Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better

GAURAV MENGHANI, Google Research

Deep learning has revolutionized the fields of computer vision, natural language understanding, speech recognition, information retrieval, and more. However, with the progressive improvements in deep learning models, their number of parameters, latency, and resources required to train, among others, have all increased significantly. Consequently, it has become important to pay attention to these footprint metrics of a model as well, not just its quality. We present and motivate the problem of efficiency in deep learning, followed by a thorough survey of the five core areas of model efficiency (spanning modeling techniques, infrastructure, and hardware) and the seminal work there. We also present an experiment-based guide along with code for practitioners to optimize their model training and deployment. We believe this is the first comprehensive survey in the efficient deep learning space that covers the landscape of model efficiency from modeling techniques to hardware support. It is our hope that this survey would provide readers with the mental model and the necessary understanding of the field to apply generic efficiency techniques to immediately get significant improvements, and also equip them with ideas for further research and experimentation to achieve additional gains.

CCS Concepts: • **Computing methodologies** → *Artificial intelligence*; *Natural language processing*; *Computer vision*; *Machine learning;*

Additional Key Words and Phrases: Efficient deep learning, efficient machine learning, efficient artificial intelligence, quantization, pruning, sparsity, distillation, model compression, model optimization

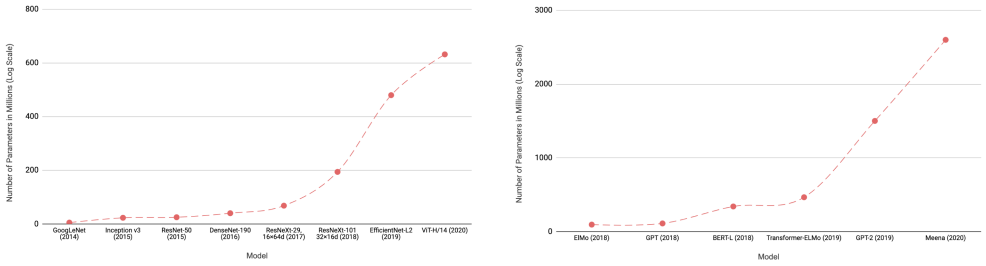
ACM Reference format:
Gaurav Menghani. 2023. Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better. *ACM Comput. Surv.* 55, 12, Article 259 (March 2023), 37 pages.
https://doi.org/10.1145/3578938

1 INTRODUCTION

Deep learning with neural networks has been the dominant methodology of training new machine learning models for the past decade. However, deep learning research has been focused on improving the state of the art, and progressive improvements on benchmarks like image classification [57, 124, 130] and text classification [21, 39, 138] have been correlated with an increase in the network complexity, number of parameters, amount of training resources required to train the network, prediction latency, and so on (Figure 1). For instance, GPT-3 comprises 175 billion parameters and costs millions of dollars to train just one iteration [21]. This excludes the cost of experimentation/trying combinations of different hyper-parameters, which is also computationally expensive.

Author's address: G. Menghani, Google Research, 1600 Amphitheatre Pkwy, Mountain View (CA) - 94043; email: gmenghani@google.com.

0360-0300/2023/03-ART259 $15.00
https://doi.org/10.1145/3578938



(a) Computer Vision Models (b) Natural Language Models

Fig. 1. Growth in the number of parameters in state-of-the-art computer vision and natural language models over time. Source: [104].

Although these models perform well on the tasks they are trained on, they might not necessarily be efficient enough for direct deployment in the real world. A deep learning practitioner might face the following challenges when training or deploying a model:

- *Sustainable server-side scaling*: Training and deploying large deep learning models is costly. Training could be a one-time cost (or could be free if one is using a pre-trained model); however, deploying and letting inference run over a long period of time could still turn out to be expensive in terms of consumption of server-side RAM, CPU, and so on. There is also a very real concern around the carbon footprint of datacenters even for organizations like Google, Facebook, and Amazon, which spend several billion dollars each per year in capital expenditure on their datacenters.
- *Enabling on-device deployment*: Certain deep learning applications need to run in real time on IoT and smart devices (where the model inference happens directly on the device), for a multitude of reasons (privacy, connectivity, responsiveness). Thus, it becomes imperative to optimize the models for the target devices.
- *Privacy and data sensitivity*: Being able to use as little data as possible for training is critical when the user data might be sensitive. Hence, efficiently training models with a fraction of the data means less data collection required.
- *New applications*: Certain new applications offer new constraints (around model quality or footprint) that existing off-the-shelf models might not be able to address.
- *Explosion of models*: Although a singular model might work well, training and/or deploying multiple models on the same infrastructure (colocation) for different applications might end up exhausting the available resources.

The common theme around the preceding challenges is *efficiency*, which is about asking if the given model achieves a competitive performance while keeping the training and inference costs fixed, or vice versa. Our goal with this survey article is to provide a detailed treatment on this subject. Specifically, the contributions are as follows:

- A mental model (Section 2) for thinking about efficiency in deep learning. This includes metrics, comparison between models, and five focus areas.
- An in-depth survey of the five focus areas (Section 3) covering algorithms, tools, and infrastructure relevant to efficient deep learning.
- A practitioner's guide to efficiency (Section 4) that provides insights into multiple possible approaches toward achieving the right tradeoffs between model quality and footprint using techniques covered in Section 3.

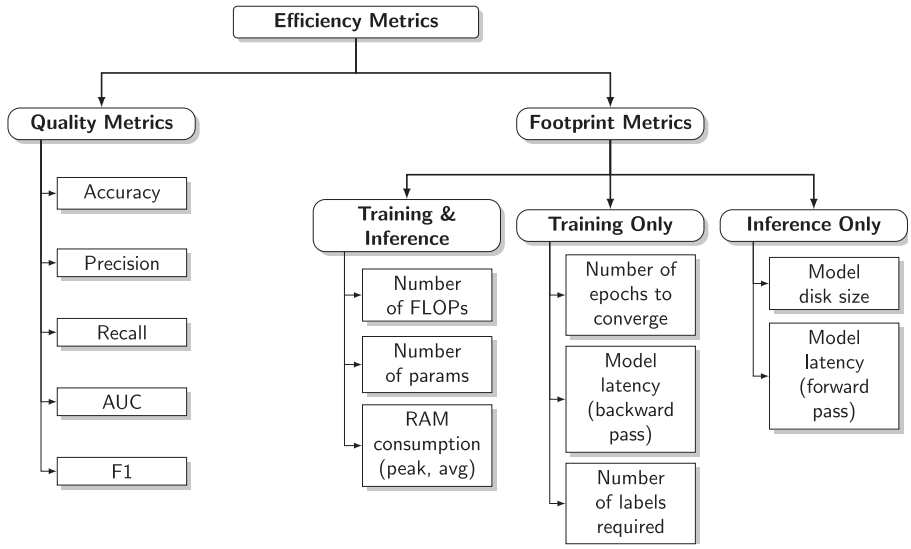


Fig. 2. A non-exhaustive taxonomy of efficiency metrics. To deploy a model, typically we can measure its feasibility via its performance (quality metrics like accuracy, precision, and recall) and cost (footprint metrics like model size, latency, and number of epochs to convergence). To compare any two given models for their relative efficiency, it is essential to compare both quality and footprint metrics.

- Insights from experiments performed using the recommendations in Section 4 with deployment and benchmarking on multiple devices. We also released an easy to reproduce IPython notebook for the preceding experiments.

There are other survey papers that cover a limited part of the field of efficient deep learning such as specific compression techniques like quantization and distillation [26, 94], architecture search [3], or hardware design [129]. However, in our findings, this is the first survey article that provides deep insights into the landscape of efficient deep learning, efficiency metrics and their tradeoffs, optimization techniques, and supporting tools and infrastructure, along with actionable tips to readers and practitioners backed by experiments.

2 A MENTAL MODEL

2.1 Efficiency Metrics

In the previous section, we motivated the problem of efficient deep learning. Although efficiency is fuzzily defined, we can somewhat formalize it using two groups of intertwined metrics around model quality and model footprint (refer to the efficiency metrics taxonomy in Figure 2):

- *Quality metrics*: These are metrics that we typically care about first, such as the model's accuracy, precision, and recall.
- *Footprint metrics*: These are cost indicators of training and deploying a model. These may be costs that we typically associate with a deep learning model such as size and latency. It can also include costs such as number of labels required to achieve the desired performance, which in many cases are expensive to obtain. We can further subdivide these costs into costs that apply only during training, or during inference, or both (see Figure 2).

If we were to be given two models, performing equally well on a given task (comparable quality metrics), we might want to choose a model that has lower footprint costs during training, inference,

or both (as appropriate for the given use case).[1] Deploying a model on devices with constrained (e.g., mobile and embedded devices) or expensive (cloud servers) compute resources requires paying attention to inference efficiency. Similarly, training large models with limited or costly training resources would require monitoring training efficiency.

Regardless of what one might be optimizing for, we want to achieve *Pareto-optimality*. This implies that any model we choose is the best for the tradeoffs we care about. As an example in Figure 3, the green dots represent Pareto-optimal models, where none of the other models (red dots) get better accuracy with the same inference latency, or the other way around. Together, the Pareto-optimal models (green dots) form our *Pareto-frontier* (denoted via the dashed lines in Figure 3). The models in the Pareto-frontier are by definition more efficient than the other models, since they perform the best for their given tradeoff. Hence, when we seek efficiency, we should be thinking about discovering and improving on the Pareto-frontier. In Section 4, we also provide suggestions on how to traverse the Pareto-frontier by exchanging some model quality for better model footprint, or vice versa.

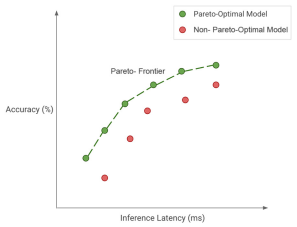


Fig. 3. Pareto-optimality. Green dots represent Pareto-optimal models (together forming the *Pareto-frontier*), where none of the other models (red dots) get better accuracy with the same inference latency, or the other way around.

2.2 Five Focus Areas

In this section, we present a survey of algorithms, techniques, tools, and infrastructure that work together to allow users to train and deploy Pareto-optimal models with respect to model quality and its footprint. We propose to structure them in five major areas, with the first four focused on modeling, and the final one around infrastructure, hardware, and tools (Figure 4):

(1) *Compression techniques*: These are general techniques and algorithms that look at optimizing the model's architecture, typically by compressing its layers. A classical example is quantization [67], which tries to compress the weight matrices of a layer, by reducing its precision (e.g., from 32-bit floating point values to 8-bit unsigned integers), with minimal loss in quality.
(2) *Learning techniques*: These are algorithms that focus on training the model differently (to make fewer prediction errors, require less data, converge faster, etc.). The improved quality can then be exchanged for a smaller footprint/a more efficient model by trimming the number of parameters if needed. An example of a learning technique is distillation [60], which allows improving the accuracy of a smaller model by learning to mimic a larger model.
(3) *Automation*: These are tools for improving the core metrics of the given model using automation. An example is **Hyper-Parameter Optimization (HPO)** [50], where optimizing the hyper-parameters helps increase the accuracy, which could then be exchanged for a model with fewer parameters. Similarly, architecture search [150] falls in this category too, where the architecture itself is tuned and the search helps find a model that optimizes both the loss/accuracy, and some other metric such as model latency or model size.
(4) *Efficient architectures*: These are fundamental blocks that were designed from scratch (convolutional layers, attention, etc.) and are a significant leap over the baseline methods used before them (**Fully Connected (FC)** layers and **Recurrent Neural Networks (RNNs)**,

[1]Comparing two different model training and inference setups might be tricky, as the metrics might hide implementation details. We encourage the reader to refer to the work of Dehghani [37] for a detailed treatment on this subject.

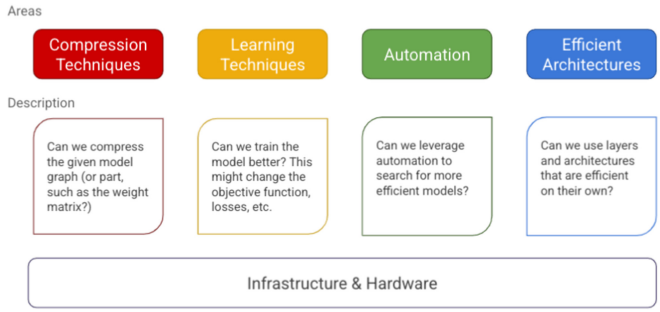


Fig. 4. Five focus areas of algorithms, techniques, and tools related to efficiency in deep learning. Compression techniques are aimed at compressing the model itself so that it has a smaller footprint. Learning techniques are focused on training the model differently so that it might require fewer resources (training steps, number of labels, etc.) to achieve the same performance. Automation tools help with optimizing model training and inference performance. Efficient architectures are hand designed with improvement in at least one of the efficient metrics in mind and can often be used as-is in models. Finally, infrastructure and hardware provide the necessary foundation for training and inference stages. We go over these five areas in detail in Section 2.2.

respectively). As an example, convolutional layers introduced parameter sharing for use in image classification, which avoids having to learn separate weights for each input pixel, and also makes them robust to overfitting. Similarly, attention layers [17] solved the problem of information bottleneck in **Sequence-to-Sequence (Seq2Seq)** models. These architectures can be used directly for efficiency gains.

(5) *Infrastructure*: Finally, we also need a foundation of infrastructure and tools that help us build and leverage efficient models. This includes the model training framework, such as TensorFlow [1] or PyTorch [105] (along with the tools required specifically for deploying efficient models like **TensorFlow Lite (TFLite)**, PyTorch Mobile, etc.). We depend on the infrastructure and tooling to leverage gains from efficient models. For example, to get both size and latency improvements with quantized models, we need the inference platform to support common neural network layers in quantized mode.

We survey each of these areas in depth in the following section.

3 LANDSCAPE OF EFFICIENT DEEP LEARNING

3.1 Compression Techniques

Compression techniques, as mentioned earlier, are usually generic techniques for achieving a more efficient representation of one or more layers in a neural network, with a possible quality trade-off. They help improve footprint metrics, such as model size, inference latency, and training time required for convergence, in exchange for as little quality loss as possible. In some cases, if the model is over-parameterized, these techniques can improve model generalization.

3.1.1 Pruning. Given a neural network $f(X, W)$, where X is the input and W is the set of parameters (or weights), pruning is a technique for coming up with a minimal subset W' such that the rest of the parameters of W are pruned (or set to 0), while ensuring that the quality of the model remains above the desired threshold (Figure 5). After pruning, we can say the network has been made *sparse*, where the sparsity can be quantified as the ratio of the number of parameters that were pruned to the number of parameters in the original network ($s = (1 - \frac{|W'|}{|W|})$). The higher the sparsity, the lesser the number of non-zero parameters in the pruned networks.

Some of the classical works in this area are **Optimal Brain Damage (OBD)** by LeCun et al. [82] and the Optimal Brain Surgeon paper by Hassibi et al. [56]. These methods usually take a network that has been pre-trained to a reasonable quality and then iteratively prune the parameters that have the lowest *saliency* score such that the impact on the validation loss is minimized. Once pruning concludes, the network is fine-tuned with the remaining parameters. This process is repeated a number of times until the desired number of original parameters are pruned (Algorithm 1).

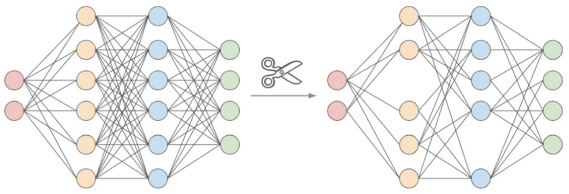


Fig. 5. A simplified illustration of pruning weights (connections) and neurons (nodes) in a neural network comprising FC layers.

OBD approximates the saliency score by using a second derivative of the parameters ($\frac{\partial^2 L}{\partial w_i^2}$), where L is the loss function and w_i is the candidate parameter for removal. The intuition is that the higher this value for a given parameter, the larger the change in the loss function's gradient if it were to be pruned.

For the purpose of speeding up the computation of the second derivatives, OBD ignores cross interaction between the weights ($\frac{\partial^2 L}{\partial w_i \partial w_j}$) and hence computes only the diagonal elements of the Hessian matrix. Otherwise, computing the full Hessian matrix is unwieldy for even a reasonable number of weights (with $n = 10^4$, the size of the matrix is $10^4 \times 10^4 = 10^8$). In terms of results, LeCun et al. [82] demonstrate that pruning reduced the parameters in a well-trained neural net by 8× (combination of both automatic and manual removal) without a drop in classification accuracy.

ALGORITHM 1: Standard network pruning with fine-tuning

Data: Pre-trained dense network with weights W, inputs X, number of pruning rounds N, fraction of parameters to prune per round p.
Result: Pruned network with weights W'.

```
W' ← W;
for i ← 1 to N do
    S ← compute_saliency_scores(W');
    W' ← W' − select_min_k(S, p|W|);
    W' ←fine_tune(X, W')
end
return W'
```

Across different pruning strategies, the core algorithm could remain similar, with changes in the following aspects (also refer to Table 1 for various results on the same MobileNet v2 architecture, which might be fair comparisons):

- *Saliency*: Although some methods [56, 82] use second-order derivatives, other methods rely on simpler magnitude based pruning [53, 54] or momentum based pruning [38], among others, to determine the saliency score.
- *Structured v/s unstructured*: The most flexible way of pruning is unstructured (or random) pruning, where all given parameters are treated equally. In structured pruning, parameters are pruned in blocks (pruning row-wise in a weight matrix, or pruning channel-wise in a convolutional filter [5, 84, 89, 98], etc.). The latter allows easier leveraging of inference-time gains in size and latency, since these blocks of pruned parameters can be intelligently skipped for storage and inference. Note that unstructured pruning can also be viewed as structured pruning with block size = 1.
- *Distribution*: The decision about how to distribute the sparsity budget (number of parameters to be pruned) could be made either by pooling in all the parameters from the network and

Table 1. Sample of Various Sparsity Results on the MobileNet v2 Architecture with Depth Multiplier = 1.0

Model Architecture	Sparsity Type	Sparsity %	FLOPs	Top-1 Accuracy %	Source
MobileNet v2-1.0	Dense (baseline)	0	$1 times$	72.0	Sandler et al. [117]
	Unstructured	75	$0.27 times$	67.7	Zhu and Gupta [149]
	Unstructured	75	$0.52 times$	71.9	Evci et al. [47]
	Structured (block-wise)	85	$0.11 times$	69.7	Elsen et al. [45]
	Unstructured	90	$0.12 times$	61.8	Zhu and Gupta [149]
	Unstructured	90	$0.12 times$	69.7	Evci et al. [47]

then deciding which parameters to prune, or by smartly selecting how much to prune in each layer individually [42, 58]. Elsen et al. [45] and Google Research [114] have found that some architectures like MobileNetV2 and EfficientNet [131] have thin first layers that do not contribute significantly to the number of parameters and pruning them leads to an accuracy drop without much gain. Hence, intuitively, it would be helpful to allocate sparsity on a per-layer basis.

- *Scheduling*: Another question is how much to prune, and when? Should we prune an equal number of parameters every round [54, 56, 82], or should we prune at a higher pace in the beginning and gradually decrease [38, 149]?
- *Regrowth*: Some methods allow regrowing pruned connections [38, 47] to keep the same level of sparsity through constant cycles of prune-redistribute-regrow. Dettmers and Zettlemoyer [38] estimate training time speedups between $2.7 times$ and $5.6 times$ by starting and operating with a sparse model throughout. However, there is a gap in terms of implementation of sparse operations on CPU, **Graphics Processing Unit (GPU)**, and other hardware.

3.1.2 Quantization. Almost all the weights and activations of a typical network are in 32-bit floating-point values. One of the ideas of reducing the model footprint is to reduce the precision for the weights and activations by *quantizing* to a lower-precision datatype (often 8-bit fixed-point integers). There are two kinds of gains that we can get from quantization: (1) lower model size and (2) lower inference latency. Often, only the model size is a constraint, and in this case we can employ a technique called *weight quantization* and get model size improvements [11], where only the model weights are in reduced precision. To get latency improvements, the activations need to be in fixed-point as well (activation quantization [67, 136]) such that all operations in the quantized graph are happening in fixed-point math as well.

Weight Quantization. A simple *scheme* for quantizing weights to get model size improvements (similar to Krishnamoorthi [75]) is as follows. Given a 32-bit floating-point weight matrix in a model, we can map the minimum weight value (x_{min}) in that matrix to 0 and the maximum value (x_{max}) to $2^b - 1$ (where b is the number of bits of precision, and $b < 32$). Then we can linearly extrapolate all values between them to an integer value in $[0, 2^b - 1]$ (Figure 6). Thus, we are able to map each floating-point value to a fixed-point value where the latter requires a lesser number of bits than the floating-point representation. This process can also be done for signed b-bit fixed-point integers, where the output values will be in the range $[-2^{\frac{b}{2}} - 1, 2^{\frac{b}{2}} - 1]$. One of the reasonable values of b is 8, since this would lead to a $32/8 = 4\times$ reduction in space, and also because of the near-universal support for `uint8_t` and `int8_t` datatypes.

During inference, we go in the reverse direction where we recover a lossy estimate of the original floating-point value (*dequantization*) using just x_{min} and x_{max}. This estimate is lossy since we lost $32 - b$ bits of information when did the rounding (another way to look at it is that a range of floating-point values map to the same quantized value).

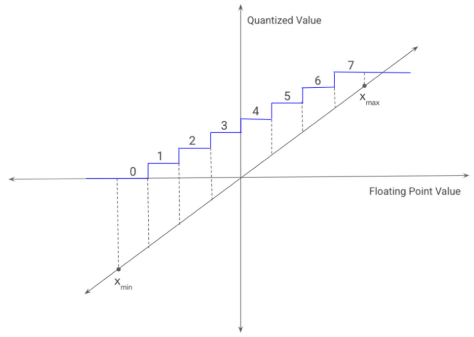


Fig. 6. Quantizing floating-point continuous values to discrete fixed-point values. The continuous values are clamped to the range x_{min} to x_{max}, and are mapped to discrete values in $[0, 2^b - 1]$ (in the figure, $b = 3$, hence the quantized values are in the range $[0, 7]$).

Jacob et al. [67] and Krishnamoorthi [75] formalize the quantization scheme with the following two constraints:

- The quantization scheme should be linear (affine transformation) so that the precision bits are linearly distributed.
- 0.0 should map exactly to a fixed-point value x_{q_0} such that dequantizing x_{q_0} gives us 0.0. This is an implementation constraint, since 0 is also used for padding to signify missing elements in tensors, and if dequantizing x_{q_0} leads to a non-zero value, then it might be interpreted incorrectly as a valid element at that index.

The second constraint described previously requires that 0 be a part of the quantization range, which in turn requires updating x_{min} and x_{max}, followed by clamping x to lie in $[x_{min}, x_{max}]$. Following this, we can quantize x by constructing a piece-wise linear transformation as follows:

$$\text{quantize}(x) = x_q = \text{round}\left(\frac{x}{s}\right) + z. \tag{1}$$

s is the floating-point *scale* value (can be thought of as the inverse of the slope, which can be computed using x_{min}, x_{max} and the range of the fixed-point values). z is an integer *zero-point* value, which is the quantized value that is assigned to $x = 0.0$. This is the terminology followed in the literature [67, 75] (Algorithm 2).

The dequantization step constructs $\hat{x}$, which is a lossy estimate of x, since we lose precision when quantizing to a lower number of bits. We can compute it as follows:

$$\text{dequantize}(x_q) = \hat{x} = s(x_q - z). \tag{2}$$

Since s is in floating-point, $\hat{x}$ is also a floating-point value (Algorithm 3). Note that the quantization and dequantization steps can be performed for signed integers too by appropriately changing the value $x_{q_{min}}$ (which is the lowest fixed-point value in b-bits) in Algorithm 2.

We can utilize Algorithms 2 and 3 for quantizing and dequantizing the model's weight matrices. Quantizing a pre-trained model's weights for reducing the size is termed as *post-training quantization* in the literature [11]. This might be sufficient for the purpose of reducing the model size when there is sufficient representational capacity in the model.

There are other works in the literature [65, 83, 110] that demonstrate slightly different variants of quantization. XNOR-Net [110], binarized neural networks [65], and others use $b = 1$ and thus have weight matrices that just have two possible values 0 or 1, and the quantization function there is simply the sign(x) function (assuming the weights are symmetrically distributed around 0).

ALGORITHM 2: Quantizing a given weight matrix $\mathbf{X}$

Data: Floating-point tensor to compress $\mathbf{X}$, number of precision bits b for the fixed-point representation.
Result: Quantized tensor $\mathbf{X_q}$.

$\mathbf{X}_{min}, \mathbf{X}_{max} \leftarrow \min(\mathbf{X}, 0), \max(\mathbf{X}, 0)$;
$\mathbf{X} \leftarrow \text{clamp}(\mathbf{X}, \mathbf{X}_{min}, \mathbf{X}_{max})$;
$s \leftarrow \frac{x_{max} - x_{min}}{2^b - 1}$;
$z \leftarrow \text{round}\left(x_{q_{min}} - \frac{x_{min}}{s}\right)$;
$\mathbf{X_q} \leftarrow \text{round}\left(\frac{\mathbf{X}}{s}\right) + z$;
return $\mathbf{X_q}$;

ALGORITHM 3: Dequantizing a given fixed-point weight matrix $\mathbf{X_q}$

Data: Fixed-point matrix to dequantize $\mathbf{X_q}$, along with the scale s, and zero-point z values that were calculated during quantization.
Result: Dequantized floating-point weight matrix $\widehat{\mathbf{X}}$.

$\widehat{\mathbf{X}} \leftarrow s(\mathbf{X_q} - z)$;
return $\widehat{\mathbf{X}}$;

The promise with such extreme quantization approaches is the theoretical $32/1 = 32\times$ reduction in model size without much quality loss. Some of the works claim improvements on larger networks like AlexNet [77], VGG [124], and Inception [130], which might already be more amenable to compression. A more informative task would be to demonstrate extreme quantization on smaller networks like the MobileNet family [62, 117]. Additionally, binary quantization (and other quantization schemes like ternary [83], bit-shift based networks [110], etc.) promise latency-efficient implementations of standard operations where multiplications and divisions are replaced by cheaper operations like addition and subtraction. These claims need to be verified because even if these lead to theoretical reduction in FLOPs, the implementations still need support from the underlying hardware. A fair comparison would be using standard quantization with $b = 8$, where the multiplications and divisions also become cheaper, and are supported by the hardware efficiently via SIMD instructions that allow for low-level data parallelism (e.g., on x86 via the SSE instruction set, on ARM via the Neon [90] intrinsics, and even on specialized DSPs like the Qualcomm Hexagon [16]).

Activation Quantization. To be able to get *latency improvements* with quantized networks, the math operations have to be done in fixed-point representations too. This means all intermediate layer inputs and outputs are also in fixed-point, and there is no need to dequantize the weight matrices since they can be used directly along with the inputs.

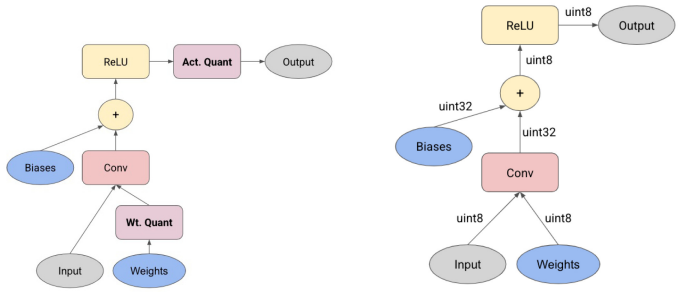
Vanhoucke et al. [136] demonstrated a 3× inference speedup using a fully fixed-point model on an x86 CPU, when compared to a floating-point model on the same CPU, without sacrificing accuracy. The weights are still quantized similar to post-training quantization; however, all layer inputs (except the first layer) and the activations are fixed-point. In terms of performance, the primary driver for this improvement was the availability of fixed-point SIMD instructions in Intel's SSE4 instruction set [35], where commonly used building-block operations like the **Multiply-Accumulate (MAC)** [34] can be parallelized. Since the paper was published, Intel has released two more iterations of these instruction sets [31], which might further improve the speedups.

Quantization-Aware Training. The network that Vanhoucke et al. [136] mention was a five-layer feed-forward network that was post-training quantized. However, post-training quantization can lead to quality loss during inference as highlighted in other works [67, 75, 139] as the networks become more complex. These could be because of (1) outlier weights that skew the computation of the quantized values for the entire input range toward the outliers, leading to fewer bits being allocated to the bulk of the range, or (2) different distribution of weights within the weight matrix—for example, within a convolutional layer, the distribution of weights between each filter might be

Table 2. Sample of Various Quantization Results on the MobileNet v2 Architecture for 8-Bit Quantization

Model Architecture	Quantization Type	Top-1 Accuracy %	Size (MB)	Latency (ms, Pixel2)
MobileNet v2-1.0 (224)	Baseline	71.9	14	89
	Post-training quantization	63.7	3.6	98
	Quantization-aware training	70.9	3.6	54

We picked results with 8-bit quantization, as they can be readily used with hardware and software that exists today. From TensorFlow [133].



(a) Quantization-Aware Training (b) Final fixed-point inference graph

Fig. 7. State-of-the-art computer vision and natural language models shown over time. (a) The injection of fake-quantization nodes to simulate quantization effect and collecting tensor statistics, for exporting a fully fixed-point inference graph (which is shown in (b)). Source: [67, 75].

different, but they are quantized the same way. These effects might be more pronounced at low bit widths due to an even worse loss of precision.

Jacob et al. [67] propose (and further detailed by Krishnamoorthi [75]) a training regime that is *quantization-aware*. In this setting, the training happens in floating-point but the forward-pass simulates the quantization behavior during inference. Both weights and activations are passed through a function that simulates this quantization behavior (*fake-quantized* is the term used by many works [67, 75]) (Figure 7).

Quantization-Aware Training (QAT) allows the network to adapt to tolerate the noise introduced by the clamping and rounding behavior during inference. Once the network is trained, tools such as the TFLite Model Converter [12] can generate the appropriate fixed-point inference model from a network annotated with the quantization nodes.

Results. Table 2 presents a comparison between the baseline floating-point model, post-training quantized, and QAT models [11]. The model with post-training quantization gets close to the baseline, but there is still a significant accuracy difference. The model size is 4× smaller, but the latency is slightly higher due to the need to dequantize the weights during inference. The model with 8-bit QAT gets quite close to the baseline floating-point model while requiring 4× less disk space and being 1.64× faster. To summarize, we recommend evaluating standard 8-bit QAT for your networks, since the resulting model quality is comparable to floating-point models, and there is broad support for faster inference with 8-bit models.

3.1.3 Other Compression Techniques. There are other compression techniques like low-rank decomposition [146] and weight sharing via k-means clustering [53], which are also actively being used for model compression [103] and might be suitable for further compressing hotspots in a model.

3.2 Learning Techniques

Learning techniques try to train a model differently to obtain better quality metrics (accuracy, F1 score, precision, recall, etc.). The improvement in quality can sometimes be traded off for a smaller footprint by reducing the number of parameters/layers in the model and achieving the same baseline quality with a smaller model (see Section 4). An incentive of paying attention to learning techniques is that they need to be applied only during the training phase, without impacting the inference.

3.2.1 Distillation. Ensembles are well known to help with generalization [55, 78]. The intuition is that this enables learning multiple independent hypotheses, which are likely to be better than learning a single hypothesis. Dietterich [40] goes over some of the standard ensembling methods, such as bagging (learning models that are trained on non-overlapping data and then ensembling them), boosting (learning models that are trained to fix the classification errors of other models in the ensemble), and averaging (voting by all the ensemble models). Buciluǎ et al. [22] used large ensembles to label synthetic data that they generated using various schemes. A smaller neural net is then trained to learn not just from the labeled data but also from this weakly labeled synthetic data. They found that single neural nets were able to mimic the performance of larger ensembles, while being $1,000\times$ smaller and faster. This demonstrated that it is possible to transfer the cumulative knowledge of ensembles to a single small model. However, it might not be sufficient to rely on just the existing labeled data.

Hinton et al. [60], in their seminal work, explored how smaller networks (students) can be taught to extract 'dark knowledge' from larger models/ensembles of larger models (teachers) in a slightly different manner. Instead of having to generate synthetic data, they use the larger teacher model to generate *soft labels* on existing labeled data. The soft labels assign a probability to each class instead of hard binary values in the original data. The intuition is that these soft labels capture the relationship between the different classes the model can learn from. For example, a truck is more similar to a car than to an apple, which the model might not be able to learn directly from hard labels. The student network learns to minimize the cross-entropy loss on these soft labels, along with the original ground truth hard labels. Since the probabilities of the incorrect classes might be very small, the logits are scaled down by a *temperature* value ≥ 1.0 so that the distribution is *softened*. If the input vector is $\mathbf{X}$, and the teacher model's logits are $\mathbf{Z^{(t)}}$, the teacher model's softened probabilities with temperature T can be calculated as follows using the familiar softmax function:

$$\mathbf{Y}_i^{(t)} = \frac{\exp(\mathbf{Z_i^{(t)}}/T)}{\sum_{j=1}^{n} \exp(\mathbf{Z_j^{(t)}}/T)}. \tag{3}$$

Note that as T increases, the relative differences between the various elements of $Y^{(t)}$ decreases. This happens because if all elements are divided by the same constant, the softmax function would lead to a larger drop for the bigger values. Hence, as the temperature T increases, we see the distribution of $Y^{(t)}$ soften further. When training along with labeled data ($\mathbf{X}$, $\mathbf{Y}$), and the student model's output ($\mathbf{Y^{(s)}}$), we can describe the loss function as

$$\begin{aligned} L &= \lambda_1 \cdot L_{\text{ground-truth}} + \lambda_2 \cdot L_{\text{distillation}} \\ &= \lambda_1 \cdot \text{CrossEntropy}(\mathbf{Y}, \mathbf{Y^{(s)}}; \theta) + \lambda_2 \cdot \text{CrossEntropy}(\mathbf{Y^{(t)}}, \mathbf{Y^{(s)}}; \theta). \end{aligned} \tag{4}$$

CrossEntropy is the cross-entropy loss function, which takes in the labels and the output, and θ are the student model's parameters (the teacher model's parameters remain frozen). For the first loss term, we pass along the ground truth labels, and for the second loss term, we pass the

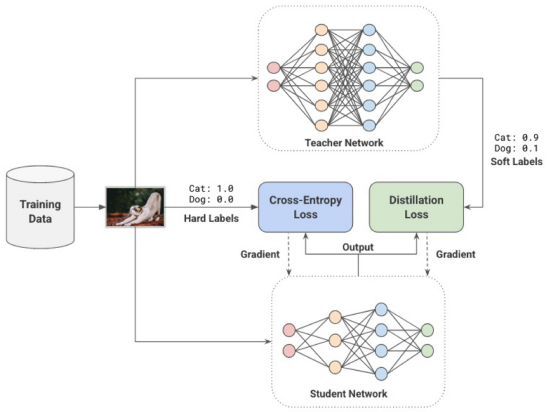


Fig. 8. Distillation of a smaller student model from a larger pre-trained teacher model.

corresponding soft labels from the teacher model for the same input. λ_1 and λ_2 control the relative importance of the standard ground truth loss and the distillation loss, respectively. When $\lambda_1 = 0$, the student model is trained with just the distillation loss. Similarly, when $\lambda_2 = 0$, it is equivalent to training with just the ground truth labels. Usually, the teacher network is pre-trained and frozen during this process, and only the student network is updated (Figure 8).

Hinton et al. [60] were able to closely match the accuracy of a 10-model ensemble for a speech recognition task with a single distilled model. Urban et al. [135] did a comprehensive study demonstrating that distillation significantly improves performance of shallow student networks as small as a multi-layer perceptron with one hidden layer on tasks like CIFAR-10. Sanh et al. [118] use the distillation loss for compressing a BERT [39] model (along with a cosine loss that minimizes the cosine distance between two internal vector representations of the input as seen by the teacher and student models). Their model retains 97% of the performance while being 40% smaller and 60% faster on CPU.

It is possible to adapt the general idea of distillation to work on intermediate outputs of teachers and students. Zagoruyko and Komodakis [147] transfer intermediate *attention maps* between teacher and student convolutional networks. The intuition is to make the student focus on the parts of the image the teacher is paying attention to. MobileBERT [127] uses a progressive-knowledge transfer strategy where they do layer-wise distillation between the BERT student and teacher models, but they do so in stages, where the first l layers are distilled in the l-th stage. Along with other architecture improvements, they obtain a 4.3× smaller and 5.5× faster BERT with small losses in quality. Another idea that has been well explored is exploiting a model trained in a supervised learning regime to label unlabeled data. Blum and Mitchell [19], in their paper from 1998, report halving the error rate of their classifiers by retraining on a subset of pseudo-labels generated using the previous classifiers. This has been extended through distillation to use the teacher model to label a large corpus of unlabeled data, which can then be used to improve the quality of the student model [92, 142, 143].

Overall, distillation has been empirically shown to improve both the accuracy (quality) and the speed of convergence of student models across many domains (training footprint). Hence, it enables training smaller models that might otherwise not have an acceptable quality for deployment.

3.2.2 Data Augmentation. When training large models for complex tasks in a supervised learning regime, the size of the training data corpus correlates with improvement in generalization.

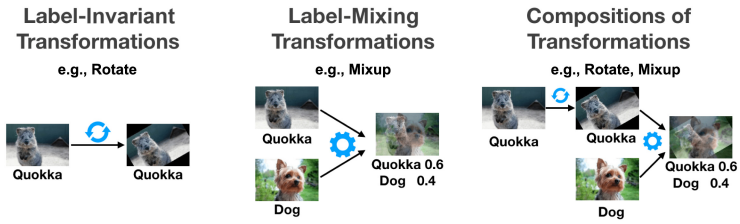


Fig. 9. Some common types of data augmentation. Source: [86].

Sun et al. [126] demonstrate logarithmic increase in the prediction accuracy with increase in the number of labeled examples. However, getting high-quality labeled data often requires a human in the loop and could be expensive, which is also why we listed the number of label examples required as a footprint metric to be potentially optimized in Section 2.

Data augmentation is a nifty way of addressing the scarcity of labeled data, by synthetically inflating the existing dataset through some *augmentation methods*. These augmentation methods are transformations that can be applied cheaply on the given examples such that the new label of the augmented example does not change, or can be cheaply inferred. As an example, consider the classical image classification task of labeling a given image to be a cat or a dog. Given an image of a dog, translating the image horizontally/vertically by a small number of pixels, rotating it by a small angle, and so forth would not materially change the image, so the transformed image should still be labeled as 'dog' by the classifier. This forces the classifier to learn a robust representation of the image that generalizes better across these transformations.

The transformations described earlier have long been demonstrated to improve accuracy of convolutional networks [30, 123]. They have also been a core part of seminal works in image classification. A prime example is AlexNet [77], where such transformations were used to increase the effective size of the training dataset by 2048×, which won the ImageNet competition in 2012. Since then, it has become common to use such transformations for image classification models (Inception [130], XCeption [27], ResNet [57], etc.).

We can categorize data augmentation methods as follows (Figure 9):

- *Label-invariant transformations*: These are some of the most common transformations, where the transformed example retains the original label. These can include simple geometric transformations such as translation, flipping, cropping, rotation, distortion, scaling, and shearing. However the user has to verify the label-invariance property with each transformation for the specific task at hand. Table 3 lists a few label-invariant transforms and their respective contributions in the improvement of validation accuracy for a model.
- *Label-mixing transformations*: Transformations such as Mixup [148] mix inputs from two different classes in a weighted manner and treat the label to be a correspondingly weighted combination of the two classes (in the same ratio). The intuition is that the model should be able to extract out features that are relevant for both classes. Other transformations like Sample [66] also seem to help.
- *Data-dependent transformations*: In this case, transformations are chosen such that they maximize the loss for that example [48], or are adversarially chosen so as to fool the classifier [52].
- *Synthetic sampling*: These methods synthetically create new training examples. Algorithms like SMOTE [25] allow rebalancing the dataset to make up for skew in the datasets, and GANs can be used to synthetically create new samples [149] to improve model accuracy.
- *Composition of transformations*: These are transformations that are themselves composed of other transformations, and the labels are computed depending on the nature of transformations that are stacked.

Table 3. Breakdown of the Impact of Different Image Transformations on the Validation Accuracy of a Model Trained on the CIFAR-10 Dataset

Transformation	Validation Accuracy Improvement (%)
rotate	1.3
shear-x	0.9
shear-y	0.9
translate-x	0.4
translate-y	0.4
sharpness	0.1
autoContrast	0.1

From Cubuk et al. [36].

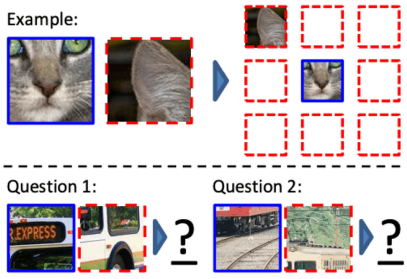
To summarize, data augmentation helps improve model efficiency by reaching a better performance point with the same number of labels, or alternatively achieving the same performance with fewer labels, thus improving the labeling footprint. One can also trade the model quality gains from data augmentation for a smaller model size by applying any other compression technique on top (this is discussed in detail in Section 4).

3.2.3 Self-Supervised Learning. The supervised learning paradigm relies heavily on labeled data. As mentioned earlier, it requires human intervention, and it is expensive as well. To achieve reasonable quality on a non-trivial task, the amount of labeled data required is large too. Although techniques like data augmentation and distillation help, they too rely on the presence of some labeled data to achieve a baseline performance.

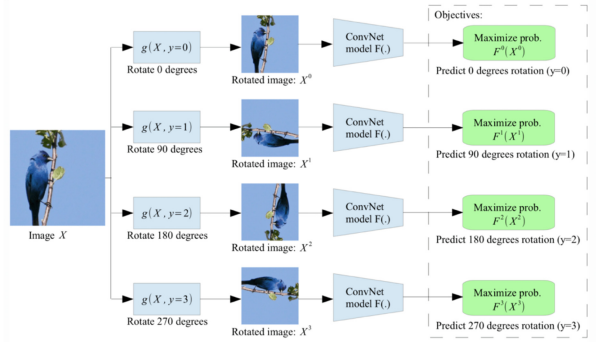
Self-supervised learning avoids the need for labeled data to learn generalized representations by aiming to extract more supervisory bits from each example. Since it focuses on learning robust representations of the example itself, it does not need to focus narrowly on the label. This is typically done by solving a *pretext task* where the model pretends that a part/structure of the input is missing and learns to predict it. Since unlabeled data is vast in many domains (books, Wikipedia, and other text for **Natural Language Understanding (NLU)**, web images and videos for computer vision, etc.), the model would not be bottlenecked by data for learning to solve these pretext tasks.

Once the models learn generic representations that transfer well across tasks, they can be adapted to solve the target task by adding some layers that project the representation to the label space, and fine-tuning the model with the labeled data. Since the labeled data is not being used for learning rudimentary features but rather how to map the high-level representations into the label space, the quantum of labeled data is going to be a fraction of what would have been required for training the model from scratch. From this lens, fine-tuning models pre-trained with self-supervised learning are *data-efficient* (they converge faster, attain better quality for the same amount of labeled data when compared to training from scratch, etc.) [39, 63].

An example of this two-step process of pre-training on unlabeled data and fine-tuning on labeled data has gained rapid acceptance in the community. ULMFiT (Universal Language Model Finetuning) [63] pioneered the idea of training a general-purpose language model, where the model learns to solve the pretext task of predicting the next word in a given sentence, without the need of an associated label. The authors found that using a large corpus of preprocessed unlabeled data such as the WikiText-103 dataset (derived from English Wikipedia pages) was a good choice for the pre-training step. This was sufficient for the model to learn general properties about the language, and the authors found that fine-tuning such a pre-trained model for a binary classification problem



(a) Detecting relative order of patches.



(b) Predicting the degree of rotation of a given image.

Fig. 10. Examples of pretext tasks for vision problems. Source: [41].

(IMDb dataset) required only 100 labeled examples ($\approx 10\times$ less labeled examples otherwise). If we add a middle step of pre-training using unlabeled data from the same target dataset, the authors report needing $\approx 20\times$ fewer labeled examples.

This idea of pre-training followed by fine-tuning is also used in BERT [39] (and other related models like GPT, RoBERTa, and T5) where the pre-training steps involve learning to solve two tasks. First, the masked language model where about 15% of the tokens in the given sentence are masked and the model needs to predict the masked token. The second task is, given two sentences A and B, to predict if B follows A. The pre-training loss is the mean of the losses for the two tasks. Once pre-trained, the model can then be used for classification or Seq2Seq tasks by adding additional layers on top of the last hidden layer [63]. When it was published, BERT beat the state of the art on 1 NLU tasks.

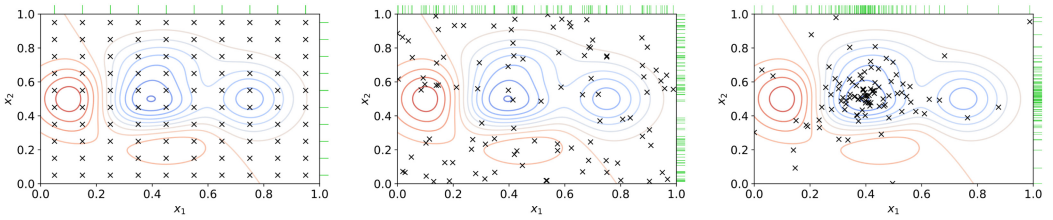
Similar to NLU, the pretext tasks in vision have been used to train models that learn general representations. Doersch et al. [41] extract two patches from a training example and then train the model to predict their relative position in the image (Figure 10(a)). They demonstrate that using a network pre-trained in this fashion improves the quality of the final object detection task, as compared to randomly initializing the network. Similarly, another task is to predict the degree of rotation for a given rotated image [49]. The authors report that the network trained in a self-supervised manner this way can be fine-tuned to perform nearly as well as a fully supervised network.

Just like data augmentation, self-supervised learning helps with training data-efficient models that can be trained with fewer labels to reach the same/better performance. The improved performance as mentioned can be combined with compression techniques to reduce the model footprint as well (please refer to Section 4).

3.3 Automation

It is possible to delegate some of the work around efficiency to automation and letting automated approaches search for ways of training more efficient models. Apart from reducing work for humans, it also lowers the bias that manual decisions might introduce in model training. The tradeoff is that these methods might require large computational resources and hence have to be carefully applied. Automation methods can be designed to improve either footprint metrics, quality metrics, or both.

3.3.1 Hyper-Parameter Optimization. One of the commonly used methods that fall under this category is HPO [145]. Hyper-parameters such as initial learning rate and weight decay have to



(a) Grid Search (b) Random Search (c) Bayesian Optimization

Fig. 11. An illustration of some common hyper-parameter search algorithms with two variables x_1 and x_2. The x marks a trial picked by the HPO algorithm. The contours are color coded, where the red shades imply a higher loss, and blue shades imply a lower loss. Source: [33].

be carefully tuned for faster convergence [70]. They can also decide the network architecture such as the number of FC layers, number of filters in a convolutional layer, and so forth.

Experimentation can help us build an intuition for the *range* in which these parameters might lie, but finding the best values requires a search for the exact values that optimize the given objective function (typically the loss value on the validation set). Manually searching for these quickly becomes tedious with the growth in the number of hyper-parameters and/or their possible values. Hence, let us explore possible algorithms for automating the search. To formalize this, let us assume, without the loss of generalization, that we are optimizing the loss value on the given dataset's validation split. Then, let $\mathcal{L}$ be the loss function, f be the model function that is learned with the set of hyper-parameters (λ), x be the input, and θ be the model parameters. With the search, we are trying to find λ^* such that

$$\lambda^* = \underset{\lambda \in \Lambda}{\operatorname{argmin}} \mathcal{L}(f_\lambda(x; \theta), y). \tag{5}$$

Λ is the set of all possible hyper-parameters. In practice, Λ can be a very large set containing all possible combinations of the hyper-parameters, which would often be intractable since hyper-parameters like learning rate are real valued. A common strategy is to approximate Λ by picking a finite set of *trials*, $S = \{\lambda^{(1)}, \lambda^{(2)}, \ldots, \lambda^{(n)}\}$, such that $S \in \Lambda$, and then we can approximate Equation (5) with

$$\lambda^* \approx \underset{\lambda \in \{\lambda^{(1)}, \ldots, \lambda^{(n)}\}}{\operatorname{argmin}} \mathcal{L}(f_\lambda(x; \theta), y). \tag{6}$$

As we see, the choice of S is crucial for the approximation to work. The user has to construct a range of reasonable values for each hyper-parameter $\lambda_i \in \lambda$. This can be based on prior experience with those hyper-parameters.

A simple algorithm for automating HPO is *grid search* (also referred to as parameter sweep), where S consists of all the distinct and valid combinations of the given hyper-parameters based on their specified ranges (Figure 11). Each trial can then be run in parallel since each trial is independent of the others, and the optimal combination of the hyper-parameters is found once all the trials have completed. Since this approach tries all possible combinations, it suffers from the *curse of dimensionality*, where the total number of trials grows very quickly.

Another approach is *random search*, where trials are sampled randomly from the search space [18]. Since each trial is independent of the others, it can still be executed in parallel. However, there are few critical benefits of random search:

(1) Since the trials are i.i.d. (not the case for grid search), the resolution of the search can be changed on-the-fly (if the computational budget has changed, or certain trials have failed).

(2) The likelihood of finding the optimal λ^* increases with the number of trials, which is not the case with grid search.
(3) If there are K real-valued hyper-parameters, and N total trials, grid search would pick $N^{\frac{1}{K}}$ trials for each hyper-parameter. However, not all hyper-parameters might be important. Random search picks a random value for each hyper-parameter per trial. Hence, in cases with low effective dimensionality of the search space, random search performs better than grid search.

Bayesian optimization based search [2, 97] is a *model-based* sequential approach where the search is guided by actively estimating the value of the objective function at different points in the search space, then spawning trials based on the information gathered so far. The estimation of the objective function is done using a *surrogate function* that starts off with a prior estimate. The trials are created using an *acquisition function* that picks the next trial using the surrogate function, the likelihood of improving on the optimum so far, whether to explore/exploit, and so on. As the trials complete, both of these functions will refine their estimates. Since the method keeps an internal model of how the objective function looks and plans the next trials based on that knowledge, it is model based. Additionally, since the selection of trials depends on the results of the past trials, this method is sequential. Bayesian optimization improves over random search in that the search is guided rather than random, thus fewer trials are required to reach the optimum. However, it also makes the search sequential (although it is possible to run multiple trials in parallel, overall it will lead to some wasted trials).

One of the strategies to save training resources with the preceding search algorithms is the *early stopping* of trials that are not promising. Google's Vizier [50] uses the median stopping rule for early stopping, where a trial is terminated if its performance at a time step t is below the the median performance of all trials rununtil that point in time. Other algorithms for HPO include the following:

(1) *Population-based training* [68]: This method is similar to evolutionary approaches like genetic algorithms, where a fixed number of trials (referred to as the population) are spawned and trained to convergence. Each trial starts with a random set of hyper-parameters and is trained to a pre-determined number of steps. At this point, all trials are paused, and every trial's weights and parameters might be replaced by the weights and parameters from the 'best' trial in the population so far (the *exploitation* phase). For *exploration*, these hyper-parameters are perturbed from their original values. This process repeats until convergence. It combines both the search and training in a fixed number of trials, which run in parallel. It also only works with adaptive hyper-parameters like learning rate, weight-decay, and so forth, but it cannot be used where hyper-parameters change the model structure. Note that the criteria for picking the *best* trial does not have to be differentiable.
(2) *Multi-armed bandit algorithms*: Methods like successive halving [69] and hyper-band [85] are similar to random search, but they allocate more resources to the trials performing well. Both of these methods need the user to specify the total computational budget B for the search (can be the total number of epochs of training, for instance). They then spawn and train a fixed number of trials with randomly sampled hyper-parameters while allocating the training budget. Once the budget is exhausted, the worst-performing fraction ($\frac{\eta-1}{\eta}$) of the trials are eliminated, and the remaining trials' new budget is multiplied by η. In the case of successive halving, η is 2, so the bottom $\frac{1}{2}$ of the trials are dropped, and the training budget for the remaining trials is doubled.

3.3.2 Neural Architecture Search. **Neural Architecture Search (NAS)** can be thought of an extension of HPO wherein we are searching for parameters that change the network architecture itself. We find that there is consensus in the literature [46] around categorizing NAS as a system comprising the following parts:

(1) *Search space*: These are the operations that are allowed in the graph (convolution ($1 \times 1, 3 \times 3, 5 \times 5$), densely connected, pooling, etc.), as well as the semantics of how these operations and their outputs connect to other parts of the network.
(2) *Search algorithm and state*: This is the algorithm that controls the architecture search itself. Typically, the standard algorithms that apply in HPO (grid search, random search, Bayesian optimization, evolutionary algorithms) can be used for NAS as well. However, using reinforcement learning [150] and gradient descent [88] are popular alternatives too.
(3) *Evaluation strategy*: This defines how we evaluate a model for fitness. It can simply be a conventional metric, such as validation loss or accuracy, or it can also be a compound metric, as in the case of MNasNet [131], which creates a single custom metric based on accuracy as well as latency.

The user is supposed to either explicitly or implicitly encode the search space. Together with the search algorithm, we can view this as a *controller* that generates sample candidate networks (Figure 12). The evaluation stage will then train and evaluate these candidates for fitness. This fitness value is then passed as feedback to the search algorithm, which will use it for generating better candidates. Although the implementation of each of these blocks varies, this structure is common across the seminal work in this area.

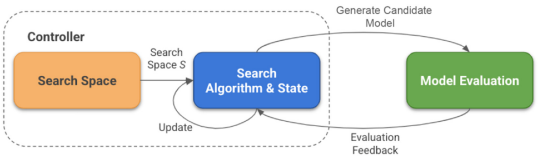


Fig. 12. Neural architecture search. The controller can be thought of as a unit that encodes the search space, the search algorithm itself, and the state it maintains. The algorithm generates candidate models in the search space S and receives an evaluation feedback to generate better candidate models.

Zoph and Le [150] demonstrate that end-to-end neural network architectures can be generated using reinforcement learning. In this case, the controller is an RNN that generates the architectural hyper-parameters of a feed-forward network one layer at a time (number of filters, stride, filter size, etc). They also support adding skip connections. The network semantics are baked into the controller, so generating a network that behaves differently requires changing the controller. Additionally, training the controller itself is expensive (taking 22,400 GPU hours [151]), since the entire candidate network has to be trained from scratch for a single gradient update to happen. In a follow-up work, Zoph et al. [151] come up with a refined search space where instead of searching for the end-to-end architecture, they search for *cells*: a 'normal cell' that takes in an input, processes it, and returns an output of the same spatial dimensions, and a 'reduction cell' that processes its input and returns an output whose spatial dimensions are scaled down by a factor of 2. Each cell is a combination of B blocks. The controller's RNN generates one block at a time, where it picks outputs of two blocks in the past, the respective operations to apply on them, and how to combine them into a single output. The normal and reduction cells are stacked in alternating fashion (N normal cells followed by one reduction cell, where N is tunable) to construct an end-to-end network for CIFAR-10 and ImageNet. Learning these cells individually rather than learning the entire network seems to improve the search time by 7×, when compared to the end-to-end network search in the work of Zoph et al. [150], while beating the state of the art in CIFAR-10 at that time.

Other approaches such as evolutionary techniques [113], differentiable architecture search [88], progressive search [87], and parameter sharing [107] try to reduce the cost of architecture search (in some cases reducing the compute cost to a couple of GPU days instead of thousands of GPU days). These are covered in detail in the work of Elsken et al. [46].

Most of the early papers focused on finding the architectures that performed best on quality metrics like accuracy, unconstrained by the footprint metrics. However, when focusing on efficiency, we are often interested in specific tradeoffs between quality and footprint. Architecture search can help with multi-objective searches that optimize for both quality and footprint. MNasNet [131] is one such work. It incorporates the model's latency on the target device into the objective function directly, as follows:

$$\underset{m}{\text{maximize}} \quad ACC(m) \times \left[\frac{LAT(m)}{T}\right]^{w}, \tag{7}$$

where m is the candidate model, ACC is the accuracy metric, and LAT is the latency of the given model on the desired device. T is the target latency. w is recommended to be -0.07. FBNet [141] uses a similar approach with a compound reward function that has a weighted combination of the loss value on the validation set and the latency. However, instead of measuring the latency of the candidate model on a device, they use a pre-computed lookup table to approximate the latency to speed up the search process. They achieve networks that are up to 2.4× smaller and 1.5× faster than MobileNet [117], while finishing the search in 216 GPU hours. Other works such as MONAS [64] use reinforcement learning to incorporate power consumption into the reward function along with hard constraints on the number of MAC operations in the model, and discover Pareto-frontiers under the given constraints.

3.4 Efficient Architectures

Another common theme for tackling efficiency problems is to go back to the drawing board, and design layers and models that are efficient by design to replace the baseline. They are typically designed with some insight that might lead to a design that is better in general, or it might be better suited for the specific task, and lead to better footprint or quality metrics, or both. In this section, we lay out examples of such efficient layers and models to illustrate this idea.

3.4.1 Vision. One of the classical examples of efficient layers in the vision domain are the convolutional layers, which improved over FC layers in vision models. FC layers suffer from two primary issues:

(1) First, FC layers ignore the spatial information of the input pixels. Intuitively, it is hard to build an understanding of the given input by looking at individual pixel values in isolation. They also ignore the spatial locality in nearby regions.
(2) Second, using FC layers also leads to an explosion in the number of parameters when working with even moderately sized inputs. A 100×100 RGB image with three channels would lead to each neuron in the first layer having 3×10^4 connections. This makes the network susceptible to overfitting also.

Convolutional layers avoid this by learning 'filters,' where each filter is a 3D weight matrix of a fixed size (3×3, 5×5, etc.), with the third dimension being the same as the number of channels in the input. Each filter is convolved over the input to generate a feature map for that given filter. These filters learn to detect specific features, and convolving them with a particular input patch results in a single scalar value that is higher if the feature is present in that input patch.

Hence, the core idea behind the efficiency of these is that the same filter is used everywhere in the image, regardless of where the filter is applied (enforcing spatial invariance while sharing the

parameters). Going back to the example of a 100×100 RGB image with three channels, a 5×5 filter would imply a total of 75 ($5 \times 5 \times 3$) parameters. Each layer can learn multiple unique filters and still be within a very reasonable parameter budget. This also has a regularizing effect, wherein a dramatically reduced number of parameters allows for easier optimization, and reducing the likelihood of overfitting.

Convolutional layers are usually coupled with pooling layers, which allow dimensionality reduction by subsampling the input (aggregating a sliding 2D window of pixels, using functions like max, avg, etc.). Pooling would lead to smaller feature maps for the next layer to process, which makes it faster to process. LeNet5 [81] was the first convolutional network that included convolutional layers, pooling, and so on. Subsequently, many iterations of these networks have been proposed with various improvements. AlexNet [77], Inception [130], ResNet [57], and others have all made significant improvements over time on known image classification benchmarks using convolutional layers.

Depth-Separable Convolutional Layers. In the convolution operation, each filter is used to convolve over the two spatial dimensions and the third channel dimension. As a result, the size of each filter is $s_x \times s_y \times$ `input_channels`, where s_x and s_y are typically equal. This is done for each filter, resulting in the convolution operation happening both spatially in the x and y dimensions, and depth-wise in the z dimension.

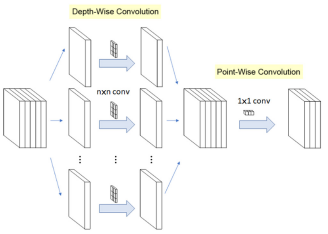


Fig. 13. Depth-separable convolution. Source: [134].

Depth-separable convolution breaks this into two steps (Figure 13):

(1) Doing a point-wise convolution with 1×1 filters such that the resulting feature map now has a depth of `output_channels`.
(2) Doing a spatial convolution with $s_x \times s_y$ filters in the x and y dimensions.

These two operations stacked together (without any intermediate non-linear activation) results in an output of the same shape as a regular convolution, with much fewer parameters ($1 \times 1 \times$ `input_channels` $\times$ `output_channels`) + ($s_x \times s_y \times$ `output_channels`), v/s $s_x \times s_y \times$ `input_channels` $\times$ `output_channels` for the regular convolution). Similarly, there is an order of magnitude less computation since the point-wise convolution is much cheaper for convolving with each input channel depth-wise (for more calculations, refer to the work of Sandler et al. [117]). The Xception model architecture [27] demonstrated that using depth-wise separable convolutions in the Inception architecture allowed reaching convergence sooner in terms of steps and a higher accuracy on the ImageNet dataset while keeping the number of parameters the same.

The MobileNet model architecture [117], which was designed for mobile and embedded devices, also uses the depth-wise separable layers instead of the regular convolutional layers. This helps them reduce the number of parameters as well as the number of multiply-add operations by 7× - 10× and allows deployment on Mobile for computer vision tasks. Users can expect a latency between 10 and 100 ms depending on the model. MobileNet also provides a knob via the depth multiplier for scaling the network to allow the user to trade off between accuracy and latency.

3.4.2 *Natural Language Understanding.*

Attention Mechanism and Transformer Family. One of the issues plaguing classical Seq2Seq models for solving tasks such as **Machine Translation (MT)** was that of the information bottleneck. Seq2Seq models typically have one or more encoder layers that encode the given input sequence ($\mathbf{x} = (x_1, x_2, \ldots, x_T)$) into a fixed-length vector(s) (also referred to as the context, $\mathbf{c}$), and one or more decoder layers that generate another sequence using this context. In the case of MT, the input

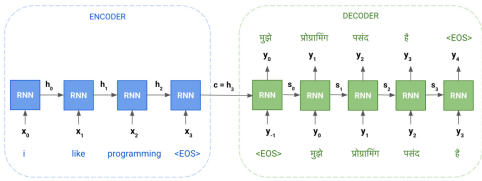


Fig. 14. Information bottleneck in a Seq2Seq model for translating from English to Hindi. The context vector c that the decoder has access to is fixed, and is typically the last hidden state (h_T).

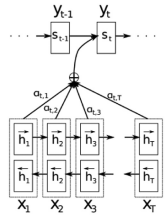


Fig. 15. Attention module learning a weighted context vector for each output token from the hidden states. Source: [17].

sequence can be a sentence in the source language, and the output sequence can be the sentence in the target language.

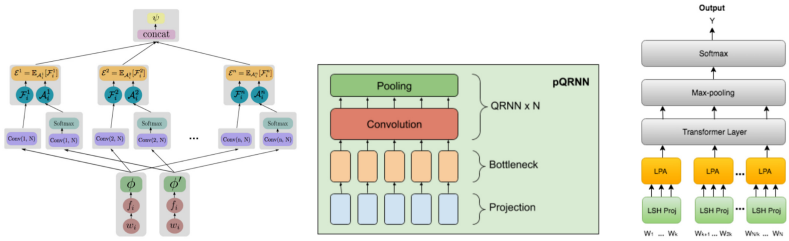
However, in classical Seq2Seq models such as that of Sutskever et al. [128], the decoder layers could only see the hidden state of the final encoder step ($c = h_T$). This is a *bottleneck* because the encoder block has to squash all the information about the sequence in a single context vector for all the decoding steps, and the decoder block has to somehow infer the entire encoded sequence from it (Figure 14). It is possible to increase the size of the context vector, but it would lead to an increase in the hidden state of all the intermediate steps, and make the model larger and slower.

The attention mechanism was introduced in the work of Bahdanau et al. [17] to be able to create a custom context vector for each output token, by allowing all hidden states to be visible to the decoder and then creating a weighted context vector, based on the output token's alignment with each input token (Figure 15). Essentially, the new weighted context vector is $c_i = \sum_j^T \alpha_{ij}.h_j$, where α_{ij} is the learned alignment (attention weight) between the decoder hidden state s_{i-1} and the hidden state for the j-th token (h_j). α_{ij} could be viewed as how much attention should the i-th input token be given when processing the j-th input token. This model is generalized in some cases by having explicit Query (Q), Key (K), and Value (V) vectors, where we seek to learn the attention weight distribution (α) between Q and K, and use it to compute the weighted context vector ($\mathbf{c}$) over V. In the preceding encoder-decoder architecture, Q is the decoder hidden state s_{i-1} and $K = V$ is the encoder hidden state h_j. Attention has been used to solve a variety of NLU tasks (MT, question answering, text classification, sentiment analysis), as well as vision, multi-modal tasks, and so forth. [24]. We refer the reader to the work of Chaudhari et al. [24] for further details on the taxonomy of attention models.

The Transformer architecture [138] was proposed in 2017, which introduced using self-attention layers for both the encoder and the decoder. They demonstrated that attention layers could be used to replace traditional RNN-based Seq2Seq models. The self-attention layer, the query, key, and value vectors are all derived from the same sequence by using different projection matrices.

Self-attention also allows parallelizing the process of deriving relationships between the tokens in the input sequences. RNNs inherently force the process to occur one step at a time—that is, learning long-range dependencies is $O(n)$, where n is the number of tokens. With self-attention, all tokens are processed together and pairwise relationships can be learned in $O(1)$ [138]. This makes it easier to leverage optimized training devices like GPUs and **Tensor Processing Units (TPUs)**. The authors reported up to 300× less training FLOPs as required to converge to a similar quality when compared to other recurrent and convolutional models. Tay et al. [132] discuss the computation and memory efficiency of several Transformer variants and their underlying self-attention mechanisms in detail.

As introduced earlier, the BERT model architecture [39] beat the state of the art in several NLU benchmarks. BERT is a stack of Transformer encoder layers that are pre-trained using a



(a) PRADO Model. Source: [72].

(b) PQRNN Model. Source: [73].

(c) ProFormer Model. Source: [120].

Fig. 16. Collection of notable random projection based models.

bi-directional masked language model training objective. It can also be used as a general-purpose encoder that can then be used for other tasks. Other similar models like the GPT family [21] have also been used for solving many NLU tasks.

Random Projection Layers and Models. Pre-trained token representations such as word2vec [93] and GloVe [106] are common for NLU tasks. However, since they require a d-dimensional vector for storing each token, the total size consumed by the table quickly grows very large if the vocabulary size V is substantial ($O(V.d)$). If model size is a constraint for deployment, we can either rely on compression techniques (as illustrated earlier) to help with embedding table compression, or evaluate layers and models that can work around the need for embedding tables. Random projection based methods [72, 73, 111, 112] are one such family of models that do so. They propose replacing the embedding table and lookup by mapping the input feature x (unicode token/word token, etc.) into a lower-dimensional space. This is done using the random projection operator $\mathbb{P}$, such that $\mathbb{P}(x) \in \{0,1\}^{T.r}$, which can be decomposed into T individual projection operations each generating an r-bit representation ($\mathbb{P}(x) = [\mathbb{P}_1(x), \ldots, \mathbb{P}_T(x)]$, where $\mathbb{P}_i(x) \in 0, 1^r$). T and r can be manually chosen.

Each random projection operation $\mathbb{P}_i$ is implemented using Locality Sensitive Hashing (LSH) [23, 112], each using a different hash function (via different seeds). For theoretical guarantees about the random projection operation, refer to work of Charikar [23], which demonstrates that the operation preserves the similarity between two points in the lower-dimensional space it maps these points to (this is crucial for the model to be learn the semantics about the inputs). If this relationship holds in the lower-dimensional space, the projection operation can be used to learn discriminative features for the given input. The core benefit of the projection operation when compared to embedding tables is $O(T)$ space required instead of $O(V.d)$ (T seeds required for T hash functions). However, random projection computation is $O(T)$ too v/s $O(1)$ for embedding table lookup. Hence, the projection layer is clearly useful when model size is the primary focus of optimization.

Across the various papers in the projection model family, there are subtle differences in implementation (computing complex features before [112] v/s after the projection operation [72, 119], generating a ternary representation instead of binary [72, 73], applying complex layers and networks on top like attention [72] and QRNN [73]). Some of the projection-based models (Figure 16) have demonstrated impressive results on NLU tasks. PRADO [72] generates n-gram features from the projected inputs, followed by having a multi-headed attention layer on top. It achieved accuracies comparable to standard LSTM models, while being 100× smaller, and taking 20 to 40 ms for inference on a Nexus 5$times$ device. PQRNN [73] is another projection-based model that additionally uses a fast RNN implementation (QRNN) [20] on top of the projected features. Kaliamoorthi et al. [73] report outperforming LSTMs while being 140× smaller, and achieving 97.1% of the quality of a BERT-like model while being 350× smaller.

ProFormer [120] introduces a local projected attention layer, which combines the projection operation with localized attention. Sankar et al. [120] demonstrate reaching ≈ 97.2% BERT-base's performance while occupying only 13% of BERT-base's memory. ProFormer also had 14.4 million parameters, compared to the 110 million parameters of BERT-base.

3.5 Infrastructure

In this section, we provide a non-exhaustive but comprehensive survey of leading software and hardware infrastructure components that do not improve any footprint or quality metrics by themselves but are critical to unlocking model efficiency in practice.

3.5.1 TensorFlow Ecosystem. TensorFlow [1] is a popular machine learning framework that has been used in production by many large enterprises. It has some of the most extensive software support for model efficiency.

TFLite for On-Device Use Cases. TFLite [13] is a collection of tools and libraries designed for inference in low-resource environments. At a high level, we can break down the TFLite project into two core parts:

- *Interpreter and op kernels*: TFLite provides an interpreter for running specialized TFLite models. The interpreter and the provided implementations of common neural net operations (op kernels) are primarily optimized for inference on ARM-based processors as of the time of this writing. They can also leverage smartphone DSPs such as Qualcomm's Hexagon [16] for faster execution. The interpreter also allows the user to set multiple threads for execution.
- *Converter*: The TFLite converter, as the name, suggests that it is useful for converting the given TensorFlow model into a single flatbuffer file for inference by the interpreter. Apart from the conversion itself, it handles a lot of internal details like getting a graph ready for quantized inference, fusing operations, adding other metadata to the model, and so on. With respect to quantization, it also allows post-training quantization as mentioned earlier.

XLA for Server-Side Acceleration. Typically, a TensorFlow model graph is executed by TensorFlow's executor process, and it uses standard optimized kernels for running it on CPU, GPU, and so on. XLA [14] is a graph compiler that can optimize linear algebra computations in a model by generating new kernels that are customized for the graph. These kernels are optimized for the model graph in question. For example, certain operations that can be fused together are combined in a single composite op. This avoids having to do multiple costly writes to RAM, when the operands can directly be operated on while they are still in cheaper caches. Kanwar et al. [74] report a 7× increase in training throughput and a 5× increase in the maximum batch size that can be used for BERT training using XLA. They reported training a BERT model for $32 on Google Cloud.

3.5.2 PyTorch Ecosystem. PyTorch [105] is another popular machine learning platform actively used by both academia and industry. It is often compared with TensorFlow in terms of usability and features.

On-Device Use Cases. PyTorch also has a light-weight interpreter that enables running PyTorch models on Mobile [8], with native runtimes for Android and iOS. This is analogous to the TFLite interpreter and runtime as introduced earlier. Similar to TFLite, PyTorch offers post-training quantization [9], and other graph optimization steps such as constant folding, fusing certain operations together, putting the channels last (NHWC) format for optimizing convolutional layers.

General Model Optimization. PyTorch also offers the Just-in-Time (JIT) compilation facility [10], which might seem similar to TensorFlow's XLA but is actually a mechanism for generating a serializable intermediate representation (high-level IR, per Li [86]) of the model from the code in

TorchScript [10], which is a subset of Python. TorchScript adds constraints on the code that it can convert, such as type checks, which allows it to sidestep some pitfalls of typical Python programming, while being Python compatible. It allows creating a bridge between the flexible PyTorch code for research and development to a representation that can be deployed for inference in production. For example, exporting to TorchScript is a requirement to run on mobile devices [8]. This representation is analogous to the static inference mode graphs generated by TensorFlow. The alternatives for XLA in the PyTorch world seem to be the Glow [115] and TensorComprehension [137] compilers. They help in generating the lower-level intermediate representation that is derived from the higher-level IR (TorchScript, TF Graph). These low-level deep learning compilers are compared in detail in the work of Li [86].

PyTorch offers a model tuning guide [7], which details various options that machine learning practitioners have at their disposal. Some of the core ideas in the guide include the following:

- Enabling device-specific optimizations, such as the cuDNN library, and Mixed-Precision Training with NVIDIA GPUs (explained in the GPU section).
- Fusion of point-wise operations (add, subtract, multiply, divide, etc.) using PyTorch JIT. Even though this should happen automatically, adding the `torch.jit.script` decorator to methods that are completely composed of point-wise operations can force the TorchScript compiler to fuse them.
- Enabling buffer checkpointing allows keeping the outputs of only certain layers in memory and computing the rest during the backward pass. This specifically helps with cheap to compute layers with large outputs like activations. A reduced memory usage can be exchanged for a larger batch size, which improves utilization of the training platform (CPU, GPU, TPU, etc.).
- Train with Distributed Data Parallel Training, which is suitable when there is a large amount of data and multiple GPUs are available for training. Each GPU gets its own copy of the model and optimizer, and operates on its own subset of the data. Each replica's gradients are periodically accumulated and then averaged.

3.5.3 Hardware-Optimized Libraries. We can further extract efficiency by optimizing for the hardware the neural networks run on. A prime deployment target is ARM's Cortex family of processors. Cortex supports SIMD (Single-Instruction Multiple Data) instructions via the Neon [90] architecture extension. SIMD instructions are useful for operating upon registers with vectors of data, which are essential for speeding up linear algebra operations through vectorization of these operations. QNNPACK [44] and XNNPACK [15] libraries are optimized for ARM Neon for mobile and embedded devices, and for x86 SSE2, AVX architectures, and so forth. QNNPACK supports several common ops in quantized inference mode for PyTorch. XNNPACK supports 32-bit floating-point models and 16-bit floating-point for TFLite. If a certain operation is not supported in XNNPACK, it falls back to the default implementation in TFLite. Similarly, there are other low-level libraries like Accelerate for iOS [6] and NNAPI for Android [4] that try to abstract away the hardware-level acceleration decision from higher-level machine learning frameworks.

3.5.4 Hardware: GPUs and TPUs. GPUs were originally designed for accelerating computer graphics but began to be used for general-purpose use cases with the availability of the CUDA library [32] in 2007, and libraries like cuBLAS for speeding up linear algebra operations. In 2009, Raina et al. [108] demonstrated that GPUs can be used to accelerate deep learning models. In 2012, following the AlexNet model's [77] substantial improvement over the next entrant in the ImageNet competition further standardized the use of GPUs for deep learning models. Since then, NVIDIA has released several iterations of its GPU microarchitectures with increasing focus on

deep learning performance. It has also introduced Tensor Cores [101, 125], which are dedicated execution units in their GPUs that are specialized for deep learning applications. Tensor Cores support training and inference in a range of precisions (fp32, TensorFloat32, fp16, bfloat16, int8, int4). As demonstrated earlier in quantization, switching to a lower precision is not always a significant tradeoff, since the difference in model quality might often be minimal.

Tensor Cores optimize the standard MAC operation [34], $\mathbf{A} = (\mathbf{B} \times \mathbf{C}) + \mathbf{D}$, where $\mathbf{B}$ and $\mathbf{C}$ are in a reduced precision (fp16, bfloat16, TensorFloat32), whereas $\mathbf{A}$ and $\mathbf{D}$ are in fp32. The core speedup comes from doing the expensive matrix multiplication in a lower precision. The result of the multiplication is in fp32, which can be relatively cheaply added with $\mathbf{D}$. When training with reduced precision, NVIDIA reports between 1× and 15× training speedup depending on the model architecture and the GPU chosen [125]. Tensor Cores in NVIDIA's latest Ampere architecture GPUs also support faster inference with sparsity (specifically, structured sparsity in the ratio 2:4, where two elements out of a block of four elements are sparse) [100]. They demonstrate an up to 1.5× speedup in inference time, and up to 1.8× speedup in individual layers. NVIDIA also offers the cuDNN library [100] that contains optimized versions of standard neural network operations such as FC, convolution, batch-norm, and activation.

TPUs are proprietary application-specific integrated circuits (ASICs) that Google has designed to accelerate deep learning applications with TensorFlow. Because they are not general-purpose devices, they need not cater for any non-machine learning applications (which most GPUs have had to), hence they are finely tuned for parallelizing and accelerating linear algebra operations. The first iteration of the TPU was designed for inference with 8-bit integers, and was being used in Google for a year prior to their announcement in 2016 [71]. Subsequent iterations of the TPU architectures enabled both training and inference with TPUs in floating-point too.

The core architecture of the TPU chips leverages the systolic array design [79, 80], where a large computation is split across a mesh-like topology, where each cell computes a partial result and passes it on to the next cell in the order, every clock step (in a rhythmic manner analogous to the systolic cardiac rhythm). Since there is no need to access registers for the intermediate results, once the required data is fetched the computation is not memory bound. Each TPU chip has two Tensor Cores (not to be confused with NVIDIA's tensor cores), each of which has a mesh of systolic arrays. There are four inter-connected TPU chips on a single TPU board. To further scale training and inference, a larger number of TPU boards can be connected in a mesh topology to form a 'pod.' As per publicly released numbers, each TPU chip (v3) can achieve 420 teraflops, and a TPU pod can reach 100+ petaflops [121].

TPUs have been used inside Google for applications like training models for Google Search, general-purpose BERT models [39], applications like DeepMind's world-beating AlphaGo and AlphaZero models [122], and many other research applications [131]. They have also set model training time records in the MLPerf benchmarks. Similar to the GPUs, TPUs support the bfloat16 data type [140], which is a reduced-precision alternative to training in full floating-point 32-bit precision. XLA support allows transparently switching to bfloat16 without any model changes.

4 A PRACTITIONER'S GUIDE TO EFFICIENCY

So far, we presented a broad set of tools and techniques in the efficient deep learning landscape. In this section, we present a practical guide for practitioners to use, and how these tools and techniques work with each other. As mentioned earlier, what we seek are *Pareto-optimal* models, where we would like to achieve the best possible result in one dimension while holding the other dimensions constant. Typically, one of these dimensions is *quality*, and the other is *footprint*. Quality-related metrics could include accuracy, F1, precision, recall, and AUC, among others, whereas footprint-related metrics can include model size, latency, RAM, and so forth.

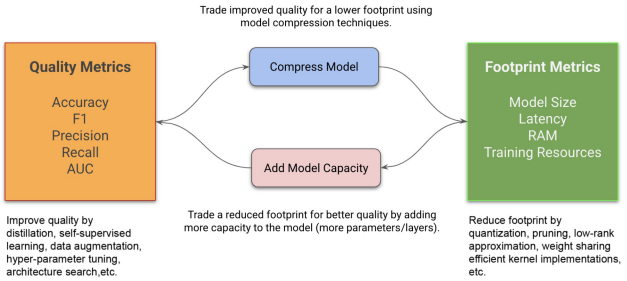


Fig. 17. Tradeoff between model quality and footprint. There exists a tradeoff between model quality and model footprint. Model quality can be improved with techniques like distillation, data augmentation, and hyper-parameter tuning. Compression techniques can in turn help trade off some model quality for a better model footprint. Some/all of the improvement in footprint metrics can also be traded for better quality by simply adding more model capacity.

Table 4. Examples of Techniques to Use in the Grow, Shrink, and Improve Phases

Grow (Model Capacity)	**Shrink** (Footprint)	**Improve** (Quality)
Add layers, width, etc., either manually or using width/depth/compound scaling multipliers	Reduce layers, width, etc., either manually or using width/depth/compound scaling multipliers	Manual tuning (architecture/hyper-parameters/features, etc.)
	Compression Techniques: Quantization, pruning, Low-rank factorization, etc.	**Learning Techniques:** Data augmentation, distillation, unsupervised learning, etc.
	Automation: Hyper-parameter optimization, architecture search, etc.	**Automation:** Hyper-parameter optimization, architecture search, etc.
	Efficient Layers & Models: Projection, PQRNN, (NLU), separable convolution (vision), etc.	**Efficient Layers & Models:** Transformers (NLU), Vi-T (vision), etc.

Naturally, there exists a tradeoff between quality and footprint metrics. A higher-capacity/deeper model is more likely to achieve better accuracy, but at the cost of model size, latency, and so on. However, a model with fewer parameters, although possibly suitable for deployment, is also likely to be worse in accuracy. As illustrated in Figure 17, we can start from a model with better quality metrics, and exchange some of that quality for better footprint by naively compressing the model/reducing the model capacity (*shrink*). Similarly, it is possible to naively improve quality by adding more capacity to the model (*grow*). Growing can be addressed by the author of the model via appropriately increasing model capacity and tweaking other hyper-parameters to improve model quality. Shrinking can be achieved via compression techniques (quantization, pruning, low-rank approximation, etc.), efficient layers and models, architecture search via automation, and so on. In addition, we can *improve* the quality metrics while keeping the footprint the same through learning techniques (distillation, data augmentation, self-supervised learning), automation (hyper-parameter tuning), and so forth. (Table 4 presents more examples.)

Combining these three phases, we propose two strategies toward achieving Pareto-optimal models:

(1) *Shrink-and-Improve for footprint-sensitive models*: If, as a practitioner, you want to reduce your footprint while keeping quality the same, this could be a useful strategy for on-device deployments and server-side model optimization. Shrinking should ideally be minimally

lossy in terms of quality (can be achieved via learned compression techniques, architecture search, etc.), but in some cases even naively reducing capacity can also be compensated by the Improve phase. It is also possible to do the Improve phase before the Shrink phase.

(2) *Grow-Improve-and-Shrink for quality-sensitive models*: When you want to deploy models that have better quality while keeping the same footprint, it might make sense to follow this strategy. Here, the capacity is first added by growing the model as illustrated earlier. The model is then improved using via learning techniques, automation, and so on, then shrunk back either naively or in a learned manner. Alternatively, the model could be shrunk back either in a learned manner directly after growing the model too.

We consider both of these strategies as a way of going from a potentially non-Pareto-optimal model to another one that lies on the Pareto-frontier with the tradeoff that is appropriate for the user. Each efficiency technique individually helps move us closer to that target model.

4.1 Experiments

To demonstrate what we proposed previously, we undertook the task of going through the exercise of making a given deep learning model efficient. Concretely, we had the following goals with this exercise:

(1) Achieve a new Pareto-frontier using the efficiency techniques, hence demonstrating that these techniques can be used in isolation as well as in combination with other techniques in the real world by machine learning practitioners.
(2) With various combinations of efficiency techniques and model scaling, demonstrate the tradeoffs for both 'Shrink-and-Improve' and 'Grow-Improve-and-Shrink' strategies for discovering and traversing the Pareto-frontier.

We picked the problem of classifying images in the CIFAR-10 dataset [76] on compute constrained devices such as smartphones and IoT devices. We designed a deep convolutional architecture where we could scale the model capacity up or down by increasing or decreasing the 'width multiplier' (w) value. In the implementation, w scales the number of filters for the convolutional layers (except the first two). Hence, using different values of w in $[0.1, 0.25, 0.5, 0.75, 1.0]$, we obtain a family of models with different quality and footprint tradeoffs. We trained these models with some manual tuning to achieve a baseline of quality v/s footprint metrics. In this case, we measured quality through accuracy, and footprint through number of parameters, model size, and latency. In terms of techniques, we used quantization for shrinking, and data augmentation and distillation for improving. Many other techniques could be used to further drive the point home (automation like HPO, efficient layers like separable convolutions) but were skipped to keep the interpretation of the results simpler. We used the TensorFlow-backed Keras APIs [28] for training and the TFLite [13] framework for inference. The latencies were measured on three kinds of devices, low-end (Oppo A5), mid-end (Pixel 3XL), and high-end (Galaxy S10), in order of their increasing CPU compute power. The model size numbers reported are the sizes of the generated TFLite models, and the latency numbers are the average single-threaded CPU latency after warmup on the target device. The code for the experiments is available via an Jupyter notebook.

Table 5 compiles the results for six width multipliers in increasing order, ranging from 0.05 to 1.0. Between the smallest to the largest models, the number of params grows by $\approx 91.4\times$, and the model size grows by $\approx 80.2\times$. The latency numbers also grow between $3.5\times$ and $10\times$ based on the device. Within the same row, footprint metrics will not change since we are not changing the model architecture. In Table 5, we purely work with techniques that will improve the model quality (data augmentation and distillation). Table 6 reports the numbers for the quantized versions of the

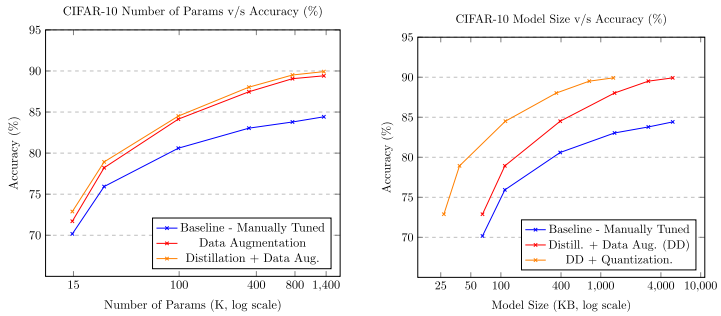
Table 5. Quality and Footprint Metrics for Floating-Point Models for the CIFAR-10 Dataset

Width Multiplier	# Params (K)	Model Size (KB)	Accuracy (%)			Average Latency (ms)		
			Baseline	Augmentation	Augmentation + Distillation	Oppo A5	Pixel 3XL	Galaxy S10
0.05	14.7	65.45	70.17	71.71	72.89	6.72	0.6	0.78
0.1	26	109.61	75.93	78.22	78.93	6.85	1.7	0.85
0.25	98.57	392.49	80.6	84.14	84.51	8.15	2.02	0.93
0.5	350.05	1,374.11	83.04	87.47	88.03	11.46	2.8	1.33
0.75	764.87	2,993.71	83.79	89.06	89.51	16.7	4.09	1.92
1	1,343.01	5,251.34	84.42	89.41	89.92	24	5.99	2.68

Table 6. Quality and Footprint Metrics for *Quantized* Models for the CIFAR-10 Dataset

Width Multiplier	# Params (K)	Model Size (KB)	Accuracy (%)			Average Latency (ms)		
			Baseline	Augmentation	Augmentation + Distillation	Oppo A5	Pixel 3XL	Galaxy S10
0.05	14.7	26.87	69.9	71.72	72.7	4.06	0.49	0.43
0.1	26	38.55	75.98	78.19	78.55	4.5	1.25	0.47
0.25	98.57	111	80.76	83.98	84.18	4.52	1.31	0.48
0.5	350.05	359.31	83	87.32	87.86	6.32	1.73	0.58
0.75	764.87	767.09	83.6	88.57	89.29	8.53	2.36	0.77
1	1,343.01	1,334.41	84.52	89.28	89.91	11.73	3.27	1.01

Each model is the quantized equivalent of the corresponding model in Table 5.



(a) Number of Params v/s Accuracy (b) Model Size v/s Accuracy

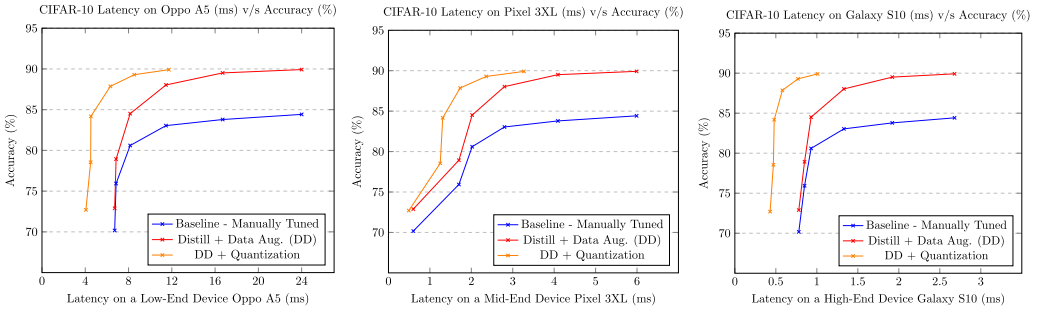
Fig. 18. Change in accuracy with respect to number of params and model size. Each point on a curve is a model from Table 5 in (a) and from Table 6 in (b).

corresponding models in Table 5. We use quantization for the Shrink phase, to reduce the model size by ≈ 4×, and reduce the average latency by 1.5× and 2.65×. Figures 18 and 19 plot the notable results from Tables 5 and 6.

4.2 Discussion and Insights

Let us try to interpret the preceding data to validate if our strategies can be used practically.

Shrink-and-Improve for footprint-sensitive models. Refer to Table 5 and Figure 18. Assume that our goal was to deploy the model with width multiplier (w) = 1.0 and accuracy 84.42%, but the bottleneck was the model size (5.25 MB) and latency on a low-end device (24 ms on Oppo A5).



(a) Low-End Device Latency (b) Mid-Tier Device Latency (c) High-End Device Latency

Fig. 19. Average latency of models on different devices (low-, mid-, and high-end smartphones). The orange curve denotes the quantized models in addition to being trained with distillation and data augmentation.

This is the classic case of the footprint metrics not meeting the bar, hence we could apply the Shrink-and-Improve strategy by first naively scaling our model down to a width multiplier (w) of 0.25. This smaller model when manually tuned, as seen in Table 5, achieves an accuracy of 80.76%. However, when we use a combination of data augmentation and distillation from a separately trained larger teacher model with an accuracy of 90.86%, the accuracy of the smaller model improves to 84.18%, very close to the target model that we want to deploy. The size of this smaller model is 392.49 KB, which is 13.8× smaller, and the latency is 8.15 ms, which is 2.94× faster at a comparable accuracy. It is possible to further compress this model by using quantization for some additional shrinking. The same smaller model ($w = 0.25$) when quantized in Table 6 is 111 KB in size (47.3× *smaller*) and has a latency of 4.52 ms (5.31× *faster*), while retaining an accuracy of 84.18%. It is possible to do this for other pairs of points on the curves.

Grow-Improve-and-Shrink for quality-sensitive models. Assume that our goal is to deploy a model that has footprint metrics comparable to the model with $w = 0.25$ (392.49 KB model size, 0.93 ms on a high-end Galaxy S10 device) but an accuracy better than the baseline 80.6% (refer to Table 5). In this case, we can choose to first grow our model to $w = 0.5$. This instantly blows up the model size to 1.37 MB (3.49× bigger) and latency to 1.33 ms (1.43× slower). However, we ignore that for a bit and improve our model's quality to 88.03% with data augmentation and distillation. Then using quantization for shrinking (refer to Table 6), we can get a model that is 359.31 KB in size (32 KB smaller) and has a 0.58-ms latency on Galaxy S10 (1.6× faster), with an accuracy of 87.86%, an absolute 7.10% increase in accuracy while keeping the model size approximately the same and making it 1.6× faster. It is also possible to apply this strategy to other pairs of models.

Thus, we have verified that the preceding two strategies can work both ways, whether your goal is to optimize for quality metrics or footprint metrics. We were also able to visually inspect through Figures 18 and 19 that efficiency techniques can improve on the Pareto-frontiers constructed through manual tuning. To contain the scope of experimentation, we selected two sets of efficiency techniques (compression techniques (quantization), and learning techniques (data augmentation and distillation). Hence, it would be useful to explore other techniques as well such as automation (for HPO to further improve on results), and efficient layers and models (separable convolution as illustrated in MobileNet [117] could be used in place of larger convolutional layers). Finally, we would also like to emphasize paying attention to the performance of deep learning models (optimized or not) on under-represented classes and out-of-distribution data to ensure model fairness, since quality metrics alone might not be sufficient for discovering deeper issues with models [61].

5 FUTURE TRENDS

As deep learning continues to grow, the interest in making training and efficient deep learning will continue to develop. We envision the following themes to be the most prominent in the coming years:

- *Cheaper training for large models*: Training very large models is prohibitively expensive. However, there has been progress made on this front. Generative models like DALL-E [109], Imagen [116], and Parti [144] require hundreds of GPUs/TPUs over multiple weeks, which implies more than $100K in training costs. Even larger language models like PaLM (540 B parameters) [29] are estimated to run into multiple millions of dollars to train [59]. Therefore, optimizing for training resources for these models would be at the forefront for both industry and academia.
- *Inference costs*: Again, very large language models like PaLM, although extremely versatile and have established state-of-the-art results, are also expensive to query. Bringing down inference costs through either smarter model optimization or training smaller models with learning techniques would be key to bringing the advancements of these larger models to production use cases.
- *Dedicated hardware for specialized use cases*: Custom hardware such as EdgeTPU [51] and Jetson [102] have been designed for efficient inference for embedded and edge devices. With autonomous driving, home automation, robotics, and several other use cases pushing deep learning to new applications, we would find specialized hardware designed to accelerate deep learning use cases.
- *Efficient architectures*: Transformers have gone beyond standard language tasks, and have started being used in vision use cases through models like ViT [43] and subsequent optimizations like MobileViT [91]. We will continue to see further standardization in efficient architectures like transformers across multiple domains.
- *Benchmarks for efficient deep learning*: With the propagation of large models, the industry will start to work together to set up benchmarks for measuring training and inference efficiency on various tasks such as MLPerf [95, 96]. Results on these benchmarks would start to become the new targets for researchers to optimize once they reach a desirable model performance with unoptimized networks. This is already the case with research groups advertising that their methods train a ResNet model on ImageNet to a certain accuracy in a short duration [99].

6 CONCLUSION

In this article, we started with demonstrating the rapid growth in deep learning models, and motivating the fact that someone training and deploying models today has to make either implicit or explicit decisions about efficiency. However, the landscape of model efficiency is vast.

To help with this, we laid out a mental model for the readers to wrap their heads around the multiple focus areas of model efficiency and optimization. The surveys of the core model optimization techniques give the reader an opportunity to understand the state of the art, apply these techniques in the modeling process, and/or use them as a starting point for exploration. The infrastructure section also lays out the software libraries and hardware that make training and inference of efficient models possible.

Next, we presented a section of explicit and actionable insights supplemented by code for a practitioner to use as a guide in this space. That section will hopefully give concrete and actionable takeaways, as well as tradeoffs, to think about when optimizing a model for training and deployment. We also offered a section enumerating some of the broad trends in deep learning efficiency

that are likely to have a significant impact on how we train and deploy deep learning models in the near future.

It is our hope that in addition to a survey of the field, we have also provided the reader a mental framework to think and reason about deep learning efficiency, as well as practical techniques to go about achieving footprint and quality gains in real-world deep learning models.

ACKNOWLEDGMENTS

We would like to thank the Learn2Compress team at Google Research for their support with this work. We would also like to thank Akanksha Saran and Aditya Sarawgi for their help with proof-reading and suggestions for improving the content.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Retrieved February 1, 2023 from https://www.tensorflow.org/. (Software available from tensorflow.org.)

[2] Apoorv Agnihotri and Nipun Batra. 2020. Exploring Bayesian optimization. *Distill* 5, 5 (2020), e26.

[3] Bahriye Akay, Dervis Karaboga, and Rustu Akay. 2022. A comprehensive survey on optimizing deep learning models by metaheuristics. *Artificial Intelligence Review* 55, 2 (Feb. 2022), 829–894.

[4] Android Developers. 2021. Neural Networks API | Android NDK | Android Developers. Retrieved June 3, 2021 from https://developer.android.com/ndk/guides/neuralnetworks.

[5] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems* 13, 3 (2017), 1–18.

[6] Apple Authors. 2021. Accelerate | Apple Developer Documentation. Retrieved June 3, 2021 from https://developer.apple.com/documentation/accelerate.

[7] PyTorch Authors. 2021. Performance Tuning Guide—PyTorch Tutorials 1.8.1+cu102 Documentation. Retrieved June 3, 2021 from https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html.

[8] PyTorch Authors. 2021. PyTorch Mobile. Retrieved June 3, 2021 from https://pytorch.org/mobile/home.

[9] PyTorch Authors. 2021. Quantization Recipe—PyTorch Tutorials 1.8.1+cu102 documentation. Retrieved June 3, 2021 from https://pytorch.org/tutorials/recipes/quantization.html.

[10] PyTorch Authors. 2021. torch.jit.script—PyTorch 1.8.1 Documentation. Retrieved June 3, 2021 from https://pytorch.org/docs/stable/generated/torch.jit.script.html.

[11] TensorFlow Authors. 2021. Post-Training Quantization | TensorFlow Lite. Retrieved June 3, 2021 from https://www.tensorflow.org/lite/performance/post_training_quantization.

[12] Tensorflow Authors. 2021. TensorFlow Lite Converter. Retrieved June 3, 2021 from https://www.tensorflow.org/lite/convert.

[13] Tensorflow Authors. 2021. TensorFlow Lite | ML for Mobile and Edge Devices. Retrieved June 3, 2021 from https://www.tensorflow.org/lite.

[14] TensorFlow Authors. 2021. XLA: Optimizing Compiler for Machine Learning | TensorFlow. Retrieved June 3, 2021 from https://www.tensorflow.org/xla.

[15] XNNPACK Authors. 2021. XNNPACK. Retrieved June 3, 2021 from https://github.com/google/XNNPACK.

[16] XNNPACK Authors. 2021. XNNPACK Backend for TensorFlow Lite. Retrieved June 3, 2021 from https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/delegates/xnnpack/README.md/#sparse-inference.

[17] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*.

[18] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, 2 (2012), 281–305.

[19] Avrim Blum and Tom Mitchell. 1998. Combining labeled and unlabeled data with co-training. In *Proceedings of the 11th Annual Conference on Computational Learning Theory*. 92–100.

[20] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. 2017. Quasi-recurrent neural networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR '17)*.

[21] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33 (NeurIPS'20)*.

[22] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 535–541.
[23] Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*. 380–388.
[24] Sneha Chaudhari, Varun Mithal, Gungor Polatkan, and Rohan Ramanath. 2021. An attentive survey of attention models. *ACM Transactions on Intelligent Systems and Technology* 12, 5 (2021), Article 53, 32 pages. https://doi.org/10.1145/3465055
[25] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research* 16 (2002), 321–357.
[26] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2018. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine* 35, 1 (2018), 126–136. https://doi.org/10.1109/MSP.2017.2765695
[27] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1251–1258.
[28] Francois Chollet. 2020. The Keras Blog. Retrieved June 4, 2021 from https://blog.keras.io.
[29] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, et al. 2022. PaLM: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).
[30] Dan C. Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. 2011. High-performance neural networks for visual object classification. *CoRR* abs/1102.0183 (2011). http://arxiv.org/abs/1102.0183.
[31] Contributors to Wikimedia Projects. 2021. AVX-512—Wikipedia. Retrieved June 3, 2021 from https://en.wikipedia.org/w/index.php?title=AVX-512&oldid=1025044245.
[32] Contributors to Wikimedia Projects. 2021. CUDA—Wikipedia. Retrieved June 3, 2021 from https://en.wikipedia.org/w/index.php?title=CUDA&oldid=1025500257.
[33] Contributors to Wikimedia Projects. 2021. Hyperparameter Optimization—Wikipedia. Retrieved June 3, 2021 from https://en.wikipedia.org/w/index.php?title=Hyperparameter_optimization&oldid=1022309479.
[34] Contributors to Wikimedia Projects. 2021. Multiply-Accumulate Operation—Wikipedia. Retrieved June 3, 2021 from https://en.wikipedia.org/w/index.php?title=Multiply-accumulate_operation&oldid=1026461481.
[35] Contributors to Wikimedia Projects. 2021. SSE^4—Wikipedia. Retrieved June 3, 2021 from https://en.wikipedia.org/w/index.php?title=SSE4&oldid=1023092035.
[36] Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. 2020. RandAugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 702–703.
[37] Mostafa Dehghani, Yi Tay, Anurag Arnab, Lucas Beyer, and Ashish Vaswani. 2022. The efficiency misnomer. In *Proceedings of the International Conference on Learning Representations*.
[38] Tim Dettmers and Luke Zettlemoyer. 2019. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840* (2019).
[39] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*.
[40] Thomas G. Dietterich. 2000. Ensemble methods in machine learning. In *Proceedings of the International Workshop on Multiple Classifier Systems*. 1–15.
[41] Carl Doersch, Abhinav Gupta, and Alexei A. Efros. 2015. Unsupervised visual representation learning by context prediction. In *Proceedings of the IEEE International Conference on Computer Vision*. 1422–1430.
[42] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. 2017. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*.
[43] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proceedings of the 9th International Conference on Learning Representations (ICLR '21)*. https://openreview.net/forum?id=YicbFdNTTy.
[44] Marat Dukhan, Yiming Wu Wu, and Hao Lu. 2020. QNNPACK: Open Source Library for Optimized Mobile Deep Learning—Facebook Engineering. Retrieved June 3, 2021 from https://engineering.fb.com/2018/10/29/ml-applications/qnnpack.
[45] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. 2020. Fast sparse ConvNets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14629–14638.
[46] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *Journal of Machine Learning Research* 20, 55 (2019), 1–21.

[47] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. 2020. Rigging the lottery: Making all tickets winners. In *Proceedings of the International Conference on Machine Learning*. 2943–2952.

[48] Alhussein Fawzi, Horst Samulowitz, Deepak Turaga, and Pascal Frossard. 2016. Adaptive data augmentation for image classification. In *Proceedings of the 2016 IEEE International Conference on Image Processing (ICIP'16)*. IEEE, Los Alamitos, CA, 3688–3692.

[49] Spyros Gidaris, Praveer Singh, and Nikos Komodakis. 2018. Unsupervised representation learning by predicting image rotations. In *Proceedings of the 6th International Conference on Learning Representations (ICLR '18)*.

[50] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1487–1495.

[51] Google. 2021. Edge TPU Performance Benchmarks | Coral. Retrieved June 3, 2021 from https://coral.ai/docs/edgetpu/benchmarks.

[52] Arjun Gopalan, Da-Cheng Juan, Cesar Ilharco Magalhaes, Chun-Sung Ferng, Allan Heydon, Chun-Ta Lu, Philip Pham, George Yu, Yicheng Fan, and Yueqi Wang. 2021. Neural structured learning: Training neural networks with structured signals. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*. 1150–1153.

[53] Song Han, Huizi Mao, and William J. Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Proceedings of the 4th International Conference on Learning Representations (ICLR '16)*.

[54] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*. 1135–1143.

[55] Lars Kai Hansen and Peter Salamon. 1990. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12, 10 (1990), 993–1001.

[56] Babak Hassibi, David G. Stork, and Gregory J. Wolff. 1993. Optimal brain surgeon and general network pruning. In *Proceedings of the IEEE International Conference on Neural Networks*. IEEE, Los Alamitos, CA, 293–299.

[57] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[58] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*. 784–800.

[59] Lennart Heim. 2022. Estimating xn–PaLM-kd53c's training cost. *Blog.heim.* Retrieved February 1, 2023 from https://blog.heim.xyz/palm-training-cost.

[60] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2014. Distilling the knowledge in a neural network. In *Proceedings of the NeurIPS 2014 Deep Learning Workshop*.

[61] Sara Hooker, Nyalleng Moorosi, Gregory Clark, Samy Bengio, and Emily Denton. 2020. Characterising bias in compressed models. *arXiv preprint arXiv:2010.03058* (2020).

[62] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, et al. 2019. Searching for MobileNetV3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1314–1324.

[63] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, Volume 1 (Long Papers)*. 328–339.

[64] Chi-Hung Hsu, Shu-Huan Chang, Jhao-Hong Liang, Hsin-Ping Chou, Chun-Hao Liu, Shih-Chieh Chang, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Da-Cheng Juan. 2018. Monas: Multi-objective neural architecture search using reinforcement learning. *arXiv preprint arXiv:1806.10332* (2018).

[65] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

[66] Hiroshi Inoue. 2018. Data augmentation by pairing samples for images classification. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*.

[67] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.

[68] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, et al. 2017. Population based training of neural networks. *arXiv preprint arXiv:1711.09846* (2017).

[69] Kevin Jamieson and Ameet Talwalkar. 2016. Non-stochastic best arm identification and hyperparameter optimization. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics (AISTATS'16)*. 240–248.

[70] Jeremy Jordan. 2020. Setting the learning rate of your neural network. *Jeremy Jordan.* Retrieved February 1, 2023 from https://www.jeremyjordan.me/nn-learning-rate.

[71] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.

[72] Prabhu Kaliamoorthi, Sujith Ravi, and Zornitsa Kozareva. 2019. PRADO: Projection attention networks for document classification on-device. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP'19)*. 5012–5021.

[73] Prabhu Kaliamoorthi, Aditya Siddhant, Edward Li, and Melvin Johnson. 2021. Distilling large language models into tiny and effective students using pQRNN. *arXiv preprint arXiv:2101.08890* (2021).

[74] Pankaj Kanwar, Peter Brandt, and Zongwei Zhou. 2021. TensorFlow 2 MLPerf Submissions Demonstrate Best-in-Class Performance on Google Cloud. Retrieved June 3, 2021 from https://blog.tensorflow.org/2020/07/tensorflow-2-mlperf-submissions.html.

[75] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv:1806.08342* (2018). https://arxiv.org/abs/1806.08342v1.

[76] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto.

[77] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*. 1097–1105.

[78] Anders Krogh and Jesper Vedelsby. 1994. Neural network ensembles, cross validation and active learning. In *Proceedings of the 7th International Conference on Neural Information Processing Systems (NIPS'94)*.

[79] H. T. Kung and C. E. Leiserson. 1980. Introduction to VLSI systems. *Algorithms for VLSI Processor Arrays*, C. A. Mead and L. Conway (Eds.). Addison-Wesley, Reading, MA, 271–292.

[80] Hsiang-Tsung Kung. 1982. Why systolic architectures? *IEEE Computer* 15, 1 (1982), 37–46.

[81] Yann Lecun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86, 11 (Nov. 1998), 2278–2324. https://doi.org/10.1109/5.726791

[82] Yann LeCun, John S. Denker, and Sara A. Solla. 1990. Optimal brain damage. In *Advances in Neural Information Processing Systems 2*. 598–605.

[83] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. In *Proceedings of the 30th Conference on Neural Information Processing Systems (NIPS'16)*.

[84] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient ConvNets. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'16): Poster*.

[85] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research* 18, 1 (2017), 6765–6816.

[86] Sharon Y. Li. 2020. Automating data augmentation: Practice, theory and new direction. *SAIL Blog*. Retrieved February 1, 2023 from http://ai.stanford.edu/blog/data-augmentation.

[87] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*. 19–34.

[88] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable architecture search. In *Proceedings of the 7th International Conference on Learning Representations (ICLR'19)*.

[89] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. 2019. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 3296–3305.

[90] Arm Ltd. 2021. SIMD ISAs | Neon—Arm Developer. Retrieved June 3, 2021 from https://developer.arm.com/architectures/instruction-sets/simd-isas/neon.

[91] Sachin Mehta and Mohammad Rastegari. 2021. MobileViT: Light-weight, general-purpose, and mobile-friendly vision transformer. *arXiv preprint arXiv:2110.02178* (2021).

[92] Gaurav Menghani and Sujith Ravi. 2019. Learning from a teacher using unlabeled data. *arXiv preprint arXiv:1911.05275* (2019).

[93] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. 2017. Advances in pre-training distributed word representations. *arXiv preprint arXiv:1712.09405* (2017).

[94] Rahul Mishra, Hari Prabhat Gupta, and Tanima Dutta. 2020. A survey on deep neural network compression: Challenges, overview, and solutions. *arXiv preprint arXiv:2010.03954* (2020).

[95] MLCommons. 2022. v2.0 Results. Retrieved June 29, 2022 from https://mlcommons.org/en/training-normal-20.

[96] MLCommons. 2022. v2.0 Results. Retrieved June 29, 2022 from https://mlcommons.org/en/inference-datacenter-20.

[97] Jonas Močkus. 1975. On Bayesian methods for seeking the extremum. In *Proceedings of the Optimization Techniques IFIP Technical Conference*. 400–404.

[98] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning convolutional neural networks for resource efficient inference. *arXiv:1611.06440* (2017).
[99] MosaicML. 2022. Blazingly Fast Computer Vision Training with the Mosaic ResNet and Composer. Retrieved June 29, 2022 from https://www.mosaicml.com/blog/mosaic-resnet.
[100] NVIDIA. 2020. GTC 2020: Accelerating Sparsity in the NVIDIA Ampere Architecture. Retrieved June 3, 2021 from https://developer.nvidia.com/gtc/2020/video/s22085-vid.
[101] NVIDIA. 2020. Inside Volta: The World's Most Advanced Data Center GPU | NVIDIA Developer Blog. Retrieved June 3, 2021 from https://developer.nvidia.com/blog/inside-volta.
[102] NVIDIA. 2021. NVIDIA Embedded Systems for Next-Gen Autonomous Machines. Retrieved June 4, 2021 from https://www.nvidia.com/en-us/autonomous-machines/embedded-systems.
[103] Rina Panigrahy. 2021. Matrix Compression Operator. Retrieved June 5, 2021 from https://blog.tensorflow.org/2020/02/matrix-compression-operator-tensorflow.html.
[104] PapersWithCode.com. 2021. Papers with Code—The Latest in Machine Learning. Retrieved June 3, 2021 from https://paperswithcode.com.
[105] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32 (NeurIPS'19)*. 8024–8035.
[106] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP'14)*. 1532–1543.
[107] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient neural architecture search via parameters sharing. In *Proceedings of the International Conference on Machine Learning*. 4095–4104.
[108] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*. 873–880.
[109] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical text-conditional image generation with CLIP latents. *arXiv preprint arXiv:2204.06125* (2022).
[110] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision*. 525–542.
[111] Sujith Ravi. 2019. ProjectionNet: Learning efficient on-device deep networks using neural projections. In *Proceedings of the 36th International Conference on Machine Learning*.
[112] Sujith Ravi and Zornitsa Kozareva. 2018. Self-governing neural networks for on-device short text classification. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 887–893.
[113] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 4780–4789.
[114] Google Research. 2021. Fast Sparse ConvNets—GitHub Repository. Retrieved June 3, 2021 from https://github.com/google-research/google-research/tree/master/fastconvnets.
[115] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).
[116] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, et al. 2022. Photorealistic text-to-image diffusion models with deep language understanding. *arXiv preprint arXiv:2205.11487* (2022).
[117] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.
[118] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. In *Proceedings of the 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing (NeurIPS'19)*.
[119] Chinnadhurai Sankar, Sujith Ravi, and Zornitsa Kozareva. 2019. Transferable neural projection representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT'19)*. 3355–3360.
[120] Chinnadhurai Sankar, Sujith Ravi, and Zornitsa Kozareva. 2021. ProFormer: Towards on-device LSH projection based transformers. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics*. 2823–2828.
[121] Kaz Sato. 2021. What Makes TPUs Fine-Tuned for Deep Learning? | Google Cloud Blog. Retrieved June 3, 2021 from https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning.
[122] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, et al. 2020. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (2020), 604–609.

[123] Patrice Y. Simard, David Steinkraus, and John C. Platt. 2003. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the 7th International Conference on Document Analysis and Recognition (ICDAR'03)*.

[124] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556* (2015).

[125] Dusan Stosic. 2020. Training Neural Networks with Tensor Cores—Dusan Stosic, NVIDIA. Retrieved June 3, 2021 from https://www.youtube.com/watch?v=jF4-_ZK_tyc.

[126] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. 2017. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE International Conference on Computer Vision*. 843–852.

[127] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBERT: A compact task-agnostic BERT for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 2158–2170. https://doi.org/10.18653/v1/2020.acl-main.195.

[128] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27 (NeurIPS'14)*. 3104–3112.

[129] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE* 105, 12 (2017), 2295–2329.

[130] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

[131] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2820–2828.

[132] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2023. Efficient transformers: A survey. *ACM Computing Surveys* 55, 6 (2023), Article 109: 28 pages.

[133] TensorFlow. 2021. Model Optimization | TensorFlow Lite. Retrieved June 3, 2021 from https://www.tensorflow.org/lite/performance/model_optimization.

[134] Sik-Ho Tsang. 2019. Review: Xception—With depthwise separable convolution, better than Inception-v3 (image classification). *Medium*. Retrieved February 1, 2023 from https://towardsdatascience.com/review-xception-with-depthwise-separable-convolution-better-than-inception-v3-image-dc967dd42568.

[135] Gregor Urban, Krzysztof J. Geras, Samira Ebrahimi Kahou, Özlem Aslan, Shengjie Wang, Abdelrahman Mohamed, Matthai Philipose, Matthew Richardson, and Rich Caruana. 2017. Do deep convolutional nets really need to be deep and convolutional? In *Proceedings of the 5th International Conference on Learning Representations (ICLR'17)*.

[136] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop (NIPS'11)*.

[137] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[138] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS'17)*. 1–11.

[139] Peisong Wang, Qiang Chen, Xiangyu He, and Jian Cheng. 2020. Towards accurate post-training network quantization via bit-split and stitching. In *Proceedings of the International Conference on Machine Learning*. 9847–9856. http://proceedings.mlr.press/v119/wang20c.html.

[140] Shibo Wang and Pankaj Kanwar. 2021. BFloat16: The Secret to High Performance on Cloud TPUs | Google Cloud Blog. Retrieved June 3, 2021 from https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

[141] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. FBNet: Hardware-aware efficient ConvNet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10734–10742.

[142] Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V. Le. 2020. Self-training with noisy student improves ImageNet classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10687–10698.

[143] I. Zeki Yalniz, Hervé Jégou, Kan Chen, Manohar Paluri, and Dhruv Mahajan. 2019. Billion-scale semi-supervised learning for image classification. *arXiv preprint arXiv:1905.00546* (2019).

[144] Jiahui Yu, Yuanzhong Xu, Jing Yu Koh, Thang Luong, Gunjan Baid, Zirui Wang, Vijay Vasudevan, et al. 2022. Scaling autoregressive models for content-rich text-to-image generation. *arXiv preprint arXiv:2206.10789* (2022).

[145] Tong Yu and Hong Zhu. 2020. Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689* (2020).

[146] Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. 2017. On compressing deep models by low rank and sparse decomposition. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*. IEEE, Los Alamitos, CA, 67–76. https://doi.org/10.1109/CVPR.2017.15

[147] Sergey Zagoruyko and N. Komodakis. 2016. Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer. In *Proceedings of the 5th International Conference on Learning Representations (ICLR'16)*. 1–13.

[148] Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. 2018. *mixup*: Beyond empirical risk minimization. In *Proceedings of the 6th International Conference on Learning Representations (ICLR '18)*. 1–13.

[149] Michael Zhu and Suyog Gupta. 2018. To prune, or not to prune: Exploring the efficacy of pruning for model compression. In *Proceedings of the 6th International Conference on Learning Representations (ICLR '18)*.

[150] Barret Zoph and Quoc V. Le. 2017. Neural architecture search with reinforcement learning. In *Proceedings of the 5th International Conference on Learning Representations (ICLR '17)*.

[151] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 8697–8710.

Received 13 July 2021; revised 30 June 2022; accepted 22 November 2022