

with Deep Convolutional Neural Networks" which realized the potential that graphics processing units or GPUs had to radically speed up training. The insight was thus: GPUs are designed to perform large scale matrix and vector computations very efficiently in parallel, which are the exact computations which comprise the majority of tensor operations within deep learning models. Krizhevsky et al. wrote GPU implementations of the operations necessary to run a convolutional neural network, dramatically speeding up the training of their model. Their model had 60 million trainable parameters between five convolutional layers and three fully-connected layers, distributed across two GPUs via a specialized parallel network architecture. This was necessary to compensate for the relative lack of memory on GPUs at the time, and thus one GPU was tasked with handling the top of the image and one the bottom, with data shared between the two at limited points through the model.

Within this architecture they introduced "new and unusual" components to the convolutional neural network. The first of the components were ReLU nonlinearities, which were added due to their desirable non-saturating quality as described in Chapter 1. Next was local normalization, which served to normalize values over a local spatial area. The authors acknowledge that while ReLU nonlinearities do not explicitly require normalization of their values in the same way that saturating functions do, normalization still provided boosts to the model's test accuracy. Finally, they used overlapping pooling, which was simply a pooling operation where the stride size was less than the kernel size. This was in contrast to the predominant usage of pooling at the time, where stride size was strictly equal to kernel size, but they found that overlapping pooling helped guard against over-fitting. The former and latter of these innovations are now ubiquitous in modern network design, while local normalization has been superseded by batch normalization.

Their model was trained with label-preserving transformations and dropout, and over the ILSVRC subset of ImageNet found a top-1 test error of 37.5% and 17.0% top-5 test error. This was enough to win the ILSVRC-2012 challenge (Deng et al., 2009) by a landslide, with their top-5 test error over 10 percentage points better than the runner-up. This was the groundbreaking result of the paper; that deep convolutional networks were the new state-of-the-art in computer vision, and thus the computer vision revolution began.

3.3.2 VGGNet

The next architecture to bring about design innovations that would cement themselves into the foundation of modern network design was VGGNet, published in 2014 by Simonyan and Zisserman. The first design choice was to exclusively use operations with small kernels, with every operation in the entire network having a kernel size of 1x1 or 3x3. This is in contrast to earlier networks such as AlexNet, which used kernel sizes up to 7x7 or 11x11 in the initial stages of the network. Simonyan and Zisserman point out that while the larger receptive field of the single 7x7 kernel is advantageous for processing larger details in the image, a stack of three 3x3 kernels maintains the same receptive field. See Figure 3.2 for a 1D example of this phenomenon.

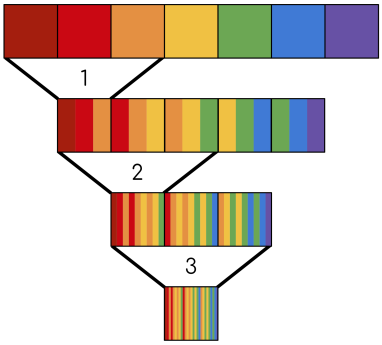


Fig. 3.2 The receptive field of a stack of three 1D convolutions, each with a kernel size of 3 pixels. Notice the stacking effect of the convolutions; by the third filter a single pixel contains information (in this example, the original color) from all 7 original pixels.

The stack of 3x3 kernels requires fewer parameters than the 7x7 case ($3 \times (3 \times 3) = 27$ vs $7 \times 7 = 49$)), and nonlinearities can be placed between each layer of the stack, allowing for much greater flexibility in the functions that can be approximated by the stack.

The second design choice was to ensure that in each location where the spatial dimensions of the data are halved through the use of a stride-2 pooling operation, there is a corresponding upscaling of the channels by a factor of 2. The rationale for this is explained in a later paper by He et al. (2015); the computational complexity of the operation is preserved. The reasoning is thus: halving the spatial dimensions of an image reduces the number of available locations to convolve by a factor of four. From equation 2.28 (replicated below), the number of computations required to compute a single channel C_n^{out} of a multichannel convolution scales by factor of M:

$$\mathbf{C}_n^{out} = \sum_{m=1}^{M} (\mathbf{C}_m^{in} \circledast \mathbf{K}_{nm}). \tag{3.1}$$

Therefore, the number of operations to compute C_n^{out} for all $n \in 0, 1, \ldots, N$ scales by $N * M$. By doubling the input and output channels of the operation, this complexity becomes $2N * 2M = 4NM$, thus compensating for the 4x reduction of spatial locations. While there is no analysis to motivate this choice from a model performance perspective in either paper, its conceptual tidiness has caused it to become a widespread convention in network design.

Using these concepts, Simonyan and Zisserman present networks of varying scales, the largest being 19 layers deep. They found that depth was a crucial factor in terms of network performance, with their deepest models performing best. Their models were evaluated over the ILSVRC ImageNet dataset with a number of training schema, and their best performing configuration achieved 23.7% top-1 accuracy and 6.8% top-5 accuracy.

3.3.3 Inception

Published just two weeks after the VGGNet paper was "Going Deeper With Convolutions" by Szegedy et al. (2014). The paper starts by noting that the deeper-is-better conclusion reached by VGGNet comes with major drawbacks; exponentially increased computational

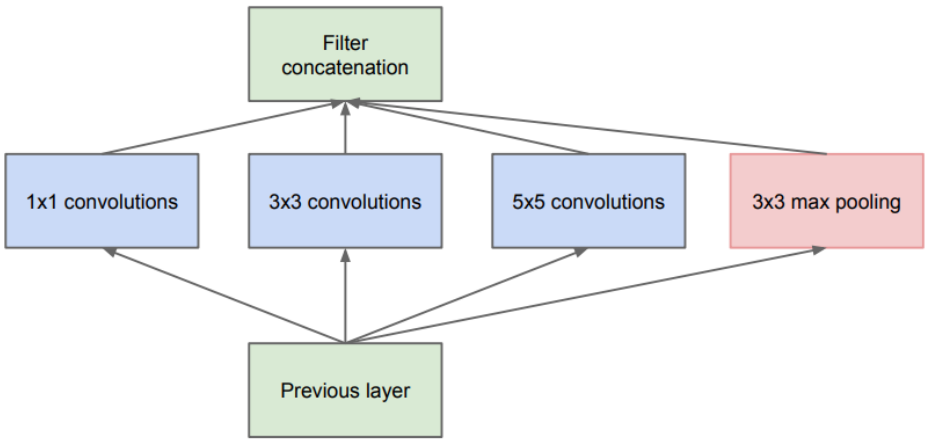


Fig. 3.3 The Inception cell as shown in the original paper.

complexity and increased sensitivity to overfitting. They conclude that to fix these issues, network design must move from dense connectivity, where every filter is connected to every subsequent filter, to sparse connectivity. However, due to the relative computational inefficiency of sparse tensor operations, Szegedy et al. instead attempt to emulate sparse connectivity through dense operations, replacing a single convolutional layer with a parallel set of 4 operations (1x1 convolutions, 3x3 convolutions, 5x5 convolutions, and 3x3 max pooling) dubbed the Inception cell, as shown in Figure 3.3.

In the simplest form of the Inception cell, each of the four parallel operations receives an entire copy of the input information, and outputs a tensor containing a certain fraction of the original channels. This fraction is chosen such as to weight this operation against the others in the parallel group, and this weighting is varied throughout the network. The distribution of weights is chosen in accordance with the authors' hypothesis that the data passing through early layers contains small clusters of local correlation, much like the arguments in Krizhevsky (2009) on data correlation of real images. These small clusters are then distributed into larger patches later in the network by the convolutional stacking phenomenon shown in Figure 3.2. As such, 1x1 convolutions constitute a large fraction of the earlier layers of the model, having the smallest kernel size of the available operations. Later layers distribute the workload more to the operations with larger kernels like the 3x3 and 5x5 convolution. The output tensors from the four parallel operations are then concatenated together channel-wise, returning a tensor with the same shape as the original output. This emulates the filters acting over the tensor sparsely, in that only certain parts of the output tensor have been operated on by any specific filter. The Inception cells are then stacked nine times in the final model, with pooling layers distributed throughout.

In addition to the novel non-linear architecture of the cell itself, the paper also introduces the concept of auxiliary classifiers. These arise to solve the vanishing gradient issue through such a deep network; the signal from the loss function needs to be able to traverse the entirety of the way back up the model in order for every component to learn effectively. In large networks, this signal often degrades before it can reach the upper layers of the model, meaning many parameters end up being wasted. To mitigate this, additional classifiers are placed at

intermediate points throughout the model. The predictions outputted by these auxiliary classifiers are passed through the loss function, and a small fraction of this loss is added to the overall loss of the model. This incentivises the model to find locations in the loss landscape that produce adequate performance from the auxiliary classifiers in addition to good performance from the final classifier. Essentially, auxiliary classifiers partition a single model into an overlapping group of models that each individually need to be effective. Furthermore, the auxiliary classifier introduces multiple sources of gradient signal all throughout the model, allowing training to happen much more effectively in the earlier layers of the model. This allows the Inception model to perform well at depths previously unseen in network design, with around 100 convolutional layers in the final model.

The model is evaluated over the ILSVRC ImageNet subset, and the best single model found a top-5 error rate 10.07%. Their best overall score came from an ensemble of seven models, which together achieved a top-5 error rate 6.67%, which won the 2014 ILSVRC competition. This performance demonstrated that non-linear networks were one potential path[1] towards to unlocking higher performance of convolutional neural networks, and furthermore that extremely deep models were now feasible through the use of auxiliary classifiers.

3.3.4 ResNet

The next year, another network further demonstrated the abilities of non-linear networks, and that was ResNet, published by He et al. The authors open the paper by commenting on the conclusions drawn from models like Inception and VGGNet, that deeper models tend to be better performing ones. From this, they ask if more performance can be found by simply making an even deeper model. However, they note the same vanishing gradient problem as raised by the Inception authors; the gradient signal does not reach the upper layers of the model. To demonstrate this issue, they set up a thought experiment: imagine a shallow network A that performs well on a given problem, and a deep network B constructed by adding layers onto A. There exists a learned configuration for network B such that each added layer reduces to the identity, and thus this configuration would have equivalent performance to network A. However, despite the existence of such a solution, existing training methods and networks are unable to reach it.

To address this problem, the authors configure the network to explicitly model residual functions, changing the function modeled by from $F(x)$ to $F(x)+x$. Pairs of stacked convolutional layers are chosen as the granular 'function' of the configuration, used to model some $F(x)$. Inputs are passed into the convolutional group, but are also copied along an identity connection that circumvents the operations altogether. The output of the convolutional group is then summed into the identity connection, thus implementing the residual function $F(x)+x$. This configuration is visualized in Figure 3.4.

The rationale behind the design decision is that when these blocks are stacked, it creates an unbroken chain of identity connections that runs the entire length of the model. Gradient

[1] A set of branching paths, one might say.

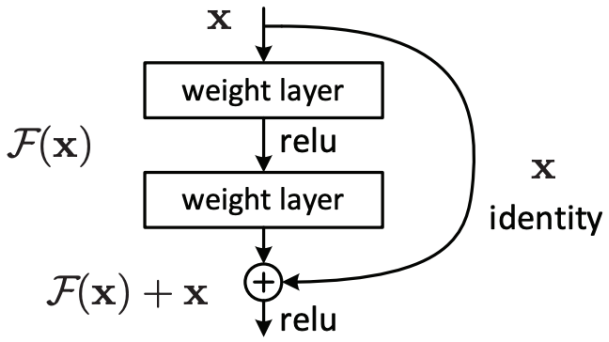


Fig. 3.4 The ResNet block as shown in the original paper. Notice the two discrete paths of the configuration: the convolutional operation stack $F(x)$ and the identity x.

signals can thus pass backwards through the model unhindered and undiluted through the identity operations, while still providing meaningful gradient data through the convolutions. In addition to this block, a second configuration is designed called the bottleneck block. This block contains three convolutional layers, the first a 1x1 convolution that reduces channel dimension, second a 3x3 convolution at these smaller dimensions, and third a 1x1 convolution to return the channel dimension to the original value. These serve to reduce the parameter size and GPU memory footprint of the models. In the paper, models are constructed that use the non-bottleneck blocks in 18 and 34 layer configurations, and bottleneck blocks for 50, 101, and 152 layer configurations. All configurations are then evaluated over the ILSVCR ImageNet subset. If the hypothesis that residual connectivity mitigates the vanishing gradient problem is true, then the deeper and therefore larger models should have a greater ability to model the overall problem, and therefore have better performance. To further confirm this, non-residual models are also evaluated in 18 and 34 layer configurations, identical to the 18 and 34 layers residual models but with the identity connections removed. In the non-residual configuration, the deeper 34 layer model performed marginally worse in both train and test accuracy than the 18 layer model, around 0.6–1% percentage points worse on both data splits. The opposite was true for the residual configurations; the 18 layer residual model performed comparably to the 18 layer non-residual model, but the 34 layer residual model scored 3.8% percentage points better in test accuracy and around 6% better in train accuracy. The full results of the 18 and 34 layer comparison are shown in Table 3.1:

	Train Error*	Test Error
18 Layer	$\approx$ **31**%	**27.94**%
34 Layer	$\approx$ 32%	28.54%
18 Layer Residual	$\approx$ 31%	27.88%
34 Layer Residual	$\approx$ **25**%	**25.03%**

* Train errors estimated by eye from charts in the paper.

Table 3.1 The test and train error for each of the four configurations in the non-residual/residual comparison test performed in He et al. (2015).

These results confirm that the residual blocks do allow for significant improvements in both learning and generalization performance in deeper models. Much like the results of the Inception paper, it suggests that the additional parameters of deeper models can be very powerful, if a method can be found to allow for their effective training. However, upon analysis of these results the authors find that the non-residual models did not suffer from vanishing gradients despite performing poorly; the batch-normalization layers within the model ensured that there was always nonzero gradient during backpropagation, and gradient signals were "healthy" through the entire model. The authors speculate that the difficulty in training large non-residual networks is not due to vanishing gradients, but rather that they just converge very slowly. Whatever the problem may be, it is clear that the residual connections manage to resolve it.

The three larger bottleneck configurations of 50, 101, and 152 layers show further gains, with performance increasing monotonically with size up to 19.38% top-1 error and 4.49% top-5 error in the 152 layer case. This final model set the record at the time for the best single model performance on ImageNet, while an ensemble of ResNet-152s also set a record for the ensemble setting. These results further confirm that the residual block unlocks effective training for very large models, and that deepening models is a rich vein to mine for better performance.

3.4 Neural Architecture Search: First Generation

3.4.1 NAS-RL

Designing neural architectures is an immensely expensive task, in terms of time, cost, and intuition. Testing a design iteration can take multiple days of compute time on specialized hardware, and this process can be essentially endless, as the space of possible architectures is infinite. Furthermore, neural architectures of any meaningful scale are black boxes, which makes deciding which design change to make and when is a difficult, amorphous problem. That means that there is no guarantee that any design change made is one in a positive direction, and there is no quick indication that any architecture has reached the upper bound of possible performance on a problem. These barriers to entry mean that access to state-of-the-art networks is either granted through incredible luck, or more commonly, immense resources. Those who can afford to throw endless computational time at a problem can explore more of the search space, experiment with new configurations, push the boundaries, and those that are unable to cannot utilize the power of deep learning to its fullest. However, the problem of network design is very similar to that of of feature engineering; an immensely difficult and nebulous task that required a lot of time, experimentation, and intuition. However, neural networks have managed to almost entirely automate feature engineering away. The natural question is therefore: can the problem of architecture design be automated too?

While this question had existed for a while, the explosion of GPU-accelerated deep learning in the first half of the 2010s brought it to the forefront, and a 2016 paper by Zoph and Le gave

its name to the answer: Neural Architecture Search or NAS[2]. Zoph and Le's paper, fully titled "Neural Architecture Search with Reinforcement Learning", was the first to automatically design an architecture that could compete with the best human designed architectures. Their approach revolves around condensing information about network design such as filter dimensionality and connectivity into a string encoding, which can then be passed into a recurrent neural network dubbed the "controller". The controller outputs string encodings, which are then decoded into a "child network" that is subsequently trained and evaluated. The performance of this child network is then mapped back to the RNN controller, forming a reinforcement learning cycle. For each child network evaluated, the controller gains more information about what constitutes a successful neural architecture for that specific problem. In theory, this information allows the controller to generate a better formed model on the next cycle, iteratively improving the designed architecture.

For convolutional networks, NAS-RL designs convolutional layers one at a time, selecting filter height, width, stride, and total filter number for each. Additionally, the inputs of each layer can be completely specified. Child networks are trained for 50 epochs, and the validation performance over the last five epochs is used as the reward signal back to the controller. For CIFAR-10, the best model designed by their controller had 37.4 million parameters and scored 96.35% accuracy over the validation set, which at the time surpassed the record for human designed architectures of 96.26%. However, this best model was the result of both search space and architecture tuning, while the best entirely autonomously designed model had 7.1 million parameters and scored only 95.53% accuracy. The full results of this model and all others described later in this chapter are compiled in Table 3.2.

However, these results come with a caveat; they required training of 12,800 architectures across 800 GPUs to achieve. Since each architecture is trained for 50 epochs, that is 640,000 training epochs total required to run their algorithm. At a relatively conservative estimate of one GPU minute[3] per epoch, this is 1.21 years of GPU time at minimum necessary to achieve this single result. In a later paper (Zoph et al., 2017) they clarify the runtime, stating they used 800 GPUs for 28 days, which corresponds to 537,600 GPU-hours or just over 61 GPU-years. If the 61.93 years of computation was not enough of a barrier, the cost of this computational time on Microsoft Azure could run to around $2,150,400 (Microsoft Azure, 2021) at the time of writing. As such, despite being a highly successful proof of concept, Zoph and Le's algorithm did not come close to solving the original problem; to make deep learning cheaper, faster, and more accessible.

[2]At the time of original publication of Neural Architecture Search with Reinforcement Learning, the term "NAS" solely referred to the algorithm invented by Zoph and Le. However, as time passed, the term NAS grew to refer to the entire field of automated neural architecture design, and so henceforth I will refer to Zoph and Le's algorithm as NAS-RL to avoid confusion.

[3]GPU minutes/GPU hours/GPU days/etc are metrics used to normalize the effects of multi-GPU parallelism; they refer to the total length of time an algorithm would take were it run on a single GPU.

3.4.2 Neuroevolution-NAS

While using reinforcement learning to design neural architectures did produce effective architectures, the need for a recurrent controller does add significant conceptual complexity to the search process. For example, questions like, "How do we investigate whether the controller is making sensible design decisions?", or "How do we design an optimal architecture of the controller model?" arise with no clear answer provided. By turning to a simpler design algorithm these issues are avoided, and in 2017 Real et al. published "Large-Scale Evolution of Image Classifiers"[4] which used an evolutionary algorithm to evolve a pool of architectures towards state-of-the-art designs, thus creating the second school of neural architecture search: evolution. The advantage of evolutionary algorithms is that there is no need for any external controllers, in that the algorithm runs in an unsupervised, autonomous way. Additionally, the design trends preferred by the algorithm and the progression of those trends is relatively easier to interpret than is the case with reinforcement learning, simply by sampling the most favored individuals from the gene pool at any given step of the algorithm.

The crux of Real et al.'s method is the evolutionary algorithm. They define their population as composed of 1,000 individual trained architectures, with that model's validation accuracy as their fitness. The operation of the algorithm is performed in parallel, with each concurrent worker selecting two random models from the population and then sampling their fitness. The better performing model of the two is selected for 'reproduction', which entails cloning the parent model and mutating the architecture of the cloned child. Both the parent and child are then returned to the population for future selection. The worse of the two originally selected models is 'killed'; i.e, immediately removed from the gene pool.

Also crucial to the design of an evolutionary algorithm is the exact definition of an individual and the specific mechanism of each individual's reproduction. Real et al. use a directed acyclic graph to represent each individual, with vertices representing activation functions and edges representing tensor operations. These operations are either identities or convolutions, with various hyperparameters that control their specific mechanisms of operation. Mutations over these architectures are then randomly selected from a set of 11 available mutations, which either modify the graph connectivity, insert operations, adjust the hyperparameters of existing operations, or alter the training dynamics of the model. Within each mutation there is further random variability; for example, the operation insertion mutation randomly selects different operation parameters like stride and batch-normalization presence. This means that the true pool of mutations is multiplicatively larger than 11, providing an immense range of possible options for variation. The intent of each of these mutations is that they each emulate the design tweaks a human designer might make when trialing new architectures, such as to constrain the possible mutations into a familiar range.

[4]Real et al. (2017) does not provide any name for their algorithm beyond "Evolution". Future papers in the field used the term "neuroevolution" to describe the algorithm, however, this could be confused with the widely used neuroevolution genetic algorithm. To disambiguate, this thesis will use "Neuroevolution-NAS" to refer specifically to Real et al.'s algorithm.

To determine fitness of any particular individual in the population, Real et al. train each model on CIFAR-10 for roughly 28 epochs using a fixed learning rate and identical hyperparameters across each run. After training, the model is evaluated over a held-out validation set of 5,000 examples, and this performance determines the evolutionary fitness of this candidate. Depending on the size of the individual model, this training process can take anywhere from a few seconds to a few hours.

Assuming a runtime somewhere in the middle of this range, evaluating the entire population of 1,000 individuals would take on the order of 40 GPU days. To mitigate this cost, the authors use a technique they call "weight inheritance" to minimize training time. For each mutation, they identify the minimum amount of new weights that need to be reset or created in the child model. In the case of adding a new operation, all other operation weights are preserved. The same goes with operation modification mutations, only the weights of the modified mutation are reinitialized. This minimizes the training cost of the new models, as the majority of their weights are shared from their parent models.

The novelty of this algorithm is that it starts from entirely trivial conditions; a population of 1,000 single-layer models with no convolutions. The authors sought to identify whether a simple evolutionary algorithm could rival the learned intuition that NAS-RL could provide. To thoroughly explore this question, the authors test their algorithm five times and compare the performance against a randomly guided evolution. On average, Real et al.'s algorithm found an architecture with mean CIFAR-10 test accuracy of 94.1%, and roughly 5.4 million parameters, while the random algorithm's best architecture scored only 87.3%. These performances, while not matching the performance of NAS-RL, do indicate that evolved models from trivial conditions are indeed at least competitive with the state-of-the-art. However, the main caveat to all these results is the time cost of the evolutionary algorithm, reported as a total of 250 hours across 250 parallel workers. This corresponds to a true cost of around seven GPU years, which while monumentally cheaper than NAS-RL's 61 GPU years is still too expensive for the average user.

3.4.3 SMASH

While NAS-RL and Neuroevolution-NAS both demonstrated promising results, their main flaw was the prohibitive runtimes incurred by the training of thousands of candidate architectures. One month after the publication of Real et al.'s Neuroevolution-NAS paper, Brock et al. published "SMASH: One-Shot Model Architecture Search through HyperNetworks", which was one of the first papers to place emphasis on minimizing the expensive candidate model training necessitated by other contemporaries. To do this, Brock et al. frame the neural architecture search problem as follows: given a certain candidate pool, the goal of NAS is to discern the best model within. There is no need for the evaluation of these candidates to entail their comprehensive training or even to produce accurate predictions of their performance, so long as the evaluation is rank-consistent between predicted performance and true performance. This is to say, a Neural Architecture Search algorithm only needs to correctly predict performance rank to be effective; accuracy is entirely unimportant. This is the perspective from which SMASH approaches the neural

architecture search program, seeking to evaluate each candidate's approximate performance by approximating their trained weights.

To do this, they make use of a "HyperNet", an external neural network that, when given an architecture encoding as input, outputs trained weights for that architecture. This process consists of two main components; the architecture encoding and the weight prediction. To encode architectures of arbitrary convolutional models, Brock et al. define a "memory-bank" representation of networks, which conceptualizes an architecture as a series of reads and writes to discrete memory-banks. For example, each operation within a simple serial architecture would load a tensor from the memory bank, perform some operation, and then overwrite the memory-bank with the new tensor for use by the next operation. This model would thus be represented as a series of reads and writes over the same memory-bank within the memory-bank representation. The advantage of the memory-bank representation is that it lends itself well to a binary encoding. In SMASH, the memory-bank is encoded as a three dimensional tensor, which is comprised of a variety of adjacency matrices and one-hotted properties that are stacked and combined to form a single tensor.

This tensor serves as the input to their HyperNet, which takes the encoded architecture as input and outputs a large tensor, which is then mapped into various architecture weights via a specific slicing scheme based on the layer configuration. The HyperNet is trained by looping through the following steps a number of times:

1. Sample random architecture c and data batch x.
2. Pass random architecture through Hypernet H to find output weights $W = H(c)$.
3. Backpropagate error of $c(x \mid W)$ backwards through c and H.

With the Hypernet satisfactorily trained, the weights of any arbitrary number of networks can now be generated rapidly, which in turn allows for those networks' performances to be ranked. Assuming that the performance ranks from the generated weights correlates to the true performances of the model, the best model from this sampling can then be trained fully, and thus the final model is found. To evaluate the validity of the correlation assumption, Brock et al. train 50 random CIFAR-100 architectures and compare their true trained validation performance against their SMASH-weighted validation performance, which is shown in Figure 3.5. While not perfect, the correlation is remarkably strong, despite the minimal accuracy of the SMASH-weighted performances.

Another crucial experiment the authors perform to evaluate their algorithm is to test the quality and specialization of the weights generated by the HyperNet. To do this, they take the input encoded architecture c and perform some corruption of the encoding, generating a new encoding c'. The weights generated by the HyperNet $W = H(c')$ are then used to set the weights of original architecture c. This means that the HyperNet "thinks" it is generating weights for an architecture c', but is actually generating weights for c. If there are no meaningful performance differences between c and c' over a variety of corruptions, the mapping learned by H is not truly learning to tailor weights to specific architectures, but is rather a favorable general case that always produces decent results. However, the authors find that the best performance is consistently found when $c = c'$, that is, when the model's expectation of the architecture it is

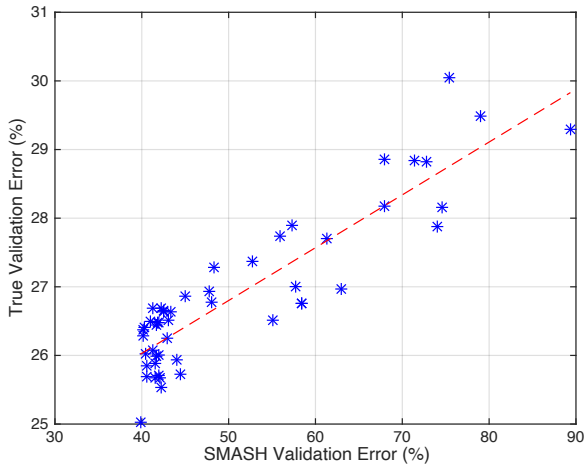


Fig. 3.5 Correlation of SMASH-weighted model performance versus their regularly trained performance, as shown in Brock et al. (2017).

being asked to generate weights for matches the actual architecture used. This implies the model does take into account the architecture at hand, and is producing specialized weightings.

With the methodology validated, Brock et al. report the performance of SMASH. They report two different CIFAR-10 performances, corresponding to two different HyperNet configurations and search space parameters. The first, named SMASHv1, achieves a 94.47% test accuracy with 4.6 million parameters, while the second scores 95.97% accuracy with 16 million parameters. While these results are again surpassed by those of NAS-RL, the runtime of SMASH is the standout advantage. While not directly reported in the paper, other sources have reported the runtime to be around 1.5 GPU days (Ren et al., 2020), which makes sense; not needing to fully train thousands of candidate architectures should drastically reduce the time costs. Furthermore, the entire algorithm runs on a single GPU, which significantly lowers the hardware costs and overhead compared to the massively parallel setups seen in competing algorithms. This is the first algorithm to demonstrate that competitive results can be attained on the order of days, not months or years, thus setting a very high bar for search efficiency.

3.4.4 PNAS

One final contribution to the first generation of neural architecture search algorithms is Liu et al.'s 2017 paper, "Progressive Neural Architecture Search", which notes the immense cost of Zoph and Le's NAS-RL and Real et al.'s Neuroevolution-NAS. They argue that the need to train thousands of candidate models is not only overly wasteful in time cost, but also simply an inefficient means of traversing the search space. To remedy this, they introduce a "progressive" means of traversing the search space of available models from most simple to most complex. This entails dividing the construction of each possible model into its component building blocks, with the simplest models comprising of just a single block while more complex models are comprised of many blocks connected in more complex patterns. From this search space, the set of the simplest models is identified, i.e, the ones that only require a single block. This is a tiny subset of the full search space, and it is relatively easy to fully train each of them in parallel. Each

of these simplest models is then expanded upon, by looking at the models that can be created by adding a single extra block to the existing configuration. This rapidly expands the space (there are around 147,000 candidate models after only two blocks), which makes fully training each of the candidate models untenable. To circumvent this issue, Liu et al. apply a learned predictor, which seeks to predict the performance of a model given its block structures. This predictor takes the form of a recurrent neural network, that takes the model configuration encoded in string-form as its input much like Zoph and Le (2017). From the set of expanded models, the predictor selects the K most promising, where K is chosen such that K models are quick and practical to fully evaluate in parallel. This process continues until a sufficiently complex and performant model is discovered. This found model is then fully trained according to a fixed set of hyperparameters.

In essence, PNAS performs a depth-first search of the search space; eliminating entire branches of the search tree if they are not favorably viewed by the predictor and instead traversing downwards towards promising leaves. From this heirarchical search tree perspective algorithms like NAS-RL and Neuroevolution-NAS appear starkly inefficient; their traversal of the search space does not take advantage of the structures of the space that can be exploited in the name of optimization. To elucidate the advantage provided by the heirachical search, Liu et al. compare the total number of model trainings necessary between their algorithm and that of NAS-RL, and they find their algorithm requires 1.8 times fewer trainings then NAS-RL to find competitive architectures, despite traversing an identically sized search space.

Liu et al. run the PNAS process five times over a variety of search space and model parameters, and report the CIFAR-10 results of the five models identified, each of which are trained 15 times with different random seeds and initializations. The best model, titled PNASNet-5, achieved a mean test accuracy of 96.59% with 3.2M parameters. However, these results share a similar caveat to those seen earlier, which is in their hardware and runtime requirements. Crucial to the PNAS algorithm is the parallel evaluation of around 1,200 models, which requires either an immense number of GPUs or a very large runtime. While the exact runtime of their algorithm is not reported, they do claim a $15\times$ speedup compared to NAS-RL, which would imply a runtime of around 2,500 GPU-hours. Again, despite excellent performance, the algorithm is out of reach to the average user.

3.5 Neural Architecture Search: Second Generation

With algorithms like NAS-RL, Neuroevolution-NAS, and PNAS demonstrating that automated network design was indeed possible, and the results of SMASH showing that efficient search can happen on the order of days, not months or years, the first generation of neural architecture search demonstrated the promise of the field in both raw performance and efficiency. However, no algorithm had succeeded in both dimensions to any significant degree, which left the door open for a new class of algorithms that could achieve state-of-the-art results in minimal runtimes. To fill this hole came the second generation of neural architecture search algorithms, which all

sought to update and optimize existing algorithms or pioneer entirely new ways of mitigating the faults of the prior generation.

3.5.1 NASNet

First of the second generation of NAS is Zoph et al.'s 2017 paper "Learning Transferable Architectures for Scalable Image Recognition". This builds upon their earlier results of NAS-RL via search space tailoring, through two specific decisions. First, they adapt a cellular design pattern for their models. This is reminiscent of the block concept of models like ResNet and Inception, where a local connectivity pattern (a cell) is designed and then stacked at varying scales to produce a final network. This modularity lends itself to significant flexibility in terms of the final network architecture, the topography and total size of the network can be altered just by changing the number or position of cells. Here, two cells are designed for each final convolutional architecture; a "reduction" cell and a "convolution" cell. In reduction cells, the channel dimensionality of each tensor that passes through the cell is doubled, while the spatial dimensions are halved. Convolution cells do not modify tensor dimensionality in any way. The cellular design philosophy dramatically reduces the search space size, as the algorithm only needs to design two cells rather than a full model. In the specific case of this paper, there are either 14 or 20 cells in a full CIFAR-10 model, thus reducing the size of searched model by a factor of 7 or 10, respectively. Furthermore, Zoph et al. note that an algorithm that returns modular cells is more generalizable than an algorithm that produces a monolithic architecture, in that these cells can be stacked in novel ways and at various channel dimensions to suit different tasks. In Zoph et al.'s experiments, they choose two reduction cells for CIFAR-10, stack N normal cells before each reduction cell, and place an additional N normal cells after the final reduction cell.

Second, Zoph et al. limit the search space down to a fixed set of 13 operations[5] and connectivity patterns. Each cell receives the output of the two previous cell h_i and h_{i-1} which are dubbed the first two hidden states, and constructs cells as follows:

1. Select two hidden states from h_i, h_{i-1}, or from the hidden states created in earlier iterations.
2. Select two operations to apply to the first and second chosen hidden states, respectively.
3. Select either element-wise addition or concatenation to combine the two operation outputs.
4. Add the combined outputs to the available hidden states.

By repeating these steps a certain number of times, cells of varying sizes and connectivities can be constructed. In their experiments, Zoph et al. settle on five iterations to produce consistently good results, therefore producing cells with around 10 operations each.

This design pattern, specifically the fixed operation set, cellular design pattern, and specific connectivity pattern, is codified as the NASNet search space, to provide a consistent baseline comparison between architectural search techniques. This is important such as to differentiate between the improvements brought by better search spaces versus better architecture search

[5]The 13 operations are: Identity, 1×3 and 1×7 rectangular convolutions, 3×3, 5×5, and 7×7 max poolings, 3×3 average poolings, 1×1 and 3×3 convolutions, and 3×3, 5×5, and 7×7 depthwise-separable convolutions.

methods, as both dimensions can produce meaningful impact on results and thus must be disentangled.

After introducing the NASNet search space, Zoph et al. run their earlier NAS-RL algorithm within it. They test a variety of different configurations, but just the best performing run and the most efficient run will be focused on here. The latter named NASNet-A uses $N = 7$, has 576 channels in the first layer of the network, and scores a 97.60% accuracy on CIFAR-10 with 27.6 million parameters. The former, NASNet-B, uses $N = 4$, has 288 channels in the first layer, and scores a 96.27% with 2.6 million parameters. While these results are very impressive, the very high runtime cost tempers their real-world applicability. Zoph et al. actually erroneously report their runtime cost in the paper, stating that their algorithm uses 500 GPUs for four days and thus costs 2,000 GPU-hours to run. This is of course a unit error, as they are reporting their GPU-*day* cost, not the correct figure of 48,000 GPU-*hours*. This is around 5.4 years, still massively outside the scope of any average user.

3.5.2 ENAS

In early 2018, Zoph and Le, along with Hieu Pham, Melody Guam, and Jeff Dean sought to address the criticisms of NAS-RL by releasing "Efficient Neural Architecture Search via Parameter Sharing", dubbed ENAS. They note that the main inefficiency of NAS-RL was the requisite training of each child network produced by the controller. To remedy this, they introduce the concept of a weight-shared supernet. The idea is that instead of creating thousands of independent child networks, one should instead create a single massive "supernet" such that every possible child network is a subgraph of the parent graph. The supernet as a whole is then trained for a single epoch over the training data. From this point onward, the general concept of the ENAS algorithm is roughly similar to that of NAS-RL; an RNN controller designs neural architectures layer-by-layer, iteratively selecting the parameters of the layer. However, once the architecture embedding is generated by the controller it is instead extracted from the supernet and evaluated over a held-out validation set. The performance on the validation set is then passed back to the controller as a reward. Before generating the next architecture, the supernet is trained for another epoch. These training phases between the supernet and the controller continue to alternate throughout the whole process, until a final architecture is selected. In essence, at any particular training epoch the supernet seeks to model the weights of any particular subnetwork after an equivalent number of epochs, while the controller attempts to identify the most promising subnetwork within the supernetwork at that point in time, with the hope that both will iteratively improve in performance over time. Since there are only two networks to train, the supernetwork and the RNN controller, the whole process can be up to 50,000 times faster than NAS-RL.

The second key concept to the ENAS paper was that of the micro-search space. While ENAS is capable of designing an entire network from the ground up (dubbed the macro-search space in the paper), it is also able to work on a cellular basis, instead designing convolutional and reduction cells that are then stacked according to a predetermined pattern: 21 total cells with reduction cells in positions 7, 14, and 21 and convolution cells in the rest. In the macro-search

space over CIFAR-10, ENAS was asked to design 12 layer networks, the best of which had 38.0 million parameters and scored an accuracy of 96.13% over the CIFAR-10 test set. In the micro-search space, ENAS produced a model with 4.6 million parameters that scored an accuracy of 97.11%.

However, the most important result from ENAS was not its raw performance, but rather the relatively small amount of time and resources necessary to achieve those results: ENAS takes only 16 hours and a single NVidia 1080Ti GPU to run. A GPU of similar performance can be accessed on the cloud for around £1-£2 an hour (Microsoft Azure, 2021), meaning that ENAS entirely remedies the prohibitive costs and runtime of NAS-RL. The sub-1 GPU day benchmark set by ENAS remains an important goal in the field of NAS to this day, with authors constantly competing to further lower the computational cost of NAS (Liu et al., 2018; Xu et al., 2020).

The cost dichotomy between NAS-RL and ENAS represents two possible directions for the field of NAS. The former looks to discover the best possible networks regardless of computational cost, looking to throw functionally infinite resources at the problem to discover the absolute limits of performance (Huang et al., 2018; Pham et al., 2018). The second looks to minimize the temporal, fiscal, and computational outlay of NAS, to bring it as close to the average user as possible. The latter direction is one that seems more magnanimously valuable to pursue; not only is achieving competitive results in the most efficient way possible an intellectually interesting problem, it is also one with a much broader accessibility. There are very few entities out there that could justify the outlay that an algorithm like NAS-RL requires simply to produce a single neural architecture, but almost everyone with access to a GPU could run something like ENAS.

3.5.3 Regularized Evolution

Contemporaneously to ENAS, Real et al. published "Regularized Evolution for Image Classifier Search", which sought to improve upon their earlier Neuroevolution-NAS paper. Here, Real et al. note the relatively poor performance of evolved models compared to the state-of-the-art hand-designed ones, and attempt to improve their evolutionary algorithm in the hopes of facilitating the discovery of higher-performing models.

The first difference introduced by their new algorithm comes down to how models are removed from the population; previously, the worse-performing model of the two selected in the random-tournament was discarded. Now, the age of each model in the population is tracked, and the oldest model is removed. In practice, this means that the model that was trained the earliest is removed from the population every time a new model is introduced. To explain the advantage of this approach, Real et al. compare their old, non-aging evolution to aging evolution. Specifically, they introduce the concept of a lucky model, one that arrives at an unreproducibly high accuracy early in the algorithm simply by random chance and might not actually perform all that well when retrained. In non-aging evolution, this result will be maintained as a promising path for evolution and as such will spawn many offspring as the algorithm attempts to refine the architecture. However, since the architecture simply scored well due to lucky training, all the children will likely perform worse and thus be wasted search time. This problem will continue

until the lucky model is removed from the search space, which will only happen when it is compared against a better model, which can only occur if the algorithm has identified such a model. However, the presence of a lucky model in the population will reduce the chance of the algorithm ever identifying a better model, as exploring the children of the lucky model distracts the algorithm from better options.

If such a lucky model is placed into an aging algorithm, it can only remain in the search space for a short period before it ages out of the population. This means that the time wasted exploring its children is limited, and unless it can produce good offspring its architectural genetics will not be preserved. This leads us to the main crux of aging evolution: architectures can only last in the population if they produce consistently good descendants, thus biasing the algorithm towards architectures that are repeatedly well-performing and are fruitful candidates for evolution.

The second key change introduced is to align the search space to the NASNet search space, by ensuring that all initial members of the population conform to the NASNet specifications and that mutations only produce similarly compliant models. This allows for direct comparison against other NASNet-conforming algorithms. Real et al. call the models produced by their algorithm AmoebaNet, and their best model over CIFAR-10 scored a 96.66% accuracy with 2.6M parameters. However, it exacerbates the runtime problems of Neuroevolution-NAS, in that it requires 450 GPUs over seven days, a total of 75,600 GPU-hours or 8.63 GPU-years. This is around a year and a half more expensive than Neuroevolution-NAS, and it is still entirely unfeasible for the average user to reproduce their results.

3.6 Differentiable Architecture Search

3.6.1 DARTS

Despite ENAS' accessibility, it has an implicit inefficiency in the need to jointly train the supernet and the recurrent controller. If the supernet contains all possible architectures, the controller serves only as a means of identifying the most promising subgraph. This is to say; training the supernet already produces the result, it just needs to be isolated in some way. Identifying a way to train the supernet whilst whittling it down to some optimal subgraph would complete architecture search without need for external controllers. This is exactly the reasoning behind Liu et al.'s 2019 paper "DARTS: Differentiable Architecture Search".

DARTS compartmentalizes architectures in a similar way to ENAS, in that it searches for a small, modular neural cell that it can stack into larger networks. These cells are explicitly graph structured in the paper, where nodes represent locations in the model where tensors are combined, such as through concatenation or summation. Edges represent operations over tensors, such as convolutions or pooling operations. Sample neural cells depicted in this graph structure are shown in Figure 3.6.

While there a number of ways to represent architectures via graph (such as the Inception paper, which represents both operations and combinations via nodes and instead uses edges to show data flow), the DARTS abstraction can be exceptionally helpful in conceptualizing the

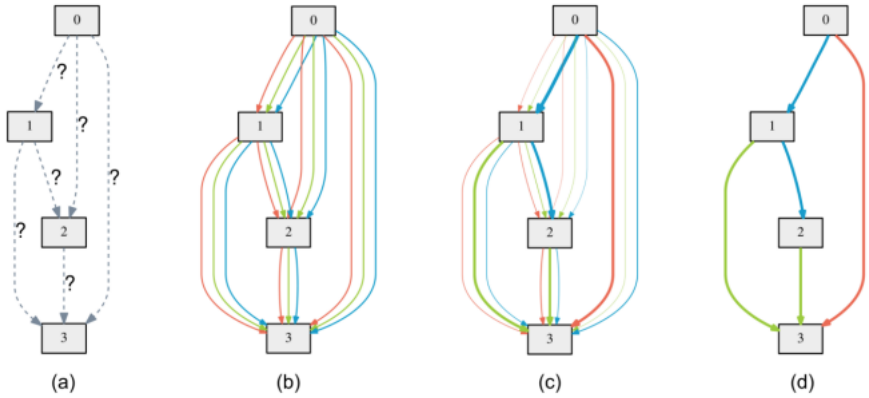


Fig. 3.6 The graph representation used by DARTS as it appears in the paper. Data flows from top to bottom, nodes are tensor combinations, and edges are transformations.

connectivity of nonlinear architectures. From this point onwards all neural architectures will be represented according to the DARTS graph representation unless otherwise specified.

To design a cell, DARTS first needs a set of candidate operations $\mathcal{O}$ and a number of nodes per cell N. These N nodes are then placed into the graph, and numbered sequentially from 0 to N. Then a set of $|\mathcal{O}|$ edges, one per operation in the search space, are placed between each pair of nodes n and m such that $m > n$. This means that each node has outbound edges to every subsequently numbered node, but not to antecedent nodes. Figure 3.6b shows this configuration n the case where $N = 4$ and $|\mathcal{O}| = 4$. This configuration forms the DARTS supernet, where every possible cell containing N nodes and $|\mathcal{O}|$ available operations appears as a subgraph.

The key to whittling this supernet down to the final architecture is translating the discrete choice of operation between each edge into a continuous one. This is accomplished by assigning a weight vector α_o of size $|\mathcal{O}|$ to each set of parallel edges. The weight vector is softmaxed and used to perform a weight sum of the parallel edges. This transforms a set of parallel operations $(o_0^{(i,j)}, o_1^{(i,j)}, \quad \ldots \quad , o_{|\mathcal{O}|}^{(i,j)})$ into a single operation $\bar{o}^{(i,j)}$, a continuous mixture of the entire set:

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp \alpha_o^{(i,j)}}{\sum_{o \in \mathcal{O}} \exp \alpha_o^{(i,j)}} o(x). \tag{3.2}$$

By making the weight vector α a differentiable parameter of the supernet, it can be learned via gradient descent; meaning that the architecture of the model is now itself learnable and optimizable. This therefore removes the need for an external controller entirely, with the possibility for combining architecture selection seamlessly with model training. However, Liu, Simonyan, and Yang choose to keep the two processes separate, this decision motivated by structuring the architecture weights and network weights as separate variables in a bilevel optimization problem. They approximate a first-order and second-order solution to this optimization problem that differ slightly in terms of their formulation, accuracy, and requisite computational complexity, but both consist of alternating between updating architecture weights and network weights, keeping one fixed whilst learning the other.

After the architecture weights and network weights have converged sufficiently, a discrete architecture is extracted by selecting the operations with the highest nonzero weights, such that only one operation per parallel operation set is selected. This discrete architecture is then retrained from scratch. On CIFAR-10, first-order DARTS discovered a 3.3 million parameter model that scored a test accuracy of $97.00 \pm 0.14\%$, taking 1.5 GPU days to complete the search. Second-order DARTS found a 3.3 million parameter model with an accuracy of $97.24 \pm 0.09\%$, but took four GPU days to run.

3.6.2 NAO

The DARTS method of operation mixing is not the only way to produce a continuous search space. Shortly after the publication of DARTS, Luo et al. published "Neural Architecture Optimization" (NAO) which used an architectural embedding space to produce a continuous search space. NAO involves three chief components, the encoder, the performance predictor, and the decoder. The encoder is a small recurrent neural network whose task is to translate architectural configurations, represented via string, into an n-dimensional embedding space. The performance predictor then maps points in the embedding space into predicted test set accuracy, taking the form of a single layer feedforward network. Finally, the decoder is another small recurrent neural network which maps embedding vectors back to architectural representations. These three components are trained jointly with a combined loss function. Once the three components are fully converged, better architectures can be derived by performing gradient descent over the embedding space. This entails sampling an architecture, mapping it into the embedding space via the encoder, then taking a step in the most negative gradient direction as determined by the performance predictor. The location of this step is then passed to the decoder, in theory producing a slightly better architecture than the first. This process is repeated until the gradient descent converges to a minima, ideally the optimal architecture within the space.

The main downside to the algorithm as presented thus far is that in order to meaningfully learn a mapping between the embedding space and performance, the performance predictor requires a significant sample of fully-evaluated architectures. For CIFAR-10, 1000 full architecture evaluations were necessary to achieve convergence, a process that took 200 GPU days to complete. To mitigate this immense computational cost, Luo et al. adapt NAO to the weight sharing system used in ENAS, where the encoder, performance predictor, and decoder operate over subgraphs of a supernet. This reduces the necessary architectural evaluations from 1,000 down to just one, that of the supernet, and reduces the runtime to just seven GPU hours.

For CIFAR-10, the best architecture found by NAO sans weight sharing had 144.6 million parameters, scored a 98.07% test accuracy, and took 200 GPU days to discover. The best architecture found by NAO with weight sharing had 2.5 million parameters, scored a 97.07% test accuracy, and took just seven GPU hours to discover.

3.6.3 ProxylessNAS

One potential limitation to the cellular approach used by algorithms like DARTS and ENAS is the assumption that the best normal and reduction cells will stack to become the best N celled model. However, this limitation is often necessary as the full supernet for an N cell model would have a intractably large VRAM footprint, whereas a single cell is much easier to fit onto the GPU. In the 2018 paper "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware", Cai et al. propose a method to avoid this restriction altogether. Cai et al.'s algorithm is structured very similarly to DARTS, where a supernet is constructed with each possible operation placed in parallel along the edges. These parallel operations are weighted via α_o much like DARTS, with the crucial difference lying in the parallel edge computation equation. Rather than using a softmax summation of each operation as per DARTS (Equation 3.2), each parallel edge is computed as follows:

$$\bar{o}^{(i,j)}(x) = \begin{cases} o_0(x), & \text{with probability } a_{o_0}^{i,j} \\ \dots & \\ o_n(x), & \text{with probability } a_{o_n}^{i,j} \end{cases} \tag{3.3}$$

The selection of operations is one-hot, in that a single operation is chosen at a time. The edge's operation weights $a_o^[i,j]$ therefore determine how frequently each operation is chosen to be the sole operation of the edge. This is the crucial difference to DARTS, and drastically reduces the VRAM requirements of the ProxylessNAS supernet. This is because only $\frac{1}{|\mathcal{O}|}$th of the candidate operations need to exist on the VRAM at any given time, signficantly reducing the amount of tensor allocations required by the model. This allows the ProxylessNAS supernet to exist in its complete form directly from initialization, meaning that the entire model can be directly searched for, rather than single cells as per DARTS.

The difficulty with this approach is that in order to effectively learn the edge weights, the gradient of each weight must be computed, which in turn is only possible by performing a forward pass through each operation. This would invalidate the VRAM benefits of the edge one-hotting, and therefore Cai et al. devise a means around this problem. Rather than learn the entirety of the edge weights at once, they instead formulate it as a tournament selection problem. Their reasoning for this is that if a particular operation is the best among all the candidates, it should also be the best in the pairwise comparisons performed in tournament selection. In practice, training of the model happens in two stages. First, the edge weights are held fixed, and the operations are chosen stochastically as per Equation 3.3 and trained normally. Next, the operation parameters are held fixed, and two operations per edge are selected at random according to the edge weights. The two corresponding edge weights are updated while holding the other $|\mathcal{O}|-2$ fixed, and as such only $\frac{2}{|\mathcal{O}|}$ths of the operations need to be held in VRAM, which largely mitigates the cost of training edge weights.

One potential difficulty with this approach is ensuring that all candidate operations in each edge are trained adequately, such that they can be compared fairly against the other candidates. Since the edge selection is binary, $|\mathcal{O}| - 1$ operations per edge are not trained during each operation training step. If an operation happens to be chosen less frequently than its competing edge-mates, it will receive less training. Even if this operation would end up being the best of the candidates after convergence, the fewer training cycles will likely cause it to perform worse than the competition. This in turn would cause it to be chosen even less frequently, furthering this cycle. However, this can likely be mitigated by careful choice of the rate of edge weight training, such as not to lock in any decisions prematurely.

After training the edge weights and operation parameters, the final architecture is chosen by selecting the operations with the highest weights from each edge. This subarchitecture is then trained from scratch. For CIFAR-10, ProxylessNAS discovers a network that achieves 97.92% accuracy with 5.7 million parameters. The time cost for the search is not reported in the paper or in any subsequent papers, but from the details provided in the paper it can be inferred to be somewhere on the order of a few GPU-days.

3.6.4 PC-DARTS

While DARTS was shown to be effective at finding state-of-the-art architectures, the supernet search method does require a significant amount of VRAM space to evaluate candidacy of operations that are eventually excluded from the model. This means that the models are size limited by how big the candidate supernet cell can be. One approach to remedy this issue is Cai et al.'s edge binarization, but another approach is introduced by Xu et al.'s 2020 paper "Partially Connected DARTS" or PC-DARTS. Here, for each parallel edge of candidate operation, only a proportion $\frac{1}{K}$ of the input channels to the edge are selected per forwards pass. This subset passes through the parallel operations, while the remaining channels pass through transparently. This is highly similar to dropout, with the difference being dropout omits the operation while PC-DARTs replaces it with the identity. This reduces the computation necessary for each candidate operation significantly as well as reduces the amount of space each operation needs to allocate by K, allowing the use of larger batch sizes or larger models.

The authors note that this channel sampling acts to reduce operational selection bias. They observe that DARTS often shows bias towards weight-free operations like poolings, as in early stages of the model's lifetime those operations do not have randomly initialized weights and are thus more reliable. Channel sampling can help this as it reduces the mathematical difference between any two operations; $\frac{1}{K}$ of an edge's output channels are operation outputs, while the other $\frac{K-1}{K}$ are simply identities. Therefore, the choice between one operation versus another affects only these $\frac{1}{K}$ channels, reducing the possible difference between any two operations and thus reducing selection bias. The downside to this channel sampling approach is that it implicitly lowers the accuracy of the architectural decisions, as operations are selected based on their performance over small subsets of the input data. This is unstable, as the particular subsets that this evaluation is made over are continuously fluctuating. To mitigate this, they introduce

edge normalization, which introduces a set of edge weights β, such that the computation of a single edge $i \to j$ is:

$$f_{i \to j}(x) = \frac{\exp \beta_{(i,j)}}{\sum_{i<j} \exp \beta_{(i,j)}} \sum_{o \in \mathcal{O}} \frac{\exp \alpha_o^{(i,j)}}{\sum_{o \in \mathcal{O}} \exp \alpha_o^{(i,j)}} o(x). \tag{3.4}$$

This adds a second weighting to the configuration; α selects which operations lie along each edge just as in DARTS, while β describes the weight of each edge in the cell. The final architectural decisions are then chosen as the product of an operation's α weight with its edge's β weight. Therefore, operations that perform abnormally well for a particular channel subset might get a beneficial bump in their α weight, but it will be normalized via their overall edge weighting which is invariant to channel sampling.

To compare the additions of partial connectivity and edge normalization, the authors perform an ablation study against regular DARTS over CIFAR-10. In their experiments, regular DARTS scored 97% test accuracy. The addition of just edge normalization increases this to 97.18%, while just partial connectivity increases it to 97.33%. Both together produce the best result, a 97.43%. However, as the paper's reviewers note, the amount to which these increases are due to actual search improvements or simply the increased model and batch size afforded by the channel sampling approach is unclear.

3.7 Conclusions

	Test Acc.	Params (M)	Search Hours
NAS-RL	95.53%	7.1	537,600
Neuroevolution-NAS	94.10%	5.4	61,320
SMASHv1	94.47%	4.6	36
SMASHv2	95.97%	16	36
PNAS	96.59%	3.2	2,500
NAS-Net-A	97.60%	27.6	48,000
NAS-Net-B	96.27%	2.6	48,000
ENAS Micro-search	96.131%	38.0	7.7
ENAS Macro-search	97.11%	4.6	14.4
Regularized Evolution	$96.66 \pm 0.06\%$	3.2	75,600
DARTS First-order	$97.00 \pm 0.14\%$	3.3	36
DARTS Second-order	$97.24 \pm 0.09\%$	3.3	96
NAO (w/o weight sharing)	**98.07%**	144.6	4,800
NAO (w/ weight sharing)	97.07%	**2.5**	7
ProxylessNAS	97.92%	5.7	N/A
PC-DARTS	97.43%	3.6	**2.4**

Table 3.2 CIFAR-10 statistics for all of the NAS models covered thus far.

	Top-1	Top-5	Params (M)	Search Hours
NAS-RL	—	—	—	—
Neuroevolution-NAS	-	—	—	—
SMASHv1	—	—	—	—
SMASHv2	61.38%	83.67%	16.2	**36**
PNAS	74.2%	91.9%	**5.1**	2,500
NAS-Net-A	74.0%	91.6%	5.3	48,000
NAS-Net-B	72.8%	91.3%	5.3	48,000
ENAS Micro-search	—	—	—	—
ENAS Macro-search	—	—	—	—
Regularized Evolution	**82.8%**	**96.1%**	86.7	75,600
DARTS First-order	—	—	—	—
DARTS Second-order	73.3%	91.3%	4.7	96
NAO (w/o weight sharing)	74.3%	91.8%	11.35	4,800
NAO (w/ weight sharing)	—	—	—	—
ProxylessNAS	74.6%	92.2%	N/A	200
PC-DARTS	75.8%	92.7%	5.3	91.2

Table 3.3 ImageNet statistics (if provided in original papers) for all of the NAS models covered thus far.

Of the differentiable architecture search algorithms outlined here, the approach outlined by DARTS is particularly appealing, for a number of reasons. First is the elegance of integrating architecture search into the gradient descent function, piggybacking alongside an already existing facet of the training cycle. The analogy to draw here is that DARTS is to architecture design what the neural net was to feature engineering; it took something that was a difficult manual process and seamlessly incorporates it into the blackbox. The second brilliance of DARTS is that it paved the way for one-shot algorithms. These are a class of NAS algorithms that meet two criteria: they entirely combine architecture search with training, and they only train a single model. While DARTS was not explicitly a one-shot algorithm, in that it only meets the latter criterion, the majority of the one-shot search concept is already sketched out by the paper. If DARTS was adapted to work over an entire model, not just single cells, it would be capable of discovering an architecture while fully training said architecture at the same time. This is the Platonic ideal of neural architecture search and the fullest realization of the architecture-search/feature-engineering analogy: there is no need for any manual labor, external controllers, or particularly complex algorithms, the network does it all for you.

3.8 NAS Efficacy versus Random Search

The results promised by these various algorithms are tempered by claims of both insufficient scientific rigor as well as comparison to random search. The first paper to raise these claims is "Random Search and Reproducibility for Random Search" by Li and Talwalkar, which identifies

three major issues with the field of Neural Architecture Search at the time of publication: inadequate baselines, complex methods, and lack of reproducibility.

The term "inadequate baselines" is used to refer to the lack of any meaningful 'control' in NAS experiments; when evaluating a search strategy, papers rarely included how well randomly selected models[6] from the search space perform. This is naturally crucial to distinguish an effective NAS algorithm from merely an over-constrained search space; a sample of randomly selected models establishes the baseline performance of the search space. A NAS algorithm that operates over a search space that exclusively contains good models is guaranteed to recover exclusively good results, which does not tell us anything about the algorithm's ability to identify promising architectures. Meanwhile, a NAS algorithm that recovers poor results in comparison to the state-of-the-art might be flawlessly identifying top-performing architectures but is hampered by a poor search space.

Li and Talwalkar's next critique of the field is its complex methods, saying that "it is unclear what NAS component (s) are necessary to achieve a competitive empirical result."(Li and Talwalkar, 2019, p. 2) NAS algorithms tend to be fairly complex, with a lot of interconnected components and methods. The final performance of a NAS algorithm could be due to a number of factors, like training routine, data augmentation policy, search-space constraints, or model hyperparameters, with the actual quality of the architecture playing a very minor role. This can be rectified by including a random control group in experiments that shares everything bar architecture search strategy with the NAS algorithm, as described above, or by performing a full ablation study on each novel component of the NAS algorithm.

Finally, the lack of reproducibility of published NAS algorithms is flagged. This often manifested as missing source code, random seed, or documentation of hyperparameter choice. Without adequate reproducibility it is impossible to ascertain whether a particular published result is due to the algorithm in question or simply random chance.

This paper was shortly followed by "Evaluating The Search Phase of Neural Architecture Search" (Yu et al., 2019), which further investigates the issues raised by Li and Talwalkar, specifically that of inadequate baselines. They point out that without comparison to random search or sufficient testing of the robustness of results, the true value of search algorithms cannot be established. This is due to the intrinsic entanglement of search space and search algorithm as described above; is a search algorithm good because it succeeds in identifying good architectures or does it simply operate within an exclusively good search space? Even if the latter is false, it does not necessarily mean the former is true: the reported result may merely be the best result cherry-picked from many trials, and is thus not truly representative of the average performance of the algorithm.

Yu et al. then investigate these concerns by evaluating ENAS, DARTS, NAO, and BayesNAS across a variety of baselines, first comparing the mean performance of each algorithm over many

[6]Randomly selected model refers to a stochastically generated model within the search space rather than algorithmically generated. The exact methodology for this differs depending on the search space and use-case, but typically random-models are constructed by replacing any algorithmic model design decisions with random choice. For example, DARTS' main design decision is the selection of edge operations via learned α parameters, and thus a random model in the DARTS space would replace the learned α values with random ones.

random seeds against the mean performance of random search. The comparison to random search addresses the space-algorithm entanglement issue, in that the mean random search performance identifies the average quality of the search space. Then the difference between the mean random search performance and the mean NAS performance is the NAS algorithm's efficacy at discerning high quality architectures within the search space, therefore decoupling the search space from the algorithm. Furthermore, comparing the average performance of a particular NAS algorithm to the published performance can reveal whether the published results are reflective of true performance or are instead a cherry-picked outlier.

To perform this study Yu et al. design an equivalent random search space to the NAS algorithms, and train both the randomly selected architectures equivalently to the searched architectures on CIFAR-10 and the Penn Treebank. This process was repeated 10 times, and the mean performance was reported as well as upper and lower bounds, derived from the best and worst runs for that particular search policy. For the Penn Treebank, the random search policy recovered the best overall architecture, and outperformed both DARTS and NAO on average. For CIFAR-10, DARTS, ENAS, and NAO all outperformed random search on average, but only by 0.38 percentage points at best. All 10 BayesNAS architectures were worse than the worst randomly selected architecture, and only NAO had all 10 architectures be better than the best randomly selected architecture. What these results demonstrate is that all four of these algorithms have benefited from a very favorable search space; arbitrary models from the space perform equivalently to the architectures selected by the algorithms.

Yu et al. then look to establish a more concrete method of establishing the quality of a search algorithm, which is to perform the search over a space where the quality of each architecture is fully enumerated. For the Penn Treebank, they limit the search space to 32 architectures, whereas for CIFAR-10 they use the roughly 462,000 given in the NASBench-101 dataset. This means that the architecture found by each algorithm can be directly compared to the optimal architecture within the search space. In the case of the severely limited Penn Treebank, not a single algorithm managed to identify the best architecture within the pool of 32. Of the four, only NAO produced better than random results on average. In the reduced CIFAR-10 experiment, only NAO produced better than random architectures, with the best architecture it identified being the 19,552nd best in the space. This ranks in the top 4%, and was better than 62% of randomly selected architecture. All other algorithms recovered much worse results, with the best DARTS model better than 24% of random architectures, and ENAS better than a meager 7%.

The results of this paper were shocking; only one of the extant state-of-the-art NAS techniques at the time of publishing was actually any better than random search, and only barely. However, there are two possible interpretations for this. The cynical one is that the NAS algorithms' poor performance is due to the fact that they are simply expensive means of arbitrarily selecting neural architectures, providing little-to-no advantage over random search. The more optimistic one, and the one that is necessary to subscribe to if one wants to maintain any motivation to continue working in the field of NAS, is that the reason the NAS algorithms performed poorly compared to random search is because the search space was artificially over-constrained; designers of NAS

algorithms incorporated into their search space design the best practices established by the near decade of research into these particular datasets. This means their implicit biases of architecture design are hard-coded into the search space, that any arbitrary architecture within the space is backed by years of theoretical foundation. This is to say, what if these results are not because Neural Architecture Search is bad, but because random search has an unfair advantage?

One such paper that explores this question is "Exploring Randomly Wired Neural Networks For Image Recognition" by Xie et al. (2020). Here, the authors note that many breakthroughs in computer vision came from novel wiring patterns, such as the residual connections of ResNet or the parallel edges of Inception. To this extent, they have concern that the actual wiring patterns that NAS algorithms can generate are still highly constrained by manual design philosophies. For example, in DARTS every node is connected to every subsequent node and the output of each node gets passed to the final cell output. This means that every DARTS cell is essentially very similar to the Inception wiring pattern, and DARTS' only real mechanism for innovation in the architecture is in operation selection, not wiring patterns. To explore this idea, Xie et al. uses random graph generators to produce completely novel wiring schemes. Examples of these novel wiring schemes can be seen in Figure 3.7. Notably, the very organic-looking wiring patterns are unlike any other seen in literature, exploring completely novel territory in terms of network design. These random graphs are then converted into network architectures, with edges representing data flow and nodes performing either aggregation, transformation, or simply identities. Aggregation is exclusively summation, as concatenation produces variable sized outputs corresponding to the number of edges entering a node. Since this value is variable given the exact random generator, summation's fixed output size makes formulation of complex models significantly simpler. Transformation in this case refers exclusively to the common ReLU-convolution-batch normalization stack, with a 3x3 separable convolution as the default convolution.

These random networks are then evaluated over ImageNet, and the authors find that networks generated via the Watts-Strogatz generator produce results that rival the state-of-the-art of both hand-designed and NAS-designed architectures. Notably, in the sub 600M FLOP small computation regime, the Watts-Strogatz random models outperform DARTS by 1.6 percentage points in Top-1 accuracy. The authors also look beyond raw performance, and explore the effects of different graph generators, graph damage (defined shortly), and convolution type. For graph generators, they notice that of the three generators used (Erdős-Rényi, Barabási-Albert, and Watts-Strogatz), all of them consistently produced models that trained to decent accuracy. More interestingly, the variance of model performance within the set of models generated by a specific generator was surprisingly low, around 0.2-0.4%. They note that this is only slightly larger than the variance found when retraining identical ResNet configurations, which is between 0.1 and 0.2%. The variance between different generators was more stratified, with the best performing (Watts-Strogatz) a clear 3% better than the worst, the Barabási-Albert. This implies that the broader design principles that differentiate the generators is more significant than the granular variations between same-generator architectures. This is to say, general design parameters like

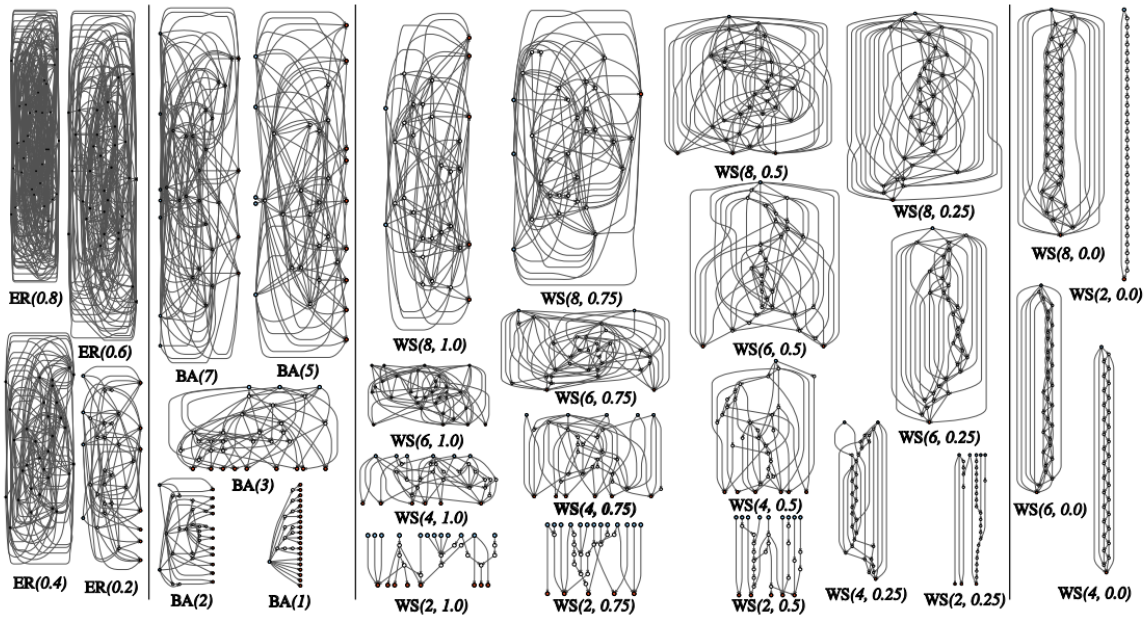


Fig. 3.7 Sample random cells generated by the three random graph generation algorithms (Erdős-Rényi, Barabási-Albert, and Watts-Strogatz).

the typical degree of nodes or average path lengths through cells is more important to final performance than the minutiae of the wiring itself.

Next they explore graph damage, the process of removing a single node or edge from a fully trained model and observing the effects of its removal on test accuracy. Here, they compare the difference in test accuracy against the degree of the removed node in the case of node removal, or the degree of the edge's target node in the case of edge removal. Models generated by different graph generators had different sensitivities to this damage, with Watts-Strogatz being the most consistently negatively affected. Specifically, Watts-Strogatz networks had a high sensitivity to the removal of high-degree nodes and edges to low-degree nodes. The latter makes sense, as each edge to a low-degree node is more precious than to a high-degree one and causes a greater comparative difference to that node's input. The former indicates that these high-degree nodes act as valuable distribution hubs within this class of model. Observing the general connectivity pattern of the Watts-Strogatz networks in Figure 3.7 clarifies this behavior, as the high degree nodes tend to be either the output node or are towards the end of the most significant path to the output, while other generation algorithms tend to not have significant reliance on any single particular path. Since the Watts-Strogatz networks are most similar to existing network design philosophies, this is useful to gain intuition as to the most influential nodes in manual or NAS designed networks.

Finally, the authors look at changing the exact convolution schema used by the transformation nodes of the network, and evaluate four different configurations while holding the architectures of the randomly generated networks fixed. In each of the four configurations, the general ReLU-convolution-batch normalization stack is preserved, while the convolution is chosen as either a 3x3 separable convolution, a 3x3 non-separable convolution, a 3x3 max pool followed by 1x1 convolution, or a 3x3 average pool followed by 1x1 convolution. Interestingly, over 30

different random architectures the general performance rank of the four convolution choices within the same architecture is more or less consistent, with the separable convolution performing markedly best by around 5 percentage points over 2nd place, followed by 3x3 convolutions or 3x3 max-pools, and 3x3 average pools performing worst. Furthermore, the rank of each architecture remains relatively consistent when convolution choice is held fixed, with the best architectures for a specific convolution remaining among the best for different convolutions. This indicates that operation selection and architecture choice have orthogonal effects on performance, that the best architectures will be the best architectures regardless of operation choice, while choosing the best operations can ensure that that architecture achieves its best performance possible. This is potentially counter-intuitive; it might be expected that different operations would prefer specific design patterns and vice versa, however this is not the case for these random networks. This result, if it is applicable to other network designs as well, is very useful in contextualizing comparative results between NAS algorithms. For example, NAO uses a different operational palate than DARTS or PC-DARTS, selecting different subsets of operations or adding in new ones depending on search constraints. Is the different performance produced by NAO to the other two algorithms therefore a result of better architecture selection, or is due to the orthogonal performance direction of the operational palette?

Finally, the last concept that stood out from this paper is implicitly rather than explicitly introduced, and this was removing the concept of normal and reduction cells entirely. Rather than design networks around a reduction cell followed by a number of repeating normal cells, these networks are simply stacks of unique reduction cells. By increasing the size of each cell, a series of cells becomes simply a possible subgraph of the larger cell, with the exact size increase necessary to result in this behavior varying depending on the exact design philosophy of the cells. For something like a DARTS cell with two input nodes connecting to each of the n intermediate nodes and each intermediate node connecting to each subsequent intermediate node, an n-node cell has $\binom{n+2}{2} - 1$ edges. The "-1" term comes from the fact that the triangular number assumes each node connects to each subsequent node. However, input nodes only connect to intermediate nodes, which means input node 0 does not connect to input node 1. There is therefore one fewer connection than given by the triangular number, and thus 1 is subtracted. With this edge count, any sequence of C, n-node cells can be replaced by a single larger m-node cell, such that:

$$\binom{m+2}{2} - 1 \geq C\left(\binom{n+2}{2} - 1\right). \tag{3.5}$$

In the case of n=7 and C=3 as in CIFAR-10, a single 12 node cell has sufficient edges to be a supergraph of the original three cell stack. Furthermore, each of these larger cells is unique in architecture compared to each other, a design feature not seen in any other cell based model thus far.

3.9 Next Steps

Intrigued by these ideas of expanding the search spaces of NAS algorithms, I wanted to apply the concepts introduced by these randomly wired networks to NAS. Specifically, it was of interest to explore how NAS might operate over these highly unconstrained search spaces, where each cell can be unique and the internal connectivity much more varied than any seen previously. Doing this entailed designing a NAS algorithm that operates over an exponentially larger and more unconstrained search space than seen in other NAS papers, and thus BonsaiNet was born.

Chapter 4

BonsaiNet

4.1 Introduction

In order to evaluate the validity of the second interpretation of the random search problem, that is, that NAS algorithms are hampered by an overconstrained search space, I designed BonsaiNet, a Neural Architecture Search algorithm that operates over a search space exponentially larger than those of DARTS, ENAS, or NAO.

This chapter will first look at the design of the Bonsai search space, and discuss the design challenges that operating in such a space presents. From there, components are designed to address these challenges, and explore how these components operate and how they influence model training dynamics. Next, these components are used to create the BonsaiNet algorithm (Algorithm 4), and explore the mechanics, mathematics, and optimizations necessary for its operation. After this, BonsaiNet is tested across a variety of experiments and configurations, and its performance is evaluated against the cutting-edge. Finally, the design trends that occur within BonsaiNet-designed networks are investigated, to see what can be learned by studying its design decisions.

4.2 Search Space

The Bonsai search space is an expansion of the search space of DARTS (Liu et al., 2018) and PC-DARTS (Xu et al., 2020), whose models are composed of stacked cells, each cell a directed graph wherein edges are tensor operations and nodes are tensor aggregations. Cells are classified as either a *normal* cell, meaning that tensor dimensionality remains unchanged throughout, or as a *reduction* cell, meaning that spatial dimensions are halved while the channel dimension is doubled. The spatial changes in reduction cells are handled by stride-2 operations along edges that receive the cell input, and the channel dimension is doubled via a 1x1 convolution prior to cell input. Within cells, every node n_i receives input from each previous node (n_{i-1}, n_{i-2}, …, n_0) in the cell. Like the DARTS space, the seven available tensor operations to convolutional models are identities, 3x3 max pooling, 3x3 average pooling, 3x3 separable convolutions, 5x5 separable convolutions, 3x3 dilated convolutions, and 5x5 dilated convolutions. While these

search spaces contain many good models, as shown by the results achieved by DARTS and PC-DARTS, there are three potential over-constrictions in the search space that were identified: *single-path edges*, *fixed cellular input*, and *cellular homogeneity*. Single-path edges refer to the constraint that within the cells, each edge E is restricted to performing a single operation o:

$$E^{out} = o(E^{in}), \tag{4.1}$$

where E^{out} is the edge output and E^{in} the edge input. This results in a total number of selectable edges equal to the size of the operation space $|\mathcal{O}|$, which is seven in algorithms like DARTS or PC-DARTS. For a cell with N nodes, with edges connecting every pair of nodes (n_i, n_j) such that $j > i$, the number of total non-input edges is given by $\binom{N}{2}$. Since there are two additional input nodes that connect to each of the N intermediate nodes but not each other, the total number of *intra*-cellular connectivities is $|\mathcal{O}|^{\binom{N}{2}+2N}$.

Next, fixed cellular input refers to how each cell C_c is restricted to receiving input from the two previous cells C_{c-1} and C_{c-2}:

$$C_c^{in_1} = C_{c-1}^{out} \tag{4.2}$$

$$C_c^{in_2} = C_{c-2}^{out} \tag{4.3}$$

Under this restraint, there is only a single *inter*-cellular connectivity pattern; all models in the space share this pattern. Finally, cellular homogeneity describes how the internal connectivities and operation selections of each category of cell are strictly identical; each normal cell is identical to every other normal cell, and the same principle holds for reduction cells. This means that any model with c cells has only two distinct cells, each duplicated a certain number of times.

To relax this search space, each of the three aforementioned design restrictions are relaxed. First, each edge is allowed to be multi-path, the sum of any arbitrary combination of operations in the operation space:

$$E^{out} = \sum_{i=0}^{|\mathcal{O}|} \alpha_o^i o_i(E^{in}) \tag{4.4}$$

where α_o is a binary vector of size $|\mathcal{O}|$. There are therefore $2^{|\mathcal{O}|}$ possible edges that may lie between any two nodes in the cell, meaning there are $2^{|\mathcal{O}|\left(\binom{N}{2}+2N\right)}$ possible n-node cells. For this to be possible, the node aggregation operation must be summation in all nodes for the same reasons that the randomly wired networks described in Section 3.8 use summation; the number of operations that enter a node is variable and the fixed output size of a summation operation regardless of input edge count makes the formulation of models much simpler. This is a fundamental difference to DARTS, where the output node of a cell performs concatenation. See Figures 4.1 and 4.2 for a detailed architecture diagram of Bonsai edges and nodes respectively.

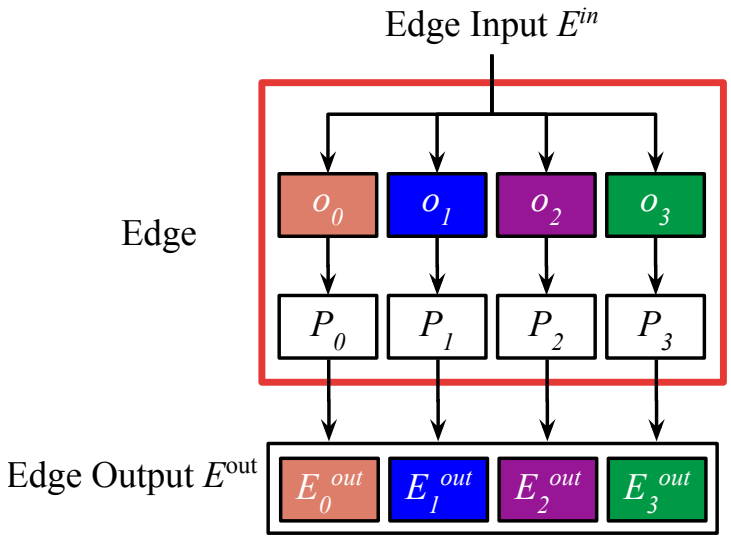


Fig. 4.1 Edges take some single input and pass it through $|\mathcal{O}|$ parallel operations. The results of these operations are then individually pruned, and the edge outputs a list of these pruned operation outputs. These special edges that perform the parallel pruning are represented in red, while black edges simply transfer data unmodified between network nodes.

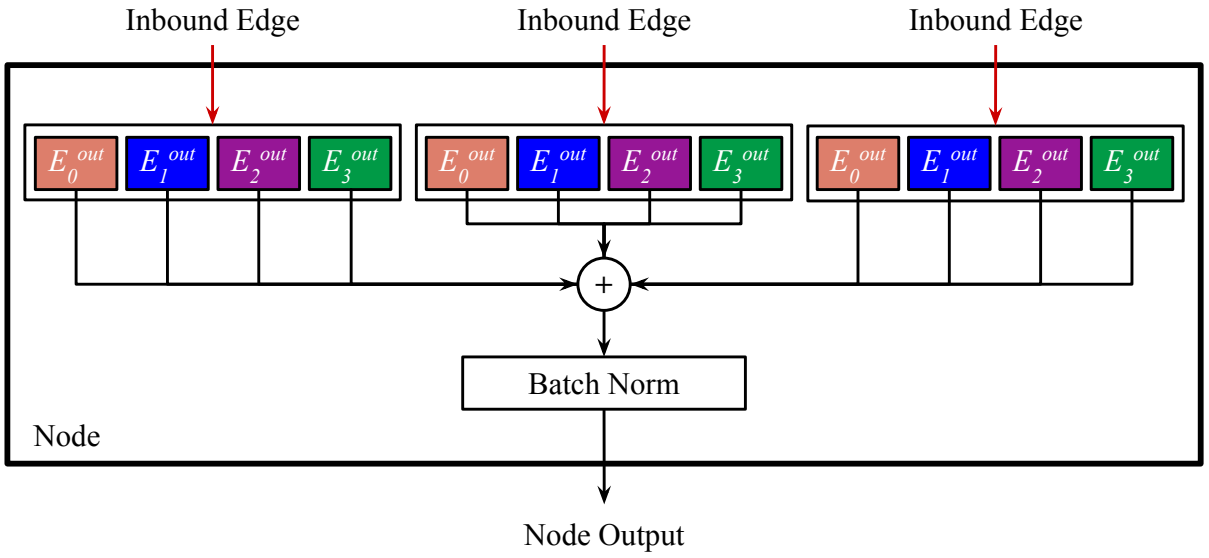


Fig. 4.2 Nodes concatenate the output lists of each inbound node via summation and batch normalization, and then pass this concatenation on as the node output. This particular Bonsai node has three inbound Bonsai edges and $|\mathcal{O}| = 4$.

Next, the total number of cellular inputs is expanded such that each cell C^c receives two inputs, the first from the previous cell and the second from any combination of previous cells or the original model input x:

$$C_c^{in_1} = C_{c-1}^{out} \tag{4.5}$$

$$C_c^{in_2} = \alpha_c^0 x + \sum_{i=1}^{c-1} \alpha_c^i C_i^{out} \tag{4.6}$$

where α_c is a binary vector of size n. These transformations happen in the *cell input handler*, which is shown in Figure 4.3.

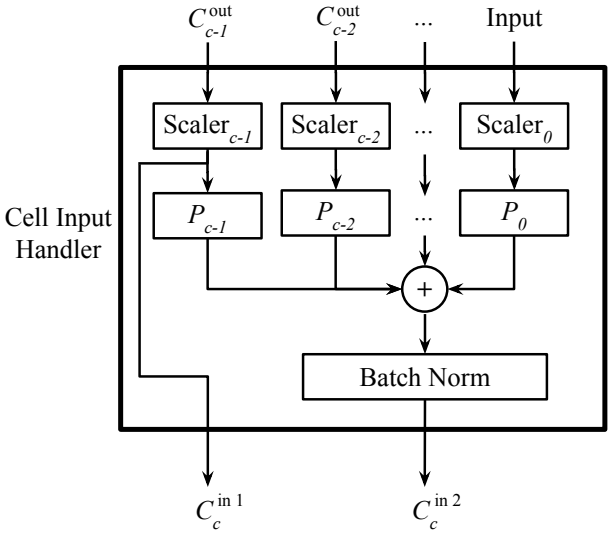


Fig. 4.3 The cell input handler. Its job is to scale the outputs of antecedent cells to be compatible with the operations in the cell and produce two outputs. The first output is just the scaled output of the directly previous cell C_{c-1}. The second output is the pruned sum of all antecedent cells $[C_{c-1}, \ldots, C_0]$ as well as the original model input.

In this configuration, the cth cell of a model has 2^c possible connectivity patterns, meaning that the total number of cellular connectivities for a model with C cells is $\prod_{c=0}^{C} 2^c$. Finally, cellular heterogeneity is allowed, which lets each cell have a distinct connectivity and operation set. See Figure 4.4. for an architectural diagram of a Bonsai cell, and Appendix A for a review of all architectural diagrams shown thus far.

The difference in search space size between the constricted DARTS space and the Bonsai space is summarized in Table 4.1. In the specific DARTS CIFAR-10 search space described

Space	#Inter-cell Cnx	#Intra-cell Cnx	#Unique Cells
DARTS	1	$\lvert\mathcal{O}\rvert^{\binom{N}{2}+2N}$	2
Bonsai	$\prod_{c=0}^{C} 2^c$	$2^{\lvert\mathcal{O}\rvert^{\binom{N}{2}+2N}}$	C
DARTS (C=10, N=4, $\lvert\mathcal{O}\rvert$=7)	1	$6.8x10^{11}$	2
Bonsai (C=10, N=4, $\lvert\mathcal{O}\rvert$=7)	$3.6x10^{16}$	$1.2x10^{21}$	10

Table 4.1 The total number of choices for each of the three design parameters in the DARTS and Bonsai search spaces, for a generic model with C cells, N nodes per cell, and $\lvert\mathcal{O}\rvert$ operations, as well as the specific model configuration used for CIFAR-10 in the DARTS paper.

in the DARTS paper there are therefore roughly $4.6x10^{23}$ possible models, assuming a fixed pattern of normal and reduction cells. The same configuration in the Bonsai space has $4.3x10^{222}$ possible models. With this dramatic increase in search space size, evaluating neural search algorithms versus random search might be more elucidating, as the design restrictions that may be artificially bolstering random search have been removed.

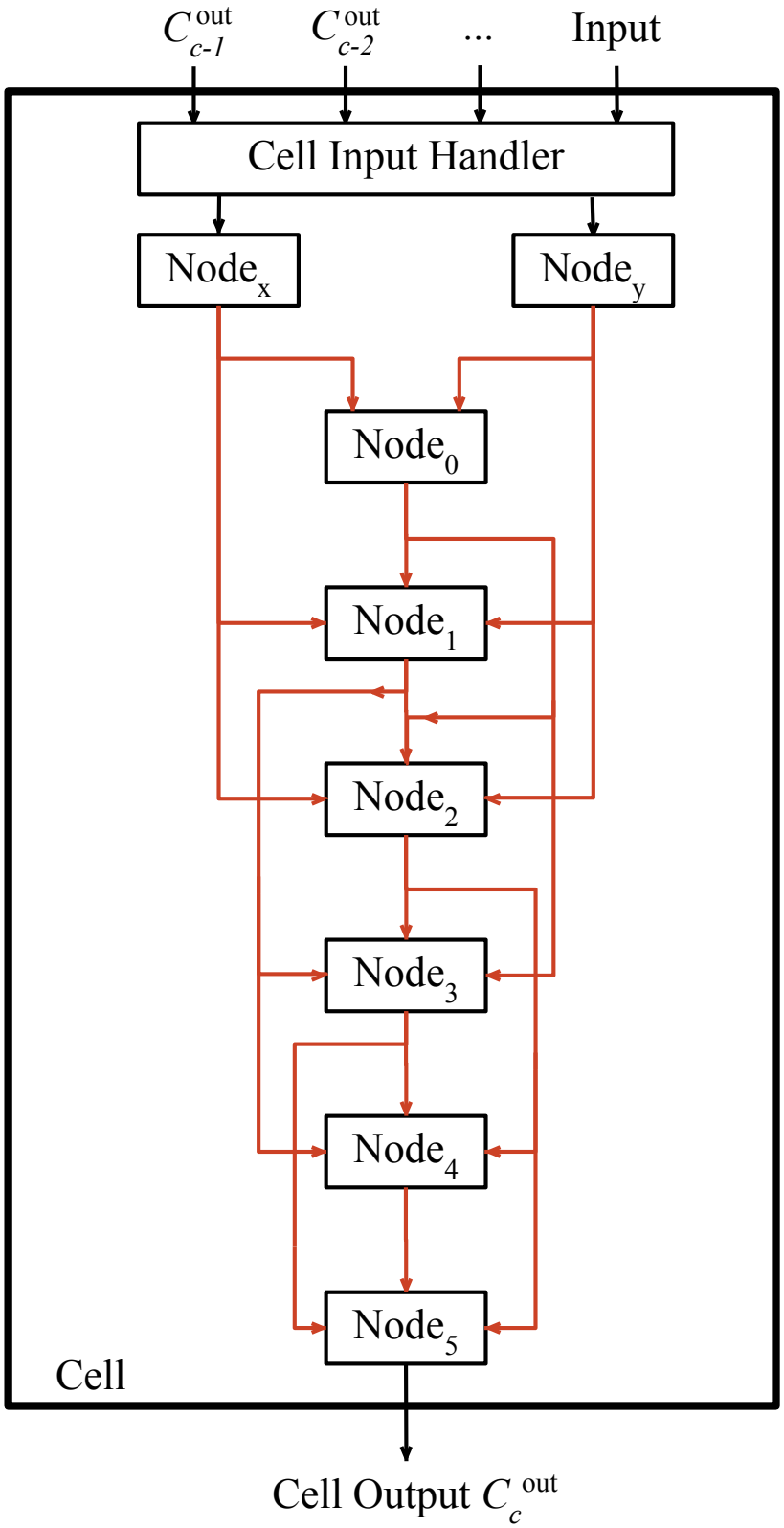


Fig. 4.4 Bonsai cell with six nodes and edge depth $d = 3$. The two outputs of the cell input handler are passed to the input nodes of the cell, called Node x and Node y. From here, each node n is connected via Bonsai edges to each node $[n+1, n+2, \ldots, n+d]$ where d is the edge depth of the model. This simply controls the width/depth ratio of the model; a depth of 1 means every node is linearly connected in serial, while a depth equal to the number of nodes means each node connects to every downstream node.

4.3 Algorithm Design Considerations

Given the various advantages of one-shot, differentiable approaches as described in Section 3.7, it was a natural choice to design one for this search space. However, the predominant difficulty arising from these types of algorithms is that they tend to revolve around whittling down a single architecture from some supernet that needs to fit entirely into GPU memory. The problem with this is that a supernet is more expensive than a single path model by a factor proportional to the size of the operation search space $\mathcal{O}$ because each edge is *superconnected*; containing every operation within $\mathcal{O}$ in parallel. Computing these parallel paths is costly in both floating point operations and memory space, and drastically limits the size of supernets that are feasible on modern consumer hardware. Furthermore, if a supernet was constructed that exactly fits onto a given GPU, any subgraph model extracted from that supernet will be roughly $|\mathcal{O}|$ times smaller and therefore be utilizing only $\approx \frac{1}{|\mathcal{O}|}th$ of the computing power and memory available. This means that the subgraph model will likely compare poorly against a model that takes more advantage of the GPU's power, as is suggested by the 'bigger-is-better' results from papers like ResNet (He et al., 2015).

The cellular homogeneity present in many differentiable NAS algorithms like DARTS is arguably a compromise to account for these exact problems; DARTS does not search for entire architectures via the supernet, but rather for single cells. The single-cell supernet can be made as large as can fit on the GPU, and the space freed after subsequently whittling the supernet down to some subgraph can then be used for subsequent stacking of the found cell. However, this comes at the cost of only searching for a single cell, which then forces the algorithm to rely on the assumption that the best cell will indeed stack into the best architecture, which is an assumption with no apparent justification in literature. Much like Cai et al. (2018), I find it dubious that the best architecture for the first cell in the network would be the same as that for the 10th, and so examining this claim is a secondary motivation for this work.

To design a one-shot differentiable algorithm that allows for cellular heterogeneity, as well as the other two properties of the Bonsai space, two components are necessary. First is a method of *pruning*, some differentiable method to trim the supernet down into optimal subgraphs. Next is a *cell search* algorithm, some way to uniquely search for each cell in the network, in such a way that each cell was aware of its position in the network and its antecedents.

4.4 Pruning

While the DARTS approach of performing a weighted sum of each candidate operation and using the softmax of the weights to determine the final operation selection is a valid pruning approach, it is one implicitly designed around single-path edges. In theory, you could keep the weighted sum of operations and remove the softmax, instead ranking operations by their weight and preserving the n most highly weighted. This however creates a variety of new issues, namely how should each edge pick n? Instead, some way of differentiably and dynamically selecting operations was necessary, such that each edge could choose which and how many operations it needed given its

particular needs. Such a component materialized in the Differentiable Pruner, introduced by Kim et al. in the 2019 paper "Differentiable Pruning Method for Neural Networks"[1].

The differentiable pruner was originally introduced as a means of pruning floating point operations (FLOPs) from neural operations, specifically to reduce the size of fully-connected layers inside of neural networks. The authors describe the problem of attempting to prune these connections via a simple gate function (like the Heaviside step function, for example) as similar to the vanishing gradient problem seen in sigmoid activations in regions far from 0. This problem is illustrated by applying the simple gate to a 1-D example, where the gate G has a single weight w that controls whether the input tensor passes through the gate or not:

$$G(x,w) = \begin{cases} 0 & w < 0 \\ x & w \geq 0 \end{cases} \tag{4.7}$$

When w is less than zero the gate is closed, whereas the gate is open when w is greater than or equal to zero. However, the gradient of this gate function with respect to the weight is zero everywhere, meaning there is no information for gradient descent to use in the process of learning w. This means that regardless of the downstream effects of the tensor passing through the gates the gate will always remain in the exact position it is initialized in. To remedy this, the authors add a saw wave S of some small amplitude $\frac{1}{M}$ to the gate function G, creating the differentiable pruner function P:

$$G(w) = \begin{cases} 0 & w < 0, \\ 1 & w \geq 0 \end{cases} \tag{4.8}$$

$$S(w) = \frac{Mw - \lfloor Mw \rfloor}{M} \tag{4.9}$$

$$P(x,w) = (G(w) + S(w))x \tag{4.10}$$

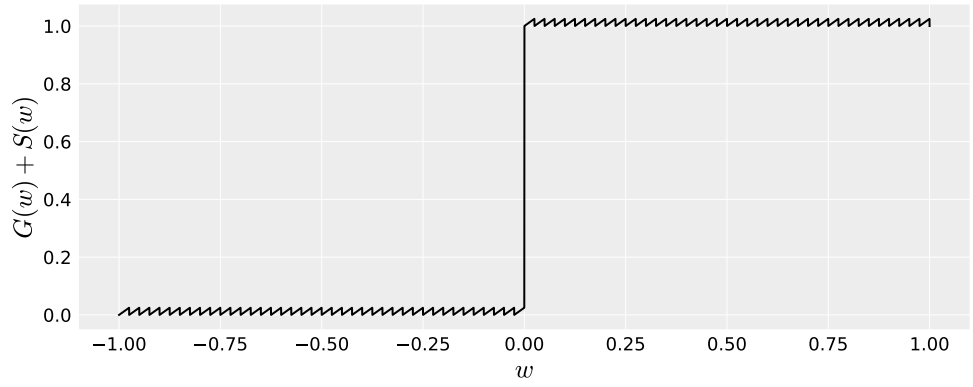


Fig. 4.5 The pruner equation $G(w) + S(w)$, with M chosen to be large enough to be perceptible.

With large enough M, $S(w)$ is effectively 0 everywhere while still having a constant gradient of 1 with respect to w. Adding this saw to the gate creates a function with the multiplicative

[1]Now titled "Plug-in, Trainable Gate for Streamlining Arbitrary Neural Networks"

properties of the gate that maintains the gradient properties of the saw wave, as visualized in Figure 4.5.

The reasoning behind the design of the differentiable pruner can be justified by comparing its gradient descent behavior against that of a simple gat in a short experiment. To start the experiment, the weights of both the simple gate and the differentiable pruner are initialized at 0.1, meaning both functions start in the open position. Then the weights are updated via gradient descent given the following loss function $\mathcal{L}$:

$$\mathcal{L} = (P(x,w) - y_{target})^2 \tag{4.11}$$

Within this specific example, y_{target} is set to 0 for the first 50 gradient steps, meaning that the loss minima will occur when each function is in the closed position and thus outputs 0. For the next 50 gradient steps y_{target} is set to x, which instead means the loss minima will occur when the functions are in the open position and output x. The progression of each function's weight can be seen in Figure 4.6.

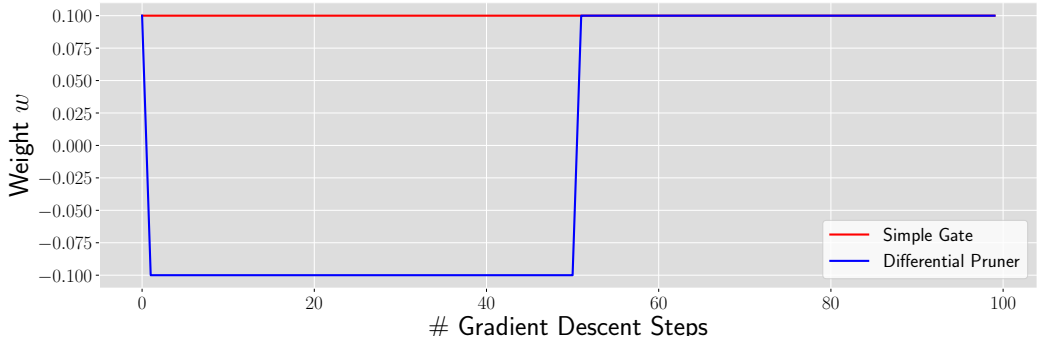


Fig. 4.6 Gradient descent of a simple gate (red) compared to the differentiable pruner (blue). The first 50 steps of gradient descent incentivize a closed gate. The second 50 steps instead reward an open gate.

Notice that the differentiable pruner quickly learns the desired gating, while the simple gate does not learn at all. This demonstrates both claims; first that a simple gate cannot train by backpropagation and second that the differentiable pruner manages to remedy this problem.

To use the differentiable pruner to perform supernet pruning, differentiable pruners are placed along every candidate edge in the supernet, meaning that every value within α_o and α_c from equations 4.4 and 4.6 is computed by a separate pruner. The operation that is performed along the operation edges is as follows:

$$E^{out} = \sum_{o \in \mathcal{O}} P_o(o(E^{in}), w_o) \tag{4.12}$$

Along cellular input edges, the calculation is:

$$C_c^{in_1} = C_{c-1}^{out} \tag{4.13}$$

$$C_c^{in_2} = P_0^c(x, w_0^c) + \sum_{i=1}^{c-1} P_i^c(C_i^{out}, w_i^c) \tag{4.14}$$

4.4.1 Deadheading

As the supernet trains, each of the pruners can operate independently to preserve or prune the edge that they lie along. However, the pruners simply perform a virtual gating of their respective operations; a closed pruner is simply a multiplication by zero, but the operation along the edge is still calculated and takes up space in memory. To actually free up computational resources, the decision of the pruner must be cemented by transforming the purely mathematical deletion into a physical one through *deadheading*. Deadheading entails removing the entire edge that the pruner operates on, such that the candidate operation is permanently removed from the supernet. Henceforth a pruner moving to an 'off' position will be referred to as a *soft* pruning, whereas deadheading and deletion is a *hard* pruning.

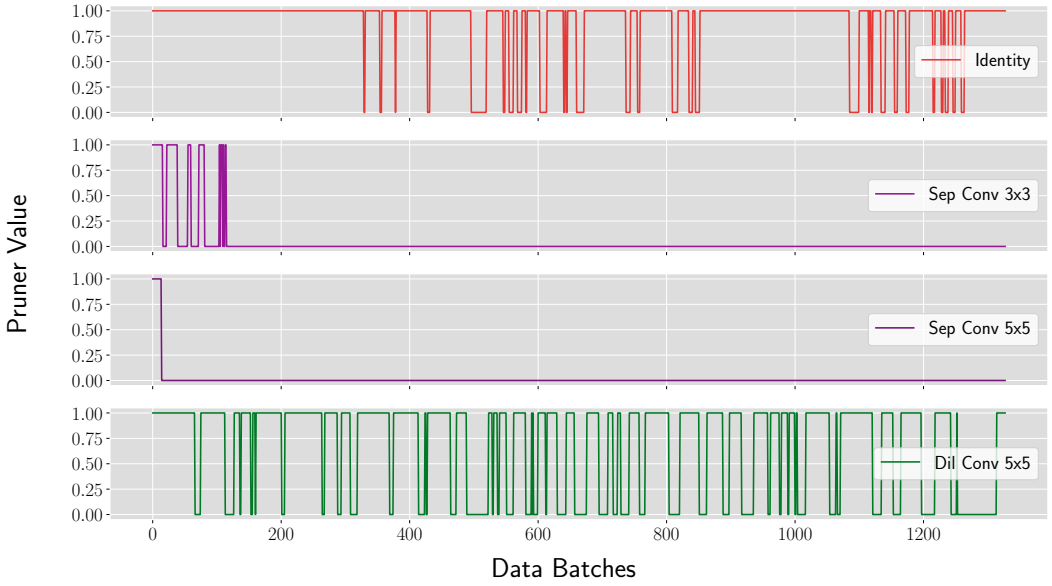


Fig. 4.7 Pruner values over the first 1,300 data batches of four edges within a CIFAR-10 Bonsai model. Notice the toe-dip behavior of the identity and 5x5 dilated convolution edges, where they toggle repeatedly between on and off states as the model experiments with their inclusion. The separable convolution edges however exhibit the permanent-off behavior, where after staying switched off for multiple data epochs they never switch back on.

To perform deadheading effectively, a policy must be designed that rapidly and accurately identifies edges that the model no longer needs. The faster this decision can be reached, the faster the search process can occur. The more accurately this decision can be made, the better the performance of the search algorithm. Designing a deadheading policy is thus balancing these

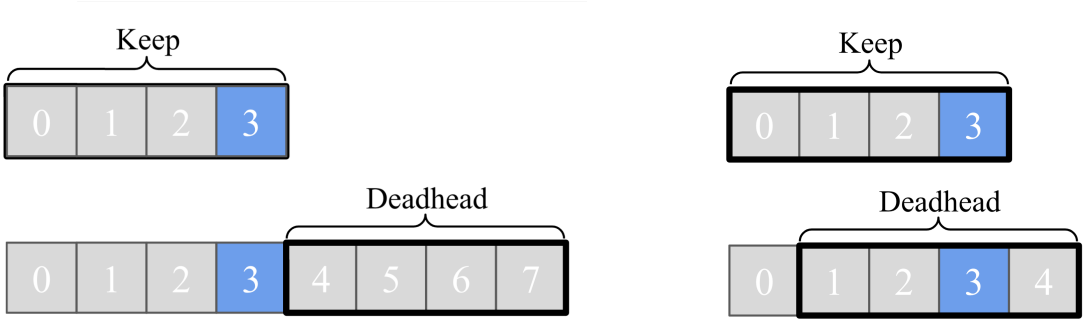
two aims; too eager and risk removing valuable model edges, too conservative and the supernet will never prune.

Crucial to the design of every iteration of deadheading policy was the observation that pruners tend to exhibit one of three common behaviors: permanent-on, toe-dipping, or permanent-off. These latter two behaviors can be observed in Figure 4.7. My hypothesis as to why these three behaviors are so common revolves around a phenomenon I call *operational codependency*. As a model trains, an operation's weights will be learned such as to complement the other operations that it interacts with. As such, each operation will tailor its output such that in collaboration with other nearby operations it performs effective tensor transformations. If the configuration of these operations changes, by something like a pruner switching off one of these neighboring operations or reintroducing one that had been switched off, this codependance breaks. The operation weights are tailored to a particular operational coalition that no longer exists, which is likely to be significantly detrimental to the model's performance if the operation's output is nonzero. Once a model has training in a specific configuration for long enough, operation codependance arises and 'burns-in' the model configuration, which means it is unlikely to find success altering the configuration. This is what causes the permanent-on and permanent-off behaviors of pruners; their local neighborhood of operations has burnt in and thus any change to the pruner state is exclusively detrimental.

On the other hand, when an operation's local neighborhood is in constant flux, with operations switching on and off rapidly, the operations do not form codependence as they simply cannot rely on the nearby operations being consistent. As such, pruners in such a neighborhood can freely 'dip their toes' into either state with little penalty, frequently experimenting and toggling between on and off. The deadheading policy needs to identify permanent-off edges to remove from the model, while preserving permanent-on edges and allowing toe-dipping edges to experiment.

The first few iterations of the deadheading policy were based solely around identifying permanent-off operations, with the first iteration tracking each pruner's state at the end of each training epoch. Every n epochs, the deadheading operation triggers, and deadheads all edges where their pruner was off at the end of the majority of the last n epochs and off at the end the most recent one. There is an implicit inefficiency in this design, which concerns the fixed window of state sampling. Since the state of the pruners is sampled in a n epoch window every n epochs, it is possible to run into cases where a pruner's decision to move to a permanent off state straddles two windows. This would mean waiting for an entire additional n epoch cycle to deadhead the edge. This is remedied by adopting a sliding window policy, wherein the deadhead policy is triggered each epoch and look at the previous n epochs. Within this sliding window, the same deadhead criteria as before is maintained: off in the last epoch and in the majority of the epochs. The sliding window policy was the second iteration of the deadheading policy that was tried for BonsaiNet, and the difference between the two policies' efficacy can be seen in Figure 4.8.

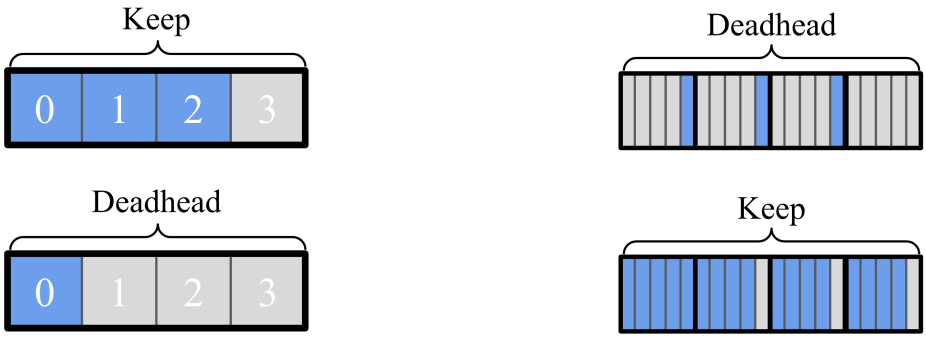
While the second iteration is definitely more time-efficient than the first, both share a fundamental flaw in practice. Since gradient updates happen on a per-batch basis, not per-epoch,



(a) Fixed window policy (b) Sliding window policy

Fig. 4.8 Comparison of a fixed and sliding window deadhead policy with a window size of 4 epochs. The edge being evaluated is on at the end of epoch 3 as shown by the blue block, but off after every other epoch. In this case, the fixed window policy needs 8 epochs to reach a deadheading decision, while the sliding window only needs 5.

there are multiple hundred gradient updates per epoch, and thus multiple hundred pruner states per epoch. Sampling just the final epoch state means the deadheading decision is based on a tiny fraction of the pruner state, which is unlikely to be meaningfully representative of the overall state. This could result in the epoch-end sampling producing a very misleading picture of the pruner's state history, and could potentially cause a deadheading policy to make decisions that do not align with how a model values an edge. For example, imagine a toe-dipping pruner that is more or less randomly fluctuating between states, but happens coincidentally to be always off at the end of epochs. Such an edge would be pruned from the model, despite the pruner making no such concrete decision. The worst case are scenarios where a per-epoch sampling policy would produce diametrically opposed decisions to that of how a model truly values an edge, and these scenarios are explored in Figure 4.9.



(a) Epoch-end sampling policy (b) Per-batch sampling policy

Fig. 4.9 Comparison of worst case scenarios for the epoch-end sampling policy. In the first case, the pruner is off 85% of the time, only on at the end of epochs 0, 1, and 2. A per-epoch sampling policy would preserve this edge, while a per-batch policy would deadhead it. In the second case, the exact opposite occurs.

The final iteration of the deadheading policy uses both a sliding window and per-batch sampling, combining the speed advantages of the sliding window with the accuracy benefits of per-batch sampling. In this final iteration, the pruner states from each of the b batches within the

previous n epochs are sampled, for a total of bn pruner states. If a pruner is on for less than 25% of those pruner states, the pruner is deadheaded. This 25% figure was chosen arbitrarily, and while it works well in practice it could be interesting to explore the efficacy of other possible options.

An example of this behavior exists in the pruners from Figure 4.7, which shows a snapshot of 1300 states for the four pruners. If the deadhead window spans the entire 1300 states, the final deadheading policy would keep the identity and dilated convolution, with those pruners remaining on for 87.3% and 66.8% of sampled states respectively. Both the 3x3 and 5x5 separable convolution would be deadheaded, having been on for only 4.1% and 1.1% of sampled states.

4.4.2 Compression

While the mere presence of pruners in a model will allow the model to remove obviously bad edges, it will need encouragement to perform more extensive pruning. This is due to codependence; models that keep a certain configuration of operations for too long will be very reticent to remove any of them. By adding a mathematical reward in the loss function for removing operations, the model is encouraged to experiment with removing operations. The process of experimentation will prevent operations from becoming codependent, thus allowing them to be pruned even faster as the penalty for removal is not particularly high. This reward is implemented by adding a *compression* term to the loss function, computed as follows:

$$\mathcal{L}_{comp} = \|c - c_{tgt}\| \tag{4.15}$$

Here, $\|.\|$ is the Euclidean norm, while c_{tgt} and c refer to the target and measured model compression respectively, where compression is some measure of how many edges have been removed from the model compared to its original state. There are a number of ways to measure this compression, and the first version of Kim et al.'s differentiable pruner paper as submitted to ICML 2019 used differentiable parameters as the metric of choice. The original number of parameters in a model was compared to the number of parameters after pruning, and the ratio between the two is the compression. For example, a model with 5 million parameters originally and 2 million parameters after pruning would have a compression of 0.4.

The issue with this metric revolves around the desired use for pruners. The intent is to trim down a supernet such that VRAM space is cleared to later scale up the model, which means that reducing the VRAM usage of the existing supernet is the primary goal. In this regard, not all parameters are created equal.

As seen in Table 4.2, there is little correlation between parameter count and VRAM cost. While identities and max pooling operations share the same parameter count of 0, they differ in VRAM cost by a factor of 4. Meanwhile, the parameter counts of the separable convolutions differ by around 1,000, but they both take up 112B on the GPU. With a parameter based compression metric, identities and pooling operations are 'free' and are infinitely cheaper than

Operation	# Parameters	VRAM Size
Identity	0	14 MB
Max Pool 3x3	0	56 MB
Separable Convolution 3x3	3,384	112 MB
Separable Convolution 5x5	4,536	112 MB

Table 4.2 Comparison of parameter count and VRAM size of a variety of tensor operations over identical input.

any of the convolutions. These operations would be unlikely to be pruned if such a metric was used in the compression term, despite them still taking up space on the GPU.

A later update to Kim et al.'s differentiable pruner paper, published as a poster in AAAI 2020, acknowledges this issue and changes the compression metric, instead comparing the FLOP ratio between the original model and the model post-pruning. However, this is still not necessarily correlated well to VRAM size, as a possibility can be imagined where an operation performs intense calculations without needing to allocate new tensors in memory. The obvious solution to these correlation issues is to directly use VRAM size as the compression metric. To do this, the VRAM cost of every edge in the model needs to be measured, which can be greatly simplified by the fact that space allocation differences between operation in the model boils down to three factors: the mathematical function the operation performs, the input dimensionality of the tensor passed to the operation, and the stride of the operation. Operations that are identical across all three of those factors will have identical VRAM size requirements, and thus all that is needed is to sample each operation in the model over every stride and input dimensionality that could appear in the model. Normal cells will contain all $|\mathcal{O}|$ operations, each with stride 1 and operating over a single input tensor dimensionality. Reduction cells will contain the same $|\mathcal{O}|$ operations, the majority with a stride of 1 and operating over some input dimensionality $[C,H,W]$. The remaining operations of a reduction cell handle spatial downscaling, and as such have a stride of 2 over an input dimensionality of $[C,2H,2W]$. As such, a normal cell has a single edge input dimensionality, while reduction cells have two. Therefore, for a model with N cells, R of which are reduction cells, there are a total of $|\mathcal{O}|((N-R)+2R)$ or $|\mathcal{O}|(R+N)$ operation sizes to sample. Figure 4.3 shows the various measurements necessary for a 3 cell, 2 reduction BonsaiNet model for CIFAR-10.

Each pruner tracks the VRAM size s of the operation it prunes, and is as such *memory-aware*. With this, two memory-aware compression metrics are defined: soft compression and hard compression. The *soft compression* $\bar{c}$ of some cell C is computed as follows:

$$\bar{c} = \frac{\text{Memory Size Unpruned Operations in } C}{\text{Memory Size All Operations in } C} = \frac{\sum_{i=0}^{\#ops \in C} s_i G(w_i)}{\sum_{i=0}^{\#ops \in C} s_i}, \tag{4.16}$$

	Stride 1			Stride 2	
	36x32x32	72x16x16	144x8x8	72x32x32	144x16x16
Identity	3.60	1.80	0.90	1.80	0.90
Avg Pool 3x3	9.00	4.50	2.25	4.50	2.25
Max Pool 3x3	27.00	13.50	6.75	13.50	6.75
Sep Conv 3x3	63.02	31.56	15.96	31.56	15.96
Sep Conv 5x5	63.02	31.56	15.95	31.56	15.95
Dil Conv 3x3	27.01	13.53	6.84	13.53	6.84
Dil Conv 5x5	27.01	13.53	6.85	13.53	6.85

Table 4.3 VRAM size in megabytes of operations versus input tensor dimensionality for a 3 cell, 2 reduction BonsaiNet model for CIFAR-10.

where $G(w_i)$ is the gate function of the ith operation's pruner. Soft compression measures the virtual mathematical compression of the model; it is the functional compression level the model is operating at. Meanwhile, the *hard compression c* measures the true allocational compression of the model, which for some cell C is computed as:

$$c = \frac{\text{Memory Size Non-Deadheaded Operations in } C}{\text{Memory Size All Operations in } C} \tag{4.17}$$

With both of these compression metrics, a separable convolution is eight times more expensive than an identity within the same cell. This lets cells make an informed decision about pruning; that is, deciding whether the task performance of a particular operation is worth its compression cost.

To translate these individual cell compressions into a metric for the entire model, the per-cell soft compressions are combined into a model compression vector $\bar{\mathbf{c}} = [\bar{c}_0, \bar{c}_1, \ldots, \bar{c}_n]$. The compression loss term is thus:

$$\mathcal{L}_{comp} = ||\bar{\mathbf{c}} - \mathbf{c}_{tgt}|| \tag{4.18}$$

Here, soft-compression is used as the metric of choice because the loss function serves to mathematically motivate model compression; the allocational details are irrelevant. This loss function is a measure of the distance between a specified target compression vector and the actual mathematical compression vector, which allows the flexibility to have distinct per-cell compression targets. Additionally, using a vector distance as opposed to something like a cellwise mean motivates each individual cell to seek their specific compression target. The importance of this can be seen by comparing the cellwise mean and vector distance compression aggregations in Table 4.4.

In this extreme example, the cellwise mean compression aggregation considers an entirely unpruned cell followed by an entirely pruned cell an equivalent compression to that of two half pruned cells. However, the goal is to distribute the burden of compression evenly throughout the cell, as an overpruned cell will disadvantage the task performance of every subsequent cell, while an underpruned cell will cause compensatory overpruning elsewhere in the model. The

Compression Metric	$\bar{\mathbf{c}}$	$\mathbf{c}_{tgt}$	$\mathcal{L}_{comp}$
Mean	[1, 0]	0.5	0
	[0.5, 0.5]	0.5	0
Vector Distance	[1, 0]	[0.5, 0.5]	0.71
	[0.5, 0.5]	[0.5, 0.5]	0

Table 4.4 The different loss penalties of the two compression aggregation strategies when targeting a uniform compression of 0.5 for an example two cell model.

vector distance aggregation motivates each cell to arrive at its specified compression level, and thus avoid over or under-pruning.

To incorporate compression loss into the model training, it is added to the main task loss as follows:

$$\mathcal{L} = \mathcal{L}_{task} + \lambda \mathcal{L}_{comp} \tag{4.19}$$

Here, λ serves to weight the compression loss against the task loss. λ must be chosen carefully, such that those two training goals are appropriately balanced; if λ is too large, the model will rapidly prune with little regard for task performance. Too small and the model will prune very slowly, and will be unlikely to perform significant pruning.

4.4.3 Choosing Lambda

This can be directly demonstrated by training and compressing three example CIFAR-10 models according to Algorithm 4, with λ values of 0, 0.1, and $1,000$. These roughly correspond to $\frac{\lambda \mathcal{L}_{comp}}{\mathcal{L}_{task}}$ ratios of 0, 0.01, and 1 respectively, therefore setting relative weights of compression versus task performance at 0, 10%, and 100%. All models will target a uniform compression of 0.5. The model with $\lambda = 0$ will "free-prune", i.e., compress solely to benefit the task performance, but will likely miss the compression target. The two models with non-zero λ will attempt to balance compression with task performance. From Equation 4.19, it might be expected that the $\lambda = 0$ model will have the best performance but the least compression, $\lambda = 1,000$ the most compression but worst performance, and $\lambda = 0.1$ somewhere in the middle. The full results of three trials are shown in Table 4.5:

λ	c	CIFAR-10 Accuracy
0.0	0.729	**94.62**
0.1	**0.50**	94.44
1000.0	0.51	89.85

Table 4.5 Compression versus accuracy for a variety of compression loss weightings.

The $\lambda = 1,000$ model performs drastically worse than the other two, around 4.5 percentage points behind both the $\lambda = 0.1$ and $\lambda = 0$ model. To further examine the differing behaviors of

the three models, specifically their pruning pace, Figure 4.10 shows the number of operation deadheads per epoch:

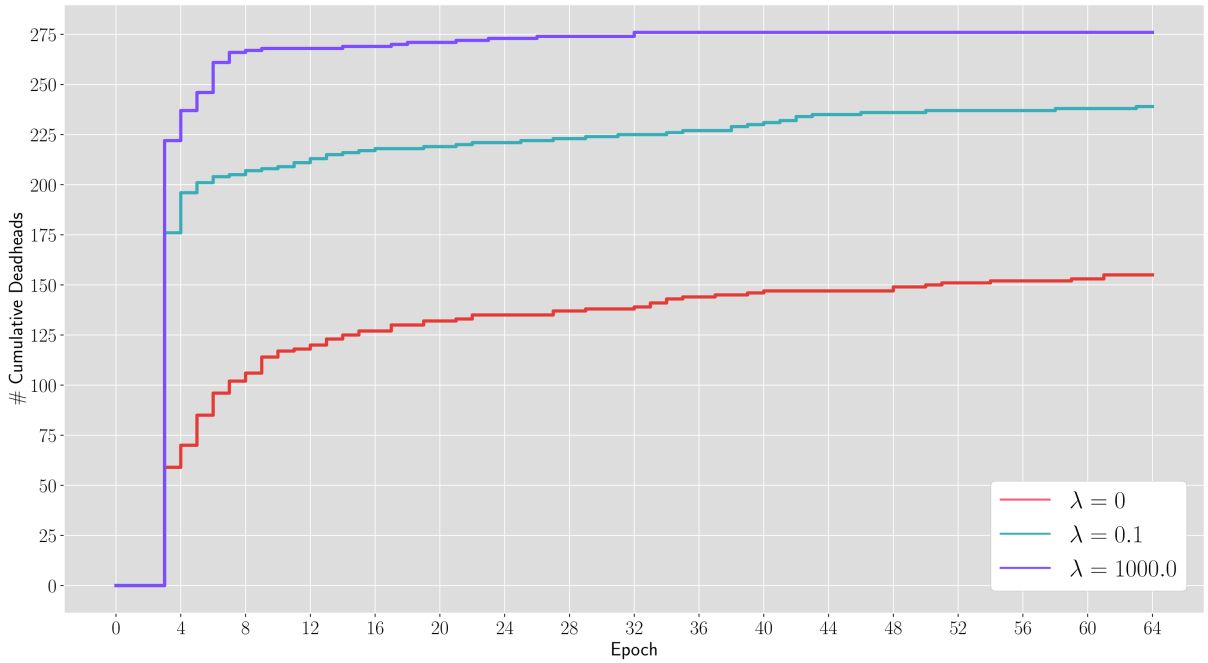


Fig. 4.10 Number of deadheads by epoch for the three models from the λ experiments. The deadheading cycle for each of these models is 4 epochs, hence the massive spikes at the fourth epoch.

Notice that the most extreme $\lambda = 1{,}000$ model performs 81% of its total deadheading at the first possible deadheading opportunity, and 96% by the third cycle. While not shown in the figure, its soft-compression is even faster: within the first few data batches it had soft-compressed to the target 0.50 mark. Its lackluster performance suggests that while the aggressive pruning strategy is good at rapidly performing compression, it lacks the refinement to ensure that this rapid compression is beneficial in the long-term to its performance. It seems likely that the memory-performance balance of operations evolves over time, and thus basing all compression on that balance within the first few data batches is short-sighted.

Meanwhile, the $\lambda = 0.1$ model achieves similar levels of performance to the free-pruning ($\lambda = 0$) model, while still attaining the goal compression. It also performs a large part (74%) of its deadheading after the very first cycle, but the remaining 26% of the total compression is more gradually distributed over the remaining training epochs. This is indicative of the approximate balance being sought with the compression λ; it needs to motivate enough compression that the model will quickly prune to the correct size, but ensure that said compression is performed in a careful and prudent manner such that it is not massively detrimental to the potential performance.

A final notable observation from this experiment is that the free-pruning $\lambda = 0$ model still compressed by around 27%; this model found that removing certain internal operations benefited its classification performance, which runs slightly counter to the general deep-learning intuition that bigger is better. This phenomenon is directly examined in Section 4.9.5. The actual decisions

being made by the model can be better understood by looking at how frequently certain operations are chosen across the model, as shown in Figure 4.11.

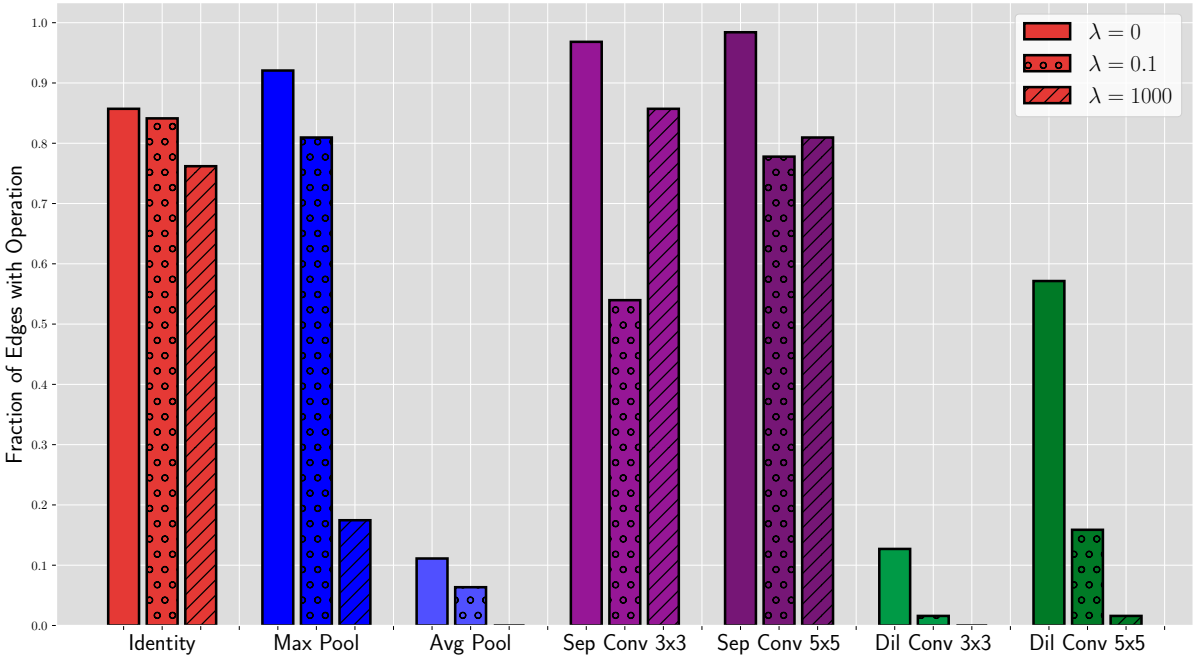


Fig. 4.11 Operation choices by the three models. The y-axis represents the fraction of the edges in the model than contain a certain operation.

Using the freely-pruning model as a reference for the purely task-optimal operation frequencies, notice that the $\lambda = 0.1$ and $\lambda = 1,000$ models vary in behavior versus the reference. Notably, they make opposite decisions in regard to the max pooling and 3x3 separable convolutions; the $\lambda = 1,000$ model removes the majority of max poolings, while preserving around 85% of the 3x3 separable convolutions. The $\lambda = 0.1$ keeps 80% of max poolings, while removing around half of the 3x3 separable convolutions. Perhaps this was a decision motivated by a tradeoff between memory cost and the immediate performance gains provided by these operations; remember, the $\lambda = 1,000$ soft-compresses near instantly. In the first few training batches, the model needs to begin moving from some randomly-initialized state towards a learned optima. This tweaking of weights from completely random to slightly learned in the first epoch is the largest single-epoch jump in performance that will likely occur in all of training, as there is simply so much potential performance to gain starting from fully random initialization[2]. The separable convolutions are immensely powerful in this early training scenario as they have a large amount of parameters and thus ample room to improve from the random state. Max poolings on the other hand are relatively memory-expensive (usually exactly half the cost of separable convolutions) but are fixed filters with no learning ability, and as such do not provide as much immediate power to the model. With this in mind, the behavior of the $\lambda = 1,000$ model can be understood; with only the context of the first few data batches, it removes en masse the most expensive operation that is providing no immediate advantage over random initialization.

[2]Case in point: the $\lambda = 0.1$ model in its randomly-initialized state has a CIFAR-10 accuracy of 10.00%. After 1 epoch, its accuracy was 57.20%, a performance gain of 47.26 percentage points. The next largest single-epoch performance gain was only 7.64 percentage points.

Given the results of these various experiments, it appears that the $\lambda = 0.1$ model provides the best compromise between task performance and pruning.

4.4.4 Integrating Pruning

With pruners and compression defined, the next step is to integrate them into the architecture. Revisiting the edge calculation as presented earlier, it computes the sum of the (potentially) pruned output of each candidate operation:

$$E^{out} = \sum_{i=0}^{|\mathcal{O}|} P_i^o(o_i(E^{in}), w_i^o) \tag{4.20}$$

While this looks to accomplish the task set out for pruners; i.e., differentiably select which operations to keep and which to prune, there is a major issue that arises in practice. This issue is that the output of any such edge E_i^{out} is the input to every subsequent edge E_{i+j}^{out} in its particular cell. Any learned weights from such a downstream operation will be learned as a function of that downstream operation's inputs, which are explicitly a function of the upstream operation configuration. A change to that upstream configuration such as an operation pruner moving from on to off will add or remove an element from the Equation 4.20 sum. This drastically changes the output distribution of the upstream edge, which in turn is reflected in the input distributions of downstream operations. When these input distributions are changed, the learned weights of downstream operations are made useless as they are no longer relevant to the actual tensor values flowing through the operations, the effects of which is highly detrimental to the loss of the model. As such, changes in operation configuration (and thus changes in pruner state) can cause massive spikes in the loss, and are thus highly discouraged in gradient descent.

This exactly mirrors the covariate shift problem described in Section 2.6.3. As such, the exact same solution to the problem as proposed by Ioffe and Szegedy (2015) is applicable: batch normalization. By modifying the edge calculation as follows:

$$E^{out} = \text{BatchNorm}\left(\sum_{i=0}^{|\mathcal{O}|} P_i^o(op_i(E^{in}), w_i^o)\right) \tag{4.21}$$

the operation configuration is decoupled from output distribution. With this calculation, the 'off' pruner state can be thought of not as setting the operation output to zero, but rather to the mean of the edge output distribution. This way, pruners can experiment with preserving or removing an operation without affecting the input distributions of every downstream edge. Furthermore, it allows models to dynamically prune during training, as operation removal does not affect the relevance of the model's existing learned weights.

4.4.5 Pruning Summary

With memory compression made possible by the combination of memory-aware pruning and deadheading, it is now possible to initialize a model and set an arbitrary compression target for it to reach. When the model is trained with compression loss, computed from the distance between the current compression and the target compression, pruners will gradually switch off in accordance to however the model sees fit. As pruners remain off for extended periods of time they will be deadheaded, slowly reducing the model's memory footprint to the desired size. Passing the output of the pruners through batch normalization ensures that pruning does not shift the output distribution of edges, ensuring that the pruning process is as non-disruptive as possible to model training. In summary, these components combine to create a method of compressing an arbitrary model to a specific VRAM size in a way that is minimally detrimental to the performance of said model.

4.5 Cell Search

To solve the problem of searching for unique cells via a supernet technique, an iterative stacking approach was chosen. To do this, a virtual supernet is created consisting of however many cells is desired. This supernet is then divided into *model sections*, groups of consecutive cells within the model, such that each section is roughly the same size and the first section can fit into a specified max VRAM footprint s_{max} in the fully superconnected state. This max VRAM footprint is often the currently available free space on the GPU VRAM, but could also refer to the space on the desired deployment hardware or simply the desired final size of the model.

To concisely and specifically describe the cell search process, the following notations are defined:

Definition 4.5.1. A model section is a group of consecutive cells, such that each section is roughly the same size. For example, a model with nine cells might be comprised of three sections of three cells each: $[C_0, C_1, C_2]$, $[C_3, C_4, C_5]$, and $[C_6, C_7, C_8]$

Definition 4.5.2. To describe a composition of model sections that form a larger model:

$$M^{[0,1,\ldots,n]}_{[c,c_1,\ldots c_n]} = M^0_{c_0} + M^1_{c_1} + \cdots + M^n_{c_n} \tag{4.22}$$

Here, $M^i_{c_i}$ refers to the ith model section hard-compressed to some level c_i, while $M^{[0,1,\ldots,n]}_{[c_0,c_1,\ldots,c_n]}$ refers to a model comprised of sections 0 through n, each hard-compressed to a level c_0 through c_n, respectively. Using this notation, a section i in the superconnected state would be written as M^i_1, while a fully compressed section (i.e., empty) would be written as M^i_0. The notation $M^0_{c_0} + M^1_{c_1}$ indicates appending section $M^1_{c_1}$ after $M^0_{c_0}$, such as to combine them into one contiguous graph $M^{[0,1]}_{[c_0,c_1]}$. This operation is shown graphically in Appendix A, Figure A.5.

Definition 4.5.3. The ith *subset* of M refers to a model $M^{[0,1,\ldots,i]}_{[c_0,c_1,\ldots,c_i]}$, where i is less than the total number of sections in the full model M.

To perform cell search using this stacking, first the initial section M^0_1 is loaded onto the GPU. A classification tower is added after this section, such that M^0_1 can be treated as a fully functional standalone model; the 0th subset of M. M^0_1 is then trained and pruned until enough space is freed such that section M^1_1 can be appended to the model. The model, the first subset of M, is now:

$$M^{[0,1]}_{[c_0,1]} = M^0_{c_0} + M^1_1 \tag{4.23}$$

The existing classification tower is then converted to an auxiliary tower, and a new classification tower is added after section 1. This extracts further benefit from the old classification tower; while technically no longer necessary, auxiliary towers help regularize a model as described in Szegedy et al. (2014). This growing and pruning process is then cycled until the final section can be added, and the model is:

$$M^{[0,1,..,n-1,n]}_{[c_0,c_1,\ldots,c_{n-1},1]} = M^0_{c_0} + M^1_{c_1} + \cdots + M^{n-1}_{c_{n-1}} + M^n_1 \tag{4.24}$$

See Appendix A, Figure A.5 for a graphical representation of this growth process and its effect on the model's connectivity.

At this point, all sections 0 through $n-1$ have compressed to some c_0 though c_{n-1}; each of these sections is now ideally at the optimal architecture for its respective compression level. However, in order for this process to be feasible, the space requirements of each section must be established, such that a) the compression target for that section of the model is known and b) when there is enough free space to stop compressing and add the next section. This is to say, it is necessary to know the largest $c_0,\ldots,c_i$ such that $M^{0,1,\ldots,i}_{c_0,\ldots,c_i} + M^{i+1}_1$ fits into the desired VRAM footprint. To do this, there needs to be an accurate estimate of the VRAM size of the ith model subset.

4.5.1 Operation and Model Size Estimation

In order to test whether some model subset M at a specific hard-compression level c will fit into s_{max}, a *simulation model* is used. A simulation model is simply some model M' that has an identical structure to M, that is, one with identical number of cells, edges, and edge connectivities. However, at initialization, M' has no operations along any of the edges. The simulation model M' is then filled with operations until it reaches c, at which point its size can be measured to check fit.

The problem of filling the model with operations can be modeled as a change-making problem (Wright, 1975), where a certain number of operations are picked such that the overall compression of the model is as close as possible to the target compression. In order to perform this accurately, an accurate estimate of the size of each operation is needed, that is, the amount of VRAM used by this specific operation within the model. Ideally, it should be an additive

operation size estimate, one such that the sum of the sizes of all component operations is equal to the total size of the model.

To do this, five factors need to be considered. First is the allocation required by the operation itself, that is, the space needed to store the weights of the operation and the pruner. Second is the allocation required to perform forward and backward computation of the operation, which requires storage of output, intermediate, and gradient tensors. Third is the the size consequences of interaction effects between operations. Namely, as each operation arrives in a node, it must be summed together with the other inbound operations. Therefore, each operation except for the very first into a node incurs a summation operation, which results in a tensor allocation. That means that in the most common case (where an operation is not the first operation to arrive at a node) this extra summation and therefore tensor allocation occurs. Fourth, the "warmup" effects displayed by certain operations must be accounted for; certain operations will allocate a certain amount of memory for their first computational pass, then ask for slightly more in the second. After the second computation, their allocation remains fixed at the new, higher level. This extra allocation comes from operations that contain batch normalizations which require tensor space to store statistics from the current and previous batches, the latter of which is unnecessary for the very first batch. Finally, the non-monotonic allocational behavior of PyTorch must be measured. The exact space used by an operation can rapidly fluctuate throughout the process of its computation as tensors are allocated and deleted as needed. PyTorch will allocate space according to the maximum allocation that occurs during an operation's lifespan; regardless of whether this maximum is a transient peak, it nevertheless sets the permanent allocational size of the operation and whether an out-of-memory error occurs.

The algorithm used to estimate the size of operations within BonsaiNet models is given in Algorithm 1:

Algorithm 1: Compensated Operation Sizing

let $\mathcal{M}$ refer to measuring the current Torch memory allocation;
let $n_{repetitions}$ refer to the number of measurements to average/summarize over;
$m = 0$;
for $0 \rightarrow n_{repetitions}$ **do**
    $\mathbf{m} = \varnothing$;
    $m_{pre} = \mathcal{M}$;
    Initialize operation o and pruner p;
    for $0 \rightarrow n_{repetitions}$ **do**
        $\mathbf{m} = \mathbf{m} \bigcup (\mathcal{M} - m_{pre})$;
        Pass sample input of dimensionality *d forwards* through o and p, store output;
        $\mathbf{m} = \mathbf{m} \bigcup (\mathcal{M} - m_{pre})$;
        Add output to itself;
        $\mathbf{m} = \mathbf{m} \bigcup (\mathcal{M} - m_{pre})$;
        Pass output through sample loss function;
        $\mathbf{m} = \mathbf{m} \bigcup (\mathcal{M} - m_{pre})$;
        Pass gradient *backwards* through o and p;
    $m = m + \max(\mathbf{m})$
Result: $\frac{m}{n}$

The algorithm first makes note of the current GPU allocation (Line 5) prior to the start of size estimation. Line 6 initializes an identical operation and pruner to the estimated operation, thus ensuring that the allocation of the operations themselves are measured. Line 8 measures the allocational requirements of the forwards pass of the operation, while lines 12 and 14 measure the backwards pass. Line 10 sums the output of the estimated operation with itself, and thus simulates the allocational interaction effects that would occur in node summations of the real model. The forwards-backwards passes are repeated $n_{repetitions}$ times (typically 5), to account for potential warmup effects. After each step of the algorithm, the current memory allocation is compared to the initial allocation (lines 7, 9, 11, and 13). This allows Line 15 to measure the size of the largest allocational peak, and thus account for the potentially non-monotonic allocation throughout the calculation of the operation. This largest allocation peak is stored as the allocation requirement for this specific simulation in line 15. Finally, n total of these simulations are run, and the average allocation requirement over the $n_{repetitions}$ simulations is returned as the estimated size of the operation.

With a hopefully accurate and additive operation size estimate (both these assumptions to be evaluated later in this section), it is possible to design a method of packing the simulation model M' to the target compression level. To do this, the problem is approached one cell at a time, with each cell allocated operations independently of one another. For each cell C and for each operation dimensionality d with that cell, the set of operations $\mathcal{O}_{C,d}$ within is collected. Since the operation size estimation is ideally an additive one, this means the total size of that set of operations can be easily estimated as the sum of the estimated size of each operation:

$$\text{size}(\mathcal{O}_{C,d}) = \sum_{o \in \mathcal{O}_{C,d}} \text{size}(o) \tag{4.25}$$

The total size of the cell is therefore:

$$\text{size}(C) = \sum_{d \in C} \sum_{o \in \mathcal{O}_{C,d}} \text{size}(o) \tag{4.26}$$

To find the allocational size of the cell that gives the desired compression, the cell size is multiplied by c_{tgt}. By the distributive property:

$$s_{tgt} = c_{tgt} * \sum_{d \in C} \sum_{o \in \mathcal{O}_{C,d}} \text{size}(o) \tag{4.27}$$

$$s_{tgt} = \sum_{d \in C} c_{tgt} \sum_{o \in \mathcal{O}_{C,d}} \text{size}(o) \tag{4.28}$$

This means that the allocational problem can be reduced to finding the set of operations $\mathcal{O}' \subset \mathcal{O}_{C,d}$ such that $\text{size}(\mathcal{O}') = c_{tgt} * \text{size}(\mathcal{O}_{C,d})$ for all C, d within the model. This can be accomplished by using a greedy change-making algorithm, that selects operations for the chosen subset according to some heuristic over the available operations. In the process of designing this algorithm three different heuristics were compared: maximum, random, and minimum selection. All three

heuristics create a set of available operations $\mathcal{O}_{avail}$, which contains all unselected operations that can be added to $\mathcal{O}'$ without increasing its total size above the target. The maximum heuristic then selects the largest operation within $\mathcal{O}_{avail}$ to be added to $\mathcal{O}'$, the minimum selects the smallest, and the random selects an operation at random. These heuristics were chosen based on solutions to the change-making problem as described in "Optimal and Canonical Solutions of the Change Making Problem", Martello and Toth (1980). Specifically, the maximum heuristic matches Martello and Toth's 'canonical solution', designed to find a solution with approximately the minimum number of operations possible. The minimum heuristic is designed to do the opposite, to approximate a solution with the maximum number of operations, while the random heuristic should find solutions that lie somewhere in the middle. Once the heuristic selects an operation, it is added to $\mathcal{O}'$, and the algorithm repeats. Once there are no more operations that are eligible for $\mathcal{O}_{avail}$, the $\mathcal{O}'$ is fully allocated and the algorithm ends. This process is detailed in full in Algorithm 2.

Algorithm 2: Operation Allocation

let $\mathcal{O}'_{M'}$ be the set of chosen model operations, $\mathcal{O}'_{M'} = \varnothing$;
let $s_{current} = 0$;
for cell $C \in M'$ **do**
  for operation dimension $d \in C$ **do**
    let $\mathcal{O}_{C,d}$ equal the set of all operations in c with dimension d;
    let $\mathcal{O}'$ be the set of chosen cell operations, $\mathcal{O}' = \varnothing$;
1   let $s_{tgt} = c_{tgt}\left(\sum_{o \in \mathcal{O}_{C,d}} \text{size}(o)\right)$;
    while $s_{current} < s_{tgt}$ **do**
      let s_{min} be the size of the smallest operation available, $\min_{o \in \mathcal{O}_d} \text{size}(o)$;
      if $s_{min} + s_{current} > s_{tgt}$ *or* $\mathcal{O}_{C,d} \setminus \mathcal{O}' = \varnothing$ **then**
        break;
      else
2   let $\mathcal{O}_{avail} = \{o | o \in \mathcal{O}_{C,d} \setminus \mathcal{O}', \quad s_{current} + \text{size}(o) \leq s_{tgt}\}$;
        Choose o_{chosen} from $\mathcal{O}_{avail}$ according to some heuristic $H(\mathcal{O}_{avail})$;
        $\mathcal{O}' = \mathcal{O}' \cup \{o_{chosen}\}$;
        $\mathcal{O}_{C,d} = \mathcal{O}_{C,d} \setminus \{o_{chosen}\}$;
3       $s_{current} = s_{current} + \text{size}(o_{chosen})$;
    $\mathcal{O}'_{M'} = \mathcal{O}'_{M'} \cup \mathcal{O}'$;

The rationale for evaluating different heuristics of operation selection is to determine the accuracy of the operation size algorithm as well as evaluate how well it aligns with the desired additive property. The three heuristics build models of varying operation counts; the maximum heuristic builds models with minimal operation counts, the minimum heuristic with maximal operation counts, and the random somewhere in the middle. If the size estimation algorithm is accurate and additive, any model produced by Algorithm 2 should be very close to s_{tgt} in size, and thus should have compression close to c_{tgt}, regardless of operation count and therefore selection heuristic. If the sizing algorithm is inaccurate or non-additive, the calculations in Algorithm 2 that rely on said sizing (lines marked 1, 2, and 3) will be similarly inaccurate. Chiefly, line 3 is crucial in determining the stopping point of the algorithm, and as such any inaccuracy will

produce models that miss s_{tgt} or c_{tgt}. High operation count models like those produced by the minimum heuristic would increase the number of times these summation inaccuracies might occur, and thus produce models of differing s_{tgt} or c_{tgt} as compared to the other heuristics. The results of operation selection across the three heuristics are shown in Figure 4.12.

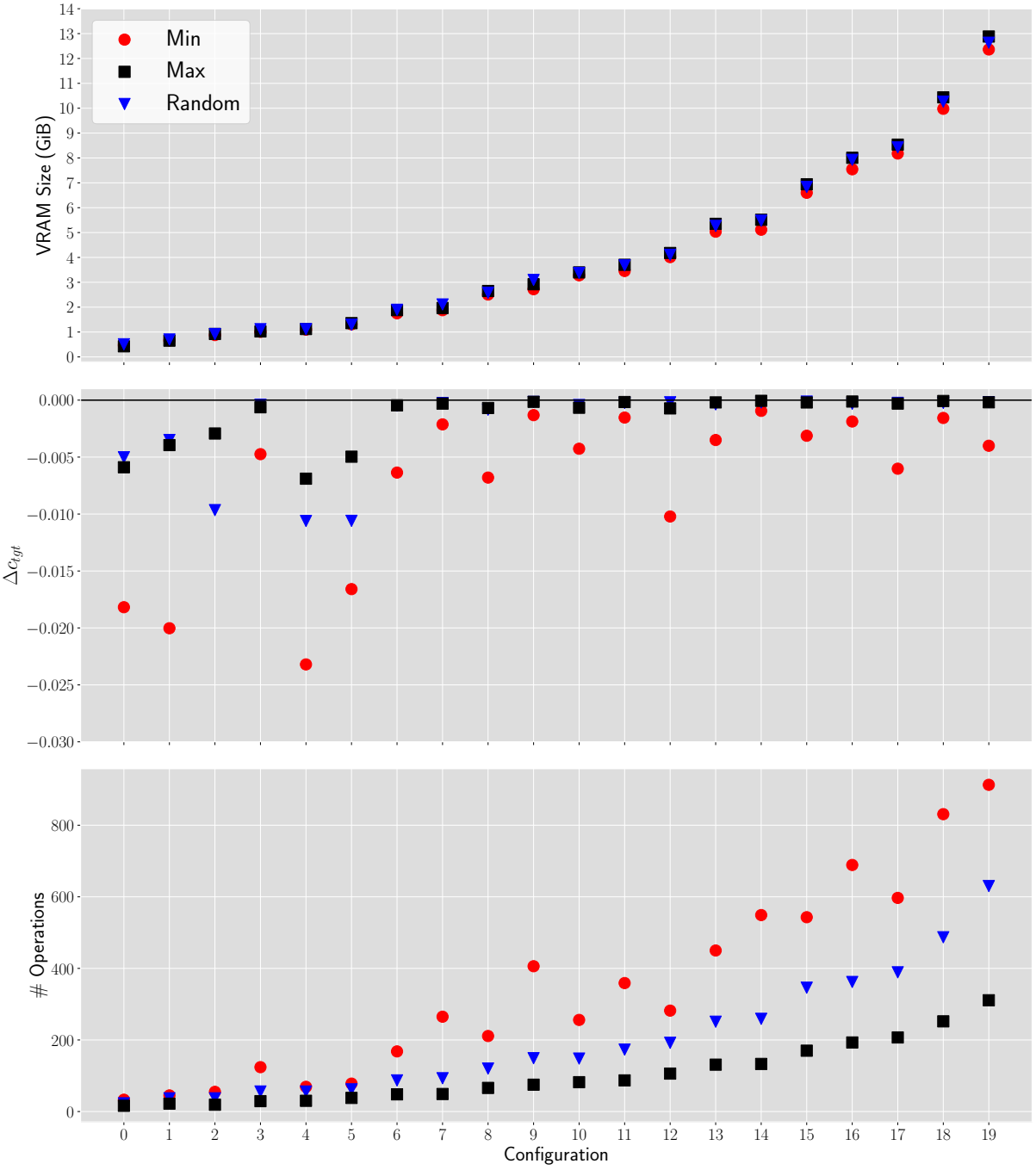


Fig. 4.12 Comparison of three different operation selection heuristics within the greedy-change making algorithm. 60 models were allocated in total, corresponding to 20 different model specifications each allocated operations by three heuristics. Shown are the VRAM size (top), difference between target compression and allocated compression (middle), and number of operations (bottom) plotted against configuration and heuristic.

To evaluate this, 20 different model configurations are packed according to the three heuristics and measure the resulting model's VRAM size, delta to compression target, and operation count.

To measure the VRAM size of any particular model, the VRAM allocation prior to the model's initialization is first noted. Then the model is moved onto the GPU, and the VRAM allocation after passing two full data batches through the model both forwards and backwards is measured. Two batches are necessary for accurate size measurement due to the same allocational "warmup" condition as described earlier. To avoid potential stuck-tensor issues if the simulation model encounters an out-of-memory error, simulation models are initialized in a subprocess forked from the main process. This means there is a separate Torch context initialized in the subprocess, which means that all tensors initialized in the subprocess are compartmentalized from those in the main process. If the subprocess simulation model encounters an out-of-memory error, the process returns a failing exit code and terminates. When the subprocess terminates, its Torch context deletes and garbage collects all tensors that were initialized within it, cleaning any potentially orphaned tensors off the GPU. Essentially, the subprocess sandboxes the simulations from the main process, meaning size testing can occur without worrying about destroying the viability of the main process.

Notice that the three heuristics do indeed produce models that are consistently ranked in terms of operation count (bottom plot); the maximum heuristic chooses the fewest operations, followed by random, and minimum chooses the most. Despite this, the VRAM size of the models are highly consistent despite the operation count disparities, meaning that maximum heuristic is choosing a few large operations, while the minimum is choosing many small operations[3]. Overall, the sizing demonstrates the exact behavior hoped for: the operation size estimator is accurate and additive, and that the compression of models is an accurate indicator of its VRAM size.

In conclusion, Algorithm 1 provides accurate and additive operation size estimates, which allows Algorithm 2 to accurately produce simulation models M' of an arbitrary compression level. This in turn checks whether a model of that compression level will fit into VRAM. This is tremendously useful for this use-case because there is no way of providing any guarantee about a model's real-world connectivity and operation count as it prunes to a certain compression level. It might prefer a small or large operation count, meaning that the way that is achieves that specific compression is highly dependent on its individual circumstances. With accurate sizing, it is possible to know for sure that any model at a particular compression will fit if the simulation model does.

4.5.2 Compression Targets

With a reliable method of checking whether an arbitrary model subset at a specific compression will fit into the desired amount of VRAM, the compression targets for each section of the model can be set. This is approached from an inverse perspective; rather than estimate the size s_M of a model subset and setting the compression target c_{tgt} as $\frac{s_{max}}{s_M}$, instead the largest compression level

[3]One thing to note is the minimum heuristic's consistent inability to fully reach c_{tgt}. This is because solutions to the change making problem assume an infinite number of available choices at each size, whereas in this case there are a finite number of operations to select. The minimum heuristic, by way of trying to select maximum operations, sometimes runs out of available operations before it can satisfactorily produce a model of size s_{tgt}.

is found such that subset plus the new section fits into a given maximum VRAM allocation s_{max}:

$$c_{tgt_i} = \max\{c \in [0,1] \mid \text{size}(M^{0,\ldots,i-1}_{c_0,\ldots,c_{i-1}} + M^i_1) \leq s_{max}\}, \tag{4.29}$$

Since the process of testing a particular simulation model size involves model initializing, operation packing, GPU transfer, and a couple of data passes, it is relatively slow; usually on the order of a couple of seconds per test. This means the total number of size tests needs to be minimized. To search the compression space (the $c \in [0,1]$ term from Equation 4.29) efficiently, a binary search tree is traversed, as per Algorithm 3:

Algorithm 3: Compression Target Search

```
let c_tgt = [0, 0, ..., 0];
let d be the maximum depth of the search tree to search for n_section ∈ [0, #sections] do
    let c = 0.5;
    let Δ = 0.25;
    for i ∈ [0, d] do
        if size(M_c^{0,...,i-1} + M_1^i) ≤ s_max then
            c_tgt_i = c;
            c = c + Δ;
        else
            c = c − Δ;
        Reduce step size unless at final step (this allows c=0 and c=1 to be attainable);
        if Δ > 2^{−depth−1} then
            Δ = Δ/2;
```

The depth parameter d of the search tree controls the search granularity: the finest increments that can be searched by the algorithm. For the purposes of BonsaiNet, the depth was set to 6 as it provides a search granularity of $\frac{1}{2^6} \approx 0.015$, meaning that the space between 0 and 1 is searched in increments of 0.015. This therefore sets the upper bound at the possible error in the compression target search algorithm, as the distance between the optimal compression target and the one returned by the algorithm will be at most 0.015. In a future implementation, the optimal search depth could also be automatically calculated from the model configuration:

$$\text{depth} = \lceil \log_2 \frac{\sum_{o \in M} \text{size}(o)}{\min_{o \in M} \text{size}(o)} \rceil, \tag{4.30}$$

where $sum_{o \in M}$ iterates over each operation o in the model. Compression is implicitly a discrete metric, where the smallest quanta is determined by the smallest operation within the model. The above equation would guarantee that the search granularity is always marginally finer than the compression difference created by toggling on or off the smallest operation in the model.

This algorithm provides us with c_{tgt} values for each section of the model. However, these targets represent the compression levels that are at the absolute extreme of what can fit into the desired VRAM footprint; the search depth of 6 means that while c_{tgt} might fit, $c_{tgt} + 0.015$ will not. As such, it is absolutely crucial that the models arrive underneath these targets, simply to guard against out-of-memory errors and the stuck tensors issues they cause. However, there

is an even larger issue with passing these compression targets to the loss function, and that is due to the means by which compression loss computes $\mathcal{L}_{comp}$. As seen in equation 4.18, the $\mathcal{L}_{comp}$ is proportional to the square distance of the actual compression to the target compression. This means that as the model approaches the target compression, $\mathcal{L}_{comp}$ falls exponentially, thus exponentially slowing further progress to the target. This initiates a negative feedback loop, eventually halting all progress at compression level marginally above the target. Therefore, in order to actually meet the compression targets the model needs to aim to overshoot them. To do this, define:

$$\bar{c}_{aim} = \begin{cases} 0.90c_{tgt}, & c_{tgt} > 0.35 \\ 0.66c_{tgt}, & c_{tgt} \leq 0.35 \end{cases} \tag{4.31}$$

And redefine:

$$\mathcal{L}_{comp} = ||\bar{\mathbf{c}} - \bar{c}_{aim}|| \tag{4.32}$$

This establishes an overshoot $\bar{c}_{aim}$ that is marginally smaller than c_{tgt}, with the margin varying by the size of c_{tgt}. The larger margin at compression levels below 0.35 is experimentally motivated; compressing down below that size tends to incur significant task performance penalties, which highly disincentivizes the model performing significant compression in those regions. Increasing the overshoot here effectively increases the weight of the compression penalty term as compared to the task performance loss, in an attempt to rebalance the two and motivate compression in high-compression domains.

There are now two different compression targets c_{tgt} and $\bar{c}_{aim}$ that arise from Algorithm 3 and Equation 4.31 respectively. c_{tgt} is a physical hardware constraint: it refers to the biggest the model can possibly be to avoid exceeding the VRAM allocation restriction s_{max}. As such, this corresponds to the hard-compression of the model: the compression produced by the permanent deletion of deadheading operations from the model. However, hard-compression must lag behind soft-compression, as deadheading can only occur when an operation has been consistently turned off. This is to say the hard-compression of an operation can only occur after it has been consistently soft-compressed, which therefore lags the two metrics. By setting $\bar{c}_{aim}$ to be proportionally smaller that c_{tgt}, this lag can be compensated for such that when the soft-compression $\bar{c}$ reaches $\bar{c}_{aim}$, the hard-compression c reaches c_{tgt}.

4.5.3 Breaking Codependence

While training a model with the adjusted $\bar{c}_{aim}$ compression target ensures that the model will eventually reach a compression level below c_{tgt} if sufficiently motivated to do so, often the speed of compression will slow significantly after the first deadheading cycle. If left unchecked, compression will eventually come to almost a complete halt, which prevents a model from reaching c_{tgt} if it starts sufficiently far from the target. This can be easily demonstrated by

looking at the number of deadheads per epoch for each of the three example models from Section 4.4.3, and the plateauing compression is shown in Figure 4.10.

Notice that the number of deadheads per epoch scales with the λ value, but trails off significantly in each model after the first 12 or so epochs. This slowing is due to two main reasons. First, the compression loss scales with the square of distance, so as the model gets closer to the aim, the rate at which it slows decreases correspondingly. The second reason is codependence as described in Section 4.4.4, where operations' weights will become more dependent on each other the longer they remain jointly active. This demotivates compression, as any removed operation will negatively affect any and all codependent operations. While the presence of batch normalizations along edge outputs mitigates these effects to some degree, they can only ensure that edge outputs lie within a consistent overall distribution, not that the content of that output aligns with what is expected by downstream operations.

There are three ways to try to avoid or break this codependence. The first way, and the easiest, is to ensure that the model configuration is such that none of the c_{tgt} values of any model subsection are particularly challenging. As seen in Table 4.5, models will naturally compress to levels around 0.70, and have no trouble getting down to 0.50. Beyond around 0.40, however, and models start taking quite a long time reaching targets due to the aforementioned slowing factors. By simply ensuring that the configuration avoids such values this problem is circumvented, which can be accomplished by strategies like reducing the number of operations per edge or nodes per cell.

Second, the slowing compression can be countered by increasing the compression penalty over time. Here, the λ value is doubled after each deadheading cycle where $c > c_{tgt}$. This means that the penalty for not reaching the compression target increases for each cycle that the target is not met, the idea of which is that it compensates for the exponentially decreasing compression loss. The issue is that this can run into a similar problem as exhibited by the $\lambda = 1,000$ model; with a high enough λ, models are so desperate to reduce the penalty that they adopt an overly greedy pruning strategy. If the model spends too much time above the compression target, the doubling λ could eventually force it into suboptimal pruning.

Finally, the observation that the first deadheading cycle is most effective can be exploited: simply repeat the first deadheading cycle as many times as necessary. Essentially, the model loops through the first n epochs (where n is equivalent to the deadhead cycle length) a number of times, reinitializing the model weights after each deadheading cycle. This forcibly breaks codependence, as the model weights simply cannot form long term dependencies on each other when they are consistently reinitialized. After reinitialization, the model can more safely experiment with switching off operations as it will not severely affect other operations' viability, which leads to larger amounts of compression in a comparatively quick time. During search, the reinitialization and pruning cycle can repeat as many times as necessary until $c < c_{tgt}$. To preserve parity between reinitialized models and non-reinitializing models, any epochs that the model spends in this reinitialization cycle are subtracted from its training epoch budget. If compared against a non-reinitializing model that trained for 600 epochs, a reinitializing model that spent 100

epochs cycling through reinitialization and pruning would only receive 500 subsequent epochs for training.

Along with speeding up compression, this latter approach can help models more quickly adapt to and integrate new sections; if a new section is added without global reinitialization the model can struggle to utilize the new operations. This is again similar to the covariate shift problem from Section 4.4.4, as the model now needs to pass the outputs of all of its trained operations through these new randomly initialized operations that have been added. However, this is almost exclusively detrimental to the performance of the already trained operations in earlier sections, considering they were trained to be optimal on their own. In these circumstances the model may simply prune out all of the new operations, entirely unable to use them beneficially. By reinitializing such a model, there is no distinction between old and new operations, and all operations can be learned simultaneously and collaboratively. This should provide a performance benefit to the model, as it will allow the model to better utilize the power of all of its operations jointly. This is explored in Table 4.6.

	CIFAR-10 Accuracy	Search Time (hours)[a]	Total 1080Ti Runtime (hours)
Normal	96.46	2.38	88.06
Reinitialization	**96.72**	**1.72**	**82.31**

[a]This refers to the time required for a model to grow to its full size as per Algorithm 4, after which it trains with no mandatory compression targets.

Table 4.6 Search performance for reinitializing and non-reinitializing models given identical search configurations.

Here, notice that the reinitializing model completes its compression targets around 40 minutes faster than the non-reinitializing model, and after training the model is 0.26 percentage points more accurate. The performance gain is unlikely to be caused by better search performance (i.e., is not related to the intrinsic capabilities of the architecture found in the search phase) but rather by the reinitializing model's ability to better use and train its operations jointly.

4.6 Bonsai Algorithm

The main search algorithm can now be stated, which due to its growth and pruning cycles is dubbed the Bonsai process. The complete algorithm for searching and training a model is shown in Algorithm 4.

The Bonsai algorithm consists of three phases: Preparation, Growth Cycling, and Training. Preparation, which starts at line 1, first uses Algorithms 3 and 2 to determine the compression targets for each subset of the model. After this, the first model subset is initialized. From here, the Growth Cycling phase begins, starting at line 2. This first involves the training and deadheading cycles described in Section 4.5, wherein the model subset is trained and deadheaded in repeating cycles. This starts at line 3, and continues until the desired compression level is met. At this point (line 4) the next section can be appended onto the current model subset. Once the entire model

Algorithm 4: The Bonsai Process

1 Determine the compression targets $c_{tgt} = [c_1, \ldots, c_n]$;
  Initialize the first model subset M_1^0;
  Initialize search epoch counter $E_{search} = 0$;
2 **for** *i in* $[1, \#_{subsets}]$ **do**
    $E_{cycle} = 0$;
3   **while** $c_i > c_{tgt,i}$ **do**
      Train + prune towards $\bar{c}_{aim}$;
      Deadhead;
      $E_{search} = E_{search} + 1$;
      $E_{cycle} = E_{cycle} + 1$;
      if $E_{cycle} >$ Deadhead Cycle Length **then**
        Reinitialize model, and/or set $\lambda = 2\lambda$;
4   Convert classification tower to auxiliary tower;
    Add next model subset M_1^i;
    Add new classification tower;
5 Set the L_{comp} penalty weighting λ to 0;
  Train + prune freely for the remaining $E - E_{search}$ epochs;

has been fully constructed via this process, the Training phase can begin, starting at line 5, which simply involves training the constructed model for the desired number of epochs. For practical details of how this algorithm was implemented in PyTorch, see section A.5 in Appendix A.

4.7 Bonsai Models

With the search algorithm and implementation details described, the use of BonsaiNet in practice can now be explored. There are six architectural hyperparameters that control the structure and behavior of BonsaiNet over a specific dataset:

1. **GPU Space**: The maximum amount of VRAM space in gigabytes the Bonsai model is allowed to allocate. This is typically either the maximum available space on the GPU, or the desired final size of the model such as to fit within a specific hardware constraint. This sets the compression level that each pattern has to fit into.

2. **Scale**: The channel size of the first cell. Upscaling the channel count from the input to the scale value is handled by the model initializer, which mirroring DARTS is a 3x3 convolution followed by a batch-normalization.

3. **Nodes**: The number of nodes within each cell, including the output node but not including the two input nodes. Therefore, a cell with **Nodes=***n* has $n-1$ intermediate nodes, 1 output node, and 2 input nodes.

4. **Depth**: The number of subsequent nodes each node connects to. For example, if depth is 3, a node n only sends output to nodes $n+1$, $n+2$, and $n+3$. This acts in combination with **Nodes** to control the size and linearity of each cell. With **Depth=Nodes**, each node

connects to each subsequent node, the maximum possible parallel connections. If **Depth**=1, the cell is completely linear.

5. **Sections**: The cellular compositions of each model section. This determines how many cells are in each section, and which are normal or reduction cells. This is a list of strings of `N` or `R` characters, corresponding to the type of cell (**N**ormal or **R**eduction) at each position in the section.

6. **Operations**: The set of operations available to the model for selection into the architecture.

4.8 Model Design Categories

The above architectural hyperparameters lend themselves to a wide variety of possible BonsaiNet models. To simplify the reporting of their results, models with similar hyperparameter configurations can be classified into categories, typically according to their cell count and cell size. The most common and/or interesting categories of hyperparameter configuration used in CIFAR-10 experiments are described in this section, while the results of those experiments are discussed in the subsequent section.

4.8.1 Small Cell

The first configuration is designed to mirror DARTS as closely as possible within the BonsaiNet search space. All such small-cell models are trained on an NVidia 1080TI GPU just as DARTS was. While the internal operational configuration of DARTS cannot be exactly mirrored by BonsaiNet due the usage of summation instead of concatenation in the output node (refer to Section 4.2 for the rationale behind this decision), the node count and depth is chosen to ensure the general connectivity and parameter count is roughly similar. The unique characteristic of the small cell configuration is the titular small cells, which individually make up a small proportion (<1/8th) of the total model size. Other than Xie et al.'s randomly-wired models from Section 3.8, the majority of all cellular CIFAR-10 models in the literature belong to this small-cell classification.

For BonsaiNet, small-cell models have 8 cells each with 4 nodes, with reduction cells placed at 1/3 and 2/3 depth of the model. This is identical to DARTS (Liu et al., 2018), at least in terms of cellular configuration. The way that these cells are allocated into model sections vary, with some building each cell as a separate section, and others building models as two sections of 3 cells and one section of 2. The most common hyperparameter configuration of small cell models is fully detailed in Appendix B, section B.1.

4.8.2 Large Cell

While the small-cell pattern is ubiquitous in NAS literature, the need to specify the exact cellular structure in terms of normal and reduction cells always felt like a significant limitation of the

search space. It also introduces a massive amount of human design into the search process; the need to specify which cells are which and how many of each is an architectural decision. If the point of neural architecture search is to eliminate the need to spend time and money manually tweaking model design, requiring that the user manually specify cellular configuration is simply shifting the design burden elsewhere. Additionally, having many small individual cells to search for can slow down the search process, especially if each cells needs to be searched for individually.

To eliminate this need, a large-cell design can be used where the repeating cell patterns are replaced by a single large cell. Specifically, any repeating number of normal cells can be replaced by a larger normal cell, and a reduction cell followed by any number of normal cells can be replaced by a larger reduction cell. This latter case is true because reduction cells only differ from subsequent normal cells in the edges that directly connect to the cell inputs, while all of the other edges in a reduction cell perform identical operations as an identically-dimensioned normal cell would. This reduces the possible variability of the model, as the only real variable now is the total number of reductions. This is usually relatively constant for any particular dataset, usually two for CIFAR and ImageNet such as in Liu et al. (2018); Xie et al. (2020); Xu et al. (2020) or Pham et al. (2018). Following the cell combination rules outlined above, the small cell pattern can be transformed into a large cell pattern:

Small Cell:	NN	RNN	RNN
Large Cell:	N	R	R

To ensure that there are a similar number of operations in the large-cell models, an analogue of Equation 3.5 can be used to determine the node count:

$$\sum_{i=0}^{n'+1} \min(n'-i+1, d') \geq C \sum_{i=0}^{n+1} \min(n-i+1, d) \tag{4.33}$$

Since $C = 2$, $n = 4$, and $d = 4$ for the cell group in the small-cell configuration while $C = 3$, $n = 4$, and $d = 4$ for the next two, Equation 4.33 states that the large-cell should have 28 edges in the first large cell and 42 in the next two. Choosing n' and d' as Cn and n respectively approximates these values well, while ensuring that the linearity of the large-cell model is relatively the same as that of the small-cell. This results in a 3 cell model, with the first cell having 8 nodes and a depth of 4, while the remaining cells have 12 nodes and a depth of 4. The advantage of these models is that they have seven fewer patterns than the small cell models, which means there are simply less cycles of growing and pruning necessary before the model reaches its full size. This reduces the total search time of these models drastically, typically to around two and a half hours total (see Table 4.8).

4.8.3 Type-2 Large Cell

The DARTS-based configuration that inspired small-cell and initial large-cell models focuses heavily on parameter efficiency. While this a worthwhile target in its own right, additionally

exploring configurations that maximized classification performance while still fitting on a 1080Ti was of particular interest. Doing this entails creating cells that were larger not just in node count, but also in scale; that is, the number of channels per operation within the model. This has been shown in various papers such as ResNet (Szegedy et al., 2014) to be an effective means of increasing performance, and thus a configuration that increased the scale of models from 36 to 64 was chosen for the Type-2 large cell. To compensate for the increased model size that the increased channel size brings, the cell node counts were fixed at 7 each, and the depth set to 3. Otherwise, the configuration was identical to that of the initial large cells. This configuration is dubbed the Type-2 Large Cell as described in Appendix B.3,

4.8.4 3090 Large Cell

While all previous models were designed to be searched for and trained on the available 11 GB of VRAM of an NVidia 1080TI, I also had access to an NVidia 3090 with 24 GB of VRAM. To utilize this space, the Type-2 Large Cell configuration was expanded by increasing the scale and batch size to 96.

4.9 CIFAR-10 NAS Experiments and Results

In this section, the training performances of the various cellular configurations are reported. Additionally, the design and results of various ablation studies are detailed. Finally, a detailed comparison of these results against NAS literature is shown in Table 4.17.

Regardless of architectural configuration, all models are trained with a cosine-annealed learning rate with $\eta_{max} = 0.1$ and $\eta_{min} = 0$ over 600 epochs (see Equation A.1). Models trained over exactly identical configurations and implementations have highly consistent performance, usually all within ± 0.1 percentage points of each other (see the sets of identical Bonsai models trained within Geada et al. (2020) for examples of this).

4.9.1 Small Cell Results

Nine small cell models have been produced by BonsaiNet according to identical architectural hyperparameter choices, the performance of which are are shown in Table 4.7. Among these nine small-cell models are the ones described in "Bonsai-Net: One-Shot Neural Architecture Search via Differentiable Pruners"(Geada et al., 2020), published in CVPR-NAS. In this paper, the consistency of BonsaiNet is demonstrated by training four such small cell models under identical conditions, and recovered a mean accuracy of $96.65 \pm 0.06\%$, mean parameter count of 2.95 ± 0.11M, mean memory allocation of $7.21 \pm .14$ GB, and mean search time of 14.4 GPU-hours.

In general, the small-cell configuration produces decently performing models in a relatively quick time, with their main advantage coming from their highly efficient parameter count. For a full comparison against the literature, see Table 4.17. However, neither their search time or

Run	Test Accuracy	Parameters (M)	Search GPU-Hours	Total GPU-Hours
Lowest Acc.	96.05	2.50	10.6	80.93
Median Acc.	96.65	3.65	11.50	92.95
Highest Acc.	96.83	5.00	42.28	149.48

Table 4.7 The performance metrics of the lowest, median, and highest accuracy small cell CIFAR-10 BonsaiNet models within the nine total. Search hours refers to the GPU-hours required to grow the model to full size, while total runtime is this search time plus the GPU-hours required to train the model fully.

accuracy seemed to indicate the maximal performance of BonsaiNet, which thus motivated the design of the large cell experiments.

4.9.2 Large Cell Results

Table 4.8 shows the results of the three models of this specific configuration that were trained, and the configuration is given in Appendix B.2.

Run	Test Accuracy	Parameters (M)	Search GPU-Hours	Total GPU-Hours
1	96.69	2.64	2.53	68.53
2	96.46	2.22	2.38	88.06
3	96.72	3.18	1.72	82.31

Table 4.8 Various performance metrics of the three large cell CIFAR-10 BonsaiNet models that trained to the full 600 epochs.

The accuracy of these models is similar to the small-cell models. However, these models are significantly faster to search for, taking on the order of two hours, and tend to have around one million fewer parameters than the median small cell model. Since these large-cell models are essentially supersets of the small-cell models, they can form small-cell analogues if that is found to be beneficial, but have the freedom to explore connectivity patterns that smaller cells simply do not have the nodes to allow for. The large-cell model's relatively comparable performance at higher parameter efficiency perhaps suggests the restrictiveness of the small-cell configuration is artificially lowering the efficiency of such small-cell models.

4.9.3 Type-2 Large Cell Results

The performances of the three models of this configuration are shown in Table 4.8.

Here, notice that the accuracy has markedly increased compared to the small-cell and type-1 large cells, with the *minimum* type-2 accuracy outpacing the *median* accuracy of the other configurations. This accuracy increase comes at the expense of parameter efficiency, however the runtime is roughly similar.

Run	Test Accuracy	Parameters (M)	Search GPU-Hours	Total GPU-Hours
1	96.82	5.62	1.43	102.63
2	96.96	5.81	1.66	87.09
3	97.04	7.39	2.54	81.91

Table 4.9 Various performance metrics of the three type-2 large cell CIFAR-10 BonsaiNet models that trained to the full 600 epochs.

4.9.4 3090 Large Cell Results

Only one 3090 Large Cell model has been run, and its performance is shown in Table 4.10.

Run	Test Accuracy	Parameters (M)	Search GPU-Hours	Total GPU-Hours
1	97.07	23.36	1.66	75.59

Table 4.10 The performance of the single 3090 large cell CIFAR-10 BonsaiNet model.

While the increased channel size dramatically increased the model's parameters and VRAM requirements, it also brought some marginal performance gains. The increased batch size also contributed to the VRAM size, but significantly reduced the search and training time of the model. Only one run was performed of this, simply due to limited access to the hardware and the rather lackluster improvements shown by this configuration.

4.9.5 Evaluating the Supernet

In addition to the 3090 large cell models, a number of supernet experiments models were conducted on an NVidia 3090. The 24 GB of VRAM space allows for the 1080Ti type-2 large cell supernet to fit entirely unpruned into memory, meaning the Bonsai process is not actually necessary for that configuration on the 3090. Therefore, the efficacy of the BonsaiNet subnets can be evaluated compared to both the unpruned supernet as well as a freely pruned supernet, where the former is purely the supernet of all possible operations trained without the ability to prune, while the latter is the same supernet that can prune throughout training with $\lambda = 0$, i.e, pruning purely to benefit accuracy. The results of this comparison tell us a few crucial pieces of information; first, how effective is pruning at removing connections from a model in a minimally detrimental way, and second, how effective is the Bonsai process' growing and pruning method of building subnets compared to the free pruning of the supernet?

To conduct this experiment, four models of equal configuration are evaluated: an unpruned supernet, a freely-pruning supernet, and two BonsaiNet runs constrained to 10.5 GB of VRAM. The no-pruning and free-pruning require 14.92 GB of VRAM at first initialization, and thus can only be trained on high-VRAM GPUs like the 3090. Meanwhile, the BonsaiNet runs operate within a fixed VRAM footprint, in this case specified at 10.5 GB to match the maximum available

VRAM of the much cheaper 1080Ti GPU[4]. Two such BonsaiNet runs are performed, one on the 3090 and one on the 1080Ti. The former two runs evaluate BonsaiNet's ability to operate within a limited VRAM footprint, either one artificially limited as in the case of the 3090 or physically limited in the case of the 1080Ti.

		VRAM (GB)				Total
Run	Test Acc.	Final	Max Required	Params. (M)	c	GPU-Hours
Unpruned	97.08	14.92	14.92	12.04	1.0	75.6
Free Pruning	**97.19**	10.18	14.92	6.63	0.67	60.2
BonsaiNet 3090	96.99	7.85	**10.5**	**6.20**	0.55	**42.0**
BonsaiNet 1080Ti	97.04	**7.58**	**10.5**	7.39	0.48	81.9

Table 4.11 Four type-2 large cell models, with varying methods of determining the final architecture. All four models are trained identically, with the only difference being the BonsaiNet 1080Ti model is trained on a 1080Ti GPU while the rest are trained on a 3090.

These results reveal a few very interesting details. First, and the most surprising, is that not only is the pruned supernet significantly smaller and faster than the unpruned supernet, it is actually much better performing as well. This is the opposite result to what was expected; I had assumed that the unpruned supernet would be the best model in the space simply due to its size and flexibility. However, even with no compression term in the loss function, i.e, a λ value of 0, the freely pruning model pruned down to a compression level of 67%. One third of the supernet's operational allocation was deemed detrimental by the freely pruning model, which indicates that pruning is not purely a space-conserving operation; it provides loss benefits as well. This is seen in the free-pruned model's accuracy performance, which is a significant improvement over the unpruned model. This is surprising, as differentiable pruning had been chosen in the hopes that it was minimally detrimental to performance, but instead it appears to be of substantial accuracy benefit. This likely is because the smaller and optimized pruned supernet can train faster due to having a smaller parameter count, and thus can reach higher accuracy within the fixed training epoch count,

Second, while the Bonsai process is less effective than purely pruning the raw supernet, it manages to find a minimally detrimental subnet of the raw supernet. On average, the two Bonsai models were only 0.07 percentage points less accurate than the raw subnet, but used 1.4× smaller VRAM footprint, 1.7× fewer parameters, and produced models with 1.9× smaller final VRAM size. These latter three figures demonstrate BonsaiNet's spatial and parameter efficiency which is likely its strongest attribute; BonsaiNet is able to identify a highly efficient subnet of a larger supernet that fits within a target VRAM, despite that supernet being much larger than that target VRAM size. This means that in cases where the target VRAM is equivalent to the maximum available, BonsaiNet finds excellent subnets from a supernet that would otherwise be impossible to construct. In the 3090 case, wherein the supernet would be constructable within available VRAM space, BonsaiNet instead differentiates itself in the speed by which it produces these

[4]At time of writing (March 1st, 2022), used 1080Tis can be found on Ebay for £450 ($650), while used 3090s are listed at £1700 ($2276).

extremely efficient models. On the 3090, the joint Bonsai search and training process took just 42 hours compared to the 75.6 of the raw supernet or 60.2 of the freely-pruning model.

4.10 Random Search

While these results thus far indicate that BonsaiNet can consistently produce highly performant and efficient models, these need to be compared against random search in order to decouple the quality of BonsaiNet as a search algorithm versus the quality of its search space, as urged in Li and Talwalkar (2019) and Yu et al. (2019). To evaluate how well BonsaiNet compares to random search, the small-cell configuration is compared against two different random selection techniques. The rationale for having two random search techniques is due to the two-stage process by which BonsaiNet produces an architecture: first, the model is grown to its final size via the Bonsai algorithm. Second, the model trains and prunes for the remaining training epochs. This second stage does not grow the model but does remove a significant number of internal connections, and thus does modify the architecture in a non-trivial manner. As such, the two levels of random search are designed to test the effects of replacing one or both stages of architecture search with a random search instead. See Table 4.12 to see which combinations of stages are used by the two random methods:

	Train w/ Prune	Train w/o Prune
Bonsai Growth	Normal	-
Random Growth	Random 1	Random 2

Table 4.12 The comparative function served by each of the random search techniques.

Both algorithms are designed to emulate some target Bonsai model $M^{[0,1,\ldots,n]}_{[c_0,c_1,\ldots,c_n]}$, that is, a model with n sections where section 0 is compressed to c_0, section 1 to c_1, etc. This Bonsai model was produced via the two stages of architecture search, where stage 1 produced a model $M^{[0,1\ldots,n]}_{[c_0,c_1,\ldots,1]}$. This model then underwent subsequent pruning during stage 2, which compressed that model into $M^{[0,1,\ldots,n]}_{[c'_0,c'_1,\ldots,c'_n]}$, where $c'_0 \leq c_0$, $c'_1 \leq c_1$, and so on. From this observation, the two random algorithms can be presented:

Random 1 emulates the output of stage 1, generating some random model $R^{[0,1,\ldots,n]}_{[c_0,c_1,\ldots,1]}$ where operations are chosen at random to meet those compression targets. This model is then trained *with* pruning (where the pruning is performed freely, i.e, $\lambda = 0$) according to identical hyperparameters as used in the training of the target Bonsai model M. This is random *growth* of a model, followed by guided pruning afterwards. This algorithm is presented in full in Appendix A, Algorithm 7.

Random 2 emulates the output of stage 2, generating some random model $R^{[0,1,\ldots,n]}_{[c_0,c_1,\ldots,c_n]}$. This model is trained *without* pruning according to identical hyperparameters as used in the training of the target Bonsai model M. This is equivalent to the random search in Yu et al. (2019) and Li

and Talwalkar (2019), that is, a purely random generation of an architecture. This algorithm is presented in full in Appendix A, Algorithm 8.

By comparing the accuracies and parameter counts of models produced by the two random search algorithms against Bonsai models, both the efficacy of the Bonsai algorithm in producing full size models and the performance impact of free-pruning during training can be evaluated respectively. Each of the three techniques (Bonsai, Random 1, and Random 2) are evaluated three times each, for a total of nine runs. The average performance and parameter count of these nine runs are noted in Table 4.13.

	Test Accuracy	Parameters (M)
BonsaiNet	$\mathbf{96.65 \pm 0.06}$	$\mathbf{2.95 \pm 0.05}$
Random 1	95.27 ± 0.05	3.89 ± 0.12
Random 2	95.19 ± 0.13	3.03 ± 0.01

Table 4.13 The small cell configuration compared to the two levels of random search. Error bounds computed as the standard deviation divided by the square root of number of samples.

Notably, BonsaiNet produces both the best and smallest models compared to random search within the small-cell configuration search space. If it is assumed that the random models are indicative of the quality of the "average" model, these results are indicative of BonsaiNet's ability to consistently discern better-than-average models within the search space, at the very least. Additionally, Random 1 outperforms Random 2, which indicates that a model that starts at some larger size c and compresses to c' has better performance than a model that started at c' in the first place.

These results can also be used to test one of the original intents of BonsaiNet; to evaluate whether NAS models perform poorly compared to random search simply due to the high average quality of model in their space, by significantly derestricting said search space. In this regard, a simple comparison of a random model from the BonsaiNet small-cell search space versus the random models from the DARTS search space that are explored in Yu et al. (2019) is sufficient, and this comparison is in Table 4.14.

Search Space	Method	Test Accuracy
DARTS	DARTS	96.62 ± 0.23
	NAO	96.86 ± 0.17
	ENAS	96.76 ± 0.10
	BayesNAS	95.99 ± 0.25
	Random	$\mathbf{96.48 \pm 0.18}$
Bonsai	BonsaiNet	96.65 ± 0.06
	Random 2	$\mathbf{95.19 \pm 0.13}$

Table 4.14 Random search compared to NAS methods across the Yu et al. and Bonsai search spaces. DARTS results taken from Yu et al. (2019). Error bounds computed as the standard deviation divided by the square root of number of samples.

From this, it is seen that the average NAS model in the DARTS search space is roughly similar in performance to the average random model in this space, with the average mean accuracy of the four NAS methods coming to 96.56 ± 0.18 compared to random's 96.48 ± 0.18. However, in the Bonsai space random models are markedly worse on average than those of the DARTS space, with an average accuracy worse by 1.29 percentage points. Despite this, BonsaiNet produces roughly equivalently performing models to the DARTS search space NAS models. This is not to claim BonsaiNet's superiority over the other NAS methods, but rather to argue that their unfavorable comparison to random search is indeed due to a simple overconstriction of their search spaces. These random search experiments over the Bonsai space show that by deconstricting the search space, the difference between intelligent and random architecture selection becomes much more apparent and the advantage of NAS becomes clear.

4.11 ImageNet

	Top-1	Top-5	Params (M)	Search Hours	Total Hours
PNAS	74.2%	91.9%	**5.1**	2,500	-
NAS-Net-A	74.0%	91.6%	5.3	48,000	-
NAS-Net-B	72.8%	91.3%	5.3	48,000	-
Regularized Evolution	**82.8%**	**96.1%**	86.7	75,600	-
NAO (w/o weight sharing)	74.3%	91.8%	11.35	4,800	-
(all models above this line are still searching by the time BonsaiNet has produced a fully-trained model)					
SMASHv2	61.38%	83.67%	16.2	36	-
DARTS Second-order	73.3%	91.3%	4.7	96	-
ProxylessNAS	74.6%	92.2%	N/A	200	-
PC-DARTS	75.8%	92.7%	5.3	91.2	-
BonsaiNet	60.46%	82.05%	8.8	**10.52**	418.57

Table 4.15 BonsaiNet performance on ImageNet, as compared to the literature.

BonsaiNet was also evaluated over the ImageNet-1K dataset, following the training and data processing procedures of DARTS and using the CIFAR-10 ImageNet Large Cell configuration. While searching very efficiently, BonsaiNet was relatively uncompetitive in final accuracy. This could be for a number of reasons, for example node count, depth, or deadhead settings. Due to the immense computational cost of training ImageNet models, only one single run was performed, and therefore tuning of the algorithm was not possible.

4.12 NASComp

BonsaiNet was additionally evaluated over the NASComp-2022 datasets (see Appendix D for details about these datasets). To fairly compare BonsaiNet against the competition entries, the evaluations were performed such as to conform to the competition ruleset as closely as possible. To this end, only 12 hours of combined search and training time were allocated to each dataset,

and the per-dataset configurations were adapted from the 3090 Large-Cell configuration. These configuration modifications were chosen via a globally-applicable heuristic, such as to not bias design choices unfairly through personal knowledge of dataset idiosyncrasies.

- **GPU Space**: 10, the maximum VRAM space of an NVidia 1080Ti
- **Scale**: c, such that $64 \times c \times h_{dataset} \times w_{dataset} = 2,097,152 = 64 \times 32 \times 32 \times 32$, that is, of equivalent dimensional product to a 32 channel CIFAR-10 model.
- **Nodes**: 7+n, n increased until all compression targets within $[0.5, 0.8]$
- **Depth**: 3+n, n increased until all compression targets within $[0.5, 0.8]$
- **Sections**: [[N], [R], [R]]
- **Operations**: Identity, 3x3 Max Pool, 3x3 Average Pool, 3x3 Separable Convolution, 5x5 Separable Convolution, 3x3 Dilated Convolution, 5x5 Dilated Convolution

Furthermore, the BonsaiNet runs did not use any data augmentation policies, as personal knowledge of the datasets would unfairly advantage the selection of beneficial or feasible augmentation procedures. The training procedure was also identical to that of the CIFAR-10 runs, with the only difference being the epoch count was chosen such as to roughly fill the remaining time allocation after search was completed.

The chief evaluation difference is that the BonsaiNet runs were performed on a 1080Ti, while competition evaluations occurred on an NVidia V100 GPU via an Azure NC6sV3 instance (Microsoft Azure, 2021). However, this difference should only disadvantage BonsaiNet,as the 1080Ti has 6 GiB less VRAM and is roughly 1.72 times slower than the V100 according to a Lambda Labs (2018) benchmark.

Table 4.16 details the results of the BonsaiNet runs over the NASComp-2022 datasets. BonsaiNet would have finished first overall, despite using no data augmentation as compared to the significant, searched-for augmentation policies of the actual competition winner. While the

	Dataset			
Algorithm	Sadie	Chester	Isabella	Total Score
ResNet-18	80.33%	57.83%	62.02%	200.18
NASComp-2022 Winner	**96.08%**	**62.98%**	61.42%	220.48
BonsaiNet	95.49%	61.32%	**64.92%**	**221.73**

Table 4.16 BonsaiNet performance on NASComp-2022 datasets.

results need to be taken with a grain of salt in that they were perhaps performed with an unfair advantage due to design biases induced by dataset experience, they do show BonsaiNet's ability and flexibility to produce good networks given semi-arbitrary configurations over novel datasets.

4.13 Model Analysis

With BonsaiNet returning consistently well-performing models, as well as ones that are consistently better than random search, looking at the actual architectures it is generating might

uncover some interesting insights. Ideally, this consistent performance above random search is indicative of intelligent architecture design, which should appear as repeating patterns within found architectures. To investigate this potentiality, the architectures of 12 fully-trained Bonsai models of various small and large-cell CIFAR-10 configurations are analyzed. Then, of the 1062 total edges across these 12 models, the operation occurrence frequency and operation co-occurrence frequency can be measured; that is, the total fraction of edges that contain a certain operation and the total fraction of edges that contain a certain pair of operations. If there are global patterns in this data, i.e, consistent behavior across the 12 models, then it is indicative of intelligent, or at least deliberate, design choices.

4.13.1 Operation Selection Frequencies

A first line of analysis is the operation occurrence frequency: the fraction of the edges within a model that contained a certain operation of the seven available in the small and large-cell configurations. These frequencies are shown in Figure 4.13.

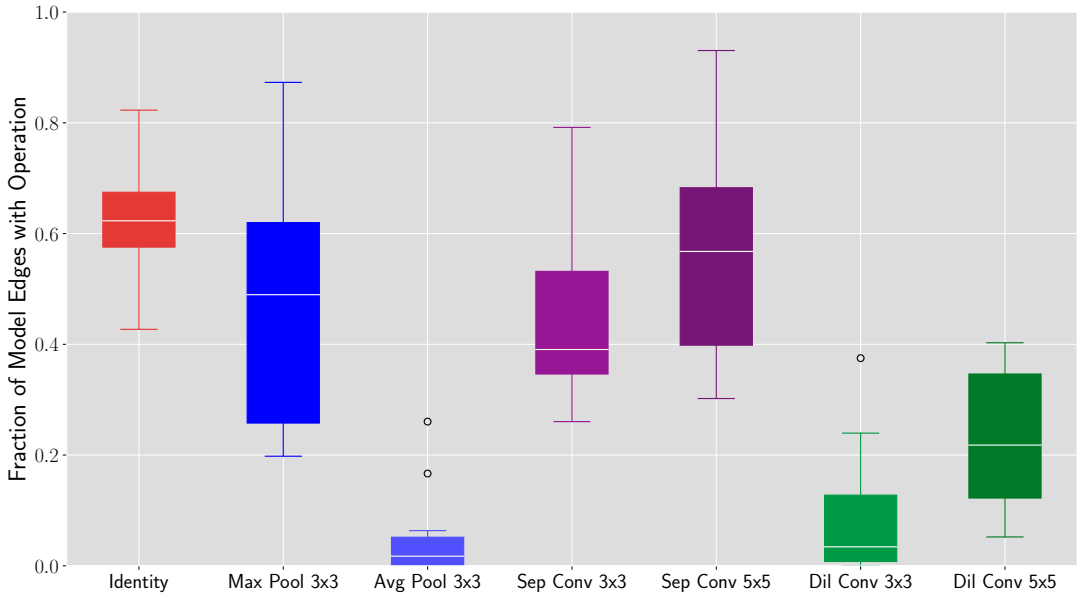


Fig. 4.13 Fraction of edges within a single model that contain the given operation, averaged over 12 different CIFAR-10 Bonsai models of various configurations. This indicates clear operational preferences, with dilated convolutions and average poolings the least favored.

Notably, average poolings only appear in 1.7% of edges in the median model; this corresponds to about once per model. 3x3 dilated convolutions are similarly unpopular, showing up in only 3.4% of the edges in the median model. On the other hand, identities, 3x3 max poolings and separable convolutions are highly preferred, with one model selecting 5x5 separable convolutions in as many as 93% of its edges. One way to explain this consistency across all 12 models is that it could be due to simply the compression loss weightings of each operation by size, that the preferred operations are smaller in VRAM size and thus their presence is less heavily penalized in the loss function. Conversely, it would also be expected that the least popular operations would be the largest and most heavily penalized. However, as Table 4.3 shows, this is not the

case; average poolings are the second smallest operation in the space, while dilated convolutions are third smallest. Despite these operations being the second and third 'easiest' to keep, they are systematically removed from the models. On the other hand, separable convolutions are the largest operation by almost twofold, yet the models are more than willing to pay this price. This is all indicative of some separate reasoning beyond pure compression loss penalties.

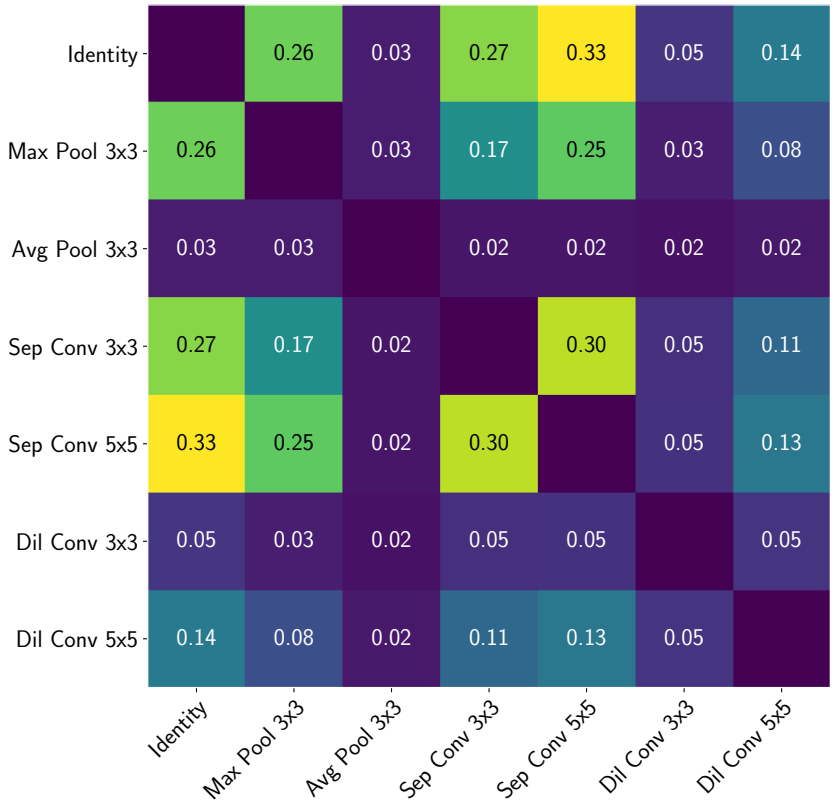


Fig. 4.14 Frequency of operation co-occurrence within an edge, aggregated over 12 different CIFAR-10 Bonsai models of various configurations.The most common pairing is [Identity, 5x5 Separable Convolution], which occurs in 355 of 1062 edges or 33% of the time.

Next is operation co-occurrence; how often operations appear together in the same edge. Here, all 1062 edges within the 12 examined models are examined, and measure the frequency by which any pair of operations appear together within those 1062 edges. This is presented in Figure 4.14. Interestingly, the first and third most common operation co-occurrences are [Identity-5x5 Separable Convolution] and [Identity=3x3 Separable Convolution] respectively. Such an edge constituting of either pair of operations exactly matches the connectivity pattern of ResNet models, that is, parallel identity and separable convolutions that are subsequently summed together. The optimistic perspective on this outcome is that BonsaiNet organically discovered residual connections on its own, isolating those two particular patterns as its favorites among the 49 possible operation pairs. This would indicate that the design intuitions it learns closely aligns with those that human network designers have discovered, which supports that BonsaiNet designs models intelligently and deliberately, as well as demonstrates the strengths of human design patterns. This reasoning has a natural next step, which is that if some of BonsaiNet's behaviors align with the pinnacle of human design, perhaps BonsaiNet's other

design proclivities align with as-of-yet undiscovered best practices; there is potentially a wealth of design knowledge that can be gained by studying these models.

More pessimistically, it could be theorized that the implicit parallel design of the search space heavily biases the discovery of ResNet-esque patterns, and that Identity-Separable Convolution pairs are simply the most viable option given that bias. However, this still implies that BonsaiNet can identify good operation pairings from poor ones, which is still a meaningful benchmark to achieve.

4.14 Comparisons and Conclusions

	Test Acc.	Params (M)	Search Hours
NAS-Net	95.53%	7.1	37,755
ENAS Micro-search	96.131%	38.0	7.7
ENAS Macro-search	97.11%	4.6	14.4
DARTS First-order	$97.00 \pm 0.14\%$	3.3	36
DARTS Second-order	$97.24 \pm 0.09\%$	3.3	96
NAO (w/o weight sharing)	**98.07%**	144.6	4,800
NAO (w/ weight sharing)	97.07%	**2.5**	7
ProxylessNAS	97.92%	5.7	N/A
ENAS/DARTS/NAO Random	$96.48 \pm 0.18\%$	-	-
PC-DARTS	97.43%	3.6	2.4
Bonsai SC	96.83%	3.38	11.50
Random 1 SC	95.32%	4.54	–
Random 2 SC	95.19%	3.68	–
Bonsai LC1	96.69%	2.64	2.53
Bonsai LC2	97.04%	7.39	2.54
Bonsai LC3090	97.07%	23.36	**1.66**
Bonsai LC2 (no prune, no growth)	97.08%	12.04	0
Bonsai LC2 (prune, no growth)	97.17%	6.63	0

Table 4.17 CIFAR-10 statistics for all of the NAS models covered thus far. Total runtime is not consistently reported in literature, and thus is not shown here, but in theory should be relatively consistent between similarly sized models.

When compared against other leading NAS algorithms (see Table 4.17), some strengths of BonsaiNet become apparent. Notably, it performs search extremely rapidly, identifying very competitive models within three hours. An important comparison here is that of DARTS, which uses an identical GPU specification as the non-3090 BonsaiNet runs do. For example, the best Bonsai LC2 model outperformed the DARTS First-order model, while requiring 14 times fewer search hours. The greatest advantage of BonsaiNet however is its robustness and flexibility, in that it provides consistent performance over various different configurations. Depending on the particular configuration, BonsaiNet models can be optimized for whichever target is most important to the particular use-case. If desired, BonsaiNet models can aim for raw, top-end classification performance to maximize the potential performance within a certain VRAM

requirement. Alternatively, BonsaiNet models can be designed to be highly parameter, FLOP, and time efficient, and can find optimal architectures within those constraints. As such, BonsaiNet has the potential to be a powerful and versatile tool for use in a broad variety of deep learning contexts.

4.15 Future Work

The main area of remaining research for BonsaiNet is further comparisons against random search in the various large cell configurations, as well as evaluating its robustness across more diverse cellular configurations. The former is relatively straightforward to perform; simply perform level 1 and 2 random searches in the large cell configurations at identical compression levels as used by BonsaiNet. The latter is again relatively straightforward, simply involving testing many different configurations for performance. The difficulty here is picking meaningful comparisons to make with regards to configuration performance; the small-cell configuration draws direct parallels to DARTS which offers obvious comparisons, but appropriate comparisons in the literature for more idiosyncratic configurations will be scarcer. One possible approach is to unify both points; arbitrarily select a configuration and train a BonsaiNet model with it, then compare that to random search. This provides an idea of the robustness of BonsaiNet in these various configurations, by showing that it does or does not consistently outperform random search. However, each such test of either BonsaiNet or random search takes around three or four days, which means such an experiment is very time-consuming and thus had to be deprioritized versus other experiments.

Another potential avenue of exploration is to try and design an automatic configurator for BonsaiNet, some wrapper that given a dataset and task chooses an optimal configuration by which to run BonsaiNet. Development of such a tool would be benefited by the above robustness work, as data from that experiment could inform how configurational choices influence Bonsai's performance for better or worse.

Finally, exploring the efficacy of Bonsai's growth and pruning in an unconstrained search space is an interesting avenue to explore. That is to say, a scenario wherein the new sections added to Bonsai were not predetermined, and instead edges were added dynamically according to the model's specific needs. This question led to the development of SpiderNet, an extension of BonsaiNet that grows dynamically edge by edge as new space is created via pruning.

Chapter 5

SpiderNet

5.1 Introduction

While BonsaiNet reliably produces good models within a given configuration, its search flexibility is implicitly limited by its configuration, specifically by the need to specify the per-cell node count and depth. While the operations between these specified nodes can vary, the actual edge structure of any cell is fixed. Therefore, there is a significant measure of human design and intuition necessary to choose ideal node and depth parameters, or barring that, a lot of brute force experimentation. This feels rather counterintuitive to the general point of neural architecture search, which strives to eliminate the need for finicky intuition or hyperparameter grid searches. Arguably, these algorithms that rely on user configuration of the final structural connectivity are just moving the burden of search one level higher, replacing architecture search with *architecture-search* search. To this end, what is the point of reporting that a NAS algorithm can produce an amazing architecture in a few GPU-hours, if it takes a dozen tries to figure out how to configure said algorithm?

With that question in mind, it is of interest to explore models that approached the model growth problem from the opposite direction; rather than specify the structure of the initial supernet and use Bonsai to find some optimal subnet, the goal was to start with some minimum viable model and allow the model to dynamically expand the search space as it saw fit. By letting the model choose its own overall structure as well as the specifics of its connectivity, in theory the flexibility and ease-of-applicability of NAS could be greatly improved.

In this chapter, Section 5.2 will first cover some necessary background material. Section 5.3 will next cover the components and general design of SpiderNet, an algorithm that meets these specifications. Next, Section 5.4 will cover the neural network theory that is used to calculate SpiderNet's operational heuristics. With design and theory covered, the full SpiderNet algorithm is stated in Section 5.5. Various experimental designs over this algorithm are described in Section 5.6, the results of which are discussed in Sections 5.7 and 5.8. Finally, Section 5.11 explores some open questions around SpiderNet, before wrapping up in Section 5.12.

5.2 Background

To understand certain specific components of SpiderNet, two concepts must first be discussed: SHAP and train-free model metrics.

5.2.1 SHAP

SHAP or SHapley Additive exPlanations was introduced by Lundberg and Lee (2017) as a means of adapting game theory's Shapley value into an explainable AI context. Shapley values are a measure of an individual's contribution to a coalition, where a coalition is a group of cooperating members and the value of a coalition C is defined as some function F with respect to its members. The Shapley value of an individual is calculated by computing the mean difference in coalition value between each pair of coalition permutations that differ only by the inclusion of that particular individual:

$$\phi_i = \frac{1}{\text{\# coalition members}} \sum_{\text{all coalitions } C \text{ excluding i}} \frac{F(C \bigcup i) - F(C)}{\text{\# coalitions excluding } i \text{ of size } |C|}, \tag{5.1}$$

where $F(C \bigcup i)$ is the coalition value of the coalition formed by adding member i to coalition C, "all coalitions C excluding i" refers to every possible permutation of the available coalition members that excludes member i, and "# coalitions excluding i of size $|C|$" refers to the total number of coalition permutations that exclude member i but have an equal number of members as C.

For example, given some coalition that consists of three members Isabelle, James, and Kate, Isabelle's Shapley value ϕ_I can be calculated by first enumerating the following values for each of the four coalitions C that exclude Isabelle:

Coalition C	$F(C \bigcup i)$	$F(C)$	Coalitions excluding i of size $\|C\|$
(J,K)	F(I, J, K)	F(J, K)	(J, K)
(J)	F(I, J)	F(J)	(J), (K)
(K)	F(I, K)	F(K)	(J), (K)
$(\varnothing)$	F(I)	$F(\varnothing)$	$(\varnothing)$

Therefore:

$$\phi_{\text{Isabelle}} = \frac{1}{4}\left(\frac{F(I,J,K) - F(J,K)}{1} + \frac{F(I,J) - F(J)}{2} + \frac{F(I,K) - F(K)}{2} + \frac{F(I) - F(\varnothing)}{1}\right)$$

The Shapley values can then be used to model coalition value additively, such that:

$$F(C) = F(\varnothing) + \sum_{i=1}^{N} \phi_i x_i \tag{5.2}$$

where x_i is a binary value that indicates the presence or absence of the ith member of the coalition C. Under this formulation, ϕ_i gives the exact contribution that member i had on the coalition value $F(C)$.

To adapt this concept from game theory to explainable AI, Lundberg and Lee (2017) uses a model's input features as the coalition and the model's output as a coalition value, and therefore the Shapley value ϕ_i would model how much a feature i affected the model's output value. To avoid needing to sample massive numbers of coalition pairs when explaining many-feature models, Lundberg and Lee devise a sampling algorithm that uses a weighted linear regression to recover the Shapley values of features with exponentially fewer samples than raw computation of Shapley values would require. Additionally, they create a formulation to mathematically approximate the effects of removing input features without needing to reformulate models that may not allow for variable-length inputs. From this, SHAP can provide an easily interpretable explanation as to how the inputs $\vec{x}$ to a model M affected $M(\vec{x})$, simply by looking at $\phi_0, \ldots, \phi_n$.

5.2.2 Train-Free Metrics

A train-free metric refers to methods that evaluate a model's quality via sampling methods that involve no backpropagation or weight sampling, ideally in such a way that the metric correlates strongly with final model performance. Two model characteristics that train-free metrics commonly seek to quantify are *trainability* and *expressability*.

Trainability refers to the effectiveness of gradient descent at optimizing a model. This factor is one of the reasons for performance differences between models of vastly different parameter counts; for example, VGGNet-E has 144M parameters and scores a 74.5% top-1 test accuracy on ImageNet, while ResNet-152 has less than half at 60.1M yet scores a 80.62% (He et al., 2015; Simonyan and Zisserman, 2015). Trainability lets us explain this disparity; the ResNet is more trainable than the VGGNet, which lets it use its parameters more effectively and efficiently which more than makes up for its lower parameter count. Expressability, on the other hand, refers to the complexity of function that the model can approximate. Expressability is the culprit behind the bigger-is-better phenomena seen in ResNets or VGGNets; with more parameters, the bigger models can approximate more complex functions (and therefore have higher expressability) and thus tend to perform better. Of course, expressability is more complex than just parameter count, as a model's operations and their interactions are very important to the nature of the functions the model can represent. These two measures are best used in combination, as the best models will have both high trainability and expressability; that is, they easily reach their full potential and that full potential is powerful.

One such paper that applies train-free metrics to a NAS context is Chen et al. (2021), titled "Neural Architecture Search on ImageNet in Four GPU Hours: A Theoretically Inspired Perspective". This paper explores the use of two train-free metrics, the neural tangent kernel and linear region count, as means trainability and expressability respectively. These are then used to differentiate between different models in a search space to rapidly speed up the search process.

5.2.3 Neural Tangent Kernel

To measure trainability, Chen et al. use Jacot et al.'s *Neural Tangent Kernel* or NTK, which is used to analytically model the training dynamics of neural networks through differential equations. The exact derivation of the NTK is complex and not relevant to the discussion here, but in summary the derivation is based around observing network training dynamics, approximating the network as a linear function through Taylor expansion and analytically representing the dynamics of that approximation's output as an ordinary differential equation (Dwaraknath, 2019). This differential equation of training dynamics ends up being a function of the pairwise inner product of the *feature maps* of a model at initialization, where feature map is defined as:

$$\phi = \nabla_\theta f(\mathbf{X}, \theta_0) \tag{5.3}$$

This ϕ is a matrix, where $\phi_{i,j}$ is the gradient of the ith parameter given the jth input datapoint and the initial state of the model parameters θ_0, thus measuring the direction and magnitude of change of each parameter in the model given the entirety of the training data $\mathbf{X}$. The pairwise inner product $\phi^T\phi$ represented by Φ therefore gives the dot product of the column vectors $\phi_{*,i}$ and $\phi_{*,j}$ for all pairs i, j:

$$\Phi_{i,j} = \phi_{i,0}^T\phi_{0,j} + \phi_{i,1}T\phi_{1,j} + \cdots + \phi_{i,p}^T\phi_{p,j} \tag{5.4}$$

$$= \phi_{0,i}\phi_{0,j} + \phi_{1,i}\phi_{1,j} + \cdots + \phi_{p,i}\phi_{p,j} \tag{5.5}$$

Therefore, the ith element of the diagonal $\Phi_{i,i}$ gives the sum of the squared gradient of each parameter in the model induced by the ith datapoint, which measures the magnitude of the training contribution of that particular datapoint between initialization and some arbitrary point in the model's future. Specifically, this product is

$$\frac{\partial f(\mathbf{X}, \theta_t)}{\partial t} = -\Phi(f(\mathbf{X}, \theta_t) - \mathbf{Y}) \tag{5.6}$$

where θ_t are the model parameters at some time t, $f(\mathbf{X}, \theta_t)$ is the model output over the training inputs given the parameter values θ_t, and $\mathbf{Y}$ are the correct training labels of the data. A situation where $f(\mathbf{X}, \theta_t) = \mathbf{Y}$ represents model convergence and perfect train accuracy, which is always the target of training. By replacing $f(\mathbf{X}, \theta_t) - \mathbf{Y}$ with u, the above differential equations can be solved as:

$$u(t) = u(0)e^{-\Phi t} \tag{5.7}$$

Since $u(t) = 0$ represents convergence and $u(0)$ is the initial state of the model, the fully-trained state of the model is attained once the initial state $u(0)$ fully decays to 0. The exponential rate of this decay is thus governed by Φ. By performing a spectral decomposition of Φ, this can be

written as:

$$u(t) = u(0)e^{-tQ\Lambda Q^{-1}} \tag{5.8}$$

$$= u(0)Qe^{-t\Lambda}Q^{-1} \tag{5.9}$$

Since Λ is a diagonal matrix of eigenvalues, it is then possible to define $e^{-t\Lambda} = \Lambda'$, where Λ' is a diagonal matrix with $\Lambda'_{i,i} = e^{-t\lambda_i}$. Therefore, the decay of the initial state into the converged state can be represented as a linear combination of exponential decays along each eigenvector, the rates of which are given by the corresponding eigenvalues λ. The thing that makes all of this so useful is that the matrix Φ is invariant with respect to time (i.e., training steps) if the actual model is relatively close to its linear Taylor-expanded representation (Dwaraknath, 2019). This means that the relative rate of training convergence can be measured using these eigenvalues, without needing to perform any training on the model whatsoever. Chen et al. choose to mark the largest eigenvalue as λ_0 and the smallest as λ_m, and use these to define $\kappa = \frac{\lambda_0}{\lambda_m}$ as their relative measure of trainability. Large κ values indicate that the model converges much more quickly along certain directions versus others, while a low κ value indicates more uniform convergence along all directions.

In practice, the measurement of NTK is highly expensive: the Φ matrix needs an equal number of rows as parameters in the model, which means its construction can be prohibitively large on any meaningfully sized model. To account for this, Chen et al. use a "slim" version of the candidate model as a proxy for the full model's NTK. This slim model only has a single channel throughout; each convolution operation only has a single kernel and all operations operate exclusively in monochrome. The idea is that the NTK of this single-sliced model will adequately approximate the NTK of the full model. However, this is never justified or even mentioned in the original paper, only from investigating the source code did I uncover this approximation.

Regardless, Chen et al. provide a proof-of-concept of their NTK measurements through analysis of the models within NAS-Bench-201 (Dong and Yang, 2020), a collection of 15,625 different neural architectures and their corresponding CIFAR-100 performances. By determining the κ value of all 15,625 models at initialization, they find a strong negative correlation with final test accuracy, with a Kendall-tau correlation of -0.42. The full results of their study are shown in Figure 5.1:

Notice that while κ ranges from around 40 to 850,000 across the entire NAS-Bench-201 collection, the best models all lie in the region closest to zero. Furthermore, there is a steep performance drop off after $\kappa = 50,000$ or so, with no subsequent model scoring above 70% test accuracy. Therefore, while low κ values do not guarantee good performance, high κ values prohibit it.

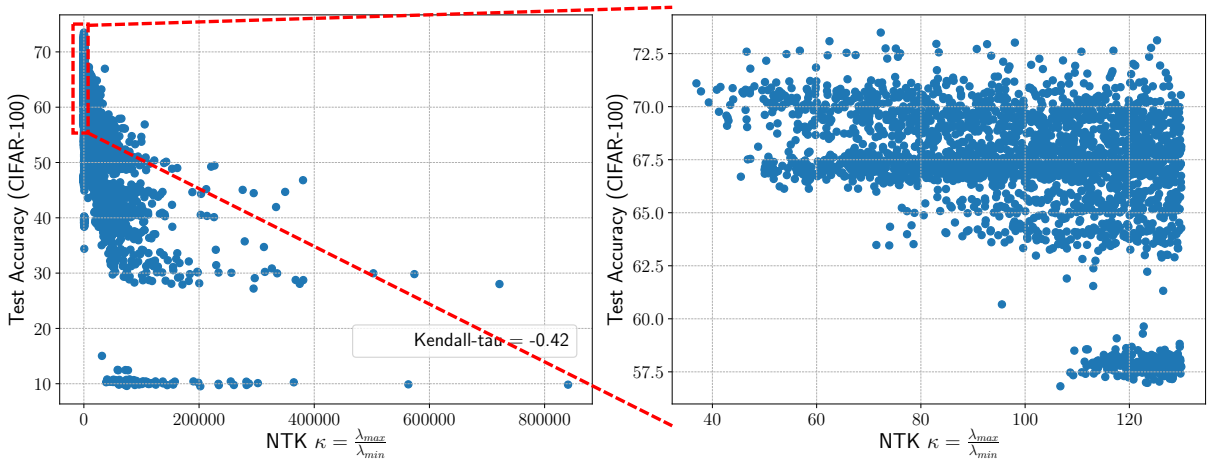


Fig. 5.1 The correlation between κ and CIFAR-100 test accuracy, as shown in Chen et al. (2021). The left plot shows all 15,625 models, while the right plot zooms in on the narrow rectangular region highlighted in red. The two horiztonal dashed lines are simply how Chen et al. indicate that the right plot is a zoomed in version of the left.

5.2.4 Linear Region Count

To measure this performance potential, Chen et al. use a metric called *Linear Region Count* or LRC, which counts the number of discrete *linear regions* discernible by the model. To fully understand LRC two concepts from theoretical analysis of ReLU-based neural networks (Pascanu et al., 2013) must first be discussed: activation responses and linear regions. An activation response represents the collective activity of a model's ReLU operations for a particular input, taking the form of a binary vector of equal length to the total number of ReLUs in the model. The nth element of this vector is 1 if the ReLU was activated (i.e, produced a nonzero output) while processing the input in question, and 0 otherwise. A linear region represents an area of input that produces an identical activation response. Figure 5.2 shows a simple 2D example of both concepts. In essence, a ReLU operation folds the input space along some hyperplanar boundary.

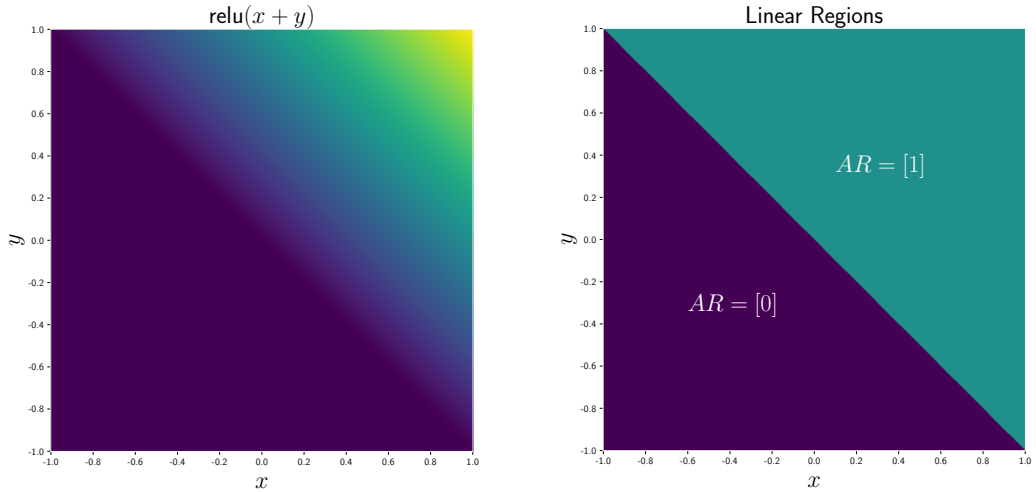


Fig. 5.2 On the left is a simple 2D model $f(x,y) = \text{relu}(x+y)$. The right shows the two activation responses (AR) and linear regions of that model.

Multiple ReLU operations perform compositional, piecewise foldings, further segmenting the input space as shown in Figure 5.3.

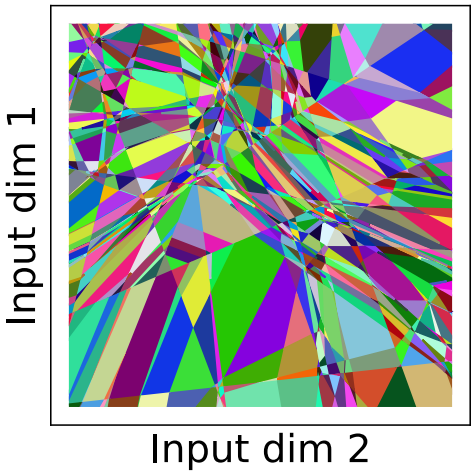


Fig. 5.3 The linear regions of a more complex model given a 2-dimensional input, as shown in Chen et al. (2021).

The linear region count therefore measures how efficiently the model performs those piecewise foldings; two models can have the same number of ReLU operations but perform drastically differently in LRC. For example, one model might produce exactly orthogonal foldings and therefore have maximal LRC, while another's foldings may be entirely coplanar and therefore only have a LRC of 2. The crux of all of this is that the more discrete regions of the input space the model can produce, the more detail it can mathematically distinguish and thus act upon. As such, LRC is used as a proxy for a model's learning potential. Chen et al. again examine this correlation over NAS-Bench-201, and find a Kendall-tau correlation of 0.5 as shown in Figure 5.4. Examining this plot provides a similar conclusion to that of the NTK analysis: while low LRC can sometimes provide good performance, high LRC ensures it.

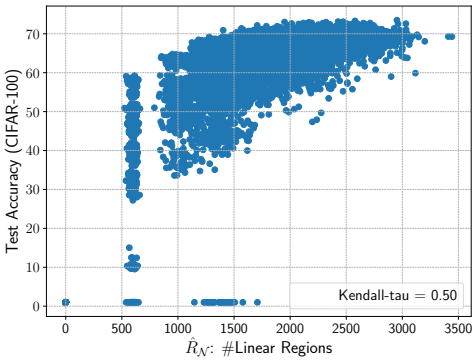


Fig. 5.4 LRC versus model performance, NAS-Bench-201 models on CIFAR-100 (Chen et al., 2021).

To measure LRC in practice, Chen et al. divide the potential inputs of the model into small 3x3 monochromatic regions. A few thousand of these regions are passed into the model, and the total number of discrete activation responses produced by those inputs is recorded. This actually performs a subtly different measurement than LRC: rather than absolutely measure

LRC, it measures the number of linear regions produced by that specific subset of inputs. The hope is that the sample is large enough to adequately distinguish differences in LRC between candidate models, but runs the risk of inaccurately representing the true LRC. To make a worst-case example, imagine a model that has only n possible linear regions, but those n happen to exactly align with the sample regions chosen in the LRC sampling. This model would report the maximum possible LRC in the sampling, and would therefore score equivalently to a perfect model with infinite LRC. This means that Chen et al.'s LRC sampling cannot meaningfully distinguish the LRC of two models if both models' true LRC equals or exceeds the sample size. While increasing the sample size could potentially circumvent this issue, this soon becomes prohibitively expensive. Chen et al. therefore use the same slim models from the NTK measurement to sample LRC from; the slim models have a much smaller LRC than the full-sized models, one that is sufficiently smaller than the sample size of 3,000 chosen by Chen et al.. However, this incurs the same approximation problem as NTK, in that it must be assumed that the LRC of the slim models correlates with the LRC of the full-sized model, again as assumption not touched upon in the original paper and only revealed in the implementation.

5.2.5 Practical Concerns and a Joint NTK-LRC Metric

Caveats about the slim model approximations aside, the practical issue with using LRC or NTK is that they are both exclusively parameter-value and input dependant: they provide a meaningful result only when comparing models with identical parameter values over identical inputs. In order to evaluate models in the search space, Chen et al. use a supernet approach, where any model in the space can be represented as a subgraph of the supernet. To compare any two models in the space they then create two identical copies of the supernet, initialize both with identical parameter values, and then disconnect the irrelevant edges from each supernet. This creates two subgraphs with identical parameters but different connectivities, and therefore different architectures. This process can be adapted to compare any arbitrarily large set of models, and this is exactly how Chen et al. evaluate all the models within the NAS-Bench-201 search space to find the correlations between NTK, LRC, and final test accuracy.

From these results, Chen et al. discover that by adding the descending NTK rank to the ascending LRC rank of all models[1] within the search space provides a much better correlation with model performance. This makes sense, if lower NTKs and higher LRCs correlate with better performance, a metric that jointly rewards low NTK and high LRC would potentially have greater predictive power. The joint NTK-LRC metric provided a Kendall-Tau correlation of 0.64 with test accuracy which improves upon the -0.42 and 0.5 of NTK and LRC respectively. The correlation also demonstrates that the slim model approximations used in the calculations of NTK and LRC work well enough, at least across the search spaces and datasets examined in the paper.

[1] Here, lower NTKs would provide larger descending ranks, and higher LRCs provide larger ascending ranks, therefore producing larger combined ranks for models with low NTK and high LRC as compared to other models in the set.

5.3 Design

5.3.1 Models

Fundamentally, SpiderNet models are near identical in construction to BonsaiNet models. They are cell structured, where each cell contains edges and nodes. Edges consist of parallel operations, one of each available operation. Each edge contains operation pruners, to allow deletion of operations from edges. Nodes perform tensor aggregation, again summation for the same reasons as BonsaiNet. The crucial difference is that each cell has an input node for each previous cell and the original model input; therefore, the fourth cell has four input nodes (one for Cell 1, 2, and 3, plus the model input), the third has two, etc. Each cell is initialized with just its input nodes, one intermediate node, and a single output node. One edge is then placed between each input and the intermediate node, and then a single edge is placed between the sole intermediate node and the output node. This configuration is shown in Figure 5.5:

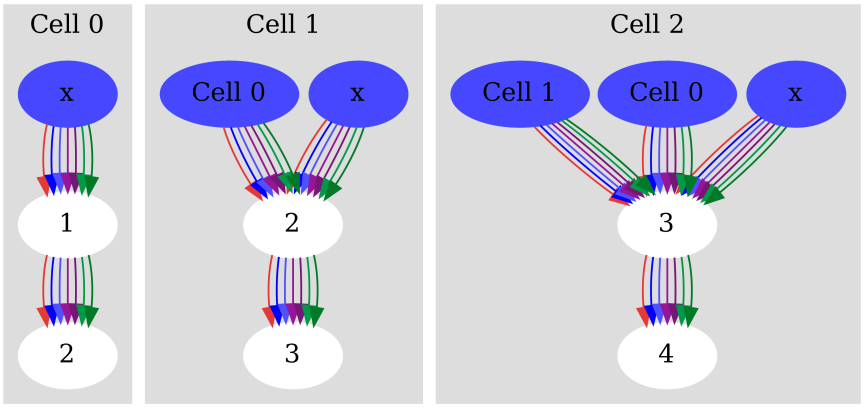


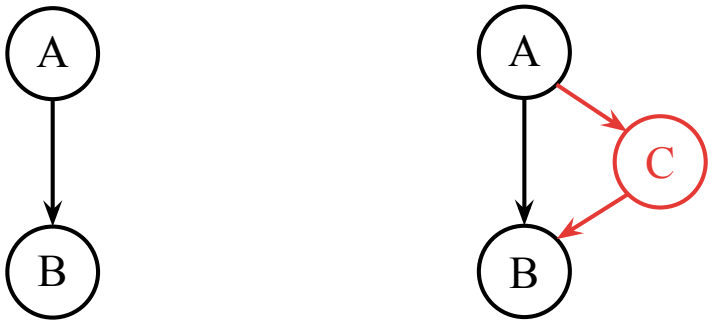
Fig. 5.5 The initial state of a three cell SpiderNet model, with the input nodes marked in blue.

This starting point can be thought of as a suitably ‘minimal’ starting point as it provides two edges per input: one to the single intermediate and one to the output. Additionally, the single intermediate has an edge to the output that can combine and synthesize information from the inputs. Evolutions of this architecture can therefore reinforce the pathways from the input or to strengthen the ‘mixing’ portion of the cell.

5.3.2 Mutation

To accomplish the goal of a dynamically expanding search space, it is first important to define what “expanding the search space” actually entails. There are many ways to define model expansion, and a number of ways of actually designing the mechanism the model would use to accomplish it. Primarily, to allow the cell’s internal connectivity to be evolving and expanding, graph mutations were focused on as the means of expansion. The term “mutation” is chosen here in reflection of Real et al. (2017)’s usage of the term, where it is used to describe graph modification operations that are applied to candidate models that are similar to “actions that a human designer may take when improving an architecture.” The same principle applies here, in

that mutations refer specifically to the graph modification, not necessarily to a stochastic process occurring at random within a gene pool. Specifically, the target was a single mutation operation that when composed with edge deletion via pruners could construct any possible directed acylic graph given some arbitrary starting point. The operation eventually selected is referred to as a *triangular mutation*, which acts on an arbitrary edge $A \rightarrow B$ that connects two nodes A and B within a model, adding a third node C and two new edges. These edges are wired such that for the nodes A, B, and C, one node sends two outputs, one node receives one input and sends one output, and one node receives two inputs, as shown in Figure 5.6.



(a) Original Edge (b) Post mutation

Fig. 5.6 The triangular mutation, with new edges and nodes depicted in red.

This operation is composable with edge deletion to create a huge variety of graph patterns, as they all crucially introduce a branching path into a linear node relationship. This creates a seed for further branching by later mutations along any of these three edges, which allows a small number of mutations to create immense cellular complexity.[2] For architecture examples of models generated by such triangular mutations, Figure 5.7 shows a model after 45 triangular mutations were performed at random. To further see the architectural variance, see more examples in Appendix C.

A crucial question revolves around what operations the new edges should be initialized with. There are two possible perspectives to take here. It could be argued that the new edges should be initialized only with operations present in the original edge; if an edge has deadheaded out a number of operations, it has decided that these operations are not necessary in this location and thus they should not be placed in the new edges. On the other hand, new edges might need the opportunity to decide for themselves which operations they need. Considering that these new edges are not guaranteed to serve the same purpose within the cell, predetermining their initial operations from a different edge seems premature. For this reason, it seems that this second approach is the better one, and as such new edges are initialized with all operations from within the operation space. However, this is based purely on conceptual reasoning, and would require experimental work to fully justify.

[2]The cells split and grow much like the branching propagation of a spider plant, hence the name SpiderNet and thus continuing the botanical naming scheme started by BonsaiNet.

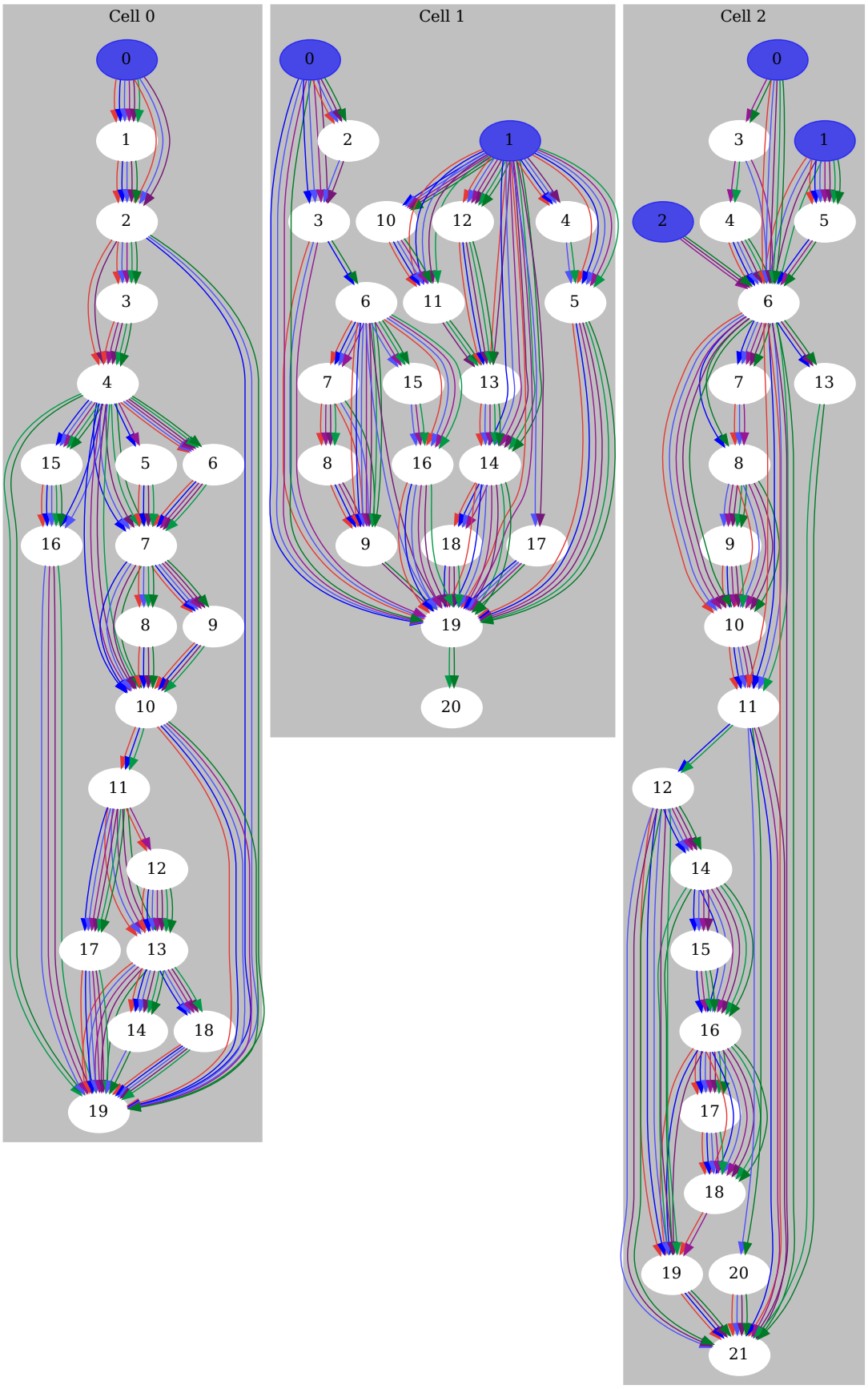


Fig. 5.7 A sample SpiderNet model after 45 random mutations. Cellular inputs are shown in blue.

5.3.3 Mutation Practicalities

Mutations run afoul of the same codependence problem as identified in Section 4.5.3: if new, untrained operations are added midway through model training, they alter the distributions that downstream operations are expecting, worsening the model's performance. As such, the new operations were almost instantly pruned away to restore the connectivity to the model's codependent expectations as best it can, meaning that the mutation was usually exclusively destructive. To remedy this, the model weights are reinitialized after each mutation, much like how BonsaiNet resets model weight after pruning cycles. This resets any codependency, allowing the model to make use of all new operations if it so desires.

5.3.4 Selecting Edges to Mutate

With the edge mutation operation defined, the next step is to identify the mutation criteria; how should the model select an edge to perform this mutation on? There are three ways to approach this problem; stochastically, mathematically, or experimentally. Stochastic mutation is easy enough, just randomly select an edge to mutate each time. If the model does not like the new edges, it can simply prune away the operations and wait for a better mutation next time. Mathematical mutation would involve defining some metric that is measured over each edge, and then choosing whichever edge maximizes/minimizes this metric for mutation.

Designing such a metric that determines the most valuable edges to mutate is tricky, but the key insight here is that the models carry implicit information about the value of their edges via those operations' pruner weights. There is a variety of summary statistics over these pruner weights, each of which can then be investigated to see if it makes for a suitable metric for mutation selection. Each potential metric acts over the collective weights of the pruners in the edge from the last n batches. Therefore, when looking at seven pruners over the last 100 batches, there are 700 data points that the metric operates over. Four such summary statistics over those batches were identified as candidate metrics:

- **Mean of Pruner Weight**: The mean of the collective weights. Higher values of this metric would correspond to edges with more operations that are strongly preserved, while lower values imply few operations and strong deadheading.

- **Variance of Pruner Weight**: The variance of the collective weights. High values imply the pruners in this edge are fluctuating significantly, without confidence in whether these operations should be preserved or removed. Low values indicate the pruner weights are stable, without much movement from batch to batch.

- **Mean of Absolute Pruner Weight**: The mean of the absolute value of the collective weights. High values imply that the pruners in the edge have made strong decisions *in either direction* about the operations within, while low values indicate that most operations in the edge are close to the decision boundary.

- **Variance of Absolute Pruner Weight**: The variance of the absolute value of the collective weights. High values mean that the decisions are varied and fluctuating between batches. Low values again imply consistency in the decision making of the pruners, in either direction.

In addition to these mathematical metrics, experimental means of determining edge value are also viable as potential candidates. Here, the importance of each edge in the model is sampled, and this sampled importance is used to inform mutation. Ideally, the sampling should reflect the independent value of each edge, so that multiple edges can be mutated simultaneously without worrying about interaction effects. One possible system is Shapley values, specifically those identified by Lundberg and Lee (2017) as described in Section 5.2.1.

For the purposes of SpiderNet, SHAP is reframed; rather than examine how varying the input features can affect model output, instead the focus in on how variable model connectivity affects the output, given fixed input. To do this, the edge connectivity of a SpiderNet model is modeled as a binary vector $\vec{e}$ of length equivalent to the total number of edges in the model. A 0 at a particular position i indicates the ith edge is omitted from the model (corresponding to replacing that edge with an identity), while a 1 marks the inclusion of that edge. A subset of the training data is chosen as the fixed input, as therefore the model is transformed from a function $F(\text{inputs} \mid \text{edges})$ to $F(\text{edges} \mid \text{inputs})$, that is, the edge configuration can be varied given a fixed input to see how it alters the output.

To practically do this, each edge in the model graph is assigned an index. When given some binary edge configuration vector $\bar{e}$ (i.e., the new 'inputs' to the model under the new formulation), the edge with index i receives the value $\bar{e}_i$. If $\bar{e}_i = 1$, the edge performs its usual operation. Otherwise, it simply performs a zero operation, effectively removing itself from the model. Once the edge configuration vector inputted to the model has toggled each edge as appropriate, the fixed set of input images is passed through the model to determine its output.

This model can then be passed into SHAP, which returns a vector of importances $\vec{\phi}$ for each edge in the model, which indicates how much each edge contributed to the final accuracy of the model. Large Shapley values indicate that an edge is highly important to a model's performance; its omission significantly harms the model's accuracy. Small Shapley values on the other hand indicate the opposite, that the edge can be removed relatively harmlessly. In the extreme, negative Shapley values indicate that an edge is actually detrimental to model performance, and their removal improves accuracy. While this has obvious utility as a pruning heuristic, it could also be useful in indicating which edges would benefit most from reinforcement via the triangular mutation.

5.3.5 Comparing Metrics

A convincing argument could be made that any of the above metrics in either direction would be a good mutation target; for example, you could say that a large mean pruner weight indicates that an edge is highly favored, and thus should be further strengthened with more edges. Alternatively, a low mean pruner weight could indicate that the current edge is not valuable, and that it needs

more reinforcement to be useful. The SHAP metric is similarly ambiguous; is it better to add more edges around the most important edges in the model, or is better to reinforce the weaker areas? Given that these arguments can be made for all eight of the possible metrics, it was best to experimentally test each to determine which produces the best models. To do this, 3 models were tested with identical configuration for each of the 10 metrics (five statistics, with two directions each). Each model underwent cyclical reinitialization similar to the way BonsaiNet does as described in Section 4.5.3. Here, models were trained until their first deadhead cycle, then the three edges with the best metric values were chosen for mutation. The model was then reinitialized and trained for another deadheading cycle, for a total of five cycles. After these five cycles, the model was trained for 64 epochs to compare their top-end performance. 64 epochs are chosen because comparative performance after 64 epochs between models is a decent predictor of their final performance rankings. To demonstrate this, Figure 5.8 shows the Spearman correlation between *n*th epoch accuracy and a final accuracy aggregated over 29 different BonsaiNet and SpiderNet models, where 64 epochs shows a Spearman correlation to final rank of 0.66.

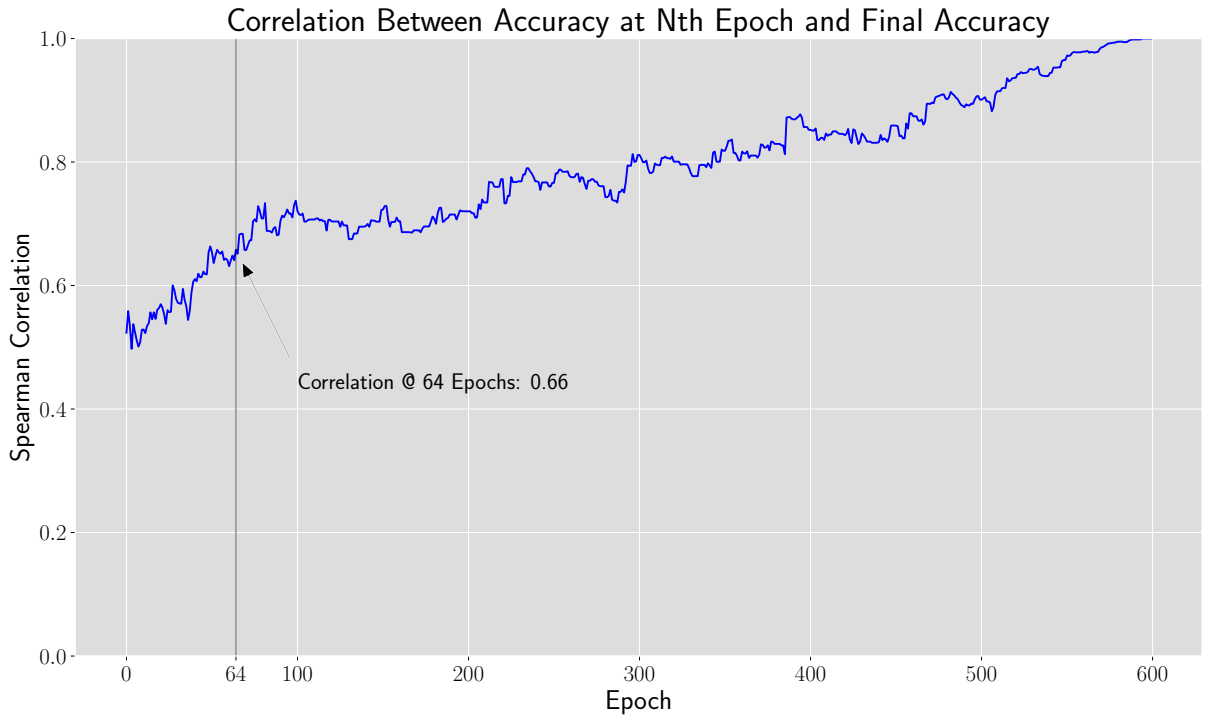


Fig. 5.8 Correlation between accuracy at 64 epochs and their accuracy at 600 epochs over 29 Bonsai and Spider models.

From this experiment, the metrics that typically produce the best models can be determined. More importantly, it can be determined how good a metric is at identifying the best mutation candidate by comparing it against its opposite metric; if choosing edges that have the maximum values of some metric identifies the best possible operations to mutate, choosing the minimum valued edges should be the worst possible. Therefore, the aim is to find a metric which produces a large performance gap between its minimum and maximum case. While a metric not having a clear difference between its maximum and minimum case does not imply it is poor at distin-

guishing good and bad mutation candidates (as it could simply be indicative of a non-linear or bell-curved ranking; perhaps the maximum and minimum is the worst and the most central value is optimal), choosing a metric with a linear ranking is simplest conceptually and mathematically. These differences between minimum selection and maximum selection are shown in Figure 5.9, which represents the comparative *search* performance of each metric.

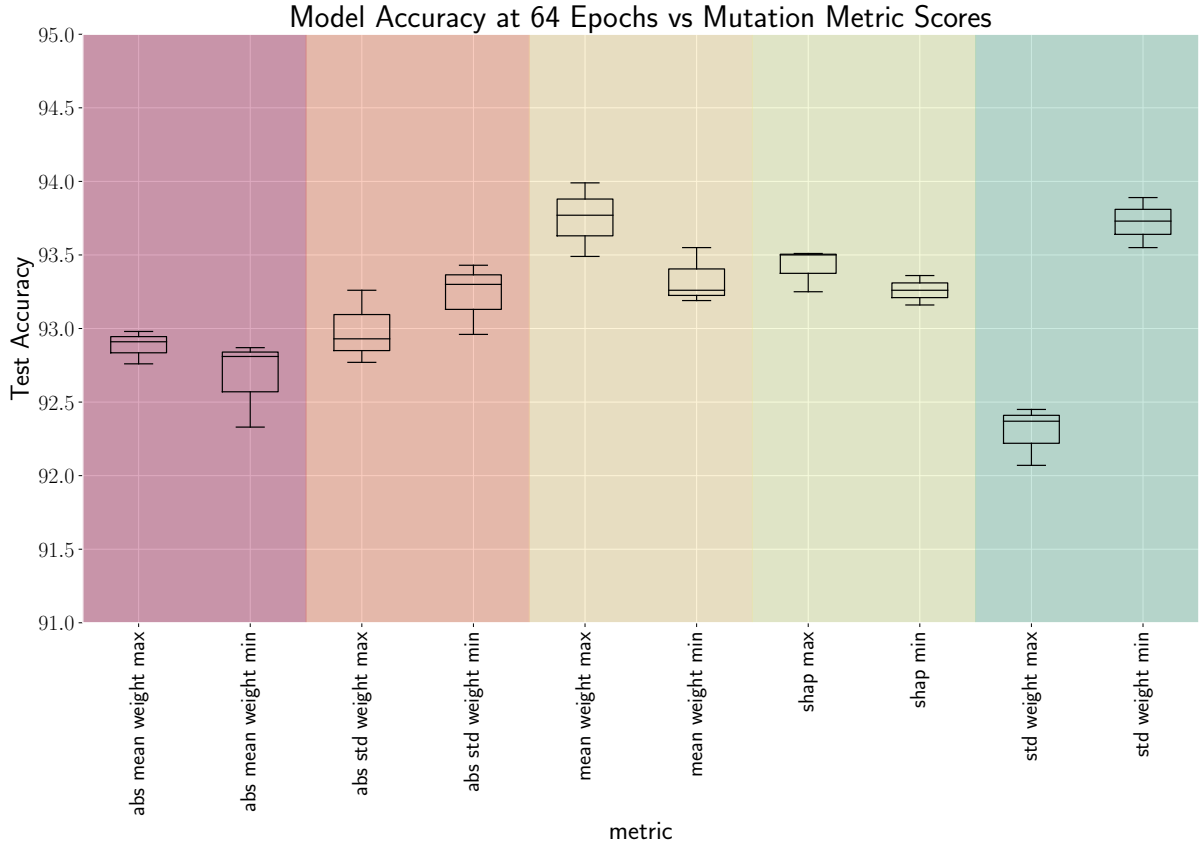


Fig. 5.9 Performance after five mutation cycles and 64 epochs of training for each of the 10 mutation metrics, with the results of three separate models shown in each case. This shows each metric's relative ability to search for architectures, that is, their how well each metric mutates edges based on their *future* effect on performance.

A few things stand out from these results. First, the best overall models were produced by mutating the edges with the maximum mean or minimum standard deviation[3] of pruner weights. This indicates that edges with many operations preserved confidently or pruners that are steadfast in their weights are the best to mutate. More interestingly, notice that while minimum variance models produce some of the best models, the maximum variance models produce three of the four worst models in the entire experiment. This is indicative of that ideal linear ranking that was hoped for, which indicates meaningful difference between the minimum and maximum values of the pruner weight variance metric.

An alternative perspective to take is to identify which metrics produce the most consistent improvement to models cycle-on-cycle. This therefore looks at how a metric might affect a model's performance in the directly subsequent cycle, thus evaluating short-term performance

[3]This metric is also referred to as maximum/minimum variance throughout this thesis.

changes versus the lifetime changes evaluated in Figure 5.9. In essence, the previous analysis gave each metric's ability to select edges that *will* limit model performance, whereas now it will be each metric's abilty to identify edges that *have already* hindered performance. To view this, the accuracy of each model after each mutation cycle and training period is plotted for each of the 10 metrics in Figure 5.10, which represents the comparative *tuning* performance of each metric.

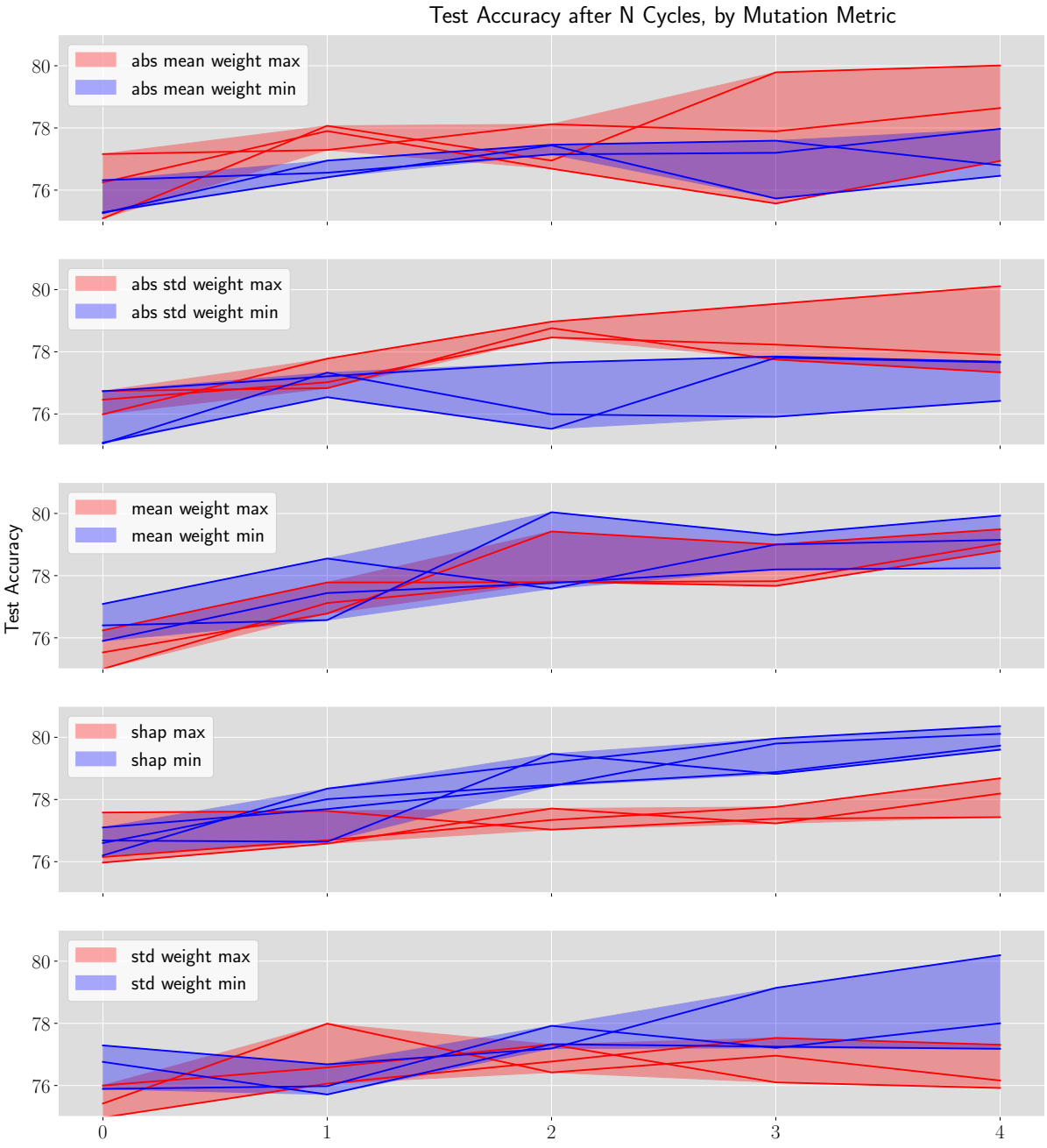


Fig. 5.10 Performance at the end of each mutation cycle for each of the 10 mutation metrics, with the performances of each of the three models overlaid. This indicates each metric's relative ability to tune architectures, that is, how well each metric mutates edges based on their *extant* effect on performance.

There are a few key insights within this plot. Notably, most metrics trend upward cycle-on-cycle, with the exception perhaps being maximum standard deviation. However, this is likely just

due to the model size increasing cycle-on-cycle, as each post-cycle mutation adds a significant number of differentiable parameters to each model. Another interesting thing to note is that the minimum and maximum SHAP metrics are the only two that produce consistently ranked models; all minimum SHAP models perform better than all maximum SHAP models from the second cycle onwards. This is in contrast to its relatively poor, non-discriminative performance in the search case. This indicates that the minimum SHAP model is excellent at identifying edges that have already hindered performance, but that neither metric is particularly able to make accurate predictions about the future performance impacts of edges. This result does make intuitive sense in hindsight; the SHAP metric produces a score that describes each edge's exact influence on the current model accuracy, but has no ability to discern future accuracy.

Of these candidates, minimum variance seems most promising, given its strong performance in Figure 5.9. Since it produces the largest and clearest distinction between maximum and minimum values of the metric after the fixed mutation cycles, it appears to prioritize beneficial mutations. However, there is still one more candidate to consider, which will be looked at in the following section.

5.4 Train-Free Mutations

One final approach that can be taken to determine optimal mutations is to look at a train-free metric like NTK-LRC. Such a metric would allow for the entire model to be mutated immediately, which would drastically reduce search cost.

5.4.1 Searching via Joint NTK-LRC Metric

While in theory the NTK-LRC metric allows for every candidate in the search space to be evaluated 'for-free' in terms of model training, the actual computation of the two metrics still comes with a non-trivial computation cost. For reference, calculating the NTK-LRC of a model comparable in size to those produced by BonsaiNet takes around a minute. Therefore, for larger search spaces the entirety of the space still cannot be fully enumerated, and must be traversed in some intelligent manner. In the case of the DARTS space, Chen et al. (2021) create a tournament selection (Miller et al., 1995) algorithm to find good architectures. Here, a 'champion' model M is compared against a set of challenger models $\mathcal{M}$, each $M' \in \mathcal{M}$ created by removing one operation at random from M. The model $M' \in \mathcal{M}$ with the best joint NTK-LRC is then chosen as the next champion, and the process repeats until a satisfactory model is found. In essence, this is a train-free method of iteratively pruning a supernet to find an optimal subnet, and from this approach Chen et al. find state-of-the-art results comparable to those found by DARTS.

However, as has been discussed at length in thesis, any random model within the DARTS space has the potential to provide state-of-the-art results. How might this method fare in an unbound search space like that of SpiderNet? To examine this, Chen et al.'s tournament selection can be adopted to measure the quality of potential mutations. Given some model M, the set of all models $\mathcal{M}$ that are one mutation away from M is enumerated. The NTK and LRC of each

mutated model M' is compared against the original unmutated model M, and the model $M' \in \mathcal{M}$ that maximizes the combined NTK-descending and LRC-ascending ranks is selected, if such a model exists. This process can then be repeated a number of times, in theory improving the quality of the model after each cycle.

The practical caveat to this approach is the aforementioned cost of evaluating NTK and LRC; if each takes around a minute to evaluate each potential mutation, exhaustively comparing every possible mutation can take around an hour for larger models. This means that the process of finding a single mutation can take an hour, which over the course of many mutation cycles gets prohibitively expensive. To avoid this, mutation candidates are instead sampled at random, and after a certain configurable number of 'good' mutations are found (i.e., ones that do not increase NTK and do not decrease LRC, therefore producing a model that is at worst equivalent to the original model) the sampling ends. Of these found good mutations, the best one in terms of NTK-LRC is performed on the model. This drastically reduces the number of NTK-LRC computations necessary to find a good mutation, other than in the worst-case scenario where every mutation candidate is bad. While this does generally speed up mutation time significantly, it does come at the potential expense of mutation quality; when the candidates are fully-enumerated, the absolute best mutation can be chosen each time. When sampling, only a relatively good one is guaranteed.

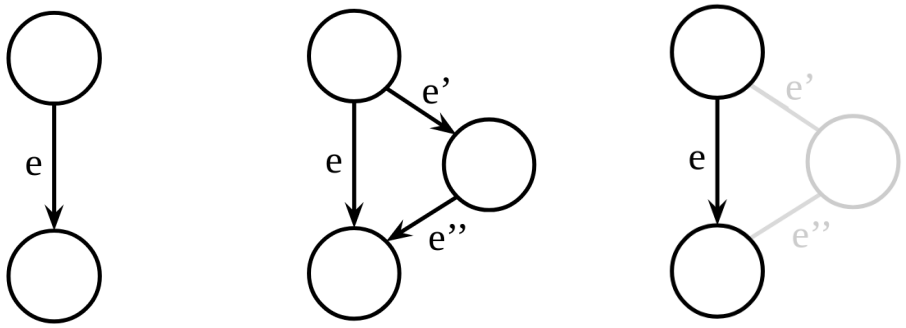
With this, NTK-LRC can now be used as a mutation metric, and Algorithm 5 describes how the NTK-LRC mutation metric selects candidate edges for mutation.

Algorithm 5: NTK-LRC Mutation Selection

Input : Some model M and sample size n_{good}
Let $s(x)$ refer to the VRAM size of some component x as per Algorithm 1;
Let s_{max} be the maximum permitted VRAM allocation;
1 Create S, the slim model of M;
Let $\mathcal{E} = \varnothing$;
2 **for** edge e in M, selected in random order **do**
3 Let S' be an exact copy of S;
4 Mutate e in S', thus creating two new edges e' and e'';
5 Create an identical copy of S' and label the two S'_{on} and S'_{off};
6 Within S_{off}, disconnect e' and e'';
7 Compute NTK and LRC for S_{off} and S_{on};
8 **if** $\text{NTK}_{on} \leq \text{NTK}_{off}$ and $\text{LRC}_{on} \geq \text{LRC}_{off}$ **then**
Add e into the set of good mutation candidates $\mathcal{E}$;
9 **if** $|\mathcal{E}| \geq n_{good}$ **then**
break;
if $|\mathcal{E}| > 0$ **then**
10 Let e^* be the candidate mutation with best joint NTK-LRC in $\mathcal{E}$;
11 **if** $s(M) + 2s(e^*) < s_{max}$ **then**
Output : e^*
Output : $\varnothing$

The NTK-LRC algorithm works by first creating a slim model S of our SpiderNet model M in line 1, as described in Section 5.2.3 and Chen et al. (2021). This is done by initializing an exact copy of the SpiderNet model, and then reducing the channel count of each operation

down to 1. Next, line 2 selects edges at random from the model as potential mutation candidates. Before trialling the mutation of some selected edge e, line 3 creates an identical copy S' of our slim model S; all trial mutations are performed on the copy, thus keeping the original slim model S as an unmodified template that can be used again. Edge e is then mutated within S' in line 4, thus creating two new edges e' and e'' as per the rules of triangular mutation. Next, a copy of the mutated S' is made in line 5. This creates two identical mutated models, the original S' and the copy, which are then labeled as S'_{on} and S'_{off}. Within S'_{off}, the new edges e' and e'' created by the mutation are disconnected (simply by multiplying their outputs by 0); this means that S'_{off} is now mathematically identical to the original unmutated model S, but physically has the same parameters as S'_{on}. See Figure 5.11 for a simplified visualization of S, S'_{on}, and S'_{off} at this stage of the algorithm.



(a) S (b) S'_{on}, mutated e (c) S'_{off}, e' and e'' disconnected

Fig. 5.11 S, S'_{on} and S'_{off}, shown in the local area around the candidate edge e. Notice that while S'_{on} and S'_{off} are operationally identical, (that is, contain identical copies of e, e' and e'' and all component operations within), S'_{off} is mathematically identical to S.

This is necessary because NTK-LRC is only useful as a comparative metric between two identically parametered-models, as described in Section 5.2.5. Since mutation implicitly modifies a model's operations and therefore parameters, S and S'_{on} cannot be meaningfully compared. Using S'_{on} and S'_{off} remedies that issue, by creating two identically-parametered models that mathematically represent a mutated and unmutated model respectively. As such, the NTK-LRC delta induced by the mutation can then be measured, which happens in line 7.

If this mutation produced a favorable NTK-LRC (that is, decreased NTK and increased LRC) it is added to the set of found good mutations $\mathcal{E}$ in line 8. This whole process repeats until a satisfactory number of good mutations are found or all edges have been sampled, at which point the algorithm exits the edge sampling loop (line 9). If any good edges are found, then the one with the most favorable joint NTK-LRC is selected in line 10 as the final mutation choice. If this mutation is performable on M within the available memory space (line 11), it is returned as the selected mutation. Otherwise, no edge is returned.

5.5 SpiderNet Algorithm

A variety of ways to select good mutations within models have now been covered, which means there is now a variety of ways to grow models within the SpiderNet algorithm. As such, all necessary components for the full algorithm are present, which is stated in Algorithm 6.

The algorithm first starts by initializing a minimum viable model M and choosing the mutation metric μ. The rest of the algorithm is very conceptually similar to BonsaiNet: the model is trained and pruned, then a number of mutations are performed (if possible), then the whole process repeats until the model has reached sufficient size. Then the full model trains and prunes freely until is fully trained.

Algorithm 6: The SpiderNet Algorithm

Initialize 'minimum viable model' M as shown in Figure 5.5;
1 Choose mutation metric μ and sort direction `min` or `max`;
Let $s(x)$ refer to the VRAM size of some component x as per Algorithm 1;
Let s_{max} be the maximum permitted VRAM allocation;
Let E_{train} refer to the desired number of training epochs;
for *i in* $[1, \#_{mutation\ cycles}]$ **do**
  if inter-cycle training+pruning is required/desired **then**
    Reset model parameters;
    for *Epoch in* $[1, \#_{epochs\ per\ cycle}]$ **do**
2   Train + prune;
3   Deadhead;
  Compute metric scores for each edge in model: $\vec{\mu} = \{\mu(e) \mid e \in M\}$;
  Choose top n edges from $\vec{\mu}$ according to sort direction;
  for Each chosen edge e **do**
    if $s(M) + 2s(e) < s_{max}$ **then**
      Perform mutation on edge e;
4 Train + prune freely for the E_{train} epochs;

5.6 CIFAR-10 Experiment Designs

Given the above algorithm and metrics, a variety of experiments can be designed over CIFAR-10 to evaluate their various efficacies, the details of which are presented in this section.

5.6.1 Model and Training Configuration

Each SpiderNet model has much the same sorts of configurable hyperparameters as BonsaiNet, and each model presented in this section adheres to the following design configuration:

- **GPU Space**: 9.5 if run on an NVidia 1080Ti, 20 if using the NVidia 3090.
- **Scale**: 64
- **Sections**: [[`N`], [`R`], [`R`]]

- **Operations**: Same as DARTS: Identity, 3x3 Max Pool, 3x3 Average Pool, 3x3 Separable Convolution, 5x5 Separable Convolution, 3x3 Dilated Convolution, 5x5 Dilated Convolution

The training parameters of the models like learning rate, learning rate annealing, and dropout, are identical to those used in the BonsaiNet experiments. For exact details, see Section 4.9.

5.6.2 Evolution via Minimum Variance

Given the promising results of the minimum variance metric as shown in Figure 5.9, this was chosen as the primary mutation metric for full training experiments. Much like the BonsaiNet algorithm (Algorithm 4), models train in cycles of four epochs. After those four epochs a deadheading pass is performed, after which point mutation is attempted. Here, the n edges with the minimum variance of their collective pruner weights are selected as mutation candidates. Optionally, this selection can happen more granularly, instead selecting $\lfloor n/c \rfloor$ edges from each of the c cells within the model, such as to balance growth throughout each cell. For each candidate mutation, the sum of the VRAM sizes of the new operations is considered, and if there is sufficient VRAM space to add these new operations to the existing model, the mutation is performed. Otherwise, the model skips this mutation and evaluates the next one. After all n mutations have been evaluated and potentially performed, all the tracked statistics are cleared. This ensures that the next batch of statistics that is gathered will reflect the new architecture of the model.

After a number of mutation cycles the model is considered to have grown to its full size, and trains uninterrupted to the full desired epoch count. Usually, 10 mutation cycles with three mutations per cycle (such as to perform one mutation per cell per cycle) is sufficient to build a model comparable in size to the 1080Ti large-cell Bonsai models, and as such allows a fair comparison to the two. Additionally, to explore SpiderNet's capabilities in the larger 3090 domain, runs were also performed with 20 mutation cycles with three mutations per cycle. To further ensure parity, training is performed on CIFAR-10 with identical hyperparameters to that of the Bonsai large-cell models, as discussed in Appendix B.

5.6.3 Evolution via Joint NTK-LRC Metric

Using NTK-LRC as the mutation metric μ provides an optimal mutation to perform, if such a mutation exists and fits in memory. Much like in Minimum Variance, this process is again split across each cell, such that an optimal mutation is looked for and potentially performed within each cell each mutation cycle.

One interesting question that arises when using NTK-LRC as a train-free metric regards pruning; how should it be done, considering that the differentiable pruners require training to operate? One option is to use NTK-LRC as a slightly more expensive replacement for the normal mutation metrics and perform the training and pruning loop as normal. Since this will be slower than using the other metrics like minimum variance that are calculated as a by-product of the

training and pruning cycle, this approach would require excellent performance in order to justify the additional time cost.

5.6.4 Random Search and Ablation Studies

Following in the pattern of Bonsai and "Evaluating The Search Phase of Neural Architecture Search" (Yu et al., 2019), a random search is devised to compare SpiderNet's ability to find good networks within the search space. In order to do this, it is first necessary to enumerate the different 'non-random' components of the algorithm. In the case of SpiderNet, there are three: guided mutation by metric, inter-cycle pruning, and intra-training pruning, from lines 1, 2, and 4 of Algorithm 6 respectively.

Replacing guided mutation with a random measure is simple; rather than choose the edges with optimal metric values for mutation, an edge is selected at random. Each random run then attempts to perform as many random mutations as was attempted during a guided mutation run, typically 45 (15 cycles of three mutations each). Not all of these mutations will actually occur due to VRAM space constraints, but both methods will have an equal number of mutation *opportunities*.

Next is inter-cycle pruning, that is, the round of differentiable pruning that occurs after each round of mutations. To replace this with a random measure, a random selection of operations is deadheaded after each mutation cycle. To ensure a fair comparison, the random experiment takes an existing SpiderNet run as a template for deadhead frequency, recording the number of deadheads that occurred after each mutation cycle (line 3 of Algorithm 6). The template run used in these experiments comes from a Evolution via Minimum Variance run, with the following per-cycle deadhead counts:

$$\vec{dh} = [4,2,5,4,7,13,10,11,14,13,15,17,16,24,26,22]$$

Then in the random run, $\vec{dh}_i$ operations are selected at random for deadheading after the ith mutation cycle. Another option is to simply not perform inter-cycle deadheading altogether as an ablation study, to examine the necessity of its inclusion. Both options are considered in these experiments.

The final non-random component is intra-training pruning, the free pruning that occurs during training. As Section 4.9.5 showed, this can be immensely powerful in boosting network performance. Therefore, in order to be able to differentiate performance stemming from mutation policy versus intra-training pruning, it is crucial to examine intra-train pruning's effects as best as possible. To do this, the number of deadheads that occurred during the template run's 600 epochs of training are observed (line 4 of Algorithm 6). For the specific template run used it was 153. Therefore, once the random runs have finished mutating, 153 operations are deadheaded at random from the network. The network is then trained without pruning. In addition, random runs are performed that do not undergo this final deadheading, as well as randomly grown runs that

can freely prune via differentiable pruners as normal. Both of these latter runs serve as ablation study regarding inter-cycle and intra-train pruning.

Label	Mutation	Inter-cycle Pruning	Intra-train pruning
Random 1	15 Random Cycles	Random via $\vec{dh}$	153 randomly chosen
Random 2	15 Random Cycles	None	None
Random 3	15 Random Cycles	None	Differentiable
Random 4	15 Random Cycles	Random via $\vec{dh}$	Differentiable

Table 5.1 Details of the four random experiments that were run.

Four different combinations of random/non-random components are chosen, and are detailed in Table 5.1. These four were chosen for their specific comparative properties. Namely, Random 1 is the pure-random run that replaces each guided component of Spider with its random equivalent, thus serving as a purely random reference point against which to compare all experiments against. Random 2 is a pruning ablation study, which in comparison to Random 1 will inform us of the effects of the random pruning performed. Random 3 evaluates how well guided-pruning can mesh with random mutation. This will be useful as direct comparison against a guided-run, to determine how much of its performance stems from the guided-mutation versus guided intra-train pruning. Finally, Random 4 evaluates how random pruning during the mutation stage (which increases the amount of space available for mutations, therefore leading to an increased number of mutations as more of the 45 opportunities will be successful) might benefit final performance.

5.7 CIFAR-10 Results

In Table 5.2 the results of the two guided mutation methods (Minimum Variance and NTK-LRC) are compared against the four random methods. A comparison against BonsaiNet and other NAS methods is shown in Table 5.6 later in the chapter.

5.8 Discussion

First, the overall ramifications of the results presented in Table 5.2 will be explored. Then, the table will be broken into specific subsets of runs whose comparisons provide interesting insights.

5.8.1 Overall Results

Of the guided mutation runs, the NTK-LRC run is superior in total runtime, search time, and accuracy. The only place it falls short is in parameter count, which is 1.4M higher than that of the Minimum Variance run. While only one run of each metric was trained to the full 600 epochs due to time constraints, these performance trends were also visible among runs that were terminated early due to runtime bugs and power failures that plagued early Spider experiments. Interestingly, the concerns regarding the expense of NTK-LRC mutation speculated in Section 5.6.3 (that

Growth Strategy	Allotted VRAM	Test Accuracy	Search Time	Total Time	Params
Min. Variance	9.5	96.92%	2.9	56.8	6.2
Min. Variance	20	97.08%	6.6	89.8	7.4
NTK-LRC	6.5*	96.78%	5.8	36.2	**4.0**
NTK-LRC	20	**97.13%**	**5.4**	74.9	8.8
Random 1	20	96.96%	-	69.9	12.9
Random 2	20	97.03%	-	81.4	16.8
Random 2	9.5	96.78%	-	**30.5**	**4.0**
Random 3	20	96.97%	-	85.4	11.1
Random 4	20	96.81%	-	84.7	9.1

*This was run was performed on the 3090, the limited GPU access was due to improper VRAM deallocation of a previous experiment. However, this run demonstrates some interesting properties about SpiderNet's ability to perform in low allocation regimes, so is included in these results.

Table 5.2 Results of the nine SpiderNet runs conducted. All search and run times reported in this section are given in GPU hours, max VRAM in gigabytes, and parameter counts in millions.

NTK-LRC would be much more expensive than Minimum Variance and thus would need greatly improved performance to be worthwhile) ended up entirely unfounded. While mutation via NTK-LRC did provide increased performance, mutation via NTK-LRC was actually over an hour faster than via Minimum-Variance and produced a model that trained around 15 hours faster, and thus was pretty significantly cheaper.

Additionally, the runs of lower VRAM specification demonstrate that the SpiderNet performs well under different constraints: apart from differing VRAM caps, these runs were configured identically. This demonstrates a versatility for performing NAS in different hardware domains, specifically one that comes with very little need for configuration tuning. While the configurations that performed well in the high-VRAM domain may not be optimal for low-VRAM domains (nor are they guaranteed to be optimal even in their intended range), SpiderNet finds very good models regardless. This is exciting from an "applied NAS" perspective, in that it minimizes the need to tune the algorithm to the desired domain. This means that SpiderNet has fulfilled the goals that were outlined in the introduction of this chapter (Section 5.1) wherein it was of interest to explore an algorithm free of the structural configuration tuning that weighs down fixed-structure algorithms like DARTS or BonsaiNet.

In terms of the randomly mutated models, the non-pruned Random 2 produced the best performance, while the purely random Random 1 produced the fastest model by runtime. This is an interesting result that ran contrary to my expectations, which stemmed from the results of the Bonsai supernet experiments (Section 4.9.5). In those, an unpruned supernet performed very well, while an identical supernet that was differentiably pruned produced better accuracy in fewer parameters and less overall runtime. The analogue here would be Random 2 and Random 3; if the unpruned supernet of Random 2 performed so well, it might be expected that a similarly-grown supernet that could differentiably prune would again produce better accuracy more cheaply. However, Random 3 produced worse accuracy four hours slower than Random 2, although it did use 5.7M fewer parameters. The same could be said about the Random 1 and Random 4 pair; all

prior results about differentiable pruning would suggest that using it as a replacament for random pruning in the training stage should increase accuracy and decrease cost. However, Random 4 was significantly less accurate than Random 1 as well as around 15 hours slower overall.

One possible explanation to these results is that all previous pruner experiments were performed over identical supernets, while these four random runs each have a different randomly generated structure. Given how drastically the structure of SpiderNet models can vary (see Appendix C for examples of this), the performance differences seen may be the result of the architecture's implicit capabilities, moreso than the effects of pruning. To fully decouple these effects, the random methods would all need to be performed over the same randomly-mutated model, that is, pick a random sequence of mutations that are then deterministically performed every time. However, this is a tangential line of experimentation to the main focus of the work, and thus was not prioritized.

5.8.2 Best NAS versus Best Random

The first analyzed pairing is the best guided run against the best random run, shown in Table 5.3:

Growth Strategy	Allotted VRAM	Test Accuracy	Search Time	Total Time	Params
NTK-LRC	20	**97.13%**	5.4	**74.9**	**8.8**
Random 2	20	97.03%	-	81.4	16.8

Table 5.3 The best random run (Random 2) versus the best guided run (NTK-LRC).

The most immediate conclusion that may be drawn from this is that there are only marginal accuracy differences between the guided and random runs, with the best guided run scoring only 0.10 percentage points better than the best random run. However, there is a large parameter difference between the two; the guided run used less than half the parameters of the random run. Additionally, the guided run from start to finish took around six hours less than the random run. These latter two points are particularly interesting; the whole random-search condemnation of NAS stems from random search providing a model of equivalent accuracy and parameter count "for free". In any search space wherein every candidate model has an identical count of single-path edges (DARTS, for example) this might be a valid assumption, as the single-path restriction (see Section 4.2) means that every possible model contains an identical number of operations (equal to the total number of edges e in the model), meaning training times will only vary based on the computational complexity of the selected operations. Meanwhile, if operations (and thus per-edge parameter counts) are selected from a uniform distribution (such as in Li and Talwalkar (2019)), it is expected that the total number of parameters in any randomly generated model will also be roughly similar.[4] As a result, neither the Yu et al. (2019) or Li and Talwalkar (2019) papers that started the random search discussion consider model training time a factor,

[4]To test this, 100,000 models were generated from a single-path, $e = 57$ Bonsai search space via uniform operation selection. The distribution of parameter counts was highly normal (Shapiro-Wilk statistic 0.9996, p=4.1×10^{-12}), with a mean of 1.67M and standard deviation of 0.28M.

and only the latter mentions parameter count, albeit only to point out the equivalent parameter counts of their random and NAS models.

Meanwhile, the SpiderNet search space is entirely unbounded; models with only three operations are valid members of the search space as are those with 300 or 3,000. Therefore, the time costs of training and parameter counts of any such model in the space varies wildly, and is a massively important factor in determining the quality of any given model. In this regard, the SpiderNet NTK-LRC model's speed is notable; not only does it provide an equivalently accurate (in fact, marginally better) model than the best random-search model, this model is so parameter and time-efficient that its 5.4 hour search time handicap is more than doubly made up over the course of training. Additionally, the NTK-LRC run only used around 15 GB of its maximum allotted 20 GB, while Random 2 used the full 20 GB, another efficiency not considered in random-search studies.

5.8.3 Best NAS versus Pure Random

Next is the comparison of the best guided run to the purely random run, shown in Table 5.4:

Growth Strategy	Allotted VRAM	Test Accuracy	Search Time	Total Time	Params
NTK-LRC	20	**97.13%**	5.4	74.9	**8.8**
Random 1	20	96.96%	-	**69.9**	12.9

Table 5.4 The fastest random run (Random 1) versus the best guided run (NTK-LRC).

The purely random run compares more favorably to the best NTK-LRC run, with a faster training time and closer parameter count, at the expense of marginally worse accuracy than Random 2. However, it still has around 4 million more parameters than the NTK-LRC run. These results show that random search is still fairly competitive in the SpiderNet space, albeit consistently less parameter efficient and marginally worse accuracy-wise.

5.8.4 Constricted NAS versus Constricted Random

Growth Strategy	Allotted VRAM	Test Accuracy	Search Time	Total Time	Params
Min. Variance	9.5	**96.92%**	2.9	56.8	6.2
NTK-LRC	**6.5**	96.78%	5.8	36.2	**4.0**
Random 2	9.5	96.78%	-	**30.5**	**4.0**

Table 5.5 The constricted random run versus the constricted guided runs.

The three runs in Table 5.5 comprise a set of runs where overall, top-end accuracy was less prioritized in favor of time-efficiency (the Minimum Variance run) or parameter-efficiency (the NTK-LRC and Random 2 runs). The latter two perform nearly identically, with the NTK-LRC taking 5.7 hours longer due to its search time costs but requiring 3 GB less VRAM. Meanwhile,

the minimum variance run has 2.2 million more parameters, but searches very quickly and scores very highly in accuracy considering the VRAM constraints.

5.8.5 Literature Comparisons

As a final point, Table 5.6 is an extension of Table 4.17, serving to compare SpiderNet against BonsaiNet and the literature. From this, it is seen that the SpiderNet search space produces consistently excellent models that are comparable with the state-of-the-art. However, randomly generated models in the space are also quite competitive, much more so than the random Bonsai models.

Algorithm	Test Acc.	Params (M)	Search Hours
NAS-Net	95.53%	7.1	37,755
ENAS Micro-search	96.131%	38.0	7.7
ENAS Macro-search	97.11%	4.6	14.4
DARTS First-order	$97.00 \pm 0.14\%$	3.3	36
DARTS Second-order	$97.24 \pm 0.09\%$	3.3	96
NAO (w/o weight sharing)	**98.07%**	144.6	4,800
NAO (w/ weight sharing)	97.07%	**2.5**	7
ProxylessNAS	97.92%	5.7	N/A
ENAS/DARTS/NAO Random	$96.48 \pm 0.18\%$	-	-
PC-DARTS	97.43%	3.6	2.4
Bonsai SC	96.83%	3.38	11.50
Bonsai Random 1 SC	95.32%	4.54	–
Bonsai Random 2 SC	95.19%	3.68	–
Bonsai LC1	96.69%	2.64	2.53
Bonsai LC2	97.04%	7.39	2.54
Bonsai LC3090	97.07%	23.36	**1.66**
Bonsai LC2 (no prune, no growth)	97.08%	12.04	0
Bonsai LC2 (prune, no growth)	97.17%	6.63	0
Spider Min-Variance	97.08%	7.4	6.6
Spider NTK-LRC	97.13%	8.8	5.4
Spider Random 1	97.03%	12.9	–
Spider Random 2	96.96%	16.8	–
Spider Random 3	96.97%	11.1	–
Spider Random 4	96.81%	9.1	–

Table 5.6 CIFAR-10 statistics for all of the NAS models covered thus far.

5.9 ImageNet

SpiderNet was also evaluated over the ImageNet-1K dataset, following the training and data processing procedures of DARTS. A batch size of 128 and scale of 48 was used, identical to DARTS and BonsaiNet. Again, SpiderNet searched very efficiently, taking just 3.6 hours to find an architecture. While still not fully competitive with state-of-the-art models, SpiderNet was

	Top-1	Top-5	Params (M)	Search Hours	Total Hours
PNAS	74.2%	91.9%	**5.1**	2,500	-
NAS-Net-A	74.0%	91.6%	5.3	48,000	-
NAS-Net-B	72.8%	91.3%	5.3	48,000	-
Regularized Evolution	**82.8%**	**96.1%**	86.7	75,600	-
NAO (w/o weight sharing)	74.3%	91.8%	11.35	4,800	-
(all models above this line are still searching by the time SpiderNet has produced a fully-trained model)					
SMASHv2	61.38%	83.67%	16.2	36	-
DARTS Second-order	73.3%	91.3%	4.7	96	-
ProxylessNAS	74.6%	92.2%	N/A	200	-
PC-DARTS	75.8%	92.7%	5.3	91.2	-
BonsaiNet	60.46%	82.05%	8.8	10.52	418.57
SpiderNet	72.27%	90.60%	7.9	**3.62**	381.46

Table 5.7 SpiderNet performance on ImageNet

much closer; performing some tuning and trying again could potentially close the gap. However, the difference between the BonsaiNet and SpiderNet performances is indicative that SpiderNet's minimal configuration paradigm is indeed useful: the lack of available compute time by which to tune the model configurations meant each algorithm could run just once, which in turn limited the manual intuition regarding dataset best practices that could be garnered. SpiderNet, by virtue of self-defining the majority of its search configuration, managed to find a much more performant model than the heavily manually-configured BonsaiNet.

5.10 NASComp

SpiderNet was additionally evaluated over the NASComp-2022 datasets, again in such a way as to align with the competition ruleset. Here, SpiderNet was allocated 12 hours per dataset of search and train time, performing 10 mutation cycles before training for n epochs, n chosen to fully utilize the remaining allocated time. Each model had c channels, such that $64 \times c \times h_{dataset} \times w_{dataset} = 2,097,152 = 64 \times 32 \times 32 \times 32$, that is, of equivalent dimensional product to a 32 channel CIFAR-10 model. Again, no data augmentation was used to eliminate potential biases being introduced into policy selection due to insider knowledge of the particular datasets. Table 5.8 details SpiderNet's performance over the datasets.

	Dataset			
Algorithm	Sadie	Chester	Isabella	Total Score
ResNet-18	80.33%	57.83%	62.02%	200.18
NASComp-2022 Winner	**96.08%**	**62.98%**	61.42%	220.48
BonsaiNet	95.49%	61.32%	**64.92%**	**221.73**
SpiderNet	94.17%	60.71%	64.54%	219.42

Table 5.8 SpiderNet performance on NASComp-2022 datasets.

SpiderNet's performance would have placed it in 2nd place in the actual competition, and is the third best submission seen thus far over the datasets. Its implicit lack of design structure is likely a particular shortcoming of the SpiderNet algorithm in this context, as the strict time-limiting of the runtime restricts the number of mutate-and-prune cycles possible and thus the diversity of discoverable architectures. However, SpiderNet is still competitive against the state-of-the-art, which is impressive given its minimal configuration.

5.11 Future Work

There is a variety of open-questions about SpiderNet's design and search space that invite exploration. Unfortunately, many of these questions would be highly time-consuming to answer, and as such could not be explored in this work.

5.11.1 Mutation Metrics

The 11 mutation metrics explored in this work are by no means an exhaustive list of the possible ways by which to select mutations. Metrics revolving around the dynamics of pruner weights especially constitute a limitless area of exploration, and perhaps other summarization metrics over the last n epochs similar to mean or variance could produce interesting results. Additionally, the NTK-LRC results show that more complex measures can prove to be more fruitful. As such, more exploration of the mutation selection metrics, as well looking at the various correlations between different metrics and with performance could serve to further differentiate guided-mutation with random mutations.

A related question to that of mutation metric choice is the determinism of their decisions; how similar are the decisions made by the same mutation metric over different runs? This is particularly relevant to SpiderNet, given that each model starts at identical conditions. If there is an optimal mutation to make, an ideal metric would identify this mutation each time, but so far none of the mutation metrics has achieved this consistency.

5.11.2 Growth Strategies

Another area with a lot of flexibility is that of the growth strategies of the models. For example, how many cycles of pruning should be performed between each mutation stage, if any? In the case of the NTK-LRC runs, they do not require any training to calculate their mutation metrics and so could be made significantly faster by skipping inter-cycle pruning. However, removing inter-cycle pruning would lower the total number of mutations possible, as the model would only ever increase in VRAM allocation. It is possible that having more mutations provides more ability for the models to strengthen areas of concern and thus benefit performance, but alternatively it could produce models so complex as to hinder their convergence abilities.

Another consideration in this decision is the interdependence of mutation and inter-cycle pruning. That is to say, how are mutation choices affected by the per-edge operation choices

made by the pruners, and how well do the opposing forces of growth and pruning interact? While weight reinitialization (Section 5.3.3) prevents them from being entirely oppositional, judging how well they cooperate is hard.

5.11.3 Initial States and Bottlenecking

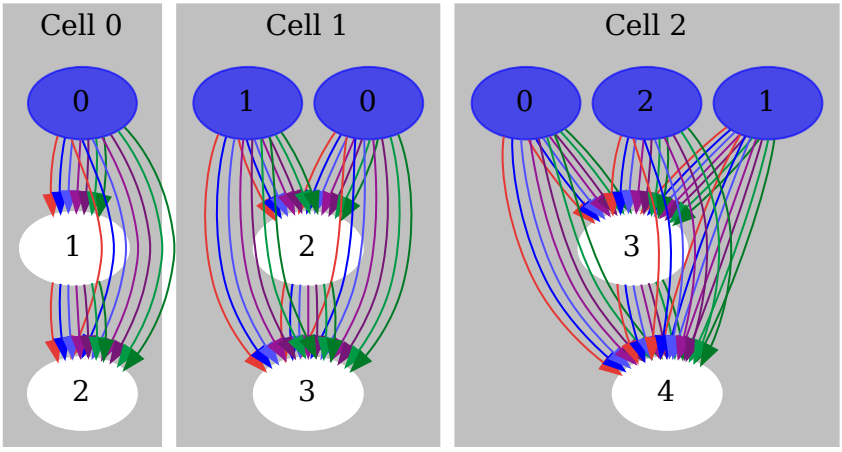


Fig. 5.12 A potential improvement to the initial state of a three cell SpiderNet model, one that eliminates bottlenecking between input-processing and input-mixing.

One observation that stems from comparing fully-grown models to their initial state (see Appendix C) is that the division of the initial cell state into the 'input-processing' section and 'input-mixing' sections (for example, in cell 1 of the initial state, edges $1 \rightarrow 2$ and $0 \rightarrow 2$ belong to the former, while $2 \rightarrow 3$ is in the latter) creates a bottleneck that triangular mutations can never eliminate. That is to say that the connection between the two cell sections is always a single node through which all information flows. For example, in Model 1 in Appendix C, Figure C.2, the bottleneck node is node 18 in cell 1, and node 16 in cell 2. While it is unclear whether this is detrimental to model performance, slightly changing the initial model state would eliminate this issue. For example, adding edges that span from each input node to the output would provide a variety of different paths from input to output, easing the bottleneck as shown in Figure 5.12.

This is only one of infinite potential initial states however, and each has the ability to drastically alter the final structure of mutated models. Studying the effects introduced by different initial states would be very interesting, to see if the differences in eventual structure translate to performance differences.

5.11.4 The False Equivalence of Random Search

The greatest point of interest amongst of all the SpiderNet results is the triangular mutation's ability to produce very high quality networks from minimal starting conditions. In this sense, this work is comparable to that of Real et al. (2018)'s Regularized Evolution, which likewise grew models from minimal conditions to state-of-the-art capabilities. Indeed, the whole concept of "minimal initial conditions" is directly inspired by the "trivial initial conditions" described in this

paper. However, what is interesting about the triangular mutation is that it can do this so quickly and consistently, without much need to carefully consider *where* it should occur. The random SpiderNet results show this clearly, as although they are all slightly worse in various metrics than the guided SpiderNet runs, they are still comparable in performance to the best BonsaiNet runs.

However, this is not to say that the guided-mutation work is all worthless, in fact, the opposite. Crucially, it revealed that comparison to random search in these idiosyncratic search spaces can create a false-equivalency. If the entire end-to-end process of finding and training a model takes as long or longer for a random algorithm than NAS, then any accuracy similarities between the two become significantly less relevant. Section 5.8.2 make this very clear; spending five hours searching is rewarded by increased accuracy, lowered VRAM, and halved parameter counts, and then the search cost is entirely refunded and more by the increased training speed. This completely erodes any argument that random search's comparable accuracy detracts from NAS; such an argument entirely ignores the three other dimensions by which NAS and SpiderNet have been shown to provide benefits.

5.12 Conclusion

This chapter first covered SHAP and NTK-LRC, background material crucial for the design of certain components within the chapter. Next, it described the design of SpiderNet, a Neural Architecture Search algorithm that evolves from minimal initial conditions. A number of mutation metrics were then introduced to guide this algorithm, and were evaluated over truncated training regimes to determine their efficacy. Additionally, a train-free mutation based around a joint Neural Tangent Kernel and Linear Region Count metric was also explored. A number of experiments were then conducted, trialling the full SpiderNet algorithm with different mutation metrics, along with a variety of ablation studies. These results demonstrated that SpiderNet produces highly-competitive state-of-the-art models, but furthermore show that SpiderNet excels against random search in a number of other dimensions beyond pure performance, namely, its superior overall runtime and efficiency.

Chapter 6

Conclusion

This thesis first covered the mechanics of neural networks: their fundamental construction, how neurons compose to form more complex functions, and how they learn. Next was the exploration of how to operate these networks, starting with how to train them, then how to identify and diagnose potential issues like over-fitting, and then some more advanced techniques to increase their performance. After that, more specialized components specifically used in the construction of convolution neural networks were discussed, as were the image datasets used to evaluate this class of model. The final pieces of the rudiments were the practicalities of neural network research, specifically the hardware and software frameworks necessary for efficient work.

With the fundamentals covered, the world of state-of-the-art convolutional neural networks was entered, starting with the networks like AlexNet, Inception, and ResNet that were formative in modern architecture design. From there, the first neural architecture search techniques like NAS-RL and Neuroevolution were examined, ones that demonstrated that automated methods could indeed build competitive architectures. However, these first approaches used huge amounts of computational power, and were impractical for real-world uses. This brought us to the second generation of NAS algorithms like ENAS and Regularized Evolution, that sought to optimize these earlier algorithms and reduce their costs. This drive towards efficiency led us to differentiable approaches like DARTS and ProxylessNAS, that made use of gradient descent to perform both architecture search and model training, an elegant and conceptually compact solution to the NAS problem. However, these results could not be discussed without also exploring the criticisms that plagued all of these approaches: a random search policy in their search space often performed equally well as these intelligent methods. This set the stage for the research questions explored in this work.

6.1 Search Space Expansions

My hypothesis as to why random search performed so well as compared to intelligent search concerned search space design bias; that the extensive research into manual architecture design over common benchmark datasets like CIFAR and ImageNet contaminated how NAS researchers built their search spaces. These ‘constricted’ search spaces were so shaped by this contaminatory

bias as to exclusively contain good models, and therefore, NAS or random search policies would perform well regardless of their decisions.

6.1.1 BonsaiNet

Testing this hypothesis first entailed building the Bonsai search space. By allowing each edge in the models to be multi-path (i.e, contain multiple mixed operations), expanding the potential cellular inputs to be any possible mixture of prior cell outputs, and removing the restriction of cellular homogeneity, the search space expanded by a factor of roughly 10^{200} as compared to the constricted spaces used in Li and Talwalkar (2019) or Yu et al. (2019). While much larger, the Bonsai space is also crucially a superset of those constricted spaces; all the good architectures are still present while dramatically increasing the probability for bad architectures to exist. Indeed, this had the desired effect of lowering the average quality of randomly selected models, with a random network from the Bonsai search space scoring a 95.19% accuracy on CIFAR-10, while a random model in the constricted space scored a 96.48%. With a lowered random search benchmark, NAS has more room by which to demonstrate its abilities, if it can truly discern quality models within a search space.

However, the size of the Bonsai search space presented its own challenges to NAS, as many of the design restrictions in the constricted space are put in place due to hardware restrictions like VRAM space. To perform NAS in the Bonsai search space, an algorithm needed to be designed that could perform effectively. This algorithm became BonsaiNet, which uses differentiable pruners and the concept of model compression to iteratively build networks. First, BonsaiNet divides a model supernet into roughly even 'sections' that are all small enough to fit uncompressed into the available GPU space. BonsaiNet then iteratively builds itself by instantiating a model section, then uses the differentiable pruners within to compress the model until the next section can be appended. This process cycles until the model has grown to the full size as specified in the configuration, at which point it can train normally.

This process provides many advantages, namely that models derived from very VRAM-expensive supernets can be trained within smaller VRAM footprints. This is evidenced by the experiments within Section 4.9.5, which found that BonsaiNet only needed 10.5 GB of VRAM to identify an optimal subnet within a 15 GB supernet. While 'only' a 30% space savings, it marks the difference between being able to run the network on an NVidia 1080Ti and needing a more expensive, more spacious GPU. Additionally, the found network was only minimally less accurate than one isolated directly from pruning the full supernet while requiring a final VRAM footprint of only 7.39 GB, which is small enough to fit on the majority of consumer GPUs. Therefore, in a search space that contained models up 15 GB large, BonsaiNet was able to find one that was only 7.39 GB yet offered equal accuracy performance to the largest models. In this sense, BonsaiNet is exceptionally capable at finding optimal, efficient models within search spaces that would ordinarily be well outside the spatial capabilities of the available hardware.

BonsaiNet's raw capabilities aside, it also consistently found models that had significant accuracy improvements over random search: the average small-cell BonsaiNet model scored a 96.65% test accuracy over CIFAR-10. This is exactly in line with performance of NAS

algorithms in the constricted space that were presented in Yu et al. (2019), which on average found models of around 96.55% test accuracy. Therefore, despite the average quality of models in the space decreasing, BonsaiNet was still able to identify the good models still present. The expanded search space did not seem to hinder the calibre of BonsaiNet's discovered models either, as they are comparable to the good models that were being discovered in the original constricted space. This is made explicitly clear in Figure 6.1:

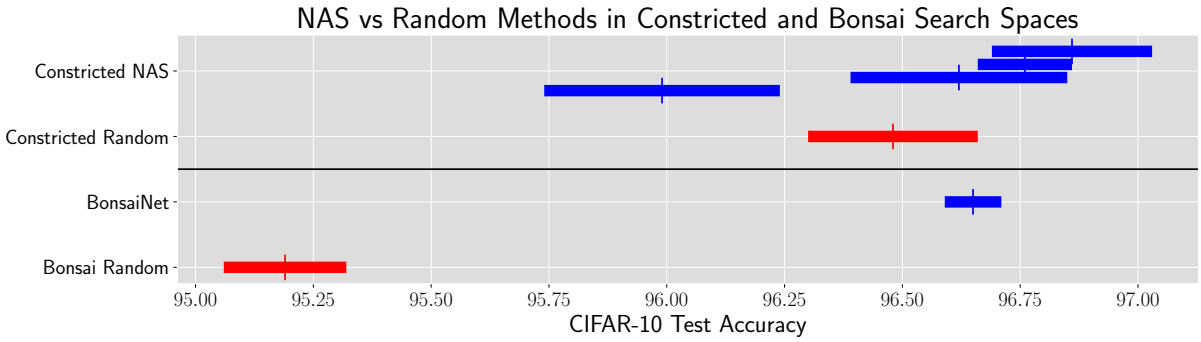


Fig. 6.1 Random and NAS models' performances on the constricted Yu et al. and Bonsai search spaces. From top to bottom, the constricted NAS entries correspond to NAO, ENAS, DARTS, and BayesNAS respectively.

This indicates that the random search hypothesis was indeed correct: random search only compares well to NAS when the search space contains exclusively good models; its comparable ability is merely an illusion caused by an over-constricted search space. In a space with greater performance variance, random search suffers while NAS is still able to identify the high-performance models.

6.1.2 SpiderNet

While BonsaiNet demonstrates NAS' ability to differentiate between good and bad models in search spaces containing models of more variable accuracies, SpiderNet revealed a different virtue that NAS can possess. SpiderNet takes the expanded search space of BonsaiNet to its logical extreme, instead operating in an infinite search space bound only by available search time. By nature of its mutations and unbounded search space, SpiderNet produces very complex, organic-looking models as compared to BonsaiNet (compare Figure A.4 to Figure C.2 for a clear example of this). Specifically, SpiderNet defines a triangular mutation, which adds a new node and two edges to an existing edge to create a new, triangularly connected system. By performing these mutations over edges in a model, the model can be dynamically grown. Differentiable pruners lying along these edges can then conversely prune the model as desired, meaning models exist in a constantly growing and shrinking search space. Different metrics were explored to guide where these mutations should occur, and found that performing mutations on edges with a minimum pruner variance or optimal NTK-LRC scores was best. Interestingly, traversing this space via random mutation often produces networks that were nearly as accurate as those produced by minimum-variance or NTK-LRC guided mutations.

However, the models produced by the guided mutations were much more time and parameter efficient than the randomly generated models. Often, the few hours that a guided-mutation search took were made up for during training, as the much more time-efficient model could train significantly faster than the random models. This erodes one of random search's predominant advantage: its lack of search cost. This is best demonstrated in Table 5.3; the guided mutations took 5.4 hours to produce a network as compared to the functionally zero cost of producing a network at random. However, the guided mutation model took 12 hours less to train than the random one, meaning the end-to-end time costs were around 6.5 hours cheaper. Additionally, the guided mutation model used half as many parameters and scored 0.1% percentage points higher accuracy than the random model.

Therefore, if NAS can be more parameter efficient and outright faster end-to-end than random search, why does it matter that random search can produce models that are *almost* as good? If random search cannot outperform NAS on raw accuracy, the only reason to use it would be its search cost advantage. However, when considering the full context of how these networks are used, the search cost constitutes a tiny fraction of the total time required to produce a fully trained model, around 7% in this example. Conceding some time in this 7% to prioritize efficiency in the remaining 93% is more than worth it. Training and evaluation costs of the networks produced by NAS is significantly more relevant to the practical costs of NAS than search cost, considering that cloud providers bill hourly. Perhaps the focus on search-time seen in literature is a holdover from the era of algorithms like NAS-RL that needed 61 years of search time to produce an architecture, in which case you might be able to get results faster by training a random network with pen and paper, but this is no longer the case. Search costs are minimal compared to training costs, and thus I argue that purely accuracy-based comparisons to random search are obfuscating the true utility of NAS.

6.2 Minimizing Configuration

Finally, as a by-product of the work on BonsaiNet and research into other NAS methods, I noticed a troubling problem with most NAS algorithms. Due to the configurational requirements (for example, needing to specify node count and depth in BonsaiNet), there is still a massive amount of manual design necessary when working with NAS. While BonsaiNet was shown to be fairly robust over configuration changes, producing competitive models across the various small and large cell domains, a significant amount of time was spent tweaking these configurations in hopes of producing even better models. This is exactly contrary to the goals of NAS, that is, to minimize the amount of tinkering necessary in the design of neural networks. As such, two developments that are presented in this thesis provide proof-of-concept towards minimizing the configurational options necessary of NAS algorithms.

First is the large-cell concept that BonsaiNet can be configured to use and SpiderNet exclusively uses. Here, a series of small cells consisting of stacking combinations of normal and reduction cells are replaced by a few reduction cells, of equal number to the total number of desired reductions in the cell. By sufficiently increasing the size of these reduction cells, all

possible stacks of the ubiquitous N-normal-cells-plus-a-reduction pattern are simply subgraphs of the large reduction cell. This means that rather than having to specify the overall cellular structure of a model, i.e, how many normal cells should be placed before and after each reduction cell, one can just specify the total number of reductions. This reduces a two-dimensional configuration space ($N_{normal} \times N_{reduction}$) to a single value, $N_{reduction}$.

Second is SpiderNet's ability to shape itself into an appropriate size as it sees fit. Since its starting architecture is defined simply by the smallest possible initial state, it does not need any configuration as to cellular size. As such, this eliminates any need to specify cellular node counts or depths, as both values dynamically change per-cell as the model mutates.

The difference these changes make can easily be seen by a thought-experiment. Imagine a NAS method that takes 50 GPU hours to produce a fully trained model, and each of its configurational parameters has three possible values. If neither large-cells nor a self-shaping method are used, there are four dimensions of configuration: $\left(N_{normal} \times N_{reduction} \times N_{nodes} \times N_{depth}\right)$. In this case, this simple 3 value grid search would cost 4,050 GPU hours to complete, just under 169 GPU days. Meanwhile, a NAS method with large-cells and self-shaping needs just a single value, $N_{reduction}$, which would mean the grid search would take just six GPU days.

SpiderNet is by no means entirely free of configuration options; it still requires the user to specify batch size, initial channel size, and of course reduction count. That said, it drastically reduces the amount of tuning necessary to produce good results, and as such is an excellent step towards the ideal of a truly configuration-free NAS algorithm.

6.3 Final Thoughts

The results presented here all bode very well for the real-world applicability of NAS. In practice, NAS would be best applied to novel problems, ones without extensive research into their best practices. The whole idea of NAS is to eliminate the need for such research in the first place. In such cases, biasing the NAS search space towards good results is impossible, at least not deliberately. It can be assumed that these 'uninformed' search spaces would likely contain both good and bad models, as what constitutes good and bad cannot be confidently known at the outset. The work in this thesis shows that NAS is capable of differentiating between said good models and bad models, and is thus well-suited to exactly this application to novel problems. Furthermore, if designed such as to prioritize this, NAS algorithms can produce highly time and parameter efficient models, meaning that using NAS can not only reduce development costs, but also long-term deployment costs. Finally, the work on a minimally-configured NAS algorithm demonstrates that NAS can define its own architectural requirements for success which minimizes the time required to tune these algorithms. As such, BonsaiNet and SpiderNet demonstrate the greatest attributes and potential of NAS: it can be an out-of-the-box solution that identifies fast, efficient, and powerful neural networks at a cost accessible to all.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283.

Anthony, M. (2001). *Discrete Mathematics of Neural Networks*. Society for Industrial and Applied Mathematics.

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2017). SMASH: one-shot model architecture search through hypernetworks. *CoRR*, abs/1708.05344.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. *CoRR*, abs/2005.14165.

Cai, H., Zhu, L., and Han, S. (2018). Proxylessnas: Direct neural architecture search on target task and hardware. *CoRR*, abs/1812.00332.

Chen, W., Gong, X., and Wang, Z. (2021). Neural architecture search on ImageNet in four GPU hours: A theoretically inspired perspective. *CoRR*, abs/2102.11535.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. *IEEE Conference on Computer Vision and Pattern Recognition*.

DeVries, T. and Taylor, G. W. (2017). Improved regularization of convolutional neural networks with Cutout. *CoRR*, abs/1708.04552.

Dong, X. and Yang, Y. (2020). Nas-bench-201: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations (ICLR)*.

Dwaraknath, R. V. (2019). Understanding the neural tangent kernel. https://rajatvd.github.io/NTK/. Accessed: 2021-07-30.

European Space Agency (2022). Sentinel online. https://sentinel.esa.int/web/sentinel/sentinel-data-access. Accessed: 2022-08-15.

Geada, R. and McGough, A. S. (2022). SpiderNet: Hybrid differentiable-evolutionary architecture search via train-free metrics. *CVPR-NAS 2022*.

Geada, R., Prangle, D., and McGough, A. S. (2020). Bonsai-Net: One-shot neural architecture search via differentiable pruners. *CVPR-NAS 2020*.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. *Proceedings of Machine Learning Research*, 15:315–323.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *ICLR*.

Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q. V., and Chen, Z. (2018). Gpipe: Efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167.

Isabella Steward Gardner Museum (2022). Isabella steward gardner music collection. https://www.gardnermuseum.org/experience/music. Accessed: 2022-08-12.

Jacot, A., Gabriel, F., and Hongler, C. (2018). Neural tangent kernel: Convergence and generalization in neural networks. *CoRR*, abs/1806.07572.

Karpathy, A. (2021). Karpathy.ai. https://karpathy.ai/. Accessed: 2021-12-29.

Kim, J., Park, C., Jung, H.-J., and Choe, Y. (2019a). Differentiable pruning method for neural networks. *CoRR*, abs/1904.10921.

Kim, J., Park, C., Jung, H.-J., and Choe, Y. (2019b). Plug-in, trainable gate for streamlining arbitrary neural networks. *CoRR*, abs/1904.10921.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. *ICLR*.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. *University of Toronto*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *NIPS*.

Krogh, A. and Hertz, J. A. (1992). A simple weight decay can improve generalization. *NIPS*.

Lambda Labs (2018). Deep learning gpu benchmarks - v100 vs 2080 ti vs 1080 ti vs titan v. https://lambdalabs.com/blog/best-gpu-tensorflow-2080-ti-vs-v100-vs-titan-v-vs-1080-ti-benchmark/. Accessed: 2022-08-15.

Larsson, G., Maire, M., and Shakhnarovich, G. (2016). Fractalnet: Ultra-deep neural networks without residuals. *CoRR*, abs/1605.07648.

LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., and Jackel, L. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, pages 541–551.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition.

Li, L. and Talwalkar, A. (2019). Random search and reproducibility for neural architecture search. *CoRR*, abs/1902.07638.

Lin, M., Chen, Q., and Yan, S. (2013). Network In Network. *CoRR*, abs/1312.4400.

Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L., Fei-Fei, L., Yuille, A. L., Huang, J., and Murphy, K. (2017). Progressive neural architecture search. *CoRR*, abs/1712.00559.

Liu, H., Simonyan, K., and Yang, Y. (2018). DARTS: differentiable architecture search. *CoRR*, abs/1806.09055.

Loshchilov, I. and Hutter, F. (2016). SGDR: stochastic gradient descent with warm restarts. *ICLR*, abs/1806.09055.

Lundberg, S. and Lee, S. (2017). A unified approach to interpreting model predictions. *NIPS*, abs/1705.07874.

Luo, R., Tian, F., Qin, T., Chen, E., and Liu, T.-Y. (2018). Neural architecture optimization. *CoRR*, abs/1808.07233.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *JMLR Workshop and Conference Proceedings*, volume 30.

Martello, S. and Toth, P. (1980). Optimal and canonical solutions of the change making problem. *European Journal of Operation Research*, 4.

Microsoft Azure (2021). Linux virtual machines pricing. https://azure.microsoft.com/en-gb/pricing/details/virtual-machines/linux/. Accessed: 2021-02-26.

Miller, B. L., Miller, B. L., Goldberg, D. E., and Goldberg, D. E. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212.

NVidia (2021). Cuda zone. https://developer.nvidia.com/cuda-zone. Accessed: 2021-09-13.

Pascanu, R., Montufar, G., and Bengio, Y. (2013). On the number of response regions of deep feed forward networks with piece-wise linear activations. *CoRR*, abs/1312.6098.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018). Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268.

PyTorch (2021). PyTorch. https://pytorch.org/. Accessed: 2021-06-21.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Network Society*.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Le, Q. V., and Kurakin, A. (2017). Large-scale evolution of image classifiers. *CoRR*, abs/1312.4400.

Ren, P., Xiao, Y., Chang, X., Huang, P., Li, Z., Chen, X., and Wang, X. (2020). A comprehensive survey of neural architecture search: Challenges and solutions. *CoRR*, abs/2006.02903.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2014). ImageNet large scale visual recognition challenge. *CoRR*, abs/1409.0575.

Samsung R&D Institute China (2021). SRC-B won the 1st place in CVPR-NAS 2021 Competition.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi and go through self-play. *Science*.

Simonyan, K. and Zisserman, A. (2015). Very deep convolution networks for large-scale image recognition. *ICLR*.

Smith, S. L. and Le, Q. V. (2017). A Bayesian perspective on generalization and stochastic gradient descent. *CoRR*, abs/1710.06451.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *JMLR*.

Stinchcombe, M. and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*.

Sumbul, G., Charfuelan, M., Demir, B., and Markl, V. (2019). Bigearthnet: A large-scale benchmark archive for remote sensing image understanding.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. *CoRR*.

Tan, M. and Le, Q. V. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946.

Torralba, A., Fergus, R., and Freeman, W. T. (2008). 80 million tiny images: a large dataset for non-parametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

UK Department for Business, Energy, and Industrial Strategy (2021). Quarterly energy prices, December 2021. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/1043434/Quarterly_Energy_Prices_December_2021.pdf. Accessed: 2021-12-30.

Walsh, F. (2020). AI outperforms doctors diagnosing breast cancer. *BBC News*.

Wright, J. (1975). The change-making problem. *ACM*, 22.

Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

Xie, S., Kirillov, A., Girshick, R., and He, K. (2020). Exploring randomly wired neural networks for image recognition. *CoRR*, abs/1904.01569.

Xu, Y., Xie, L., Zhang, X., Chen, X., Qi, G.-J., and amd Hongkai Xiong, Q. T. (2020). PC-DARTS: partial channel connections for memory-efficient architecture search. *CoRR*, abs/1907.05737.

Ye, T. (2018). Visual object detection from lifelogs using visual non-lifelog data.

Yu, K., Sciuto, C., Jaggi, M., Musat, C., and Salzman, M. (2019). Evaluating the search phase of neural architecture search. *CoRR*, abs/1902.08142.

Zoph, B. and Le, Q. V. (2017). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2017). Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012.

Appendix A

BonsaiNet Architecture Diagrams and Supplementary Algorithms

A.1 Edges

Edges (Figure A.1) take some single input and pass it through $|\mathcal{O}|$ parallel operations. The results of these operations are then individually pruned, and the edge outputs a list of these pruned operation outputs. A critical distinction to make here is that these special Bonsai edges that perform this parallel operation pruning will be represented in red, while normal graph edges are represented in black. These normal graph edges simply indicate unmodified data flow between components of the model.

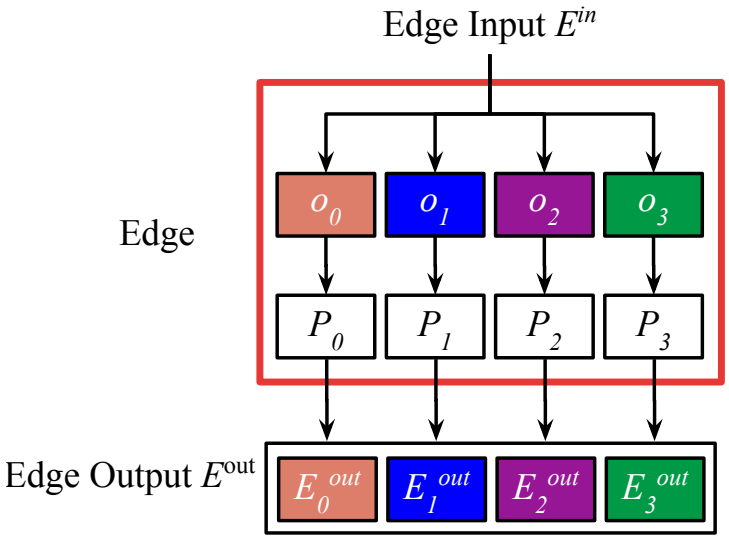


Fig. A.1 Bonsai edge where $|\mathcal{O}| = 4$.

A.2 Nodes

Nodes (Figure A.2) concatenate the output lists of each inbound node via summation and batch normalization, and then pass this concatenation on as the node output.

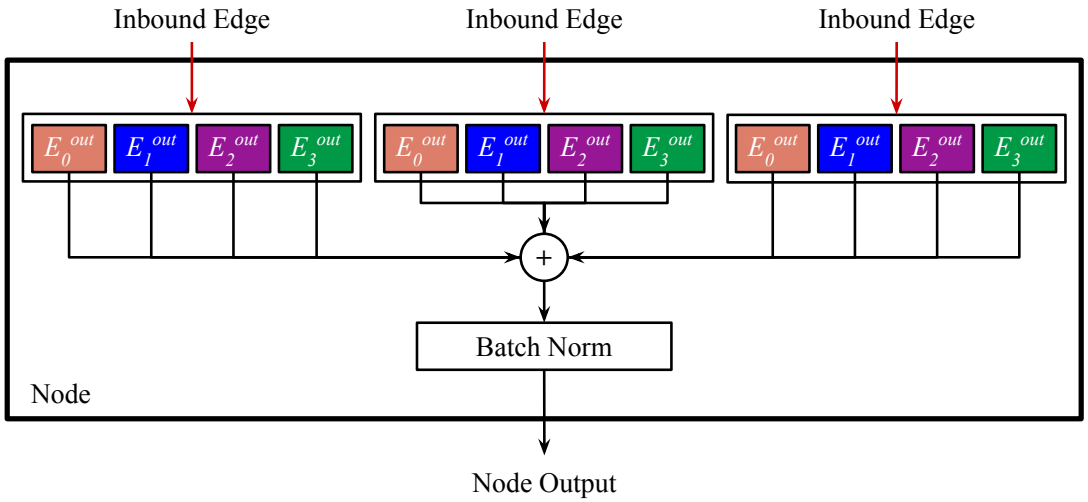


Fig. A.2 Bonsai node with three inbound Bonsai edges and $|\mathcal{O}| = 4$.

A.3 Cells

Each cell receives the output of each antecedent cell and the original model input. These inputs are processed through the cell input handler. The job of the input handler (Figure A.3) is to scale the outputs of antecedent cells to be compatible with the operations in the cell and produce two outputs. The first output is just the scaled output of the directly previous cell C_{c-1}. The second output is the pruned sum of all antecedent cells $[C_{c-1}, \ldots, C_0]$ as well as the original model input.

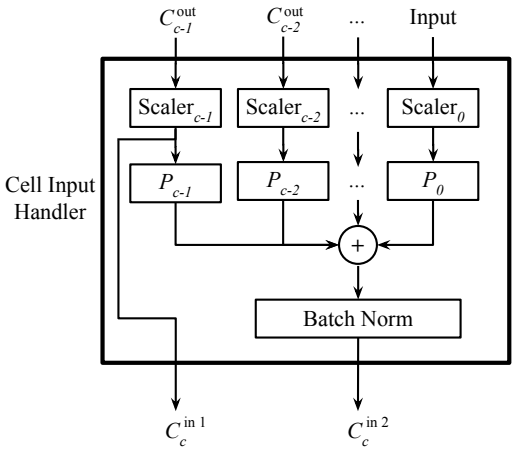


Fig. A.3 The cell input handler.

These two outputs are passed to the input nodes of the cell, called Node x and Node y. From here, each node n is connected via Bonsai edges to each node $[n+1, n+2, \ldots, n+d]$ where d is the edge depth of the model. This simply controls the width/depth ratio of the model; a depth of 1 means every node is linearly connected in serial, while a depth equal to the number of nodes means each node connects to every downstream node.

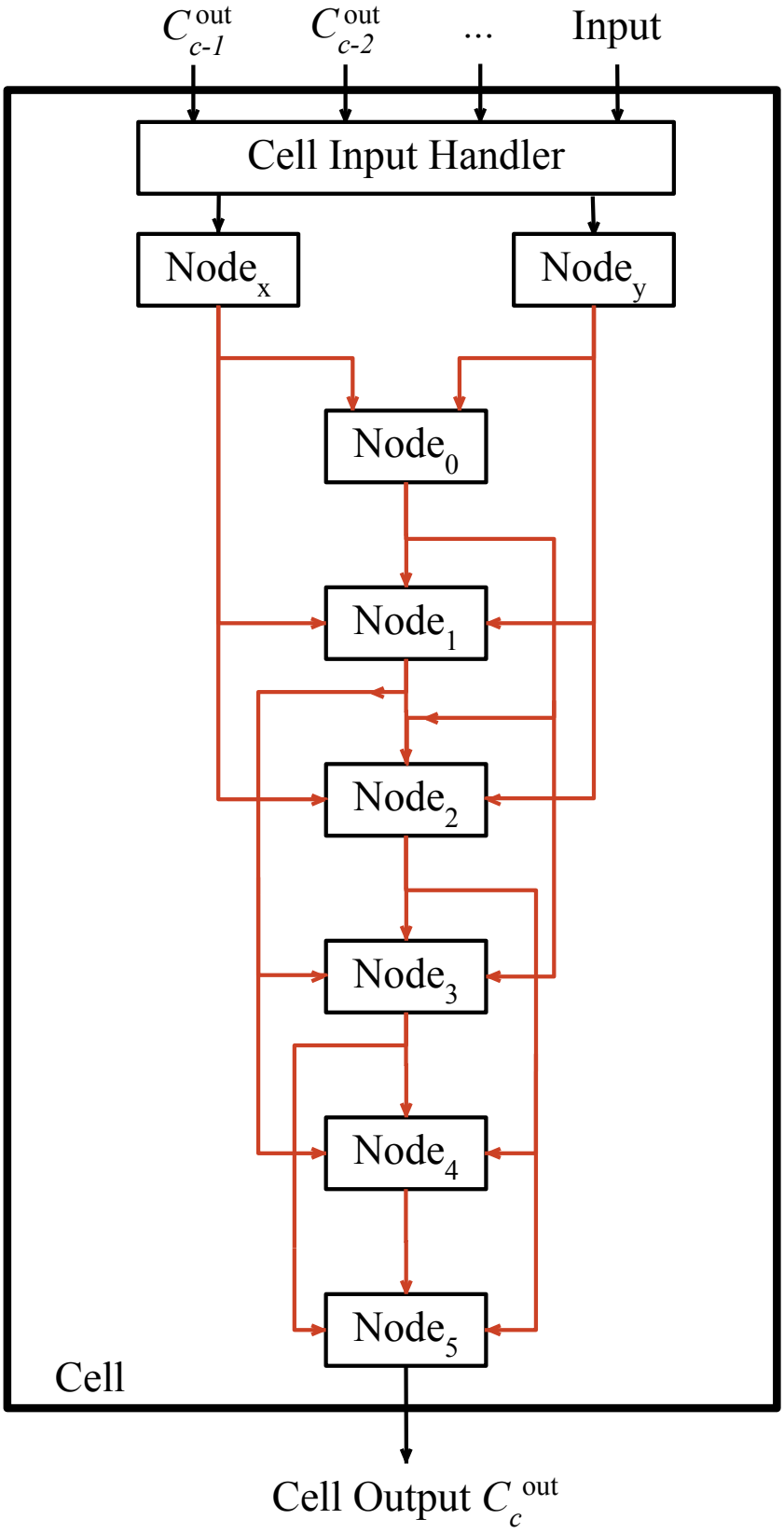


Fig. A.4 Bonsai cell with six nodes and edge depth $d = 3$. This depth means each node connects to each of the next three nodes.

A.4 Models

Bonsai models are comprised of stacked cells, where each cell receives the output of all previous cells and the original model input. Cells are added in *model sections* of *n* cells as per the Bonsai search algorithm. The progression of a model from one section to three sections, where each section consists of a single cell, is shown in Figure A.5.

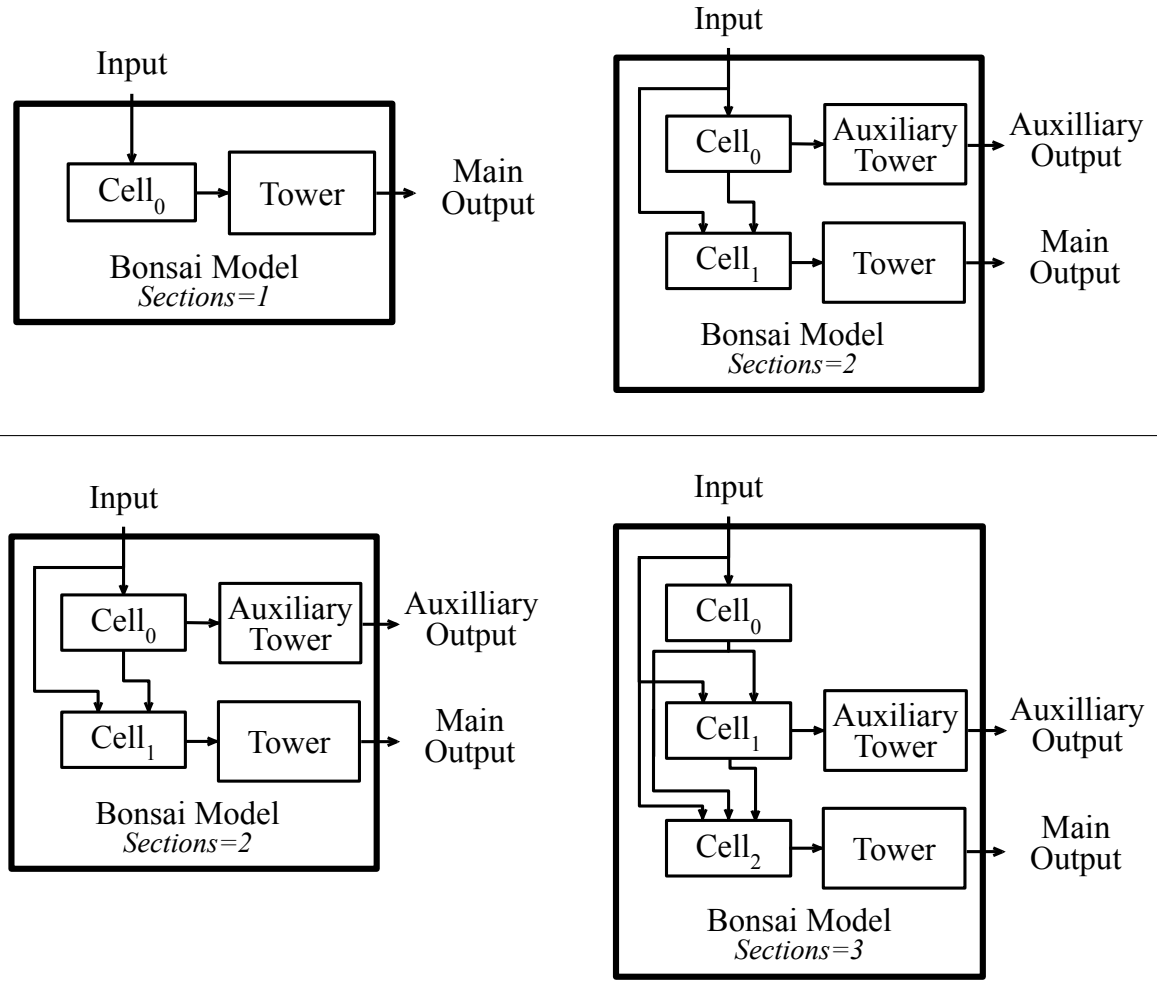


Fig. A.5 The top row shows the progression of the model from one section to two sections. Note how the old output tower from the previous section is converted into an auxilliary tower in the next. The bottom row shows the progression from two sections to three sections.

A.5 Implementation Details

This section will discuss a few notes on how these algorithms are practically implemented within PyTorch. PyTorch was chosen as the framework for this implementation specifically due to its *dynamic computational graph*, that is, the graph structure it uses to determine the forwards and backwards propagation of data and gradients through the model is dynamically constructed each pass. This is in contrast to Tensorflow's static computational graph, which is determined once at initialization. The dynamic graph allows the network connectivity to be modified in-place with ease during model training, a fundamental necessity for pruning and deadheading.

A.5.1 Working with Dynamic Parameter Counts

An unforeseen snag of the Bonsai algorithm is the need for the parameter count of the model to change over time. Deadheading drops a number of parameters from the model, while the addition of model sections can often multiplicatively increase the total number of model parameters. The PyTorch optimizer needs to keep track of which parameters need to be differentiated, which have been deleted, and which have been added, otherwise the backpropagated changes will only take effect on the initial parameter configuration of the model.

Ensuring that the optimizer is aware of the parameters removed by deadheading is relatively straightforward; each pruner carries a `switch_off` method that can be called during the deadheading process. If the pruner is deadheaded, the `switch_off` method marks the pruner's weight parameter's `require_grad` flag to `False`, which indicates to the Torch optimizer that this parameter need not be updated during backpropagation. Furthermore, if an operation is deadheaded, it is excluded from the forwards calculation of the model. This removes the operation from the computation graph, which means that the backwards pass that the optimizer performs through the graph never comes across the deadheaded operation. In combination, setting the flag and removing the operation from the computation graph ensures that the optimizer can skip the deadheaded operation, which speeds up the forwards-backwards passes of the model after each deadheading.

Tracking new parameters is slightly trickier. While one approach is to collect all the new parameters from an added section and simply append it to the optimizer, those parameters would be adjusted according to the current learning rate. Since BonsaiNet models are trained according to a cosine annealing learning rate, sections added at later stages of training would then see their first training under a significantly lower learning rate then the earlier sections began training with. The question is, is that a desired behavior? My view is that since these new parameters are arriving into the model at a random initialization, each set of new parameters starts about roughly equally 'far' from optimized. Therefore, using the global, already annealed learning rate for a set of new parameters will mean they will learn more slowly than the original model parameters were able to. Instead, each new set of parameters P added to the model tracks the

epoch T^P_{added} at which they were added to their model, and sets their annealing rate accordingly:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\left(\frac{T_{curr} - T^P_{added}}{T_{max} - T^P_{added}}\right)\pi\right)\right), \tag{A.1}$$

where T_{curr} is the current, global model training epoch and T_{max} is the desired total epoch count. Therefore at the final epoch T_{max} each set of parameters will be trained with a learning rate of η_{min}, regardless of when they were added to the model. In practice, most Bonsai models get to their full size within a few dozen epochs, so the epoch and thus annealing schedule difference between the original and final parameters is relatively minimal. However, under certain configurations like the small cell case (see Section 4.8.1) the model spends quite a while pruning to accommodate for large sections, separating the original and final sections by over 100 epochs. In this case the learning rate difference between the two is significantly more noticeable, and it would likely be detrimental to train the new sections at the earlier sections' much lower learning rate. As such, the separate per-section learning rate ensures that the training of each section is consistent throughout the model. However, further experiments would be necessary to definitively confirm this.

A.5.2 Pruner Implementation

While the extensive usage of pruners through the supernet means there is a tremendous amount of architectural flexibility, it does mean that the pruner calculation functions are called very frequently and as such need to be made as efficient as possible. To optimize the pruner function requires creating gradient-safe tensor versions of equations 4.8 and 4.9 with the minimum number of function calls. *Gradient-safe* refers to ensuring that the gradient of the proposed tensor operation has the expected gradient throughout the reasonable working range of the operation, and its output and gradient are always valid (finite and non-null). This is crucial because an invalid output or gradient from a gradient-unsafe operation will cause every subsequent operation in the network to produce invalid values, which then spread to the rest of the network in backpropagation. This exemplifies the danger of gradient-unsafe operations, as a single invalid output or gradient will rapidly corrupt an entire model.

Through experimenting with a variety of gate implementations, the following was chosen as having the minimum number of underlying CUDA function calls and fastest execution time while remaining gradient safe:

$$G(w) = \texttt{boolean}(w > 0) \tag{A.2}$$

Since Torch computes gradients discretely, the computed gradient for all values of w is uniformly 0; it is impossible to sample a point that actually lies on the $w = 0$ discontinuity of this particular function.

The next focus is the saw implementation. This needs to oscillate imperceptibly around 0 with a constant gradient of 1, while remaining strictly greater than 0. This second criterion is

important to make sure that the sign of the incoming tensor is never flipped, such as to ensure the pruner is as transparent as possible. One possible implementation mirrors the equation given in the original Differentiable Pruner paper (Kim et al., 2019a):

$$S(w) = \frac{Mw - \lfloor Mw \rfloor}{M} \tag{A.3}$$

This "raw" implementation is gradient-safe and performs mathematically correctly. However, it consists of four discrete operations (the multiplication of M by w, the floor operation, the subtraction, and the final division), which entails four total tensor initializations to store the intermediate calculations of the operation as well as a high number of CUDA function calls. As previously stated, the saw is in the absolute innermost loop of the model execution, and thus is a crucial area for optimization. As such, the following optimized implementation was designed:

$$S(w) = w \pmod{\frac{1}{M}} \tag{A.4}$$

This is mathematically identical to the original raw implementation, but makes use of the in-built CUDA function for computing moduli called `remainder`. The theoretical advantage here is that `remainder` is CUDA-native, meaning its forward and backward calculation are already optimized on a compiler level. The function only involves a single CUDA operation, meaning there is only a single new tensor that needs to be initialized which should save some VRAM space compared to the previous implementation. Additionally, since $\frac{1}{M}$ is constant throughout the lifespan of the pruner it can be precomputed at initialization, further reducing the runtime. Table A.1 compares the performance of the two implementations:

		Runtime per 100 calls, ms		
Saw	CUDA Functions	Forward	Forward+Backward	VRAM
Raw	2 mul, 2 floor, 1 sub, 4 tensor init	13.3	129.1	1200 Kb
Optimized	1 remainder, 1 tensor init	2.8	18.3	100 Kb

Table A.1 Comparison of raw and optimized saw implementations. CUDA function calls and runtimes determined via the PyTorch profiler, while VRAM allocations were measured in a process similar to that of Bonsai operation sizing detailed in Algorithm 1.

Checking runtime across both forward and forward-backward use cases is very important, specifically to check that any performance advantage is present across both use cases. While only checking the forward pass is the easier test to perform, the forward-backward use case is much more representative of the real-world application of any particular operation. This is due to heavy reliance on the backwards operation during SGD training, and the cost of the backwards operation being entirely independent of the cost of the forward operation. There are many times in the development of this optimizations wherein two candidate functions were compared, and while one was faster than the other in the forward use case, its backward operation was orders of magnitude slower and thus completely cancelled out the forward advantage. Operations like tensor indexing and the `sign` function are two such examples of these deceptive functions. The

remainder operation thankfully does not exhibit such deceptiveness, with results showing that the optimized saw runs around seven times faster than the raw saw in both use-cases. Furthermore, the need to initialize only a single tensor for the optimized saw results in a 12× VRAM savings as compared to the raw saw.

A.6 Random Search Algorithms

Algorithm 7: Random 1

Given some BonsaiNet model $M^{[0,1,\ldots,n]}_{[c_0,c_1,\ldots,c_n]}$ to emulate;
Initialize an empty model R;
for *i in* $[0, n-1]$ **do**
    Fill model section R^i with operations at random until it reaches compression c_i;
    Add the model section to the random model: $R = R + R^i_{c_i}$;
Add the final section uncompressed: $R = R + R^n_1$;
Train the resultant model $R^{[0,1,\ldots,n]}_{[c_0,c_1,\ldots,1]}$ **with pruning** and according to the same hyperparameters used for M;

Algorithm 8: Random 2

Given some BonsaiNet model $M^{[0,1,\ldots,n]}_{[c_0,c_1,\ldots,c_n]}$ to emulate;
Initialize an empty model R;
for *i in* $[0, n]$ **do**
    Fill model section R^i with operations at random until it reaches compression c_i;
    Add the model section to the random model: $R = R + R^i_{c_i}$;
Train the resultant model $R^{[0,1,\ldots,n]}_{[c_0,c_1,\ldots,1]}$ **without pruning** and according to the same hyperparameters used for M;

Appendix B

BonsaiNet Configurations

B.1 CIFAR-10 Small Cell

- **GPU Space**: 10.5, the maximum VRAM space of an NVidia 1080Ti
- **Scale**: 36, taken from DARTS
- **Nodes**: 4
- **Depth**: 4
- **Sections**: [[N, N], [R], [N, N], [R], [N, N]], taken from DARTS
- **Operations**: Same as DARTS: Identity, 3x3 Max Pool, 3x3 Average Pool, 3x3 Separable Convolution, 5x5 Separable Convolution, 3x3 Dilated Convolution, 5x5 Dilated Convolution

B.2 CIFAR-10 Large Cell

- **GPU Space**: 10.5, the maximum VRAM space of an NVidia 1080Ti
- **Scale**: 64
- **Nodes**: [8, 12, 12]
- **Depth**: [4, 4, 4]
- **Sections**: [[N], [R], [R]]
- **Operations**: Same as DARTS: Identity, 3x3 Max Pool, 3x3 Average Pool, 3x3 Separable Convolution, 5x5 Separable Convolution, 3x3 Dilated Convolution, 5x5 Dilated Convolution

B.3 CIFAR-10 Type-2 Large Cell

- **GPU Space**: 10.5, the maximum VRAM space of an NVidia 1080Ti
- **Scale**: 64
- **Nodes**: 7

- **Depth**: 3
- **Sections**: [[N], [R], [R]]
- **Operations**: Same as DARTS: Identity, 3x3 Max Pool, 3x3 Average Pool, 3x3 Separable Convolution, 5x5 Separable Convolution, 3x3 Dilated Convolution, 5x5 Dilated Convolution

B.4 CIFAR-10 3090 Large Cell

- **GPU Space**: 20, the maximum VRAM space of an NVidia 3090
- **Scale**: 96
- **Nodes**: 7
- **Depth**: 3
- **Sections**: [[N], [R], [R]]
- **Operations**: Same as DARTS: Identity, 3x3 Max Pool, 3x3 Average Pool, 3x3 Separable Convolution, 5x5 Separable Convolution, 3x3 Dilated Convolution, 5x5 Dilated Convolution

B.5 ImageNet 3090 Large Cell

- **GPU Space**: 20, the maximum VRAM space of an NVidia 3090
- **Scale**: 48
- **Batch Size**: 128
- **Nodes**: 7
- **Depth**: 4
- **Sections**: [[N], [R], [R]]
- **Operations**: Same as DARTS: Identity, 3x3 Max Pool, 3x3 Average Pool, 3x3 Separable Convolution, 5x5 Separable Convolution, 3x3 Dilated Convolution, 5x5 Dilated Convolution

Appendix C

SpiderNet Found Architecture Diagrams

In this appendix, a few examples of SpiderNet architectures are presented to demonstrate the immense range of structural designs afforded by the algorithm.

C.1 Initial State

For reference, all SpiderNet models start in the following state.

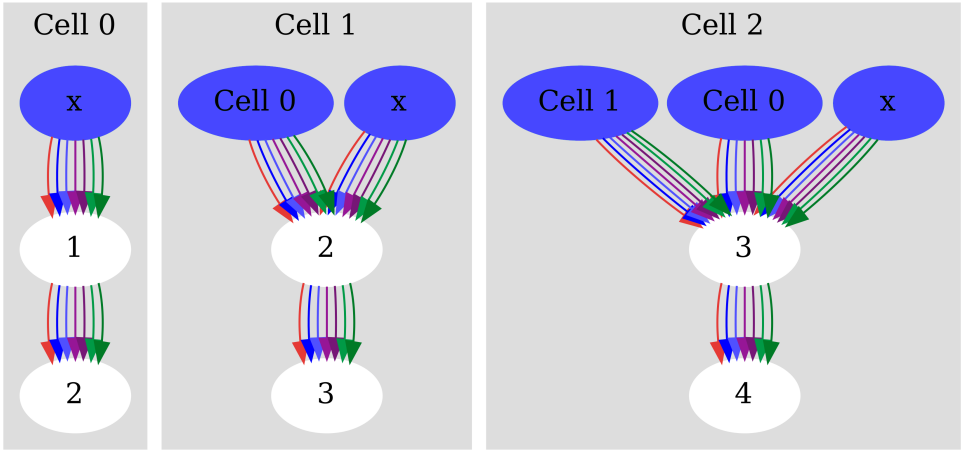


Fig. C.1 The initial state of a three cell SpiderNet model, with the input nodes marked in blue. The different colors of edges represent different operations, following the same operational color scheme used throughout this work.

C.2 SpiderNet Models

Depicted in this section are a few SpiderNet models, generated via the Random 2 method described in Section 5.6.4. All models are displayed in the same format as the initial state diagram, with input nodes in blue and edges color-coded by operation.

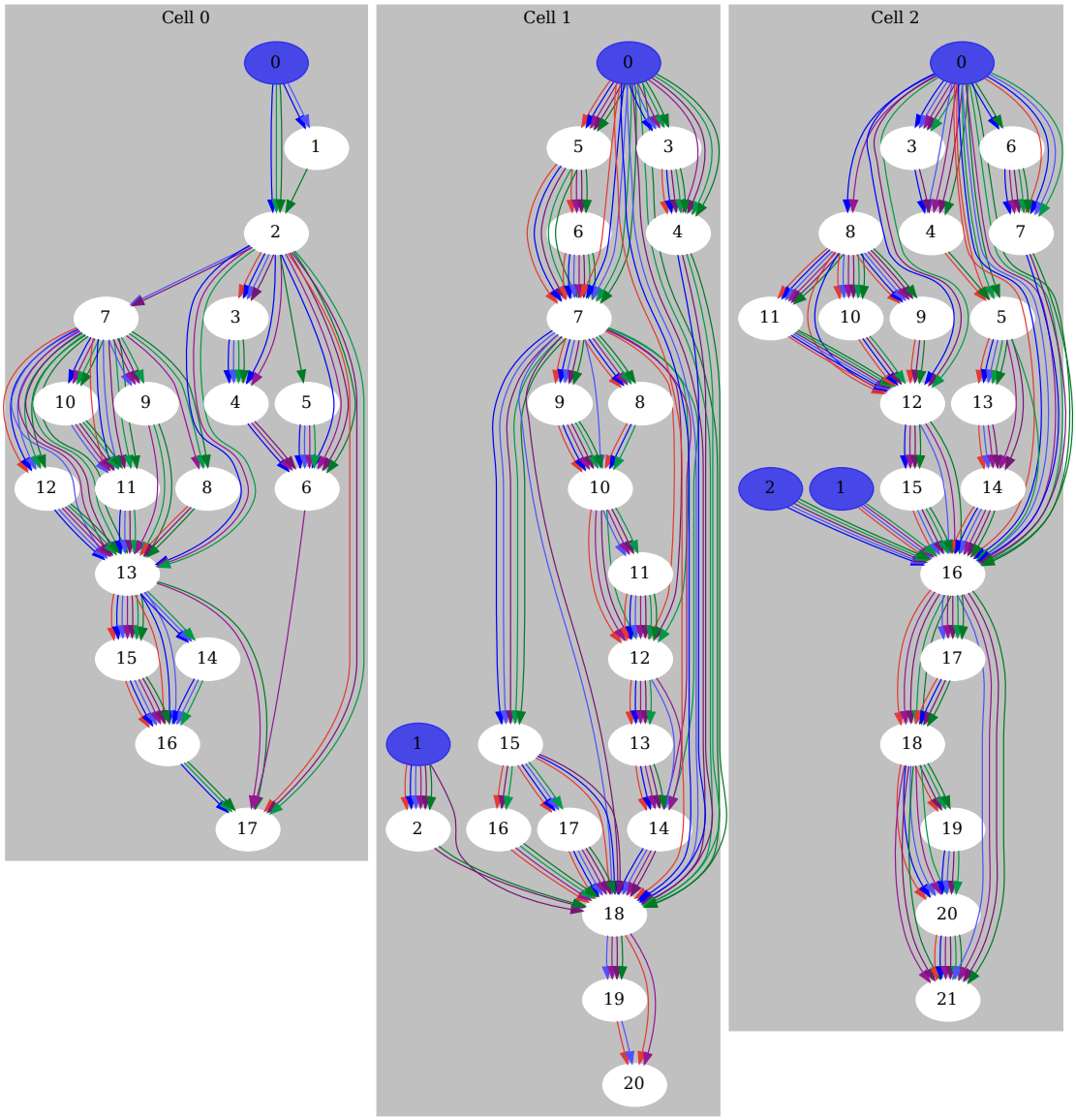


Fig. C.2 Model 1. Notice the bottlenecking produced by node 18 in cell 1, and node 16 in cell 2.

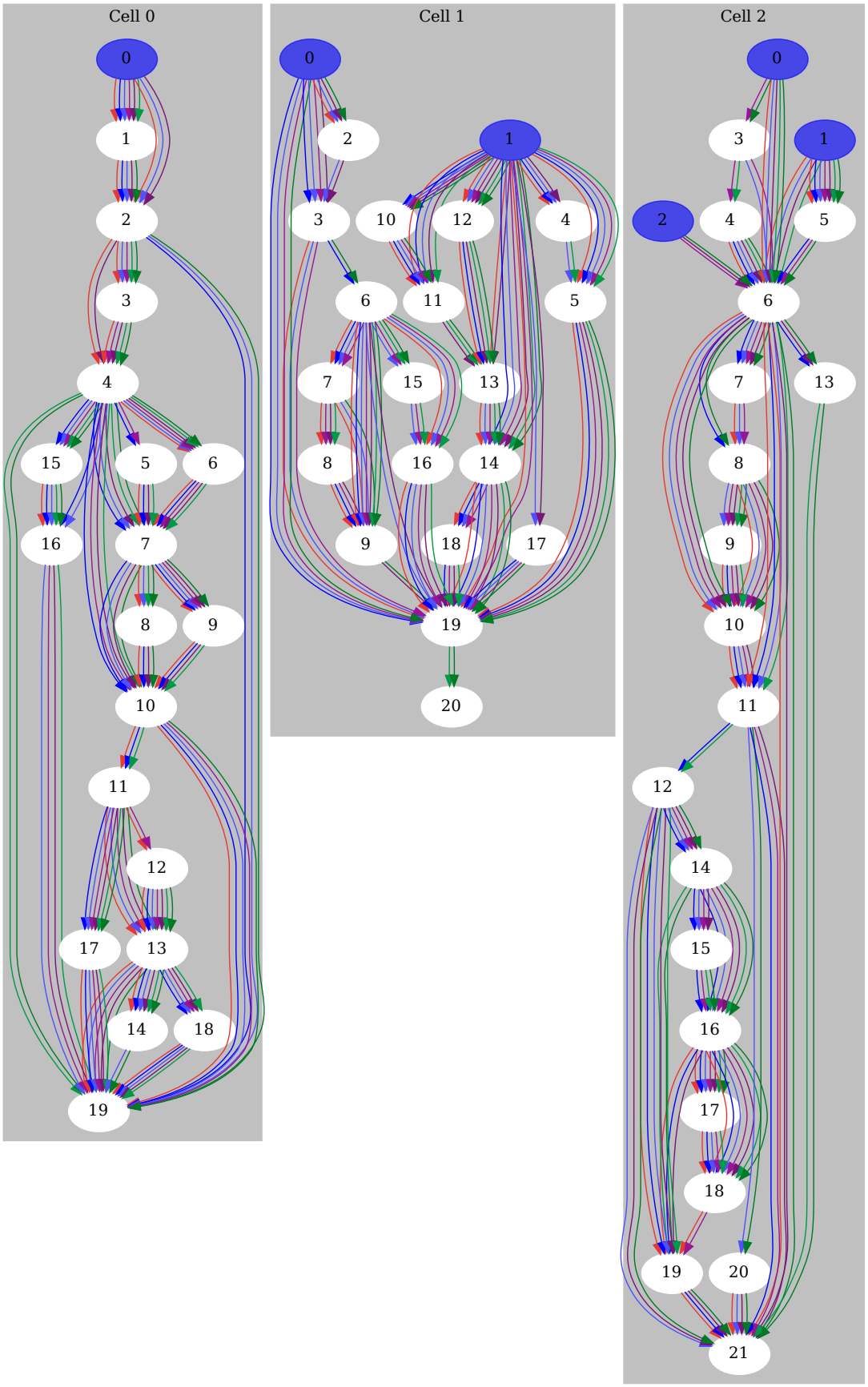


Fig. C.3 A second randomly grown SpiderNet model.

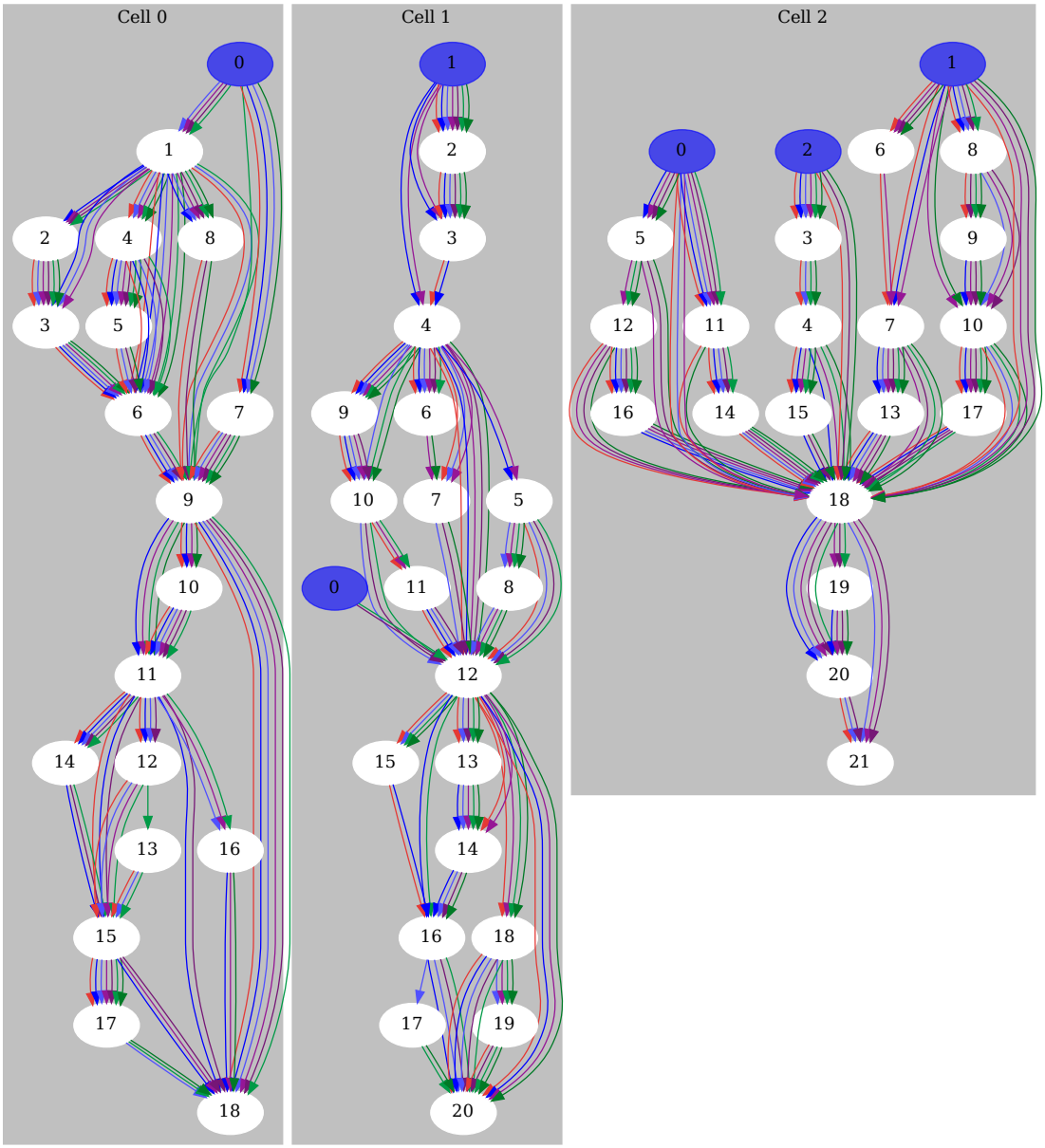


Fig. C.4 A third randomly grown SpiderNet model.

Appendix D

CVPRNAS NAS Competitions

D.1 Introduction

While looking at ways to evaluate BonsaiNet and SpiderNet, it began to become of interest to consider what exactly constitutes a ‘good’ NAS algorithm. In literature, NAS algorithms are almost exclusively evaluated over well-known benchmark datasets like CIFAR-10 and ImageNet. By this metric, the best NAS algorithms are the ones that do best over these specific datasets; these are the algorithms that get extensively cited and are used as measuring sticks for subsequent algorithms. This becomes a positive feedback loop as good performance on these specific datasets becomes more and more synonymous with ‘good NAS algorithm’, which in turn raises the performance necessary for publication. The work to evalute BonsaiNet and SpiderNet was often guilty of the same problem; the focus frequently narrowed to just these benchmark datasets, and the original aim of NAS was forgotten. For the longest time the mantra was “97%+ on CIFAR-10 and it is publishable", without considering whether the broader scope. The entire NAS field had boiled down to a leaderboard, where the only metric of importance was performance on the two computer vision benchmarks.

However, the real world use case of NAS was still a pressing concern; NAS is for novel problems and tasks, where the best practices were unknown. It is for problems where the cumulative decades of research might not apply, and when there is not much time to spend researching potential candidates. This disconnect between the answers to the questions of *“What makes a good NAS algorithm?”* and *“How should a NAS algorithm be used?”* became glaringly, unavoidably obvious. Even worse, that disconnect gave rise to a more troubling question: Do NAS algorithms work in the real world, or are they simply tailored to these existing datasets?

With that realization, it was important to devise a way to bring the former two questions into agreement, and find an answer to the latter; and this process took the form of two NAS competitions held at CVPR-NAS 2021 and 2022.

D.2 Competition Design

The general idea of the competition was to separate the design of a NAS algorithm from the datasets it will run over, such that the search algorithm design was not specifically tailored to the dataset it would run over. To do this, competitors were asked to design components of a NAS pipeline, the specific breakdown of which is shown in Table D.1.

Component	NASComp-2021	NASComp-2022
Data Preprocessing	Fixed	User-designed
Data Augmentation	Fixed	User-designed
Search Algorithm	User-designed	User-designed
Training Policy	Fixed	User-designed

Table D.1 The NASComp pipeline components for the 2021 and 2022 editions of the competition, broken down by whether they were supplied by us, the organizers (fixed) or by the competitor (user-designed).

The user-submitted components would be plugged into the server-side NAS pipeline, which would then pass each server-side dataset through the pipeline, thus searching for and training one model per dataset. The datasets were kept secret and permanently server-side, meaning that competitors had no idea the contents of the datasets that they were being evaluated over.

Competitors were then scored on how their best test accuracy compared against a ResNet-18:

$$\text{Dataset Score 2021} = 10 \cdot \frac{\text{Accuracy} - \text{ResNet accuracy}}{100 - \text{ResNet accuracy}} \tag{D.1}$$

$$\text{Dataset Score 2022} = 10 \cdot \max\left(\frac{\text{Accuracy} - \text{ResNet accuracy}}{100 - \text{ResNet accuracy}}, -1\right) \tag{D.2}$$

Therefore, a perfect 100% test accuracy was worth 10 points, a test accuracy equivalent to ResNet-18 was worth 0 points, and negative points were scored for test accuracies lower than ResNet-18. The final submission score was the sum of all individual dataset scores. This motivated trying to balance performance across all n datasets, as good performance in one dataset could be wiped out by poor performance in another.

D.3 Datasets

Crucial to the conceit of the competitions was the creation of novel datasets, such that no "best-practices" existed for any of the datasets involved in the competition, meaning competitors could not necessarily submit something like a ResNet and expect high quality results. Furthermore, it meant that since the datasets were entirely unknown and unseen, competitors could not tailor their submissions to the particular nuances of the datasets. To reduce the scope of the task being asked of the competitors, each of the datasets used for the competition was structured as image classification tasks, that is, given some four-dimensional tensor of size [batch, channel, height, width], classify each image in the batch into one of n classes. Competitors were given

a per-dataset metadata file that contained minimal information regarding the dataset, namely, its shape, total number of images, number of classes in the classification problem, and the benchmark ResNet-18 score.

D.4 2021 Datasets

Six datasets were used or developed for the 2021 edition of the competition, divided into 3 *development* datasets and 3 *evaluation* datasets. The former were given to competitors directly to test their algorithms with, while the latter were held secret until after the competition.

D.4.1 AddNIST

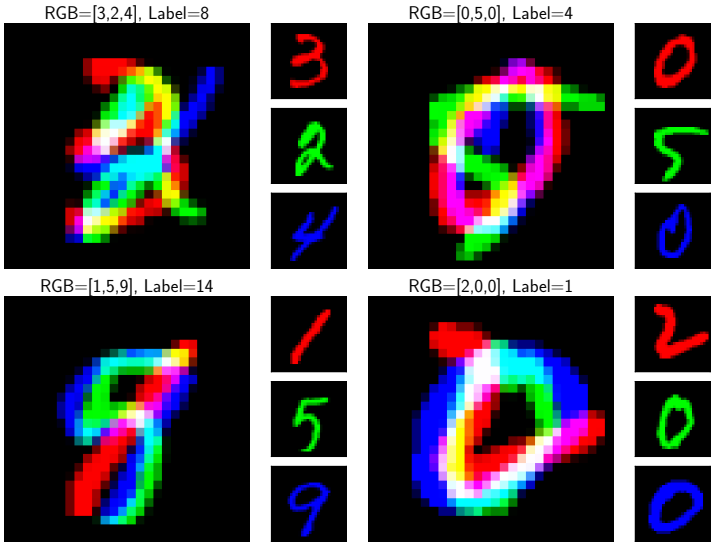
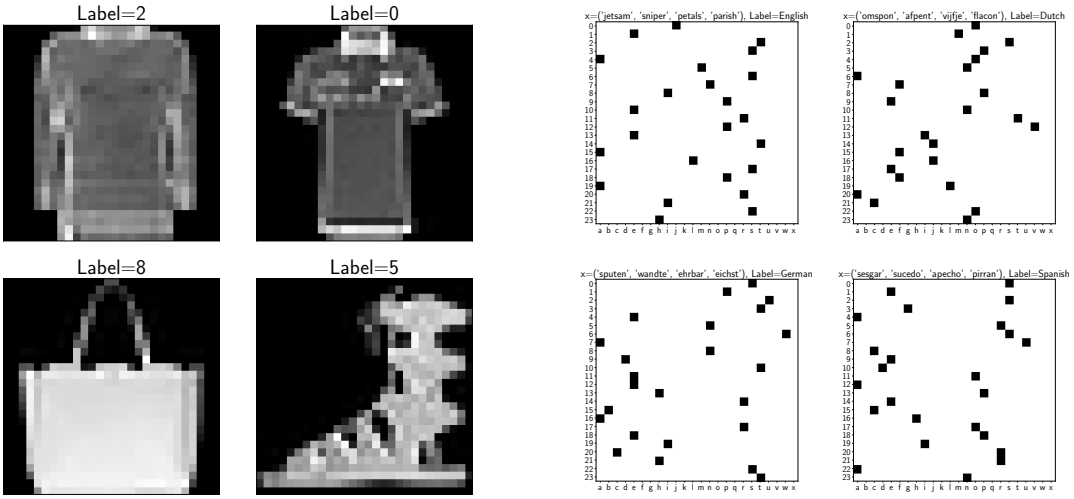


Fig. D.1 An example of the AddNIST data

The first development dataset was called AddNIST. Each image contains three MNIST images, one in each of the RGB channels. The output class is $(r+g+b)-1$, where r, g, and b refer to the digits represented by the three MNIST images. The three images are chosen such that $(r+g+b)-1<20$. The train and validation split use Torch MNIST train images, while test split uses test images. Combinations are chosen at random, but weighted such that each of the 20 classes are evenly represented in both data splits. Each of the three component MNIST images are normalized according to the MNIST mean and standard deviation before they are combined into the 3 image datapoints. See Figure D.1 for examples. Our benchmark model scored a 92.08% on this dataset, and the submission record was 95.06%, achieved by Atech AutoML.

D.4.2 FashionMNIST

The second development dataset was just FashionMNIST(Xiao et al., 2017), see Figure D.2a for examples. Our benchmark model scored a 92.87% on this dataset, and the submission record was a 94.44%, achieved by Atech AutoML.



(a) FashionMNIST data

(b) Language identification data

Fig. D.2 The FashionMNIST and Language datasets.

D.4.3 Language

The third development dataset was codenamed language. Here, we loaded the Aspell language dictionary for 10 languages that all use the Latin alphabet: English, Dutch, German, Spanish, French, Portuguese, Swedish, Zulu, Swahili, and Finnish. We then filtered these words to only those that use 6 letters total. Of these six letter words, all that used diacritics (letters such as é or ü), y, or z were removed, meaning there were 24 possible letters within each word. These words were then combined into random groups of 4, and one hot encoded. This creates a 24x24 matrix, which constitutes the input image. The six letter words were divided into train and test groups to prevent train/test leakage, meaning that there were no words shared across the train and test word sets used to generate the train and set images. The image class refers to the original language that the four words come from. See Figure D.2b for examples. Our benchmark model scored an 87.00% on this dataset, and the submission record was 89.71%, achieved by ‘yonga‘. We were surprised by how well models could learn this data, given how random it looks to the human eye.

D.4.4 MultNIST

The first evaluation dataset was codenamed MultNIST. The process of this is very similar to AddNIST, except the output class is $(r * g * b) \mod 10$; the last digit of the product of ‘r‘, ‘g‘, and ‘b‘. All other processing is identical to that of AddNIST. Our benchmark model scored a 91.55% on this dataset, and the submission record was 95.45%, achieved by Atech AutoML.

D.4.5 CIFARTile

Fig. D.3 An example of the CIFARTile data

The second evaluation dataset was codenamed CIFARTile. This takes images from CIFAR-10 and tiles them into a 2x2 grid. The label for the grid is the total number of discrete classes in the tiling minus 1 (to ensure that the classes are ‘0, 1, 2, 3‘]. So for example, a tile of ‘[horse, horse, frog, cat]‘ has three discrete classes and thus has a label of 2. The train and validation data splits get images from the train set of CIFAR-10, while the test data split gets images from the test set. For each grid, a total number of classes ‘nclasses‘ is chosen between 1 and 4. If ‘nclasses‘ is:

1. 1 class is selected at random and all four images in the tile are from that one class.
2. 2 classes are selected at random, and there are two images of each class in the tile, placed randomly into the tiling.
3. 3 classes are selected at random. There are two images of the first class and one of the second and third, with the images placed randomly into the tiling.
4. 4 classes are selected at random. There is one image of each class, placed randomly into the tiling.

Each individual image in the tile is processed as per the recommended CIFAR-10 augmentation and normalization policy used in the PyTorch documentation: a 32 pixel random crop with padding 4, a random horizontal flip, and a normalization around the global channel mean and

standard deviation. See Figure D.3 for examples. Our benchmark model scored a 45.56% on this dataset, and the submission record was 73.08%, achieved by SRCB VC Lab.

D.4.6 Gutenberg

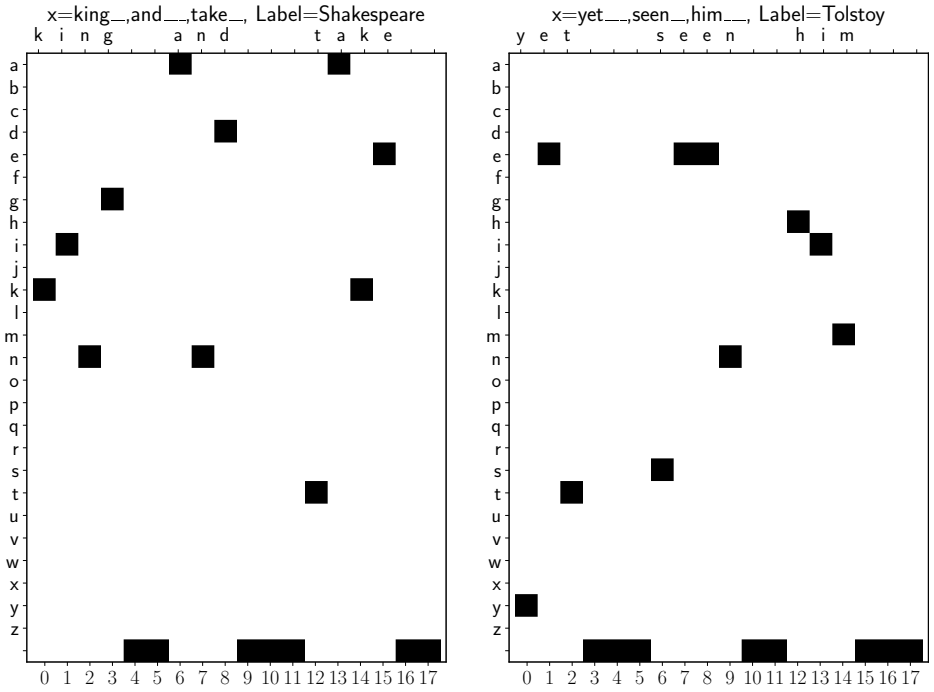


Fig. D.4 Examples of the Gutenberg data

The third evaluation dataset was codenamed Gutenberg. Here, the following texts were downloaded from Project Gutenberg (CITATION) from six different authors:

- **Thomas Aquinas**: Summa I-II, Summa Theologica Part III, On Prayer and the Contemplative Life
- **Confucius**: The Sayings of Confucius, The Wisdom of Confucius
- **Hawthorne**: The Scarlet Letter, The House of the Seven Gables
- **Plato**: The Republic, Symposium, Laws
- **Shakespeare**: Romeo and Juliet, Macbeth, Merchant of Venice, King Lear, Twelfth Night
- **Tolstoy**: War and Peace, Anna Karenina

Each text was an English translation of the source material, with authors chosen to represent a wide variety of cultures, time periods, and languages. From each text, basic text preprocessing is performed; removing punctuation, mapping diacritics to their base letters, and removing common 'structure' words (things like "chapter", "scene" or "prologue". The texts were then split into sequences of words. From these word sequences, consecutive sequences of three words that were between 3 and 6 letters long were extracted, called "phrases". Phrases that appeared in multiple

authors' corpuses were removed. Each word in each phrase were padded with underscores if they were shorted than 6 letters. For example, for Shakespeare, you might have something like `such__sweet_sorrow` or `lady__doth__protest`. These phrases were then one-hotted, and the label corresponds to the original author that wrote the phrase. See Figure D.4 for examples. Our benchmark model scored a 40.98% on this dataset, and the submission record was 50.85%, achieved by Atech AutoML.

D.5 2022 Datasets

For the the 2022 competition, the five novel 2021 datasets (AddNIST, MultNIST, Language, Gutenberg, and CIFARTile) were used as development datasets, and three new evaluation datasets were designed.

D.5.1 Sadie

The first of the 2022 evaluation datasets was codenamed Sadie, as reference to the satellite imagery the dataset was derived from. The Sadie dataset used the BigEarthNet dataset (Sumbul et al., 2019) as a foundation, but does not use the original land-cover classification labels provided alongside the dataset. Instead, Sadie takes advantage that the BigEarthNet patches are taken from images of 10 distinct European countries (Austria, Belgium, Finland, Ireland, Kosovo, Lithuania, Luxembourg, Portugal, Serbia, Switzerland) to instead modify the task into a country identification problem. This is done by identifying the Sentinel (European Space Agency, 2022) patch from which the BigEarthImage was cropped, and then cross-referencing the coordinates of that patch with a map of Europe. Following this process, 10,000 images from each country were extracted from BigEarthNet, see Figure D.5 for examples. The ResNet-18 benchmark scored 80.33% accuracy, and the competition record of 96.08% was attained by the Sun Yat-sen University team.

D.5.2 Chester

The second evaluation dataset was codenamed Chester, as reference to the chess games from which the dataset was sourced. Here, all recorded games from Chess.com that involved one of 8 grandmasters (Bobby Fischer, Garry Kasparov, Magnus Carlsen, Viswanathan Anand, Hikaru Nakamura, Anatoly Karpov, Fabiano Caruana, and Mikhail Tal) were scraped. From each of those games, the final 15% of board states were extracted, where the nth board state refers to the particular positioning of pieces after n moves. Then, the boards were one-hot encoded by piece type and color, creating a 12×8×8 tensor. The task was then to predict the final game outcome (white win, draw, black win) from the 12-channel input image, and some examples can be seen in Figure D.6. The ResNet-18 benchmark scored 57.826% accuracy, and the competition record of 62.98% was again attained by the Sun Yat-sen University team.

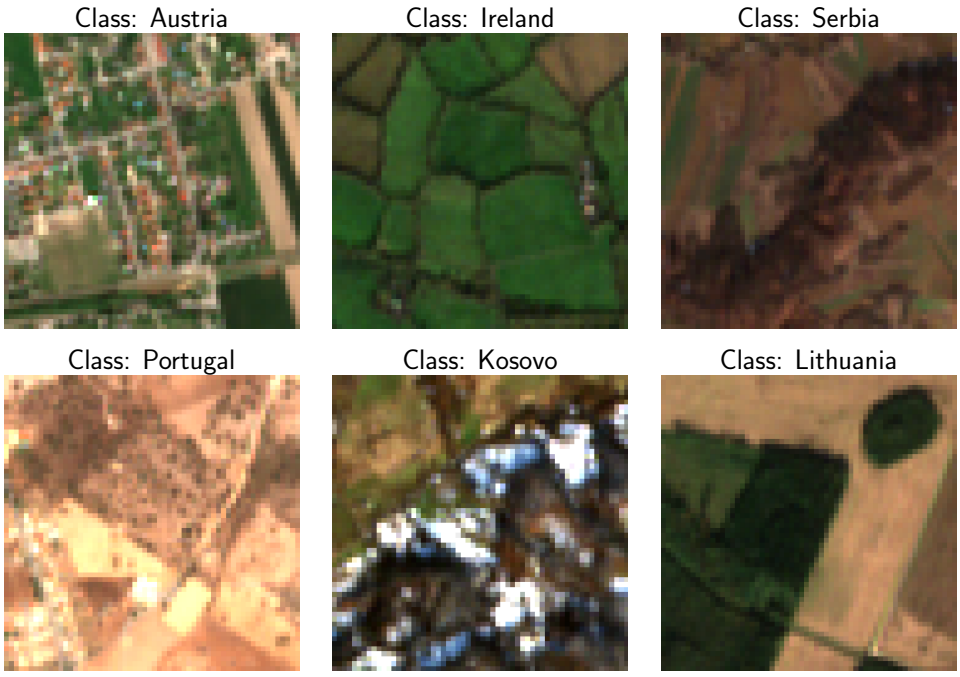


Fig. D.5 Examples of the Sadie dataset

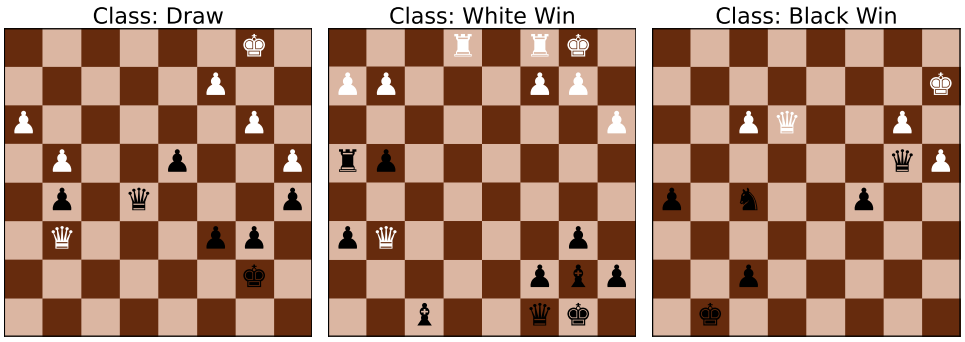


Fig. D.6 Examples of the Chester dataset

D.5.3 Isabella

The final evaluation dataset was codenamed Isabella, from the Isabella Stewart Gardner museum in Boston that published the data. Here, recordings of concerts from within the ISG museum were downloaded from the museum website (Isabella Steward Gardner Museum, 2022), each labeled by era of composition, i.e, Baroque, Classical, Romantic, and 20th Century. The recordings were then split into non-overlapping 5 second snippets, each of which were converted into spectrograms, examples of which can be seen in Figure D.7 The task was to predict which of the four compositional eras the particular spectrogram belonged to. An error in the metadata specifications for this dataset meant that competitors were told that the dataset had 6 labels as opposed to the correct 4, which potentially might negatively skew results. In fairness, all benchmarks of this thesis' work as well as any general results using the Isabella dataset will also use the incorrect 6-class format. The ResNet-18 benchmark scored 62.02% accuracy, which no competitor was able to beat. The closest was again the Sun Yat-sen University team, who scored a 61.42% accuracy.

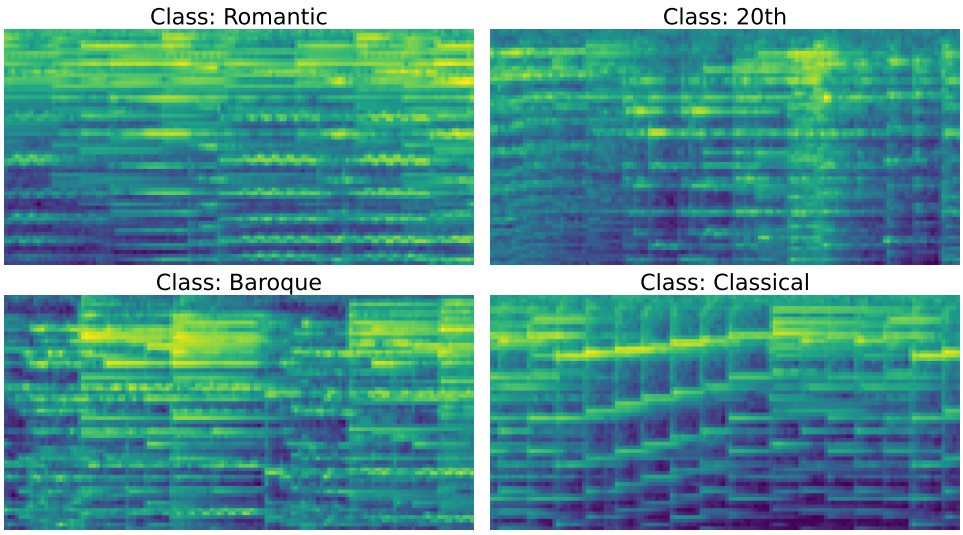


Fig. D.7 Examples of the Isabella dataset

D.6 Competition Engagement

The 2021 competition saw 75 entrants, 341 total submissions, and 11 teams qualify for the final competition phase over the 3 evaluation datasets. The Samsung Research Center Beijing team won the competition, with the University of Friedburg team coming in second, and Kakao Enterprise coming in third, and all three teams were invited to speak at the CVPR-NAS 2021 workshop.

The 2022 edition saw lesser engagement (likely due to the absence of an offered cash prize), but the competition still received around 25 entries, with the Sun Yat-sen University team being the only team to successfully qualify through to the final round, and thus were winners by default.