EXPLOITING IMAGE RESOLUTION HOLISTICALLY IN COMPUTER VISION MODELS

EDDIE YAN

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:
Luis Ceze, Chair
Xi Wang
Zachary Tatlock

Program Authorized to Offer Degree:
Computer Science & Engineering

ABSTRACT

EXPLOITING IMAGE RESOLUTION HOLISTICALLY
IN COMPUTER VISION MODELS

Eddie Yan

Chair of the Supervisory Committee:
Professor Luis Ceze
Computer Science & Engineering

Modern computer vision systems are built upon a complex stack of hardware and software, from general-purpose processors to specialized accelerators, and low-level operator libraries to expressive deep learning frameworks. However, from this complexity arises many opportunities for optimization across the stack. From the lens of image resolution, a fundamental hyperparameter of computer vision, we propose methods for optimizing models and characterize the space of choices as introduced by the hyperparameter of resolution.

In the process, we cover related topics such as object scale (as introduced by data augmentations), image storage (including methods for efficient multi-resolution storage), and deep learning kernel tuning. An understanding of these topics allows us to consider resolution with respect to deep learning choices holistically, enabling efficient inference from the metrics of computational cost, latency, accuracy, and storage bandwidth use. We describe the mechanisms which enable efficient inference according to these metrics, spanning kernel tuning, image data layout, and model architecture pipelines.

ACKNOWLEDGEMENTS

Grad school is a perilous trail where many have helped me.

Much of the work contained here was contributed by coauthors and collaborators, notably Luis Ceze, Kaiyuan Zhang, Xi Wang, Karin Strauss, Tianqi Chen, Lianmin Zheng, and Yanping Huang in roughly chronological order. I have been guided by the inexorable optimism of Luis, who amazingly is simultaneously the most senior and least cynical computer architect that I know. Beyond Luis, I must thank all those whom aided me along the way. My mentors in industry: Daniel Johnson and Mark Stephenson at NVIDIA and Yanping Huang at Google.

The sampanistas, especially those in my cohort: Meghan Cowan, Emily Furst, Ming Liu, and Liang Luo. The ΞΞΞΞΞΞΞΞΞ, which includes Amrita Mazumdar and Luis Vega among those not already named. The original bachΞΞΞΞΞΞΞ-ers. The greater samp(l)(a) community.

Jeff Huang and Gifford Cheung, whose mentorship paved the way for my path to a PhD. Mario Saenger, who had the patience to introduce a high school student to research. My parents, who never questioned my journey despite having endured a much tougher time in grad school than I did.

Those in the illustrious twitch chats that I have frequented [gypsy9W, nyokenFace, jaeyunThink, terroruThumbsup, artoFace, x80smuXn00bmulLet, BlanderStrudel, x17steRobocopsagi, ClicksNCuts, justto5Think, salW, juggrW].

1

INTRODUCTION

INTRODUCTION Computer vision models using deep learning are becoming increasingly powerful in their ability and accuracy. However, their rapidly improving utility is often surpassed by their massive computational requirements. As a consequence of these requirements, machine learning researchers and computer architects have looked towards classical tradeoff spaces to find ideal operating points for models. These tradeoff spaces leverage techniques such as quantization (reducing the precision of numerical representations used by the model) and pruning (reducing the number of nonzero model weights), to varying degrees of success in improving model efficiency. The challenge with many methods is that approaches may either be hardware friendly (introducing little additional irregularity in execution) or model friendly (introducing few constraints on model weights), but rarely both. The goal of this work is to introduce an additional tradeoff space for efficient models: resolution, which has the potential to be both model and hardware friendly.

At a glance, neural networks depend on a tall stack of supporting technologies to enable performance and developer productivity. Fundamentally, achieving efficient model implementations depends on highly specialized (often vendor-specific) linear algebra libraries (e.g., BLAS, MKLDNN, and cuDNN), as naive implementations of operators can trail libraries by several orders of magnitude. Similarly, the de facto standard general-purpose image training set ImageNet [75] comprises 1.2 million images. To enable developer productivity on large datasets while maintaining hardware utilization, deep learning frameworks such as PyTorch [69] handle tasks such as automatic differentiation, parallel data augmentation preprocessing, and model checkpointing, to name a few example of the breadth of software support required. Additionally, the model architectures of computer vision models are themselves highly complex arrangements of deep learning operators—the latest architectures are often the result of human insight paired with brute-force machine optimization [62, 74, 90]. This superficial glance at the technology stack behind neural network reveals that neural network pipelines for computer vision are staggeringly complex. However,

from this complexity, a rich set of opportunities for optimization emerges. Performing neural network inference touches nearly every aspect of system design and implementation, from storage (images and image formats), to computer architecture (for kernel or operator-level optimizations), with the neural network architecture and model execution pipeline at the highest level.

From the singular hyperparameter of image resolution, we show methods and opportunities for enabling efficient neural network inference at multiple levels of the system stack. While image resolution itself is a simple concept, the choice of image resolution has a nuanced impact on system requirements throughout the neural network stack. At the basic level, the amount of data that must be stored and read scales with image resolution. Model accuracy generally increases with image resolution [90, 94]. Due to the typical design of current convolutional model architectures, computational complexity scales roughly quadratically with image resolution.

However, each of these relationships holds nuances that either can enable favorable tradeoffs or prevent expected efficiency gains from materializing. For example, while computational complexity may scale quadratically with image resolution, realizing computation savings in terms of wall-clock time may be difficult if reducing complexity also serves to reduce hardware utilization given a particular computer architecture and algorithm implementation. On the opposite side, decreasing image resolution may increase model accuracy compared to existing approaches when carefully compensating for the change in image scale.

THE ISSUE OF SCALE DEPENDENCE From a neural network's perspective, image resolution, combined with the cropping or framing of object(s) in the image, determines the apparent scale of objects. At training time, a typical approach is to apply data augmentation so that objects are presented at a wide range scales and contexts (with various occlusions of the object and background) [56]. However, despite the effectiveness of data augmentation in improving a model's robustness to object scale, they remain sensitive to the distribution of object scales seen at training time.

This issue raises a fundamental question about neural network features. If scale is an important attribute of objects, do models explicitly capture or encode scale in their intermediate representations? If models indeed represent scale, then this property can potentially be leveraged to "normalize" the scale of objects ahead of time to improve inference performance.

While intermediate activations of neural networks are typically difficult to interpret directly, we study the sensitivity of activations to object scales using a model pipeline that uses neural network activations (from a pretrained model) as input and an alternative training objective intended to rank the relative scales of input examples.

Using this pipeline, we measure the predictive power of neural network activations on the ranking task to understand the extent to which activations encode scale, along with attributes corresponding to other data augmentations. Additionally, we weigh the relative contributions of activations from different layers to gauge which layers encode scale information the most. If models capture scale information, this property enables them to discern the image resolution that maximizes inference accuracy. Chapter 3 studies the effect of scale (due to cropping) alongside other augmentations from the perspective of neural network activations.

ENABLING PERFORMANCE AT DIFFERENT RESOLUTIONS As alluded to previously, while the apparent scale of objects is modulated by the resolution and crop size of an input image, the true computational cost of inference given an image resolution depends on the implementation strategy of each operator in the model. The relative efficiency of inference typically decreases with decreasing resolution, as hardware utilization falls and/or the share of computation that cannot be parallelized begins to dominate the total execution time. While losses in efficiency can be partially recovered by crafting custom kernels for every operator and resolution combination, this process quickly becomes tedious and intractable given the large number of operators present in even a single model, resolution combination. The difficulty in handcrafting implementations is amplified by the present diversity of hardware architectures and memory hierarchies: the optimal implementation strategy will vary widely between a CPU with main memory caches and specialized vector extensions and a GPU with a large register file and high-bandwidth scratchpad memories. Recent work has proposed automatic compiler optimizations that use machine-learning guided search [15, 17, 110] to rapidly comb through a wide range of candidate implementations, and we build on this effort to generate efficient implementations for each resolution, studying ways to reduce the time spent during the search process.

An alternative presentation of the search process for generating efficient implementations takes the form of multiple phases where each phase can apply varying strategies. A natural arrangement is a *proposal* phase that selects potentially efficient implementations based on prior knowledge, and a *prediction* phase that estimates the performance of the proposed candidate implementations. This separation enables a modular pipeline where the performance of different proposal and prediction strategies can be compared which each have distinctly important roles in the search pipeline. Proposal strategies must balance exploitation (choosing kernel configurations that are likely to be good) and exploration (choosing kernel configurations that are from underexplored regions of the space. Prediction strategies should balance the accuracy and compute cost of the prediction model:

at the extreme end of compute cost, it would be cheaper to measure the performance of configuration on actual hardware than to use a prediction model. Chapter 4 evaluates various proposal and prediction pipelines for automatic search of machine learning kernels.

ENABLING STORAGE EFFICIENCY AT DIFFERENT RESOLUTIONS Exposing multiple resolutions as a hyperparameter can reduce computational complexity when low resolution inference is used, but wastes storage bandwidth if full images are loaded for every resolution. A similar issue also occurs when storing and serving multiple resolution versions of images, especially for large datasets such as those found for social media services. While the motivation for serving multiple resolution versions (images are shown in different contexts and devices vs. for different scales/model quality effects) differs, the requirements are similar. By leveraging properties of frequency domain representations that are common in popular image formats, we can specialize the data layout of images to match the desired resolutions. Specializing the data layout requires only a single copy of each image, without redundant lossy copies for lower resolution or quality versions. For neural network inference, this enables reading a different relative fraction of each image's total data for inference at a per-example granularity, without modification to the downstream neural network pipeline. Chapter 4 describes the methods and evaluates multi-resolution image storage via frequency domain data layouts.

END-TO-END SPECIALIZATION FOR RESOLUTION With an understanding of how the choice of image resolution can be used to tune model quality, and how to enable efficient inference (in terms of computation and storage), we study the impact of end-to-end specialization for image resolution at inference time. As modern model architectures are agnostic to input resolution (when looking purely at input and operator shapes, rather than model quality), a natural framing of this specialization is the task of choosing the best resolution for each input image. From a storage perspective, the objective is to balance resolution-adjacent choices such as the crop size and image quality of each image to minimize the amount of data that needs to be read from storage. Additionally, we note the importance of specializing the implementation of each model for a given resolution.

We propose a two-model pipeline that first predicts the optimal resolution for inference using a lightweight model followed by a larger backbone model that performs inference at the selected resolution. In tandem, the amount of data read from storage is calibrated to the model's preferences for image quality at each resolution, and the implementation of the model at each resolution is tuned to maximize utilization. Chapter 6 provides

more details on end-to-end approaches on specializing for image resolution.

In this thesis, we propose and evaluate support for efficient multi-resolution image inference in terms of model accuracy, storage, computation, spanning image formats, compute kernel tuning, and a dynamic resolution approach that removes the need to statically choose a resolution ahead of time, finding a favorable accuracy vs. compute cost tradeoff. We show that specializing for resolution improves performance, increases accuracy, and reduces the amount of data read from storage. Additionally,we show that dynamic resolution approach is a viable alter-native to finetuning for a specific object scale.

2

RELATED WORK

The area of machine learning systems comprises a rapidly expanding body of work, a testament to the importance of the field and the sheer quantity of topics it covers. While the core objectives of machine learning systems research can often be summarized with a few key metrics pertaining to model quality, algorithmic efficiency, and computational costs, the methods for achieving these objectives are diverse and have spawned numerous subfields. Here, we cover the most relevant related work for machine learning systems optimizations.

Improving the the computational efficiency of neural network inference (both with and without retraining) is a rapidly evolving field of research due to the high computational costs associated with modern convolutional architectures. Frequently, these approaches introduce a quality–computational cost or accuracy–computational tradeoff space. Beyond resolution, architecture agnostic approaches include exploiting quantization [25, 72, 113], weight pruning [24, 43], input masking [104], temporal redundancy [11], model cascades [79], and alternative numerical representations [46, 50, 58].

QUANTIZATION Neural network quantization can take many forms, ranging from truncation (removing bits of precision) under existing representation schemes to alternative representations. Regardless of the approach, the primary aim of quantization is to improve the efficiency of model inference by reducing the complexity of the hardware or the number of software operations required due to the lower bitwidth of operands. Another important form of savings from quantization comes in the form of the reduction in memory requirements, particularly for the intermediate activations and weights of a model.

Beyond the method by which numerical representations can be quantized, much of quantization research focuses on how quantization can be performed while minimizing the loss in model quality. Approaches include finetuning the model after quantization and specializing the extent of quantization at a per-channel or per-layer basis. Finally, we note the im-

portance of matching quantization schemes to hardware/software friendly approaches in realizing theoretical performance gains in real applications.

NEURAL NETWORKS AND IMAGE RESOLUTION The issue of scale dependence (often stemming from variations in image resolution) is an area of active research with model architecture [48], finetuning [94], data augmentation [34], and equivariance-based approaches [84] being used. While we present one of many possible motivations for multi-resolution models and one possible strategy for addressing the problem of scale dependence in vision models, the ability for an inference stack to flexibly switch between different image resolutions is useful. Even in the ideal case of fully scale-equivariant models, image resolution remains a tunable hyperparameter dictating the amount of information or fine-grained detail in image in addition to the capacity of feature maps. The overall approach of mapping image data to different resolutions and tuning resolution specific kernels is orthogonal to the motivation behind multi-resolution support.

OPTIMIZING NEURAL NETWORK STORAGE Specializing storage with domain-specific knowledge for either increased capacity and performance is also an active area research [42, 64, 76]. Prior work has touched on cases where the relative importance of image data (e.g., critical format bits vs. noisy coefficient values) can be matched to storage at different levels of reliability [30, 42].

At training time, retaining the intermediate activations of models for backpropagation can become problematic from a memory capacity perspective. Here, recomputation [14, 51] can be used to favorably trade additional computation for memory storage requirements when training large models.

MIXTURE OF EXPERTS MODELS We draw inspiration from Mixture-of-Experts (MoE) approaches in machine learning [41, 59, 78, 103], where model architectures use a weighted combination of "experts" to increase model capacity. Here, our two-model pipeline can be considered a modified MoE that uses weight-sharing, with the different experts being the different resolution variations of the backbone model. While prior work has enforced sparse-weighting [78] or soft-conditioning [103] to efficiently implement the MoE, we use explicit control flow and train the scale and backbone models on separate objectives.

STORING MANY IMAGES The need to efficiently store many images has become apparent with the overwhelming growth of social media services that often host and serve images to a wide variety of users and devices.

Recent work has aimed to reduce overheads due to metadata for small files [8] as well as develop SSD friendly caching algorithms [92]. Related work has also investigated the quality–density trade-off for approximate storage, showing that matching the importance of image data with the reliability of storage can improve storage efficiency [30]. Using custom progressive JPEG limits metadata overheads when only storing a single version of each image and can improve caching behavior as different versions of an image share data. Grouping scans of progressive JPEG is related to ordering image data from most to least important, but the binary format used here is not amenable to storage on approximate storage media.

Progressive JPEG images can also be partially deleted gracefully by discarding high frequency data first—improving storage elasticity. The concept of motifs: descriptions of computation needed to reconstruct a file discussed in [80, 81] is implicitly implemented by a dynamic resizing storage scheme as only the highest quality version of an image is stored while lower quality versions are implicitly defined by motifs.

Dynamic resizing has precursors in image processing systems such as zimg [115] that allow clients to upload and request images with added operations such as cropping and scaling. To the best of our knowledge, these systems do not vary the amount of data read based on quality via a progressive frequency domain encoding. Dynamic resizing has also been used by Flickr [1] and Facebook [36]: in addition to storing multiple versions of each photo, Facebook incorporates "Resizers" when the requested version requires additional processing. Finally, progressive JPEG has been recently used by Facebook [6] to reduce data consumption and speed up the apparent loading of images on the client side; the latter is achieved by rendering an acceptable quality scan before all scans have been transmitted. However, this approach does not involve dynamic resizing or customizing progressive JPEG.

OPTIMIZING DEEP LEARNING WORKLOADS An important requirement of efficient multi-resolution or scale support is the availability of high performance kernel implementations for each combination of resolution, model, and hardware. As the number of such possible combinations grows very quickly, we rely on work in automatic deep learning kernel optimizations [15, 20, 71, 110] to generate these kernels with minimal programmer effort.

To evaluate tuning pipelines, we present an environment for deep learning kernel optimization; this environment can be analogous to simulation environments for reinforcement learning agents, where researchers can prototype ideas and algorithms without needing to build a virtual or physical world from scratch. Ray [67] is an analogous environment for reinforcement learning. Our approach differs in its performance-driven objective that can leverage measurements on real hardware and the ready

availability of out-of-the-box benchmarks (e.g., reference workloads such as popular computer vision models). Park [63] is an environment for reinforcement learning-based optimization for general systems challenges, such as device placement, circuit design, and load balancing. While reinforcement learning can also be applied to the challenge of automatic kernel optimization, we focus on providing generic support for optimization pipelines for a wide breadth of hardware devices and kernels. Similarly, Vizier [27] presents an optimization service for general blackbox functions. The proposed environment can be viewed as a Vizier-like service for white-box deep learning kernels where support for experimenting with the optimization algorithms themselves is a design goal.

On the benchmark front, our environment is analogous to recent work such as NAS-Bench-101 [106] which provides a reference dataset for NAS alongside benchmark tasks. However, we while we provide a dataset, we do not settle on a fixed dataset as hardware targets and models are continuously evolving. Our benchmark tasks have a similar flavor to the NAS-Bench tasks, as neural architecture search presents a discrete optimization problem similar in structure to that of program optimization. Concretely, the main differences are in the feasibility of evaluation (seconds vs. hours to evaluate candidates), and the objectives (performance vs. validation accuracy). Additionally, we encourage users to collect performance data and run experiments on hardware that they have available, whereas this may not be feasible for NAS-Bench tasks. However, it is entirely possible that variations of successful kernel optimization algorithms and pipelines can be successfully applied to neural architecture search, and we hope to see cross-pollination across the problems.

The task of automatic program optimization is not new, and has been recently visited with several different approaches, such as analytical models [68], statistical cost model guided search [16], tree search [3], static cost models [49], and statistical cost models driven by handpicked features [60]. We aim to provide a flexible framework for researchers to explore new innovations. Additionally, parallel work on machine learning for systems work includes automatic device placement [65], graph optimization [45], parallelization [44], architecture search [89] [116], among others.

3

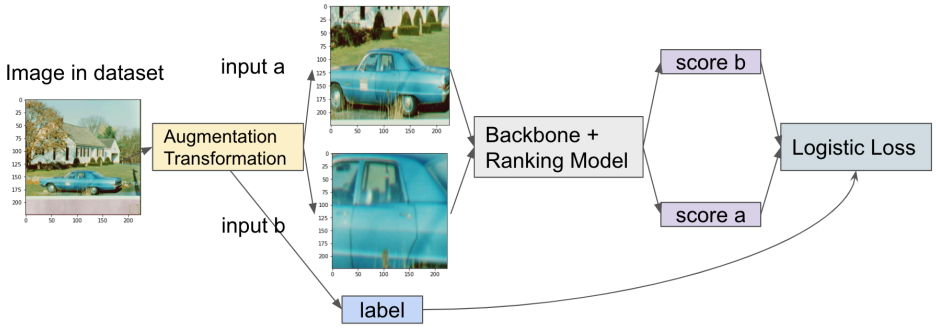
WHAT DO COMPUTER VISION MODEL FEATURES ENCODE?

INTRODUCTION

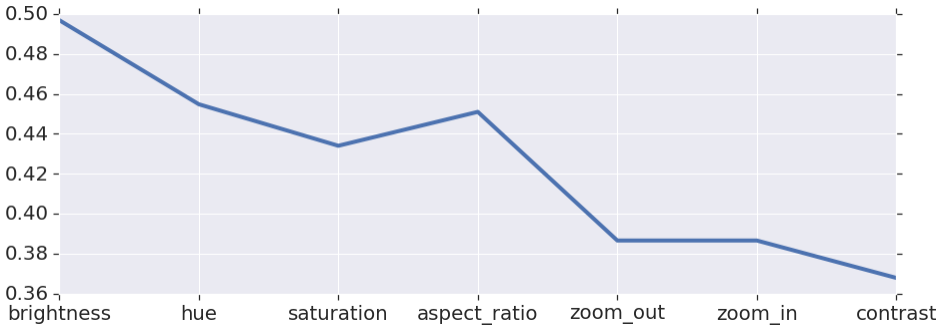
Convolutional neural networks (CNNs) have enjoyed tremendous success on popular computer vision problems. Ideally, vision models for these tasks would be equivariant to perturbations such as color, translation, scale, and rotation. Translation invariance has been partially architected in CNNs [108], and building models with other equivariant properties is an active area of research. These properties include rotation, reflection, and scale among others [19, 47, 54, 85, 101, 114]. In spite of their success, CNN models remain worryingly sensitive to small changes [28] in the training data with respect to desirable equivariants. The typical [56], yet effective [107] approach to build robust models is to leverage brute force via data augmentation.

However, current understanding of the effects of data augmentations is limited, and using data augmentations often requires ad-hoc or task-specific heuristics. An instance of this problem occurs when objects are shown to models at different scales: popular models for classification exhibit a noticeable drop in accuracy when the scale of their test-time data does not match that of their training-time data [93]. Here, the proposed heuristic is to finetune the models for the expected distribution of test resolutions—information that may not be easily available. In parallel, we observe that enhanced data augmentation can lead to dramatic improvements in accuracy, especially in adversarial circumstances [99], but this requires rearchitecting models to effectively leverage adversarial examples.

The importance of data augmentation leads to natural questions about what useful concepts models learn from data augmentations. As data augmentations are often intended to reflect natural priors (e.g., objects belonging to the same class have variations in scale), a relevant question is how these priors are captured by the model. Concretely, we ask whether variations corresponding to data augmentations are encoded by models, and where this encoding takes place. For example, do models encode bright-



(a) Training pipeline used in our evaluation.



(b) Relative importance of the first block of ResNet-18 for predicting each of the data augmentation ranking tasks.

Figure 1: Can layer activations from CNNs encode input variations introduced by data augmentation? For a given image, a pair of inputs is generated by varying the extent of a data augmentation (e.g., scale), along with a label ranking the extent of the augmentations. The inputs are then fed to a frozen backbone model to extract features for a pairwise ranking model. Figure 1b shows that early ResNet layers are more important for encoding low-level augmentation transformations (brightness and saturation).

ness variations in the earlier layers, in the later layers, or both? Which data augmentations correspond to *low-level* model features, and which correspond to *high-level* model features?

We search for answers to these questions by investigating whether intermediate activations of models capture input differences introduced by data augmentation. First, we define a set of attributes (scale, aspect ratio, and color transformations) that are desirable invariants (equivariants) for models and commonly targeted by the data augmentation of current computer vision models [21]. Following these definitions, we propose several experiments, introducing a data augmentation ranking task, as illustrated in Figure 1a, to understand whether CNNs implicitly learn a representation for these attributes, comparing against baseline models relying on primi-

tive features. These experiments measure the predictive performance of a ranking model that uses intermediate features collected from pre-trained models to predict augmentation attributes. Following these experiments, we inspect the relative importance of features used in the ranking model to understand the relative importance of layers in modeling data augmentation attributes.

Our results show that CNNs implicitly learn to encode attributes of popular data augmentations, such as scale, aspect ratio, saturation, and contrast without being explicitly trained on these objectives. Additionally, we find that these attributes are typically encoded in the earlier layers of networks, suggesting that models learn to normalize input variations introduced by data augmentations. Later layers appear relatively more important for aspect ratio and scale, which can be considered higher-level than attributes such as brightness and saturation, as shown in Figure 1b. We present data augmentation prediction as tool to improve the currently limited interpretability [61] of CNNs.

A RANKING MODEL FOR AUGMENTATIONS

To assess whether neural network features encode data augmentation transformations, we propose a ranking task that predicts the *relative* extent of augmentation attributes given intermediate neural network features. We employ a ranking model instead of a regression approach since obtaining the *absolute* extent of augmentation is difficult. For example, for the task of predicting the scale of an object, it is difficult to design a numerical definition of scale that is consistent across many different input examples and object classes. We use a separate ranking model as it facilitates interpretability over blackbox approaches that only consider the final output or accuracy of model predictions. As we show in subsection 3.5.1, we can leverage the ranking model weights to infer the importance of different layers to the ranking tasks.

To circumvent the requirement of precisely-labeled data for augmentation attributes, we only try to rank the relative values of augmentation attributes. We use pairwise rank-loss [18], which can be considered a binary classification task for pairs of input examples. For the case of scale, the task is to decide whether the scale of the object in one image is greater than the scale of the object in the other. More formally, for each i, j pair of examples the loss function is defined as

$$\log\left(1 + \exp(-\operatorname{sgn}\left(v_i - v_j\right) \times \left(f\left(x_i\right) - f\left(x_j\right)\right))\right)$$

where v_i, v_j, x_i, x_j, and f denote the true augmentation parameters, input to the ranking model, and ranking model respectively. This is equivalent

Figure 2: Example of our definition of scale (row 1), aspect ratio (row 2), hue (row 3), and saturation (row 4). We order the extent of each augmentation transformation from left to right.

to logistic loss where each label is determined by the predicate $v_i > v_j$. For each image in the dataset, we produce pairs of images by applying an augmentation transformation parameterized by different random values.

CHOOSING AND DEFINING AUGMENTATIONS

We describe our definitions of scale, aspect ratio, hue, contrast, saturation, and brightness in this section, focusing on the constraint that our definitions must yield an ordering or ranking of input examples. Figure 2 shows examples for some augmentations considered. We choose these augmentations based on the following criteria: (1) Ease of implementation: given an unlabeled set of images, it is straightforward to infer an ordering of these augmentations For example, smaller crops correspond to a larger view of the same object. (2) Popularity in training pipelines: each of the transfor-

mations considered are either partially or fully implemented in standard TensorFlow [2]. (3) Diversity in abstraction level: scale and aspect ratio can be considered higher level image features that require some degree of understanding, whereas color attributes can almost be directly inferred from raw pixel values with limited context.

Scale

We carefully settle on a narrow definition of object scale, avoiding semantic definitions of scale, especially between different objects. For example, we are not attempting to assess whether models capture facts such as "elephants are bigger than dogs." We choose a pragmatic definition of scale corresponding to the *solid angle* of an object or the proportion of the field of view occupied by an object.

This definition of scale captures the problem exhibited by the "train-test" resolution discrepancy [93], where test-time crops of images that occupy a smaller area than training-time crops reduce model accuracy and reflects the random cropping augmentation method that is commonly used to present objects of different scales at training time. This definition is also distinct from resolution; one can craft arbitrary examples where both high and low resolution images of the same object map to the same scale after they are cropped and resized.

Additionally, we add the qualification that we consider scale to be invariant to occlusion or cropping as long as the object is still partially visible in the frame. We use this qualification to disentangle scale from the related but separate concept of *bounding-box area* occupied by an object in a frame. Figure 2 gives examples following our definition of scale. Section 3.3 describes our sampling process and the range of scales considered.

From this definition of scale, we define two ranking tasks: "zoom-out" and "zoom-in." For the "zoom-out" task, we generate pairs of input images that zoom-out from the bounding boxes of objects to generate input images with different scales. We uniformly sample two values in the range $[0.1, s]$, where s is the smallest of the total vertical or horizontal distance from the border of the bounding box to the boundaries of the image. For the images in the dataset we use (subsection 3.4.1), s is expected to be at least 0.3. For the "zoom-in" task, the different scales are generated by zooming-in on bounding boxes to different extents. We uniformly randomly sample two values in $[0.5, 0.9]$ that determine the fraction of the bounding box to trim before resizing the result to the input size of the backbone model 224×224 for each pair of inputs. We define the zoom-in and zoom-out tasks separately because although they may be of similar difficulty for a human evaluator, intuitively the zoom-out task may be easier as the area occupied

by an object is a highly accurate proxy for scale when the object of interest does not occupy the entire frame.

Aspect Ratio

Models are naturally exposed to a range of aspect ratios of objects at training time through random cropping and natural variation in the input distribution. Random cropping is an important source of aspect ratio variation, as many augmentation pipelines do not consider the original aspect ratios of objects as a constraint on the crop dimensions. With respect to aspect ratio, we define the ranking order from wide to thin, or the ratio of vertical to horizontal pixels present in the input after cropping (but before resizing). Note that while ordering the aspect ratio between two arbitrary objects is difficult, and this definition suffices when only considering different crops of the same object.

The aspect ratio task uses the same pipeline as the scale tasks, with the objective changed to ranking the ratio of vertical to horizontal pixels. To generate each input image, we sample four random uniform values in $[0.4, 0.4]$ that determine the number of horizontal and vertical pixels to trim from each input image.

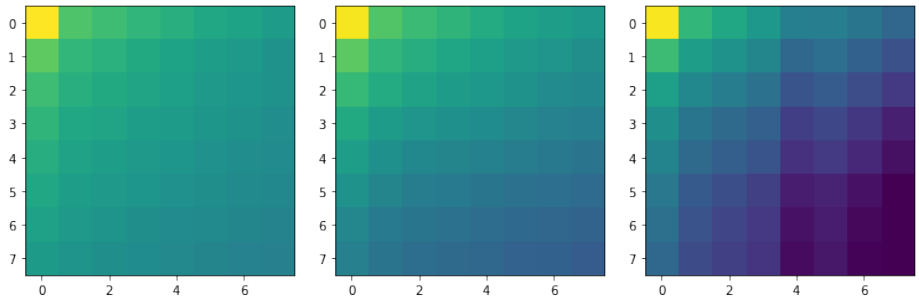
Hue, Saturation, Contrast, Brightness

Hue, saturation, and contrast are common distortions applied to input images. As each of these augmentations are parameterized by either relative multipliers or absolute deltas to the original image, these parameters lend themselves naturally to an ordering for ranking. We include brightness as a sanity check that should be trivially encoded for both the CNN backbones and baselines. While we consider contrast a color transformation, it is arguably higher-level than the other augmentations as discerning contrast requires non-local information.

We again sample of random uniform values for each ranking task. For both saturation and contrast, we sample the relative multipliers used to apply the transformation to determine the ranking labels (in the range $[0.5, 1.5]$). For hue, we rank the delta relative to the original image (in the range $[-0.2, 0.2]$).

METHODOLOGY

To understand whether CNN activations capture attributes of data augmentations, we adopt an experiment pipeline similar to one used to extract position information from CNNs [40]. We also use the intermediate activations as input to a predictor from a pre-trained vision model with



(a) (b) (c)

Figure 3: Average magnitude of frequency coefficients of an 8×8 DCT applied patch-wise to images at increasing scales (from left to right). Frequency coefficients are ordered in a zig-zag pattern, with lowest frequency in the top left and highest frequency in the bottom right. The magnitude of higher frequency coefficients decreases as scale increases.

frozen parameters, but with several key differences. Instead of attempting to generate a two-dimensional output, our prediction task is learning to rank input examples according to their data augmentations. Our ranking model uses only average pooling and a single linear layer to allow easy interpretation of the model weights. In the case of position information, the ground-truth can be generated deterministically, and it is the same across all images. However, in the case of general data augmentation, ranking labels are generated on-the-fly, in tandem with the augmentations.

Dataset

We use a subset of the ImageNet [22] training dataset in our experiments. Specifically, we limit our subset to images that have exactly one bounding box to mitigate the effect of partially cropping only some objects in view. We also choose images with bounding boxes that span at least 30% of the input image, with the additional requirement that the borders of the bounding box must be at least 30% of the image dimensions away from edges of the image. Together, these requirements ensure that there is range to zoom out from bounding boxes and to provide reasonable resolution when zooming in on a bounding box. These constraints reduce the original 1.2 million image ImageNet dataset to roughly $86,000$ images, which we split into a $65,000$ image training set and a $21,000$ image validation set. For simplicity, we use this dataset for all of our ranking tasks, even those that do not require bounding box constraints.

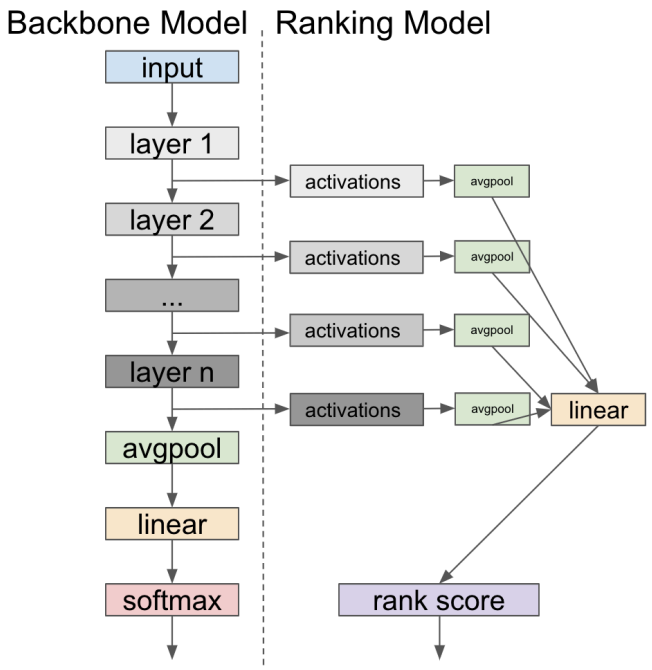


Figure 4: Backbone and ranking model used in our evaluation. The backbone model is a pre-trained CNN (such as ResNet-18), with parameters frozen. The activations from the backbone are average-pooled to align their spatial dimensions and fed to a linear layer that produces a score for the ranking objective.

Baseline Comparisons

We also evaluate two baselines that either use an 8×8 discrete cosine transform (DCT) to generate features (to understand the impact of frequency information), or are passed the input images directly (passthrough). Figure 3 shows an example of how the magnitude of frequency coefficients change with the scale of an object. For the DCT baseline, we apply average pooling to the DCT features while the spatial dimensions of the passthrough baseline are not reduced.

Ranking Model and Training Pipeline

Our ranking model uses the intermediate activations from a pre-trained CNN as inputs to rank instances of a given data augmentation transformation. Figure 4 shows a high-level diagram of the relationship between the backbone model and the ranking model. For our experiments, we use ResNet-18/50 [32] as the backbone, although this approach is compatible with any feedforward CNN. To unify the spatial dimensions of each layer,

we apply global average pooling to reduce each activation tensor to a tensor with 1×1 spatial resolution, preserving the channels. The average-pooled tensors are then fed to a single linear layer that computes the ranking score for a given input example. For each each pair of input examples, we use the ranking scores and logistic loss to fit the linear layer's parameters.

The training pipeline begins with iteration through a dataset of images, where each image is used to generate a pair of input examples. Each input example is transformed according by sampling a random variable and the current augmentation ranking task (e.g., scale). At this time, a label for this pair of input examples can be computed as a boolean expression of the random variables (e.g., scale_a > scale_b?). A collection of pairs and labels comprise a batch that is used to fit the linear layer with logistic loss. Note that the parameters of the backbone model are frozen during training of the ranking model to prevent the ranking task from affecting the intermediate features of the backbone. We use the same approach with the baselines, with average pooling omitted for the passthrough baseline.

Where are data augmentations encoded?

We use the weights of the linear layer to measure the relative importance of the activations for each layer of the backbone model. Due to the simplicity of the linear ranking model, we can measure the contribution of each layer of the backbone by taking the product of the weights and the corresponding standard deviation in the layer activations.

EVALUATION

We begin the evaluation with the accuracy results (Table 1) for each of the pairwise ranking tasks. Due to the binary nature of a pairwise ranking task, the accuracy of random guessing is 50%. For all tasks, we find that the ResNet backbones either match or substantially outperform the baselines, particularly on the augmentations that do not manipulate color. This suggests CNNs may implicitly model scale and aspect ratio as components of features.

Prior work has compared the early layers of CNN to the discrete cosine transform (DCT) [29]. To some extent, we expect the DCT (Figure 3) and low-level features of earlier layers to act as a proxy for scale and/or aspect ratio. Intuitively, two views of the same object at different scales are expected to contain different frequency domain representations, where the smaller scale view is expected to have more high frequency components than the larger scale view. The details of the object exhibit higher spatial frequency as they appear finer in the image. If CNNs capture some ele-

ments of frequency domain transforms in convolution layers, we would expect that this information could be used to better infer scale information. Other augmentations, such as hue and saturation, may present cues in the absolute or relative values of the color channels early in network architectures.

When comparing results for the scale tasks, we note that the performance of the ResNet backbone was substantially lower for the "zoom-out" than "zoom-in" task. This drop in accuracy was surprising as it was thought that the ranking model could rely on the later layers and localization as a proxy for scale, although it is possible that the use of average pooling in the ranking model could have limited localization information. Additionally, performance on the zoom-out task may have suffered as a consequence of it being more fine-grained than the zoom-in task: many images may have a limited amount of slack in which crop sizes can be increased without overstepping image boundaries. Still, the performance of the ResNet backbones far surpassed the DCT baseline on both scale tasks, suggesting that CNNs have stronger cues for object scale than spatial frequency.

This result suggests another source of scale information may appear in the higher-level representations of networks. With the knowledge that activations late in CNNs (e.g., at the last layer) map neatly to class labels [111], it is plausible that high-level features map coarsely to scale as well (e.g., objects that are large on average or small on average). However, we attempt to avoid trivial cues for scale via a very simple ranking model (Figure 4) and by applying average pooling to the activations before ranking.

Across some tasks, we observe that the ranking using the ResNet-18 backbone sometimes outperforms the ResNet-50 backbone. We suspect that this is due to the large increase in the number of input dimensions to the ranking model when ResNet-50 is used (due to the increase in total number of channels), and regularizing the weights of the ranking model could yield improved performance. The heavy overfitting of the passthrough baseline can likely be attributed to reliance on absolute position (no average pooling is used) that is not generalizable to the validation set. An alternative hypothesis is that ResNet-50 yields lower ranking performance because it more successfully normalizes away perturbations caused by augmentations. This hypothesis is interesting as it suggests that models with stronger performance may do a better job of eliminating differences created by data augmentations.

Hue appears to be the least favorable task for the ResNet backbones (relative to the baselines). We suspect that this may be due to the narrow range of hue considered, or the difficultly in assessing the absolute delta in hue from the original image. We expect the easier task of ranking the raw value of hue rather than the magnitude to be easier. On the opposite end, contrast appears to be the least favorable task for the baselines (relative to the

ResNet backbones), especially of the color augmentations. We expect that this is because contrast describes the image as a whole and consequentially is a higher-level attribute than hue or saturation. Accordingly, contrast depends more on later layers of the backbones than the other color transformations (Figure 5).

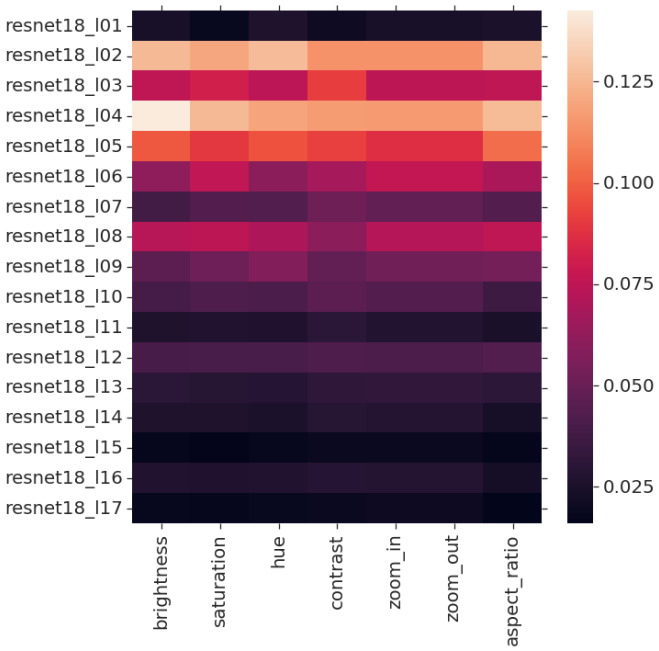
We find that the baseline backbones achieve their highest performance on the color tasks. This is relatively unsurprising, as some color attributes (such as saturation) may be discernible by the raw values of the input color channels. More surprisingly, however, was that while the early layers were favored especially for the color-focused transformations, the most highly weighted layer was not the stem of the ResNet, models but rather a few layers later.

Which layers encode the augmentations?

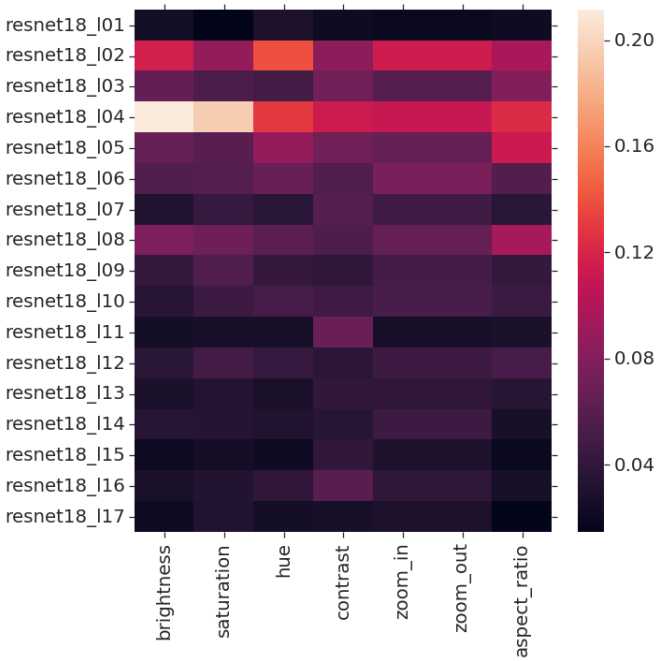
Figure 5a and Figure 5b show the relative importance of ResNet-18 layers for the ranking tasks when taking the mean and max across the channels respectively. A general trend is that the earlier layers are weighted more highly for all of the ranking tasks. Interestingly, this trend occurs even when taking the max across channels despite the later layers having more channels than the early layers.

Another difference is that slightly deeper layers appear more important (or alternatively, early layers are less important) for contrast, aspect ratio and scale (zoom in and zoom out). This pattern may be the result of contrast, scale, and aspect ratio being a higher-level attribute than brightness and saturation. We see a similar trend for the mean (Figure 5c) and max (Figure 5d) of feature importance across channels for ResNet-50. For the aspect ratio and zoom in tasks, the most highly weighted layer (when taking the max across channels) occurs later in the model. In both ResNet-18 and ResNet-50, shortcut layers seem to be neglected by the ranking models. In ResNet-50, however, the later layers appear to be more highly utilized (especially when taking the maximum across channels) than in ResNet-18 though this effect might be accounted for by ResNet-50's greater number of channels increasing the chances that some channel in a layer may be weighted highly.

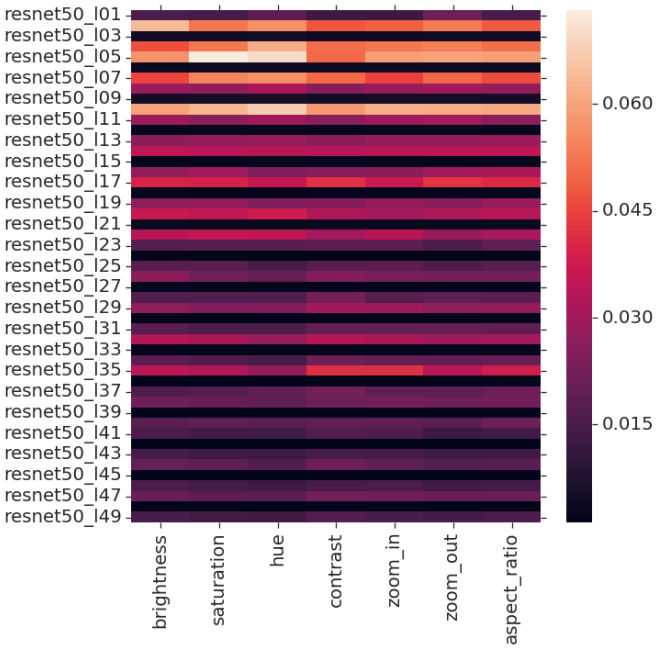
To further validate the trend of early layers more strongly encoding augmentation attributes, we rerun a selection of experiments, using activations from only a few ResNet-18 layers at a time. If the early layers are more relevant for capturing or encoding augmentation attributes, then we should observe a drop in accuracy when using activations from later layers. Indeed, Table 2 shows this drop, suggesting that even if neural networks


resnet18_l01
resnet18_l02
resnet18_l03
resnet18_l04
resnet18_l05
resnet18_l06
resnet18_l07
resnet18_l08
resnet18_l09
resnet18_l10
resnet18_l11
resnet18_l12
resnet18_l13
resnet18_l14
resnet18_l15
resnet18_l16
resnet18_l17
brightness
saturation
hue
contrast
zoom_in
zoom_out
aspect_ratio
0.125
0.100
0.075
0.050
0.025

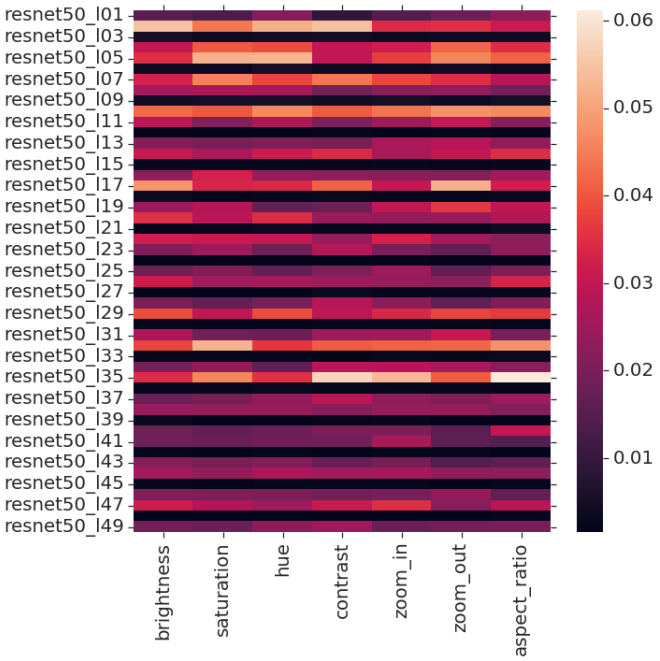
(a)


resnet18_l01
resnet18_l02
resnet18_l03
resnet18_l04
resnet18_l05
resnet18_l06
resnet18_l07
resnet18_l08
resnet18_l09
resnet18_l10
resnet18_l11
resnet18_l12
resnet18_l13
resnet18_l14
resnet18_l15
resnet18_l16
resnet18_l17
brightness
saturation
hue
contrast
zoom_in
zoom_out
aspect_ratio
0.20
0.16
0.12
0.08
0.04

(b)



(c)



(d)

Figure 5: Weightings of activations for ranking tasks with a ResNet-18 backbone (a, b) and ResNet-50 backbone (c, d), with the sum of each task normalized to 1.0. Ranking tasks are ordered from left to right roughly from low-level (color perturbations) to high-level (scale and aspect ratio). Early layers are more important for lower-level ranking tasks, such as color attributes. Color represents mean (a, c) and max (b, d) value across channels.

encode augmentations, this signal begins to be normalized away in later layers, a trend we discuss further in section 3.6.

DISCUSSION

SPECIALIZATION VS. NORMALIZATION For augmentations that are encoded or captured by CNN activations, we ask *where* or at what depth? We describe this question as the specialization vs. normalization question: we posit that data augmentations that are encoded by earlier layers are *normalized* away by the model, whereas attributes that are encoded in later layers incur *specialization*. Intuitively, if a model captures augmentation attributes in early layers but discards this information by the later layers, it has normalized away the augmentation. However, if a model retains augmentation differences in later layers, the intuition is that this augmentation incurs specialization in the same way that the last layer is specialized at a per-class granularity.

The importance of activations from earlier layers relative to those from later layers for our ranking objectives suggests that attributes such as scale are normalized away by CNNs. This phenomenon appears more desirable than the alternative where augmentation attributes are encoded and preserved throughout the model, indicating limited generalization at the output. The lower ranking accuracy when using a ResNet-50 backbone (vs. ResNet-18) may indicate that more accurate models do a better job of normalizing away augmentations.

AN ADVERSARIAL "RANKING MODEL" An alternative we considered was a GAN that proposes augmented images that attempt to fool the backbone model, taking activations of a pre-trained backbone as input. However, a difficulty of this approach is that some popular augmentations (scale transformations) are not easily expressible using standard vision operators or are not differentiable. Still, we see adversarial augmentations as an important related problem: what augmentations are the most difficult for current models?

CAN RANKING OBJECTIVES BE USED AS PRE-TRAINING TASKS? That neural networks appear to encode data augmentation transformation attributes raises the question of whether these attributes are inherently useful for vision tasks. If it is useful for neural network models to encode these attributes, would a source of accurate scale, aspect ratio, or color information improve their performance? Figure 6 shows the results of an experiment where a backbone is pre-trained without class labels via the downstream ranking task (aspect ratio). We find that pre-training to rank

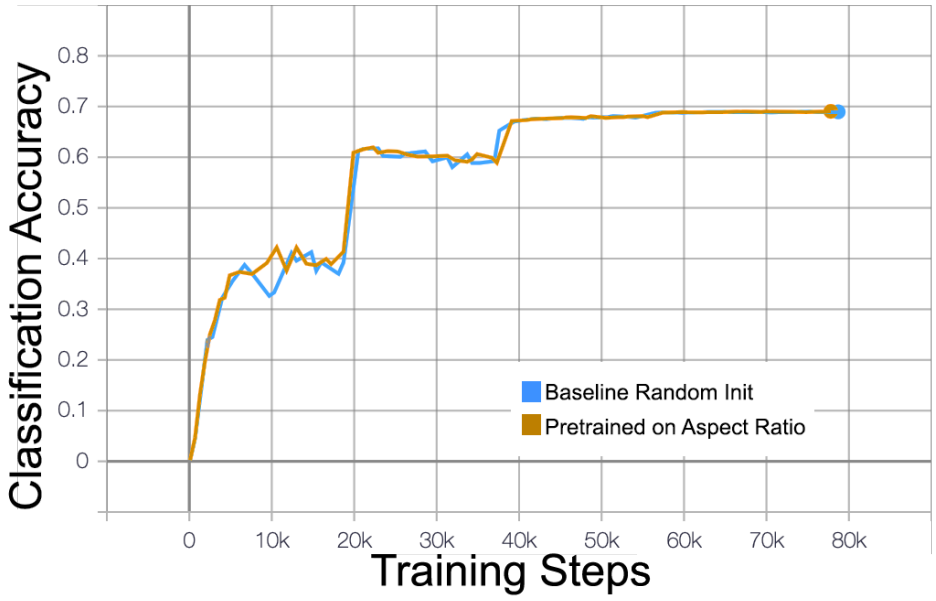


Figure 6: ImageNet classification accuracy vs. training steps of a from-scratch model compared to that of a backbone model pre-trained on the aspect ratio ranking task. Classification performance does not improve, suggesting that encoding augmentations is not inherently desirable.

augmentations does not improve classification performance, with no improvement over training from scratch. The lack of improvement seems to support the hypothesis that the ability to encode augmentations is not inherently desirable, and that normalization is the desired effect.

MODEL SELECTION AND DESIGN In using a simple linear layer to build our ranking model, we sacrifice model performance for interpretability. It may be entirely possible that with sufficient representation power in the ranking model, data augmentation transformations can be recovered with high accuracy using only deep network layers. Still, we believe that using a linear ranking model reveals that augmentation transformations are prominent in neural network features in early layers.

	Zoom In-Train	Zoom In-Val
Passthrough	97.7	46.4
DCT	56.8	46.6
ResNet-18	93.9	**90.1**
ResNet-50	90.8	84.9
	Zoom Out-Train	Zoom Out-Val
Passthrough	98.5	51.8
DCT	57.5	52.4
ResNet-18	82.4	**68.8**
ResNet-50	77.6	64.8
	Aspect Ratio-Train	Aspect Ratio-Val
Passthrough	98.7	54.9
DCT	54.1	57.7
ResNet-18	87.6	80.9
ResNet-50	85.9	**81.3**
	Hue-Train	Hue-Val
Passthrough	94.0	65.0
ResNet-18	87.6	**71.6**
ResNet-50	84.0	66.0
	Saturation-Train	Saturation-Val
Passthrough	97.5	98.9
ResNet-18	97.5	98.3
ResNet-50	95.2	94.0
	Contrast-Train	Contrast-Val
Passthrough	100.0	62.0
ResNet-18	100.0	**100.0**
ResNet-50	99.7	99.7
	Brightness-Train	Brightness-Val
Passthrough	100.0	100.0
ResNet-18	100.0	100.0
ResNet-50	99.3	98.8

Table 1: Accuracies for ranking models that use the baselines and ResNet backbones across the ranking tasks. ResNet features encode many augmentation attributes to a high degree of accuracy, particularly high-level ones such as scale and aspect ratio. ResNet features also beat the baselines on contrast by a wide margin. The accuracy of the ranking model can be used as a proxy to determine to what degree an augmentation attribute is encoded in the CNNs.

	Zoom In-Train	Zoom In-Val
ResNet-18 Block 1	95.5	92.2
ResNet-18 Block 2	95.8	**92.8**
ResNet-18 Block 3	93.7	89.1
ResNet-18 Block 4	93.2	90.0
ResNet-18 Block 5	90.0	85.1
ResNet-18 Block 6	87.5	82.9
	Aspect Ratio-Train	Aspect Ratio-Val
ResNet-18 Block 1	75.7	78.7
ResNet-18 Block 2	87.6	86.3
ResNet-18 Block 3	86.5	**88.8**
ResNet-18 Block 4	87.7	87.2
ResNet-18 Block 5	80.1	79.1
ResNet-18 Block 6	66.7	62.8
	Hue-Train	Hue-Val
ResNet-18 Block 1	75.1	**77.5**
ResNet-18 Block 2	77.6	76.6
ResNet-18 Block 3	77.8	73.0
ResNet-18 Block 4	81.0	72.5
ResNet-18 Block 5	82.5	67.5
ResNet-18 Block 6	82.1	65.8

Table 2: Ranking accuracy when only using features from a block of ResNet-18, ordered from early to later layers. The early blocks yield higher accuracy, indicative of early layers more strongly encoding augmentation attributes.

4

LEVERAGING IMAGE RESOLUTION FOR EFFICIENT STORAGE

INTRODUCTION

Images are ubiquitous on the modern web. With the rapid expansion of social media services, the largest social media networks now host billions of images [36]. At the same time, neural network datasets are rapidly growing in scale, with the 1.2 million image ImageNet dataset [75] being eclipsed by JFT-300M [87].

Image hosts face the challenge of handling the massive rates at which users upload images, especially as scaling of cost per gigabyte slows [31]. This issue is compounded by the need to store each image at multiple resolutions to support different contexts or devices. In 2010, Facebook stored up to 4 different versions of each image [8], later reporting that *dynamic resizing* was also performed [36]. *dynamic resizing* saves capacity by generating low resolution copies of images on the fly without committing them to storage. Faced with a similar problem, Flickr [1] switched to dynamic resizing and reported that doing so helped to eliminate the need for storage capacity upgrades for an entire year.

Image resolution is also a hyperparameter for neural network training and inference. While neural networks are typically trained at a fixed resolution, the use of global average pooling in modern architectures makes them resolution-agnostic from a shape-correctness perspective. Enabling flexibility of resolution at inference time can dedicate more computation to difficult images that require more detail (or as we will later see, choose the best object scale for inference). However, this again requires loading different resolutions of images for inference, analogous to serving different resolution versions of images to users in different contexts.

While dynamic resizing is an attractive method for reducing storage overheads, it introduces two main trade-offs. First, computation is traded for capacity: when an uncached image is requested, the image must be decoded and resized. Second and perhaps more importantly, bandwidth is traded for capacity: reading the entire source image for resizing can waste

bandwidth. Bandwidth can be precious in cold storage scenarios that sacrifice performance for cost and density [9] or when an access misses in the cache.

We propose repurposing *progressive JPEG* to reduce both read bandwidth and storage overheads. The progressive JPEG standard specifies a variant of JPEG images originally designed for bandwidth-constrained networks. In a progressive JPEG image, image data is partitioned and arranged by frequency content instead of by vertical position in an image (scanline), allowing for a lossy preview before the entire image has been downloaded. We demonstrate that by repurposing progressive JPEG, a significant portion of read bandwidth can be saved by reading only the necessary image data for resizing. Additionally, we show that tuning encode-time parameters to match predefined image sizes can further reduce read bandwidth.

Finally, we characterize the cost of decoding custom progressive JPEG directly on the client relative to decoding resized baseline images. We find that the computation–bandwidth trade-off favors transcoding images on the server side, where the computational costs are comparable to a baseline dynamic resizing scheme.

BACKGROUND: PROGRESSIVE JPEG

The progressive JPEG standard was originally designed to allow partially transmitted images to be previewed [95]. Progressive JPEG works by exploiting the fact that partitioning image data in the frequency domain from low to high frequency roughly corresponds to partitioning image data from coarse to fine details. By initially decoding only low frequency data, a preview can be rendered with an incomplete image file.

As with baseline JPEG images, progressive JPEG encoding involves transforming image data to the frequency domain with a discrete cosine transform (DCT). In the frequency domain, intensity values become frequency coefficients; in the case of JPEG, there are 64 coefficients for each 8×8 pixel region. Progressive JPEG partitions frequency coefficients into groups called scans. Figure 7 shows a sketch of the progressive JPEG encode process and partitioning of scans. A single scan can contain a single coefficient, an approximation of a single coefficient, multiple coefficients, or approximations of multiple coefficients; the fundamental property is that scans contain refinements of image data. Only the first scan is necessary to display a low quality image preview.

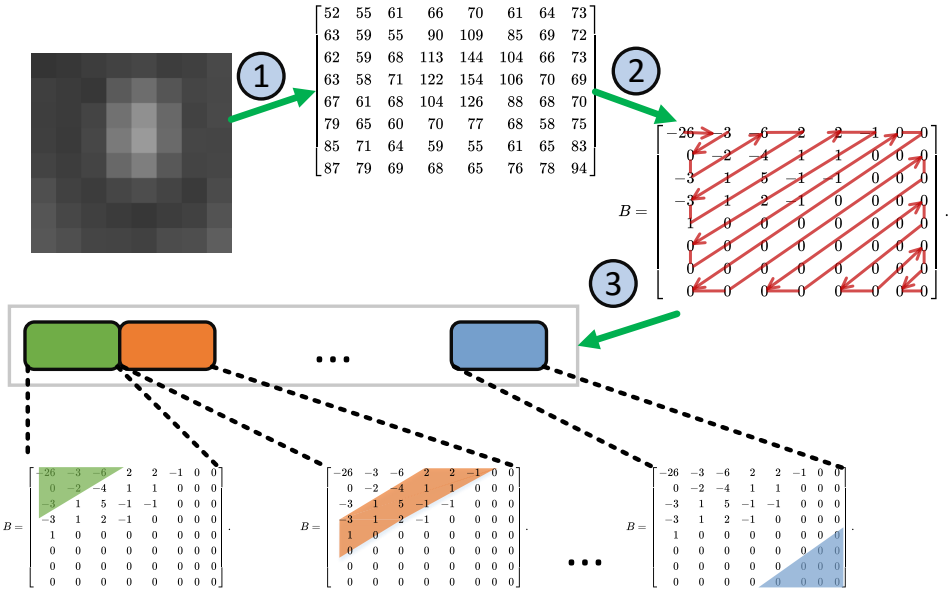


Figure 7: Sketch of Progressive JPEG Encoding: 1. Images are divided into 8×8 macroblocks. 2. Intensity values are transformed to the frequency-domain using a DCT. Red arrows indicate the low to high frequency order of coefficients. 3. Highlighted regions denote scans.

APPROACH

In order to resize a baseline JPEG image to a reduced resolution, the full image must be read. We repurpose progressive JPEG, reading only the scans necessary for a specific image quality for the resized image. To further reduce the amount of data that must be read, we tune progressive JPEG parameters to match predefined image resolutions. We specify resolutions relative to source images (e.g. 10% of a 500×500 image is a 50×50 image).

Defining Image Quality

Using progressive JPEG and dynamic resizing in place of static baseline images requires an image quality metric and quality threshold to determine when have we read enough image data. For each resolution, we define image quality using the peak signal-to-noise ratio (PSNR). To compute the PSNR, the reduced resolution image (which may be lossier) is compared against a source image scaled to the same resolution. For our experiments, we choose a PSNR threshold of 32 dB as the cutoff where no additional image data (or scans) of a progressive JPEG image needs to be read. Our technique of customizing progressive JPEG is orthogonal to the choice of

quality metric and threshold, but higher quality thresholds will reduce savings.

Tuning Progressive JPEG Encoding

We used the `jpegtran` [39] transcoder, which allows the groupings of frequency coefficients and their successive approximations in scans to be customized. We implement a greedy algorithm that enumerates groupings of coefficients (scan configurations) and chooses a configuration based on the resulting PSNR value. Configurations are enumerated by adding coefficient approximations until the PSNR target is met; this process is repeated for all predefined resolutions.

The algorithm is characterized by the following pseudocode which finds the next coefficient approximation to include; some details such as color channels are omitted.

```
best_psnr = 0;
best_coeff = None;
for coeff ∈ (0, max_coeff(config) + c_depth) do
    if approx[coeff] = 0 then
        continue;
    end if
    //approx[] is initialized to a_depth
    temp_approx = approx[coeff] - 1;
    temp_config = config + (coeff, temp_approx);
    psnr = calc_psnr(source_image, temp_config);
    if psnr > best_psnr then
        best_coeff = (coeff, temp_approx);
        best_psnr = psnr;
    end if
end for
return best_coeff;
```

We tune the *coefficient depth* (`c_depth`) parameter used by the greedy algorithm, as it can reduce the search time by pruning the enumerated configurations. We find that reducing this parameter leads to more space-efficient configurations, perhaps by pruning configurations that are locally optimal (in terms of PSNR) but inefficient.

An artifact of the `jpegtran` encoder is that it is limited to at most 100 scans in a given image. This limit also effectively constrains the maximum *approximation depth* (`a_depth`) parameter for images where more scans are needed for approximation refinements. However, to our benefit, the encoder also supports specifying multiple frequency coefficients (within the same color channel) that share the same approximation level in a single scan. This feature allows us to work around the 100 scan limit in many cases; we implement a simple algorithm that identifies the longest inter-

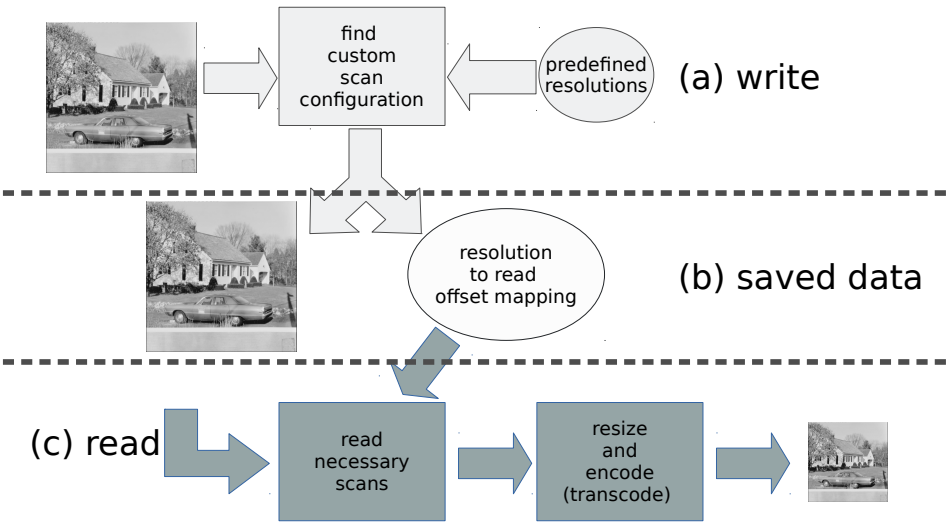


Figure 8: Sketch of a dynamic resizing scheme using custom progressive JPEG. Given an input image and predefined target resolutions (a), a suitable scan configuration is found. This process produces a mapping (b) of resolutions to scans (file offsets in bytes). Given a requested image and resolution (c), the necessary scans are read and the image transcoded.

vals of coefficients that share approximation levels and merges these coefficients into single scans. "Merging" can also be done with the first (DC) coefficients across channels. Merging allows us to encode images using configurations that would otherwise exceed the 100 scan limit of the encoder. Still, when this limit is exceeded, we reduce the maximum *approximation depth* used by the greedy algorithm.

Proposed Read-Write Process

We envision a read/write scheme (Figure 8) where, at write time, files are losslessly transcoded using a custom scan configuration as they are added to the system. At read time, only the necessary scans are read before the resulting image is transcoded to baseline JPEG. The mapping between the requested resolution and how many scans to read (offset within the file) is a result of the custom progressive JPEG configuration process.

EVALUATION

We evaluate the storage overheads (capacity, bandwidth) required for the four storage schemes shown in Table 3. For each approach, we evaluate the

scheme	stored data
baseline (static)	stores source, pre-resized images
baseline (dynamic)	stores source images
progressive (dynamic) (ours, naïve)	stores source images in progressive format
custom progressive (dynamic) (ours, preferred)	stores source images in tuned progressive format

Table 3: Description of each evaluated scheme. The first two schemes represent baselines used by current systems.

storage overheads when three image resolutions in addition to the original may be requested: 10%, 25%, and 50%. For our custom progressive JPEG scheme, we also evaluate the compute overheads relative to dynamic resizing with baseline JPEG images. This comparison attempts to answer the question of whether it is beneficial to offload resizing from the image host to the requesting client—an option not possible with dynamic resizing on baseline images.

Compute Overheads

Many existing JPEG decoders support progressive JPEG and custom progressive JPEG, raising the question of whether progressive JPEG images should be served directly to clients without transcoding to baseline JPEG. However, decoding progressive JPEG images is more computationally expensive than decoding (equivalent) baseline images [57]. We therefore consider the computational overheads of two schemes: (1) the preferred scheme where custom progressive JPEG images are transcoded to baseline images on the server side, and (2) where custom progressive JPEG images are served directly to the client, offloading computation from the server. Offloading transcode (2) is not possible when using baseline images as it would be equivalent to sending the entire source image. For server side transcode (1), we calculate the overhead by measuring the time to decode a custom progressive JPEG image versus a full baseline source image for resizing. For client side decode (2), we calculate overhead by measuring the time it takes to decode a custom progressive JPEG versus a previously resized baseline image.

Dataset and Encoder

We perform our evaluation with the MIRFLICKR [37] dataset, using 24,988 JPEG images with an average resolution of 462×399. We use the original images as the source baseline JPEG images and the ImageMagick `con-`

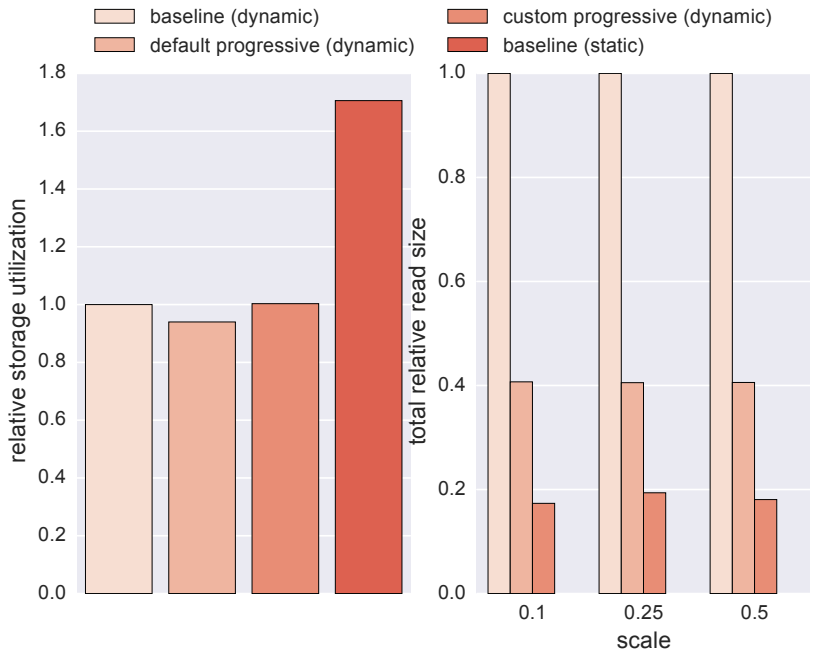


Figure 9: Storage utilization (left) and read sizes measured by the amount of data read to achieve a PSNR of at least 32 dB (right). Note that the PSNR of the resulting images with each scheme can be different despite this lower-bound: default progressive JPEG overshoots the quality target. Overall, dynamic resizing schemes provide similar and substantial storage savings over static resizing. Custom progressive JPEG provides the most bandwidth savings (up to 5.8× vs. baseline).

`vert` [38] tool to generate resized baseline images. For progressive JPEG images, we use `jpegtran` with the `-optimize` and `-progressive` flags. For progressive JPEG images with custom scan configurations, we use `jpegtran` with the `-progressive` and `-scans [file]` flags. In all cases, `jpegtran` performs transcoding losslessly. We also iteratively reduce the quality level parameter until the PSNR drops below 32 dB to avoid inflating the capacity usage of static baseline images. Still, the quality level of the resized baseline images is not strictly equivalent; we compute PSNR on progressive images before they have been re-encoded to baseline images. For many (static baseline) images resized to 10% at a quality setting of 100, we observed that the PSNR was below 32 dB despite acceptable visual quality.

Overall, we find that dynamic resizing dramatically reduces storage overheads significantly (by 41%). Additionally, using custom progressive JPEG for dynamic resizing yields the most efficient use of storage bandwidth.

Storage Capacity

Unsurprisingly, storing baseline images along with resized images uses the most storage capacity (Figure 9). Progressive JPEG is slightly more space-efficient than baseline JPEG [86], though all dynamic resizing approaches are similar in storage utilization. Normalized to dynamic baseline JPEG, dynamic custom progressive JPEG incurs 0.3% storage overhead while dynamic progressive JPEG provides 6.0% storage savings. Custom progressive JPEG likely suffers the small additional overhead due to the increased number of scans.

Read Bandwidth

We consider the case where the requested resolutions of images are not cached[1]. We estimate the read bandwidth requirements of each method using the amount of data read necessary to achieve a satisfactory PSNR for all 24,988 images. Here, baseline pre-resized images are omitted because their PSNR values were not comparable; pre-resized images should offer competitive if not better bandwidth savings relative to custom progressive JPEG. A substantial portion of read bandwidth can be saved just by using progressive JPEG for dynamic resizing: 59% for 10% resolution, with similar improvements for other scales. Customizing progressive JPEG improves savings to 83% for the 10% resolution case. The savings in read bandwidth (Figure 9) from custom progressive JPEG can largely be explained as a PSNR–read size trade-off. This trade-off is evident for default progressive JPEG, which overshoots the quality target (reading enough scans to meet the quality target results in an average PSNR of around 37-38 dB). Interestingly, the progressive schemes seem to require roughly the same amount of read data for all three image scales; this may be a limitation of using PSNR to define image quality.

1 If the requested resolutions were already cached, we would expect performance to be identical under each scheme.

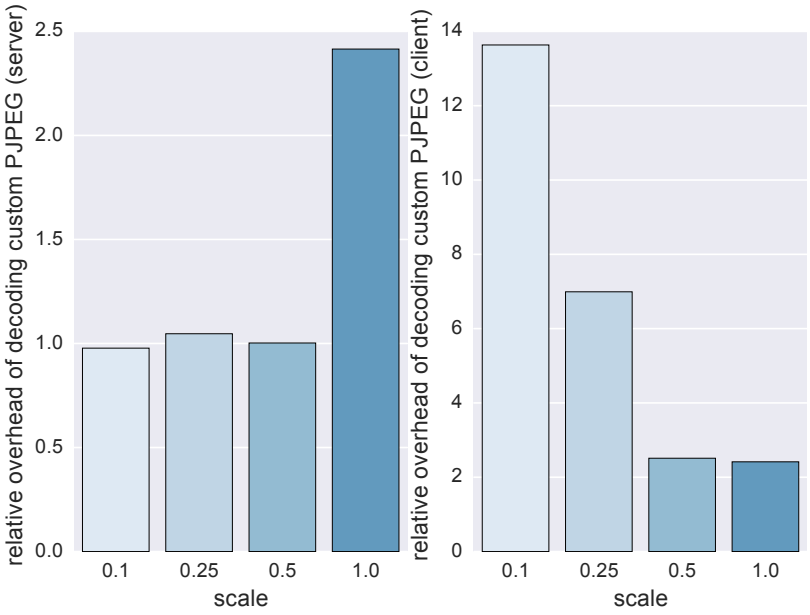


Figure 10: Relative overhead of transcoding on the server (left) and relative overhead of decoding custom progressive JPEG on the client (right).

Compute Overheads

For lower resolutions, the decode overheads (Figure 10) of custom progressive JPEG may be prohibitive (up to 13.6× slower than baseline JPEG) on the client side. However, the computational cost of decoding a baseline source image is comparable (1.0-2.4×) to that of decoding a custom progressive JPEG image. Given this compute–bandwidth trade-off, it makes more sense to transcode custom progressive JPEG images on the server than to decode custom progressive JPEG on the client.

DISCUSSION

We find that customizing progressive JPEG provides a substantial advantage in terms of read size over default progressive JPEG for our quality target. One caveat is that customizing progressive JPEG relies on trading image quality for read size; there is no inherent improvement to the JPEG standard. Rather, custom progressive JPEG facilitates partitioning images at a fine granularity so that this partitioning matches quality specifications closely. In this sense, default progressive JPEG can be viewed as an lower-bound on the bandwidth savings of custom progressive JPEG: 2.5× at a 37-38 dB threshold. Decoding progressive JPEG images is also more com-

putationally expensive (by up to 13.6×) than decoding their baseline counterparts, enough so that it does not make sense to push decoding to the client. Still, decoding progressive JPEG images partially for transcoding on the server is comparable in terms of compute to decoding full baseline images, so transcoding on the server with custom progressive JPEG remains a reasonable approach.

5

TUNING PIPELINES FOR COMPUTER VISION KERNELS

INTRODUCTION

Machine learning program optimization is an important domain of research and the intersection of disciplines including computer architecture, deep learning, systems, and programming languages. Most broadly, the goal of machine learning program optimization research is to enhance programmer productivity and hardware efficiency for research and production tasks. Beyond program optimization, there has been tremendous progress in automatic optimization techniques to relieve the burden on human engineering effort at all levels of the hardware and software stacks. Recent advances include automatic optimizations at the architecture [62] [116], graph [45], kernel [16], and hardware design levels [52] [53] [66] [73], where performance is quickly becoming competitive with human engineers. In work where the contribution requires human insight into model architectures (e.g., resolution and model width scaling [90] or novel combinations of *operators* (e.g., convolutions with different kernel sizes) [91], it is common to see the last mile of improvement being reached with automated methods such as neural architecture search (NAS). However, in each subdomain of automatic optimization, there is room for standardized environments for evaluation. We focus on the rapidly developing line of work of kernel-level deep learning compilers [100] [15], where automated approaches show promise.

To advance the field, automatic program optimization must support a wide range of models: the commonplace, and the cutting edge across all deep learning domains. However, automatic optimization research depends heavily on performance evaluation on real-world hardware. Realistic benchmarks and systems for evaluation for researchers to perform high fidelity evaluations of proposed ideas and algorithms are crucial. This raises a considerable barrier for researchers wishing to improve automatic optimization: compelling work must be general across many domains of models, yet also demon stably improve performance on real hardware and workloads.

Additionally, optimizing machine learning programs presents a broad and challenging problem domain with many reasonable choices for algorithms and modeling. There has been little standardization of how to faithfully evaluate benchmarks or even which benchmarks are the most meaningful; the current trend has been to evaluate on a bag of popular models and report accuracy and performance numbers yet without any clear-cut method for others to extend (e.g., with new search techniques or algorithms) or *reuse* existing work (e.g., on new models or new hardware devices).

Furthermore, the rapid rate of innovation and daunting scope of cutting-edge machine learning models raises challenges for researchers aiming at building better systems. At the core is the tension between innovation and replication: in order to perform a compelling evaluation of their work, researchers must typically replicate the entire end-to-end functionality of exiting deep learning software stacks. This evaluation requirement comes at the cost of innovation as time must be dedicated to rebuilding portions of existing solutions rather than prototyping new ideas and system designs. Consequently, performance-driven research and current deep learning breakthroughs are commonly months or years out-of-sync. For example, depthwise convolutions were immediately adopted by the research community following the introduction of the MobileNet [35] architecture—yet took considerable time to reach hardware specific vendor libraries.

Finally, deep learning compiler research is a quickly moving subfield of systems research, with novel techniques being proposed at all levels of the stack, ranging from graph level transformations (e.g., operator fusion and more general graph rewriting) [45] [5] and code generation (automatic kernel optimization) to semantic-altering transformations such as quantization [10] [113] [26] and hardware-driven neural architecture search [105]. We aim to standardize a task for the sub-problem of optimizing deep learning programs at the granularity of single operators, or *kernels*.

Researchers stick with known workflows with short evaluation times: defining and optimizing operators for a new hardware device takes time. Alternatively, efficient and flexible code-generation for hardware devices dramatically reduces the iteration time for hardware architects and adoption burden for end-users. Performance-driven deep learning researchers have noticed this limitation [7]. Projects such as TensorFlow XLA and NVIDIA TensorRT are scrambling to cover and support as many workloads demanded by researchers and practitioners as possible, and automated techniques are attractive for such tasks, yielding ripe research opportunities.

The central issue is the absence of a unified *environment* and *dataset* for machine learning program optimization that allows researchers to quickly prototype algorithms in a device-agnostic fashion. We introduce the SeaNet

to provide an easily extensible automatic deep learning program optimization environment, with datasets provided to allow researchers to quickly explore ideas and algorithms.

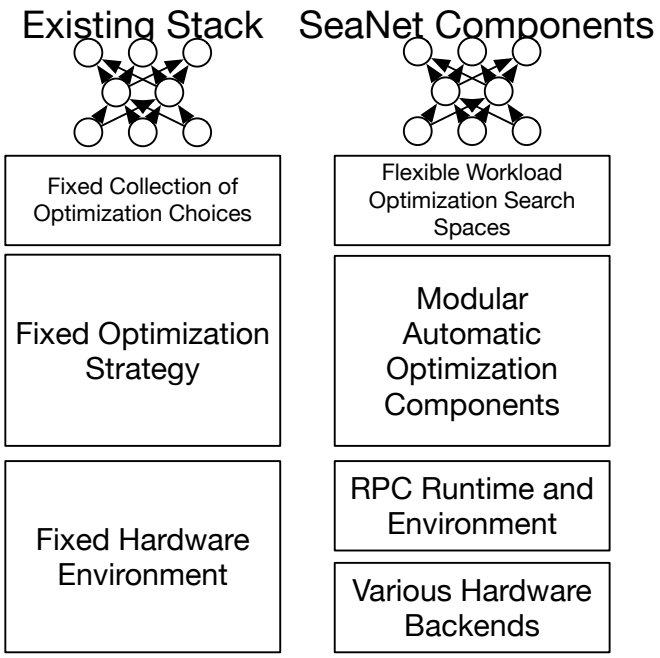


Figure 11: SeaNet provides flexibility and extensibility to existing deep learning program optimization stacks. Researchers can use SeaNet to experiment with novel automatic optimization components for off-the-self of bespoke models. Hardware engineers and researchers can propose new search spaces of optimizations for deep learning operators. SeaNet can be used to build collections of optimized kernels for low-level system libraries (e.g., cuDNN, MIOpen).

SEANET WORKFLOWS AND TASKS

We describe the typical tasks we aim to optimize via SeaNet in this section. Posed as questions, they can be described as: **(1) How can I quickly generate the best code (optimization) for the operators in a deep learning model? (2) How can I predict the peak-performance of a given workload?** Here, workload refers to a deep learning *operator* (e.g., convolution, matrix multiply, or attention) with some instantiated *shape* (e.g., kernel sizes, input sizes, or strides). Concretely, an instance of the first task could be: "optimize all convolution and matrix multiply shapes found in a model such as ResNet-18." Similarly, an instance of the second task could be: "what is the predicted latency of all the convolution and matrix multiply shapes in InceptionV3?"

The first question arises from the challenge of deploying a model to a hardware target. In this setting, a researcher or developer has prepared a deep learning model for some application, and wishes to compile the optimal version for some target hardware (e.g., cloud servers, mobile phones, IoT devices, etc.). Given a single workload out of a collection of workloads presented in a model, generating the optimal code can be approximated by choosing the right configuration for a set of choices in a discrete search space. We describe the search space in more detail in section 5.5. These knobs can be attributes such as the tiling of loops in the workload, the parallelization strategy, memory layout, etc. We expect this task to be driven by statistical methods that rely on *measurements on real hardware* or *simulation* in the optimization loop, as shown in Figure 12. Accordingly, the *cost-modeling* aspect of this task can be modularized and evaluated in isolation using a static dataset, which we provide alongside SeaNet.

The second question arises from the challenge of performance driven neural architecture search. This challenge is an emerging task, with particular relevance in latency-driven settings (e.g., mobile inference workloads) [89] [13]. Whereas classical neural architecture search focuses on the singular objective of optimizing accuracy or loss on some validation set, performance driven neural architecture search aims to balance accuracy with some hardware constraint, such as *latency*. Unfortunately, when the performance of each architectural choice (usually at the granularity of workloads) is no longer easily obtainable via a static library, performance-driven optimization becomes infeasible without rapid performance estimates from another source.

To address this issue, we introduce the task of *peak-performance prediction*. Peak performance prediction takes as input a collection of pre-optimized workloads (e.g., workloads that have already been optimized in accordance with the first task), and the corresponding performance achieved on each workload after tuning. The objective is to accurately predict the performance of a previously *unseen* workload, so that an outer algorithm such as neural architecture search can decide if the workload is worth using or optimizing. Additionally, peak performance prediction is useful even when the collection of fixed workloads to be optimized is fixed: an optimization system can use peak performance prediction to prioritize optimization resources between the different workloads, as workloads that are already at their predicted maximum performance can be deprioritized.

SEANET DEVELOPER API

In this section, we present the SeaNet modular automatic optimization pipeline aimed at tackling the first task of generating the best operator im-

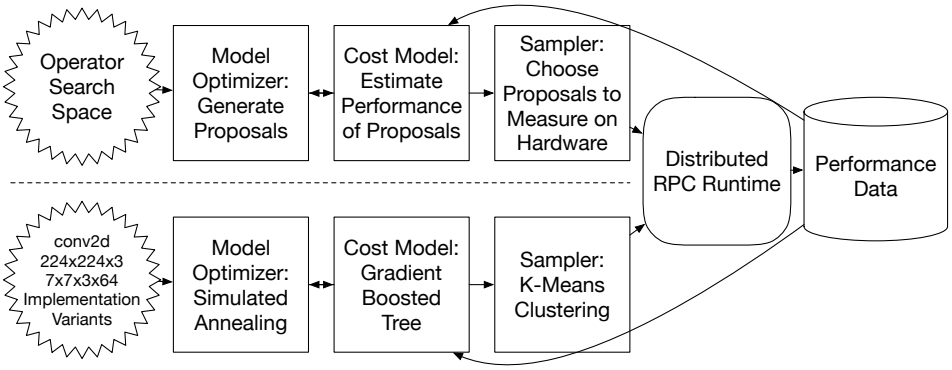


Figure 12: Organization of optimization modules outlined by the SeaNet benchmark task: the proposer (model optimizer), cost model, and sampler. The bottom half denotes an example pipeline instantiated with concrete options for each of the modules.

plementations using as few evaluations of a simulator or real hardware as possible. The pipeline starts from a *search space* of possible implementations for each program, with up to billions of *configurations* (a point in the search space with instantiated values for all tunable implementation parameters). Next in the pipeline is *model optimizer*, which proposes promising configurations to a *cost-model* or value function, which estimates the performance of the proposed configurations. These performance results are filtered by a *sampler* that decides which configurations to evaluate on real hardware. This pipeline is shown with a concrete example in Figure 12. After a round of measurements on hardware, the cost model is updated and the pipeline repeated. One critical feature of the optimization API is that it is device-agnostic, as simulator and performance measurement on hardware devices is handled by the RPC subsystem (section 5.5) of SeaNet.

Each of these modular components can be implemented with algorithms of varying complexity. For example, a model optimizer could simply propose random configurations, use heuristics such as simulated annealing, or contain a machine learning model itself (e.g., in the case of a reinforcement learning agent). Similarly, a cost model may be a gradient tree-boosting model taking in AST-level features as input, or a TreeGRU [88] model operating on program ASTs directly. Sampling algorithms range from greedy samplers (measure the top-k most promising configurations according to the cost model), or more disciplined techniques that attempt to balance exploration and exploitation (e.g., using k-means [4] clustering to blend similar data points together). Finally, a researcher may choose to eschew implementing some or all of the modules in favor of alternative strategies, such as a RL-agent based method. In such a strategy, the model optimizer

simply becomes a wrapper around an RL algorithm and the sampler becomes a passthrough.

Model Optimizer API

As it is often too expensive to exhaustively estimate the performance of all possible configurations of a kernel, an efficient *model optimizer* is crucial in deciding promising candidates. Blackbox proposal strategies include genetic algorithms, random search, and many Bayesian optimization strategies. Similarly, if we consider each configuration to exist in a connected discrete search space, we can view the task as a traversal problem applicable to reinforcement learning and simulated annealing. Here, the objective is to propose the most performance kernel configuration given as few cost model or real-hardware evaluations as possible. For blackbox models, this amounts to the fewest number of iterations—and for the statistical approaches, this amounts to the fewest training examples.

The model optimizers are queried for updated *proposals* or promising configurations to evaluate on real hardware. To implement a model optimizer, the user needs only to implement a single function in the interface, `find_maximums`, which is called to collect a batch of proposals. Intuitively, the `find_maximums` is called to get configurations that maximize the performance returned by the cost model. We show an example of a model optimizer in listing Listing 1.

Cost Model API

While evaluating the performance of proposed kernels is relatively cheap compared to traditional Bayesian optimization settings, hardware resources typically limit the number of feasible experiments (kernels to profile on real hardware) to the order of thousands for practical problems. This limitation means that cost models (either statistical, analytical, or simulated) are often useful for boosting the performance of algorithms that rely on some estimate of performance. One subproblem of optimization can be viewed as building an accurate and efficient cost model—one that faithfully captures the performance of hardware with relatively few samples. Ideally, such a cost-model should also be generalizable across hardware devices to avoid the engineering burden of specializing cost-models for each and every target hardware platform.

A cost model implements a minimum of two functions: `fit`, and `predict`. `fit` takes as input a set of data points (configurations and their corresponding perfomance) in a canonical format, and updates the state of the cost model given the new data. `predict` takes as input a set of configu-

Listing 1: Simple example of a naive random model optimizer. A reference to a cost model is passed as *model*, the number of proposals is specified with *num*, and *excl* contains a set of any points in the search space that should be excluded from consideration.

```
def find_maximums(self, model, num, excl):
    self.visited = set()
    size = len(self.task.config_space)
    for i in range(0, n_iter):
        roll = np.random.randint(0, size)
        if roll in self.visited or\
           roll in excl:
            continue
        else:
            self.visited.add(roll)
            rolls += roll
    picked = rolls
    for i in range(0, self.n_iter):
        self.visited.add(rolls[i])
    scores = model.predict(picked)
    points = sorted(picked, key=lambda item:
        scores[picked.index(item)],
    reverse=True)
    return points[:num]
```

Listing 2: Simplified example of a gradient tree boosting cost model using the XGBoost [14] library. `_get_feature` can be any function that presents an arbitrary feature representation to the cost model. For example, this representation may simply be a concatenation of the choices, an AST representation of the lowered code, or a dataflow graph of operations.

```
def fit(self, xs, ys,...):
    x_train = self._get_feature(xs)
    y_train = np.array(ys)
    y_max = np.max(y_train)
    y_train = y_train / max(y_max, 1e-8)
    index = np.random.permutation(len(x_train))
    dtrain = xgb.DMatrix(x_train[index],
                         y_train[index])
    self._sample_size = len(x_train)
    self.bst = xgb.train(...)
def predict(self, xs,...):
    feas = self._get_feature(xs)
    dtest = xgb.DMatrix(feas)
    return self.bst.predict(dtest)
```

ration points, and outputs their corresponding costs. We show a sample implementation of a cost model in listing Listing 2.

Sampler API

Additionally, reducing the number of proposed configurations to a feasible number that are measured on real hardware is an effective way to speed up the optimization loop. One observation is that implementation search spaces can contain many configurations or programs that are close both in terms of performance and in terms of configuration. Identifying this scenario and pruning configurations that are likely to be similar reduces the amount of hardware time needed to train an accurate cost model or find highly performing examples [4] A sampler takes in a collection of configuration proposals and chooses k proposals to be evaluated (e.g., on real hardware or in simulation). A sampler only needs to implement a `sample` function, which takes the proposals, their corresponding costs (as evaluated by the cost model), and k as input. We show an example of a sampler in listing Listing 3.

Listing 3: Example of a *k*-means clustering based sampler that attempts to achieve sample diversity. Points refers to the current proposed set of configurations, scores are their corresponding scores as given by a cost model, and the plan size is the number of points (configurations) that should be selected for measurement.

```
def sample(self, points, scores, plan_size):
    #convert configuration index to coordinates
    X = [point2knob(point, self.dims)
         for point in points]
    kmeans = cluster.KMeans(plan_size)
    means = kmeans.fit(X).cluster_centers_
    means = np.round(means)
    plan = [int(knob2point(mean, self.dims))
            for mean in means]
    return plan
```

SEANET AS A DATASET

We provide a dataset to quickly evaluate deep learning kernels out of the box in two ways: per-configuration performance for a fixed search space, and peak prediction performance for a given workload (e.g., Conv2D with a certain shape). Users can use data for the first scenario to prototype cost models for Task 1 (introduced alongside Task 2 in section 5.2), and data for the Task 2 scenario to prototype peak performance prediction models. Additionally, users are free to collect their own dataset using in-house hardware devices by leveraging SeaNet's search space definitions for deep learning operators.

The Configuration Search Space

The SeaNet API gives access to a rich collection of kernel implementation search spaces for common deep learning models on a wide range of hardware backends, ranging from ARM and x86 CPUs to mobile OpenCL GPUs and CUDA server-class GPUs. Figure 13 gives a logical overview of an example search space. This search space is built on top of *schedule primitives* [15, 70] that describe possible optimizations to apply to kernels. These search spaces define possible implementations of deep learning kernels for their corresponding hardware devices. The complexity of a search space varies depending on the target hardware device and operator, with the largest search spaces comprising billions of configurations. Typically, we find that search spaces for devices with intricate memory hierarchies (e..g,

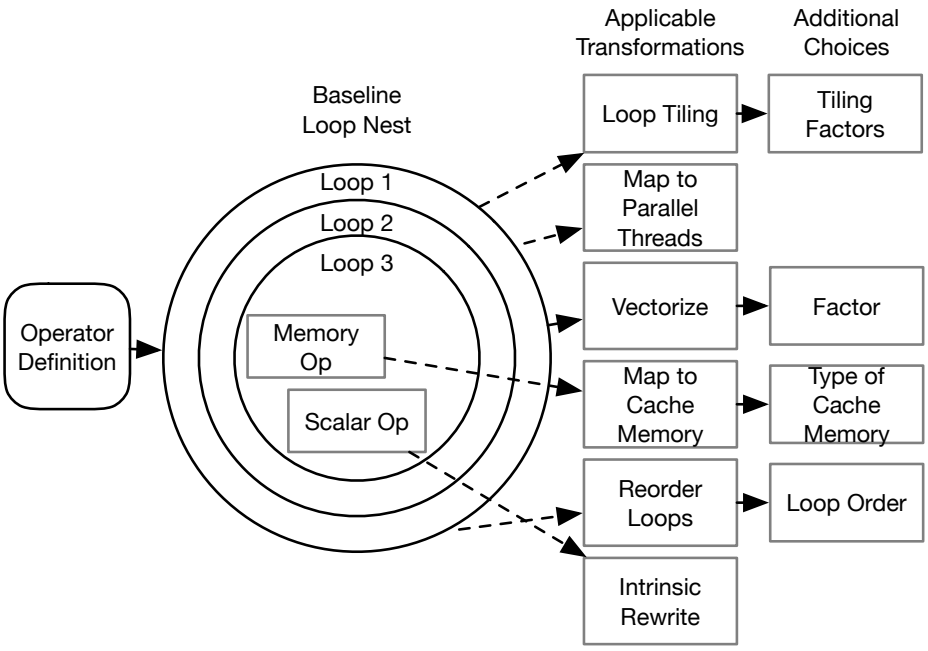


Figure 13: Organization of an operator search space in SeaNet following its operator definition. Deep learning kernels present a loop nest of operations, with various transformations available to different components of the loop nest. Depending on the transformation, additional parameters may need to be decided. Note that the search space definition can be recursive (e.g., if a loop nest is tiled, the resulting loop nest can optionally be further tiled).

GPUs) have more configurations (to cover the wide variety of program implementations) and are more difficult to optimize due to the sharp performance cliffs associated with cache hierarchies.

Standalone Configuration Performance Prediction

We provide several sample datasets spanning thousands of configuration points for rapid prototyping of cost models. These datasets are obtained from randomly sampling the optimization search space of a typical convolution operator (e.g., from a ResNet model) on various hardware devices. These datasets are collected from a range of hardware devices (desktop GPU, mobile CPU, and mobile CPU) and cover a wide range of performance. Concretely, the dataset is a collection of configurations (possible implementations of a given workload), to their measured performance on hardware. Table 4 shows a comparison of baseline cost models on this dataset. Note that a SeaNet user can quickly collect a comparable dataset on other hardware devices they have available by leveraging the RPC in-

1080 Ti (Desktop GPU)	
Cost Model	MAE (GFLOPS)
Linear Regression	52.6
2-MLP	49.6
Gradient Tree Boosting	**44.7**
ARM Cortex A-72 (Mobile CPU)	
Cost Model	MAE (GFLOPS)
Linear Regression	1.40
2-MLP	**1.23**
Gradient Tree Boosting	1.34
Mali T-860 MP4 (Mobile GPU)	
Cost Model	MAE (GFLOPS)
Linear Regression	3.09
2-MLP	3.77
Gradient Tree Boosting	**1.69**

Table 4: Standalone evaluation of various cost model baselines on workloads collected from different hardware devices. MAE refers to the mean absolute error of the regression on a held-out set in billions of floating-point operations per second. Note that the absolute scale difference can be partially attributed to the typical peak performance achievable on each hardware device.

frastructure (section 5.5). Data collection is fast, usually on the order of thousands of configuration points per hour.

Peak Performance Prediction

Additionally, we provide a dataset of pretuned operators for thousands of workloads found in popular deep learning models, corresponding to thousands of hours of machine time. These pretuned operators correspond to specific configurations expected to approximate the peak performance of a given hardware device for these workloads. Predicting peak performance is an important task for areas such as graph optimization and performance driven neural architecture search. In this setting, a model is trained on data that maps a collection of pretuned workloads (based on features such as their shapes, implementation strategy, and the hardware type) to achieved performance. An accurate model is critical when optimization is too expensive to run in the loop of a larger NAS or graph optimization pipeline. Concretely, the peak prediction dataset can be viewed as a mapping of workloads (a given operator with specific semantics such as shape, stride, padding) to latency (execution time).

All of the data collected for the tasks discussed thus far and the environment depends on the SeaNet RPC infrastructure. This section discusses the low-level infrastructure used to collect performance measurements (either for building or a dataset, or in an online experiment) from hardware. Deep learning compiler optimizations are typically highly specialized to leverage domain-specific insights about their workloads. As evaluating compiler optimizations on multiple hardware and software platforms quickly becomes tedious, we provide a generic RPC Infrastructure as part of the SeaNet environment. Crucially, this infrastructure is *transparent* to the researchers developing new optimization algorithms. Under the hood, the RPC system also provides tremendous flexibility as it enables fine-grained control of the device runtime. For example, the RPC system enables us to ensure that idiosyncratic system parameters (e.g., CPU affinity on a big.LITTLE SoC) are correctly configured.

One of the main advantages of having a shared framework for benchmarking optimization algorithms is reproducibility. Modern machine learning pipelines commonly have numerous hyperparameters and subtle implementations differences that often go unreported in published work yet can make profound differences in evaluations results. Due to the systems-performance driven nature of kernel optimization, hyperparameters now threaten reproducibility on two frontiers: experiment configuration and measurement. In addition to experiment hyperparameters, the conditions of performance measurement (e.g., how wall clock execution time is measured) is often an ad-hoc choice left to researchers. Conveniently, sharing a common RPC infrastructure for evaluating optimization algorithms allows the burden of sensible performance measurement guidelines to be collectively addressed: wisdom about the idiosyncrasies of each hardware platform can be shared instead of being independently discovered through each evaluation. Examples of pitfalls that researchers must account for/discover include: JIT kernel compile time (for languages such as CUDA and OpenCL), hardware power states that take time to warm-up/cool down (virtually all modern CPUs and hardware accelerators), and thermal throttling on devices with limited thermal dissipation (e.g., single-board computers and mobile phones). We provide customized RPC *servers* for a variety of hardware platforms to alleviate the burden of device-specific housekeeping on researchers.

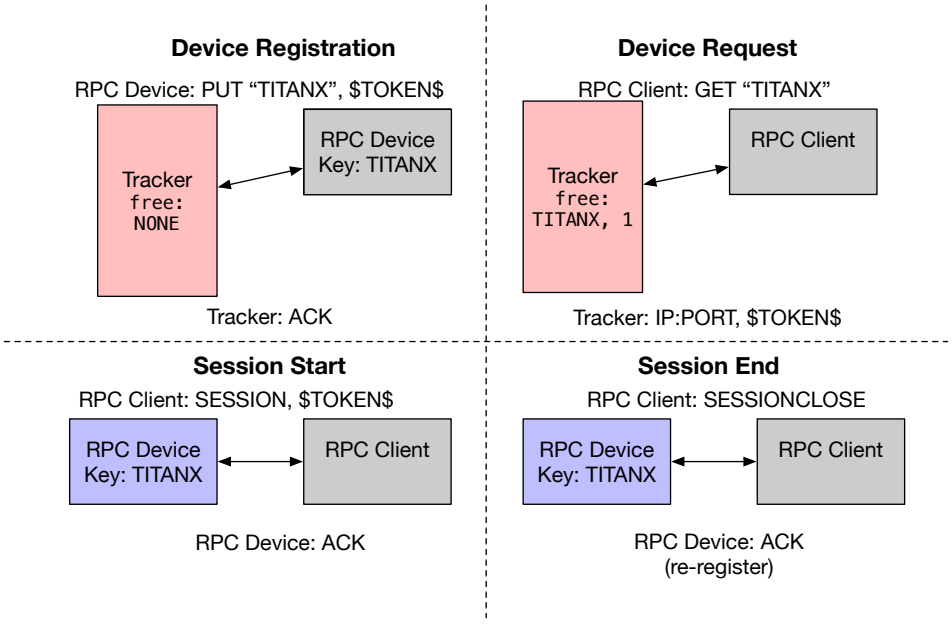


Figure 14: Summary of the Control Plane (RPC Tracker) Protocol. 1. An available device registers itself with the RPC tracker, providing a one-time use “token“ value along with its device type (in this case “TITANX”). 2. An RPC client requests a device of type “TITANX.” The tracker provides the address and one-time use token of an available device. 3. The RPC client connects to the provided device and begins a session. 4. The RPC client terminates its session; the RPC device registers itself as free to the tracker. Note that this logic is typically *internal* to an optimization pipeline and transparent to the user seeking to change optimization algorithms.

RPC Protocol

At a high level, the RPC system is composed of a central tracker which multiplexes RPC servers across many RPC sessions. RPC servers can be individual mobile phones, single-board computers (e.g., Raspberry Pi, RK33999), hardware accelerator boards (e.g., Ultra96), or servers hosting multiple GPUs and CPUs. We describe our RPC protocol in two parts: a *control* plane which handles resource allocation and load balancing across many hardware devices, and a *data* plane which handles communications between hardware devices and clients issuing requests. Together, these components of the protocol enable streamlined prototyping and deployment of neural network implementations.

CONTROL PLANE We provide a tracker (i.e. resource manager) to coordinate the resources provided by a pool of devices. Figure 14 shows a

basic protocol of the resource registration. In our setup, each RPC runtime registers itself with the tracker when it is available for performance measurement/profiling. When a pipeline wishes to profile a configuration, it requests a free RPC session corresponding to its desired device type. The tracker removes the requested device from the pool of free devices, and the devices re-registers itself with the tracker when profiling has been completed. The pool of devices can be multiplexed across many developers or strategies for optimizing deep learning workloads on hardware. Our tracker is inspired by the model used by the cluster management system such as Mesos [33] and Kubernetes [12], with the additional challenge of providing a runtime environment that runs on heterogeneous embedded devices.

DATA PLANE The RPC data plane does not require each user to have an identical development environment—the RPC runtime simply runs the provided code. This flexibility allows different search spaces as well as search strategies (e.g., cost model implementations, optimization algorithms) to share the same pool of devices seamlessly. The data plane of the RPC runtime allows as much compilation to be performed on the client side (or before the RPC boundary) as possible. This reduces the burden on devices that may be a poor fit for heavy-duty compilation (of the programs that they run), such as IoT devices or mobile phones. The RPC runtime enables portability of optimization strategies and computational graph declarations across devices. After developing an optimization strategy for a hardware device target (e.g., Raspberry Pi), a user can quickly evaluate the same optimization strategy on an NVIDIA GPU as both share the same RPC interface.

TIMING MEASUREMENT Designing a method to accurately measure the wall-clock time of deep learning workloads (with a wide range of arithmetic complexity) that generalizes across different types of hardware (with a wide range of performance and environments) is difficult. In order to collect *repeatable* measurements across different devices and workloads, we focus on minimizing transient system effects and dispatch overheads that may inaccurately bias performance measurement. Hardware devices with different power states are an example of platforms where accurate and repeatable timing measurement is tricky. In some cases, a given workload's timing measurement will appear to be faster or slower depending on the device's power state. This behavior can lead to surprising results, such as a very fast kernel appearing to be slower as it finishes *too quickly*, before a hardware device can ramp up to a higher power state. We take a simple approach to address this problem, by allowing the measurement function to specify a minimum repeat time, or the minimum amount of time a loop of

HW	OK	Compile	Timeout/Misc.
gfx900 (AMD GPU)	61	143	84
rk3399 (ARM CPU)	55	86	147
pixel2 (ARM CPU)	89	86	113

Table 5: Breakdown of failures tolerated when randomly traversing the search space of 9 VGG-16 [83] workloads for various hardware targets. 32 configurations were sampled for each workload. Note that "compile" failures include configurations that violate constraints when checking against limits such as scratchpad size, available threads, etc.

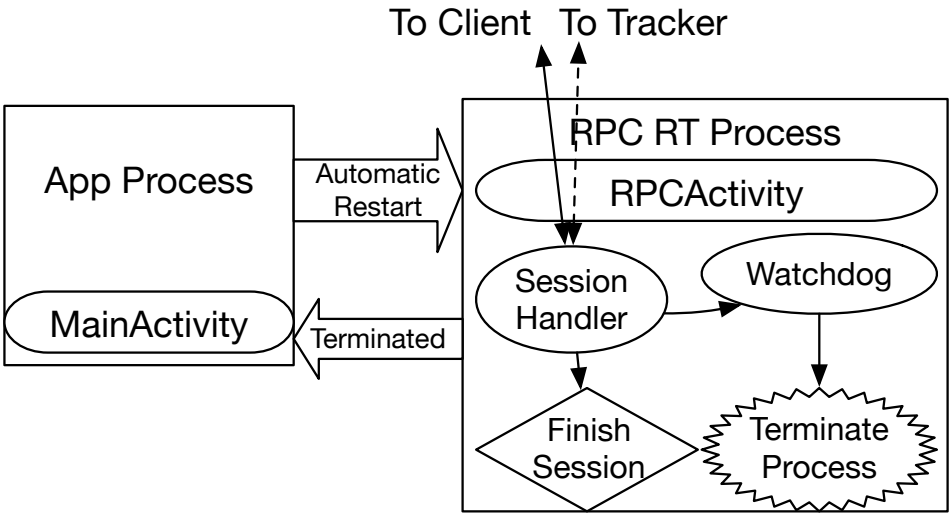


Figure 15: Overview of the Android RPC runtime implementation. A second activity is dedicated to the RPC session in a separate process for fault tolerance, and also to ensure that the RPC session receives high OS priority by keeping its parent process on-screen. RPC session timeouts are enforced by a watchdog which "races" against the session processor. If the watchdog finishes first, the RPC process is terminated and restarted.

measurements must take. If the minimum time is not met, the number of iterations of the measurement loop is increased. Here, the goal is to force hardware devices into the highest sustainable power state and to amortize away/discard runs that executed at a lower power state.

Fault Tolerance

At compile time, given the size of search spaces for each operator and hardware backend, each search space may contain many invalid configurations. Invalid configurations may violate such restrictions such as scratchpad memory sizes on GPUs, maximum thread block/workgroup sizes in

CUDA/OpenCL capable devices, or allocate more registers than the hardware supports. Table 5 shows an example of the distributions of failures encountered in practice. At run time, each measurement forks a new workhorse process so that a buggy configuration that crashes the workhorse does not terminate the RPC server or prevent the device from re-registering with the RPC tracker. Additionally, for devices with more exotic runtime environments, such as Android smartphones, we use an app with a separate watchdog (shown in Figure 15) process which restarts the workhorse RPC runtime if necessary.[1] Finally, at the control plane level, the tracker gracefully switches between devices if a device becomes faulty and fails to respond to incoming requests. This property is essential in ensuring that a few faulty devices do not interfere with long running optimization tasks by polluting the measurement data.

BASELINE IMPLEMENTATIONS AND EVALUATION

To highlight the flexibility of SeaNet across algorithms and hardware devices, we present descriptions of baseline implementations of SeaNet modules and evaluations of their performance starting with the first task, optimizing kernels for a given deep learning model. We visualize the efficiency of different optimization pipelines by observing which pipeline achieves the best performance (of an evaluated program configuration) in the fewest *trials* (measurements on hardware). In our evaluation, we first seek to characterize two main types of optimization pipelines first: blackbox pipelines (e.g., evolutionary algorithms and random search), and cost-model driven baselines. The pipelines evaluated on either a desktop GPU (NVIDIA 1080 Ti) or a Mobile CPU (ARM Cortex-A72) as denoted in their corresponding figure titles. We then explore which sampling and model optimizing strategies are best given a particular cost model for the first task. Following the first task, we visit the second task of predicting peak performance using different regression models.

TASK 1: OPTIMIZATION PIPELINE VARIANTS For the first task, we evaluate the efficiency of a variety of optimization pipelines. At the highest level, these can be divided between *blackbox* approaches that do not attempt to explicitly model the performance of different configurations in a search space, and *cost-model* driven approaches that use a cost-model to estimate the performance of previously unseen configurations. The cost model can any feature representation that can be derived from program configurations as input. For simplicity, we base our features on the concrete values

1 We use this approach as the Android OS allocates CPU time depending on whether your process is responsible for the current on-screen Activity.

of the choices in each configuration space in this evaluation, though in practice more sophisticated approaches (e.g., using AST structures) are feasible. We first compare the performance of two blackbox pipelines an genetic algorithm (GA) and random search with a cost-model driven pipeline that proposes many configurations randomly before filtering them through a cost model and then sampling among a promising subset of the proposed configurations randomly in Figure 16. Here, we see that using a cost model can improve the efficiency of optimization (by using fewer hardware measurements) to reach the same level performance on more difficult workloads, yet all three methods remain comparable on easier workloads. Next, we seek to characterize the difference between cost-model pipelines that use a model optimizer to propose promising configurations. In this setting, we introduce a simulated annealing (SA) model optimizer and an evolutionary algorithm model optimizer (EA) alongside the random model optimizer. We refer to the optimizer as EA while it is logically identical to the blackbox GA, with the difference being that the fitness scores of the GA are obtained through actual hardware measurement (blackbox), whereas the EA queries a cost model for fitness scores. In this comparison (Figure 17), we see that the evolutionary algorithm performs best with random sampling on many of the challenging workloads. Interestingly, for the workload shown, the SA optimizer is the worst, though for many other workloads it is competitive with the EA optimizer. We suspect that in many cases, the optimizers require hyperparameter tuning for ideal efficiency.

We then seek to characterize the differences in the remaining component of the optimization pipeline: the sampler. Here, we provide a k-means and greedy sampler alongside the random sampler which chooses configurations filtered by the cost model randomly. The k-means sampler first clusters points proposed by the model optimizer with k set to the number of measurements to be performed, whereas the greedy sampler simply picks the top-k highest scoring proposals. Here, we find that the k-means sampler tends to do well on workloads where exploring a diverse set of configurations in the search space is beneficial. Finally, we show an example of all pipelines evaluated so far in Figure 19. While the relative efficiency of different pipelines can be tricky to interpret, we see that all pipelines surpass random search, and that evolutionary strategies (including blackbox versions) do well.

TASK 2: PEAK PERFORMANCE PREDICTION Moving to the second task, we seek to characterize the effectiveness of different typical regression models for predicting peak performance. We train a type of model on a subset of pretuned-operator performance data and use it to predict performance on the held-out set. Figure 20 shows these results, compared with linear regression and MLP baselines. With minimal data preprocessing (converting

dataset features to one-hot encodings), we find that peak-performance on a gradient tree boosting model yields promising results. We expect that peak performance prediction can be integrated with performance driven NAS approaches that include with program optimization in the NAS search loop.

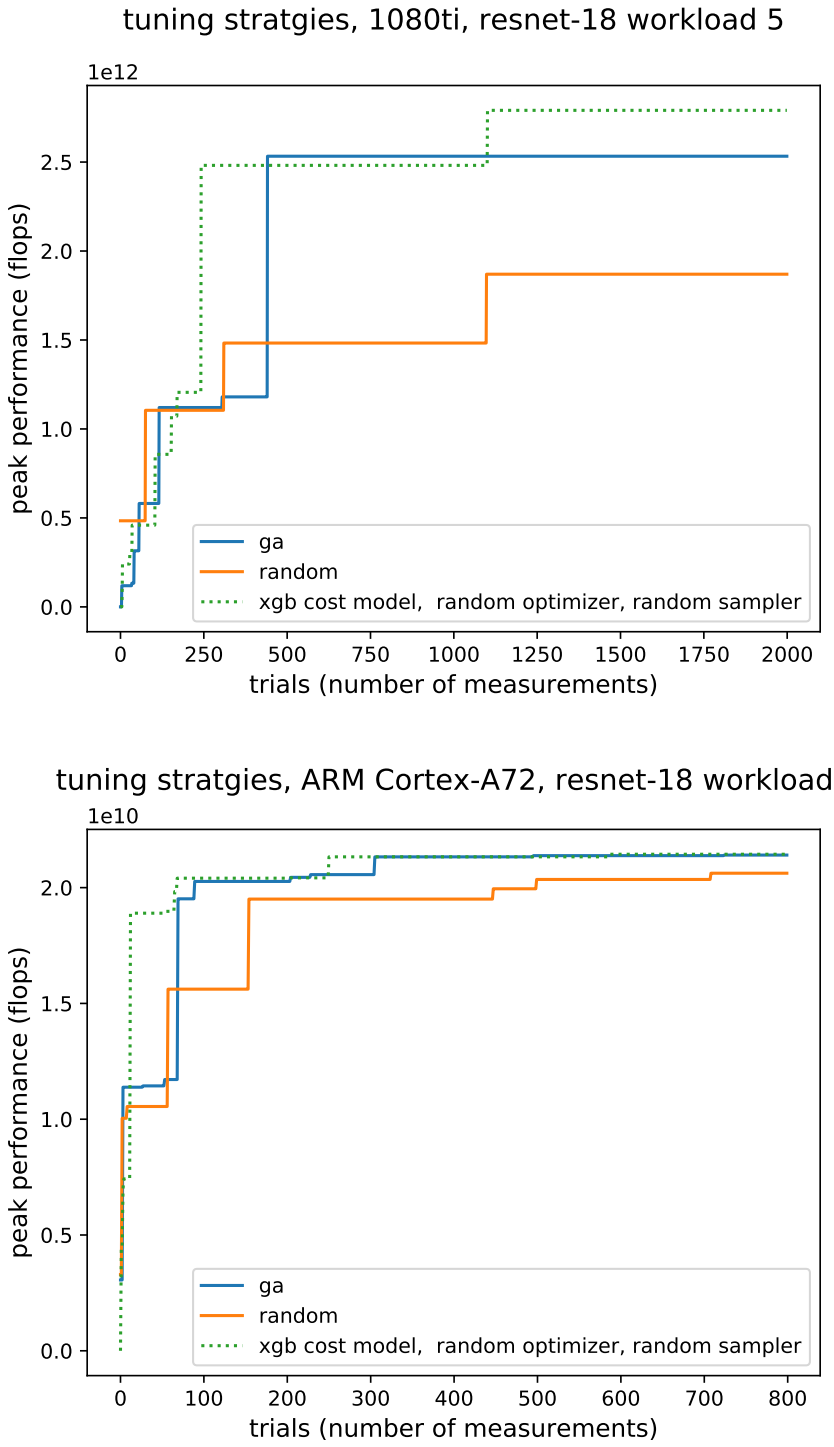


Figure 16: Task 1: Blackbox optimization pipelines compared with cost-model driven pipelines on a convolution kernel from ResNet-18 [32]. Here, the XGBoost cost model is used to filter randomly proposed configurations that are then randomly sampled. The bottom plot shows an example of an "easier" optimization task, where all three approaches quickly achieve good performance.

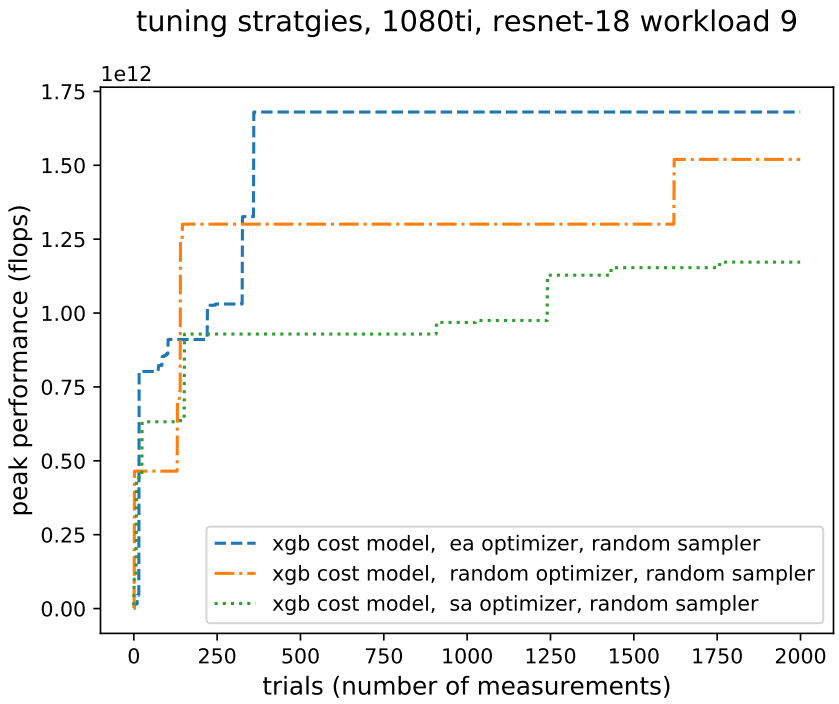


Figure 17: Task 1: Comparison of model optimizers with random sampling. On many workloads, we find that the evolutionary algorithm works the best with random sampling.

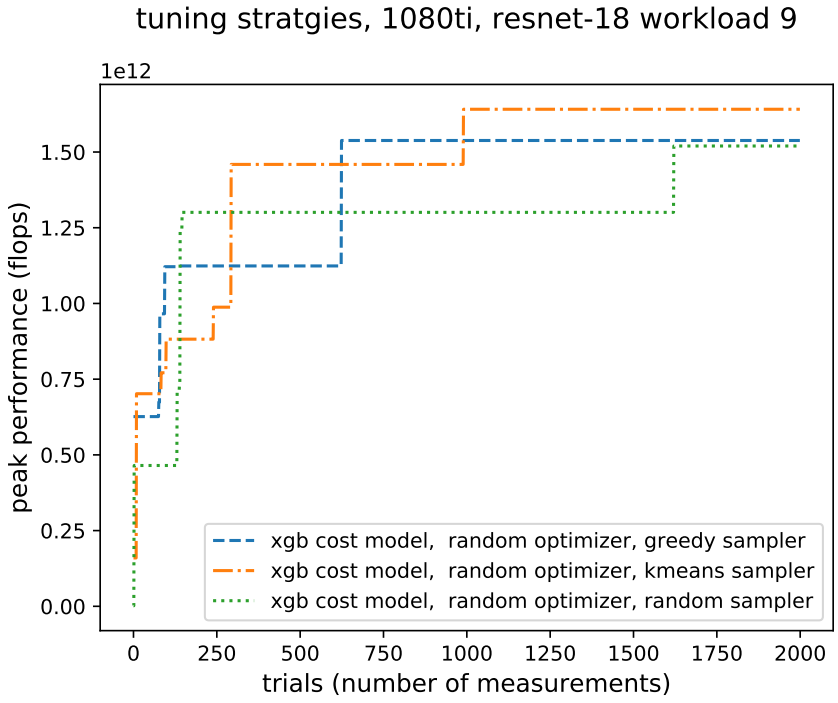


Figure 18: Task 1: Comparison of samplers with random proposals. On many workloads, we find that k-means sampling works the best with random proposals. Intuitively, this makes sense as K-Means sampling achieves the highest sample diversity.

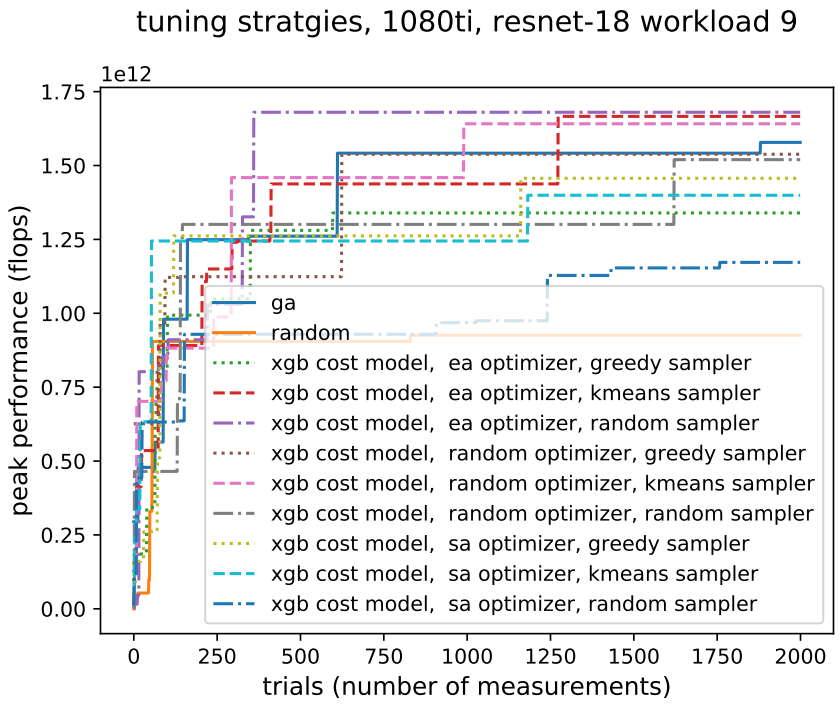
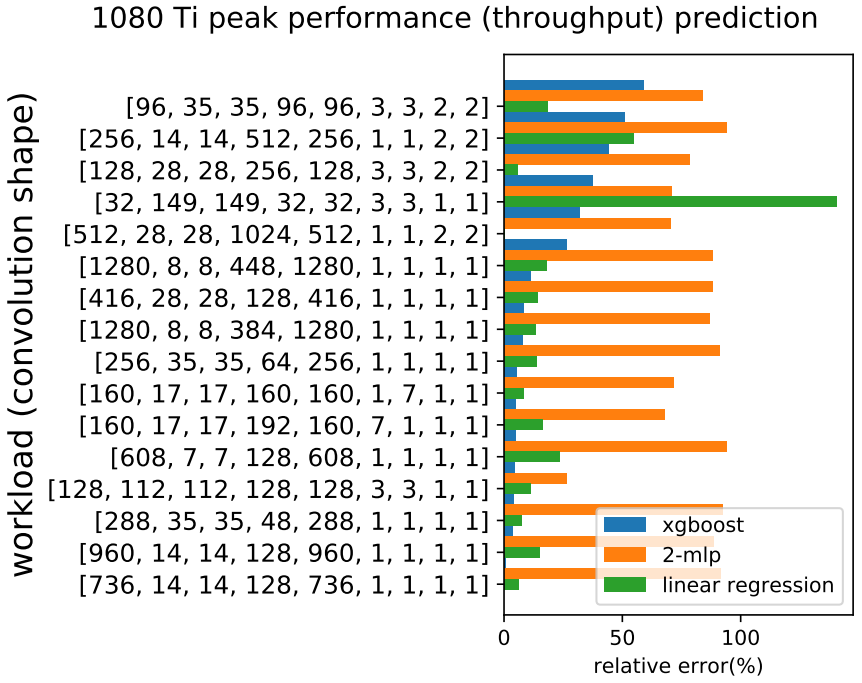
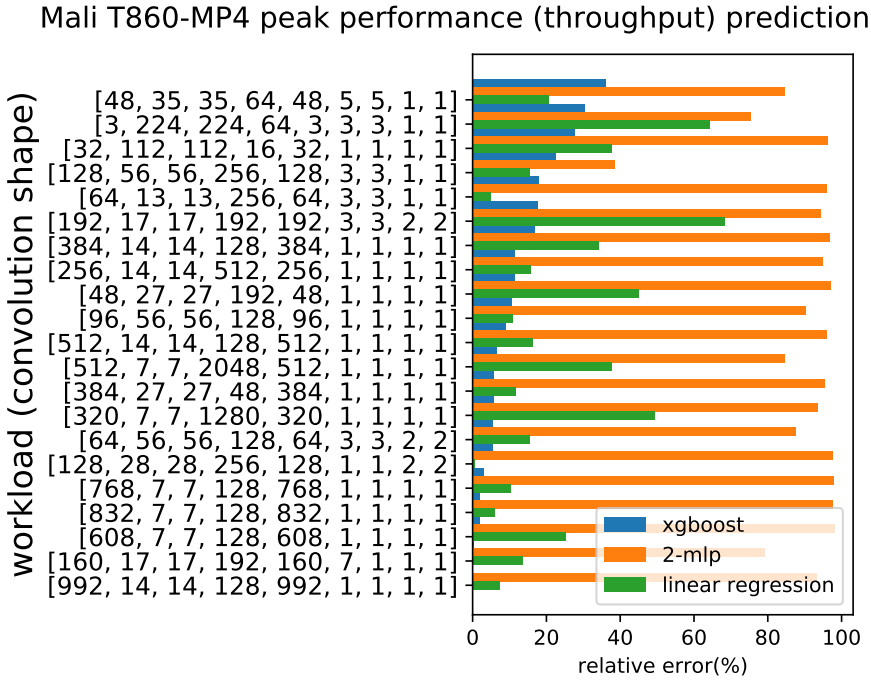


Figure 19: Task 1: Comparison of all pipelines. All methods outperform the random baseline on this workload, with many of the best performing pipelines relying on an evolutionary algorithm (either as the optimizer or in blackbox form).


1080 Ti peak performance (throughput) prediction
workload (convolution shape)
[96, 35, 35, 96, 96, 3, 3, 2, 2]
[256, 14, 14, 512, 256, 1, 1, 2, 2]
[128, 28, 28, 256, 128, 3, 3, 2, 2]
[32, 149, 149, 32, 32, 3, 3, 1, 1]
[512, 28, 28, 1024, 512, 1, 1, 2, 2]
[1280, 8, 8, 448, 1280, 1, 1, 1, 1]
[416, 28, 28, 128, 416, 1, 1, 1, 1]
[1280, 8, 8, 384, 1280, 1, 1, 1, 1]
[256, 35, 35, 64, 256, 1, 1, 1, 1]
[160, 17, 17, 160, 160, 1, 7, 1, 1]
[160, 17, 17, 192, 160, 7, 1, 1, 1]
[608, 7, 7, 128, 608, 1, 1, 1, 1]
[128, 112, 112, 128, 128, 3, 3, 1, 1]
[288, 35, 35, 48, 288, 1, 1, 1, 1]
[960, 14, 14, 128, 960, 1, 1, 1, 1]
[736, 14, 14, 128, 736, 1, 1, 1, 1]
xgboost
2-mlp
linear regression
0
50
100
relative error(%)


Mali T860-MP4 peak performance (throughput) prediction
workload (convolution shape)
[48, 35, 35, 64, 48, 5, 5, 1, 1]
[3, 224, 224, 64, 3, 3, 3, 1, 1]
[32, 112, 112, 16, 32, 1, 1, 1, 1]
[128, 56, 56, 256, 128, 3, 3, 1, 1]
[64, 13, 13, 256, 64, 3, 3, 1, 1]
[192, 17, 17, 192, 192, 3, 3, 2, 2]
[384, 14, 14, 128, 384, 1, 1, 1, 1]
[256, 14, 14, 512, 256, 1, 1, 1, 1]
[48, 27, 27, 192, 48, 1, 1, 1, 1]
[96, 56, 56, 128, 96, 1, 1, 1, 1]
[512, 14, 14, 128, 512, 1, 1, 1, 1]
[512, 7, 7, 2048, 512, 1, 1, 1, 1]
[384, 27, 27, 48, 384, 1, 1, 1, 1]
[320, 7, 7, 1280, 320, 1, 1, 1, 1]
[64, 56, 56, 128, 64, 3, 3, 2, 2]
[128, 28, 28, 256, 128, 1, 1, 2, 2]
[768, 7, 7, 128, 768, 1, 1, 1, 1]
[832, 7, 7, 128, 832, 1, 1, 1, 1]
[608, 7, 7, 128, 608, 1, 1, 1, 1]
[160, 17, 17, 192, 160, 7, 1, 1, 1]
[992, 14, 14, 128, 992, 1, 1, 1, 1]
xgboost
2-mlp
linear regression
0
20
40
60
80
100
relative error(%)

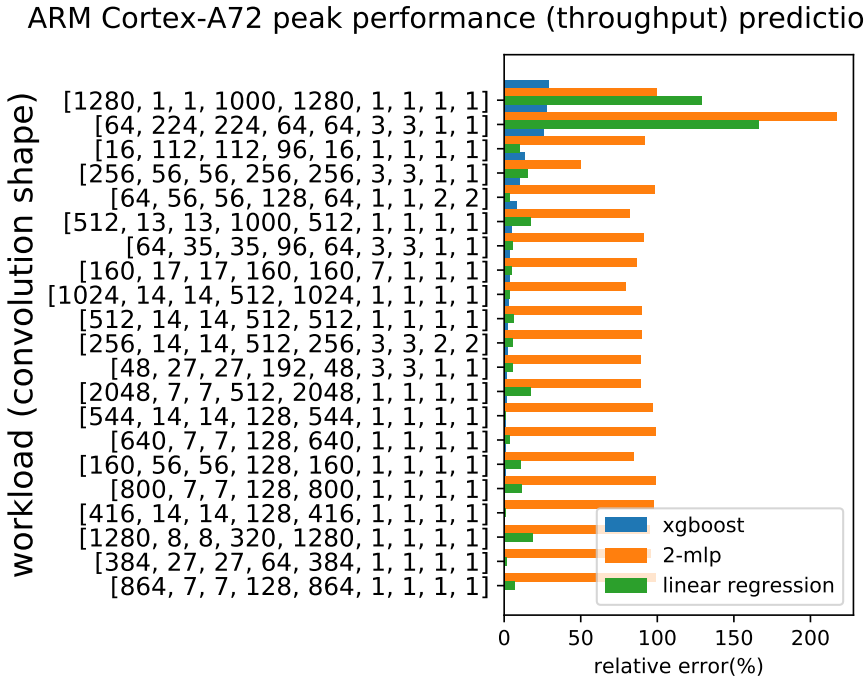


Figure 20: Task 2: Relative error of predicted vs. actual peak-performance with various regression models trained on pretuned workloads for the NVIDIA 1080Ti, Mali T860-MP4, and ARM Cortex-A72. The vertical axis denotes shape parameters of 2D convolution. Workloads are sorted from highest error to lowest. Orange bars show actual measured performance.

6

END-TO-END RESOLUTION AND MODEL CO-OPTIMIZATION

INTRODUCTION

The choice of image resolution is an important hyperparameter for neural networks. Input images for computer vision models typically comprise a wide range of resolutions, qualities, and object scales, yet the choice of input resolution is typically a static one. Furthermore, computation and memory requirements scale approximately quadratically with input resolution, increasing the cost of scaling input resolution dramatically. In this work, we claim that image resolution is an underexploited hyperparameter in neural network models, and quantify the impact that the choice of resolution has on system resources such as compute throughput and storage bandwidth. We frame the choice of neural network resolution at inference time as existing in a space of many related parameters, such as crop area, the quantity of data to read, and the configuration of compute kernels (Figure 21). We study questions about the impact of image resolution in the static resolution setting, followed by the question of whether image resolution can be a dynamic choice.

WHAT IS THE IMPACT OF IMAGE RESOLUTION? Modern computer vision models typically run at a fixed resolution, with this resolution often chosen in tandem with the model architecture. While recent work has drawn attention to the importance of proper resolution scaling with respect to computational cost and model accuracy [90], the true wall-clock latency impact on inference systems remains ambiguous. When scaling the input resolution of a model, the number of arithmetic operations incurred by increasing floating-point operations can be easily counted, but it is unclear whether hardware utilization is identical across all resolutions such that wall-clock time scales commensurately. Additionally, storage capacity and bandwidth are precious resources that are intertwined with image resolution, raising additional questions regarding the quantity of image data that computer vision models need for accurate inference.

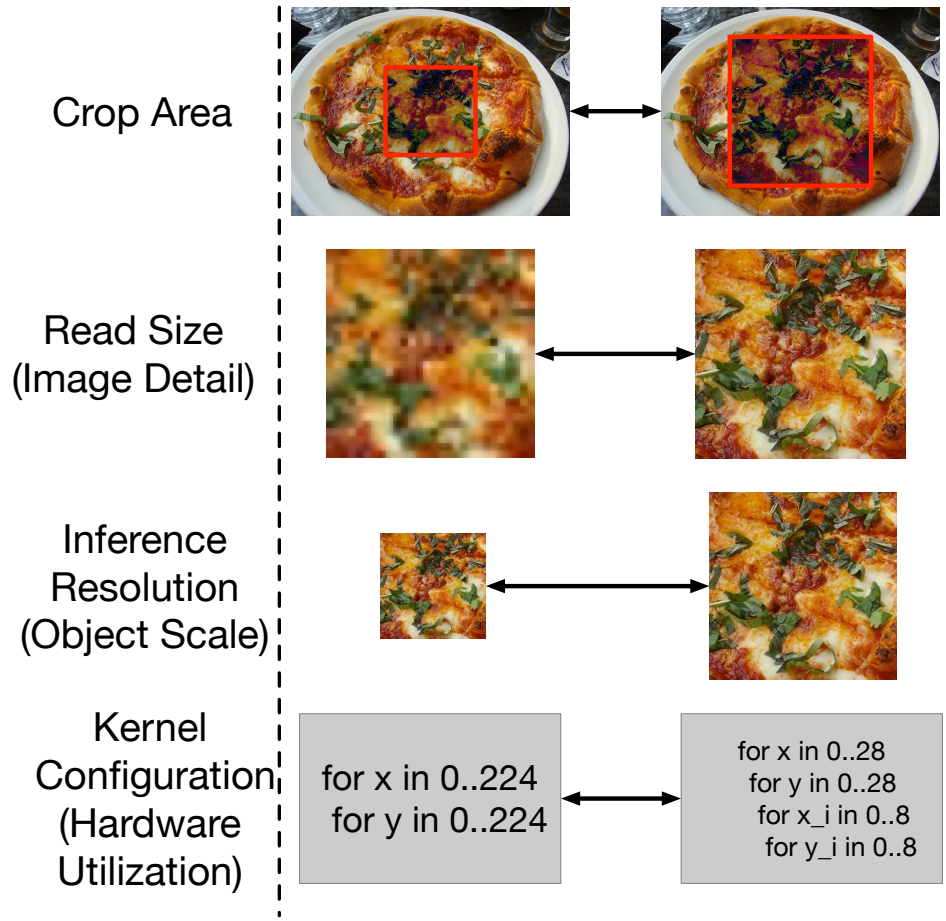


Figure 21: The choice of inference resolution in neural networks introduces associated tightly coupled choices, controlling properties such as the apparent size of objects (crop area), image detail (read size), and inference latency (compute kernel configuration). Each choice potentially impacts one or more of model accuracy, inference time, and data storage bandwidth.

To understand the impact of image resolution on compute resources, we characterize the utilization gap of neural networks at different resolutions on typical inference hardware (commodity CPUs), and the extent to which this gap can be closed through autotuning of compute kernel implementations. To understand the impact of image resolution on storage resources, we characterize the accuracy drop incurred by progressively reducing the amount of image data read for model inference.

CAN RESOLUTION BE CHOSEN DYNAMICALLY? Additionally, we pose the question of whether image resolution needs to be a static hyperparameter for model inference. Intuitively, not all classification tasks or categories require the same level of image detail for accurate evaluation, so it is likely that not all input examples require the same resolution while preserving model accuracy. Furthermore, image resolution in neural networks is closely tied to the perceived *scale*, or relative sizes of objects in images. Recent work [94] has pointed out that the choice of resolution implicitly biases the model towards a specific distribution of object scales (the apparent size of objects) based on data augmentation choices at training time. While a proposed fix [94] is to fine-tune the model for the expected distribution of object scales at test-time, this solution relies on the assumption that the test distribution is known and fixed. The issue is a lack of flexibility during inference, where a mismatch between the desired or expected object size can be corrected by a change in inference resolution.

To understand the potential benefits of dynamism in this scenario, we evaluate a two-model pipeline that uses a lightweight model to select the best inference resolution for a larger backbone model, with the goal being to recover most of the accuracy of choosing the "correct" resolution for inference. We describe this issue in more detail in Section 6.2. Furthermore, in the general case, we expect that multiple-resolution support can be valuable for cases where the difficulty of input images varies, and resolution becomes a hyperparameter controlling the amount of information given to the computer vision model.

With these guiding questions, we characterize the tradeoff space of resolution in neural networks, with the aim of tuning several parameters in tandem: the compute kernel implementations for each resolution, how much data is read for each image during inference, and the implicit object scale in each image (*the crop size*), together with image resolution. Along the way, we describe the methods that enable this tradeoff space, including operator autotuning, storage–image format calibration, and a dynamic resolution model pipeline.

Efficient support for multi-resolution inference spans storage, algorithm, and computation perspectives. From a storage perspective, we aim to minimize the number of bytes that need to be read (or transferred over the network) for inference. From an algorithm perspective, we aim to minimize the number of compute operations (FLOPs). From a compute perspective, we aim to maximize the utilization of the underlying hardware or achieve scalability across difference inference resolutions.

IMAGE QUALITY METRIC Starting with the storage perspective, we focus on the issue of limiting the amount of image data that is read or stored for neural network inference. A contrived but relevant example occurs when large images are resized for inference: computer vision models typically perform inference at well below even 1 megapixel resolution ($448 \times 448 \approx 0.2$MP). When resizing large images to low resolution, we can avoid reading unnecessary or fine details of the image. However, this approach requires a method to calibrate or map image quality (as a proxy for neural network accuracy) to bytes read from storage. Image quality metrics such as Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity (SSIM) [97] provide relatively fast estimates of image quality given a source or reference image. These quality metrics allow us to quantitatively compare model accuracy with the quality or level of detail present in an input image. Here, we focus on structural similarity, though the choice of metric is orthogonal to other system choices.

To be effective and efficient at inference or ingestion time, the computation required for the image quality metric should be much lower than that of the downstream computer vision model. This caveat means that while attractive, image quality metrics that are expensive to compute (e.g., those that rely on features from neural networks [109]) are too expensive at this stage in the pipeline.

SHAPE-AGNOSTIC MODELS From an algorithm perspective, a fundamental requirement of flexible multi-resolution inference support is that the computer vision models do not require a fixed input resolution. For many popular modern models, this property is achieved implicitly as they use a global average pooling layer [112] to connect the resolution-agnostic convolution layers to resolution-dependent fully connected layers. However, even with model architectures that require a fixed resolution, multi-resolution support can be achieved with brute force: train models for every resolution.

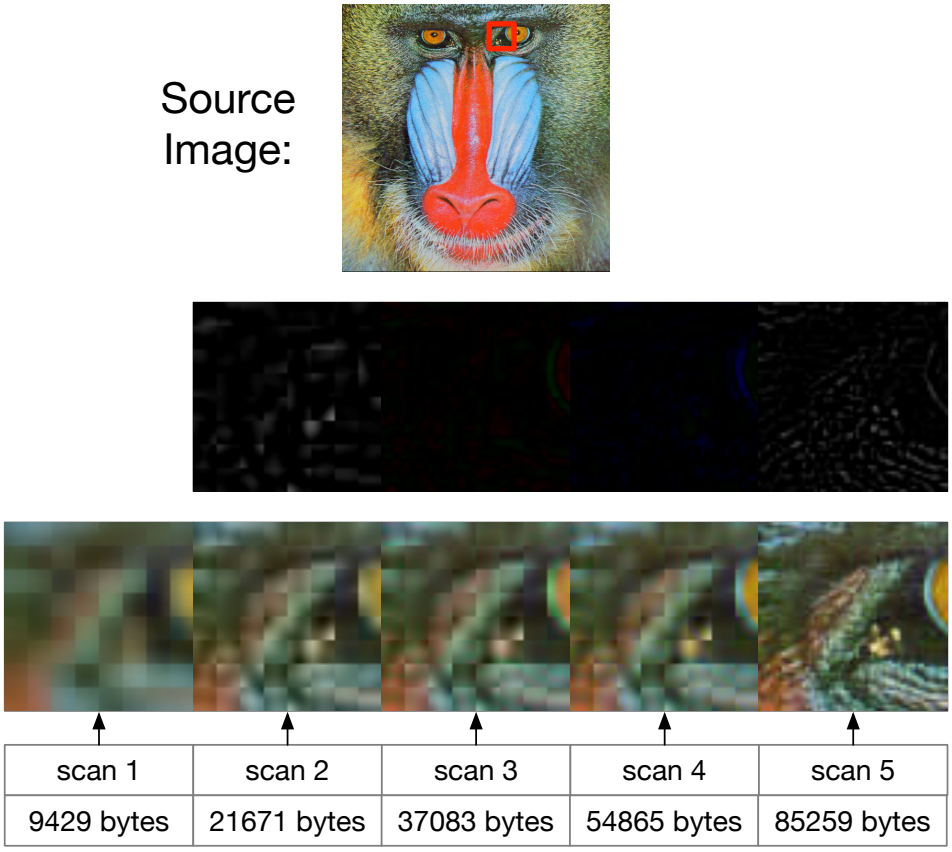


Figure 22: Example of an JPEG image with a progressive encoding (enlarged to show detail). Each image scan refines previous image data by including higher frequency coefficients. The image difference vs. the previous scan is show above each crop; cumulative number of bytes read are shown below.

A PROGRESSIVE IMAGE ENCODING When describing the requirement for an image quality metric, we made the assumption that reading fewer bytes of image data gracefully degrades image quality. However, this assumption requires an image encoding that progressively improves image quality with the amount of data read. Fortunately, this property is satisfied by frequency domain image representations, and only a frequency-domain aware data layout is necessary to provide this property. The progressive JPEG standard is a popular instantiation of a frequency-domain aware data layout that provides a progressive image encoding.

Progressive JPEG achieves this property by arranging image data in multiple passes of increasing detail. Concretely, the data layout groups image data roughly in frequency domain order, relying on the property that lower frequency coefficients tend to encode coarse image details first. By transmitting (when reading or sending data) or rendering coarse details first, a

lossy preview of the image can be generated before all the image data has been received or rendered.

Conveniently, this format can also be used to partition image data for lower resolution versions or previews [102]. As an extreme example, using the first (DC) component of each 8×8 macroblock yields a subsampled image at 1/8th the original resolution. By leveraging a quality metric, we can target resolutions in between 1/8th and the original resolution by mapping frequency coefficient groupings (called scans in JPEG) to target resolutions. Figure 22 shows an example of how image detail increases as more scans of a progressive JPEG image are rendered. We use this property explicitly to selectively load a fraction of the total image data when executing neural network inference at different image resolutions: lower resolutions require fewer progressive JPEG scans and fewer bytes of image data.

SPECIALIZING OPERATOR IMPLEMENTATIONS From a computation perspective, potential savings from reduced floating-point operations (FLOPs) materialize only when high compute utilization can be achieved across each resolution choice. As the compute utilization of deep learning operators can be highly dependent on input shapes (e.g., resolution), operator implementations that are specialized for the range of resolutions are necessary. Here, we leverage prior work on automatic tensor program optimization [17, 71] to generate resolution-specialized operator implementations while minimizing programmer effort.

CROP SIZES, RESOLUTION, AND SCALE Efficient multiple-resolution support can be motivated by the lack of scale invariance (more formally, equivariance) [84] in current computer vision models. This issue stems from the fact that while convolution operators are translation equivariant, they are not scale equivariant [94]. Figure 23 shows an example of how different crop sizes can present objects at different scales to neural networks, and the corresponding change to inference resolution required to compensate for scale differences. The lack of scale equivariance results in models being sensitive to the distribution of object scales, and even with the typical remedy of data augmentation (e.g., in the form of random cropping), model performance can be improved by fine-tuning on a known scale distribution [94]. We characterize the impact of the issue of popular neural network architectures' *lack* of scale invariance by evaluating model accuracy at several crop sizes—we will see that the favored resolutions for model inference heavily depends on the image crop size due to this phenomenon.

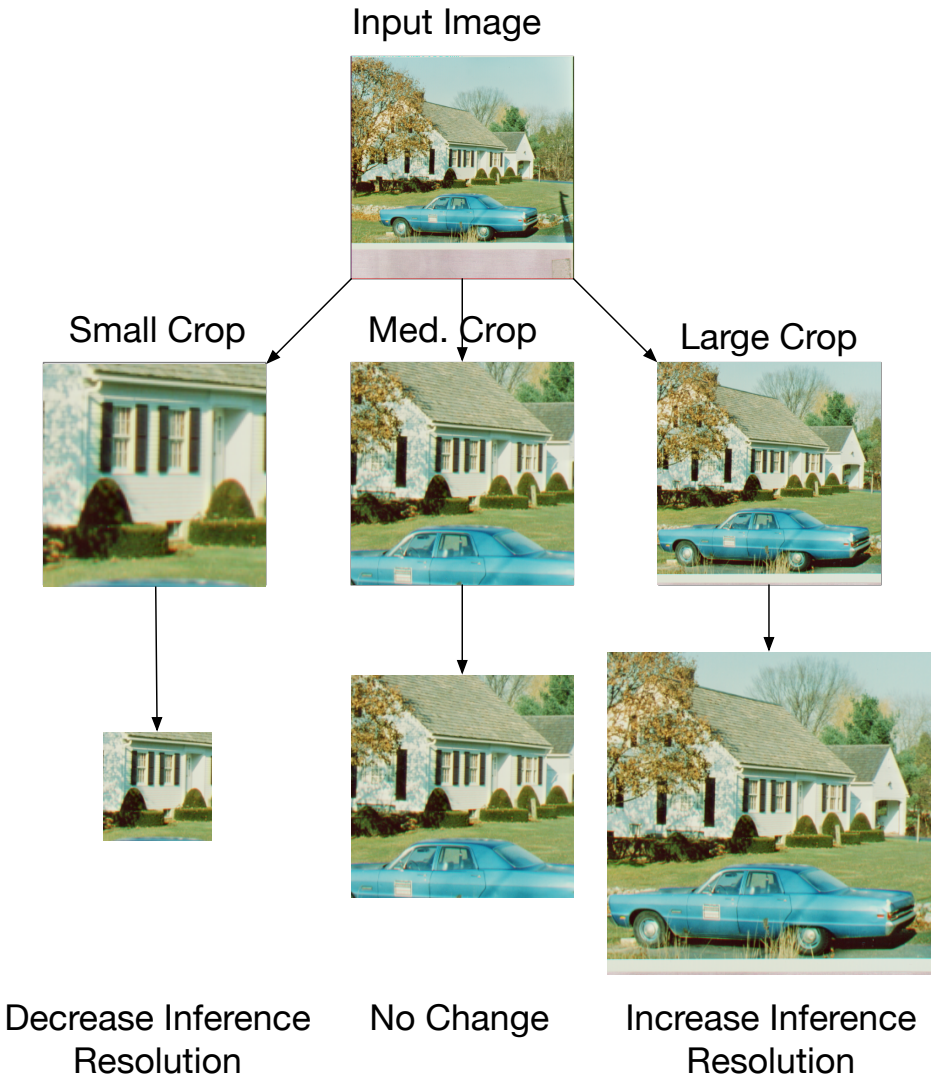


Figure 23: Neural networks are sensitive to the apparent scale of objects. This apparent scale is determined by the image crop and the resolution used to evaluate the model. We show three different crops of the same image and the required change to the inference resolution required to match the object scales across the images.

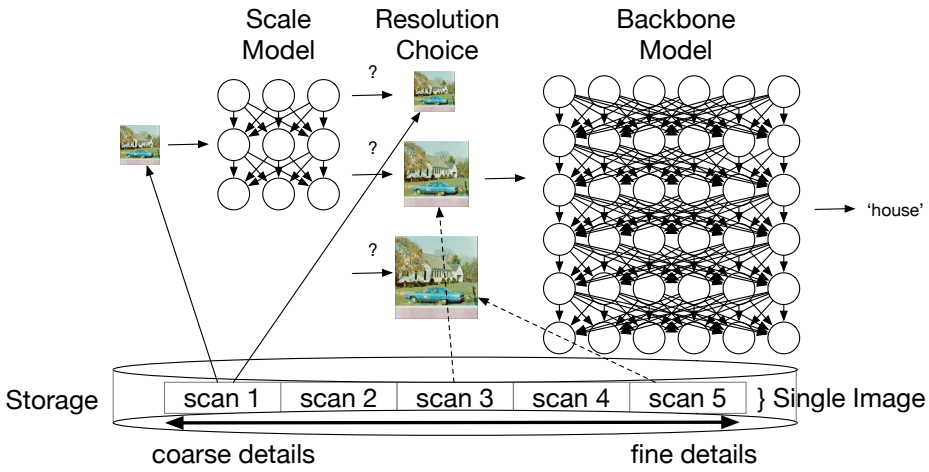


Figure 24: Example of a dynamic resolution system: images are stored with a progressive encoding that arranges each image as a sequence of scans. Low resolution images are first sent to a small scale model that predicts the best resolution for inference. If necessary, additional image data is read to to to generate the appropriate resolution for inference.

CHOOSING IMAGE RESOLUTION FROM OBJECT SCALE

We describe a simple way to leverage a multi-resolution enabled inference pipeline dynamically for object scales, by using two neural network models in sequence (Figure 24) such that resolution is chosen automatically. We refer to the first model as the *scale* model, as it roughly attempts to predict the appropriate scale for neural network inference. We refer to the second model as the *backbone* model; the backbone model performs the specified computer vision task at the chosen scale (resolution). While this example two model pipeline introduces an additional control flow decision of which resolution to execute, this decision is at a coarse granularity (an entire image inference), we believe that control flow at a per-example granularity acceptable for inference workloads–especially those that typically run at batch size 1 (e.g., on CPUs).

SCALE MODEL The scale is trained with a multilabel classification objective: it aims to predict whether a trained backbone model will be accurate at a given resolution for a given image. At inference time, we select the resolution chosen by the scale model using the resolution with the highest predicted likelihood of making a correct prediction. We find that as determining object scales does not require fine image details, the scale model can be lower in resolution (e.g., 112×112) relative to the backbone model without significantly sacrificing accuracy. This reduction in resolu-

Model	Resolution	GFLOPs	Accuracy
ResNet-18	112×112	0.5	47.8
ResNet-18	168×168	1.1	62.8
ResNet-18	224×224	1.8	69.5
ResNet-18	280×280	2.9	**70.7**
ResNet-18	336×336	4.2	70.1
ResNet-18	392×392	5.8	69.4
ResNet-18	448×448	7.3	68.9

Table 6: Example of compute complexity scaling with input resolution (in billions of floating-point operations). Here, the accuracy values are obtained by performing inference on a model trained at 224×224 resolution, indicating the train-test resolution discrepancy [94] where higher resolutions do not improve accuracy due to a scale mismatch.

tion, combined with a choice of efficient model architecture minimizes the additional computational cost of the backbone model.

BACKBONE MODEL The backbone model for each dataset split (see Figure 25) is trained as a standard classification model without additional modification. Note that we do not train separate backbones for each resolution, instead relying on the input-shape agnostic operators of modern model architectures such as ResNet to reduce training costs. Running the model at a different resolution than it was trained with does not degrade accuracy, provided the object scales are matched. However, the inference cost (in terms of FLOPs) increases nearly quadratically with the backbone model, as the computational complexity of convolution layers depends on the area of the input feature map. True wall-clock scaling is slightly better, as higher compute complexity tends to also increase the utilization of hardware execution.

SCALE MODEL TRAINING The multilabel classification objective introduces the problem of another data split, as training a scale model required an already trained backbone model. To leverage all available data when training the scale model, we train the scale model using a cross-validation style approach (Figure 25). Several backbone models are trained on disjoint parts of the training set, and the scale model is trained by alternating backbone models and the corresponding training sets. For our evaluation, we train four different backbone models on 3/4ths of the ImageNet and Cars datasets, and train the scale model using the corresponding 1/4th split for each backbone. When measuring end-to-end accuracy, we use a backbone trained on the full training set.

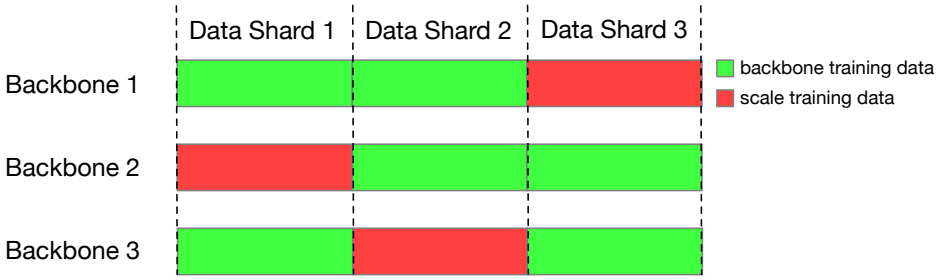


Figure 25: Training a two model pipeline via a cross-validation style approach.

CHOOSING HOW MUCH DATA TO READ FOR EACH RESOLUTION

Given a quality metric, such as SSIM, we can use the quality metric to compare model accuracy against the amount of data read to establish curves (Figure 26, Figure 27) for model accuracy and image quality for each resolution. These curves can form the basis for a storage policy that chooses the amount of data to read for a given resolution requested by a computer vision model. For our purposes, we use structural similarity, as it is more convenient to design a calibration algorithm that searches in the range [0.0, 1.0] than with PSNR where the value for "perfect" image quality goes to infinity. Note that as the amount of data required for accurate inference is a data-dependent task, we pose this as a *calibration* task, where a small amount of training data is reserved to tune quality thresholds determining when the amount of image data read is sufficient.

DATASETS For storage calibration, we use two datasets chosen for their differences in resolution distribution and type of classification. The first is the popular ImageNet dataset [75] comprising 1 million images of 1,000 object classes. The second is the Stanford Cars dataset [55] comprising 16,185 images of 196 fine-grained object classes. While the Cars dataset contains fewer images (with less than 1/100th the training set size of ImageNet), some are of considerably higher resolution than the ImageNet dataset, which yields potential differences when calibrating storage for inference at a given resolution. The average dimensions of training images in the Cars dataset are 699 × 482 pixels while the average dimensions of images in the ImageNet dataset are 472 × 405 pixels.

CALIBRATION PROCEDURE To search for the minimal quality (SSIM threshold) that satisfies the accuracy target, we run binary search of over the SSIM interval [0.94, 1.0], and terminate the search after the step size falls below 0.0001, with the constraint that no more than 0.05% accuracy is lost for each of the resolutions. We use three different train/validation splits of

Cars and ImageNet (shown as the different seeds in Figure 26 and Figure 27) generated on the training set of the respective dataset. We limit the number of images used for calibration to 10,000 (to reduce the amount of computation required). We use the median SSIM threshold found by this search for each resolution in our storage evaluation.

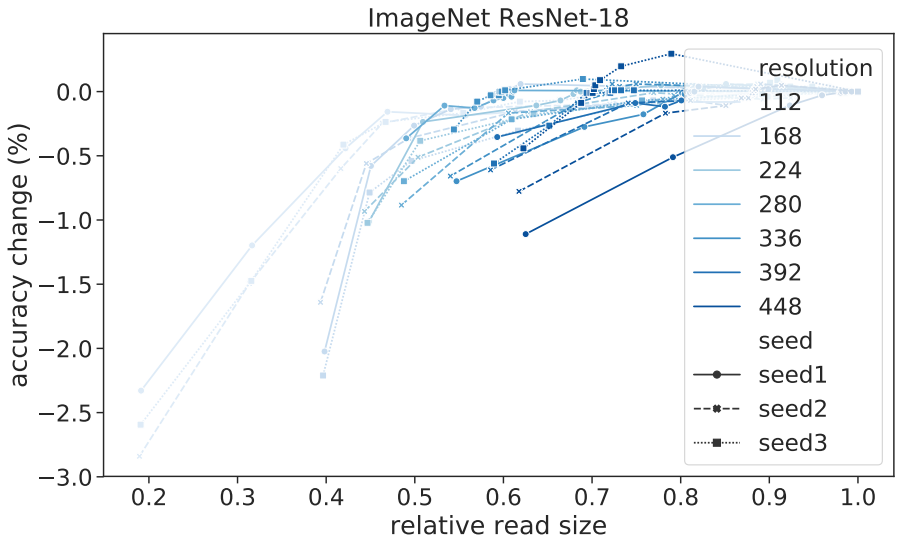
Figure 26 and Figure 27 show accuracy vs. the relative amount of data read (normalized to 1.0), averaged over a collection of images in the training set of ImageNet and Cars respectively. Lower resolutions require less image data for the same SSIM value, but accuracy degrades more rapidly with respect to the amount of image data read. Interestingly, while there is a general trend of accuracy increasing with the amount of image data read, the points at which the maximum accuracy is reached is not necessarily when all the image data is read.

We find substantial differences in the relationship between image quality and model accuracy for ImageNet vs. Cars. These differences can be attributed to the respective distributions of image resolution the datasets, but another potential explanation is the difference in types of image features most important for different datasets. Most immediately, the curves of accuracy vs. image read size appear to be shifted left for Cars vs. ImageNet: accuracy is better preserved even when images are loaded at low fidelity for Cars. If ImageNet favors fine-grained texture details while Cars favors abstract shapes, this can explain the different image quality requirements.

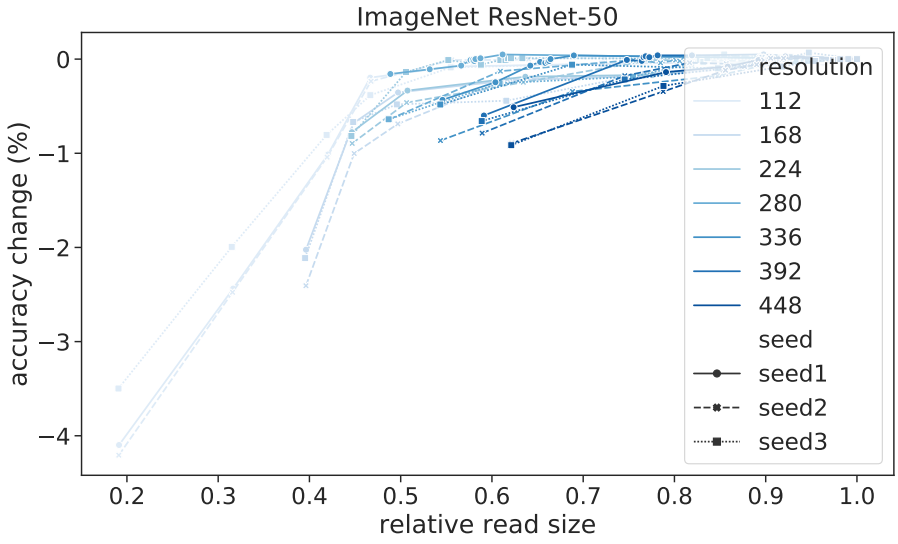
One trend common to both datasets is that higher image resolutions often required lower image quality (when compared to the ground truth resized image) to maintain model accuracy. This trend is surprising as intuitively, one might expect a benefit of higher input resolution to be the inclusion of details lost after resizing to lower resolutions. In fact, this trend is pronounced enough that maintaining accuracy at higher inference resolutions may require less image data to be read than for inference at lower resolutions. For Cars, minimal accuracy losses were observed at higher resolutions even when just over half the image data was read. On ImageNet, this effect was less pronounced, with over 80-90% of image data required to minimize accuracy loss at higher resolutions.

MAXIMIZING HARDWARE UTILIZATION FOR EACH RESOLUTION

Our goal with respect to compute kernels is to find the optimal implementation for every resolution, ideally preserving hardware utilization across all resolutions. Common computationally intensive operators for computer vision models such as 2D convolution typically require highly specialized implementations on modern hardware such as GPGPUs or even CPUs with

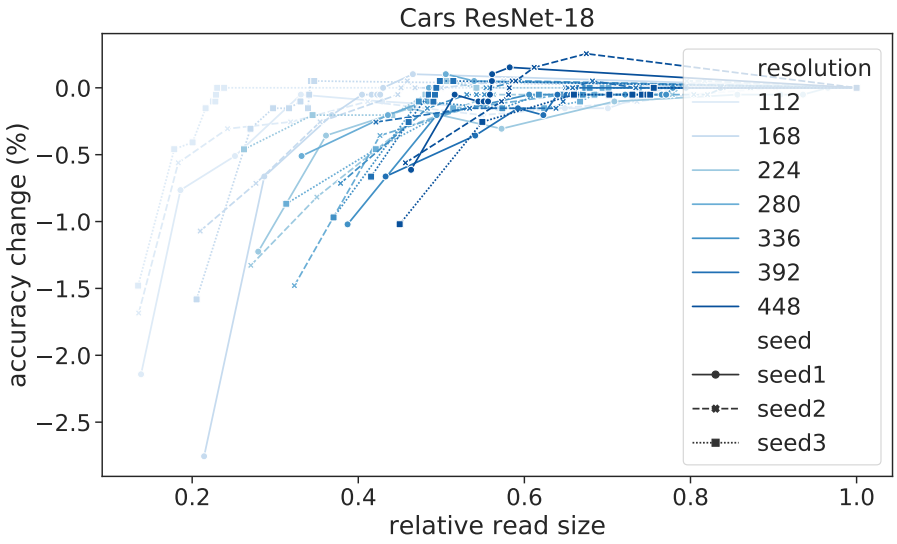


(a)

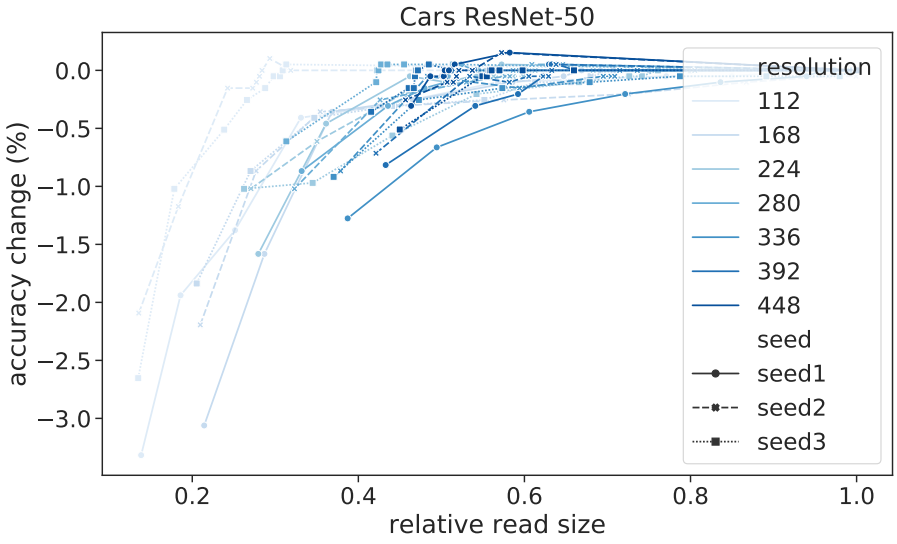


(b)

Figure 26: Storage calibration: relative top1 accuracy change of ResNet-50 and ResNet-18 on ImageNet at different resolutions with varying amounts of image data read vs. reading all image data at each resolution.



(a)



(b)

Figure 27: Storage calibration: relative top1 accuracy change of ResNet-18 and ResNet-50 on Stanford Cars at different resolutions with varying amounts of image data read.

wide vector instruction sets. As these specialized implementations depend on hardware implementation details such as the organization and size of the compute units (e.g., CUDA cores) and memory hierarchy (e.g., Shared/Scratchpad memory, caches, and global DRAM), they are also highly dependent on input sizes and data layouts. These dependencies mean that implementations are also highly sensitive input shapes for each operator, which are dependent on the input resolution to a neural network model. While it is possible to manually craft implementations for each neural model and resolution, the combination of the two pose a tedious engineering challenge. Here, we leverage prior work on automatic compiler optimizations [17] for shape-specific deep learning kernels to generate a specialized implementation for each resolution. With the use of autotuning, we characterize the throughput gap (Figure 28) between high and low resolution inference stemming from decreasing hardware utilization, especially among library implementations that may be overfitted to specific resolutions.

Operator tuning searches a wide space of possible parameter choices for the highest performing combination. Figure 13 shows an abstract example of the choices for a deep learning operator. These parameter choices include loop tiling factors, data layouts, and loop orderings among others. This search can be viewed as a black-box optimization problem (in fact, the underlying code generator and hardware comprise multiple black boxes), and is performed by directly measuring running times of each implementation on hardware. We note that this process can be expensive (on the order of hours per-neural network resolution and model), but this cost can be amortized quickly with many neural network inferences. As we target inference pipelines, we focus on optimizations for commodity x86 CPUs, although the approach is general across all hardware devices such as GPGPUS[1].

EVALUATION

We cover several aspects of efficiency in our evaluation, starting with optimizing utilization and wallclock time latency through operator autotuning for each resolution in subsection 6.6.1. Next, we move to the relationship between inference resolution, image crop sizes, and computational cost in subsection 6.6.2. Finally, we cover the impact of storage calibration for reducing the amount of read data necessary for each inference resolution in subsection 6.6.3.

1 A highly related problem on GPGPUs is the issue of sustaining hardware utilization at batch size 1, where the may not be enough computation to "fill the machine" per inference example.

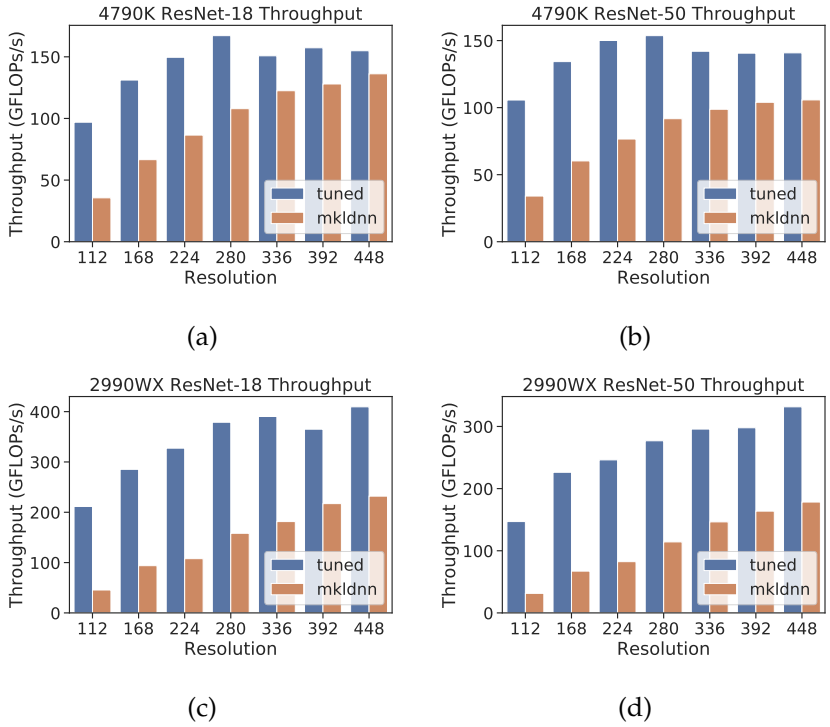


Figure 28: Throughput of ResNet-18 and ResNet-50 at different resolutions (plotted according to GFLOPs/s) using the Intel MKLDNN Library compared with a tuned implementation for each resolution measured on Intel 4790K and AMD 2990WX processors. Tuning better sustains throughput at lower resolutions.

Closing the Throughput Gap for Each Resolution

Figure 28 compares the inference time for ResNet-18 and ResNet-50 using a library implementation (Intel MKLDNN) and using a tuned version for each resolution. Here, we consider the typical inference scenario of batch size one. While there is an absolute improvement in performance, we emphasize that the tuned implementations achieve better throughput (even compared to tuned high resolution kernels) for lower resolutions that have fewer operations. Here, the challenge is to keep the utilization of the hardware high even when the model contains roughly $\frac{1}{16}$th the number of operations. We observe the same effect on both the AMD 2990WX and 4790K (Figure 28 (a) vs. (c) and (b) vs. (d)). Ideally, the bar plot of throughput would be flat across all the resolutions, indicating perfect scaling and identical hardware utilization across each of the resolutions.

We find that the scaling differences between specialized and library implementations are large, especially when comparing the two extremes in image resolution. Concretely, this translates into a speedup of only 3.9×

and 4.9× when comparing inference at 448 × 448 to 112 × 112 on 4790K for ResNet-18 and ResNet-50 respectively. Note that ideal scaling with GFLOPs in each case is 15.0× for ResNet-18 and 15.2× for ResNet-50. With tuning, more of the ideal scaling in each case is recovered as 9.4× and 11.4× speedups are achieved when switching from 448 × 448 to 112 × 112 resolution. This pattern is more pronounced on AMD 2990WX (which likely has less targeted optimizations in MKLDNN), where the untuned scaling from 448 × 448 to 112 × 112 is 2.9× and 2.7× vs. 7.7× and 6.7× for the tuned approach on ResNet-18 and ResNet-50 respectively. Even when comparing higher resolutions, the impact of imperfect scaling is measurable: 3.2× speedup is achieved when switching from ResNet-18 @ 448×448 to 224×224 using tuned kernels and only 1.9× using MKLDNN on 2990WX. The results are similar on 4790K: (MKLDNN 2.5× vs. 3.9× tuned on 4790K). For lower resolution inference, we find the ideal operating point to be 168 × 168 due to the lower compute complexity while most of the throughput of higher resolutions is also attainable with tuning (68-70% on 2990WX, and 85-95% on 4790K).

Accuracy vs. FLOPs

To highlight the flexibility of a dynamic approach to resolution compared to static approaches that perform inference at a fixed resolution, we compare the accuracy of a dynamic resolution switching approach at several different center crop ratios (25%, 39%, 56%, and 75%)[2]. Note that the best static choice of resolution changes for different crop sizes, as predicted by the "train-test resolution discrepancy [94]." While the choice of crop ratio is an evaluation hyperparameter, in practice it is unknown if a model is to be deployed on data from an unknown distribution of object sizes.

The dynamic resolution two model pipeline does not incur a significant overhead in compute complexity, as we use a lightweight, low-resolution architecture for the scale model (MobileNet v2) [77]. We found in early experiments that predicting scale does not require fine-grained image details, and model accuracy did not vary significantly with the capacity of the model architecture (e.g., MobileNet vs. deep ResNets). In this case, the MobileNetv2 architecture used for the scale model corresponds to 0.08 GFLOPs at 112×112 compared to the 1.8GFLOPs of ResNet-18 and 4.1GFLOPs of ResNet-50 at 224 × 224, an almost negligible amount of overhead.

Figure 30 and Figure 29 shows the accuracy achieved by static and dynamic resolution approaches across a range of crop sizes on ImageNet. On

2 "75%" corresponds to the common practice of selecting a center crop (e.g., of 224 pixels from a 256 × 256 image, or 448 pixels from a 512 × 512 image), though the true area is closer to 77%.

the ImageNet dataset, the best static resolution for the 56%, 75%, and 100% center crops was 280 × 280, as expected due to the use of random cropping during training favors slightly larger object scales. However, using the full crop (including *more* of the image area) decreases model accuracy as object scales are biased towards smaller images. The two-model dynamic resolution pipeline attains most of the accuracy of the best static resolution for each approach at a lower FLOP cost, and is pareto-optimal and near the apex of accuracy for most resolution configurations. We note that the best static resolution at at 25% crop drops, as expected, to 224 × 224, and the scale model adjusts accordingly.

As expected, model accuracy is best for higher resolution inference when a larger crop size is used for evaluation and vice versa for lower resolution inference. We point out the dramatic accuracy improvement achieved at the lowest resolution of ResNet-50 on Stanford Cars when switching from a default 75% center crop to a 25% crop; Top-1 accuracy improves from roughly 50% to above 70% for the smaller center crop.

Interestingly, while the general trends in the Accuracy vs. FLOPs curves for ImageNet and Stanford Cars are similar when comparing different crop sizes, there are several distinct differences in their shapes. We note that the accuracy drop with small crops for higher resolutions is much more dramatic in Cars than on ImageNet. At a 25% center crop, the accuracy at 448 × 448 is *lower* than at 112 × 112 for Cars, but it remains higher for ImageNet. Additionally, the accuracy gain at higher resolutions with larger center crops is much more modest in ImageNet (less than 1%) than in Cars (up to 5%).

Accuracy vs. Storage Bandwidth

We compare model accuracy and the amount of data read at several different crop sizes in Table 7 and Table 8. Here, the baseline approach reads the entirety of image data for every resolution. Again, the best accuracy achieved by each approach depends on the crop size. We find that the read savings on the validation set are highly data dependent, as was the case at calibration time (using training data). Few inference resolutions on ImageNet reach above 20% data savings, whereas many resolution/model configurations reach 40% savings with minimal (< 0.1%) accuracy loss.

Due to the quality threshold values being calibrated using the backbone model, the potential read bandwidth savings of the dynamic resolution approach are bounded by the amount of data used at 112 × 112 resolution (the resolution of the scale model). It may be possible to reduce this limit by separately calibrating image quality for the scale model, as the scale model likely requires less image detail to discern object scales.

We observe the most accuracy losses when the amount of data read is limited at smaller crop sizes. This trend can be attributed to our use of only 75% center crops for calibration, and can likely be mitigated by calibrating quality thresholds for other crop scenarios, at the expense of additional compute cost. Overall, however, we find that calibration generalizes well from the training set to the validation set, especially considering the simple image statistics computed by SSIM. These results indicate that the savings depends on the classification task (the relevance of fine details for accuracy), as well as the distribution of resolution among the input resolutions. We note that both the Cars and ImageNet dataset contain images only modestly higher resolution than 448×448 on average.

DISCUSSION

In scenarios where the distribution of object scales at test time is well known, using a static model and the appropriate center crop size is likely to yield a good trade between accuracy and computational cost. We see the dynamic resolution approach as being potentially most useful when some kind of load balancing or latency adjustment is desirable. In such a scenario, one can adjust the crop size for evaluation to reduce the average computational cost of the model pipeline, as the scale model will automatically compensate for the change in object scale. The scale model also has the advantage of improving the robustness of the pipeline to the distribution of object scales. However, the relationship between model accuracy, crop size, and input resolution is data and task-dependent. At low resolutions, the effect of mismatched object scales (not balancing crop size with resolution) is much more pronounced in Cars.

From the perspective of read bandwidth from storage, we find substantial potential read bandwidth savings for both the Cars and ImageNet datasets. Achieving the best savings/accuracy loss ratio will require tailoring storage calibration for each model/dataset scenario, but in our experience a small number of images ($\approx$30,000) is sufficient for calibration.

QUALITY METRICS We note that the use of structural similarity is a crude choice for image quality, especially for neural networks that favor quality metrics more in line with human perception [109]. Perhaps due to the choice of quality metric, a complicating issue occurs when higher resolutions potentially require *lower* image quality or less image data than lower resolutions, further complicating the tradeoffs that can be made between accuracy, compute costs, and storage costs. More sophisticated quality metrics that map better to neural network perceptual quality, as well as ones that are reference free [96] can potentially further improve bandwidth

savings. We have also made the implicit assumption that no image data is discarded from storage. However, if an inference system is chosen such that the storage-accuracy and flops-accuracy tradeoff is to be made ahead of time, further savings can be obtained by (1) cropping images ahead of time, and (2) resizing them ahead of time losslessly. Due to these factors, we consider the bandwidth savings a lower-bound on potential savings in the absence of further more domain information.

ALTERNATIVE LOSS FUNCTIONS FOR THE SCALE MODEL We note that the current dynamic resolution model chooses resolutions solely based on their predicted accuracy given an input image. It is possible to also incorporate inference costs (e.g., latency) for each resolution for further finetuning. Still, using a two-model approach remains pareto-optimal in terms of accuracy vs. FLOPs while improving model accuracy over the default static resolution in most scenarios.

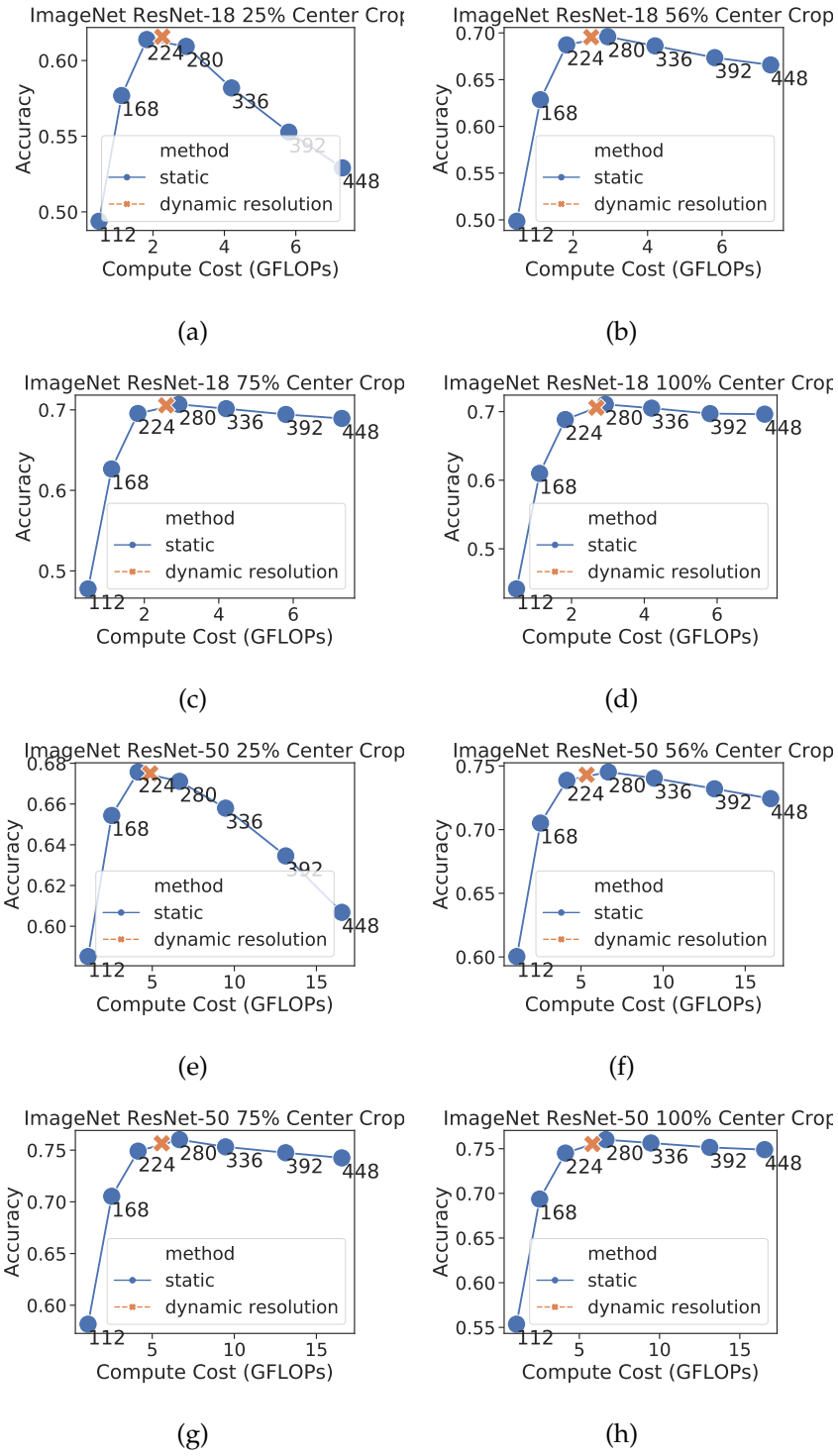


Figure 29: Accuracy vs. FLOPs comparison with static and dynamic resolution approaches using ResNet-18 (a-d)and 50 (e-h) on ImageNet. Crop sizes increase from left to right from 25% on the left to 100% on the right. Smaller crops favor lower resolutions more, while larger crops favor higher resolutions due to the models' dependence on object scale. The dynamic resolution approach operates at nearly the apex of every pareto curve, without hardcoding resolution.

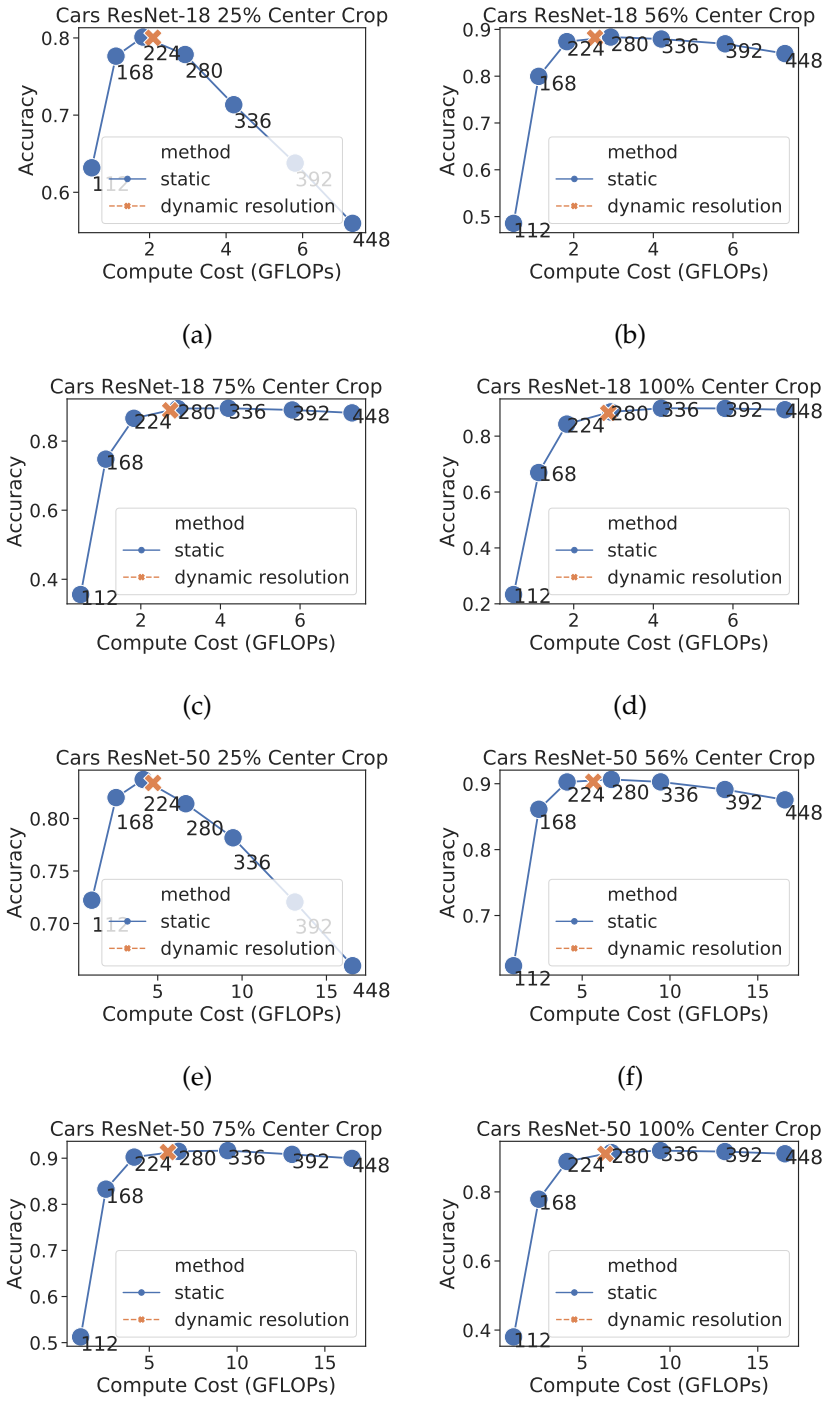


Figure 30: Accuracy vs. FLOPs comparison with static and dynamic resolution approaches using ResNet-18 (a-d) and 50 (e-h) on Stanford Cars. Crop sizes increase from left to right from 25% on the left to 100% on the right. Smaller crops favor lower resolutions more, while larger crops favor higher resolutions due to the models' dependence on object scale. The dynamic resolution approach operates at nearly the apex of every pareto curve, without hardcoding resolution.

ImageNet Res	ResNet-18 75% Crop Default	ResNet-18 75% Crop Calibrated	ResNet-18 56% Crop Default	ResNet-18 56% Crop Calibrated	ResNet-18 25% Crop Default	ResNet-18 25% Crop Calibrated	Read Savings
112	47.8	47.7	49.9	49.8	49.4	49.3	16.4%
168	62.7	62.7	62.9	62.8	57.7	57.7	5.1%
224	69.5	69.5	68.7	68.7	61.4	61.4	5.8%
280	70.7	70.7	69.6	69.5	60.9	60.7	20.2%
336	70.1	70.1	68.6	68.5	58.2	57.5	27.7%
392	69.4	69.4	67.4	67.2	55.3	54.7	17.8%
448	68.9	69.0	66.6	66.5	52.9	52.6	9.5%
dynamic	70.6	70.6	69.6	69.4	61.5	61.5	11.2,10.6,8.6%
ImageNet Res	ResNet-50 75% Crop Default	ResNet-50 75% Crop Calibrated	ResNet-50 56% Crop Default	ResNet-50 56% Crop Calibrated	ResNet-50 25% Crop Default	ResNet-50 25% Crop Calibrated	Read Savings
112	58.2	58.1	60.0	60.0	58.5	58.5	7.4%
168	70.5	70.5	70.5	70.5	65.4	65.4	2.1%
224	74.9	74.9	73.9	73.9	67.6	67.5	8.9%
280	76.0	76.0	74.5	74.6	67.1	67.0	19.2%
336	75.3	75.3	74.0	74.0	65.8	65.7	6.2%
392	74.7	74.7	73.2	73.1	63.5	63.2	9.1%
448	74.2	74.2	72.4	72.3	60.7	60.4	8.0%
dynamic	75.7	75.6	74.3	74.3	67.5	67.5	6.8,6.7,6.5%

Table 7: ImageNet read bandwidth savings; comparing accuracy when reading all data vs. reading the quantity of data according to storage calibration. Accuracy degradation > 0.1% highlighted. Read savings for the dynamic pipeline are for each crop size.

Cars	ResNet-18 75% Crop		ResNet-18 56% Crop		ResNet-18 25% Crop		
Res	Default	Calibrated	Default	Calibrated	Default	Calibrated	Read Savings
112	35.6	35.6	48.6	48.6	63.2	63.1	31.8%
168	74.8	74.7	80.0	79.6	77.6	77.3	59.4%
224	86.6	86.6	87.4	87.3	80.1	80.1	20.5%
280	89.4	89.4	88.4	88.4	77.9	77.9	29.8%
336	89.5	89.5	87.9	88.0	71.3	71.4	31.4%
392	89.0	89.0	86.9	86.9	63.8	63.8	37.2%
448	88.2	88.1	84.8	84.7	56.0	55.8	43.0%
dynamic	88.9	88.9	88.2	88.2	80.0	80.0	25.2,24.0,21.6%
Cars	ResNet-50 75% Crop		ResNet-50 56% Crop		ResNet-50 25% Crop		
Res	Default	Calibrated	Default	Calibrated	Default	Calibrated	Read Savings
112	51.2	50.8	62.4	62.0	72.2	71.5	68.8%
168	83.3	83.3	86.1	86.1	82.0	81.9	30.7%
224	90.2	90.2	90.3	90.2	83.7	83.6	40.9%
280	91.5	91.4	90.6	90.6	81.4	81.4	51.9%
336	91.6	91.6	90.3	90.3	78.2	78.1	6.5%
392	90.8	90.8	89.1	89.1	72.0	71.9	39.8%
448	90.0	89.9	87.6	87.5	66.0	65.6	49.3%
dynamic	91.3	91.2	90.3	90.2	83.4	83.3	48.8,47.1,43.1%

Table 8: Cars read bandwidth savings; comparing accuracy when reading all data vs. reading the quantity of data according to storage calibration. Accuracy degradation > 0.1% highlighted. Read savings for the dynamic pipeline are for each crop size.

7

CONCLUSION

CNNS ENCODE DATA AUGMENTATIONS Typical convolutional neural networks encode attributes corresponding to data augmentations such as object scale, aspect ratio, and various color transformations. However, the signal augmentation is most prevalent in the early layers of models, with the predictive power of activations decreasing with layer depth. This trend suggests that neural networks normalize perturbations introduced by data augmentations.

TUNING PIPELINES FOR DEEP LEARNING KERNELS Automatic machine learning program optimization is becoming a fruitful area of research for both machine learning algorithms and downstream optimization tasks such as NAS and graph optimization. However, one major impediment to researchers currently working in the field is the lack of a reusable system stack spanning predefined search spaces to device-specific code generation and transparent runtimes for benchmarking implementations and prototyping optimization pipelines. We presented SeaNet, which aims to fill this gap by enabling machine learning researchers to plug in their custom optimization algorithms into challenging optimization tasks. We presented two typical problem settings: the optimization of a collection of kernels corresponding to deep learning workloads, and peak-performance prediction for previously unseen kernels. Additionally, we give a detailed description of a device-portable RPC system that enables users to quickly move between different hardware devices without affecting their algorithm implementation. Finally, we presented evaluations of modular implementations for the tasks on various hardware devices to highlight the flexibility of the SeaNet environment.

QUALITY AND BANDWIDTH FOR MULTI-RESOLUTION IMAGE STORAGE Faced with growing demand for storage, image hosting services are increasingly turning to dynamic image resizing to improve the efficiency of image storage. We showed that progressive encodings can dramatically reduce the amount of data that needs to be read for resizing images—potentially

saving over 80% of read bandwidth when tuned encode-time parameters used. Finally, we give an estimate of progressive JPEG decode overheads which suggests that while serving custom progressive images directly to energy-constrained devices is difficult to justify, transcoding custom progressive JPEG on the server side incurs acceptable overheads.

END-TO-END OPTIMIZATION FOR RESOLUTION Image resolution is a fundamental hyperparameter in computer vision with ties to compute complexity, operator optimizations, and storage bandwidth. The best choice of resolution also depends on other choices such as crop sizes and their effect on the distribution of object scales. We systematically characterized the dependencies and relationships between these tradeoffs, and describe methods for maximizing efficiency with respect to compute and storage cost, encompassing both cases where resolution is a static choice and dynamic at inference time.

8

FUTURE WORK

IMPROVING IMAGE STORAGE A limitation of our current storage implementation is the cost of evaluating custom scan configurations. Our naïve implementation takes days to process 24,988 images on 8 cluster nodes (12 cores/12 threads per node) with Westmere-class CPUs. Due to this computational cost, for our end-to-end evaluation in Chapter 6, we used the default scan configuration to estimate the storage bandwidth requirements for each approach. However, we suspect that this time can drastically reduced without sacrificing significant bandwidth savings by aggressively pruning the search space or applying machine learning techniques to choose scan configurations. While the customization process for progressive JPEG is currently expensive, it only needs to be performed once, at write time.

The PSNR metric is limited in its relevance to perceived visual quality [98]. We often found that the PSNR of higher resolution resizes was higher than that of lower resolution resizes even with less image data read—this issue may be mitigated with conservative PSNR thresholds. To the best of our knowledge, there is no standard, widely used method of computing the image quality of a resized image derived from a source image.

Finally, an issue when using progressive JPEG for dynamic resizing is the *minimum* resolution of the resized images. Progressive JPEG is less efficient in terms of read bandwidth for resizes smaller than 10% of the source image, which may limit savings when the source images are much higher in resolution than their resized versions. This threshold is due to JPEG's use of 8×8 macroblocks: even a single frequency coefficient represents at least $\frac{1}{64}$ of the total image data. Even when approximations are used, this approach may require more read bandwidth than pre-resized images. Still, using progressive JPEG should be more space-efficient than baseline JPEG for dynamic resizing.

We expect that a solution to reduce the cost of enumerating custom JPEG scan configurations will be to prune the search space to a much smaller subset of likely "good" configurations. It may be possible to obtain comparable results by only trying a few custom scan configurations per image—with

this subset being determined by identifying the best configurations when naïvely re-encoding a larger dataset of images. Along these lines, even choosing from a larger pool of configurations may be tractable if a machine learning model is applied to each image to choose the best configuration.

DATA AUGMENTATIONS FOR PRETRAINING Data augmentations are an interesting topic of study in unsupervised or semi-supervised learning settings. From one perspective, a reasonable objective is to use augmentations to enforce consistency between perturbed input examples originating from the same source. From another, augmentations that distort images can be potentially useful in creating pre-training tasks (e.g., correctly ordering shuffled image patches, reorienting a rotated image). However, not all augmentations appear to be useful as components of pre-training tasks, as it may be desirable for the downstream fine-tuned model to normalize away such augmentations. In these situations, it may be useful to enforce that neural networks remain equivariant to augmentation attributes, just as convolution has been classically motivated as a translation equivariant operator.

SOLVING THE SCALE EQUIVARIANCE PROBLEM Another approach to solving the scale invariance problem is to change the region of interest that is used as input to a computer vision model dynamically, depending on the positioning and scale of an object in a frame. This approach would effectively be an extension of neural network models that use a version of the attention mechanism popular in natural language processing models. While convolution-based architectures are currently more common in computer vision, recent work [23] has shown that with sufficient computation, attention-based architectures can also be competitive with state-of-the-art convolution approaches.

TUNING DEEP LEARNING KERNELS While tremendous progress has been made in reducing the amount of human engineering effort needed to produce fast kernels for deep learning, much work remains to be done. Current approaches have largely focused on a narrow range of dense linear-algebra inspired operators, although it is unclear whether these operators have been chosen simply because of the ease of mapping them efficiently to hardware or because of their suitability to deep learning architectures. Optimizing arbitrary computation, especially on an end-to-end computation graph remains challenging, as even the subproblem of choosing how to slice the graph is nontrivial.

Perhaps the clearest example of ”slicing” the graph is the current dichotomy between ”graph optimization” and ”kernel optimization.” Current approaches have demonstrated clear benefits by partitioning different

kernels in deep learning computation graphs. However, these approaches still consider deep learning kernels as indivisible black boxes to tame the search space of possible and valid rewrites. The development of future primitives is likely hamstrung by these limitations, as even arbitrary "numpy" style scripting remains difficult for optimization.

Finally, the need for any human insight into the structure of kernel implementations appears unsatisfying in the presence of "tabula rasa" [82] approaches for reinforcement learning. Even in the absence of template driven approaches, current state-of-the-art tuning methods require a considerable amount of human insight to design the search space [110] or promising transformations available to the optimizer. Ideally, to minimize programmer effort and maximize generalization, a "tabula rasa" approach would allow for competitive optimizations without specifying a constrained search space or narrow set of available transformations tailored to specific architectures ahead of time.

ATTENTION MECHANISMS IN VISION One interesting application of attention mechanisms may be to improve model efficiency by focusing on a few relevant patches in an input image. This approach may also support the model in normalizing scale differences in input images, e.g., by choosing just enough patches roughly normalize object scales. However, a challenge of using high-level information (e.g., saliency or relevance) to guide control flow in models is that a large amount of computation may be required to develop the high-level features, so that by the time the decision to select relevant patches can be made, little additional computation can be saved.

REFERENCES

[1] A Year Without a Byte. https://code.flickr.net/2017/01/05/a-year-without-a-byte/ (cited on pages 13, 33).

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pages 265–283 (cited on page 19).

[3] Andrew Adams, Karima Ma, Luke S. Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. In *ACM Trans. Graph.* 38 (2019), 121:1–121:12 (cited on page 14).

[4] Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. Reinforcement learning and adaptive sampling for optimized DNN compilation. In *CoRR* abs/1905.12799 (2019). arXiv: 1905.12799. URL: http://arxiv.org/abs/1905.12799 (cited on pages 47, 50).

[5] Andrew Anderson and David Gregg. Optimal dnn primitive selection with partitioned boolean quadratic programming. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ACM. 2018, pages 340–351 (cited on page 44).

[6] Tomer Bar. Faster Photos in Facebook for iOS. https://code.facebook.com/posts/857662304298232/faster-photos-in-facebook-for-ios/ (cited on page 13).

[7] Paul Barham and Michael Isard. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, ACM. 2019, pages 177–183 (cited on page 44).

[8] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in Haystack: Facebook's photo storage. In *OSDI*, volume 10. 2010, pages 1–8 (cited on pages 13, 33).

[9] Richard Black, Austin Donnelly, Dave Harper, Aaron Ogus, and Anthony Rowstron. Feeding the pelican: using archival hard drives for cold storage racks. In *8th USENIX Workshop on Hot Topics in Storage*

and File Systems (HotStorage 16), USENIX Association. 2016 (cited on page 34).

[10] Michaela Blott, Thomas B Preusser, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. Finn-r: an end-to-end deep-learning framework for fast exploration of quantized neural networks. In *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (2018), page 16 (cited on page 44).

[11] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. Eva^2: exploiting temporal redundancy in live computer vision. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE. 2018, pages 533–546 (cited on page 11).

[12] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. In *Commun. ACM* 59.5 (Apr. 2016), pages 50–57. ISSN: 0001-0782. DOI: 10.1145/2890784. URL: http://doi.acm.org/10.1145/2890784 (cited on page 56).

[13] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *arXiv e-prints*, arXiv:1812.00332 (Dec. 2018), arXiv:1812.00332. arXiv: 1812.00332 [cs.LG] (cited on page 46).

[14] Tianqi Chen and Carlos Guestrin. Xgboost: a scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, ACM. 2016, pages 785–794 (cited on pages 12, 50).

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pages 578–594. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/chen (cited on pages 7, 13, 43, 51).

[16] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31*. Edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., 2018, pages 3389–3400. URL:

http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs.pdf (cited on pages 14, 43).

[17] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, 2018, pages 3389–3400 (cited on pages 7, 72, 80).

[18] Wei Chen, Tie-Yan Liu, Yanyan Lan, Zhi-Ming Ma, and Hang Li. Ranking measures and loss functions in learning to rank. In *Advances in Neural Information Processing Systems*, 2009, pages 315–323 (cited on page 17).

[19] Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, 2016, pages 2990–2999 (cited on page 15).

[20] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pages 305–316 (cited on page 13).

[21] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: learning augmentation strategies from data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2019, pages 113–123 (cited on page 16).

[22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: a large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, Ieee. 2009, pages 248–255 (cited on page 21).

[23] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: transformers for image recognition at scale. In *arXiv preprint arXiv:2010.11929* (2020) (cited on page 94).

[24] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: finding sparse, trainable neural networks. In *arXiv preprint arXiv:1803.03635* (2018) (cited on page 11).

[25] Joshua Fromm, Meghan Cowan, Matthai Philipose, Luis Ceze, and Shwetak Patel. Riptide: fast end-to-end binarized neural networks. In *Proceedings of Machine Learning and Systems 2020* 2 (2020) (cited on page 11).

[26] Joshua Fromm, Shwetak Patel, and Matthai Philipose. Heterogeneous bitwidth binarization in convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2018, pages 4006–4015 (cited on page 44).

[27] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: a service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM. 2017, pages 1487–1495 (cited on page 14).

[28] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *arXiv preprint arXiv:1412.6572* (2014) (cited on page 15).

[29] Lionel Gueguen, Alex Sergeev, Ben Kadlec, Rosanne Liu, and Jason Yosinski. Faster neural networks straight from jpeg. In *Advances in Neural Information Processing Systems*, 2018, pages 3933–3944 (cited on page 23).

[30] Qing Guo, Karin Strauss, Luis Ceze, and Henrique S Malvar. High-density image storage using approximate memory cells. In *ACM SIGPLAN Notices* 51.4 (2016), pages 413–426 (cited on pages 12, 13).

[31] Preeti Gupta, Avani Wildani†, Ethan L. Miller, Daniel Rosenthal, Ian F. Adams, Christina Strong, and Andy Hospodor. An economic perspective of disk vs. flash media in archival storage. In *IEEE MASCOTS*, 2014 (cited on page 33).

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pages 770–778 (cited on pages 22, 61).

[33] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11. Boston, MA: USENIX Association, 2011, pages 295–308. URL: `http://dl.acm.org/citation.cfm?id=1972457.1972488` (cited on page 56).

[34] Elad Hoffer, Berry Weinstein, Itay Hubara, Tal Ben-Nun, Torsten Hoefler, and Daniel Soudry. Mix & match: training convnets with mixed image sizes for improved accuracy, speed and scale resiliency. In *arXiv preprint arXiv:1908.08986* (2019) (cited on page 12).

[35] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: efficient convolutional neural networks for mobile vision applications. In *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861 (cited on page 44).

[36] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ACM. 2013, pages 167–181 (cited on pages 13, 33).

[37] Mark J. Huiskes and Michael S. Lew. The MIR Flickr retrieval evaluation. In *MIR '08: Proceedings of the 2008 ACM International Conference on Multimedia Information Retrieval*, Vancouver, Canada: ACM, 2008 (cited on page 38).

[38] ImageMagick Studio, LLC. ImageMagick. 2008 (cited on page 39).

[39] Independent JPEG Group and others. Libjpeg. 2014 (cited on page 36).

[40] Md Amirul Islam, Sen Jia, and Neil DB Bruce. How much position information do convolutional neural networks encode? In *International Conference on Learning Representations*, 2019 (cited on page 20).

[41] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. In *Neural computation* 3.1 (1991), pages 79–87 (cited on page 12).

[42] Djordje Jevdjic, Karin Strauss, Luis Ceze, and Henrique S Malvar. Approximate storage of compressed and encrypted videos. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pages 361–373 (cited on page 12).

[43] Yu Ji, Ling Liang, Lei Deng, Youyang Zhang, Youhui Zhang, and Yuan Xie. Tetris: tile-matching the tremendous irregular sparsity. In *Advances in Neural Information Processing Systems*, 2018, pages 4115–4125 (cited on page 11).

[44] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *2nd Conference on Systems and Machine Learning (SysML)*, Palo Alto, CA, USA, Apr. 2019 (cited on page 14).

[45] Zhihao Jia, Matei Zaharia, and Alex Aiken. Optimizing DNN Computation with Relaxed Graph Substitutions. In *2nd Conference on Systems and Machine Learning (SysML)*, Palo Alto, CA, USA, Apr. 2019 (cited on pages 14, 43, 44).

[46] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. A study of bfloat16 for deep learning training. In *arXiv preprint arXiv:1905.12322* (2019) (cited on page 11).

[47] Angjoo Kanazawa, Abhishek Sharma, and David Jacobs. Locally Scale-Invariant Convolutional Neural Networks. 2014. arXiv: 1412.5104 [cs.CV] (cited on page 15).

[48] Angjoo Kanazawa, Abhishek Sharma, and David W. Jacobs. Locally scale-invariant convolutional neural networks. In *CoRR* abs/1412.5104 (2014). arXiv: 1412.5104. URL: http://arxiv.org/abs/1412.5104 (cited on page 12).

[49] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. A code generator for high-performance tensor contractions on gpus. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019. Washington, DC, USA: IEEE Press, 2019, pages 85–95. ISBN: 978-1-7281-1436-1. URL: http://dl.acm.org/citation.cfm?id=3314872.3314885 (cited on page 14).

[50] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pages 1–6 (cited on page 11).

[51] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. In *arXiv preprint arXiv:2006.09616* (2020) (cited on page 12).

[52] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *SIGARCH Comput. Archit. News* 44.3 (June 2016), pages 115–127. ISSN: 0163-5964. DOI: 10.1145/3007787.3001150. URL: http://doi.acm.org/10.1145/3007787.3001150 (cited on page 43).

[53] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: a language and compiler for application accelerators. In *ACM Sigplan Notices*, volume 53. 4. ACM. 2018, pages 296–311 (cited on page 43).

[54] Risi Kondor and Shubhendu Trivedi. On the generalization of equivariance and convolution in neural networks to the action of compact groups. In *arXiv preprint arXiv:1802.03690* (2018) (cited on page 15).

[55] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE international conference on computer vision workshops*, 2013, pages 554–561 (cited on page 76).

[56] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012, pages 1097–1105 (cited on pages 6, 15).

[57] Tom Lane. JPEG image compression FAQ, part 1/2. `http://www.faqs.org/faqs/jpeg-faq/part1/index.html` (cited on page 38).

[58] Vincent T Lee, Armin Alaghi, John P Hayes, Visvesh Sathe, and Luis Ceze. Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE. 2017, pages 13–18 (cited on page 11).

[59] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: scaling giant models with conditional computation and automatic sharding. In *arXiv preprint arXiv:2006.16668* (2020) (cited on page 12).

[60] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. In *ACM Trans. Graph. (Proc. SIGGRAPH)* 37.4 (2018), 139:1–139:13 (cited on page 14).

[61] Zachary C Lipton. The mythos of model interpretability. In *Queue* 16.3 (2018), pages 31–57 (cited on page 17).

[62] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: differentiable architecture search. In *arXiv preprint arXiv:1806.09055* (2018) (cited on pages 5, 43).

[63] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani, Songtao He, Frank Cangialosi, Shaileshh Bojja Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Mohammad Alizadeh. Park: an open platform for learning augmented computer systems. In (2019) (cited on page 14).

[64] Amrita Mazumdar, Brandon Haynes, Magdalena Balazinska, Luis Ceze, Alvin Cheung, and Mark Oskin. Vignette: perceptual compression for video storage and processing systems. In *arXiv preprint arXiv:1902.01372* (2019) (cited on page 12).

[65] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org. 2017, pages 2430–2439 (cited on page 14).

[66] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Lianmin Zheng, Eddie Yan, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. A hardware-software blueprint for flexible deep learning specialization. In *IEEE Micro* (2019) (cited on page 43).

[67] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: a distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pages 561–577. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/moritz (cited on page 13).

[68] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. In *ACM Transactions on Graphics (TOG)* 35.4 (2016), page 83 (cited on page 14).

[69] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: an imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, 2019, pages 8026–8037 (cited on page 5).

[70] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: decoupling algorithms from schedules for high-performance image processing. In *Communications of the ACM* 61.1 (2017), pages 106–115 (cited on page 51).

[71] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Acm Sigplan Notices* 48.6 (2013), pages 519–530 (cited on pages 13, 72).

[72] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, Springer. 2016, pages 525–542 (cited on page 11).

[73] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G. Wei, and D. Brooks. A case for efficient accelerator design space exploration via bayesian optimization. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, July 2017, pages 1–6. DOI: 10.1109/ISLPED.2017.8009208 (cited on page 43).

[74] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33. 2019, pages 4780–4789 (cited on page 5).

[75] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. In *International journal of computer vision* 115.3 (2015), pages 211–252 (cited on pages 5, 33, 76).

[76] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. In *ACM Transactions on Computer Systems (TOCS)* 32.3 (2014), pages 1–23 (cited on page 12).

[77] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pages 4510–4520 (cited on page 82).

[78] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: the sparsely-gated mixture-of-experts layer. In *arXiv preprint arXiv:1701.06538* (2017) (cited on page 12).

[79] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pages 3646–3654 (cited on page 11).

[80] Helgi Sigurbjarnarson, Petur O. Ragnarsson, Juncheng Yang, Ymir Vigfusson, and Mahesh Balakrishnan. Enabling space elasticity in storage systems. In *9th ACM International on Systems and Storage Conference (SYSTOR)*, Haifa, Israel, 2016 (cited on page 13).

[81] Helgi Sigurbjarnarson, Petur Orri Ragnarsson, Ymir Vigfusson, and Mahesh Balakrishnan. Harmonium: elastic cloud storage via file motifs. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA: USENIX Association, 2014. URL: `https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/sigurbjarnarson` (cited on page 13).

[82] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. In *arXiv preprint arXiv:1712.01815* (2017) (cited on page 95).

[83] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *arXiv preprint arXiv:1409.1556* (2014) (cited on page 57).

[84] Ivan Sosnovik, Michał Szmaja, and Arnold Smeulders. Scale-Equivariant Steerable Networks. 2019. arXiv: `1910.11093 [cs.CV]` (cited on pages 12, 72).

[85] Ivan Sosnovik, Michał Szmaja, and Arnold Smeulders. Scale-equivariant steerable networks. In *arXiv preprint arXiv:1910.11093* (2019) (cited on page 15).

[86] Steve Souders. *Even faster web sites: performance best practices for web developers*. O'Reilly Media, Inc., 2009 (cited on page 40).

[87] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision*, 2017, pages 843–852 (cited on page 33).

[88] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*, 2015 (cited on page 47).

[89] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pages 2820–2828 (cited on pages 14, 46).

[90] Mingxing Tan and Quoc V Le. Efficientnet: rethinking model scaling for convolutional neural networks. In *arXiv preprint arXiv:1905.11946* (2019) (cited on pages 5, 6, 43, 67).

[91] Mingxing Tan and Quoc V Le. Mixnet: mixed depthwise convolutional kernels. In *arXiv preprint arXiv:1907.09595* (2019) (cited on page 43).

[92] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: advanced photo caching on flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pages 373–386 (cited on page 13).

[93] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Herve Jegou. Fixing the train-test resolution discrepancy. In *Advances in Neural Information Processing Systems 32*. Edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pages 8250–8260. URL: `http://papers.nips.cc/paper/9035-fixing-the-train-test-resolution-discrepancy.pdf` (cited on pages 15, 19).

[94] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy. In *Advances in Neural Information Processing Systems*, 2019, pages 8252–8262 (cited on pages 6, 12, 69, 72, 75, 82).

[95] Gregory K Wallace. The JPEG still picture compression standard. In *IEEE transactions on consumer electronics* 38.1 (1992), pages xviii–xxxiv (cited on page 34).

[96] Zhou Wang and Alan C Bovik. Reduced-and no-reference image quality assessment. In *IEEE Signal Processing Magazine* 28.6 (2011), pages 29–40 (cited on page 84).

[97] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. In *IEEE transactions on image processing* 13.4 (2004), pages 600–612 (cited on page 70).

[98] Zhou Wang, Hamid R Sheikh, and Alan C Bovik. No-reference perceptual quality assessment of JPEG compressed images. In *IEEE International Conference on Image Processing*, volume 1. IEEE. 2002 (cited on page 93).

[99] Cihang Xie, Mingxing Tan, Boqing Gong, Jiang Wang, Alan Yuille, and Quoc V Le. Adversarial examples improve image recognition. In *arXiv preprint arXiv:1911.09665* (2019) (cited on page 15).

[100] Yu Xing, Shuang Liang, Lingzhi Sui, Xijie Jia, Jiantao Qiu, Xin Liu, Yushun Wang, Yi Shan, and Yu Wang. Dnnvm: end-to-end compiler leveraging heterogeneous optimizations on fpga-based cnn accelerators. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019) (cited on page 43).

[101] Yichong Xu, Tianjun Xiao, Jiaxing Zhang, Kuiyuan Yang, and Zheng Zhang. Scale-invariant convolutional neural networks. In *arXiv preprint arXiv:1411.6369* (2014) (cited on page 15).

[102] Eddie Yan, Kaiyuan Zhang, Xi Wang, Karin Strauss, and Luis Ceze. Customizing progressive JPEG for efficient image storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017 (cited on page 72).

[103] Brandon Yang, Gabriel Bender, Quoc V Le, and Jiquan Ngiam. Soft conditional computation. In *arXiv preprint arXiv:1904.04971* 3.4 (2019), page 5 (cited on page 12).

[104] Haichuan Yang, Yuhao Zhu, and Ji Liu. Energy-constrained compression for deep neural networks via weighted sparse projection and layer input masking. In *arXiv preprint arXiv:1806.04321* (2018) (cited on page 11).

[105] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: platform-aware neural network adaptation for mobile applications. In *The European Conference on Computer Vision (ECCV)*, Sept. 2018 (cited on page 44).

[106] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-bench-101: towards reproducible neural architecture search. In *Proceedings of the 36th International Conference on Machine Learning*, edited by Kamalika Chaudhuri and Ruslan Salakhutdinov. Volume 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, Sept. 2019, pages 7105–7114. URL: `http://proceedings.mlr.press/v97/ying19a.html` (cited on page 14).

[107] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *arXiv preprint arXiv:1611.03530* (2016) (cited on page 15).

[108] Richard Zhang. Making convolutional networks shift-invariant again. In *Proceedings of the 36th International Conference on Machine Learning*, edited by Kamalika Chaudhuri and Ruslan Salakhutdinov. Volume 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, Sept. 2019, pages 7324–7334. URL: http://proceedings.mlr.press/v97/zhang19a.html (cited on page 15).

[109] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pages 586–595 (cited on pages 70, 84).

[110] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: generating high-performance tensor programs for deep learning. In *arXiv preprint arXiv:2006.06762* (2020) (cited on pages 7, 13, 95).

[111] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016 (cited on page 24).

[112] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pages 2921–2929 (cited on page 70).

[113] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: training low bitwidth convolutional neural networks with low bitwidth gradients. In *arXiv preprint arXiv:1606.06160* (2016) (cited on pages 11, 44).

[114] Wei Zhu, Qiang Qiu, Robert Calderbank, Guillermo Sapiro, and Xiuyuan Cheng. Scale-equivariant neural networks with decomposed convolutional filters. In *arXiv preprint arXiv:1909.11193* (2019) (cited on page 15).

[115] zimg - A lightweight and high performance image storage and processing system. http://zimg.buaa.us/ (cited on page 13).

[116] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pages 8697–8710 (cited on pages 14, 43).