Efficiently Designing Efficient Deep Neural Networks

Alvin Wan

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-69
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-69.html

May 11, 2022

Acknowledgement

To my parents, for loving and supporting me

Efficiently Designing Efficient Deep Neural Networks

by

Alvin Wan

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph E. Gonzalez, Chair
Professor Angjoo Kanazawa
Professor Kurt Keutzer
Bichen Wu

Spring 2022

Abstract

Efficiently Designing Efficient Deep Neural Networks

by

Alvin Wan

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph E. Gonzalez, Chair

A number of competing concerns slow adoption of deep learning for computer vision on "edge" devices. Edge devices provide only limited resources for on-device algorithms to employ, constraining power, memory, and storage usage. Examples include mobile phones, autonomous vehicles, and virtual reality headsets, which demand both high accuracy and low latency, two objectives competing for resources.

To tackle this sisyphean task, modern methods expend gargantuan amounts of computation to design solutions, exceeding thousands of GPU hours or years of GPU compute to design a single neural network. Not to mention, these works maximize just one performance metric – accuracy – under a single set of resource constraints. What if the set of resource constraints changes? If additional performance metrics rise to the forefront, such as explainability or generalization? Modern methods for designing efficient neural networks are handicapped by excessive computation requirements for goals too singularly and narrowly sighted.

This thesis tackles the bottlenecks of modern methods directly, achieving state-of-the-art performance by efficiently designing efficient deep neural networks. These improvements don't only reduce computation or only improve accuracy; instead, our methods improve performance and reduce computational requirements, despite increasing search space size by orders of magnitude. We also demonstrate missed opportunities with performance metrics beyond accuracy, redesigning the task so that accuracy, explainability, and generalization improve jointly, an impossibility by conventional wisdom, which suggests explainability and accuracy participate in a zero-sum game.

This thesis culminates in a set of models that set new flexibility and performance standards for production-ready models: those that are state-of-the-art accurate, explainable, generalizable, and configurable for any set of resource constraints in just CPU minutes.

To my Parents

for loving and supporting me

Contents

Contents **ii**

List of Figures **iv**

List of Tables **vi**

1 Introduction **1**
1.1 Evolution of Efficient Neural Networks 1
1.2 Efficient Design for Efficient Neural Networks 2
1.3 Thesis Outline 4

2 Background **5**
2.1 Manual, Efficient Design 5
2.2 Automatic, Efficient Design 6
2.3 Concerns Beyond Accuracy 8

3 Searching an Octillion Neural Network Architectures in 24 Hours **10**
3.1 Introduction 10
3.2 Motivation 11
3.3 Neural Architecture Search for Spatial and Channel Dimensions 12
3.4 Experiments 19
3.5 Implementation 23
3.6 Conclusion 24

4 Jointly Searching Neural Architectures and Training Recipes **27**
4.1 Introduction 27
4.2 Motivation 28
4.3 Joint Architecture-Recipe Search via Predictor Pretraining 30
4.4 Experiments 34
4.5 Ablations 39
4.6 Implementation 41
4.7 Discussion 43
4.8 Conclusion 45

5 Jointly Improving Accuracy, Generalization, and Explainability **46**
5.1 Introduction 46
5.2 Motivation 47
5.3 Neural-Backed Decision Trees 48
5.4 Experiments 51
5.5 Interpretability 55
5.6 Applications 57
5.7 Ablations 62
5.8 Implementation 64
5.9 Conclusion 66

6 Conclusion **68**
6.1 Review 68
6.2 Impact 69
6.3 Future Work 70

Bibliography **71**

List of Figures

3.1 Differentiable Neural Architecture Search 11
3.2 How Channel Masking Works 13
3.3 Challenges in Channel Search 14
3.4 Challenges (and Solutions) in Spatial Search 17
3.5 Minimizing Pixel "Contamination" in Spatial Search 18
3.6 Searched FBNetv2 Architectures 19
3.7 Near-Constant Memory Cost of Our DMaskingNAS 21
3.8 FBNEtV2 Accuracy vs. FLOPs 21
3.9 FBNetV2 Accuracy vs. Model Size 23
3.10 FBNetV2 Accuracy vs. FLOPs (Large) 23

4.1 FBNetV3 Accuracy vs. FLOPs (Small) 28
4.2 FBNetV3 Searched Recipe Improves Accuracy 29
4.3 Predictor Pretraining 32
4.4 Pretraining Boosts Predictor Performance 32
4.5 Rank Correlation Improves with Training 33
4.6 FBNetV3 Search Process 36
4.7 FBNetv3 CIFAR-10 Accuracy vs. FLOPs 38
4.8 Search Recipe Training Curve 42
4.9 Training Curves for Object Detection 44
4.10 ImageNet Accuracy vs. FLOPs using Distilled Models 45

5.1 Hard and Soft Decision Trees 49
5.2 Building Induced Hierarchies 50
5.3 NBDT ImageNet Results 53
5.4 CIFAR-10 Blurry Images 56
5.5 Types of Ambiguous Labels 58
5.6 ImageNet Ambiguous Labels 59
5.8 Maximum Similarity Examples 60
5.9 Node Meaning 60
5.10 CIFAR10 Induced Hierarchies 61
5.11 Visualization of Path Traversal Frequency 62
5.12 Tree Supervision Loss Variants 63

5.13 Example Survey Question for Mechanical Turks 66
5.14 CIFAR-100 Tree Visualization on WideResNet28x10 67

List of Tables

3.1 DMaskingNAS Surpasses Prior Art in Number of Design Choices 13
3.2 Macro-Architecture Specification 20
3.3 Micro-Architecture Specification 21
3.4 FBNetV2 ImageNet Classification Performance 22
3.5 ImageNet FLOP-Efficient Classification 24
3.6 ImageNet Parameter-Efficient Classification 24
3.7 Macro-Architecture for Large Search Space 25
3.8 Macro-Architecture for FLOP-Efficient Search Space 25
3.9 Macro-Architecture for Parameter-Efficient Search Space 26

4.1 Training Recipes Affect Architecture Ranking 28
4.2 FBNetV3 Search Space Specification 35
4.3 Accuracy improvements with the searched training recipes on existing neural networks. Above, ResNeXt101 refers to the 32x8d variant. 36
4.4 FBNetV3 Performance Comparisions 37
4.5 FBNetV3 Backbone Improves Object Detection 39
4.6 Accuracy comparison for the searched models with swapped training recipes. . . 39
4.7 Performance improvement by the predictor-based evolutionary searchsearch. *: Models derived from constrained iterative optimization. 40
4.8 FBNetV3 Improvement using EMA, Distillation 40
4.9 Searched Training Recipe 42
4.10 Baseline Architecture for Recipe-Only Search 43
4.11 Effect of Distillation on Model Performance 44

5.1 NBDT Results 52
5.2 Comparision of Hierarchies 53
5.3 Comparisons of Losses 54
5.4 Mid-Training Hierarchy 54
5.5 NBDT Outperforms Original Neural Network 55
5.6 NBDT Exhibits Zero-Shot Superclass Generalization 55
5.7 Comparison of Inference Modes 64
5.8 Tree Supervision Loss 64

Acknowledgments

Thank you to all my collaborators, mentors, friends, and family for their unending and unconditional support. This thesis would not have been possible, had this support network not believed in me before I did.

My adviser Joseph E. Gonzalez or, colloquially, "Joey", deserves all the gratitude I could give and more – giving me guidance when I needed it (a.k.a., all the time), encouraging me to explore research beyond his lab, and supporting me in both the highs and the lows. I am especially grateful for his willingness to fight for me – against the naysayers and the ones that said I didn't belong. His unwavering faith and seemingly infinite breadth of knowledge gave me the flexibility to pivot wildly, and his dedication to my growth – at times, at the cost of his own – is one of the most important reasons for any success I can claim today.

My committee members Kurt Keutzer and Angjoo Kanazawa also deserve a huge thanks: Kurt, for introducing the art of storytelling and emphasizing time and time again that communicating results were just as important if not more important than obtaining results; Angjoo, for welcoming me into her group as one of her own, an endless enthusiasm, and for pushing me to tackle new frontiers in efficient, production-ready research.

Mentors Vaishaal Shankar, Bichen Wu, and Richard Zhang shaped my research philosophy today to a significant degree, being the first mentors to take me under their wing. I am thankful to these mentors for investing their time in me: Vaishaal, for introducing me to research despite my complete lack of prior experience, and for giving me a healthy amount of skepticism for deep learning papers; Bichen, for demonstrating what it takes to produce world-class and famous papers, letting me tag along at my first (and only) ever in-person conference, and vouching for me to all that will listen; and Richard, for his tough love, ingraining in me the best practices for computer vision research.

To the Facebook Reality Labs Mobile-Vision team, I am grateful they took a chance on me as a first-year PhD student and gave me the production-aware mentality that I still carry with me today. Thank you to Peter Vajda for letting me use (and subsequently fall asleep on) your desk, Peizhao Zhang for mentoring me and pushing me to be more results-driven, Xiaoliang Dai for saving me from impending doom (by showing me how to get those results), Bichen Wu (again) for believing in me, Matthew Yu for showing me how to "make impact" (insert clapping) and for hacking with me, Zijian He for supporting me, and all the other team members for their welcoming spirit: Tao Xu, Kan Chen, Zhen Wei. I thank my mentors from FAIR as well: Yuandong Tian, Saining Xie, Zhicheng Yan.

To the Tesla AutoPilot team, I am thankful for the experience of working with such a lean and highly productive team. A big thanks to Lane McIntosh for always guiding me towards

the highest-impact work, showing me what it means to move fast by prioritizing judiciously, Micael Carvalho for exemplifying work ethic and convincing me join AutoPilot, Christian Cosgrove for always being patient and supportive, Andrej Karpathy for appreciating my memes and basic donuts, and the team in general for achieving the impossible: John Emmons, Kate Park, Bradley Emi, Ethan Knight, Danny Hung, and Patrick Li.

A big thanks to labmates and coauthors in Joey's group: Suzie Petryk, Lisa Dunlap, Brijen Thananjeyan, Daniel Rothchild, Paras Jain, Tianjun Zhang, Charles Packer, Matthew Wright, Xin Wang; in Kurt's group: Amir Gholami, Xiangyu Yue, Peter Jin; and in Angjoo's group: Vickie Ye, Matthew Tancik; Sarah Bargel from Boston University, for introducing me to explainability and for always pushing for results you knew were possible (that I doubted, until we actually got there); and Geoff Tison, Robert Avram from UCSF, for introducing me to computer vision applications in medicine. Thank you to my co-author undergraduate collaborators: Daiyaan Arfeen, Daniel Ho, Victor Sun, Scott Lee, Henry Jin, Jihan Yin, Henk Tilman, Younjin Song and all other past and present undergraduate collaborators. Our hack sessions in the 5th floor Soda lounge and elsewhere were a blast.

I thank my industry mentors and colleagues for introducing me to productionizing AI from ground zero: Andy Terrel, Yiwen Wang, Casey Allen, and last but not least, Andy Barkett – who would read my excruciatingly long, panicky email essays and responded with his own equally long email essays to calm me down.

Thank you to the friends and communities that have kept me sane over the years, throughout COVID and beyond: Brijen Thananjeyan, Derek Wan, Andrew Liu, Joyce Lo, Sinho Chewi, Andy Palan, and the Jackbox group.

I thank my mom and dad, whose love and support see no bound, making sacrifices to ensure my brother and I live a good life with plentiful opportunities - coming to America, learning English alongside us, toiling away at work, and more. My work past and present, including this thesis, are just as much their accomplishment as they are mine.

Finally, I would like to thank Emily, who has stuck it out with me through thick and thin, keeping me healthy, sane, and high spirited. Like my mom, she's always right, and this thesis means Emily won the bet – I finished the PhD after all.

Thank you all!

Chapter 1

Introduction

A number of competing concerns slow adoption of deep learning for computer vision on "edge" devices. Edge devices provide only limited resources for on-device algorithms to employ, constraining power, memory, and storage usage. Examples include mobile phones, autonomous vehicles, and virtual reality headsets, which demand both high accuracy and low latency, two objectives competing for resources.

1.1 Evolution of Efficient Neural Networks

To tackle this sisyphean task, modern methods passed through three milestone moments in designing efficient neural networks.

1. Sacrifice Accuracy for Efficiency. Early work took existing deep neural networks and applied a suite of different efficiency techniques, including quantization, pruning, compression, and hyperparameter tuning. Despite significant progress, these post-processing steps and architecture-agnostic approaches reached a saturation point: there were only so many bits to quantize, only so many layers and channels to prune etc. This resulted in a new series of approaches: Instead of working backwards from prior art, design new architectures from the ground up with efficiency in mind.

2. Manually Trade off Accuracy and Efficiency. Subsequent work designed a series of efficient "building blocks", or group of deep learning operations configured to maximize accuracy per parameter or compute–then integrated those into efficient neural architectures built from scratch. These networks saw significantly higher efficiency and could be made even more efficient by combining improved architectures with architecture-agnostic efficiency techniques from before. These efficient building blocks would become a mainstay of efficient neural networks but likewise hit a wall: With a design space easily containing 10^{18} different candidate architectures, how could manual design possibly consider all possibilities? The

natural next step was to consider automatic ways of designing efficient neural networks, or Neural Architecture Search.

3. Automatically Trade off Accuracy and Efficiency. Neural Architecture Search is effective, producing state-of-the-art neural networks that are both more efficient and more accurate than their predecessors. However, these methods were critically flawed in two ways:

1. **Computationally Expensive**: Methods like AutoML expend gargantuan amounts of computation to design solutions, exceeding thousands of GPU hours or years of GPU compute to design a single neural network. Modern methods for designing efficient neural networks are handicapped by excessive computation requirements.
2. **Narrow Focus**: Previous works furthermore maximize just one performance metric – accuracy – under a single set of resource constraints. What if the set of resource constraints changes? If additional performance metrics rise to the forefront, such as explainability or generalization? Modern methods have missed an opportunity to leverage–contrary to conventional wisdom–complementary objectives.

1.2 Efficient Design for Efficient Neural Networks

This thesis tackles the bottlenecks of modern methods directly, achieving state-of-the-art performance by efficiently designing efficient deep neural networks. These improvements don't only reduce computation or only improve accuracy; instead, our methods improve performance and reduce computational requirements, despite increasing search space size by orders of magnitude.

Approaches

We approached the issue of computational expense in two ways, using Neural Architecture Search (NAS) methods at opposite ends of the efficiency spectrum.

Expand Search Space for Efficient NAS Algorithm: Our first approach was to use an existing, already-computationally-cheap neural architecture search method called differentiable neural architecture search (DNAS), which took no more than 24 hours on a DGX to train fully. However, this method involved differentiating through a supergraph of possible neural networks, meaning its search space size was limited by the amount of RAM on a single node. We directly tackled this weakness in DNAS, expanding its search space by 14 orders of magnitude to produce a family of efficient neural networks that attained state-of-the-art performance. More importantly, this drastic increase in search space size incurred no extra computational or memory cost, making the accuracy win "free", called FBNetV2.

Make Inefficient NAS Algorithm more Efficient: Our second approach took the reverse direction: make a computationally-expensive neural architecture search method more computationally efficient. In particular, we chose to use a predictor-based approach, which trains a predictor to predict neural network accuracy. These predictors are expensive to train, as every sample it trains on, is itself another fully trained network: the input is a neural network architecture and the label is the accuracy. A number of papers find surrogates for this fully-trained accuracy number by training on a dataset subset, feature subset, or fewer epochs. However, these techniques then suffer from noisy data which then requires more data to compensate for, resulting in a catch 22. In response, we instead of develop a pretraining strategy using "free" architecture statistics, allowing our predictor to learn a quality archictecture representation and learn architecture accuracies successfully with 2 orders of magnitude less training data. This again produced a family of efficient neural networks that attained state-of-the-art performance, called FBNetV3.

Tackling more than Architecture Accuracy

A significant portion of our insight came from leveraging objectives other than just neural architecture accuracy, leading to new categories of joint optimization that were previously unrealized:

Joint Architecture-Recipe Search: Neural Architecture Search, true to its name, searches only for architectures but not the associated training hyperparameters (i.e., "training recipe"). This ignores the fact that different training recipes may drastically change the success or failure of an architecture, or even switch architecture rankings. To remedy this, we optimize architectures and recipes jointly in a Neural Architecture-Recipe Search (NARS). This produced the efficient state-of-the-art family of neural networks mentioned above, called FBNetV3.

Jointly Improve Accuracy, Explainability, and Generalization: Conventional wisdom says that accuracy and explainability are opposing objectives, where only one or the other can be obtained if not traded off. However, our work debunks this misconception, showing that adding explainability can actually improve both accuracy and generalization in a deep neural network, separating a black box into a series of interpretable discrete decisions. This final part is our most surprising result that defies conventional thinking: Additional objectives, like explainability, are not the antithesis to accuracy. Contrary to prior art, we show that accuracy, explainability, and generalization can all be simultaneously and jointly improved.

Result

This thesis culminates in a set of models that set new flexibility and performance standards for production-ready models: those that are state-of-the-art accurate, explainable, generalizable, and configurable for any set of resource constraints in just CPU minutes.

1.3 Thesis Outline

This thesis addresses the challenges mentioned above, showing how efficient design of efficient deep neural networks yields neural networks with state-of-the-art accuracy with even little resource consumption. Chapter 2 discusses prior art in more detail, covering previous and significant progress in designing efficient deep neural networks, as well as highlighting missed opportunities. Chapters 3 and 4 then discuss the two approaches to handling computational efficiency in neural architecture search, where the former discusses improved accuracy at no additional computational cost and the latter discusses improved accuracy at less computational cost. Chapter 4 additionally discusses joint optimization over both neural architectures and training recipes for further improvements. Chapter 5 extends this, jointly optimizing over accuracy, explainability, and generalization all at the same time. Finally, Chapter 6 concludes the thesis with a summary of impact and discussion of future directions.

Chapter 2

Background

2.1 Manual, Efficient Design

Hand-crafted, efficient neural networks see three predominant approaches: (1) compressing existing architectures, (2) designing compact architectures from scratch, and (3) non-architectural methods.

Making Existing Architectures more Efficient

Most early work compresses existing architectures. One method is pruning [33, 18, 130, 13, 34, 115, 127], where either layers or channels are removed according to certain heuristics. Various heuristics govern layer-wise or channel-wise pruning: For example, Han et al. [34] show that magnitude-based pruning can reduce parameter count by orders of magnitude without accuracy loss, and NetAdapt [128] utilizes a filter pruning algorithm that achieves a 1.2× speedup for MobileNetV2. However, with heuristics-based simplifications, pruning methods train potential architectures sequentially and separately [128], one after another – in some cases, pruning methods consider only one architecture [68, 37, 34]. This limits the design space.

Designing Efficient Architectures from Scratch

Compact architecture design aims to directly construct efficient networks, rather than trim an expensive one [47, 119]. These works design new architectures from the ground up, using new operations that are cost-friendly. This includes convolutional variants like the depthwise convolutions in MobileNet; inverted residual blocks in MobileNetV2; activations such as hswish in MobileNetV3 [41, 93, 40]; and operations like shift [118] and shuffle [71]. For example, MobileNet [41] and MobileNetV2 [93] achieve substantial efficiency improvements by exploiting a depth-wise convolution and an inverted residual block, respectively. ShuffleNetV2 [71] shrinks the model size utilizing low-cost group convolutions. Tan et al. propose a compound

scaling method, obtaining a family of architectures that achieve state-of-the-art accuracy with an order of magnitude fewer parameters than previous convolutional networks [105]. Although many of these building blocks are still used in state-of-the-art neural networks, these models rely on finely-tuned, manual decisions that are bested by automatic design. In particular, manually-designed architectures have been superseded by automatically-searched counterparts.

Non-Architectural Efficiency Improvements

Non-architectural modifications: One non-architectural approach is low-bit quantization, where weights and activations alike may be represented with fewer bits. For example, Wang et al. [113] propose hardware-aware automated quantization, which achieves a 1.4-1.95× latency reduction on MobileNet [41]. Other low-bit quantization [33] papers consider as few as two [138] or even one bit [45] representations. Still other work downsamples input non-uniformly [120, 125, 73] to reduce computational cost. These methods can be combined with architecture improvements for roughly additive reduction in latency. Other non-architecture modifications involve hyperparameter tuning, including tuning libraries from the pre-deep-learning era [7]. Several deep-learning-specific tuning libraries are also widely used [60]. A newer category of approaches automatically searches for the optimal combination of data augmentation strategies. These methods use policy search [17], population-based training [39], Bayesian-based augmentation [109], or Bayesian optimization [51].

2.2 Automatic, Efficient Design

Neural Architecture Search automates neural network design for state-of-the-art performance. Several of the most common techniques for NAS include reinforcement learning [139, 106], evolutionary algorithms [86, 87, 129], and differentiable neural architecture search (DNAS) [66, 117, 111, 32, 124].

Scoring Architectures

Zoph et al. first proposed using reinforcement learning (RL) for automated neural network design in [139]. This and other early NAS approaches are based on RL [139, 106] and evolutionary algorithms (EA) [86]. However, both approaches consume substantial computational resources.

Another direction is to exploit a performance predictor to guide the search process [19, 65]. Such approaches explore the search space by trimming progressively and lead to significant reductions in search cost.

These prior works search for only the model architecture [65, 116, 114, 96, 13] or perform neural architecture-recipe search searches on small-scale datasets (e.g., CIFAR) [4, 134]. By contrast, our NARS jointly searches both architectures and training recipes on ImageNet. To compensate for the larger search space, we (a) introduce a predictor pretraining technique to improve the predictor's rate of convergence and (b) employ predictor-based evolutionary search to design architecture-recipe pairs in just CPU minutes, for any resource constraint setting–outperforming the predictor's highest-ranked candidate before evolutionary search significantly. We also note prior work that generates a family of models with negligible or no cost after one search [32, 129, 69].

Searching Differentiably

Later works utilize various techniques to reduce the computational cost of search. One such technique formulates the architecture search problem as a path-finding process in a supergraph [117, 66, 32, 102]. Among them, gradient-based NAS has emerged as a promising tool. Wu et al. show that gradient-based, differentiable NAS yields state-of-the-art compact architectures with 421× less search cost than RL-based approaches. However, supergraph-based methods suffer from severely limited search space sizes, due to memory constraints.

Stamoulis et al. [103] introduce weight-sharing to further reduce the computational cost of search. However, kernel weight-sharing doesn't address the primary drawback of DARTS, namely a memory bottleneck yielding small search space size: Say a "mixed kernel" contains weights shared between a 3×3 and 5×5. Since it is impossible to extract a 3×3 convolution's outputs from a 5×5's (and vice versa), this mixed kernel still convolves $2\times$ and still stores 2 feature maps for backpropagation. Thus, 2 kernel-weight-sharing convolutions induce memory and computational costs of 2 vanilla convolutions.

Searching along spatial and channel dimensions has been studied both with and without NAS. Liu et al [64] develop a NAS variant that searches over varying strides for semantic segmentation. However, this method suffers from increasing memory cost as the number of possible input resolutions grows. As described above, network pruning suffers from inefficient and sequential exploration of architectures, one-by-one. Yu et al [132] amend this partially by creating a batchnorm invariant to the number input channels; after training the "supergraph" they see competitive accuracy without further training, for each possible subset of channels. Yu et al [67] expand on these slimmable networks by introducing a test-time greedy channel selection procedure. However, these methods are orthogonal to and can be combined with DMaskingNAS, as we train the sampled architecture from scratch. To address these concerns, our algorithm jointly optimizes over multiple input resolutions and channel options simultaneously, increasing memory cost only negligibly as the number of options grows. This allows DMaskingNAS to support orders of magnitude more possible architectures, under existing memory constraints.

DNAS trains quickly with few computational resources but is limited by search space size due to memory constraints. Several works seek to address this issue, by training only subsets at a time [12] or by introducing approximations [111].

2.3 Concerns Beyond Accuracy

There are a number of performance metrics other than accuracy that are likewise important for production deep learning models: two of these metrics are generalization and explainability. One school of thought suggests tackling one performance metric at a time, as performance is a zero sum game and accuracy is more important. However, as our work shows, these performance can all be jointly optimized, with mutual benefit, showing that concurrent consideration for these metrics is a missed opportunity.

Explainable AI

Saliency Maps. Numerous efforts [100, 133, 98, 137, 94, 89, 82, 104] have explored the design of saliency maps identifying pixels that most influenced the model's prediction. White-box techniques [100, 133, 98, 94, 104] use the network's parameters to determine salient image regions, and black-box techniques [89, 82] determine pixel importance by measuring the prediction's response to perturbed inputs. However, saliency does not explain the model's decision process (e.g. Was the model confused early on, distinguishing between *Animal* and *Vehicle*? Or is it only confused between dog breeds?).

Decision Trees

Transfer to Explainable Models. Prior to the recent success of deep learning, decision trees were state-of-the-art on a wide variety of learning tasks and the gold standard for interpretability. Despite this recency, study at the intersection of neural network and decision tree dates back three decades, where neural networks were seeded with decision tree weights [6, 5, 49, 48], and decision trees were created from neural network queries [55, 9, 21, 15, 16], like distillation [38]. The modern analog of both sets of work [46, 99, 30] evaluate on feature-sparse, sample-sparse regimes such as the UCI datasets [28] or MNIST [59] and *perform poorly* on standard image classification tasks.

Hybrid Models. Recent work produces hybrid decision tree and neural network models to scale up to datasets like CIFAR10 [56], CIFAR100 [56], TinyImageNet [58], and ImageNet [23]. One category of models organizes the neural network into a hierarchy, dynamically selecting branches to run inference [110, 74, 108, 88, 78]. However, these models use *impure leaves* resulting in uninterpretatble, stochastic paths. Other approaches fuse deep learning into each decision tree node: an entire neural network [79], several layers [78, 91], a linear layer [1], or some other parameterization of neural network output [54]. These models see reduced

interpretability by using k-way decisions with large k (via depth-2 trees) [1, 31] or employing an ensemble [54, 1], which is often referred to as a "black box" [14, 92].

Hierarchical Classification [97]. One set of approaches directly uses a pre-existing hierarchy over classes, such as WordNet [88, 11, 25]. However *conceptual similarity is not indicative of visual similarity*. Other models build a hierarchy using the training set directly, via a classic data-dependent metric like Gini impurity [2] or information gain [90, 8]. These models are instead *prone to overfitting*, per [107]. Finally, several works introduce hierarchical surrogate losses [121, 24], such as hierarchical softmax [77], but as the authors note, these methods quickly suffer from major accuracy loss with more classes or higher-resolution images (e.g. beyond CIFAR10). We demonstrate hierarchical classifiers attain higher accuracy *without* a hierarchical softmax.

Chapter 3

Searching an Octillion Neural Network Architectures in 24 Hours

3.1 Introduction

Differentiable Neural Architecture Search (DNAS) has demonstrated great success in designing state-of-the-art, efficient neural networks. However, DARTS-based DNAS's search space is small when compared to other search methods', since all candidate network layers must be explicitly instantiated in memory. To address this bottleneck, we propose a memory and computationally efficient DNAS variant: DMaskingNAS. This algorithm expands the search space by up to $10^{14}\times$ over conventional DNAS, supporting searches over spatial and channel dimensions that are otherwise prohibitively expensive: input resolution and number of filters. We propose a masking mechanism for feature map reuse, so that memory and computational costs stay nearly constant as the search space expands. Furthermore, we employ effective shape propagation to maximize per-FLOP or per-parameter accuracy. The searched FBNetV2s yield state-of-the-art performance when compared with all previous architectures. With up to $421\times$ less search cost, DMaskingNAS finds models with 0.9% higher accuracy, 15% fewer FLOPs than MobileNetV3-Small; and with similar accuracy but 20% fewer FLOPs than Efficient-B0. Furthermore, our FBNetV2 outperforms MobileNetV3 by 2.6% in accuracy, with equivalent model size. FBNetV2 models are open-sourced at https://github.com/facebookresearch/mobile-vision.[1]

[1]The contents of this work are in collaboration with Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, Peter Vajda, and Joseph E. Gonzalez, published at CVPR 2020 [111]. This was a part of my internship at Facebook Reality Labs and was supported by my National Science Foundation Graduate Research Fellowship under Grant No. DGE 1752814. In addition to NSF CISE Expeditions Award CCF-1730628, UC Berkeley research is supported by gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

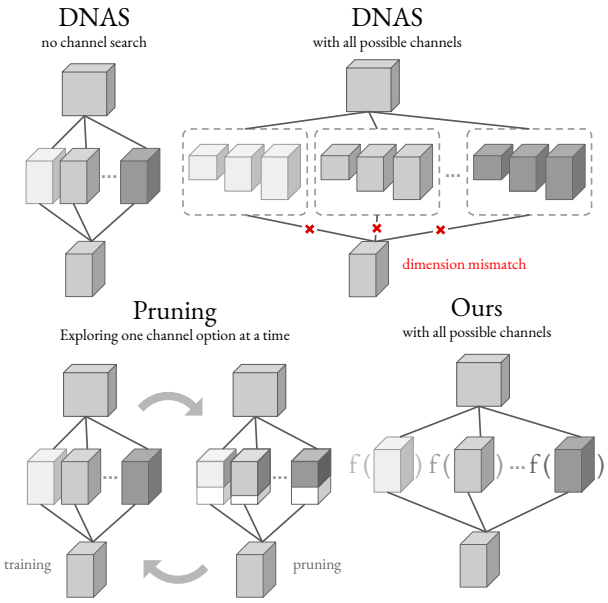


Figure 3.1: DNAS: Adding all possible numbers of filters to DNAS (top-right) increases computational and memory costs drastically, exacerbating DNAS's memory bottleneck on search space size. **Pruning**: Channel pruning (bottom-left) is limited to training one architecture at a time. **Ours**: With our weight-sharing approximation, DNAS can explore all possible number of filters simultaneously with negligible memory and computation overhead. See Fig. 3.2 for details.

3.2 Motivation

Deep neural networks have led to significant progress in many research areas and applications, such as computer vision and autonomous driving. Despite this, designing an efficient network for resource-constrained settings remains a challenging problem. Initial directions involved compressing existing networks [33] or building small networks [71, 93]. However, the design space can easily contain more than 10^{18} candidate architectures [117, 102], making manual design choices sub-optimal and difficult to scale. In lieu of manual tuning, recent work uses neural architecture search (NAS) to design networks automatically.

Previous NAS methods utilize reinforcement learning (RL) techniques or evolutionary algorithms (EAs). However, both methods are computationally expensive and consume thousands of GPU hours [140, 106]. As a result, recent NAS literature [117, 66, 83] focuses on differentiable neural architecture search (DNAS); DNAS searches over a supergraph that encompasses all candidate architectures, selecting a single path as the final neural network. Unlike conventional NAS, DNAS can search large combinatorial spaces in the time it takes to train a single model [66, 124, 117, 102]. One class of DNAS methods, based on DARTS [66], suffer from two significant limitations [29]:

- **Memory costs bound the search space.** Short of paging in and out tensors, the supergraph and feature maps must reside in GPU memory for training, which limits the search space.
- **Cost grows linearly with the number of options per layer.** This means that each new search dimension introduces combinatorially more options and combinatorial memory and computational costs.

The other class of DNAS methods, not based on DARTS, suffer from similar issues: For example, ProxylessNAS tackles the memory constraint by training only one path in the supergraph each iteration. However, this means ProxylessNAS would take a prohibitively long time to converge on an order-of-magnitude larger search space. These memory and computation issues, for all DNAS methods, prevent us from expanding the search space to explore larger spaces of configurations. Noting that feature maps typically dominate memory cost [22], we propose a formulation of DNAS (Fig. 3.1) called DMaskingNAS (Fig. 3.2) that increases the search space size by orders of magnitude. To accomplish this, we represent multiple channel and input resolution options in the supergraph with masks, which carry negligible memory and computational costs. Furthermore, we reuse feature maps for all options in the supergraph, which enables nearly constant memory cost with increasing search space sizes. These optimizations yield the following three contributions:

- **A memory and computationally efficient DNAS** that optimizes both macro- (resolution, channels) and micro- (building blocks) architectures jointly in a $10^{14}\times$ larger search space using differentiable search. To the best of our knowledge, we are the first to tackle this problem using a differentiable search framework supergraph, with substantially less computational cost and roughly constant memory cost.
- **A masking mechanism and effective shape propagation** for feature map reuse. This is applied to both the spatial and channel dimensions in DNAS.
- **State-of-the-art results** on ImageNet classification. With only 27 hours on 8 GPUs, our searched compact models lead to substantial per-parameter, per-FLOP accuracy improvements. The searched models outperform all previous state-of-the-art neural networks, both manually and automatically designed, small and large.

3.3 Neural Architecture Search for Spatial and Channel Dimensions

We propose DMaskingNAS to search over spatial and channel dimensions, summarized in Fig. 3.2. The search space would be computationally prohibitive and ill-formed without the

Table 3.1: The number of DMaskingNAS design choices eclipses that of previous search spaces: number of channels c, kernel size k, number of layers l, bottleneck type b, input resolution r, and expansion rate e.

NAS algorithm	c	k	l	b	r	e
MnasNet [106]	✓		✓	✓		✓
ProxylessNAS [12]		✓	✓	✓		✓
Single-Path NAS [102]		✓	✓			✓
ChamNet [19]	✓		✓		✓	✓
FBNet [117]		✓	✓	✓		✓
DMaskingNAS	✓	✓	✓	✓	✓	✓

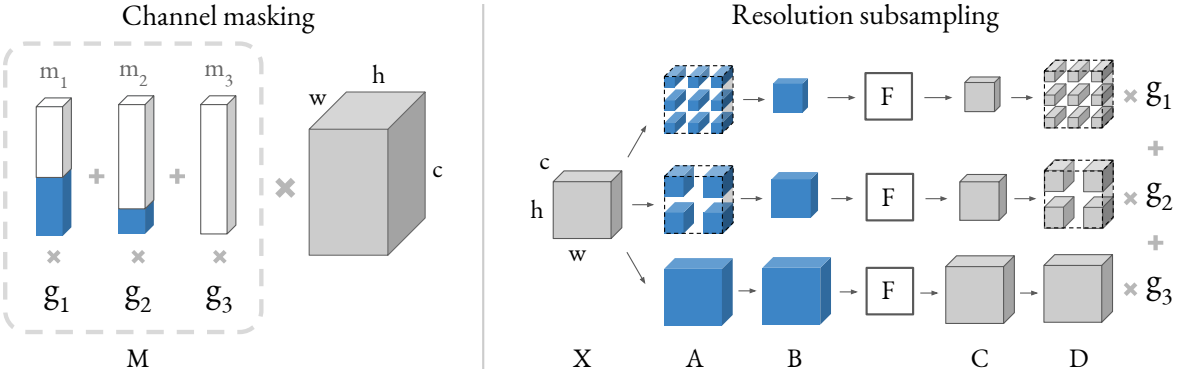


Figure 3.2: Channel Masking for channel search: A column vector mask $M \in \mathbb{R}^c$ is the weighted sum of several masks $m_i \in \mathbb{R}^c$, with Gumbel Softmax weights g_i. Each m_i has ones (white) in the first k entries and zeros (blue) in the next $c - k$ entries, for some $k \in \mathbb{Z}$. Multiplication with this mask speeds up channel search, using a weight-sharing approximation described in Fig. 3.3. **Resolution Subsampling** for input resolution: X is an intermediate output feature map for the network. A is subsampled from X using nearest neighbors. Values at the blue pixels in column A are assembled to create the smaller feature map in B. Next, run the operation F. Finally, each value in C is placed back into a larger feature map in D. Note we put values back (D) into pixels where we pulled values from (A). This process is motivated in Fig. 3.4.

optimizations described below; our approach makes it possible to search this expanded search space (Table 3.1) over channels and input resolutions.

Channel Search

To support searches over varying numbers of channels, previous DNAS methods simply instantiate a block for every channel option in the supergraph. For a convolution with k filters, this could mean up to $k(k+1)/2 \sim O(k^2)$ convolutions. Previous channel pruning methods [67] suffer from a similar drawback: each option must be trained separately, finding

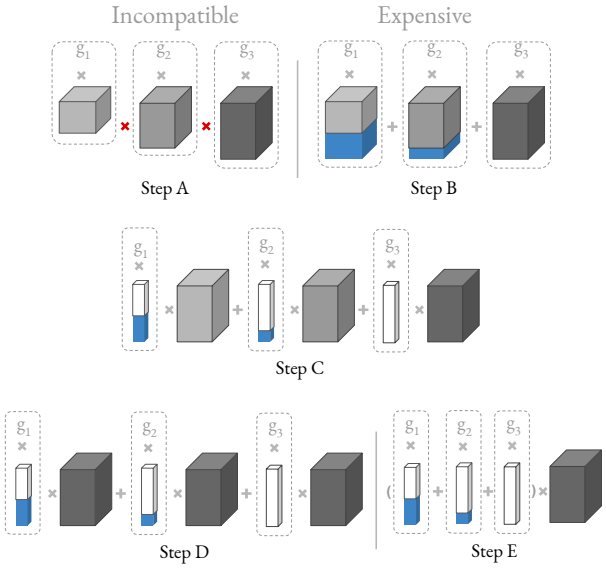


Figure 3.3: Channel Search Challenges: **Step A**: Consider 3 convolutions with varying numbers of filters. Each output (gray) will have varying numbers of channels. Thus, the outputs cannot be naively summed. **Step B**: Zero-padding (blue) outputs allows them to be summed. However, both FLOP and memory cost increases sub-linearly with the number of channel options. **Step C**: This is equivalent to running three convolutions with equal numbers of filters, multiplied by masks of zeros (blue) and ones (white). **Step D**: We approximate using weight sharing – all three convolutions are represented by one convolution. **Step E**: This is equivalent to summing the masks first, before multiplying by the output. Now, FLOP and memory cost are effectively constant w.r.t. the number of channel options.

the "optimal" channel count in one shot or iteratively. Furthermore, even without saturating the maximum number of possibilities, there are two problems, the first of which makes this search impossible:

1. **Incompatible dimensions**: DNAS is divided into several "cells". In each cell, we consider a number of different block options; the outputs of all options are combined in a weighted sum. This means that all block outputs must align dimensions. If each block adopts convolutions with different number of filters, each output will have a different number of channels. As a result, DNAS could not perform a weighted sum.
2. **Slower training, increased memory cost**: Even with a workaround, with this naïve instantiation, each convolution with a different channel option must be run separately, resulting in a $O(k)$ increase in FLOP cost. Furthermore, each output feature map must be stored separately in memory.

To address the aforementioned issues, we handle the incompatibility (Fig. 3.3, Step A):

consider a block b with varying numbers of filters, where b_i denotes this block with i filters. The maximum number of filters is k. The outputs of all blocks are then zero-padded to have k channels (Fig. 3.3, Step B). Given input x, the Gumbel Softmax output is thus the following, with Gumbel weights g_i:

$$y = \sum_{i=1}^{k} g_i \textsc{Pad}(b_i(x), k) \tag{3.1}$$

Note that this is equivalent to increasing the number of filters for all convolutions to k, and masking out the extra channels (Fig. 3.3, Step C). $\mathbb{1}_i \in \mathbb{R}^k$ is a column vector with i leading 1s and $k - i$ trailing zeros. Note that the search method is invariant to the ordering of 1s and 0s. Since all blocks b_i have the same number of filters, we can approximate by sharing weights, so that $b_i = b$ (Fig. 3.3, Step D).

$$y = \sum_{i=1}^{k} g_i (b(x) \circ \mathbb{1}_i) \tag{3.2}$$

Finally, with this approximation, we can handle the computational complexity of the naïve channel search approach: this is equivalent to computing the aggregate mask and running the block b only once (Fig. 3.3, Step E).

$$y = b(x) \circ \underbrace{\sum_{i=1}^{k} g_i \mathbb{1}_i}_{M} \tag{3.3}$$

This approximation only requires one forward pass and one feature map, inducing no additional FLOP or memory costs other than the negligible M term in Eq. 3.3 (Fig. 3.2, Channel Masking). Furthermore, the approximation falls short of equivalence only because weights are shared, which is shown to reduce train time and boost accuracy in DNAS [103]. This allows us to search the number of output channels for any block, including related architectural decisions such as the expansion rate in an inverted residual block.

Input Resolution Search

For spatial dimensions, we search over input resolutions. As with channels, previous DNAS methods would simply instantiate each block with every input resolution. This naïve method's downfalls are twofold: increased memory cost and incompatible dimensions. As before, we address both issues directly by zero-padding the result. However, there are two caveats:

1. **Pixel misalignment**: means padding cannot occur naïvely as before. It would not make sense to zero-pad the periphery of the image, since the sum in Eq. 3.1 would result in misaligned pixels (Fig. 3.4, B). To handle pixel misalignment, we zero-pad such that zeros are interspersed spatially (Fig. 3.4, C). This zero-padding pattern is uniform; except for the zeros, this is a nearest neighbors upsampling. For example, a 2× increase in size would involve zero-padding every other row and column. Zero-padding instead of upsampling minimizes "pixel contamination" across input resolutions (Fig. 3.5).

2. **Receptive field misalignment**: Since subsets of the feature map correspond to different resolutions, naïvely convolving over the full feature map would result in a reduced receptive field (Fig. 3.4, D). To handle receptive field misalignment, we convolve over subsampled input instead. (Fig. 3.4, E). Using Gumbel Softmax, we arrive at "resolution subsampling" in Fig. 3.2.

NASNet [140] introduces a similar notion of combining hidden states. These combinations are also used to efficiently explore a combinatorially large search space but are used to determine – instead of input resolution or channels – the number of times to repeat a searched cell. With the above insights, the input resolution search thus incurs constant memory cost, regardless of the number of input resolutions. On the other hand, computational cost increases sub-linearly as the number of resolutions grows.

Effective Shape Propagation

Note this calculation for effective shape is only used during training. In our formulation of the weighted sum Eq. 3.1, the output y retains the maximum number of channels. However, there exists a non-integral number of *effective* channels: say a 16-channel output has Gumbel weight $g_i = 0.8$ and a 12-channel output has weight $g_i = 0.2$. This means the effective number of channels is $0.8 * 16 + 0.2 * 12 = 15.2$. These effective channels are necessary for both FLOP and parameter computation, as assigning higher weight to more channels should incur a larger cost penalty. This effective shape is how we realize effective resource costs introduced in previous works [117, 124]: First, define the gumbel softmax weights as

$$g_i^l = \frac{\exp[(\alpha_i^l + \epsilon_i^l)/\tau]}{\Sigma_i \exp[(\alpha_i^l + \epsilon_i^l)/\tau]} \tag{3.4}$$

with sampling parameter α, Gumbel noise ϵ, temperature τ. For a convolution with Gumbel Softmax in the l^{th} layer, we define its effective output shape $\bar{S}_{out}^l$ in Eq. 3.7 using effective output channel ($\bar{C}_{out}^l$, Eq. 3.5), and effective height, width ($\bar{h}_{out}^l, \bar{w}_{out}^l$, Eq. 3.6).

$$\bar{C}_{out}^l = \Sigma_i g_i^l \cdot C_{i,out}^l \tag{3.5}$$

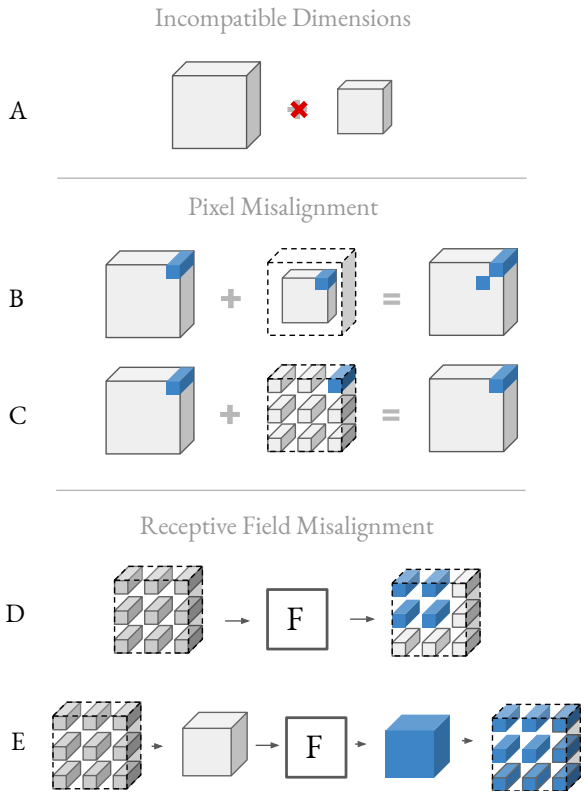


Figure 3.4: Spatial Search Challenges: **A**: Tensors with different spatial dimensions cannot be summed due to incompatible dimensions. **B**: Zero-padding along the periphery of the smaller feature map makes summing possible. However, the top-right pixels (blue) are not aligned correctly. **C**: Interspersing zero-padding spatially results in a sum that aligns pixels correctly. Note the top-right pixels of both feature maps are correctly overlapping in the sum. **D**: Say F is a convolution with 3×3 kernels. Convolving naïvely with the feature map, containing a subset (gray), results in reduced receptive field (2×2, blue) for the subset. **E**: To preserve receptive field for all searched input resolutions, the input must be subsampled before convolving. Note the receptive field (blue) is still 3×3. Furthermore, note we can achieve the same effect, without the need to construct a smaller tensor, with appropriately-strided dilated convolutions; we subsample to avoid modifying the operation F.

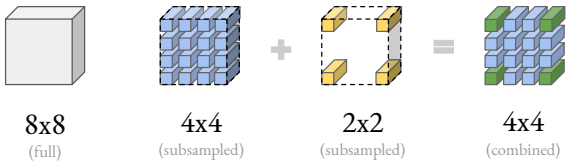


Figure 3.5: Minimizing Pixel "Contamination": On the far left, we have the original 8×8 feature map. The blue 4×4 is a feature map subsampled with nearest neighbors and zero-padded uniformly. The yellow 2×2 is also subsampled and zero-padded. Summing the 2×2 with the 4×4 yields the combined feature map to the far right. Only the green pixels in the corners hold values from both feature map sizes; these green values are "contaminated" by the lower resolution feature maps.

$$\bar{h}^l_{out} = \Sigma_i g^l_i \cdot \bar{h}^l_{in}, \bar{w}^l_{out} = \Sigma_i g^l_i \cdot \bar{w}^l_{in} \tag{3.6}$$

$$\bar{S}^l_{out} = (n, \bar{C}^l_{out}, \bar{h}^l_{out}, \bar{w}^l_{out}) \tag{3.7}$$

with batch size n, effective input width $\bar{w}_{in}$ and height $\bar{h}_{in}$.

For a convolution layer without a Gumbel Softmax, effective output shape simplifies to Eq. 3.8, where effective channel count is equal to actual channel count. For a depth-wise convolution, effective output shape simplifies to Eq. 3.9, where effective channel count is simply propagated.

$$\bar{C}^l_{out} = C^l_{out} \tag{3.8}$$

$$\bar{C}^l_{out} = \bar{C}^l_{in} \tag{3.9}$$

with actual output channel count C_{out}, effective input channel count $\bar{C}_{in}$. Then, we define the cost function for the l^{th} layer as follow:

$$\text{cost}^l = \begin{cases} k^2 \cdot \bar{h}^l_{out} \cdot \bar{w}^l_{out} \cdot \bar{C}^l_{in} \cdot \bar{C}^l_{out} \ / \ \gamma & \text{if } \texttt{FLOP} \\ k^2 \cdot \bar{C}^l_{in} \cdot \bar{C}^l_{out} \ / \ \gamma & \text{if } \texttt{param} \end{cases} \tag{3.10}$$

with γ convolution groups. The effective input channels for the $(l+1)^{th}$ layer are $\bar{C}^{l+1}_{in} = \bar{C}^l_{out}$. The total training loss consists of (1) cross-entropy loss and (2) total cost, which is the sum of cost from all layers: $\text{cost}_{total} = \Sigma_l \text{cost}^l$.

In the forward pass, for all convolutions, we calculate and return both the output tensor and effective output shape. Additionally, τ in the Gumbel Softmax Eq. 3.4 decreases throughout training, [50], forcing g^l to approach a one-hot distribution. $\text{argmax}_i g^l_i$ would thus select a path of blocks in the supergraph; a single channel and expansion rate option for each block; and a single input resolution for the entire network. This final architecture is then trained. Note this final model does not employ masking or require effective shapes.

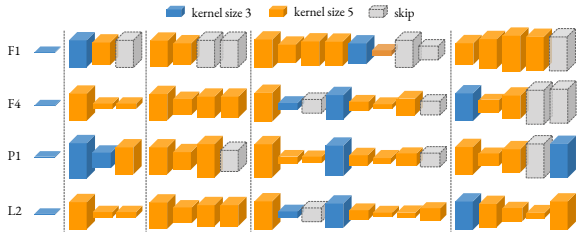


Figure 3.6: Searched FBNetV2 architectures, with colors denoting different kernel sizes and heights denoting different expansion rates. The heights are drawn to scale.

3.4 Experiments

We use DMaskingNAS to search for convolutional network architectures under different objectives. We compare our search space, performance of searched models, and search cost to previously state-of-the-art networks. Detailed numerical results are listed in Table 5.3.

Experimental Setup

We implement DMaskingNAS using PyTorch on 8 Tesla V100 GPUs with 16GB memory. We use DMaskingNAS to search for convolutional neural networks on the ImageNet (ILSVRC 2012) classification dataset [23], a widely-used NAS evaluation benchmark. We use the same training settings as reported in [117]: we randomly select 10% of classes from the original 1000 classes and train the supergraph for 90 epochs. In each epoch, we train the network weights with 80% of training samples using SGD. We then train the Gumbel Softmax sampling parameter α with the remaining 20% using Adam [53]. We set initial temperature τ to 5.0 and exponentially anneal by $e^{-0.045} \approx 0.956$ every epoch.

Search Space

Previous cell-level searches produced fragmented, complicated, and latency-unfriendly blocks. Thus, we adopt a layer-wise search space for known, latency-friendly blocks.

Table 3.3 describes the micro-architecture search space: the block structure is inspired by [93, 40] and sequentially consists of a 1×1 point-wise convolution, a 3×3 or 5×5 depth-wise convolution, and another 1×1 point-wise convolution. Table 3.2 describes the macro-architecture. The search space contains more than 10^{35} candidate architectures, which is $10^{14}\times$ larger than DNAS's [117].

Memory Cost

Our memory optimizations yield a ∼1MB increase in memory cost for every 2 orders of magnitude the channel search space grows by; for context, this 1 MB increase is just 0.1% of

Table 3.2: Macro-architecture for our largest search space, describing block type b, block expansion rate e, number of filters f, number of blocks n, stride of first block s. "TBS" means layer type needs to be searched. Tuples of three values represent the lowest value, highest, and steps between options (low, high, steps). The maximum input resolution for FBNetV2-P models is 288, for FBNetV2-F is 224, and for FBNetV2-L is 256. See supplementary material for all search spaces.

Max. Input	b	e	f	n	s
$256^2 \times 3$	3x3	1	16	1	2
$128^2 \times 16$	TBS	1	(12, 16, 4)	1	1
$128^2 \times 16$	TBS	(0.75, 3.25, 0.5)	(16, 28, 4)	1	2
$64^2 \times 28$	TBS	(0.75, 3.25, 0.5)	(16, 28, 4)	2	1
$64^2 \times 28$	TBS	(0.75, 3.25, 0.5)	(16, 40, 8)	1	2
$32^2 \times 40$	TBS	(0.75, 3.25, 0.5)	(16, 40, 8)	2	1
$32^2 \times 40$	TBS	(0.75, 3.75, 0.5)	(48, 96, 8)	1	2
$16^2 \times 96$	TBS	(0.75, 3.75, 0.5)	(48, 96, 8)	2	1
$16^2 \times 96$	TBS	(0.75, 4.5, 0.75)	(72, 128, 8)	4	1
$16^2 \times 128$	TBS	(0.75, 4.5, 0.75)	(112, 216, 8)	1	2
$8^2 \times 216$	TBS	(0.75, 4.5, 0.75)	(112, 216, 8)	3	1
$8^2 \times 216$	1x1	-	1984	1	1
$8^2 \times 1984$	avgpl	-	-	1	1
1984	fc	-	1000	1	-

the total memory cost during training. This is due to our feature map reuse as described in Sec. 3.3. We compare memory costs for DNAS and DMaskingNAS as the number of channel options increases (Fig. 3.7, left). With only 8 channel options for each convolution, DNAS fails to fit in memory during training, exceeding the 16GB memory supported by a Tesla V100 GPU. On the other hand, DMaskingNAS supports 32-option channel search, for a $32^{22} \sim 10^{33}$ in search space size (given our 22-layer search space), at nearly constant memory cost. Here, k-option channel search means that for each convolution with c channels, we search over $\{c/k, 2c/k, ..., c\}$ channels. To compare larger numbers of channel options, we reduce the number of blocks options in the search space (Fig. 3.7, right). To compute memory cost, we average the maximum memory allocated during each training step, across 10 epochs.

Search for ImageNet Models

FLOP-efficient models: We first use DMaskingNAS to find compact models (Fig. 3.6) for low computational budgets, with models ranging from 50 MFLOPs to 300 MFLOPs in Fig. 3.8. The searched FBNetV2s outperform all existing networks.

Storage-efficient models: Many real world scenarios face limited on-device storage space. Thus, we next perform searches for models minimizing parameter count, in Fig. 3.9. With

Table 3.3: Micro-architecture search space for block design: non-linearities, kernel sizes, and Squeeze-and-Excite [42].

block type	kernel	squeeze-and-excite	non-linearity
ir_k3	3	N	relu
ir_k5	5	N	relu
ir_k3_hs	3	N	hswish
ir_k5_hs	5	N	hswish
ir_k3_se	3	Y	relu
ir_k5_se	5	Y	relu
ir_k3_se_hs	3	Y	hswish
ir_k5_se_hs	5	Y	hswish
skip	-	-	-

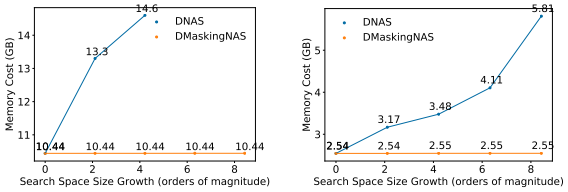


Figure 3.7: Memory Cost of DNAS vs. DMaskingNAS (Left) Conventional DNAS does not fit into memory with just 8 options per block in channel search. On the other hand, DMaskingNAS's memory cost remains roughly constant, even with 32 channel options per block. (Right) We reduce the number of block options in the search space to fit conventional DNAS into memory. The memory cost growth, as the search space increases, is significantly steeper than that of DMaskingNAS; in fact, DMaskingNAS's memory cost is nearly constant.

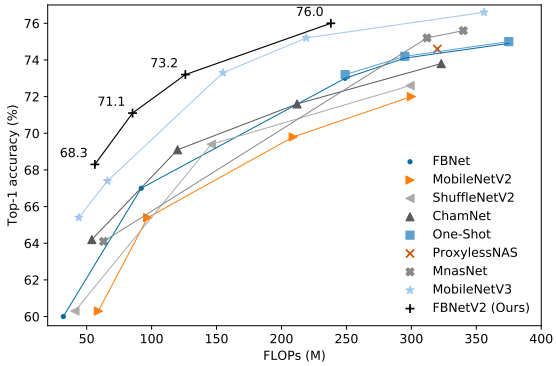


Figure 3.8: ImageNet Accuracy vs. Model FLOPs. We refer to these FLOP-efficient FBNetV2s as FBNetV2-F{1, 2, 3, 4} from left to right.

Model	Search			FLOPs	Top-1
	Method	Space	Cost (GPU hours)		Acc (%)
MobileNetV2-0.35× [93]	manual	-	-	59M	60.3
ShuffleNetV2-0.5× [71]	manual	-	-	41M	60.3
MnasNet-0.35× [106]	RL	stage-wise	91K*	63M	64.1
ChamNet-E [19]	EA	stage-wise	28K†	54M	64.2
FBNet-0.35× [117]	gradient	layer-wise	0.2K	72M	65.3
MobileNetV3-Small [40]	RL/NetAdapt	stage-wise	>91K‡	66M	67.4
FBNetV2-F1 (ours)	**gradient**	**layer-wise**	**0.2K**	**56M**	**68.3**
MobileNetV2-1.0× [93]	manual	-	-	300M	72.0
ShuffleNetV2-1.5× [71]	manual	-	-	299M	72.6
DARTS [66]	gradient	cell	0.3K	595M	73.1
FBNetV2-F3 (ours)	**gradient**	**layer-wise**	**0.2K**	**126M**	**73.2**
ChamNet-B [19]	EA	stage-wise	28K†	323M	73.8
FBNet-B [117]	gradient	layer-wise	0.2K	295M	74.1
One-Shot NAS [32]	EA	layer-wise	0.3K	295M	74.2
ProxylessNAS [12]	gradient/RL	layer-wise	0.2K	320M	74.6
MobileNetV3-Large [40]	RL/NetAdapt	stage-wise	>91K‡	219M	75.2
MnasNet-A1 [106]	RL	stage-wise	91K*	312M	75.2
FBNetV2-F4 (ours)	**gradient**	**layer-wise**	**0.2K**	**238M**	**76.0**
ResNet-50 [35]	manual	-	-	4.1B	76.0
DenseNet-169 [44]	manual	-	-	3.5B	76.2
EfficientNet-B0 [105]	RL/scaling	stage-wise	>91K‡	390M	77.3
FBNetV2-L1 (ours)	**gradient**	**layer-wise**	**0.6K**	**325M**	**77.2**

Table 3.4: ImageNet classification performance: For baselines, we cite statistics on ImageNet from the original papers. Our results are bolded. *: The search cost is estimated based on the experimental setup in [106]. †: [19] discovers 5 models with the cost of training 240 networks. ‡: The cost estimation is a lower bound. [40] and [105] combines the approach proposed in [106] with [128] and compound scaling.

similar or smaller model size (4M parameters), FBNetV2 achieves 2.6% and 2.9% absolute accuracy gains over MobileNetV3 [40] and FBNet [117], respectively.

Large models: We finally use DMaskingNAS to explore larger models for high-end devices. We compare FBNetV2-Large with networks of 300+ MFLOPs in Fig. 3.10.

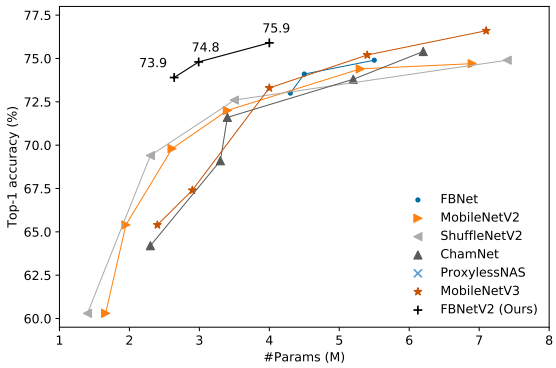


Figure 3.9: ImageNet Accuracy vs. Model Size. We refer to these as parameter-efficient FBNetV2s as FBNetV2-P{1, 2, 3} from left to right.

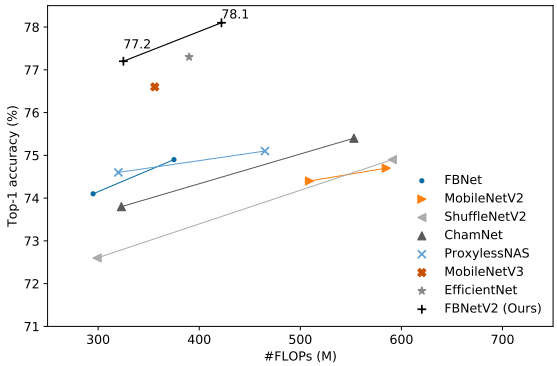


Figure 3.10: ImageNet Accuracy vs. Model FLOPs for Large Models. We refer to these large FBNetV2s as FBNetV2-L{1, 2} from left to right.

Full ImageNet Results: We include numeric results for all three categories of FBNetV2s, optimized for various resource constraints: FLOP-efficient FBNetV2-F and large FBNetV2-L in Table 3.5, parameter-efficient FBNetV2-P in Table 3.6.

3.5 Implementation

Macro-architecture Search Spaces

We list the DMaskingNAS macro-architecture search spaces for all three categories of FBNetV2s, optimized for various resource constraints: FLOP-efficient FBNetV2-F in Table 3.8, parameter-efficient FBNetV2-P in Table 3.9, and large FBNetV2-L in Table 3.7. Note that in all classes of models, the micro-architecture search space over blocks remains the same.

Model	Input	Flops	Top-1 (%)
FBNetV2-F1	128	56M	68.3
FBNetV2-F2	160	85M	71.1
FBNetV2-F3	192	126M	73.2
FBNetV2-F4	224	238M	76.0
FBNetV2-L1	224	325M	77.2
FBNetV2-L2	256	422M	78.1

Table 3.5: ImageNet FLOP-efficient classification: These are the FBNetV2 models yielded by DMaskingNAS optimizing for FLOP count and accuracy.

Model	Input	Params	Top-1 (%)
FBNetV2-P1	288	2.64M	73.9
FBNetV2-P2	288	2.99M	74.8
FBNetV2-P3	288	4.00M	75.9

Table 3.6: ImageNet parameter-efficient classification: These are the FBNetV2 models yielded by DMaskingNAS optimizing for parameter count and accuracy.

3.6 Conclusion

We propose a memory-efficient algorithm, drastically expanding the search space for DNAS by supporting searches over spatial and channel dimensions. These contributions target the main bottleneck for DNAS – high memory cost that induces constraints on the search space size – and yield state-of-the-art performance.

Table 3.7: Macro-architecture for our largest search space for **FBNetV2-L**, describing block type b, block expansion rate e, number of filters f, number of blocks n. "TBS" means layer type needs to be searched. Tuples of three values additionally represent steps between options (low, high, steps). The maximum input resolution for FBNetV2-L is 256.

Max. Input	b	e	f	n	s
$256^2 \times 3$	3x3	1	16	1	2
$128^2 \times 16$	TBS	1	(12, 16, 4)	1	1
$128^2 \times 16$	TBS	(0.75, 3.25, 0.5)	(16, 28, 4)	1	2
$64^2 \times 28$	TBS	(0.75, 3.25, 0.5)	(16, 28, 4)	2	1
$64^2 \times 28$	TBS	(0.75, 3.25, 0.5)	(16, 40, 8)	1	2
$32^2 \times 40$	TBS	(0.75, 3.25, 0.5)	(16, 40, 8)	2	1
$32^2 \times 40$	TBS	(0.75, 3.75, 0.5)	(48, 96, 8)	1	2
$16^2 \times 96$	TBS	(0.75, 3.75, 0.5)	(48, 96, 8)	2	1
$16^2 \times 96$	TBS	(0.75, 4.5, 0.75)	(72, 128, 8)	4	1
$16^2 \times 128$	TBS	(0.75, 4.5, 0.75)	(112, 216, 8)	1	2
$8^2 \times 216$	TBS	(0.75, 4.5, 0.75)	(112, 216, 8)	3	1
$8^2 \times 216$	1x1	-	1984	1	1
$8^2 \times 1984$	avgpl	-	-	1	1
1984	fc	-	1000	1	-

Table 3.8: Macro-architecture for our FLOP-efficient search space for **FBNetV2-F**. The maximum input resolution for FBNetV2-F is 224. See Table 3.7 for column names.

Max. Input	b	e	f	n	s
$224^2 \times 3$	3x3	1	16	1	2
$112^2 \times 16$	TBS	1	(12, 16, 4)	1	1
$112^2 \times 16$	TBS	(0.75, 4.5, 0.75)	(16, 24, 4)	1	2
$56^2 \times 24$	TBS	(0.75, 4.5, 0.75)	(16, 24, 4)	2	1
$56^2 \times 24$	TBS	(0.75, 4.5, 0.75)	(16, 40, 8)	1	2
$28^2 \times 40$	TBS	(0.75, 4.5, 0.75)	(16, 40, 8)	2	1
$28^2 \times 40$	TBS	(0.75, 4.5, 0.75)	(48, 80, 8)	1	2
$14^2 \times 80$	TBS	(0.75, 4.5, 0.75)	(48, 80, 8)	2	1
$14^2 \times 80$	TBS	(0.75, 4.5, 0.75)	(72, 112, 8)	3	1
$14^2 \times 112$	TBS	(0.75, 4.5, 0.75)	(112, 184, 8)	1	2
$7^2 \times 184$	TBS	(0.75, 4.5, 0.75)	(112, 184, 8)	3	1
$7^2 \times 184$	1x1	-	1984	1	1
$7^2 \times 1984$	avgpl	-	-	1	1
1984	fc	-	1000	1	-

Table 3.9: Macro-architecture for our parameter-efficient search space for **FBNetV2-P**. The maximum input resolution for FBNetV2-P is 288. See Table 3.7 for column names.

Max. Input	b	e	f	n	s
$288^2 \times 3$	3x3	1	32	1	2
$144^2 \times 16$	TBS	1	(16, 28, 4)	1	1
$144^2 \times 28$	TBS	(0.75, 4.5, 0.75)	(16, 40, 4)	1	2
$72^2 \times 40$	TBS	(0.75, 4.5, 0.75)	(16, 40, 4)	2	1
$72^2 \times 40$	TBS	(0.75, 4.5, 0.75)	(16, 48, 8)	1	2
$36^2 \times 48$	TBS	(0.75, 4.5, 0.75)	(16, 48, 8)	2	1
$36^2 \times 48$	TBS	(0.75, 4.5, 0.75)	(48, 96, 8)	1	2
$18^2 \times 96$	TBS	(0.75, 4.5, 0.75)	(48, 96, 8)	2	1
$18^2 \times 96$	TBS	(0.75, 4.5, 0.75)	(72, 128, 8)	4	1
$18^2 \times 128$	TBS	(0.75, 4.5, 0.75)	(112, 216, 8)	1	2
$9^2 \times 216$	TBS	(0.75, 4.5, 0.75)	(112, 216, 8)	3	1
$9^2 \times 216$	TBS	(0.75, 4.5, 0.75)	(112, 216, 8)	1	1
$9^2 \times 216$	1x1	-	1280	1	1
$9^2 \times 1280$	avgpl	-	-	1	1
1280	fc	-	1000	1	-

Chapter 4

Jointly Searching Neural Architectures and Training Recipes

4.1 Introduction

Neural Architecture Search (NAS) yields state-of-the-art neural networks that outperform their best manually-designed counterparts. However, previous NAS methods search for architectures under one set of training hyperparameters (i.e., a training recipe), overlooking superior architecture-recipe combinations. To address this, we present Neural Architecture-Recipe Search (NARS) to search both (a) architectures and (b) their corresponding training recipes, simultaneously. NARS utilizes an accuracy predictor that scores architecture **and** training recipes jointly, guiding both sample selection and ranking. Furthermore, to compensate for the enlarged search space, we leverage "free" architecture statistics (e.g., FLOP count) to pretrain the predictor, significantly improving its sample efficiency and prediction reliability. After training the predictor via constrained iterative optimization, we run fast evolutionary searches in just CPU minutes to generate architecture-recipe pairs for a variety of resource constraints, called FBNetV3. FBNetV3 makes up a family of state-of-the-art compact neural networks that outperform both automatically and manually-designed competitors. For example, FBNetV3 matches both EfficientNet and ResNeSt accuracy on ImageNet with up to 2.0× and 7.1× fewer FLOPs, respectively. Furthermore, FBNetV3 yields significant performance gains for downstream object detection tasks, improving mAP despite 18% fewer FLOPs and 34% fewer parameters than EfficientNet-based equivalents.[1]

[1]The contents of this work are in collaboration with Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Zijian He, Zhen Wei, Kan Chen, Yuandong Tian, Matthew Yu, Peter Vajda, and Joseph E. Gonzalez, presented at CVPR 2021 [20]. This was a part of my part-time research with Facebook Reality Labs and was supported by my National Science Foundation Graduate Research Fellowship under Grant No. DGE 1752814. In addition to NSF CISE Expeditions Award CCF-1730628, UC Berkeley research is supported by gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

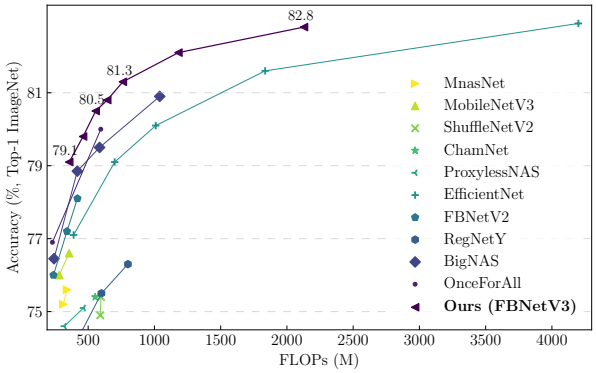


Figure 4.1: ImageNet accuracy vs. model FLOPs comparison of FBNetV3 with other efficient convolutional neural networks. FBNetV3 achieves 80.8% (82.8%) top-1 accuracy with 557M (2.1G) FLOPs, setting a new SOTA for accuracy-efficiency trade-offs.

Model \ Training	Recipe-1	Recipe-2
ResNet18 (1.4x width)	**70.8%**	73.3%
ResNet18 (2x depth)	70.7%	**73.8%**

Table 4.1: Different training recipe could switch the ranking of architectures. ResNet18 1.4x width and 2x depth refer to ResNet18 with 1.4 width and 2.0 depth scaling factor, respectively. Training recipe details can be found in Appendix 4.6.

4.2 Motivation

Designing efficient computer vision models is a challenging but important problem: A myriad of applications from autonomous vehicles to augmented reality require compact models that must be highly accurate – even under constraints on power, computation, memory, and latency. The number of possible constraint and architecture combinations is combinatorially large, making manual design a near impossibility.

In response, recent work employs neural architecture search (NAS) to design state-of-the-art efficient deep neural networks. One category of NAS is differentiable neural architecture search (DNAS). These path-finding algorithms are efficient, often completing a search in the time it takes to train one network. However, DNAS cannot search for non-architecture hyperparameters, which are crucial to the model's performance. Furthermore, supernet-based NAS methods suffer from a limited search space, as the entire supergraph must fit into memory to avoid slow convergence [12] or paging. Other methods include reinforcement learning (RL) [106], and evolutionary algorithms (ENAS) [86]. However, these methods share several drawbacks:

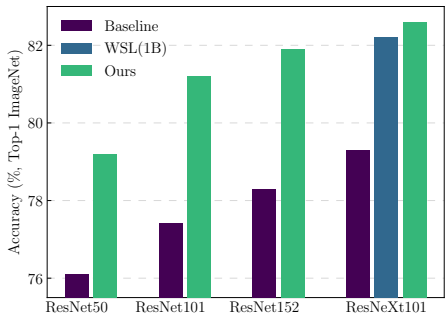


Figure 4.2: Accuracy improvement on existing architectures with the searched training recipe. WSL refers to the weakly supervised learning model using 1B additional images [72].

1. **Ignore training hyperparameters**: NAS, true to its name, searches only for architectures but not the associated training hyperparameters (i.e., "training recipe"). This ignores the fact that different training recipes may drastically change the success or failure of an architecture, or even switch architecture rankings (Table 4.1).

2. **Support only one-time use**: Many conventional NAS approaches produce one model for a specific set of resource constraints. This means that deploying to a line of products, each with different resource constraints, requires rerunning NAS once for each resource setting. Alternatively, model designers may search for one model and scale it suboptimally, using manual heuristics, to fit new resource constraints.

3. **Prohibitively large search space to search:** Naïvely including training recipes in the search space is either impossible (DNAS, supernet-based NAS) or prohibitively expensive, as architecture-only accuracy predictors are already computationally expensive to train (RL, ENAS).

To overcome these challenges, we propose Neural Architecture-Recipe Search (NARS) to address the above limitations. Our insight is three-fold: (1) To support re-use of NAS results for multiple resource constraints, we train an accuracy predictor, then use the predictor to find architecture-recipe pairs for new resource constraints in just CPU minutes. (2) To avoid the pitfalls of architecture-only or recipe-only searches, this predictor scores both training recipes and architectures simultaneously. (3) To avoid prohibitive growth in predictor training time, we pretrain the predictor on proxy datasets to predict architecture statistics (*e.g.* , FLOPs, #Parameters) from architecture representations. After sequentially performing predictor pretraining, constrained iterative optimization, and predictor-based evolutionary search, NARS produces generalizable training recipes and compact models that attain state-of-the-art performance on ImageNet, outperforming all the existing manually designed or automatically searched neural networks. We summarize our contributions below:

1. **Neural Architecture-Recipe Search**: We propose a predictor that jointly scores both training recipes and architectures, the first joint search, over *both* training recipes and architectures, at scale to our knowledge.

2. **Predictor pretraining**: To enable efficient search over this larger space, we furthermore present a pretraining technique, significantly improving the accuracy predictor's sample efficiency.

3. **Multi-use predictor**: Our predictor can be used in fast evolutionary searches to quickly generate models for a wide variety of resource budgets in just CPU minutes.

4. **State-of-the-art ImageNet accuracy** per FLOP for the searched FBNetV3 models. For example, our FBNetV3 matches EfficientNet accuracy with as low as 49.3% fewer FLOPs, as shown in Fig. 4.1.

5. **Generalizable training recipe**: NARS's recipe-only search achieves significant accuracy gains across various neural networks, as illustrated in Fig. 4.2. Our ResNeXt101-32x8d achieves 82.6% top-1 accuracy; this even outperforms its weakly-supervised counterpart trained on 1B extra images [72].

4.3 Joint Architecture-Recipe Search via Predictor Pretraining

Our goal is to find the most accurate architecture and training recipe combination, to avoid overlooking architecture-recipe pairs as prior methods have. However, the search space is typically combinatorially large, making exhaustive evaluation an impossibility. To address this, we train an accuracy predictor that accepts architecture and training recipe representations (Sec 4.3). To do so, we employ a three-stage pipeline (Algorithm 1): (1) Pretrain the predictor using architecture statistics, significantly improving its accuracy and sample efficiency (Sec 4.3). (2) Train the predictor using constrained iterative optimization (Sec 4.3). (3) For each set of resource constraints, run predictor-based evolutionary search in just CPU minutes to produce high-accuracy architecture-recipe pairs (Sec 4.3).

Predictor

Our predictor aims to predict accuracy given representations of an architecture and a training recipe. The architecture and training recipe are encoded using one-hot categorical variables (e.g., for block types) and min-max normalized continuous values (e.g., for channel counts). See the full search space in Table 4.2.

The predictor architecture is a multi-layer perceptron (Fig. 4.3) consisting of several fully-connected layers and two heads: (1) An auxiliary "proxy" head, used for pretraining the

Algorithm 1: Three-stage Constraint-aware Neural Architecture-Recipe Search

Input:
Ω: the designed search space;
n: size of candidate pool Λ in constrained iterative optimization;
m: the number of DNN candidates ($\mathcal{X}$) to train in each iteration;
T: the number of batches for constrained iterative optimization;
Stage 1: Pretrain Predictor
Generate a pool Λ with n samples with QMC sampling from the search space Ω;
Pretrain accuracy predictor u with architecture statistics;
Stage 2: Train Predictor (Constrained Iterative Optimization):
Initialize $\mathcal{D}_0$ as $\emptyset$;
for $t = 1, 2, ..., T$ **do**
  Find a batch of the most promising DNN candidates $\mathcal{X} \subset \Lambda$ based on predicted scores, $u(x)$;
  Evaluate all $x \in \mathcal{X}$ by training in parallel;
  if $t = 1$: Determine early stopping criteria;
  Update the dataset: $\mathcal{D}_t = \mathcal{D}_{t-1} \cup \{(x_1, acc(x_1)), (x_2, acc(x_2)), ...\}$;
  Retrain the accuracy predictor u on $\mathcal{D}_t$;
end
Stage 3: Use Predictor (Predictor-Based Evolutionary Search)
Initialize $\mathcal{D}^\star$ with p best-performing samples in $\mathcal{D}_T$ and q randomly generated samples paired with scores predicted by u;
Initialize $s^\star$ with the best score in $\mathcal{D}^\star$; set $s_0^\star = 0$; set $\epsilon = 10^{-6}$;
while $(s^\star - s_0^\star) > \epsilon$ **do**
  for $x \in \mathcal{D}^\star$ **do**
    Generate a set of children $\mathcal{C} \subset \Omega$ subject to resource constraints, by the adaptive genetic algorithm [19];
  end
  Augment $\mathcal{D}^\star$ with $\mathcal{C}$ paired with scores predicted by u;
  Select top K candidates from the augmented set to update $\mathcal{D}^\star$;
  Update the previous best ranking score by $s_0^\star = s^\star$;
  Update the current best ranking score $s^\star$ by the best predicted score in $\mathcal{D}^\star$.
end
Result: $\mathcal{D}^\star$, i.e., all the top K best samples with their predicted scores.

encoder, predicts architecture statistics (e.g., FLOPs and #Parameters) from architecture representations; and (2) the accuracy head, fine-tuned in constrained iterative optimization (Sec 4.3), predicts accuracy from joint representations of the architecture and training recipe.

Stage 1: Predictor pretraining

Training an accuracy predictor can be computationally expensive, as each training label is ostensibly a fully-trained architecture under a specific training recipe. To alleviate this, our insight is to first pretrain on a proxy task. The pretraining step can help the predictor to form a good internal representation of the inputs, therefore reducing the number of accuracy-architecture-recipe samples needed. This can significantly mitigate the search cost required.

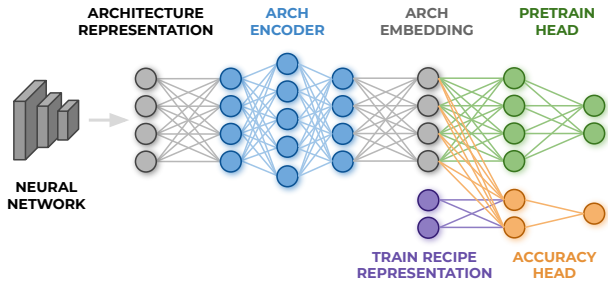
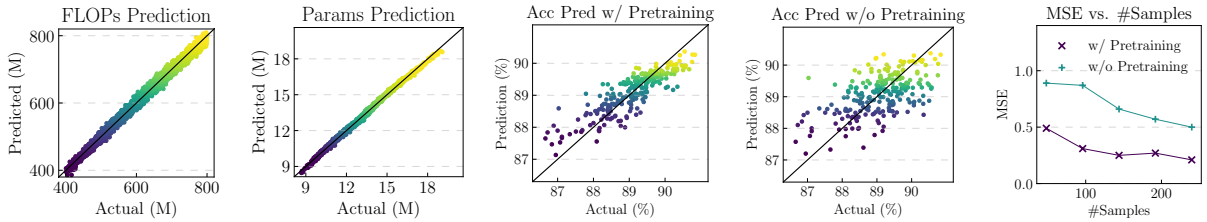


Figure 4.3: Pretrain to predict architecture statistics (top). Train to predict accuracy from architecture-recipe pairs (bottom)



(a) (b) (c) (d) (e)

Figure 4.4: (a) and (b): Predictor's performance on the proxy metrics, (c) and (d): Predictor's performance on accuracy with and without pretraining, (e): Predictor's MSE vs. number of samples with and without pretraining.

To construct a proxy task for pretraining, we can use "free" source of labels for architectures: namely, architecture statistics like FLOPs and numbers of parameters. After this pretraining step, we transfer the pretrained embedding layer to initialize the accuracy predictor (Fig. 4.3). This leads to significant improvements in the final predictor's sample efficiency and prediction reliability. For example, to reach the same prediction mean square error (MSE), the pretrained predictor only requires $5\times$ less samples than its counterpart without pretraining, as shown in Fig. 4.4(e). As a result, predictor pretraining reduces the overall search cost substantially.

Stage 2: Training predictor

In this step, we train the predictor and generate a set of high-promise candidates. As mentioned prior, our goal is to find the most accurate architecture and training recipe combination under given resource constraints. We thus formulate the architecture search as a constrained optimization problem:

$$\max_{(A,h)\in\Omega} \; acc(A,h), \text{ s.t. } g_i(A) \leqslant C_i, \;\; i = 1, ..., \gamma \tag{4.1}$$

where A, h, and Ω refer to the neural network architecture, training recipe, and designed search space, respectively. acc maps the architecture and training recipe to accuracy. $g_i(A)$

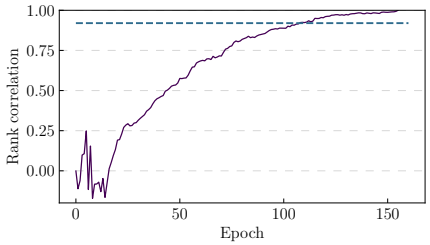


Figure 4.5: Rank correlation vs. epochs. Correlation threshold (cyan) is 0.92.

and γ refer to the formula and count of resource constraints, such as computational cost, storage cost, and run-time latency.

Constrained iterative optimization: We first use Quasi Monte-Carlo (QMC) [80] sampling to generate a sample pool of architecture-recipe pairs from the search space. Then, we train the predictor iteratively: We (a) shrink the candidate space by selecting a subset of favorable candidates based on predicted accuracy, (b) train and evaluate the candidates using an early-stopping heuristic, and (c) fine-tune the predictor with the Huber loss. This iterative shrinking of the candidate space avoids unnecessary evaluations and improves exploration efficiency.

- **Training candidates with early-stopping**. We introduce an early stopping mechanism to cut down on the computational cost of evaluating candidates. Specifically, we (a) rank samples by both early-stopping and final accuracy after the first iteration of constrained iterative optimization, (b) compute the rank correlation, and (c) find the epoch e where correlation exceeds a particular threshold (e.g., 0.92), as shown in Fig. 4.5.

  For all remaining candidates, we train (A, h) only for e epochs to approximate $acc(A, h)$. This allows us to use much fewer training iterations to evaluate each queried sample.

- **Training the predictor with Huber loss**. After obtaining the pretrained architecture embedding, we first train the predictor for 50 epochs with the embedding layer frozen. Then, we train the entire model with reduced learning rate for another 50 epochs. We adopt the Huber loss to train the accuracy predictor, i.e., $\mathcal{L} = 0.5(y - \hat{y})^2$ if $|y - \hat{y}| < 1$ else $|y - \hat{y}| - 0.5$, where y and $\hat{y}$ are the prediction and ground truth label, respectively. This prevents the model from being dominated by outliers, which shows can confound the predictor [116].

Stage 3: Using predictor

The third stage of the proposed method is an iterative process based on adaptive genetic algorithms [101]. The best-performing architecture-recipe pairs from the second stage are

inherited as part of the first generation candidates. In each iteration, we introduce mutations to the candidates and generate a set of children $\mathcal{C} \subset \Omega$ subject to given constraints. We evaluate the score for each child with the pretrained accuracy predictor u, and select top K highest-scoring candidates for the next generation. We compute the gain of the highest score after each iteration, and terminate the loop when the improvement saturates. Finally, the predictor-based evolutionary search produces high-accuracy neural network architectures and training recipes.

Note that with the accuracy predictor, searching for networks to fit different use scenarios only incurs negligible cost. This is because the accuracy predictor can be substantially reused under different resource constraints, while predictor-based evolutionary search takes just CPU minutes.

Predictor search space

Our search space consists of both training recipes and architecture configurations. The search space for training recipes features optimizer type, initial learning rate, weight decay, mixup ratio [136], drop out ratio, stochastic depth drop ratio [43], and whether or not to use model exponential moving average (EMA) [53]. Our architecture configuration search space is based on the inverted residual block [93] and includes input resolution, kernel size, expansion, number of channels per layer, and depth, as detailed in Table 4.2.

In recipe-only experiments, we only tune training recipes on a fixed architecture. However, for joint search, we search both training recipes and architectures, within the search space in Table 4.2. Overall, the space contains 10^{17} architecture candidates with 10^7 possible training recipes. Exploring such a vast search space for an optimal network architecture and its corresponding training recipe is non-trivial.

4.4 Experiments

In this section, we first validate our search method in a narrowed search space to discover the training recipe for a given network. Then, we evaluate our search method for joint search over architecture and training recipes. We use PyTorch [81], and conduct our search on the ImageNet 2012 classification dataset [23]. In the search process, we randomly sample 200 classes from the entire dataset to reduce the training time. Then, we randomly withhold 10K images from the 200-class training set as the validation set.

Recipe-only search

To establish that even modern NAS-produced architecture's performance can be further improved with better training recipe, we optimize over training recipes for a fixed architecture.

block	k	e	c	n	s	se	act.
Conv	3	-	(16, 24, 2)	1	2	-	hswish
MBConv	[3, 5]	1	(16, 24, 2)	(1, 4)	1	N	hswish
MBConv	[3, 5]	(4, 7) / (2, 5)	(20, 32, 4)	(4, 7)	2	N	hswish
MBConv	[3, 5]	(4, 7) / (2, 5)	(24, 48, 4)	(4, 7)	2	Y	hswish
MBConv	[3, 5]	$(4, 7)^1$ / $(2, 5)^2$	(56, 84, 4)	(4, 8)	2	N	hswish
MBConv	[3, 5]	$(4, 7)^1$ / $(2, 5)^2$	(96, 144, 4)	(6, 10)	1	Y	hswish
MBConv	[3, 5]	(4, 7)	(180, 224, 4)	(5, 9)	2	Y	hswish
MBConv	[3, 5]	6	(180, 224, 4)	1	1	Y	hswish
MBPool	[3, 5]	6	1984	1	-	-	hswish
FC	-	-	1000	1	-	-	-

res	lr(10^{-3})	optim	ema	p(10^{-2})	d(10^{-1})	m(10^{-1})	wd(10^{-6})
(224, 272, 8)	(20, 30)	[RMSProp, SGD]	[true, false]	(1, 31)	(10, 31)	(0, 41)	(7, 21)

Table 4.2: The network architecture configuration and search space in our experiments. MBConv, MBPool, k, e, c, n, s, se, and act. refer to the inverted residual block [93], efficient last stage [40], kernel size, expansion, #Channel, #Layers, stride, squeeze-and-excitation, and activation function, respectively. res, lr, optim, ema, p, d, m, and wd refer to resolution, initial learning rate, optimizer type, EMA, dropout ratio, stochastic depth drop probability, mixup ratio, and weight decay, respectively. Expansion on the left of the slash is used in the first block in the stage, while that on the right for the rest. Tuples of three values in parentheses represent the lowest value, highest, and steps; two-value tuples imply a step of 1, and tuples in brackets represent all available choices during search. Note that lr is multiplied by 4 if the optim chooses SGD. Architecture parameters with the same superscript share the same values during the search.

We adopt FBNetV2-L3 [111] (Appendix 4.6) as our base architecture, which is a DNAS searched architecture that achieves 79.1% top-1 accuracy with the original training method used in [111]. We set the sample pool size $n = 20K$, batch size $m = 48$ and iteration $T = 4$ in constrained iterative optimization. We train the sampled candidates for 150 epochs with a learning rate decay factor of 0.963 per epoch during the search, and train the final model with $3\times$ slower learning rate decay (i.e., 0.9875 per epoch). We show the distribution of samples at each round as well as the final searched result in our experiments in Fig. 4.6, where the first-round samples are randomly generated. The searched training recipe (Appendix 4.6) improves the accuracy of our base architecture by 0.8%.

We extend the NARS-searched training recipe to other commonly-used neural networks to further validate its generality. Although the NARS-searched training recipe was tailored to FBNetV2-L3, it generalizes surprisingly well, as shown in Table 4.3. The NARS-searched training recipe leads to substantial accuracy gains of up to 5.7% on ImageNet. In fact, ResNet50 outperforms the baseline ResNet152 by 0.9%. ResNeXt101-32x8d even surpasses the weakly supervised learning model, which is trained with 1 billion weakly-labeled images

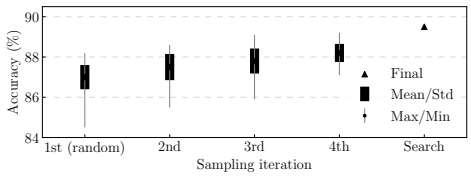


Figure 4.6: Illustration of the sampling and search process.

Model	Top-1 Accuracy (%)		
	Original	Recipe-only	Δ
FBNetV2-L3 [111]	79.1	79.9	+0.8
AlexNet [57]	56.6	62.3	+5.7
ResNet34 [35]	73.3	76.3	+3.0
ResNet50 [35]	76.1	79.2	+3.1
ResNet101 [35]	77.4	81.2	+3.8
ResNet152 [35]	78.3	81.9	+3.6
DenseNet201 [44]	77.2	80.2	+3.0
ResNeXt101 [123]	79.3	82.6	+3.3

Table 4.3: Accuracy improvements with the searched training recipes on existing neural networks. Above, ResNeXt101 refers to the 32x8d variant.

and achieves 82.2% top-1 accuracy. Notably, it is possible to achieve even better performance by searching for specific training recipe for each neural network, which would increase the search cost.

Neural Architecture-Recipe Search (NARS)

Search settings Next, we perform a joint search of architecture and training recipes to discover compact neural networks. Note that based on our observations in Sec. 4.4, we shrink the search space to always use EMA. Most of the settings are the same as in the recipe-only search, while we increase the optimization iteration $T = 5$ and set the FLOPs constraint for the sample pool from 400M to 800M. We pretrain the architecture embedding layer using 80% of the sample pool which contains 20K samples, and plot the validation on the rest 20% in Fig. 4.4. In the predictor-based evolutionary search, we set four different FLOPs constraints: 450M, 550M, 650M, and 750M and discover four models (namely FBNetV3-B/C/D/E) with the same accuracy predictor. We further scale down and up the minimum and maximum models and generate FBNetV3-A and FBNetV3-F/G to fit more use scenarios, respectively, with compound scaling proposed in [105].

Training setup For model training, we use a two-step distillation based training process: (1)

Model	Search method	Search space	Search cost (GPU/TPU hours)	FLOPs	Accuracy (%, Top-5)	Accuracy (%, Top-1)
FBNet [117]	gradient	arch	0.2K	375M	-	74.9
ProxylessNAS [12]	RL/gradient	arch	0.2K	465M	-	75.1
ChamNet [19]	predictor	arch	28K	553M	-	75.4
RegNetY [85]	pop. param.*	arch	11K	600M	-	75.5
MobileNetV3-1.25x [40]	RL/NetAdapt	arch	>91K	356M	-	76.6
EfficientNetB0 [105]	RL/scaling	arch	>91K	390M	93.3	77.3
AtomNAS [75]	gradient	arch	0.8K	363M	-	77.6
FBNetV2-L2 [111]	gradient	arch	0.6K	423M	-	78.1
FBNetV3-A	**NARS**	**arch/recipe**	10.7K	**357M**	**94.5**	**79.1**
ResNet152 [35]	manual	-	-	11G	93.8	78.3
EfficientNetB2 [105]	RL/scaling	arch	>91K	1.0G	94.9	80.3
ResNeXt101-32x8d [123]	manual	-	-	7.8G	94.5	79.3
Once-For-All [13]	gradient	-	-	595M	-	80.0
FBNetV3-C	**NARS**	**arch/recipe**	10.7K	**557M**	**95.1**	**80.5**
BigNASModel-XL [131]	gradient	arch	2.3K	1.0G	-	80.9
ResNeSt-50 [135]	manual	-	-	5.4G	-	81.1
FBNetV3-E	**NARS**	**arch/recipe**	10.7K	**762M**	**95.5**	**81.3**
EfficientNetB3 [105]	RL/scaling	arch	>91K	1.8G	95.7	81.7
ResNeSt-101 [135]	manual	-	-	10.2G	-	82.3
EfficientNetB4 [105]	RL/scaling	arch	>91K	4.2G	96.4	82.9
FBNetV3-G	**NARS**	**arch/recipe**	10.7K	**2.1G**	**96.3**	**82.8**

Table 4.4: Comparisons of different compact neural networks. For baselines, we cite statistics on ImageNet from the original papers. Our results are bolded. *: population parameterization. See 4.7 for discussions about the training tricks and additional EfficientNet comparisons.

We first train the largest model (i.e., FBNetV3-G) with the searched recipe with ground truth labels. (2) Then, we train all the models (including FBNetV3-G itself) with distillation, which is a typical training technique adopted in [13][131]. Different from the in-place distillation method in [13][131], the teacher model here is the ImageNet pretrained FBNetV3-G derived from step (1). The training loss is a sum of two components: Distillation loss scaled by 0.8 and cross entropy loss scaled by 0.2. During training, we use synchronized batch normalization in distributed training with 8 nodes and 8 GPUs per node. We train the models for 400 epochs with a learning rate decay factor of 0.9875 per epoch after a 5-epoch warmup. We train the scaled models FBNetV3-A and FBNetV3-F/G with the searched training recipes for FBNetV3-B and FBNetV3-E, respectively, only increasing the stochastic depth drop ratio for FBNetV3-F/G to 0.2. More training details can be found in Appendix 4.6.

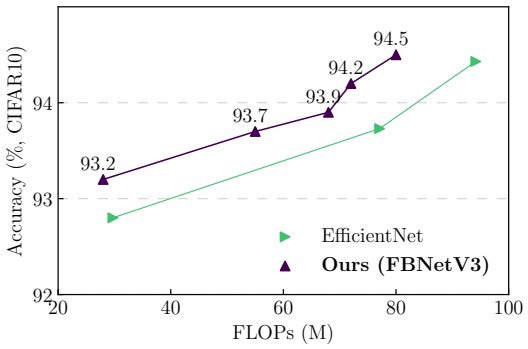


Figure 4.7: Accuracy vs. FLOPs comparison on the CIFAR-10 dataset.

Searched models We compare our searched model against other relevant NAS baselines and hand-crafted compact neural networks in Fig. 4.1, and list the detailed performance metrics comparison in Table 4.4, where we group the models by their top-1 accuracy. Among all the existing efficient models such as EfficientNet [105], MobileNetV3 [40], ResNeSt [135], and FBNetV2 [111], our searched model delivers substantial improvements on the accuracy-efficiency trade-off. For example, on low computation cost regime, FBNetV3-A achieves 79.1% top-1 accuracy with only 357M FLOPs (2.5% higher accuracy than MobileNetV3-1.25x [40] with similar FLOPs). On high accuracy regime, FBNetV3-E achieves 0.2 higher accuracy with over 7× fewer FLOPs compared to ResNeSt-50 [135], while FBNetV3-G achieves the same level of accuracy as EfficientNetB4 [105] with 2× fewer FLOPs. Note that we have further improved the accuracy of FBNetV3 by using larger teacher models for distillation, as shown in Appendix 4.7.

Transferability of the searched models

Classification on CIFAR-10 We further extend the searched FBNetV3 on CIFAR-10 dataset that has 60K images from 10 classes [56] to validate its transferability. Note that different from [105] that scales up the base input resolution to 224×224, we keep the original base input resolution as 32×32, and scale up the input resolutions for larger models based on the scaling ratio. We also replace the second stride-two block with a stride-one block to fit the low-resolution inputs. We don't include distillation for simplicity. We compared the performance of different models in Fig. 4.7. Again, our searched models significantly outperform the EfficientNet baselines.

Detection on COCO To further validate the transferability of the searched models on different tasks, we use FBNetV3 as a replacement for the backbone feature extractor for Faster R-CNN with the conv4 (C4) backbone and compare with other models on the COCO detection dataset. We adopt most of the training settings in [122] with 3× training iterations, while use synchronized batch normalization, initialize the learning rate at 0.16, switch on

Backbone	#Params (M)	FLOPs (G)	mAP
EfficientNetB0	8.0	3.6	30.2
FBNetV3-A	5.3	2.9	30.5
EfficeintNetB1	13.3	5.6	32.2
FBNetV3-E	10.6	5.3	33.0

Table 4.5: Object detection results of Faster RCNN with different backbones on COCO.

EMA, reduce the non-maximum suppression (NMS) to 75, and change to learning rate schedule to Cosine after warming up. Note that we only transfer the searched architectures and use the same training protocol for all the models.

We show the detailed COCO detection results in Table 4.5. With similar or higher mAP, our FBNetV3 reduces the FLOPs and number of parameters by up to 18.3% and 34.1%, respectively, compared to EfficientNet backbones.

4.5 Ablations

In this section, we revisit the performance improvements obtained from joint search, significance of the predictor-based evolutionary search, and the impact and generality of several training techniques.

Architecture and training recipe pairing. Our method yields different training recipes for different models. For example, we observe that smaller models tend to prefer less regularization (e.g., smaller stochastic depth drop ratio and mixup ratio). To illustrate the significance of neural architecture-recipe search, we swap the training recipes searched for FBNetV3-B and FBNetV3-E, observing a significant accuracy drop for both models, as shown in Table 4.6. This highlights the importance of correct architecture-recipe pairings, emphasizing the downfall of conventional NAS: Ignoring the training recipe and only searching for the network architecture fails to obtain optimal performance.

	FBNetV3-B Train recipe	FBNetV3-E Train recipe
FBNetV3-B Arch	**79.8%**	78.5%
FBNetV3-E Arch	80.8%	**81.3%**

Table 4.6: Accuracy comparison for the searched models with swapped training recipes.

Predictor-based evolutionary search improvements. Predictor-based evolutionary search yields substantial improvement on top of constrained iterative optimization. To demonstrate this, we compare the best-performing candidates derived from the second search stage with the final searched FBNetV3 under the same FLOPs constraints (Table 4.7). We observe an accuracy drop of up to 0.8% if the third stage is discarded. Thus, the third search stage, though requiring only negligible cost (i.e., several CPU minutes), is equally crucial to the final models' performance.

Model	Evolutionary Search	FLOPs	Accuracy
FBNetV3-B	Y	461M	79.8%
FBNetV3-B*	N	448M	79.0%
FBNetV3-E	Y	762M	81.3%
FBNetV3-E*	N	746M	80.7%

Table 4.7: Performance improvement by the predictor-based evolutionary searchsearch. *: Models derived from constrained iterative optimization.

Impact of distillation and model averaging We show the model performance on FBNetV3-G in Table 4.8 with different training configurations, where the baseline refers to the vanilla training without EMA or distillation. EMA brings substantially higher accuracy, especially during the middle stage of training. We hypothesize EMA intrinsically functions as a strong "ensemble" mechanism and thus improves single-model accuracy. We additionally observe distillation brings notable performance improvement. This is consistent with the observations in [13, 131]. Note since the teacher is a pretrained FBNetV3-G, FBNetV3-G is self-distilled. The combination of EMA and distillation improves the model's top-1 accuracy from 80.9% to 82.8%.

Model \ Training	Baseline	EMA	Dist*	Dist*+EMA
FBNetV3-G	80.9%	82.3%	82.2%	82.8%

Table 4.8: Performance improvement with EMA and distillation. *: Distillation-based training

4.6 Implementation

Training recipe used in Table 4.1

Both Recipe-1 and Recipe-2 share the same batch size of 256, initial learning rate 0.1, weight decay at 4×10^{-5}, SGD optimizer, and cosine learning rate schedule. Recipe-1 train the model for 30 epochs and Recipe-2 train the model for 90 epochs. We don't introduce training techniques such as dropout, stochastic depth, and mixup in Recipe-1 or Recipe-2.

We make the same observation when training Recipe-1 and Recipe-2 use the same #Epochs but different weight decay: The accuracy of ResNet18 (1.4x width) is 0.25% higher and 0.36% lower than that of ResNet18 (2x depth) when the weight decay is $1e^{-4}$ and $1e^{-5}$, respectively.

Base architecture in recipe-only search

We show the base architecture (a scaled version of FBNetV2-L2) used in the recipe-only search in Table 4.10, while the input resolution is 256×256. This is the base architecture used in the training recipe search in Section 4.4. It achieves 79.1% top-1 accuracy on ImageNet with the original training recipe used for FBNetV2. With the searched training recipes, it achieves 79.9% ImageNet top-1 accuracy.

Search settings and details

In the recipe-only search experiment, we set the early-stop rank correlation threshold to be 0.92, and find the corresponding early-stop epoch to be 103. In the predictor-based evolutionary search, we set the population of the initial generation to be 100 (50 best-performing candidates from constrained iterative optimizationand 50 randomly generated samples). We generate 24 children from each candidate and pick the top 40 candidates for the next generation. Most of the settings are shared by the joint search of architecture and training recipes, except the early-stop epoch to be 108. The accuracy predictor consists of one embedding layer (architecture encoder layer) and one extra hidden layer. The embedding width is 24 for the joint search (note that there is no pretrained embedding layer for the recipe-only search). We set both minimum and maximum FLOPs constraint at 400M and 800M for the joint search, respectively. The selection of m best-performing samples in the constrained iterative optimization involves two steps: (1) equally divide the FLOP range into m bins and (2) pick the sample with the highest predicted score within each bin.

We show the detailed searched training recipe in Table 4.9. We also release the searched models.

Notation	Value
lr	0.026
optim	RMSprop
ema	true
p	0.17
d	0.09
m	0.19
wd	7e-6

Table 4.9: Searched training recipe.

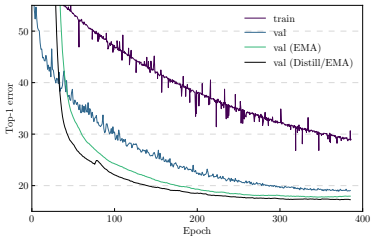


Figure 4.8: Training curve of the search recipe on FBNetV3-G.

Training settings and details

We use distributed training with 8 nodes for the final models, and scale up the learning rate by the number of distributed nodes (e.g., 8× for 8-node training). The batch size is set to be 256 per node. We use label smoothing and AutoAugment in the training. Additionally, we set the weight decay and momentum for batch normalization parameters to be zero and 0.9, respectively

We implement the EMA model as a copy of the original network (they share the same weights at $t = 0$). After each backward pass and model weights update, we update the EMA weights as

$$w_{t+1}^{ema} = \alpha w_t^{ema} + (1 - \alpha) w_{t+1} \tag{4.2}$$

where w_{t+1}^{ema}, w_t^{ema}, and w_{t+1} refer to the EMA weight at step $t + 1$, EMA weight at step t, and model weight at $t + 1$. We use an EMA decay α of 0.99985, 0.999, and 0.9998 in our experiments on ImageNet, CIFAR-10, and COCO, respectively. We further provide the training curves of FBNetV3-G in Fig. 4.8.

The baseline models (e.g., AlexNet, ResNet, DenseNet, and ResNeXt) are adopted from PyTorch open-source implementation without any architecture change. The input resolution is 224×224.

block	k	e	c	n	s	se	act.
Conv	1	3	16	1	2	-	hswish
MBConv	3	1	16	2	1	N	hswish
MBConv	5	5.46	24	1	2	N	hswish
MBConv	5	1.79	24	1	1	N	hswish
MBConv	3	1.79	24	1	1	N	hswish
MBConv	5	1.79	24	2	1	N	hswish
MBConv	5	5.35	40	1	2	Y	hswish
MBConv	5	3.54	32	1	1	Y	hswish
MBConv	5	4.54	32	3	1	Y	hswish
MBConv	5	5.71	72	1	2	N	hswish
MBConv	3	2.12	72	1	1	N	hswish
Skip	-	-	72	-	-	-	hswish
MBConv	3	3.12	72	1	1	N	hswish
MBConv	3	5.03	128	1	1	N	hswish
MBConv	5	2.51	128	1	1	Y	hswish
MBConv	5	1.77	128	1	1	Y	hswish
MBConv	5	2.77	128	1	1	Y	hswish
MBConv	5	3.77	128	4	1	Y	hswish
MBConv	3	5.57	208	1	2	Y	hswish
MBConv	5	2.84	208	2	1	Y	hswish
MBConv	5	4.88	208	3	1	Y	hswish
Skip	-	-	248	-	-	-	hswish
MBPool	-	6	1984	1	-	-	hswish
FC	-	-	1000	1	-	-	-

Table 4.10: Baseline architecture used in the recipe-only search. The block notations are identical to Table 4.2. Skip block refers to an identity connection if the input and output channel are equal otherwise a 1×1 conv.

4.7 Discussion

Comparison between recipe-only search and hyperparameter optimizers

Many well-known hyperparameter optimizers (ASHA, Hyberband, PBT) evaluate on CIFAR10. One exception is [76], which reports a 0.5% gain for ResNet50 on ImageNet by searching optimizers, learning rate, weight decay, and momentum. By contrast, our recipe-only search with the same space (without EMA) increases ResNet50 accuracy by 1.9%, from 76.1% to 78.0%.

Model	Distillation	FLOPs	Acc. (%)	Δ
EfficientNetB2	N	1050	80.3	0.0
FBNetV3-E	N	762	80.4	+0.1
FBNetV3-E	Y	762	81.3	+1.0

Table 4.11: Model comparison w/ and w/o distillation.

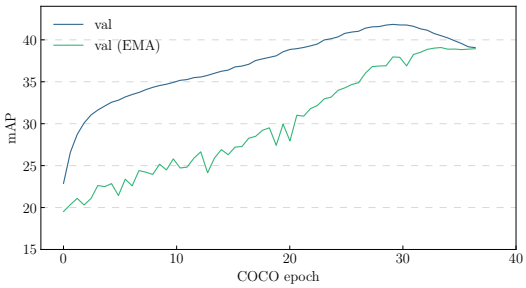


Figure 4.9: Training curves for RetinaNet with ResNet101 backbone on COCO object detection.

More discussions on training tricks

We acknowledge EfficientNet does not use distillation. For fair comparison, we report FBNetV3 accuracy without distillation. We provide an example in Table 4.11: Without distillation, FBNetV3 achieves higher accuracy with 27% less FLOPs, compared to EfficientNet. However, all our training tricks (including EMA and distillation) are used in the other baselines, including BigNAS and OnceForAll.

Generality of *stochastic weight averaging via EMA*. We observe that *stochastic weight averaging via EMA* yields significant accuracy gain for the classification tasks, as has been noted prior [10, 36]. We hypothesize that such a mechanism could be used as a general technique to improve other DNN models. To validate this, we employ it to train a RetinaNet [61] on COCO object detection [62] with ResNet50 and ResNet101 backbones. We follow most of the default training settings but introduce EMA and Cosine learning rate. We observe similar training curves and behavior as the classification tasks, as shown in Fig. 4.9. The generated RetinaNets with ResNet50 and ResNet101 backbones achieve 40.3 and 41.9 mAP, respectively, both substantially outperform the best reported values in [122] (38.7 and 40.4 for ResNet50 and ResNet101, respectively). A promising future direction is to study such techniques and extend it to other DNNs and applications.

Further improvements on FBNetV3

We demonstrate that using a teacher model with higher accuracy leads to further accuracy gain on FBNetV3. We use RegNetY-32G FLOPs (top-1 accuracy 84.5%) [26] as the teacher

model, and distill all the FBNetV3 models. We show all the derived models in Fig. 4.10, where we observe a consistent accuracy gain at 0.2% - 0.5% for all the models.

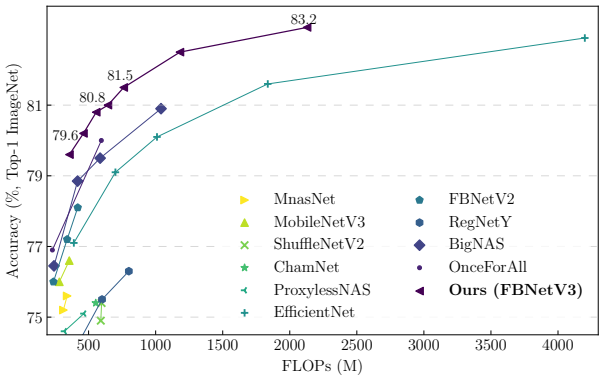


Figure 4.10: ImageNet accuracy vs. model FLOPs comparison of FBNetV3 (distilled from giant RegNet-Y models) with other efficient convolutional neural networks.

4.8 Conclusion

True to their name, previous neural architecture search methods search only over architectures, using a fixed set of training hyperparameters (i.e., "training recipe"). As a result, previous methods overlook higher-accuracy architecture-recipe combinations. However, our NARS does not, being the first algorithm to jointly search over both architectures and training recipes simultaneously for a large dataset like ImageNet. Critically, NARS's predictor pretrains on "free" architecture statistics–namely, FLOPs and #Parameters–to improve the predictor's sample efficiency significantly. After training and using the predictor, the resulting FBNetV3 architecture-recipe pairs attain state-of-the-art per-FLOP accuracies on ImageNet classification. We furthermore show generalizable training hyperparameters and a number of ablation studies to support our design choices. These results pave the way for joint architecture-recipe searches in the future, marrying neural architecture search and hyperparameter tuning algorithms under one framework.

Chapter 5

Jointly Improving Accuracy, Generalization, and Explainability

5.1 Introduction

Machine learning applications such as finance and medicine demand accurate and justifiable predictions, barring most deep learning methods from use. In response, previous work combines decision trees with deep learning, yielding models that (1) sacrifice interpretability for accuracy or (2) sacrifice accuracy for interpretability. We forgo this dilemma by *jointly improving* accuracy and interpretability using Neural-Backed Decision Trees (NBDTs). NBDTs replace a neural network's final linear layer with a differentiable sequence of decisions and a surrogate loss. This forces the model to learn high-level concepts and lessens reliance on highly-uncertain decisions, yielding (1) accuracy: NBDTs match or outperform modern neural networks on CIFAR, ImageNet and better generalize to unseen classes by up to 16%. Furthermore, our surrogate loss improves the *original* model's accuracy by up to 2%. NBDTs also afford (2) interpretability: improving human trust by clearly identifying model mistakes and assisting in dataset debugging. Code and pretrained NBDTs are at github.com/alvinwan/neural-backed-decision-trees.[1]

[1]The contents of this work are in collaboration with Lisa Dunlap, Daniel Ho, Jihan Yin, Scott Lee, Henry Jin, Suzanne Petryk, Sarah Adel Bargal, and Joseph E. Gonzalez, presented at ICLR 2021 [112]. This research was supported by my National Science Foundation Graduate Research Fellowship under Grant No. DGE 1752814. In addition to NSF CISE Expeditions Award CCF-1730628, UC Berkeley research is supported by gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

5.2 Motivation

Many computer vision applications (e.g. medical imaging and autonomous driving) require insight into the model's decision process, complicating applications of deep learning which are traditionally black box. Recent efforts in explainable computer vision attempt to address this need and can be grouped into one of two categories: (1) saliency maps and (2) sequential decision processes. Saliency maps retroactively explain model predictions by identifying which pixels most affected the prediction. However, by focusing on the input, saliency maps fail to capture the model's decision making process. For example, saliency offers no insight for a misclassification when the model is "looking" at the right object for the wrong reasons. Alternatively, we can gain insight into the model's decision process by breaking up predictions into a sequence of smaller semantically meaningful decisions as in rule-based models like decision trees. However, existing efforts to fuse deep learning and decision trees suffer from (1) significant accuracy loss, relative to contemporary models (e.g., residual networks), (2) reduced interpretability due to accuracy optimizations (e.g., impure leaves and ensembles), and (3) tree structures that offer limited insight into the model's credibility.

To address these, we propose **Neural-Backed Decision Trees (NBDTs)** to jointly improve *both* (1) accuracy and (2) interpretability of modern neural networks, utilizing decision rules that preserve (3) properties like sequential, discrete decisions; pure leaves; and non-ensembled predictions. These properties in unison enable unique insights, as we show. We acknowledge that there is no universally-accepted definition of interpretability [70, 27, 63], so to show interpretability, we adopt a definition offered by [84]: A model is interpretable if a human can validate its prediction, determining when the model has made a sizable mistake. We picked this definition for its importance to downstream benefits we can evaluate, specifically (1) model or dataset debugging and (2) improving human trust. To accomplish this, NBDTs replace the final linear layer of a neural network with a differentiable oblique decision tree and, unlike its predecessors (*i.e.* decision trees, hierarchical classifiers), uses a hierarchy derived from model parameters, does not employ a hierarchical softmax, and can be created from *any* existing classification neural network without architectural modifications. These improvements tailor the hierarchy to the network rather than overfit to the feature space, lessens the decision tree's reliance on highly uncertain decisions, and encourages accurate recognition of high-level concepts. These benefits culminate in joint improvement of accuracy and interpretability. Our contributions:

1. We propose a *tree supervision loss*, yielding NBDTs that match/outperform and outgeneralize modern neural networks (WideResNet, EfficientNet) on ImageNet, Tiny-ImageNet200, and CIFAR100. Our loss also improves the *original* model by up to 2%.

2. We propose alternative hierarchies for oblique decision trees – *induced hierarchies* built

using pre-trained neural network weights – that outperform both data-based hierarchies (e.g. built with information gain) and existing hierarchies (e.g. WordNet), in accuracy.

3. We show NBDT explanations are more helpful to the user when identifying model mistakes, preferred when using the model to assist in challenging classification tasks, and can be used to identify ambiguous ImageNet labels.

5.3 Neural-Backed Decision Trees

Neural-Backed Decision Trees (NBDTs) replace a network's final linear layer with a decision tree. Unlike classical decision trees or many hierarchical classifiers, NBDTs use path probabilities for inference (Sec 5.3) to tolerate highly-uncertain intermediate decisions, build a hierarchy from pre-trained model weights (Sec 5.3 & 5.3) to lessen overfitting, and train with a hierarchical loss (Sec 5.3) to significantly better learn high-level decisions (e.g., *Animal* vs. *Vehicle*).

Inference

Our NBDT first featurizes each sample using the neural network backbone; the backbone consists of all neural network layers before the final linear layer. Second, we run the final fully-connected layer as an oblique decision tree. However, (a) a classic decision tree cannot recover from a mistake early in the hierarchy and (b) just running a classic decision tree on neural features drops accuracy significantly, by up to 11% (Table 5.2). Thus, we present modified decision rules (Figure 5.1, B):

1. Seed oblique decision rule weights with neural network weights. An oblique decision tree supports only binary decisions, using a hyperplane for each decision. Instead, we associate a weight vector n_i with each node. For leaf nodes, where $i = k \in [1, K]$, each $n_i = w_k$ is a row vector from the fully-connected layer's weights $W \in \mathbb{R}^{D \times K}$. For all inner nodes, where $i \in [K+1, N]$, find all leaves $k \in L(i)$ in node i's subtree and average their weights: $n_i = \sum_{k \in L(i)} w_k / |L(i)|$.

2. Compute node probabilities. Child probabilities are given by softmax inner products. For each sample x and node i, compute the probability of each child $j \in C(i)$ using $p(j|i) = \textsc{Softmax}(\langle \vec{n}_i, x \rangle)[j]$, where $\vec{n}_i = (\langle n_j, x \rangle)_{j \in C(i)}$.

3. Pick a leaf using path probabilities. Inspired by [24], consider a leaf, its class k and its path from the root P_k. The probability of each node $i \in P_k$ traversing the next node in the path $C_k(i) \in P_k \cap C(i)$ is denoted $p(C_k(i)|i)$. Then, the probability of leaf and its class k is

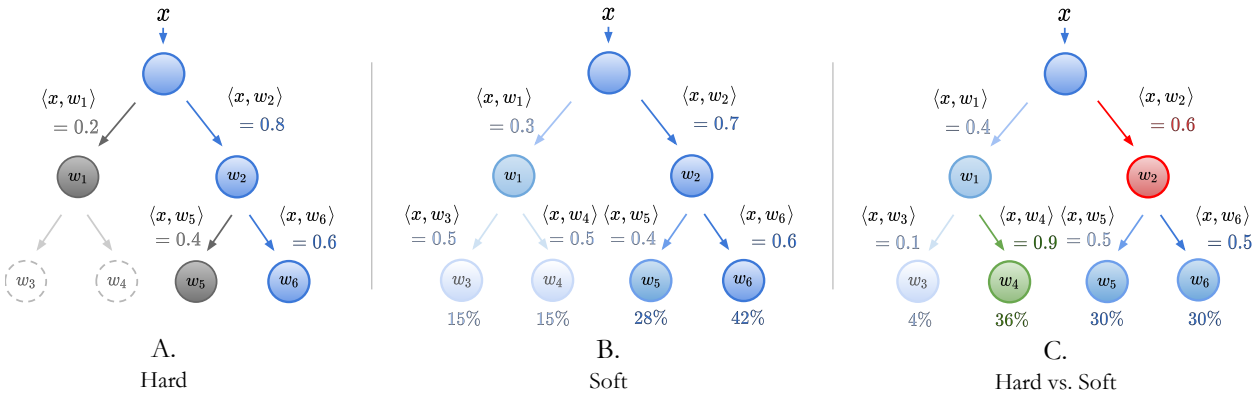


Figure 5.1: Hard and Soft Decision Trees. A. Hard: is the classic "hard" oblique decision tree. Each node picks the child node with the largest inner product, and visits that node next. Continue until a leaf is reached. **B. Soft:** is the "soft" variant, where each node simply returns probabilities, as normalized inner products, of each child. For each leaf, compute the probability of its path to the root. Pick leaf with the highest probability. **C. Hard vs. Soft:** Assume w_4 is the correct class. With hard inference, the mistake at the root (red) is irrecoverable. However, with soft inference, the highly-uncertain decisions at the root and at w_2 are superseded by the highly certain decision at w_3 (green). This means the model can still correctly pick w_4 despite a mistake at the root. In short, soft inference can tolerate mistakes in highly uncertain decisions.

$$p(k) = \Pi_{i \in P_k} p(C_k(i)|i) \tag{5.1}$$

In soft inference, the final class prediction $\hat{k}$ is defined over these class probabilities,

$$\hat{k} = \text{argmax}_k p(k) = \text{argmax}_k \Pi_{i \in P_k} p(C_k(i)|i) \tag{5.2}$$

Our inference strategy has two benefits: (a) Since the architecture is unchanged, the fully-connected layer can be run regularly (Table 5.5) or as decision rules (Table 5.1), and (b) unlike decision trees and other conditionally-executed models [107, 110], our method can recover from a mistake early in the hierarchy with sufficient uncertainty in the incorrect path (Figure 5.1 C, Appendix Table 5.7). This inference mode bests classic tree inference (Appendix 5.7).

Building Induced Hierarchies

Existing decision-tree-based methods use (a) hierarchies built with data-dependent heuristics like information gain or (b) existing hierarchies like WordNet. However, the former overfits to the data, and the latter focuses on conceptual rather than visual similarity: For example,

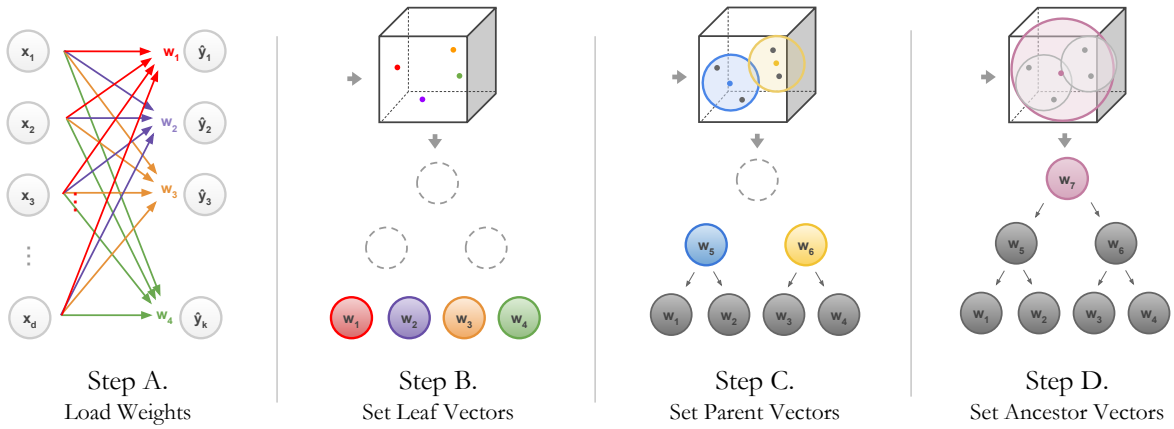


Figure 5.2: Building Induced Hierarchies. Step A. Load the weights of a pre-trained model's final fully-connected layer, with weight matrix $W \in \mathbb{R}^{D \times K}$. **Step B.** Take rows $w_k \in W$ and normalize for each leaf node's weight. For example, the red w_1 in A is assigned to the red leaf in B. **Step C.** Average each pair of leaf nodes for the parents' weight. For example, w_1 and w_2 (red and purple) in B are averaged to make w_5 (blue) in C. **Step D.** For each ancestor, average all leaf node weights in its subtree. That average is the ancestor's weight. Here, the ancestor *is* the root, so its weight is the average of all leaf weights w_1, w_2, w_3, w_4.

by virtue of being an animal, *Bird* is closer to *Cat* than to *Plane*, according to WordNet. However, the opposite is true for visual similarity: by virtue of being in the sky, *Bird* is more visually similar to *Plane* than to *Cat*. Thus, to prevent overfitting and reflect visual similarity, we build a hierarchy using model weights.

Our hierarchy requires pre-trained model weights. Take row vectors $w_k : k \in [1, K]$, each representing a class, from the fully-connected layer weights W. Then, run hierarchical agglomerative clustering on the normalized class representatives $w_k / \|w_k\|_2$. Agglomerative clustering decides which nodes and groups of nodes are iteratively paired. As described in Sec 5.3, each leaf node's weight is a row vector $w_k \in W$ (Figure 5.2, Step B) and each inner node's weight n_i is the average of its leaf node's weights (Figure 5.2, Step C). This hierarchy is the *induced hierarchy* (Figure 5.2).

Labeling Decision Nodes with WordNet

WordNet is a hierarchy of nouns. To assign WordNet meaning to nodes, we compute the earliest common ancestor for all leaves in a subtree: For example, say *Dog* and *Cat* are two leaves that share a parent. To find WordNet meaning for the parent, find all ancestor concepts that *Dog* and *Cat* share, like *Mammal*, *Animal*, and *Living Thing*. The earliest shared ancestor is *Mammal*, so we assign *Mammal* to the parent of *Dog* and *Cat*. We repeat for all inner nodes.

However, the WordNet corpus is lacking in concepts that are not themselves objects, like object attributes (e.g., *Pencil* and *Wire* are both cylindrical) and (b) abstract visual ideas like context (e.g., *fish* and *boat* are both aquatic). Many of these which are littered across our induced hierarchies (Appendix Figure 5.14). Despite this limitation, we use WordNet to assign meaning to intermediate decision nodes, with more sophisticated methods left to future work.

Fine-tuning with Tree Supervision Loss

Even though standard cross entropy loss separates representatives for each leaf, it is not trained to separate representatives for each inner node (Table 5.3, "None"). To amend this, we add a *tree supervision loss*, a cross entropy loss over the class distribution of path probabilities $\mathcal{D}_{\text{nbdt}} = \{p(k)\}_{k=1}^{K}$ (Eq. 5.1) from Sec 5.3, with time-varying weights ω_t, β_t where t is the epoch count:

$$\mathcal{L} = \beta_t \underbrace{\textsc{CrossEntropy}(\mathcal{D}_{\text{pred}}, \mathcal{D}_{\text{label}})}_{\mathcal{L}_{\text{original}}} + \omega_t \underbrace{\textsc{CrossEntropy}(\mathcal{D}_{\text{nbdt}}, \mathcal{D}_{\text{label}})}_{\mathcal{L}_{\text{soft}}} \tag{5.3}$$

Our tree supervision loss $\mathcal{L}_{\text{soft}}$ requires a pre-defined hierarchy. We find that (a) tree supervision loss damages learning speed early in training, when leaf weights are nonsensical. Thus, our tree supervision weight ω_t grows linearly from $\omega_0 = 0$ to $\omega_T = 0.5$ for CIFAR10, CIFAR100, and to $\omega_T = 5$ for TinyImageNet, ImageNet; $\beta_t \in [0, 1]$ decays linearly over time. (b) We re-train where possible, fine-tuning with $\mathcal{L}_{\text{soft}}$ only when the original model accuracy is not reproducible. (c) Unlike hierarchical softmax, our path-probability cross entropy loss $\mathcal{L}_{\text{soft}}$ disproportionately up-weights decisions earlier in the hierarchy, encouraging accurate high-level decisions; this is reflected our out-generalization of the baseline neural network by up to 16% to unseen classes (Table 5.6).

5.4 Experiments

NBDTs obtain state-of-the-art results for interpretable models and match or outperform modern neural networks on image classification. We report results on different models (ResNet, WideResNet, EfficientNet) and datasets (CIFAR10, CIFAR100, TinyImageNet, ImageNet). We additionally conduct ablation studies to verify the hierarchy and loss designs, find that our training procedure improves the *original* neural network's accuracy by up to 2%, and show that NBDTs improve generalization to unseen classes by up to 16%. All reported improvements are absolute.

Table 5.1: Results. NBDT outperforms competing decision-tree-based methods by up to 18% and can also outperform the *original* neural network by $\sim 1\%$. "Expl?" indicates the method retains interpretable properties: pure leaves, sequential decisions, non-ensemble. Methods without this check see reduced interpretability. We bold the highest decision-tree-based accuracy. These results are taken directly from the original papers (*n/a* denotes results missing from original papers): XOC [2], DCDJ [3], NofE [1], DDN [79], ANT [107], CNN-RNN [31]. We train DNDF [54] with an updated R18 backbone, as they did not report CIFAR accuracy.

Method	Backbone	Expl?	CIFAR10	CIFAR100	TinyImageNet
NN	WideResNet28x10	✗	*97.62%*	*82.09%*	*67.65%*
ANT-A*	*n/a*	✓	93.28%	*n/a*	*n/a*
DDN	NiN	✗	90.32%	68.35%	*n/a*
DCDJ	NiN	✗	*n/a*	69.0%	*n/a*
NofE	ResNet56-4x	✗	*n/a*	76.24%	*n/a*
CNN-RNN	WideResNet28x10	✓	*n/a*	76.23%	*n/a*
NBDT-S (Ours)	WideResNet28x10	✓	**97.55%**	**82.97%**	**67.72%**
NN	ResNet18	✗	*94.97%*	*75.92%*	*64.13%*
DNDF	ResNet18	✗	94.32%	67.18%	44.56%
XOC	ResNet18	✓	93.12%	*n/a*	*n/a*
DT	ResNet18	✓	93.97%	64.45%	52.09%
NBDT-S (Ours)	ResNet18	✓	**94.82%**	**77.09%**	**64.23%**

Results

Small-scale Datasets. Our method (Table 5.1) matches or outperforms recently state-of-the-art neural networks. On CIFAR10 and TinyImageNet, NBDT accuracy falls within 0.15% of the baseline neural network. On CIFAR100, NBDT accuracy outperforms the baseline by $\sim$1%.

Large-scale Dataset. On ImageNet (Table 5.3), NBDTs obtain 76.60% top-1 accuracy, outperforming the strongest competitor NofE by 15%. Note that we take the best competing results for any decision-tree-based method, but the strongest competitors hinder interpretability by using ensembles of models like a decision forest (DNDF, DCDJ) or feature shallow trees with only depth 2 (NofE).

Analysis

Analyses show that our NBDT improvements are dominated by significantly improved ability to distinguish higher-level concepts (e.g., *Animal* vs. *Vehicle*).

Figure 5.3: ImageNet Results. NBDT outperforms all competing decision-tree-based methods by at least 14%, staying within 0.6% of EfficientNet accuracy. "EfficientNet" is EfficientNet-EdgeTPU-Small.

Method	NBDT (ours)	NBDT (ours)	XOC	NofE
Backbone	EfficientNet	ResNet18	ResNet152	AlexNet
Original Acc	77.23%	60.76%	78.31%	56.55%
Delta Acc	-0.63%	+0.50%	-17.5%	+4.7%
Explainable Acc	**76.60%**	61.26%	60.77%	61.29%

Table 5.2: Comparisons of Hierarchies. We demonstrate that our weight-space hierarchy bests taxonomy and data-dependent hierarchies. In particular, the induced hierarchy achieves better performance than (a) the WordNet hierarchy, (b) a classic decision tree's information gain hierarchy, built over neural features ("Info Gain"), and (c) an oblique decision tree built over neural features ("OC1").

Dataset	Backbone	Original	Induced	Info Gain	WordNet	OC1
CIFAR10	ResNet18	*94.97%*	**94.82%**	93.97%	94.37%	94.33%
CIFAR100	ResNet18	*75.92%*	**77.09%**	64.45%	74.08%	38.67%
TinyImageNet200	ResNet18	*64.13%*	**64.23%**	52.09%	60.26%	15.63%

Comparison of Hierarchies. Table 5.2 shows that our induced hierarchies outperform alternatives. In particular, *data-dependent* hierarchies overfit, and the existing *WordNet hierarchy* focuses on conceptual rather than visual similarity.

Comparisons of Losses. Previous work suggests hierarchical softmax (Appendix 5.7) is necessary for hierarchical classifiers. However, our results suggest otherwise: NBDTs trained with hierarchical softmax see ∼3% less accuracy than with tree supervision loss on TinyImageNet (Table 5.3).

Original Neural Network. Per Sec 5.3, we can run the original neural network's fully-connected layer normally, after training with tree supervision loss. Using this, we find that the original neural network's accuracy improves by up to 2% on CIFAR100, TinyImageNet (Table 5.5).

Zero-Shot Superclass Generalization. We define a "superclass" to be the hypernym of several classes. (e.g. *Animal* is a superclass of *Cat* and *Dog*). Using WordNet (per Sec 5.3), we (1) identify which superclasses each NBDT inner node is deciding between (e.g.

Table 5.3: Comparisons of Losses. Training the NBDT using tree supervision loss with a linearly increasing weight ("TreeSup(t)") is superior to training (a) with a constant-weight tree supervision loss ("TreeSup"), (b) with a hierarchical softmax ("HrchSmax") and (c) without extra loss terms. ("None"). Δ is the accuracy difference between our soft loss and hierarchical softmax.

Dataset	Backbone	Original	TreeSup(t)	TreeSup	None	HrchSmax
CIFAR10	ResNet18	*94.97%*	**94.82%**	94.76%	94.38%	93.97%
CIFAR100	ResNet18	*75.92%*	**77.09%**	74.92%	61.93%	74.09%
TinyImageNet200	ResNet18	*64.13%*	**64.23%**	62.74%	45.51%	61.12%

Table 5.4: Mid-Training Hierarchy. Constructing and using hierarchies early and often in training yields the highest performing models. All experiments use ResNet18 backbones. Per Sec 5.3, β_t, ω_t are the loss term coefficients. Hierarchies are reconstructed every "Period" epochs, starting at "Start" and ending at "End".

Hierarchy Updates			CIFAR10			CIFAR100		
Start	End	Period	NBDT	NN+TSL	NN	NBDT	NN+TSL	NN
67	120	10	**94.88%**	**94.97%**	94.97%	**76.04%**	**76.56%**	75.92%
90	140	10	94.29%	94.84%	94.97%	75.44%	76.29%	75.92%
90	140	20	94.52%	94.89%	94.97%	75.08%	76.11%	75.92%
120	121	10	94.52%	94.92%	94.97%	74.97%	75.88%	75.92%

Animal vs. Vehicle). (2) We find unseen classes that belong to the same superclass, from a different dataset. (e.g. Pull *Turtle* images from ImageNet). (3) Evaluate the model to ensure the unseen class is classified into the correct superclass (e.g. ensure *Turtle* is classified as *Animal*). For an NBDT, this is straightforward: one of the inner nodes classifies *Animal vs. Vehicle* (Sec 5.3). For a standard neural network, we consider the superclass that the final prediction belongs to. (*i.e.* When evaluating *Animal vs. Vehicle* on a *Turtle* image, the CIFAR-trained model may predict any CIFAR *Animal* class). See Appendix 5.6 for details. Our NBDT consistently bests the original neural network by 8%+ (Table 5.6). When discerning *Carnivore vs. Ungulate*, NBDT outperforms the original neural network by 16%.

Mid-Training Hierarchy: We test NBDTs without using pre-trained weights, instead constructing hierarchies during training from the partially-trained network's weights. Tree supervision loss with mid-training hierarchies reliably improve the original neural network's accuracy, up to ∼0.6%, and the NBDT itself can match the original neural network's accuracy (Table 5.4). However, this underperforms NBDT (Table 5.1), showing fully-trained weights are still preferred for hierarchy construction.

Table 5.5: Original Neural Network. We compare the model's accuracy before and after the tree supervision loss, using ResNet18, WideResNet on CIFAR100, TinyImageNet. Our loss increases the original network accuracy consistently by $\sim .8 - 2.4\%$. NN-S is the network trained with the tree supervision loss.

Dataset	Backbone	NN	NN-S
C100	R18	75.92%	**76.96%**
T200	R18	64.13%	**66.55%**
C100	WRN28	82.09%	**82.87%**
T200	WRN28	67.65%	**68.51%**

Table 5.6: Zero-Shot Superclass Generalization. We evaluate a CIFAR10-trained NBDT (ResNet18 backbone) inner node's ability to generalize beyond seen classes. We label TinyImageNet with superclass labels (e.g. label *Dog* with *Animal*) and evaluate nodes distinguishing between said superclasses. We compare to the baseline ResNet18: check if the prediction is within the right superclass.

n_{class}	Superclasses	R18	NBDT-S
71	Animal *vs.* Vehicle	66.08%	**74.79%**
36	Placental *vs.* Vertebrate	45.50%	**54.89%**
19	Carnivore *vs.* Ungulate	51.37%	**67.78%**
9	Motor Vehicle *vs.* Craft	69.33%	**77.78%**

5.5 Interpretability

By breaking complex decisions into smaller intermediate decisions, decision trees provide insight into the decision process. However, when the intermediate decisions are themselves neural network predictions, extracting insight becomes more challenging. To address this, we adopt benchmarks and an interpretability definition offered by [84]: A model is interpretable if a human can validate its prediction, determining when the model has made a sizable mistake. To assess this, we adapt [84]'s benchmarks to computer vision and show (a) humans can identify misclassifications with NBDT explanations more accurately than with saliency explanations (Sec 5.5), (b) a way to utilize NBDT's entropy to identify ambiguous labels (Sec. 5.5), and (c) that humans prefer to agree with NBDT predictions when given a challenging image classification task (Sec. 5.5 & 5.5). Note that these analyses depend on three model properties that NBDT preserves: (1) discrete, sequential decisions, so that one path is selected; (2) pure leaves, so that one path picks one class; and (3) non-ensembled predictions, so that path to prediction attribution is discrete. In all surveys, we use CIFAR10-trained models with ResNet18 backbones.

Survey: Identifying Faulty Model Predictions

In this section we aim to answer a question posed in [84] *"How well can someone detect when the model has made a sizable mistake?"*. In this survey, each user is given 3 images, 2 of which are correctly classified and 1 is mis-classified. Users must predict which image was incorrectly classified given a) the model explanations and b) *without* the final prediction. For saliency maps, this is a near-impossible task as saliency usually highlights the main object in the image, regardless of wrong or right. However, hierarchical methods provide a sensible sequence of intermediate decisions that can be checked. This is reflected in the results: For

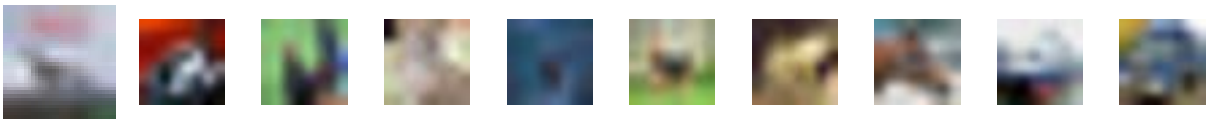


Figure 5.4: CIFAR10 Blurry Images. To make the classification task difficult for humans, the CIFAR10 images are downsampled by 4×. This forces at least partial reliance on model predictions, allowing us to evaluate which explanations are convincing enough to earn the user's agreement.

each explainability technique, we collected **600** survey responses. When given saliency maps and class probabilities, only **87** predictions were correctly identified as wrong. In comparison, when given the NBDT series of predicted classes and child probabilities (e.g., "Animal (90%) → Mammal (95%)", without the final leaf prediction) **237** images were correctly identified as wrong. Thus, respondents can better recognize mistakes in NBDT explanations nearly 3 times better.

Although NBDT provides more information than saliency maps about misclassification, a majority – the remaining 363 NBDT predictions – were not correctly identified. To explain this, we note that $\sim 37\%$ of all NBDT errors occur at the final binary decision, between two leaves; since we provide all decisions except the final one, these leaf errors would be impossible to distinguish.

Survey: Explanation-Guided Image Classification

In this section we aim to answer a question posed in [84] *"To what extent do people follow a model's predictions when it is beneficial to do so?"*. In this first survey, each user is asked to classify a severely blurred image (Fig 5.4). This survey affirms the problem's difficulty, decimating human performance to not much more than guessing: **163** of **600** responses are correct (27.2% accuracy).

In the next survey, we offer the blurred image and two sets of predictions: (1) the original neural network's predicted class and its saliency map, and (2) the NBDT predicted class and the sequence of decisions that led up to it ("Animal, Mammal, Cat"). For all examples, the two models predict different classes. In 30% of the examples, NBDT is right and the original model is wrong. In another 30%, the opposite is true. In the last 40%, both models are wrong. As shown in Fig. 5.4, the image is extremely blurry, so the user must rely on the models to inform their prediction. When offered model predictions, in this survey, **255** of **600** responses are correct (42.5% accuracy), a 15.3 point improvement over no model guidance. We observe that humans trust NBDT-explained prediction more often than the saliency-explained predictions. Out of **600** responses, **312** responses agreed with the NBDT's prediction, **167** responses agreed with the base model's prediction, and **119** responses disagreed with both model's predictions. Note that a majority of user decisions ($\sim 80\%$) agreed with either model prediction, even though neither model prediction was correct in 40% of examples, showing

our images were sufficiently blurred to force reliance on the models. Furthermore, 52% of responses agreed with NBDT (against saliency's 28%), even though only 30% of NBDT predictions were correct, showing improvement in model trust.

Survey: Human-Diagnosed Level of Trust

The explanation of an NBDT prediction is the visualization of the path traversed. We then compare these NBDT explanations to other explainability methods in human studies. Specifically, we ask participants to pick an expert to trust (Appendix, Figure 5.13), based on the expert's explanation – a saliency map (ResNet18, GradCAM), a decision tree (NBDT), or neither. We only use samples where ResNet18 and NBDT predictions agree. Of 374 respondents that picked one method over the other, **65.9%** prefer NBDT explanations; for misclassified samples, **73.5%** prefer NBDT. This supports the previous survey's results, showing humans trust NBDTs more than current saliency techniques when explicitly asked.

Analysis: Identifying Faulty Dataset Labels

There are several types of ambiguous labels (Figure 5.5), any of which could hurt model performance for an image classification dataset like ImageNet. To find these images, we use entropy in NBDT decisions, which we find is a much stronger indicator of ambiguity than entropy in the original neural network prediction. The intuition is as follows: If all intermediate decisions have high certainty except for a few decisions, those decisions are deciding between multiple equally plausible cases. Using this intuition, we can identify ambiguous labels by finding samples with high "path entropy" – or highly disparate entropies for intermediate decisions on the NBDT prediction path.

Per Figure 5.6, the highest "path entropy" samples in ImageNet contain multiple objects, where each object could plausibly be used for the image class. In contrast, samples that induce the highest entropy in the baseline neural network do not suggest ambiguous labels. This suggests NBDT entropy is more informative compared to that of a standard neural network.

5.6 Applications

In this section, we expand on details for interpretability as presented above, with an emphasis on qualitative use of the hierarchy.

Maximum Similarity Examples to Visualize Generalization

We (1) visually confirm the hypothesized meaning of each node by identifying the most "representative" samples, and (2) check that these "representative" samples represent that

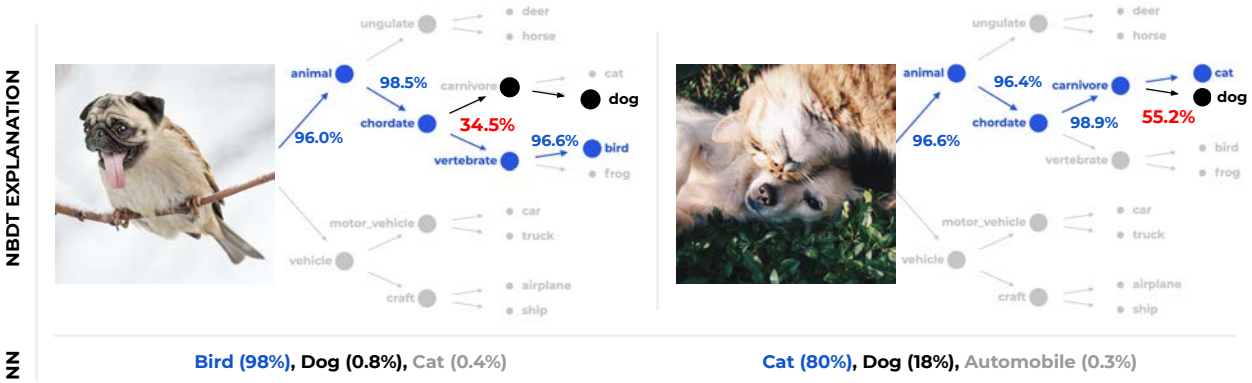


Figure 5.5: Types of Ambiguous Labels. All these examples have ambiguous labels. With NBDT (top), the decision rule deciding between equally-plausible classes has low certainty (red, 30-50%). All other decision rules have high certainty (blue, 96%+). The juxtaposition of high and low certainty decision rules makes ambiguous labels easy to distinguish. By contrast, ResNet18 (bottom) still picks one class with high probability. (Left) An extreme example of a "spug" that may plausibly belong to two classes. (Right) Image containing two animals of different classes. Photo ownership: "Spug" by Arne Fredriksen at gyyporama.com. Used with permission. Second image is CC-0 licensed at pexels.com.

category (e.g., *Animal*) and not just the training classes under that category. We define "representative" samples, or maximum similarity examples, to be samples with embeddings most similar to an inner node's representative. We visualize these examples for a model before and after the tree supervision loss (NBDT and ResNet18, respectively). The models are trained on CIFAR10, but samples are drawn from ImageNet. We observe that maximum similarity examples for NBDT contain more unseen classes than ResNet18 (Figure 5.8). This suggests that our NBDT is better able to capture high-level concepts such as *Animal*, which is quantitatively confirmed by the superclass evaluation in Table 5.6.

Explainability of Nodes' Visual Meanings

This section describes the method used in Table 5.6 in more detail. Since the induced hierarchy is constructed using model weights, the intermediate nodes are not forced to split on foreground objects. While hierarchies like WordNet provide hypotheses for a node's meaning, the tree may split on unexpected contextual and visual attributes such as *underwater* and *on land*, depicted in Figure 5.7b. To diagnose a node's visual meaning, we perform the following 4-step test:

1. Posit a hypothesis for the node's meaning (e.g. *Animal vs. Vehicle*). This hypothesis can be computed automatically from a given taxonomy or deduced from manual inspection of each child's leaves (Figure 5.9).

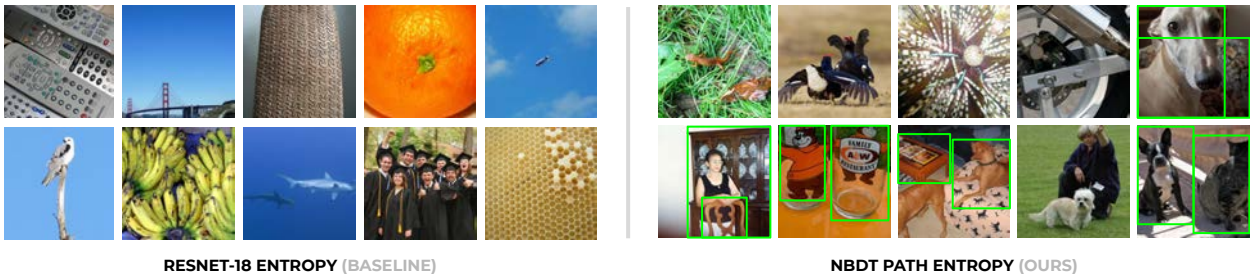
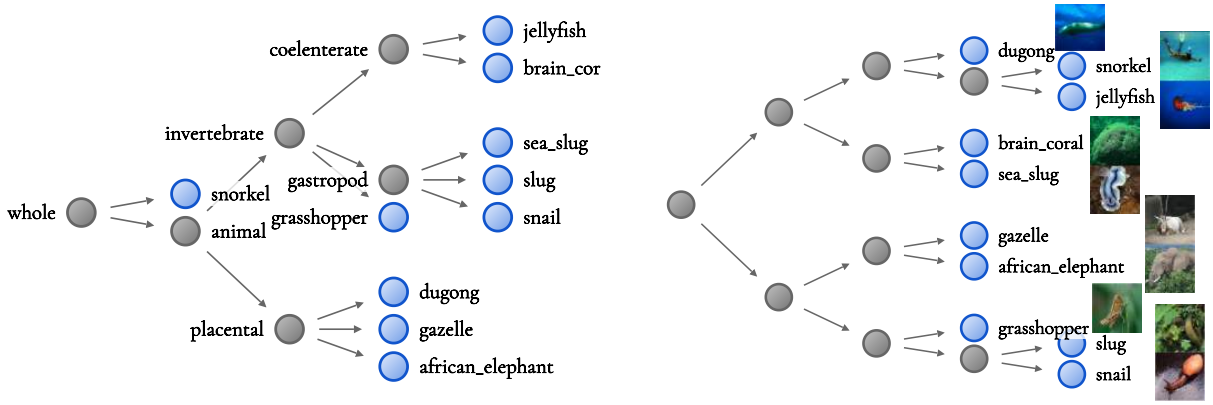


Figure 5.6: ImageNet Ambiguous Labels. These images suggest that NBDT path entropy uniquely identifies ambiguous labels in Imagenet, without object detection labels. We plot ImageNet validation samples that induce the most 2-class confusion, using TinyImagenet200-trained models. Note that ImageNet classes do not include people. (Left) Run ResNet18 and find samples that (a) maximize entropy between the top 2 classes and (b) minimize entropy across all classes, where the top 2 classes are averaged. Despite high model uncertainty, half the classes are from the training set – bee, orange, bridge, banana, remote control – and do not show visual ambiguity. (Right) For NBDT, compute entropy for each node's predicted distribution; take the difference between the largest and smallest values. Now, half of the images contain truly ambiguous content for a classifier; we draw green boxes around pairs of objects that could each plausibly be used for the image class.



(a) WordNet Hierarchy **(b) Induced Hierarchy**

2. Collect a dataset with new, unseen classes that test the hypothesised meaning from step 1 (e.g. *Elephant* is an unseen *Animal*). Samples in this dataset are referred to as out-of-distribution (OOD) samples, as they are drawn from a separate labeled dataset.
3. Pass samples from this dataset through the node. For each sample, check whether the selected child node agrees with the hypothesis.
4. The accuracy of the hypothesis is the percentage of samples passed to the correct child. If the accuracy is low, repeat with a different hypothesis.

Figure 5.9a depicts the CIFAR10 tree induced by a WideResNet28x10 model trained on

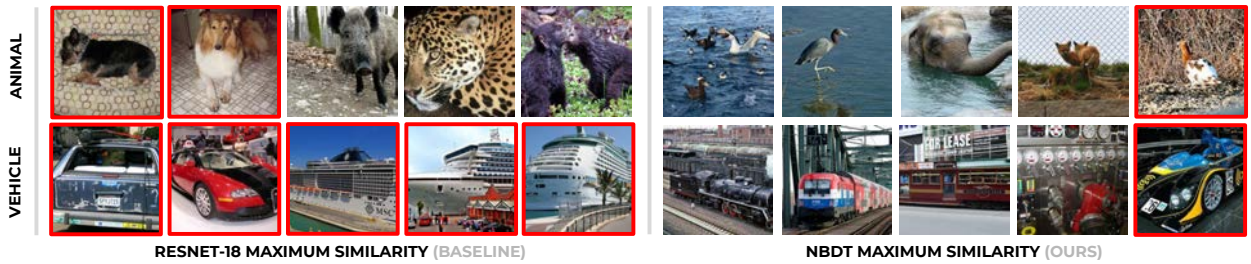


Figure 5.8: Maximum Similarity Examples. We run two CIFAR10-trained models, one trained with tree supervision loss (NBDT) and one without tree supervision loss (ResNet18). We compute the induced hierarchy of both models and find samples most similar to the *Animal*, and *Motor Vehicle* concepts. Each row represents an inner node, and the red borders indicate images that contain CIFAR10 classes. (1) Note that NBDT's concept of an animal includes classes and contexts it was not trained on; aquatic animals (top-right) and trains (bottom-right) are not a part of CIFAR10. In contrast, ResNet18 largely finds examples closely related to existing CIFAR10 classes (dog, car, boat). This is qualitative evidence that NBDTs better generalize.

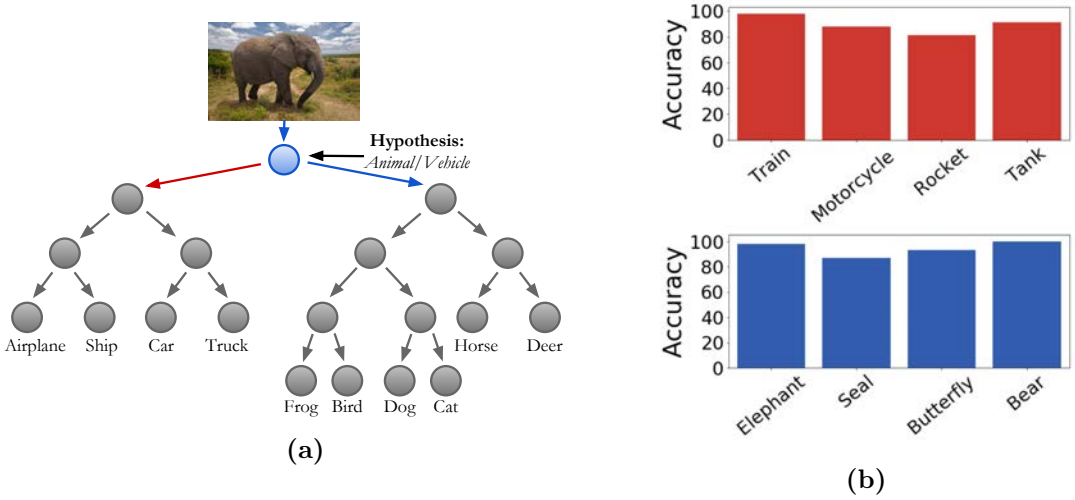
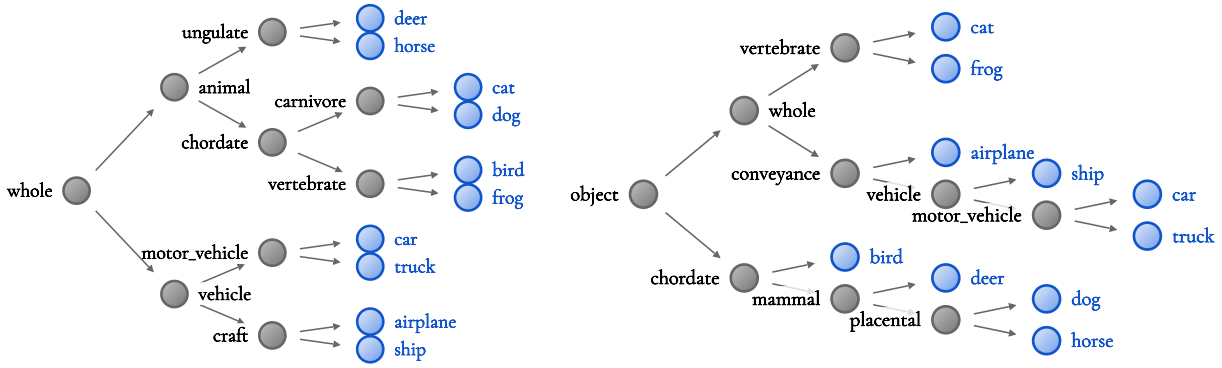


Figure 5.9: A Node's meaning. (Left) Visualization of node hypothesis test performed on a CIFAR10-trained WideResNet28x10 model, by sampling from CIFAR100 validation set for OOD classes. **(Right)** Classification accuracy is high (80-95%) given unseen CIFAR100 samples of *Vehicles* (top) and *Animals* (bottom), for the WordNet-hypothesized *Animal/Vehicle* node.

CIFAR10. The WordNet hypothesis is that the root note splits on *Animal vs. Vehicle*. We use the CIFAR100 validation set as out-of-distribution images for *Animal* and *Vehicle* classes that are unseen at training time. We then compute the hypothesis' accuracy. Figure 5.9b shows our hypothesis accurately predicts which child each unseen-class's samples traverse.



(a) WideResNet28x10 **(b) ResNet10**

Figure 5.10: CIFAR10 induced hierarchies, with automatically-generated WordNet hypotheses for each node. The higher-accuracy (a) WideResNet (97.62% acc) has a more sensible hierarchy than (b) ResNet's (93.64% acc): The former groups all *Animal*s together, separate from all *Vehicles*. By contrast, the latter groups *Airplane*, *Cat*, and *Frog*.

How Model Accuracy Affects Interpretability

Induced hierarchies are determined by the proximity of class weights, but classes that are close in weight space may not have similar visual meaning: Figure 5.10 depicts the trees induced by WideResNet28x10 and ResNet10, respectively. While the WideResNet induced hierarchy (Figure 5.10a) groups visually-similar classes, the ResNet (Figure 5.10b) induced hierarchy does not, grouping classes such as *Frog*, *Cat*, and *Airplane*. This disparity in visual meaning is explained by WideResNet's 4% higher accuracy: we believe that higher-accuracy models exhibit more visually-sound weight spaces. Thus, unlike previous work, NBDTs feature better interpretability with higher accuracy, instead of sacrificing one for the other. Furthermore, the disparity in hierarchies indicates that a model with low accuracy will not provide interpretable insight into high-accuracy decisions.

Visualization of Tree Traversal

Frequency of path traversals additionally provide insight into general model behavior. Figure 5.11 shows frequency of path traversals for all samples in three classes: a seen class, an unseen class but with seen context, and an unseen class with unseen context.

Seen class, seen context: We visualize tree traversals for all samples in CIFAR10's *Horse* class (Figure 5.11a). As this class is present during training, tree traversal highlights the correct path with extremely high frequency. **Unseen class, seen context**: In Figure 5.11b, we visualize tree traversals for TinyImagenet's *Seashore* class. The model classifies 88% of *Seashore* samples as "vehicle with blue context," exhibiting reliance on context for decision-making. **Unseen class, unseen context**: In Figure 5.11c, we visualize traversals for TinyImagenet's *Teddy Bear*. The model classifies 90% as *Animal*, belying the model's

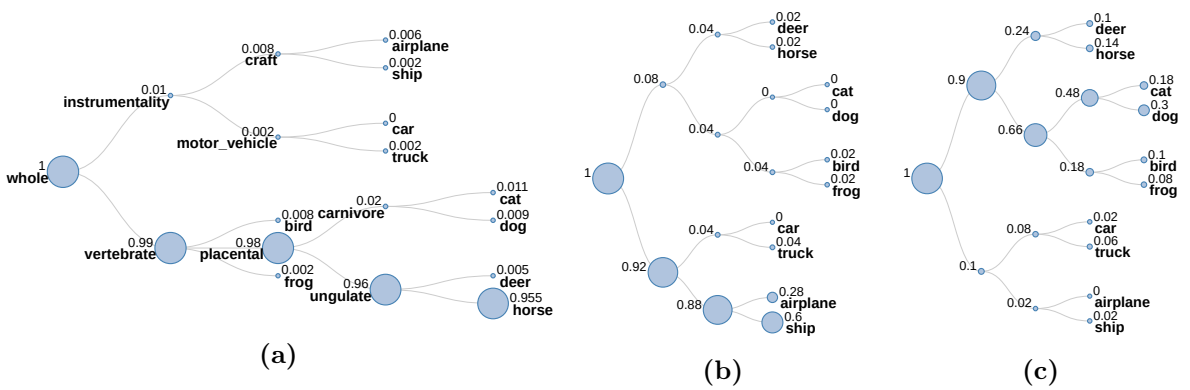


Figure 5.11: Visualization of path traversal frequency on an induced hierarchy for CIFAR10. (a) **In-Distribution: *Horse*** is a training class and thus sees highly focused path traversals. (b) **Unseen Class: *Seashore*** is largely classified as *Ship* despite not containing any objects, exhibiting model reliance on context (water). (c) **Unseen Class: *Teddy Bear*** is classified as *Dog*, for sharing visual attributes like color and texture.

generalization to stuffed animals. However, the model disperses samples among animals more evenly, with the most furry animal *Dog* receiving the most *Teddy Bear* samples (30%).

5.7 Ablations

In the context of neural network and decision tree hybrids, many works [95, 52, 126, 107] leverage conditional execution to improve computational efficiency in a hierarchical classifier. One motivation is to handle large-scale classification problems.

Hard Tree Supervision Loss

An alternative loss would be hierarchical softmax – in other words, one cross entropy loss per decision rule. We denote this the *hard tree supervision loss*, as we construct a variant of hierarchical softmax that (a) supports arbitrary depth trees and (b) is defined over a single, un-augmented fully-connected layer (e.g. k-dimensional output for a k-leaf tree). The original neural network's loss $\mathcal{L}_{\text{original}}$ minimizes cross entropy across the classes. For a k-class dataset, this is a k-way cross entropy loss. Each internal node's goal is similar: minimize cross-entropy loss across the child nodes. For node i with c children, this is a c-way cross entropy loss between predicted probabilities $\mathcal{D}(i)_{\text{pred}}$ and labels $\mathcal{D}(i)_{\text{label}}$. We refer to this collection of new loss terms as the *hard tree supervision loss* (Eq. 5.4). The individual cross entropy losses for each node are scaled so that the original cross entropy loss and the tree supervision loss are weighted equally, by default. If we assume N nodes in the tree, excluding leaves, then we would have $N + 1$ different cross entropy loss terms – the original cross entropy loss and N hard tree supervision loss terms. This is $\mathcal{L}_{\text{original}} + \mathcal{L}_{\text{hard}}$, where:

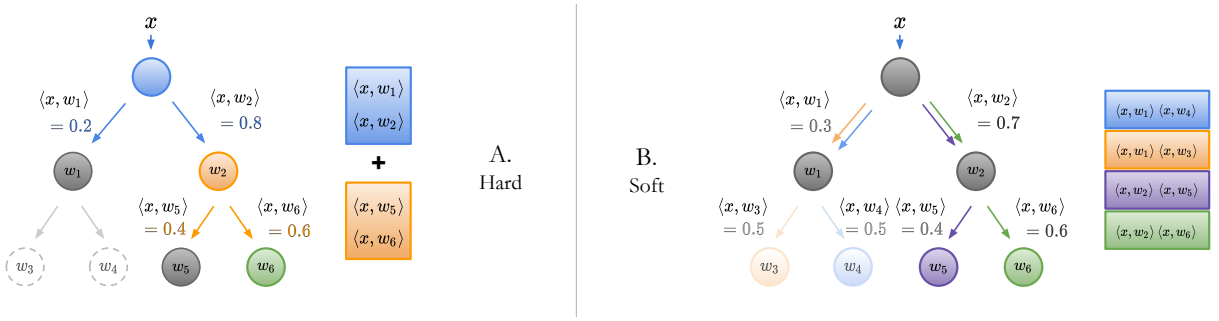


Figure 5.12: Tree Supervision Loss has two variants: **Hard Tree Supervision Loss (A)** defines a cross entropy term per node. This is illustrated with the blue box for the blue node and the orange box for the orange node. The cross entropy is taken over the child node probabilities. The green node is the leaf representing a class label. The dotted nodes are not included in the path from the label to the root, so do not have a defined loss. **Soft Tree Supervision Loss (B)** defines a cross entropy loss over all leaf probabilities. The probability of the green leaf is the product of the probabilities leading up to the root (in this case, $\langle x, w_2 \rangle \langle x, w_6 \rangle = 0.6 \times 0.7$). The probabilities for the other leaves are similarly defined. Each leaf probability is represented with a colored box. The cross entropy is then computed over this leaf probability distribution, represented by the colored box stacked on one another.

$$\mathcal{L}_{\text{hard}} = \frac{1}{N} \sum_{i=1}^{N} \underbrace{\textsc{CrossEntropy}(\mathcal{D}(i)_{\text{pred}}, \mathcal{D}(i)_{\text{label}})}_{\text{over the } c \text{ children for each node}}. \tag{5.4}$$

Hard Inference

Hard inference is more intuitive: Starting at the root node, each sample is sent to the child with the most similar representative. We continue picking and traversing the tree until we reach a leaf. The class associated with this leaf is our prediction (Figure 5.1, A. Hard). More precisely, consider a tree with nodes indexed by i with set of child nodes $C(i)$. Each node i produces a probability of child node $j \in C(i)$; this probability is denoted $p(j|i)$. Each node thus picks the next node using $\text{argmax}_{j \in C(i)} p(j|i)$.

Whereas this inference mode is more intuitive, it underperforms soft inference (Figure 5.7). Furthermore, note that hard tree supervision loss (*i.e.* modified hierarchical softmax) appears to more specifically optimize hard inference. Despite that, hard inference performs worse (Figure 5.8) with hard tree supervision loss than the "soft" tree supervision loss (Sec 5.3) used in previous sections.

Table 5.7: Comparisons of Inference Modes Hard inference performs worse than soft inference. See Table 5.1 in the main manuscript for a comparison against baselines.

Method	Backbone	CIFAR10	CIFAR100	TinyImageNet
NN	WideResNet28x10	*97.62%*	*82.09%*	*67.65%*
NBDT-H (Ours)	WideResNet28x10	97.55%	82.21%	64.39%
NBDT-S (Ours)	WideResNet28x10	**97.55%**	**82.97%**	**67.72%**
NN	ResNet18	*94.97%*	*75.92%*	*64.13%*
NBDT-H (Ours)	ResNet18	94.50%	74.29%	61.60%
NBDT-S (Ours)	ResNet18	**94.82%**	**77.09%**	**63.77%**

Table 5.8: Tree Supervision Loss Training the NBDT with the tree supervision loss ("TSL") is superior to (a) training with a hierarchical softmax ("HS") and to (b) omitting extra loss terms. ("None"). Δ is the accuracy difference between our soft loss and hierarchical softmax.

Dataset	Backbone	NN	Inference	None	TSL	HS	Δ
CIFAR10	ResNet18	94.97%	Hard	94.32%	94.50%	93.94%	+0.56%
CIFAR10	ResNet18	94.97%	Soft	94.38%	94.82%	93.97%	+0.85%
CIFAR100	ResNet18	75.92%	Hard	57.63%	74.29%	73.23%	+0.94%
CIFAR100	ResNet18	75.92%	Soft	61.93%	77.09%	74.09%	+1.83%
TinyImageNet	ResNet18	64.13%	Hard	39.57%	61.60%	58.89%	+2.71%
TinyImageNet	ResNet18	64.13%	Soft	45.51%	63.77%	61.12%	+2.65%

5.8 Implementation

Our inference strategy, as outlined above, includes two phases: (1) featurizing the sample using the neural network backbone and (2) running the embedded decision rules. However, in practice, our inference implementation does not need to run inference with the backbone, separately. In fact, our inference implementation only requires the logits $\hat{y}$ outputted by the network. This is motivated by the knowledge that the average of inner products is equivalent to the inner product of averages. Knowing this, we have the following equivalence, given the fully-connected layer weight matrix W, its row vectors w_i, featurized sample x, and the classes C we are currently interested in.

$$\langle x, \frac{1}{n}\sum_{i=1}^{|C|} w_i \rangle = \frac{1}{n}\sum_{i=1}^{|C|} \langle x, w_i \rangle = \frac{1}{n}\sum_{i=1}^{|C|} \hat{y}_i, i \in C \tag{5.5}$$

Thus, our inference implementation is simply performed using the logits $\hat{y}$ output by the network.

Experimental Setup

To reiterate, our best-performing models for both hard and soft inference were obtained by training with the soft tree supervision loss. All CIFAR10 and CIFAR100 experiments weight the soft loss terms by 1. All TinyImagenet and Imagenet experiments weight the soft loss terms by 10. We found that hard loss performed best when the hard loss weight was $10\times$ that of the corresponding soft loss weight (e.g. weight 10 for CIFAR10, CIFAR100; and weight 100 for TinyImagenet, Imagenet); these hyper-parameters are use for the tree supervision loss comparisons in Table 5.3.

Where possible, we retrain the network from scratch with tree supervision loss. For our remaining training hyperparameters, we largely use default settings found in `github.com/kuangliu/pytorch-cifar`: SGD with 0.9 momentum, 5^{-4} weight decay, a starting learning rate of 0.1, decaying by 90% $\frac{3}{7}$ and $\frac{5}{7}$ of the way through training. We make a few modifications: Training lasts for 200 epochs instead of 350, and we use batch sizes of 512 and 128 on one Titan Xp for CIFAR and TinyImagenet respectively.

In cases where we were unable to reproduce the baseline accuracy (WideResNet), we fine-tuned a pretrained checkpoint with the same settings as above, except with starting learning rate of 0.01.

On Imagenet, we retrain the network from scratch with tree supervision loss. For our remaining hyperparameters, we use settings reported to reproduce EfficientNet-EdgeTPU-Small results at
`github.com/rwightman/pytorch-image-models`: batch size 128, RMSProp with starting learning rate of 0.064, decaying learning rate by 97% every 2.4 epochs, weight decay of 10^{-5}, drop-connect with probability 0.2 on 8 V100s. Our results were obtained with only one model, as opposed to averaging over 8 models, so our reported baseline is 77.23%, as reported by the EfficientNet authors: `https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet/edgetpu#post-training-quantization`.

CIFAR100 Tree Visualization

We presented the tree visualizations for various models on the CIFAR10 dataset in above sections. Here we also show that similar visual meanings can be drawn from intermediate nodes of larger trees such as the one for CIFAR100. Figure 5.14 displays the tree visualization for a WideResNet28x10 architecture on CIFAR100 (same model listed in Table 1 of Sec. 4.2). It can be seen in Figure 5.14 that subtrees can be grouped by visual meaning, which can be

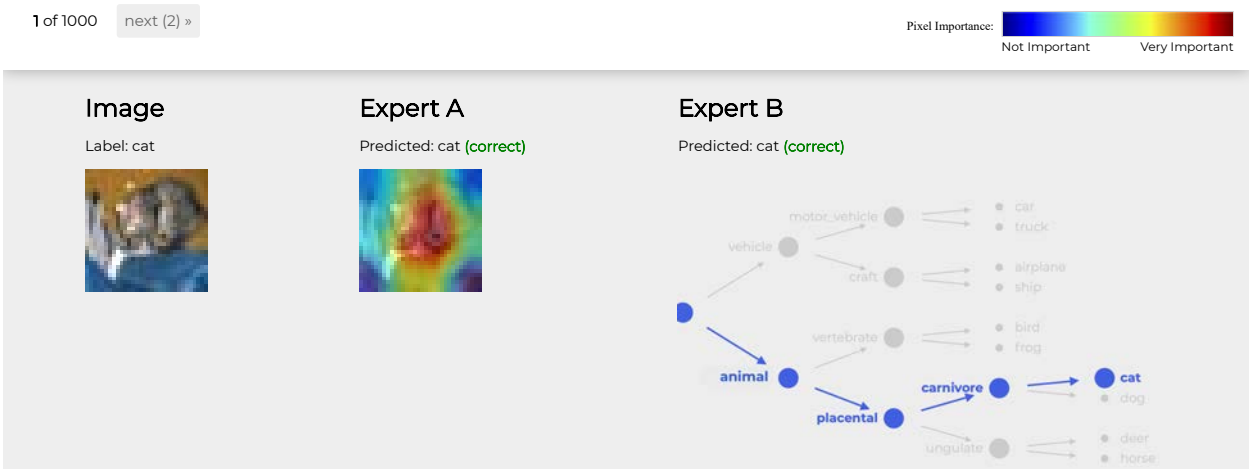
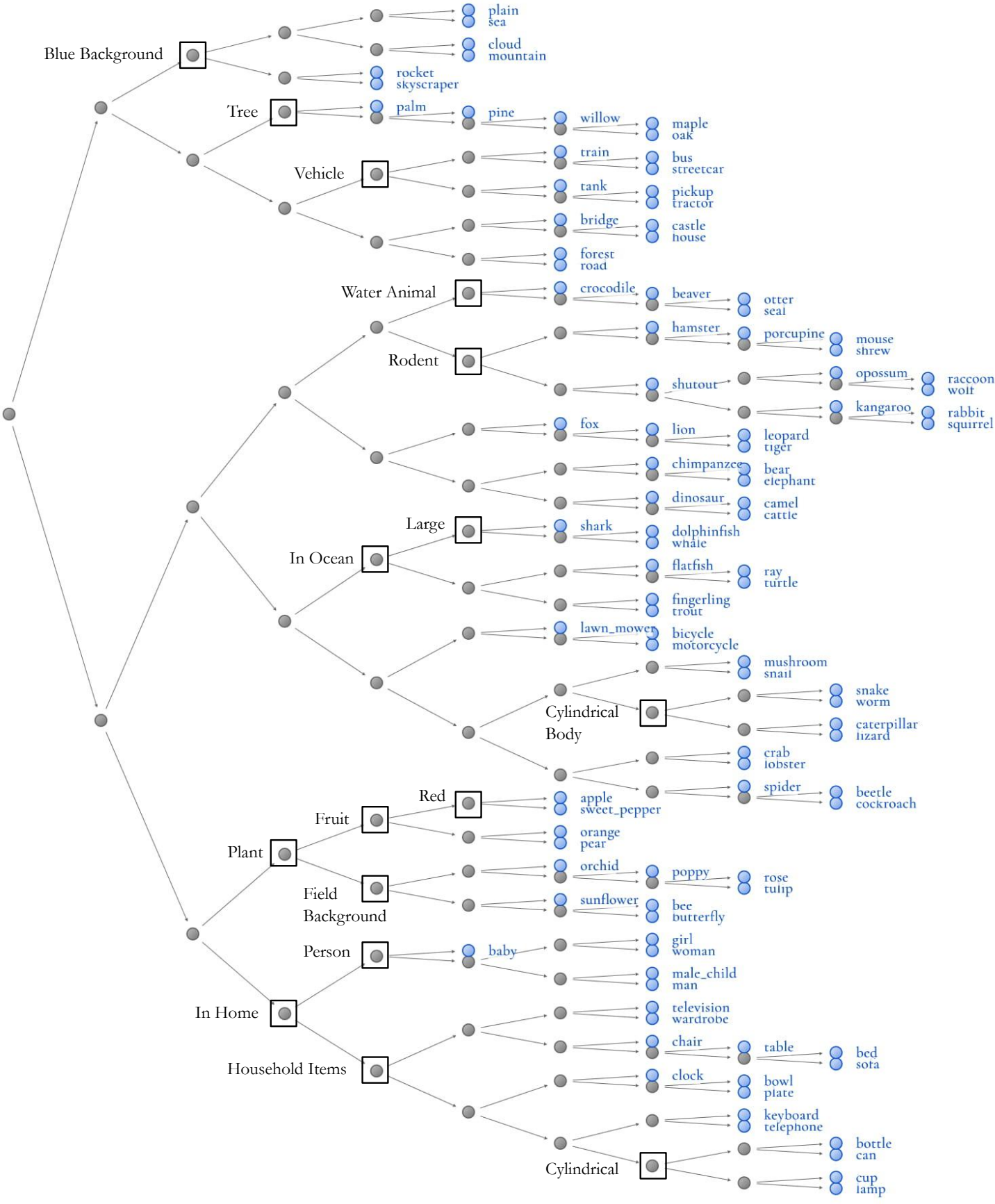


Figure 5.13: An example of a survey question presented to mechanical turks.

a Wordnet attribute like *Vehicle* or *Household Item*, or a more contextual meaning such as shape or background like *Cylindrical* or *Blue Background*.

5.9 Conclusion

In this work, we propose Neural-Backed Decision Trees that see (1) improved accuracy: NBDTs out-generalize (16%+), improve (2%+), and match (0.15%) or outperform (1%+) state-of-the-art neural networks on CIFAR10, CIFAR100, TinyImageNet, and ImageNet. We also show (2) improved interpretability by drawing unique insights from our hierarchy, confirming that humans trust NBDT's over saliency and illustrate how path entropy can be used to identify ambiguous labels. This challenges the conventional supposition of a dichotomy between accuracy and interpretability, paving the way for jointly accurate *and* interpretable models in real-world deployments.


Blue Background
plain
sea
cloud
mountain
rocket
skyscraper
Tree
palm
pine
willow
maple
oak
Vehicle
train
bus
streetcar
tank
pickup
tractor
bridge
castle
house
forest
road
Water Animal
crocodile
beaver
otter
seal
Rodent
hamster
porcupine
mouse
shrew
shutout
opossum
raccoon
wolf
kangaroo
rabbit
squirrel
fox
lion
leopard
tiger
chimpanzee
bear
elephant
dinosaur
camel
cattle
Large
shark
dolphinfish
whale
In Ocean
flatfish
ray
turtle
fingerling
trout
lawn_mower
bicycle
motorcycle
mushroom
snail
Cylindrical
Body
snake
worm
caterpillar
lizard
crab
lobster
spider
beetle
cockroach
Red
apple
sweet_pepper
Fruit
orange
pear
Plant
orchid
poppy
rose
tulip
Field
Background
sunflower
bee
butterfly
Person
baby
girl
woman
male_child
man
In Home
television
wardrobe
chair
table
bed
sofa
Household Items
clock
bowl
plate
keyboard
telephone
Cylindrical
bottle
can
cup
lamp

Chapter 6

Conclusion

6.1 Review

Efficient deep neural networks underwent three key steps: sacrifice accuracy in existing neural networks for efficiency, manually tradeoff accuracy and efficiency by designing neural networks from scratch, and finally, automatically trade off accuracy and efficiency using Neural Architecture Search. However, this last stage resulted in computationally expensive neural architecture search methods that would expend GPU years of compute to design a single neural network. In response, this thesis proposes efficient design for efficient neural networks. In particular, we summarize the major contributions of the thesis below:

1. This thesis culminates in a set of models that set new flexibility and performance standards for production-ready models: those that are state-of-the-art accurate, explainable, generalizable, and configurable for any set of resource constraints in just CPU minutes.
2. **Expanded Search Space for Differentiable Neural Architecture Search (DNAS)** A memory and computationally efficient DNAS that optimizes both macro- (resolution, channels) and micro- (building blocks) architectures jointly in a 1014× larger search space using differentiable search. To the best of our knowledge, we are the first to tackle this problem using a differentiable search framework supergraph, with substantially less computational cost and roughly constant memory cost.
3. **DNAS Masking Mechanism** A masking mechanism and effective shape propagation for feature map reuse. This is applied to both the spatial and channel dimensions in DNAS.
4. **FBNetV2 State-of-the-art ImageNet Accuracy**. With only 27 hours on 8 GPUs, our searched compact models lead to substantial per-parameter, per-FLOP accuracy

improvements. The searched models outperform all previous state-of-the-art neural networks, both manually and automatically designed, small and large.

5. **Neural Architecture-Recipe Search**: We propose a predictor that jointly scores both training recipes and architectures, the first joint search, over both training recipes and architectures, at scale to our knowledge.

6. **Predictor Pretraining** To enable efficient search over this larger space, we furthermore present a pretraining technique, significantly improving the accuracy predictor's sample efficiency.

7. **Multi-Use Predictor**: Our predictor can be used in fast evolutionary searches to quickly generate models for a wide variety of resource budgets in just CPU minutes.

8. **FBNetv3 State-of-the-art ImageNet Accuracy** per FLOP for the searched FBNetV3 models. For example, our FBNetV3 matches EfficientNet accuracy with as low as 49.3% fewer FLOPs.

9. **Neural-Backed Decision Trees**: We propose a tree supervision loss, yielding NBDTs that match/outperform and out- generalize modern neural networks (WideResNet, EfficientNet) on ImageNet, Tiny- ImageNet200, and CIFAR100. Our loss also improves the original model by up to 2%.

10. **Induced Hierarchies**: We propose alternative hierarchies for oblique decision trees – *induced hierarchies* built using pre-trained neural network weights – that outperform both data-based hierarchies (e.g. built with information gain) and existing hierarchies (e.g. WordNet), in accuracy.

11. **Improved Interpretability**: We show NBDT explanations are more helpful to the user when identifying model mistakes, preferred when using the model to assist in challenging classification tasks, and can be used to identify ambiguous ImageNet labels.

6.2 Impact

This thesis compiles my research in efficient deep learning over the past few years [112, 111, 20] which, despite its recent publications, has begun to make an impact in the efficient deep learning community:

1. **FBNetV2** [111] (Chapter 3) was published in CVPR 2020 and has amassed 141 citations since. We furthermore open-sourced the pre-trained models, with the repository receiving significant amounts of attention (706 stars).

2. **FBNetV3** [20] (Chapter 4) was published in CVPR 2021 and has garnered 52 citations since.

3. **NBDT** [112] (Chapter 5) was published in ICLR 2021 and has garnered 60 citations since, with a similarly popular open-source repository (521 stars) that includes all code for training, evaluating, predicting, and even deploying the model.

6.3 Future Work

The work described in this thesis shows that joint consideration of even more objectives and of larger search sizes is, surprisingly, the way to automatically design efficient neural networks, more efficiently. This suggests the following directions for future work:

Efficient 3D Deep Learning: Techniques for efficient deep neural networks today are all task-agnostic, despite the fact that most papers focus on image classification as a testbed. Furthermore, accuracy on ImageNet is teetering close to the widely accepted label accuracy, meaning any further accuracy improvements suggest overfitting. These two concerns together – that efficiency techniques are task-agnostic and that the ImageNet classification benchmark is possibly saturated – suggests shifting focus to other computer vision tasks, from popular tasks like object detection and segmentation to depth estimation. However, are generic efficiency techniques the best we can do, for these downstream tasks?

We believe otherwise; in particular, we believe that there is unrealized potential in redesigning portions of the task–or even the task entirely–to repurpose downstream tasks for deployment to edge devices. We furthermore believe that without redesigning the task, existing efficiency techniques will not suffice. One example is 3D instance segmentation. If we naively adapt 2D methods to a 3D voxel grid, the majority of efficiency techniques will be only counteracting the computational complexity incurred by the naive adaptation. Instead, 3D instance segmentation may take an entirely different approach, separate from its 2D counterpart, to be edge-ready.

Test-Time Training in Lieu of Efficiency For depth estimation in particular, a number of test-time deep learning training methods have proven extremely successful, inspired by classical structure-from-motion and SLAM pipelines. Unsupervised test-time training has likewise shown promise in areas like object detection, suggesting that test-time training may offer another boon for model performance, thus enabling even smaller, more compact deep neural networks. There are a large number of questions: How do we partition the sparse on-device resources between test-time training and inference? How much performance can test-time training "make up" for in a poorly-trained model? How would we trade off test-time training with standard model training?

Bibliography

[1] Karim Ahmed, Mohammadharis Baig, and Lorenzo Torresani. "Network of Experts for Large-Scale Image Categorization". In: vol. 9911. Apr. 2016.

[2] Stephan Alaniz and Zeynep Akata. "XOC: Explainable Observer-Classifier for Explainable Binary Decisions". In: *CoRR* abs/1902.01780 (2019).

[3] Seungryul Baek, Kwang In Kim, and Tae-Kyun Kim. "Deep Convolutional Decision Jungle for Image Classification". In: *CoRR* abs/1706.02003 (2017).

[4] Bowen Baker et al. "Accelerating neural architecture search using performance prediction". In: *arXiv preprint arXiv:1705.10823* (2017).

[5] Arunava Banerjee. "Initializing Neural Networks using Decision Trees". In: *Proceedings of the International Workshop on Computational Learning and Natural Learning Systems*. MIT Press, 1994, pp. 3–15.

[6] Arunava Banerjee. "Initializing neural networks using decision trees". In: (1990).

[7] James Bergstra, Daniel Yamins, and David Daniel Cox. "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures". In: (2013).

[8] Ufuk Can Biçici, Cem Keskin, and Lale Akarun. "Conditional information gain networks". In: *2018 24th International Conference on Pattern Recognition (ICPR)*. IEEE. 2018, pp. 1390–1395.

[9] Olcay Boz. "Converting A Trained Neural Network To a Decision Tree DecText - Decision Tree Extractor". In: *ICMLA*. 2000.

[10] Andrew Brock, Jeff Donahue, and Karen Simonyan. "Large scale gan training for high fidelity natural image synthesis". In: *ICLR* (2019).

[11] Clemens-Alexander Brust and Joachim Denzler. "Integrating domain knowledge: using hierarchies to improve deep classifiers". In: *Asian Conference on Pattern Recognition*. Springer. 2019, pp. 3–16.

[12] Han Cai, Ligeng Zhu, and Song Han. "Proxylessnas: Direct neural architecture search on target task and hardware". In: *arXiv preprint arXiv:1812.00332* (2018).

[13] Han Cai et al. "Once for All: Train One Network and Specialize it for Efficient Deployment". In: *ICLR*. 2020.

[14] Diogo V Carvalho, Eduardo M Pereira, and Jaime S Cardoso. "Machine learning interpretability: A survey on methods and metrics". In: *Electronics* 8.8 (2019), p. 832.

[15] Mark Craven and Jude W Shavlik. "Extracting tree-structured representations of trained networks". In: *Advances in neural information processing systems*. 1996, pp. 24–30.

[16] Mark W Craven and Jude W Shavlik. "Using sampling and queries to extract rules from trained neural networks". In: *Machine learning proceedings 1994*. Elsevier, 1994, pp. 37–45.

[17] Ekin D Cubuk et al. "Autoaugment: Learning augmentation strategies from data". In: *CVPR*. 2019.

[18] Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. "NeST: A neural network synthesis tool based on a grow-and-prune paradigm". In: *arXiv preprint arXiv:1711.02017* (2017).

[19] Xiaoliang Dai et al. "Chamnet: Towards efficient network design through platform-aware model adaptation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 11398–11407.

[20] Xiaoliang Dai et al. "Fbnetv3: Joint architecture-recipe search using predictor pretraining". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 16276–16285.

[21] Darren Dancey, David McLean, and Zuhair Bandar. "Decision tree extraction from trained neural networks". In: (Jan. 2004).

[22] *Deep Learning Performance Guide*. `https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html`.

[23] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[24] Jia Deng et al. "Hedging your bets: Optimizing accuracy-specificity trade-offs in large scale visual recognition". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2012, pp. 3450–3457.

[25] Jia Deng et al. "Large-Scale Object Classification using Label Relation Graphs". In: ().

[26] Piotr Dollár, Mannat Singh, and Ross Girshick. "Fast and Accurate Model Scaling". In: *arXiv preprint arXiv:2103.06877* (2021).

[27] Finale Doshi-Velez and Been Kim. "Towards a rigorous science of interpretable machine learning". In: *arXiv preprint arXiv:1702.08608* (2017).

[28] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: `http://archive.ics.uci.edu/ml`.

[29] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural architecture search: A survey". In: *arXiv preprint arXiv:1808.05377* (2018).

[30] Nicholas Frosst and Geoffrey E. Hinton. "Distilling a Neural Network Into a Soft Decision Tree". In: *CoRR* abs/1711.09784 (2017).

[31] Yanming Guo et al. "CNN-RNN: a large-scale hierarchical image classification framework". In: *Multimedia Tools and Applications* 77.8 (2018), pp. 10251–10271.

[32] Zichao Guo et al. "Single path one-shot neural architecture search with uniform sampling". In: *arXiv preprint arXiv:1904.00420* (2019).

[33] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[34] Song Han et al. "Learning both weights and connections for efficient neural network". In: *Proc. Advances in Neural Information Processing Systems*. 2015, pp. 1135–1143.

[35] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[36] Kaiming He et al. "Momentum contrast for unsupervised visual representation learning". In: *CVPR*. 2020.

[37] Yang He et al. "Soft filter pruning for accelerating deep convolutional neural networks". In: *arXiv preprint arXiv:1808.06866* (2018).

[38] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network". In: *arXiv preprint arXiv:1503.02531* (2015).

[39] Daniel Ho et al. "Population based augmentation: Efficient learning of augmentation policy schedules". In: *ICML* (2019).

[40] Andrew Howard et al. "Searching for mobilenetv3". In: *arXiv preprint arXiv:1905.02244* (2019).

[41] Andrew G Howard et al. "MobileNets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[42] Jie Hu, Li Shen, and Gang Sun. "Squeeze-and-excitation networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 7132–7141.

[43] Gao Huang et al. "Deep networks with stochastic depth". In: *ECCV*. 2016.

[44] Gao Huang et al. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[45] Itay Hubara et al. "Binarized neural networks". In: *Proc. Advances in Neural Information Processing Systems*. 2016, pp. 4107–4115.

[46] Kelli Humbird, Luc Peterson, and Ryan McClarren. "Deep Neural Network Initialization With Decision Trees". In: *IEEE Transactions on Neural Networks and Learning Systems* PP (Oct. 2018), pp. 1–10.

[47] Forrest N Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size". In: *arXiv preprint arXiv:1602.07360* (2016).

[48] Irena Ivanova and Miroslav Kubat. "Decision-tree based neural network (Extended abstract)". In: *Machine Learning: ECML-95*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 295–298.

[49] Irena Ivanova and Miroslav Kubat. "Initialization of neural networks by means of decision trees". In: *Knowledge-Based Systems* 8.6 (1995). Knowledge-based neural networks, pp. 333–344.

[50] Eric Jang, Shixiang Gu, and Ben Poole. "Categorical reparameterization with gumbel-softmax". In: *arXiv preprint arXiv:1611.01144* (2016).

[51] Kirthevasan Kandasamy et al. "Neural architecture search with bayesian optimisation and optimal transport". In: *NeurIPS*. 2018.

[52] Cem Keskin and Shahram Izadi. "Splinenets: Continuous neural decision graphs". In: *Advances in Neural Information Processing Systems*. 2018, pp. 1994–2004.

[53] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[54] Peter Kontschieder et al. "Deep Neural Decision Forests". In: *The IEEE International Conference on Computer Vision (ICCV)*. Dec. 2015.

[55] R. Krishnan, G. Sivakumar, and P. Bhattacharya. "Extracting decision trees from trained neural networks". In: *Pattern Recognition* 32.12 (1999), pp. 1999–2009.

[56] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[57] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *NeurIPS*. 2012.

[58] Ya Le and Xuan Yang. "Tiny ImageNet Visual Recognition Challenge". In: 2015.

[59] Yann LeCun, Corinna Cortes, and CJ Burges. "MNIST handwritten digit database". In: *ATT Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* 2 (2010).

[60] Richard Liaw et al. "Tune: A research platform for distributed model selection and training". In: *ICML AutoML Workshop* (2018).

[61] Tsung-Yi Lin et al. "Focal loss for dense object detection". In: *ICCV*. 2017.

[62] Tsung-Yi Lin et al. "Microsoft coco: Common objects in context". In: *ECCV*. 2014.

[63] Zachary Chase Lipton. "The mythos of model interpretability. CoRR abs/1606.03490 (2016)". In: *arXiv preprint arXiv:1606.03490* (2016).

[64] Chenxi Liu et al. "Auto-DeepLab: Hierarchical Neural Architecture Search for Semantic Image Segmentation". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.

[65] Chenxi Liu et al. "Progressive neural architecture search". In: *arXiv preprint arXiv:1712.00559* (2017).

[66] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "Darts: Differentiable architecture search". In: *arXiv preprint arXiv:1806.09055* (2018).

[67] Ning Liu et al. *AutoSlim: An Automatic DNN Structured Pruning Framework for Ultra-High Compression Rates*. July 2019.

[68] Zhuang Liu et al. "Learning efficient convolutional networks through network slimming". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 2736–2744.

[69] Zhichao Lu et al. "Nsganetv2: Evolutionary multi-objective surrogate-assisted neural architecture search". In: *European Conference on Computer Vision*. Springer. 2020, pp. 35–51.

[70] SM Lundberg et al. *From local explanations to global understanding with explainable AI for trees, Nat. Mach. Intell., 2, 56–67*. 2020.

[71] Ningning Ma et al. "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design". In: *arXiv preprint arXiv:1807.11164* (2018).

[72] Dhruv Mahajan et al. "Exploring the limits of weakly supervised pretraining". In: *ECCV*. 2018.

[73] Dmitrii Marin et al. "Efficient segmentation: Learning downsampling near semantic boundaries". In: *ICCV*. 2019.

[74] Mason McGill and Pietro Perona. "Deciding How to Decide: Dynamic Routing in Artificial Neural Networks". In: *ICML*. 2017.

[75] Jieru Mei et al. "AtomNAS: Fine-Grained End-to-End Neural Architecture Search". In: *ICLR* (2020).

[76] Luke Metz et al. "Using a thousand optimization tasks to learn hyperparameter search strategies". In: *arXiv preprint arXiv:2002.11887* (2020).

[77] Abdul Arfat Mohammed and Venkatesh Umaashankar. "Effectiveness of hierarchical softmax in large scale classification tasks". In: *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE. 2018, pp. 1090–1094.

[78] Calvin Murdock et al. "Blockout: Dynamic model selection for hierarchical deep networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2583–2591.

[79] Venkatesh N. Murthy et al. "Deep Decision Network for Multi-Class Image Classification". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[80] Harald Niederreiter. *Random number generation and quasi-Monte Carlo methods*. SIAM, 1992.

[81] Adam Paszke et al. "Automatic differentiation in PyTorch". In: *Proc. Neural Information Processing Systems Workshop on Autodiff*. 2017.

[82] Vitali Petsiuk, Abir Das, and Kate Saenko. "RISE: Randomized Input Sampling for Explanation of Black-box Models". In: *Proceedings of the British Machine Vision Conference (BMVC)*. 2018.

[83] Hieu Pham et al. "Efficient neural architecture search via parameter sharing". In: *arXiv preprint arXiv:1802.03268* (2018).

[84] F Poursabzi-Sangdeh et al. "Manipulating and Measuring Model Interpretability". In: *MLConf*. 2018.

[85] Ilija Radosavovic et al. "Designing Network Design Spaces". In: *CVPR* (2020).

[86] Esteban Real et al. "Large-scale evolution of image classifiers". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 2902–2911.

[87] Esteban Real et al. "Regularized evolution for image classifier architecture search". In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 4780–4789.

[88] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271.

[89] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. 2016, pp. 1135–1144.

[90] Samuel Rota Bulo and Peter Kontschieder. "Neural decision forests for semantic image labelling". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 81–88.

[91] Anirban Roy and Sinisa Todorovic. "Monocular depth estimation using neural regression forest". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 5506–5514.

[92] C Rudin. "Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead. Manuscript based on C. Rudin Please Stop Explaining Black Box Machine Learning Models for High Stakes Decisions". In: *Proceedings of NeurIPS 2018 Workshop on Critiquing and Correcting Trends in Learning*. 2018.

[93] Mark Sandler et al. "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation". In: *arXiv preprint arXiv:1801.04381* (2018).

[94] Ramprasaath R Selvaraju et al. "Grad-CAM: Visual Explanations From Deep Networks via Gradient-Based Localization". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 618–626.

[95] Noam Shazeer et al. "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer". In: *arXiv preprint arXiv:1701.06538* (2017).

[96] Han Shi et al. "Efficient Sample-based Neural Architecture Search with Learnable Predictor". In: *arXiv* (2019), arXiv–1911.

[97] Carlos N Silla and Alex A Freitas. "A survey of hierarchical classification across different application domains". In: *Data Mining and Knowledge Discovery* 22.1-2 (2011), pp. 31–72.

[98] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep inside convolutional networks: Visualising image classification models and saliency maps". In: *arXiv preprint arXiv:1312.6034* (2013).

[99] Chapman Siu. "Transferring Tree Ensembles to Neural Networks". In: *Neural Information Processing*. 2019, pp. 471–480.

[100] Jost Tobias Springenberg et al. "Striving for Simplicity: The All Convolutional Net". In: *CoRR* abs/1412.6806 (2014).

[101] Mandavilli Srinivas and Lalit M Patnaik. "Adaptive probabilities of crossover and mutation in genetic algorithms". In: *IEEE Trans. Systems, Man, and Cybernetics* 24.4 (1994), pp. 656–667.

[102] Dimitrios Stamoulis et al. "Single-path nas: Designing hardware-efficient convnets in less than 4 hours". In: *arXiv preprint arXiv:1904.02877* (2019).

[103] Dimitrios Stamoulis et al. *Single-Path NAS: Device-Aware Efficient ConvNet Design*. May 2019.

[104] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. "Axiomatic Attribution for Deep Networks". In: *International Conference on Machine Learning (ICML) 2017* (2017).

[105] Mingxing Tan and Quoc V Le. "Efficientnet: Rethinking model scaling for convolutional neural networks". In: *arXiv preprint arXiv:1905.11946* (2019).

[106] Mingxing Tan et al. "MnasNet: Platform-aware Neural Architecture Search for Mobile". In: *arXiv preprint arXiv:1807.11626* (2018).

[107] Ryutaro Tanno et al. *Adaptive Neural Trees*. 2019.

[108] Ravi Teja Mullapudi et al. "HydraNets: Specialized Dynamic Architectures for Efficient Inference". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[109] Toan Tran et al. "A bayesian data augmentation approach for learning deep models". In: *NeurIPS*. 2017.

[110] Andreas Veit and Serge Belongie. "Convolutional Networks with Adaptive Inference Graphs". In: *The European Conference on Computer Vision (ECCV)*. Sept. 2018.

[111] Alvin Wan et al. "FBNetV2: Differentiable Neural Architecture Search for Spatial and Channel Dimensions". In: *CVPR* (2020).

[112] Alvin Wan et al. "NBDT: Neural-Backed Decision Tree". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=mCLVeEpplNE.

[113] Kuan Wang et al. "HAQ: Hardware-Aware Automated Quantization with Mixed Precision". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 8612–8620.

[114] Linnan Wang et al. "Neural architecture search using deep neural networks and monte carlo tree search". In: *AAAI*. 2020.

[115] Wei Wen et al. "Learning structured sparsity in deep neural networks". In: *Proc. Advances in Neural Information Processing Systems*. 2016, pp. 2074–2082.

[116] Wei Wen et al. "Neural Predictor for Neural Architecture Search". In: *ECCV* (2020).

[117] Bichen Wu et al. "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 10734–10742.

[118] Bichen Wu et al. "Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions". In: *arXiv preprint arXiv:1711.08141* (2017).

[119] Bichen Wu et al. "Squeezedet: ified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2017, pp. 129–137.

[120] Bichen Wu et al. "Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud". In: *ICRA*. 2018.

[121] Mike Wu et al. "Beyond sparsity: Tree-based regularization of deep models for interpretability". In: *In: Neural Information Processing Systems (NIPS) Conference. Transparent and Interpretable Machine Learning in Safety Critical Environments (TIML) Workshop*. 2017.

[122] Yuxin Wu et al. *Detectron2*. https://github.com/facebookresearch/detectron2. 2019.

[123] Saining Xie et al. "Aggregated residual transformations for deep neural networks". In: *CVPR*. 2017.

[124] Sirui Xie et al. "SNAS: Stochastic neural architecture search". In: *arXiv preprint arXiv:1812.09926* (2018).

[125] Chenfeng Xu et al. "SqueezeSegV3: Spatially-Adaptive Convolution for Efficient Point-Cloud Segmentation". In: *ECCV* (2020).

[126] Brandon Yang et al. "Condconv: Conditionally parameterized convolutions for efficient inference". In: *Advances in Neural Information Processing Systems*. 2019, pp. 1307–1318.

[127] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. "Designing energy-efficient convolutional neural networks using energy-aware pruning". In: *arXiv preprint arXiv:1611.05128* (2016).

[128] Tien-Ju Yang et al. "NetAdapt: Platform-aware neural network adaptation for mobile applications". In: *Proc. European Conf. Computer Vision*. Vol. 41. 2018, p. 46.

[129] Zhaohui Yang et al. "Cars: Continuous evolution for efficient neural architecture search". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 1829–1838.

[130] Hongxu Yin et al. "Dreaming to Distill: Data-free Knowledge Transfer via DeepInversion". In: *CVPR* (2020).

[131] Jiahui Yu et al. "Bignas: Scaling up neural architecture search with big single-stage models". In: *ECCV* (2020).

[132] Jiahui Yu et al. "Slimmable Neural Networks". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=H1gMCsAqY7.

[133] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European Conference on Computer Vision (ECCV)*. Springer. 2014, pp. 818–833.

[134] Arber Zela et al. "Towards automated deep learning: Efficient joint neural architecture and hyperparameter search". In: *arXiv preprint arXiv:1807.06906* (2018).

[135] Hang Zhang et al. "ResNeSt: Split-Attention Networks". In: *arXiv preprint arXiv:2004.08955* (2020).

[136] Hongyi Zhang et al. "mixup: Beyond empirical risk minimization". In: *ICLR* (2018).

[137] Jianming Zhang et al. "Top-down neural attention by excitation backprop". In: *European Conference on Computer Vision (ECCV)*. Springer. 2016, pp. 543–559.

[138] Chenzhuo Zhu et al. "Trained ternary quantization". In: *arXiv preprint arXiv:1612.01064* (2016).

[139] Barret Zoph and Quoc V Le. "Neural architecture search with reinforcement learning". In: *arXiv preprint arXiv:1611.01578* (2016).

[140] Barret Zoph et al. "Learning Transferable Architectures for Scalable Image Recognition". In: June 2018, pp. 8697–8710. DOI: 10.1109/CVPR.2018.00907.