

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

GAURAV MENGHANI, Google Research, USA

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, resources required to train, etc. 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. Our hope is that this survey would provide the reader 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.

ACM Reference Format:

Gaurav Menghani. 2021. Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better. 1, 1 (June 2021), 43 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Deep Learning with neural networks has been the dominant methodology of training new machine learning models for the past decade. Its rise to prominence is often attributed to the ImageNet competition [45] in 2012. That year, a University of Toronto team submitted a deep convolutional network (AlexNet [92], named after the lead developer Alex Krizhevsky), performed 41% better than the next best submission. As a result of this trailblazing work, there was a race to create deeper networks with an ever increasing number of parameters and complexity. Several model architectures such as VGGNet [141], Inception [146], ResNet [73] etc. successively beat previous records at ImageNet competitions in the subsequent years, while also increasing in their footprint (model size, latency, etc.)

This effect has also been noted in natural language understanding (NLU), where the Transformer [154] architecture based on primarily Attention layers, spurred the development of general purpose language encoders like BERT [47], GPT-3 [26], etc. BERT specifically beat 11 NLU benchmarks when it was published. GPT-3 has also been used in several places in the industry via its API. The common aspect amongst these domains is the rapid growth in the model footprint (Refer to Figure 1), and the cost associated with training and deploying them.

Since deep learning research has been focused on improving the state of the art, progressive improvements on benchmarks like image classification, text classification, etc. have been correlated with an increase in the network complexity, number of parameters, the amount of training resources

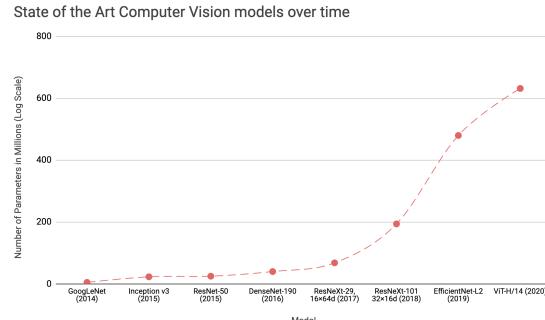
Author's address: Gaurav Menghani, gmenghani@google.com, Google Research, Mountain View, California, USA, 95054.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

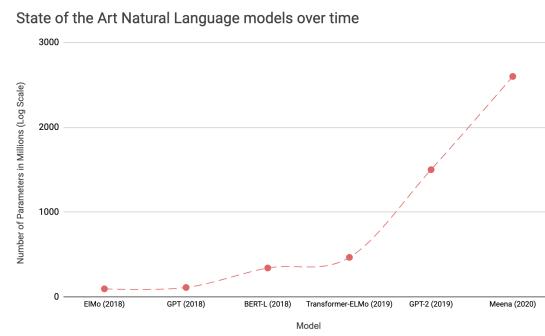
© 2021 Association for Computing Machinery.

XXXX-XXXX/2021/6-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>



(a) Computer Vision Models



(b) Natural Language Models

Fig. 1. Growth in the number of parameters in Computer Vision models over time. [118]

required to train the network, prediction latency, etc. For instance, GPT-3 comprises of 175 billion parameters, and costs millions of dollars to train just one iteration ([26]). This excludes the cost of experimentation / trying combinations of different hyper-parameters, which is also computationally expensive.

While 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. While training could be a one-time cost (or could be free if one is using a pre-trained model), deploying and letting inference run for over a long period of time could still turn out to be expensive in terms of consumption of server-side RAM, CPU, etc.. There is also a very real concern around the carbon footprint of datacenters even for organizations like Google, Facebook, Amazon, etc. which spend several billion dollars each per year in capital expenditure on their data-centers.
- **Enabling On-Device Deployment:** Certain deep learning applications need to run realtime 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 & 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 lesser 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:** While 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.

1.1 Efficient Deep Learning

The common theme around the above challenges is *efficiency*. We can break it down further as follows:

- **Inference Efficiency:** This primarily deals with questions that someone deploying a model for inference (computing the model outputs for a given input), would ask. Is the model small? Is it fast, etc.? More concretely, how many parameters does the model have, what is the disk size, RAM consumption during inference, inference latency, etc.
- **Training Efficiency:** This involves questions someone training a model would ask, such as How long does the model take to train? How many devices? Can the model fit in memory?, etc. It might also include questions like, how much data would the model need to achieve the desired performance on the given task?

If we were to be given two models, performing equally well on a given task, we might want to choose a model which does better in either one, or ideally both of the above aspects. If one were to be deploying a model on devices where inference is constrained (such as mobile and embedded devices), or expensive (cloud servers), it might be worth paying attention to inference efficiency. Similarly, if one is training a large model from scratch on either with limited or costly training resources, developing models that are designed for training efficiency would help.

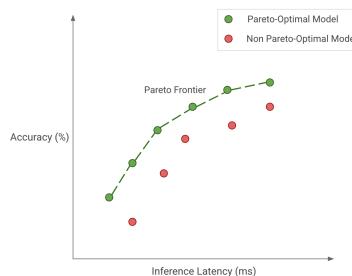


Fig. 2. Pareto Optimality: Green dots represent pareto-optimal models (together forming the pareto-frontier), where none of the other models (red dots) get better accuracy with the same inference latency, or the other way around.

Regardless of what one might be optimizing for, we want to achieve *pareto-optimality*. This implies that any model that we choose is the best for the tradeoffs that we care about. As an example in Figure 2, 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*. 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.

To achieve this goal, we propose turning towards a collection 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.

2 A MENTAL MODEL

In this section we present the mental model to think about the collection of algorithms, techniques, and tools related to efficient deep learning. We propose to structure them in five major areas, with the first four focused on modeling, and the final one around infrastructure and tools.

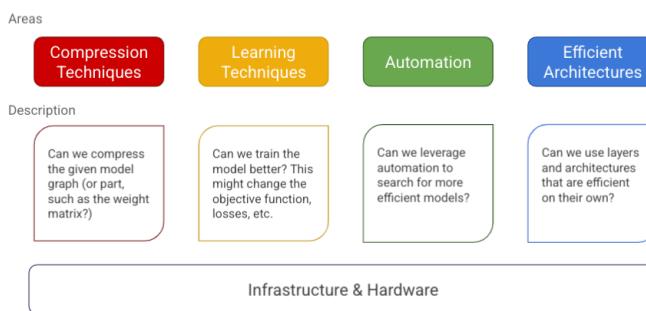


Fig. 3. A mental model for thinking about algorithms, techniques, and tools related to efficiency in Deep Learning.

- (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 [82], which tries to compress the weight matrices of a layer, by reducing its precision (eg., from 32-bit floating point values to 8-bit unsigned integers), with minimal loss in quality.
- (2) **Learning Techniques:** These are algorithms which 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 [75], 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) [61] where optimizing the hyper-parameters helps increase the accuracy, which could then be exchanged for a model with lesser parameters. Similarly, architecture search [167] 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, model size, etc.
- (4) **Efficient Architectures:** These are fundamental blocks that were designed from scratch (convolutional layers, attention, etc.), that are a significant leap over the baseline methods used before them (fully connected layers, and RNNs 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 [21] solved the problem of Information Bottleneck in 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], PyTorch [119], etc. (along with the tools required specifically for deploying efficient models such as 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 will 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. The efficiency goal could be to optimize the model for one or more of the footprint metrics, such as model size, inference latency, training time required for convergence, etc. 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. 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.

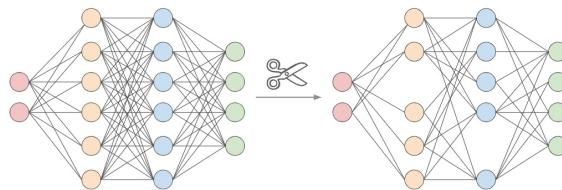


Fig. 4. A simplified illustration of pruning weights (connections) and neurons (nodes) in a neural network comprising of fully connected layers.

Some of the classical works in this area are Optimal Brain Damage (OBD) by LeCun et al. [98], and Optimal Brain Surgeon paper (OBD) by Hassibi et al. [72]. These methods usually take a network that has been pre-trained to a reasonable quality and then iteratively prune the parameters which 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).

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' .

```

1  $W' \leftarrow W;$ 
2 for  $i \leftarrow 1$  to  $N$  do
3    $S \leftarrow \text{compute\_saliency\_scores}(W');$ 
4    $W' \leftarrow W' - \text{select\_min\_k}(S, \frac{|W'|}{p});$ 
5    $W' \leftarrow \text{fine\_tune}(X, W')$ 
6 end
7 return  $W'$ 
```

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. demonstrate that pruning reduced the parameters in a well-trained neural net by 8x (combination of both automatic and manual removal) without a drop in classification accuracy.

Across different pruning strategies, the core algorithm could remain similar, with changes in the following aspects.

- **Saliency:** While [72, 98] use second-order derivatives, other methods rely on simpler magnitude based pruning [68, 69], or momentum based pruning [46] etc. 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 (such as pruning row-wise in a weight matrix, or pruning channel-wise in a convolutional filter [5, 100, 106, 112], 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 then deciding which parameters to prune, or by smartly selecting how much to prune in each layer individually [50, 74]. [52, 66] have found that some architectures like MobileNetV2, EfficientNet 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 [69, 72, 98], or should we prune at a higher pace in the beginning and gradually decrease [46, 166].

Model Architecture	Sparsity Type	Sparsity %	FLOPs	Top-1 Accuracy %	Source
MobileNet v2 - 1.0	Dense (Baseline)	0%	1x	72.0%	Sandler et al. [133]
	Unstructured	75%	0.27x	67.7%	Zhu et al. [166]
	Unstructured	75%	0.52x	71.9%	Evcı et al. [54]
	Structured (block-wise)	85%	0.11x	69.7%	Elsen et al.
	Unstructured	90%	0.12x	61.8%	Zhu et al. [166]
	Unstructured	90%	0.12x	69.7%	Evcı et al. [54]

Table 1. A sample of various sparsity results on the MobileNet v2 architecture with depth multiplier = 1.0.

- **Regrowth:** Some methods allow regrowing pruned connections [46, 54] to keep the same level of sparsity through constant cycles of prune-redistribute-regrow. Dettmers et al. [46] estimate training time speedups between 2.7x - 5.6x by starting and operating with a sparse model throughout. However there is a gap in terms of implementation of sparse operations on CPU, GPU, and other hardware.

Beyond Model Optimization: Frankle et al.’s [57] work on the Lottery Ticket Hypothesis took a different look at pruning, and postulated that within every large network lies a smaller network, which can be extracted with the original initialization of its parameters, and retrained on its own to match or exceed the performance of the larger network. The authors demonstrated these results on multiple datasets, but others such as [58, 107] were not able to replicate this on larger datasets such as ImageNet [45]. Rather Liu et al. [107] demonstrate that the pruned architecture with random initialization does no worse than the pruned architecture with the trained weights.

Discussion: There is a significant body of work that demonstrates impressive theoretical reduction in the model size (via number of parameters), or estimates the savings in FLOPs (Table 1). However, a large fraction of the results are on *unstructured* pruning, where it is not currently clear how these improvements can lead to reduction in footprint metrics (apart from using standard file compression tools like GZip).

On the other hand, structured pruning with a meaningful block size is conducive to latency improvements. Elsen et al. [52, 66] construct sparse convolutional networks that outperform their dense counterparts by 1.3 - 2.4x with $\approx 66\%$ of the parameters, while retaining the same Top-1 accuracy. They do this via their library to convert from the NHWC (channels-last) standard dense representation to a special NCHW (channels-first) ‘Block Compressed Sparse Row’ (BCSR) representation which is suitable for fast inference using their fast kernels on ARM devices, WebAssembly etc. [18]. Although they also introduce some constraints on the kinds of sparse networks that can be accelerated [19]. Overall, this is a promising step towards practical improvements in footprint metrics with pruned networks.

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 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: (a) lower model size, and (b) 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 [13], where only the model weights are in reduced precision. In order to get latency improvements, the activations need to be in fixed-point as well (Activation Quantization [82, 152], such that all the 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 [90]) 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 5). 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 the 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).

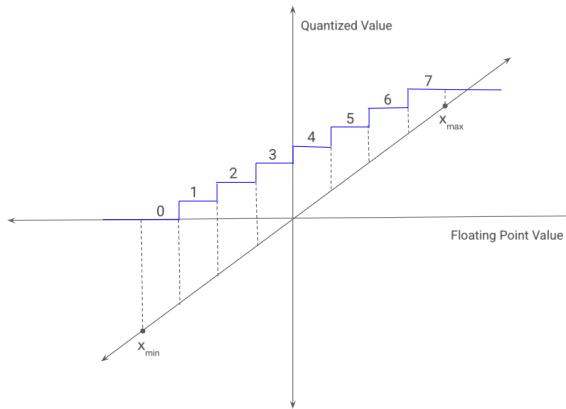


Fig. 5. Quantizing floating-point continuous values to discrete fixed-point values. The continuous values are clamped to the range x_{min} to x_{max} , and are mapped to discrete values in $[0, 2^b - 1]$ (in the above figure, $b = 3$, hence the quantized values are in the range $[0, 7]$).

[82, 90] 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 above 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:

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

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 literature [82, 90] (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:

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

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.

Algorithm 2: Quantizing a given weight matrix X

Data: Floating-point tensor to compress X , number of precision bits b for the fixed-point representation.

Result: Quantized tensor X_q .

- 1 $X_{min}, X_{max} \leftarrow \min(X, 0), \max(X, 0);$
 - 2 $X \leftarrow \text{clamp}(X, X_{min}, X_{max});$
 - 3 $s \leftarrow \frac{x_{max} - x_{min}}{2^b - 1};$
 - 4 $z \leftarrow \text{round}\left(x_{q_{min}} - \frac{x_{min}}{s}\right);$
 - 5 $X_q \leftarrow \text{round}\left(\frac{X}{s}\right) + z;$
 - 6 **return** $X_q;$
-

Algorithm 3: Dequantizing a given fixed-point weight matrix X_q

Data: Fixed-point matrix to dequantize X_q , along with the scale s , and zero-point z values which were calculated during quantization.

Result: Dequantized floating-point weight matrix \hat{X} .

- 1 $\hat{X} \leftarrow s(X_q - z);$
 - 2 **return** $\hat{X};$
-

We can utilize the above two algorithms 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 literature [13]. 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 literature [80, 99, 127] that demonstrate slightly different variants of quantization. XNOR-Net [127], Binarized Neural Networks [80] and others use $b = 1$, and thus have weight matrices which just have two possible values 0 or 1, and the quantization function there is simply the $\text{sign}(x)$ function (assuming the weights are symmetrically distributed around 0).

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 [92], VGG [141], Inception [146] etc., 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 [77, 133]. Additionally binary quantization (and other quantization schemes like ternary [99], bit-shift based networks [127], etc.) promise latency-efficient implementations of standard operations where multiplications and divisions are replaced by cheaper operations like addition, subtraction, etc. 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 which allow for low-level data parallelism (for example, on x86 via the SSE instruction set, on ARM via the Neon [108] intrinsics, and even on specialized DSPs like the Qualcomm Hexagon [19]).

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. [152] demonstrated a $3\times$ 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 [41], where commonly used building-block operations like the Multiply-Accumulate (MAC) [40] can be parallelized. Since the paper was published, Intel has released two more iterations of these instruction sets [37] which might further improve the speedups.

Quantization-Aware Training (QAT): The network that Vanhoucke et al. mention was a 5 layer feed-forward network that was post-training quantized. However post-training quantization can lead to quality loss during inference as highlighted in [82, 90, 155] as the networks become more complex. These could be because of: (a) outlier weights that skew the computation of the quantized values for the entire input range towards the outliers, leading to less number of bits being allocated to the bulk of the range, or (b) Different distribution of weights within the weight matrix, for eg. within a convolutional layer the distribution of weights between each filter might be 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. Wang et al. [155] try to retain the post-training quantization but with new heuristics to allocate the precision bits in a learned fashion. Tools like the TFLite Converter [148] augment post-training quantization with a representative dataset provided by the user, to actively correct for errors at different points in the model by comparing the error between the activations of the quantized and unquantized graphs.

Jacob et al. [82] propose (and further detailed by Krishnamoorthi et al. [90]) a training regime which 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 [82, 90]).

Assuming \mathbf{X} is the tensor to be fake-quantized, Jacob et al. [82] propose adding special quantization nodes in the training graph that collect the statistics (moving averages of x_{min} and x_{max}) related to the weights and activations to be quantized (see Figure 6(a) for an illustration). Once we have these values for each \mathbf{X} , we can derive the respective $\widehat{\mathbf{X}}$ using equations (1 and 2) as follows.

$$\begin{aligned}
 \widehat{\mathbf{X}} &= \text{FakeQuant}(\mathbf{X}) \\
 &= \text{Dequantize}(\text{Quantize}(\mathbf{X})) \\
 &= s\left(\left(\text{round}\left(\frac{\text{clamp}(\mathbf{X}, x_{min}, x_{max})}{s}\right)\right) + z\right) - z \\
 &= s\left(\text{round}\left(\frac{\text{clamp}(\mathbf{X}, x_{min}, x_{max})}{s}\right)\right)
 \end{aligned} \tag{3}$$

Since the above equation is not directly differentiable because of the rounding behavior, to optimize a loss function L w.r.t. \mathbf{X} , we can compute $\frac{\partial L}{\partial \mathbf{X}}$ by chain-rule using the Straight-Through Estimator (STE) [22]. This allows us to make the staircase function differentiable with a linear approximation (See [90] for details).

Quantization-Aware Training 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

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

Table 2. A sample of various quantization results on the MobileNet v2 architecture for 8-bit quantization [149]. We picked results on 8-bit, since from they can be readily used with hardware and software that exists today.

the TFLite Model Converter [15] can generate the appropriate fixed-point inference model from a network annotated with the quantization nodes.

Other Notable Works: Polino et al. [124] allow non-uniform distribution of precision with learning a vector of quantization-points p , along with using distillation to further reduce loss of accuracy. The results for simpler datasets like CIFAR-10 are comparable to [82, 90]. However, when working with ResNet architecture on the ImageNet dataset, they achieve lower model size and faster inference by using shallower student networks. This is not a fair comparison, since other works do not mix distillation along with quantization. Fan et al. [55] demonstrate accuracy improvement on top of standard QAT ([82]) with $b < 8$. They hypothesize that the networks will learn better if the fake-quantization is not applied to the complete tensor at the same time to allow unbiased gradients to flow (instead of the STE approximation). Instead, they apply the fake-quantization operation stochastically in a block-wise manner on the given tensor. They also demonstrate improvements over QAT on 4-bit quantized Transformer and EfficientNet networks.

Results: Refer to Table 2 for a comparison between the baseline floating-point model, post-training quantized, and quantization-aware trained models [13]. The model with post-training quantization gets close to the baseline, but there is still a significant accuracy difference. The model size is $4\times$ smaller, however the latency is slightly higher due to the need to dequantize the weights during inference. The model with 8-bit Quantization-Aware Training (QAT) gets quite close to the baseline floating point model while requiring $4\times$ less disk space and being $1.64\times$ faster.

Discussion:

- Quantization is a well-studied technique for model optimization and can help with very significant reduction in model size (often $4\times$ when using 8-bit quantization) and inference latency.
- Weight quantization is straight-forward enough that it can be implemented by itself for reducing model size. Activation quantization should be strongly considered because it enables both latency reduction, as well as lower working memory required for intermediate computations in the model (which is essential for devices with low memory availability)
- When possible, Quantization-Aware Training should be used. It has been shown to dominate post-training quantization in terms of accuracy.
- However, tools like Tensorflow Lite have made it easy to rely on post-training quantization. [148] shows that often there is minimal loss when using post-training quantization, and with the help of a representative dataset this is further shrunk down. Wherever there is an opportunity for switching to fixed-point operations, the infrastructure allows using them.
- For performance reasons, it is best to consider the common operations that follow a typical layer such as Batch-Norm, Activation, etc. and ‘fold’ them in the quantization operations.

3.1.3 Other Compression Techniques. There are other compression techniques like Low-Rank Matrix Factorization, K-Means Clustering, Weight-Sharing etc. which are also actively being used for model compression [117] and might be suitable for further compressing hotspots in a model.

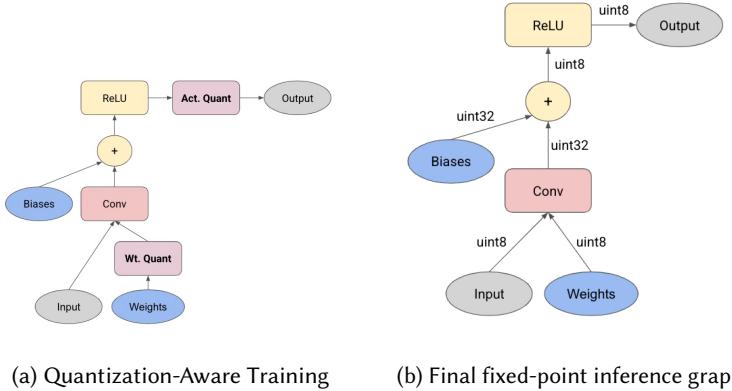


Fig. 6. (a) shows the injection of fake-quantization nodes to simulate quantization effect and collecting tensor statistics, for exporting a fully fixed-point inference graph. (b) shows the inference graph derived from the same graph as (a). Inputs and weights are in uint8, and results of common operations are in uint32. Biases are kept in uint32 [82, 90]

3.2 Learning Techniques

Learning techniques try to train a model differently in order to obtain better quality metrics (accuracy, F1 score, precision, recall, etc.) while allowing supplementing, or in some cases replacing the traditional supervised learning. 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. An incentive of paying attention to learning techniques is that they are applied only on the training, without impacting the inference.

3.2.1 Distillation. Ensembles are well known to help with generalization [71, 93]. The intuition is that this enables learning multiple independent hypotheses, which are likely to be better than learning a single hypothesis. [48] 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), averaging (voting by all the ensemble models), etc. Bucila et al. [27] 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 1000× smaller and faster. This demonstrated that it is possible to transfer the cumulative knowledge of ensembles to a single small model. Though it might not be sufficient to rely on just the existing labeled data.

Hinton et al. [75], 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 which 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 X , and the teacher model’s logits are $Z^{(t)}$, the teacher model’s softened probabilities with temperature T can be calculated as follows using the familiar softmax function:

$$Y_i^{(t)} = \frac{\exp(Z_i^{(t)}/T)}{\sum_{j=1}^n \exp(Z_j^{(t)}/T)} \quad (4)$$

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 (X, Y), and the student model’s output ($Y^{(s)}$), we can describe the loss function as:

$$\begin{aligned} L &= \lambda_1 \cdot L_{\text{ground-truth}} + \lambda_2 \cdot L_{\text{distillation}} \\ &= \lambda_1 \cdot \text{CrossEntropy}(Y, Y^{(s)}; \theta) + \lambda_2 \cdot \text{CrossEntropy}(Y^{(t)}, Y^{(s)}; \theta) \end{aligned} \quad (5)$$

`CrossEntropy` is the cross-entropy loss function, which takes in the labels and the output. For the first loss term, we pass along the ground truth labels, and for the second loss term we pass the 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. Refer to Figure 7 for an illustration of this process.

In the paper, Hinton et al. [75] 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. [151] did a comprehensive study demonstrating that distillation significantly improves performance of shallow student networks as small as an MLP with one hidden layer on tasks like CIFAR-10. Sanh et al. [134] use the distillation loss for compressing a BERT [47] model (along with a cosine loss that minimizes the cosine distance between two internal vector representation of the input as seen by the teacher and student models). Their model retains 97% of the performance of BERT-Base 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 et al. [164] transfer intermediate ‘attention maps’ between teacher and student convolutional networks. The intuition is to make the student focus on the parts of the image where the teacher is paying attention to. MobileBERT [144] 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 $4.3\times$ smaller and $5.5\times$ faster BERT with small losses in quality.

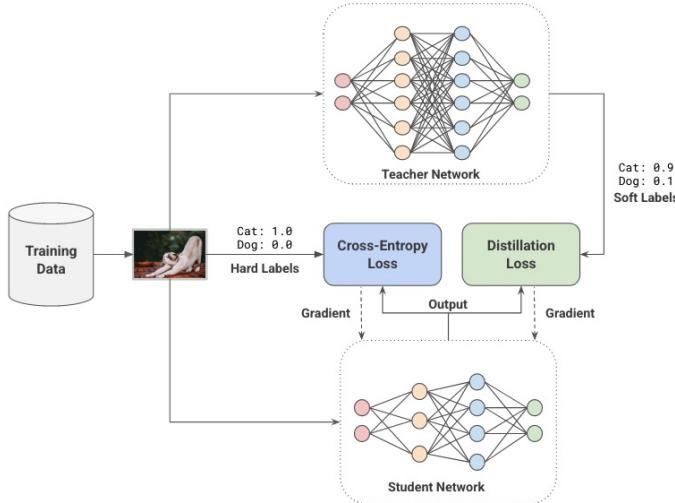


Fig. 7. Distillation of a smaller student model from a larger pre-trained teacher model. Both the teacher and student models receive the same input. The teacher is used to generate ‘soft-labels’ for the student, which gives the student more information than just hard binary labels. The student is trained using the regular cross-entropy loss with the hard labels, as well as using the distillation loss function which uses the soft labels from the teacher. In this setting, the teacher is frozen, and only the student receives the gradient updates.

Another idea that has been well explored is exploiting a model trained in a supervised training to label unlabeled data. Blum et al. [24] 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 [109, 160, 161].

Overall, distillation has been empirically shown to improve both the accuracy as well as the speed of convergence of student models across many domains. Hence, it enables training smaller models which might otherwise not be have an acceptable quality for deployment.

Discussion:

- Distillation is an adaptable technique that needs minimal changes in the training infrastructure to be used. Even if the teacher model cannot be executed at the same time as the student model, the teacher model’s predictions can be collected offline and treated as another source of labels.
- When there is sufficient label data, there is ample evidence that distillation is likely to improve the student model’s predictions. If there is a large corpus of unlabeled data, the teacher model can be used to generate pseudo-labels on the unlabeled data, which can further improve the student model’s accuracy.
- Strategies for intermediate-layer distillation have also shown to be effective in the case of complex networks. In such scenarios, a new loss term minimizing the difference between the outputs of the two networks at some semantically identical intermediate point(s) needs to be added.

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. [143] demonstrates 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.

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, etc. 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 as described above have long been demonstrated to improve accuracy of convolutional networks [36, 140]. They have also been a core part of seminal works in Image Classification. A prime example is AlexNet [92], where such transformations were used to increase the effective size of the training dataset by $2048 \times$, which won the ImageNet competition in 2012. Since then it has become common to use such transformations for Image Classification models (Inception [146], Xception [34], ResNet [73], etc.).

We can categorize data-augmentation methods as follows (also refer to Figure 8):

- **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, shearing, etc. However the user has to verify the label-invariance property with each transformation for the specific task at hand.
- **Label-Mixing Transformations:** Transformations such as Mixup [165], mixes 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 the classes. Other transformations like Sample Pairing also seem to help [81].
- **Data-Dependent Transformations:** In this case, transformations are chosen such that they maximize the loss for that example [56], or are adversarially chosen so as to fool the classifier [67].
- **Synthetic Sampling:** These methods synthetically create new training examples. Algorithms like SMOTE [30] allow re-balancing the dataset to make up for skew in the datasets, and GANs can be used to synthetically create new samples [166] 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 stacked.

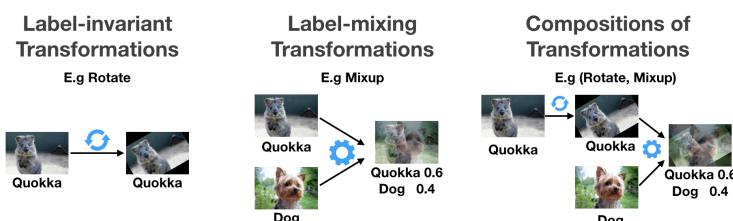


Fig. 8. Some common types of data augmentations. Source: [102]

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

Table 3. A breakdown of the contribution of various transformations on the validation accuracy of a model trained on the CIFAR-10 dataset. Source: [44].

Discussion: Apart from Computer Vision, Data-Augmentation has also been used in NLP, and Speech. In NLP, a common idea that has been used is ‘back-translation’ [162] where augmented examples are created by training two translation models, one going from the source language to the target language, and the other going back from the target language to the original source language. Since the back-translation is not exact, this process is able to generate augmented samples for the given input. Other methods like WordDropout [139] stochastically set embeddings of certain words to zero. SwitchOut [157] introduces a similarity measure to disallow augmentations that are too dissimilar to the original input. In Speech [70], the input audio samples are translated to the left / right before being passed to the decoder.

While the augmentation policies are usually hand-tuned, there are also methods such as AutoAugment [43] where the augmentation policy is learned through a Reinforcement-Learning (RL) based search, searching for the transformations to be applied, as well as their respective hyper-parameters. Though this is shown to improve accuracy, it is also complicated and expensive to setup a separate search for augmentation, taking as many as 15000 GPU hours to learn the optimal policy on ImageNet. The RandAugment [44] paper demonstrated that it is possible to achieve similar results while reducing the search space to just two hyper-parameters (number of augmentation methods, and the strength of the distortion) for a given model and dataset.

Overall, we see that data-augmentation leads to better generalization of the given models. Some techniques can be specific for their domains RandAugment (Vision), BackTranslation and SwitchOut (NLP), etc. However, the core principles behind them make it likely that similar methods can be derived for other domains too (refer to our categorization of data-augmentation methods above).

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

Self-Supervised learning (SSL) 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 (Refer to Figure 9 for examples). Since unlabeled data is vast in many domains (Books, Wikipedia, and other text for NLP, Web Images & 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

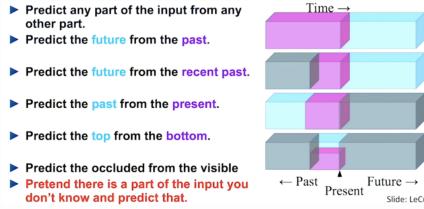


Fig. 9. General theme of pretext tasks. Source: [96]

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.) ([47, 78]).

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 NLP community. ULMFiT [78] 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 (IMDb dataset) required only 100 labeled examples ($\approx 10\times$ less labeled examples otherwise). Refer to Figure 10. 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 [47] (and other related models like GPT, RoBERTa, T5, etc.) where the pre-training steps involves learning to solve two tasks. Firstly, 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 , 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. When it was published, BERT beat the State-of-the-Art on eleven NLP tasks.

Similar to NLP, the pretext tasks in Vision have been used to train models that learn general representations. [49] extracts two patches from a training example and then trains the model to predict their relative position in the image (Refer to Figure 11(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

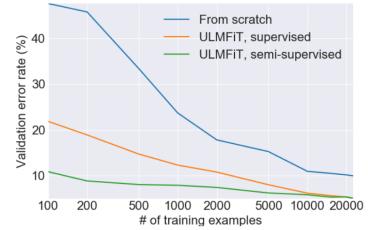
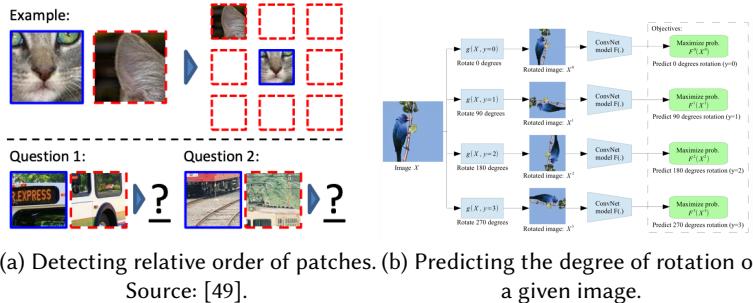


Fig. 10. Validation Error w.r.t. number of training examples for different training methods on IMDb (from scratch, ULMFiT Supervised: pre-training with WikiText-103 and fine-tuning using labeled data, ULMFiT Semi-Supervised: Pre-Training with WikiText-103 as well as unlabeled data from the target dataset and fine-tuning with labeled data). Source: [78]



(a) Detecting relative order of patches. (b) Predicting the degree of rotation of

Source: [49].

Fig. 11. Pretext tasks for vision problems.

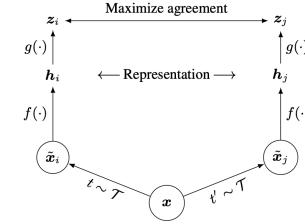
of rotation for a given rotated image [59]. The authors report that the network trained in a self-supervised manner this way can be fine-tuned to perform nearly as well as a fully supervised network.

Another common theme is Contrastive Learning, where the model is trained to distinguish between similar and dissimilar inputs. Frameworks such as SimCLR [32, 33], try to learn representations h_i and h_j for two given inputs \tilde{x}_i and \tilde{x}_j , where the latter two are differently augmented views of the same input, such that the cosine similarity of the projections of h_i and h_j , z_i and z_j (using a separate function $g(\cdot)$) can be maximized. Similarly, for dissimilar inputs the cosine similarity of z_i and z_j should be minimized. The authors report a Top-1 accuracy of 73.9% on ImageNet with only 1% labels (13 labels per class), and outperform the ResNet-50 supervised baseline with only 10% labels.

Discussion: Self-Supervised Learning (SSL) has demonstrated significant success in the general representational learning with unlabeled data, followed by fine-tuning to adapt the model to the target task with a modest number of labeled examples. Yann LeCun has likened Self-Supervision as the cake, and Supervised Learning as the icing on top [96], implying that SSL will be the primary way of training high-quality models in the future as we move beyond tasks where labeled data is abundant.

With unlabeled data being practically limitless, SSL's success is dependent on creating useful pretext tasks for the domain of interest. As demonstrated across NLP [47, 78], Vision [32, 120], Speech [60], etc., Self-Supervision is indeed not just helpful in speeding and improving convergence, but also enabling achieving high quality in tasks where it was intractable to get enough labeled samples.

Practically, for someone training Deep Learning models on a custom task (say a speech recognition model for a remote African dialect), having a pre-trained checkpoint of a model trained in a self-supervised fashion (such as wav2vec [20], which pre-trained in a similar way to BERT [47]), enables them to only spend an extremely tiny fraction of resources on both data labeling, as well as training to fine-tune to a good enough quality. In some cases, such as SimCLR [33], SSL approaches have actually beaten previous supervised baselines with sophisticated models like ResNet-50. Hence, we are hopeful that SSL methods will be crucial for ML practitioners for training high-quality models cheaply.

Fig. 12. SimCLR framework for learning visual representations.
Source: [32]

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, apart from systematically and automatically looking for optimal solutions. The trade-off is that these methods might require large computational resources, and hence have to be carefully applied.

3.3.1 Hyper-Parameter Optimization (HPO). One of the commonly used methods that fall under this category is Hyper-Parameter Optimization (HPO) [163]. Hyper-parameters such as initial learning rate, weight decay, etc. have to be carefully tuned for faster convergence [85]. They can also decide the network architecture such as the number of fully connected layers, number of filters in a convolution layer, etc.

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 learnt 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,

$$\lambda^* = \underset{\lambda \in \Lambda}{\operatorname{argmin}} \mathcal{L}(f_\lambda(x; \theta), y) \quad (6)$$

Λ is the set of all possible hyper-parameters. In practice, the 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*, $S = \{\lambda^{(1)}, \lambda^{(2)}, \dots, \lambda^{(n)}\}$, such that $S \subseteq \Lambda$, and then we can approximate Equation (6) with:

$$\lambda^* \approx \underset{\lambda \in \{\lambda^{(1)}, \dots, \lambda^{(n)}\}}{\operatorname{argmin}} \mathcal{L}(f_\lambda(x; \theta), y) \quad (7)$$

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. 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 grow very quickly.

Another approach is **Random Search** where trials are sampled randomly from the search space [23]. Since each trial is independent of the others, it can still be executed randomly. 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) Likelihood of finding the optimal λ^* increases with the number of trials, which is not the case with Grid Search.

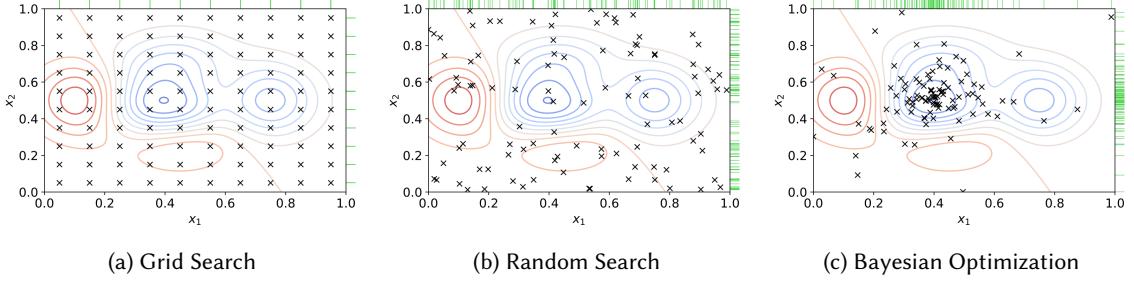


Fig. 13. Hyper-Parameter Search algorithms. Source: [39]

- (3) If there are K real-valued hyper-parameters, and N total trials, grid search would pick $N^{\frac{1}{K}}$ 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 (BO) based search [2, 111] 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, and 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* which picks the next trial using the surrogate function, the likelihood of improving on the optimum so far, whether to explore / exploit etc. As the trials complete, both 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. Also, since the selection of trials depends on the results of the past trials, this method is sequential. BO 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 (though 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 above search algorithms is the **Early Stopping** of trials that are not promising. Google's Vizier [61] uses Median Stopping Rule for early stopping, where a trial is terminated if its performance at a time step t is below the median performance of all trials run till that point of time.

Other algorithms for HPO include:

- (1) **Population Based Training (PBT)** [83]: 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 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. This is the *exploitation* part of the search. For *exploration*, these hyper-parameters are perturbed from their original values. This process repeats till convergence. It combines both the search and training in a fixed number of trials that run in parallel. It also only works with adaptive hyper-parameters like learning rate, weight-decay, etc. but 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 (SHA) [84] and HyperBand [101] are similar to random search, but they allocate more resources to the trials which

are performing well. Both 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 worse 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 SHA, η is 2, so the bottom $\frac{1}{2}$ of the trials are dropped, and the training budget for the remaining trials is doubled. For Hyper-Band η is 3 or 4. Hyper-Band differs from SHA in that the user does not need to specify the maximum number of parallel trials, which introduces a trade-off between the total budget and the per-trial allocation.

HPO Toolkits: There are several software toolkits that incorporate HPO algorithms as well as an easy to use interface (UI, as well as a way to specify the hyper-parameters and their ranges). Vizier [61] (an internal Google tool, also available via Google Cloud for blackbox tuning). Amazon offers Sagemaker [122] which is functionally similar and can also be accessed as an AWS service. NNI [131], Tune [103], Advisor [31] are other open-source HPO software packages that can be used locally.

3.3.2 Neural Architecture Search (NAS). Neural Architecture Search can be thought of an extension of Hyper-Parameter Optimization wherein we are searching for parameters that change the network architecture itself.

We find that there is consensus in the literature [53] around categorizing NAS as a system comprising of the following parts:

- (1) **Search Space:** These are the operations that are allowed in the graph (Convolution ($1 \times 1, 3 \times 3, 5 \times 5$), Fully 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 & 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 (RL) [167], and Gradient Descent [105] are popular alternatives too.
- (3) **Evaluation Strategy:** This defines how we evaluate a model for fitness. It can simply be a conventional metric like validation loss, accuracy, etc. Or it can also be a compound metric, as in the case of MNasNet [147] which creates a single custom metric based on accuracy as well as latency.

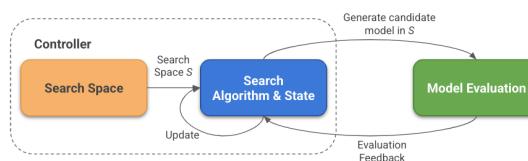


Fig. 14. Neural Architecture Search: The controller can be thought of as a unit that encodes the search space, the search algorithm itself, and the state it maintains (typically the model that helps generate the candidates). The algorithm generates candidate models in the search space S , and receives an evaluation feedback. This feedback is used to update the state, and generate better candidate models.

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’ which generates sample candidate networks

(Refer to Figure 14). 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. While the implementation of each of these blocks vary, this structure is common across the seminal work in this area.

Zoph et. al's paper from 2016 [167], demonstrated that end-to-end neural network architectures can be generated using Reinforcement Learning. In this case, the controller is a Recurrent Neural Network, which generates the architectural hyper-parameters of a feed-forward network one layer at a time, for example, number of filters, stride, filter size, etc. They also support adding skip connections (refer Figure 15). The network semantics are baked into the controller, so generating a network that behaves differently requires changing the controller. Also, training the controller itself is expensive (taking 22,400 GPU hours [168]), since the entire candidate network has to be trained from scratch for a single gradient update to happen. In a follow up paper [168], they 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 process 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 1 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 $7\times$, when compared to the end-to-end network search in [167], while beating the state-of-the-art in CIFAR-10 at that time.

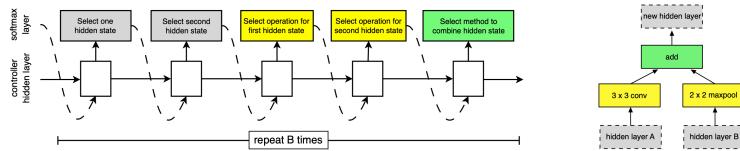


Fig. 15. A NASNet controller generating the architecture, recursively making one decision at a time and generating a single block in the image (making a total of 5 decisions). Source: [168].

Other approaches such as evolutionary techniques [130], differentiable architecture search [105], progressive search [104], parameter sharing [123], etc. 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 [53].

While 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 [147] is one such work. It incorporates the model's latency on the target device into the objective function directly, as follows:

$$\underset{m}{\text{maximize}} \quad ACC(m) \times \left[\frac{LAT(m)}{T} \right]^w \quad (8)$$

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 [159]

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 device, they use a pre-computed lookup table to approximate the latency to speed up the search process. They achieve networks that are upto $2.4\times$ smaller and $1.5\times$ faster than MobileNet, while finishing the search in 216 GPU Hours. Other works such as MONAS [79] 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.

Discussion: Automation plays a critical role in model efficiency. Hyper-Parameter Optimization (HPO) is now a natural step in training models and can extract significant quality improvements, while minimizing human involvement. In case the cost HPO becomes large, algorithms like Bayesian Optimization, Hyper-Band etc. with early stopping techniques can be used. HPO is also available in ready-to-use software packages like Tune [103], Vizier via Google Cloud [61], NNI [131], etc. Similarly, recent advances in Neural Architecture Search (NAS) also make it feasible to construct architectures in a learned manner, while having constraints on both quality and footprint [147]. Assuming several hundred GPU hours worth of compute required for the NAS run to finish, and an approx cost of \$3 GPU / hour on leading cloud computing services, this makes using NAS methods financially feasible and not similar in cost to manual experimentation with model architecture when optimizing for multiple objectives.

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 which might lead to a design that is better in general, or it might be better suited for the specific task. In this section, we lay out an examples of such efficient layers and models to illustrate this idea.

3.5 Vision

One of the classical example of efficient layers in the Vision domain are the Convolutional Layers, which improved over Fully Connected (FC) layers in Vision models. FC layers suffer from two primary issues:

- (1) 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) Secondly, 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 3 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.

These learned features are simpler in lower layers (such as edges (horizontal, vertical, diagonal, etc.)), and more complex in subsequent layers (texture, shapes, etc.). This happens because the subsequent layers use the feature maps generated by previous layers, and each pixel in the input feature map of the i -th layer, depends on the past $i - 1$ layers. This increases the *receptive field* of

the said pixel as i increases, progressively increasing the complexity of the features that can be encoded in a filter.

The core idea behind the efficiency of Convolutional Layers is that the same filter is used everywhere in the image, regardless of where the filter is applied. Hence, enforcing spatial invariance while sharing the parameters. Going back to the example of a 100×100 RGB image with 3 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 allow 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 2-D 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 [97] was the first Convolutional Network which included convolutional layers, pooling, etc. Subsequently, many iterations of these networks have been proposed with various improvements. AlexNet [92], Inception [146], ResNet [73], etc. 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 $s_x \times s_y \times \text{input_channels}$, 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 depthwise in the z dimension.

Depth-separable convolution breaks this into two steps
(Refer to Figure 16):

- (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 ($1 \times 1 \times \text{input_channels} \times \text{output_channels}$) + ($s_x \times s_y \times \text{output_channels}$), v/s $s_x \times s_y \times \text{input_channels} \times \text{output_channels}$ 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 [133]). The Xception model architecture [34] 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 [133] 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 $7 - 10 \times$ and allows deployment on Mobile for Computer Vision tasks. Users can expect a latency between 10-100ms 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.

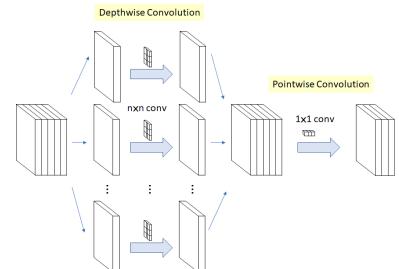


Fig. 16. Depth-Separable Convolution.
Source: [150].

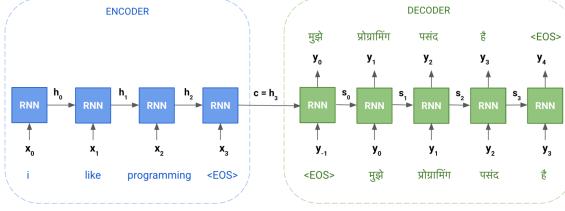


Fig. 17. Information Bottleneck in a Seq2Seq model for translating from English to Hindi. The context vector c that the decoder has access to is fixed, and is typically the last hidden state (h_T).

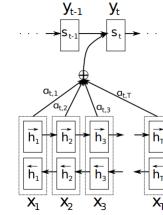


Fig. 18. Attention module learning a weighted context vector for each output token from the hidden states. Source: [21].

3.6 NLU

3.6.1 Attention Mechanism & Transformer Family. One of the issues plaguing classical Sequence-to-Sequence (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 which encode the given input sequence ($\mathbf{x} = (x_1, x_2, \dots, x_T)$) into a fixed length vector(s) (also referred to as the context, c), and one or more decoder layers which generate another sequence using this context. In the case of MT, the input 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 [145] 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 (Refer to Figure 17). 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 Bahdanau et al. [21] 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. Essentially, the new weighted context vector is $c_i = \sum_j^T \alpha_{ij} \cdot h_j$, 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 (c) over V . In the above 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 etc. [29]. We refer the reader to [29] for further details on the taxonomy of attention models.

The Transformer architecture [154] 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, i.e., learning long range dependencies is $O(n)$, where n is the number of tokens. With Self-Attention, all tokens

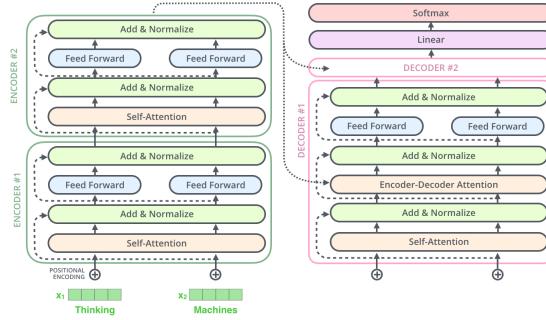


Fig. 19. Transformer with its Encoder and Decoder blocks. Source: [3].

are processed together and pairwise relationships can be learnt in $O(1)$ [154]. This makes it easier to leverage optimized training devices like GPUs and TPUs. The authors reported up to $300\times$ less training FLOPs as required to converge to a similar quality when compared to other recurrent and convolutional models.

As introduced earlier, the BERT model architecture [47] improved over several NLU tasks. BERT is a stack of Transformer encoder layers that are pre-trained using a bi-directional masked language model training objective. It can also be used as a general purpose encoder which can then be used for other tasks. Other similar models like the GPT family [26] have also been used for solving many NLU tasks.

3.6.2 Random Projection Layers & Models. Pre-trained token representations such as word2vec [110], GLoVe [121], etc. 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 [87, 88, 128, 129] 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 ($\mathbb{P}(x) = [\mathbb{P}_1(x), \dots, \mathbb{P}_T(x)]$, 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) [28, 129], each using a different hash function (via different seeds). For theoretical guarantees about the Random Projection operation, refer to [28], 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 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). On the other hand, 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 ([129]) v/s after the projection operation ([87, 135]),

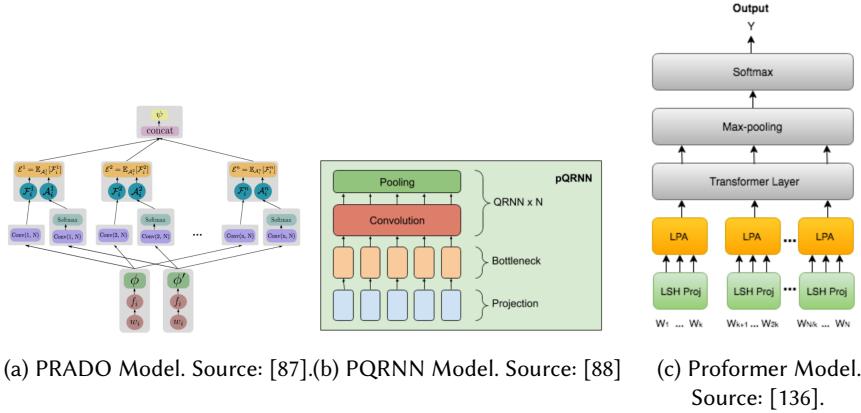


Fig. 20. Collection of notable Random-Projection based models.

generating a ternary representation instead of binary ([87, 88]), applying complex layers and networks on top like Attention ([87]), QRNN ([88]), etc.

Some of the Projection-based models (refer to Figure 20) have demonstrated impressive results on NLU tasks. PRADO ([87]) 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 \times smaller, and taking 20-40 ms for inference on a Nexus 5X device. PQRNN [88], another Projection-based model that additionally uses a fast RNN implementation (QRNN) [25] on top of the projected features. They report outperforming LSTMs while being 140 \times smaller, and achieving 97.1% of the quality of a BERT-like model while being 350 \times smaller.

Proformer [136] introduces a Local Projected Attention (LPA) Layer, which combines the Projection operation with localized attention. They demonstrate reaching performance ProFormer converges to \approx 97.2% BERT-base's performance on an average while occupying only 13% of BERT-base's memory. ProFormer also had 14.4 million parameters, compared to 110 million parameters of BERT-base.

3.7 Infrastructure

In order to be able to train and run inference efficiently, there has to be a robust software and hardware infrastructure foundation. In this section we go over both these aspects. Refer to Figure 21 for a mental model of the software and hardware infrastructure, and how they interact with each other. In this section we provide a non-exhaustive but comprehensive survey of leading software and hardware infrastructure components that are critical to model efficiency.

3.7.1 Tensorflow Ecosystem. Tensorflow (TF) [1, 14] 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.

Tensorflow Lite for On-Device Usecases: Tensorflow Lite (TFLite) [16] 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, along with implementations of common neural net operations (Fully Connected, Convolution, Max Pooling, ReLu, Softmax, etc. each of which as an *Op*). The implementation

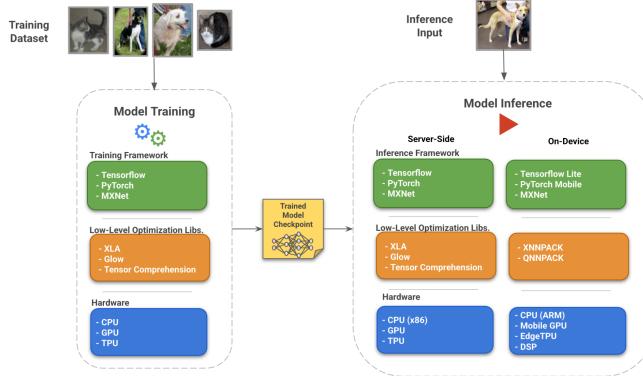


Fig. 21. A visualization of the hardware and software infrastructure with emphasis on efficiency. On the left hand side is the model-training phase, which generates a trained model checkpoint. This model is then used on the inference side, which could either be server-side (conventional machines in cloud or on-prem), or on-device (mobile phones, IoT, edge devices, etc.).

of such an operation is known as an *Op Kernel*. Both the interpreter and Op Kernels are primarily optimized for inference on ARM-based processors as of the time of writing this paper. They can also leverage smartphone DSPs such as Qualcomm's Hexagon [19] for faster execution. The interpreter also allows the user to set multiple threads for execution.

- **Converter:** The TFLite converter as the name suggests is useful for converting the given TF 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, etc. With respect to quantization, it also allows post-training quantization as mentioned earlier with an optional representative dataset to improve accuracy.

Other Tools for On-Device Inference: TF Micro [158] goes further, and consists of a slimmed down interpreter, and a smaller set of ops for inference on very low resource microcontrollers. TF Model Optimization toolkit [12] is a Tensorflow library for applying common compression techniques like quantization, pruning, clustering etc. TensorflowJS (TFJS) is a library within the TF ecosystem that can be used to train and run neural networks within the browser or using Node.js [113]. These models can also accelerated through GPUs via the WebGL interface [42]. It supports both, importing models trained in TF, as well as creating new models from scratch in TFJS.

XLA for Server-Side Acceleration: Typically a TF model graph is executed by TF's executor process and it uses standard optimized kernels for running it on CPU, GPU, etc. XLA [17] 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 which can be fused together are combined in a single composite op. This avoids having to do multiple costly writes to RM, when the operands can directly be operated on while they are still in cheaper caches. They authors report [89] a 7× increase in training throughput, and 5× increase in the maximum batch size that can be used for BERT training. This allows training a BERT model for \$32 on Google Cloud.

3.7.2 PyTorch Ecosystem. PyTorch [119] 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 Usecases: PyTorch also has a light-weight interpreter that enables running PyTorch models on Mobile [9], 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 [10], 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 [11], which might seem similar to Tensorflow’s XLA, but is actually a mechanism for generating a serializable intermediate representation (high-level IR, per [102]) of the model from the code in TorchScript [11], 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 [9]. This representation is analogous to the static inference mode graphs generated by TensorFlow. The alternative for XLA in the PyTorch world seem to be the Glow [132] and TensorComprehension [153] 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 [102].

PyTorch offers a model tuning guide [8], which details various options that ML practitioners have at their disposal. Some of the core ideas in there are:

- Turn on mixed-precision training [7] when using NVIDIA GPUs. This is described further in detail in the GPU sub-section in 3.5.4.
- Fusion of pointwise-operations (add, subtract, multiply, divide, etc.) using PyTorch JIT. Even though this should happen automatically, but adding the `torch.jit.script` decorator to methods which are completely composed of pointwise 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.).
- Enabling device-specific optimizations, such as the cuDNN library, and Mixed Precision Training with NVIDIA GPUs (explained in the GPU subsection).
- 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 replicas gradients are periodically accumulated and then averaged.

3.7.3 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 [108] 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 [51] and XNNPACK [18] libraries are optimized for ARM Neon for mobile and embedded devices, and for x86 SSE2, AVX architectures, etc. 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 isn’t 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 ML frameworks.

3.7.4 Hardware. GPU: Graphics Processing Units (GPUs) were originally designed for accelerating computer graphics, but began to be used for general-purpose usecases with the availability of the CUDA library [38] in 2007, and libraries like cuBLAS for speeding up linear algebra operations. In 2009, Raina et al. [125] demonstrated that GPUs can be used to accelerate deep learning models. In 2012, following the AlexNet model's [92] 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 [115, 142] which are dedicated execution units in their GPUs, which are specialized for Deep Learning applications. TensorCores 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 trade-off, since the difference in model quality might often be minimal.

Tensor Cores optimize the standard Multiply-and-Accumulate (MAC) operation [40], $A = (B \times C) + D$. Where, B and C are in a reduced precision (fp16, bfloat16, TensorFloat32), while A and 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 D . When training with reduced-precision, NVIDIA reports between

1 \times to 15 \times training speedup depending on the model architecture and the GPU chosen [142]. Tensor Cores in NVidia's latest Ampere architecture GPUs also support faster inference with sparsity (specifically, structured sparsity in the ratio 2:4, where 2 elements out of a block of 4 elements are sparse) [114]. They demonstrate an up to 1.5 \times speed up in inference time, and up to 1.8 \times speedup in individual layers. NVIDIA also offers the cuDNN libary [114] that contains optimized versions of standard neural network operations such as fully-connected, convolution, batch-norm, activation, etc.

TPU: 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-ML 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 [86]. Subsequent iterations of the TPU architectures enabled both training and inference with TPUs in floating point too. Google also opened up access to these TPUs via their Google Cloud service in 2018 [62].

The core architecture of the TPU chips leverages the Systolic Array design [94, 95] (refer to Figure 24), 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

$$\begin{matrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{matrix} = \left(\begin{matrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{matrix} \right) \times \left(\begin{matrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{matrix} \right) + \left(\begin{matrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{matrix} \right)$$

Fig. 22. Reduced Precision Multiply-Accumulate (MAC) operation: An illustration of the $A = (B \times C) + D$ operation. B and C are in a reduced precision (fp16, bfloat16, TensorFloat32 etc.), while A and D are in fp32. The speedup comes from doing the expensive matrix-multiplication with a reduced precision format.



Fig. 23. Common floating point format used in Training & Inference: fp32 is the standard 32-bit floating point number from IEEE-754 standard [156]. One bit is allocated for storing the sign. The exponent controls the range of the floating point value that can be expressed with that format, and the mantissa controls the precision. Note that fp16 reduces the precision as well as range. The bfloat16 format is a reasonable compromise because it keeps the same range as fp32 while trading off precision to take up a total of 16 bits. NVidia GPUs also support Tensor Float 32 format that allocates 3 more bits to the mantissa than bfloat16 to achieve better precision. However, it takes up a total of 19 bits which does not make it a trivially portable format.

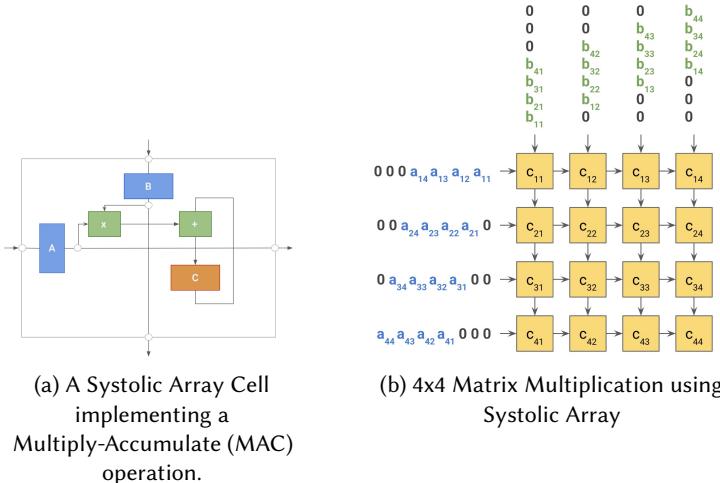


Fig. 24. Systolic Arrays in TPUs: Figure (a) shows a Systolic Array implementing a MAC operation, where the variables A and B are received by the cell, and C is the resident memory. A is passed to the horizontally adjacent cell on the right, and B is passed to the vertically adjacent cell below on the next clock tick. Figure (b) demonstrates how two 4×4 matrices are multiplied using Systolic Arrays which is a mesh of cells constructed in Figure (a). The i -th row of array is fed the i -th column of A (preceded by $i - 1$ 0s, which act as a delay). Similarly, the i -th column of the array is fed the i -th column of B (preceded by $i - 1$ 0s). The corresponding a_{ij} and b_{jk} are passed to the neighboring cells on the next clock tick.

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 4 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 [137].

TPUs have been used inside Google for applications like training models for Google Search, general purpose BERT models [47], for applications like DeepMind's world beating AlphaGo and AlphaZero models [138], and many other research applications [147]. They have also set model training time records in the MLPerf benchmarks. Similar to the GPUs, TPUs support the bfloat16 data-type [156] 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.

EdgeTPU: EdgeTPU is a custom ASIC chip designed by Google for running inference on edge devices, with low power requirements (4 Tera Ops / sec (TOPS) using 2 watts of power [64]). Like the TPU, it is specialized for accelerating linear algebra operations, but only for inference and with a much lower compute budget. It is further limited to only a subset of operations [65], and works only with int8 quantized Tensorflow Lite models. Google releases the EdgeTPU using the Coral platform in various form-factors, ranging from a Raspberry-Pi like Dev Board to independent solderable modules [63]. It has also been released with the Pixel 4 smartphones as the Pixel Neural Core [126], for accelerating on-device deep learning applications. The EdgeTPU chip itself is smaller than a US penny, making it amenable for deployment in many kinds of IoT devices.

Jetson: Jetson [116] is a family of accelerators by Nvidia to enable deep learning applications for embedded and IoT devices. It comprises of the Nano, which is a low-powered "system on a module" (SoM) designed for lightweight deployments, as well as the more powerful Xavier and TX variants, which are based on the NVidia Volta and Pascal GPU architectures. As expected, the difference within the Jetson family is primarily the type and number of GPU cores on the accelerators. This makes the Nano suited for applications like home automation, and the rest for more compute intensive applications like industrial robotics.

4 A PRACTITIONER'S GUIDE TO EFFICIENCY

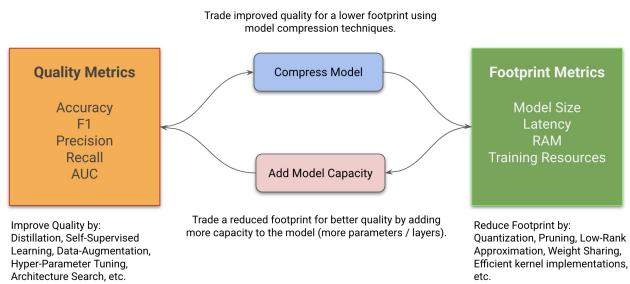


Fig. 25. Trade off between Model Quality and Footprint: There exists a trade-off between model quality and model footprint. Model quality can be improved with techniques like distillation, data-augmentation, hyper-param tuning etc. Compression techniques can in turn help trade off some model quality 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.

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,

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-Param Optimization, Architecture Search, etc.	Automation: Hyper-Param Optimization, Architecture Search, etc.
	Efficient Layers & Models: Projection, PQRNN, (NLP), Separable Convolution (Vision), etc.	Efficient Layers & Models: Transformers (NLP), Vi-T (Vision), etc.

Table 4. Examples of techniques to use in the Grow, Shrink, and Improve phases.

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, AUC, etc. While Footprint related metrics can include Model Size, Latency, RAM, etc.

Naturally, there exists a trade-off between Quality and Footprint metrics. A higher-capacity / deeper model is more likely to achieve a better accuracy, but at the cost of model size, latency, etc. On the other hand a model with lesser capacity / shallower, while possibly suitable for deployment, is also likely to be worse in accuracy. As illustrated in Figure 25, we can traverse from a model with better quality metrics, and exchange some of the quality for better footprint by naively compressing the model / reducing the model capacity (**Shrink**). On the other hand, 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 & Models, Architecture Search via Automation, etc. In addition, we can also **Improve** the quality metrics, while keeping the footprint same through Learning Techniques (Distillation, Data Augmentation, Self-Supervised Tuning), Hyper-Parameter Tuning, etc. (See Table 4 for more examples.)

Combining these three phases, we propose two strategies towards achieving pareto-optimal models:

- (1) **Shrink-and-Improve for Footprint-Sensitive Models:** If as a practitioner, you want to reduce your footprint, while keeping the 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, etc. and 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 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 trade-off that is appropriate for the user. Each efficiency technique individually helps move us closer to that target model.

4.1 Experiments

In order to demonstrate what we proposed above, 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 ML 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. In other words, provide empirical evidence that it is possible for practitioners to either reduce model capacity to bring down the footprint (shrink) and then recover the model quality that they traded off (improve), or increase the model capacity to improve quality (growing) followed by model compression (shrinking) to improve model footprint.

We picked the problem of classifying images in the CIFAR-10 dataset [91] on compute constrained devices such as smartphones, IoT devices etc. We picked a deep convolutional architecture where we could scale the model capacity up or down, by increasing or decreasing a ‘width multiplier’ (w) scalar. 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 such as Hyper-Parameter Tuning, Efficient Layers such as Separable Convolutions), but were skipped to keep the interpretation of the results simpler. We used the Tensorflow-backed Keras APIs [35] for training, and the TFLite [16] 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 IPython notebook [here](#).

Table 5 compiles the results for 6 width-multipliers in increasing order, ranging from 0.05 to 1.0. 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 $3.5 - 10\times$ 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 corresponding models in Table 5. We use Quantization for the Shrink phase, to reduce model size by $\approx 4\times$, and reduce the average latency by $1.5 - 2.65\times$. Figures 26 and 27 plot the notable results from Tables 5 and 6.

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	1374.11	83.04	87.47	88.03	11.46	2.8	1.33
0.75	764.87	2993.71	83.79	89.06	89.51	16.7	4.09	1.92
1	1343.01	5251.34	84.42	89.41	89.92	24	5.99	2.68

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	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	1343.01	1334.41	84.52	89.28	89.91	11.73	3.27	1.01

Table 6. Quality and Footprint metrics for Quantized models for the CIFAR-10 dataset. Each model is the quantized equivalent of the corresponding model in Table 5.

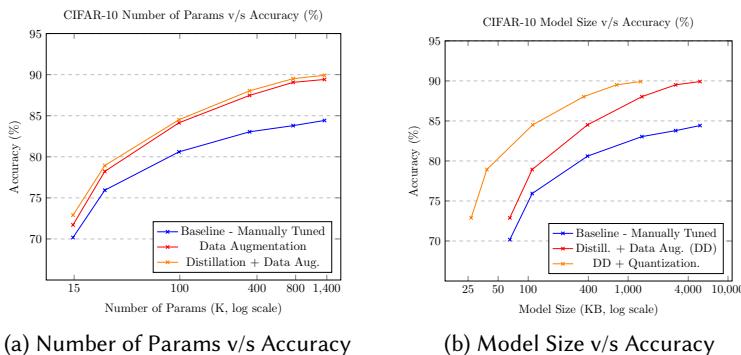


Fig. 26. Change in Accuracy with respect to Number of Params and Model Size. Each point on a curve is a model from Figure 5. The blue curve is a manually tuned baseline, the red curve includes Data Augmentation on top of baseline, and the orange curve is Distillation on top of Data Augmentation.

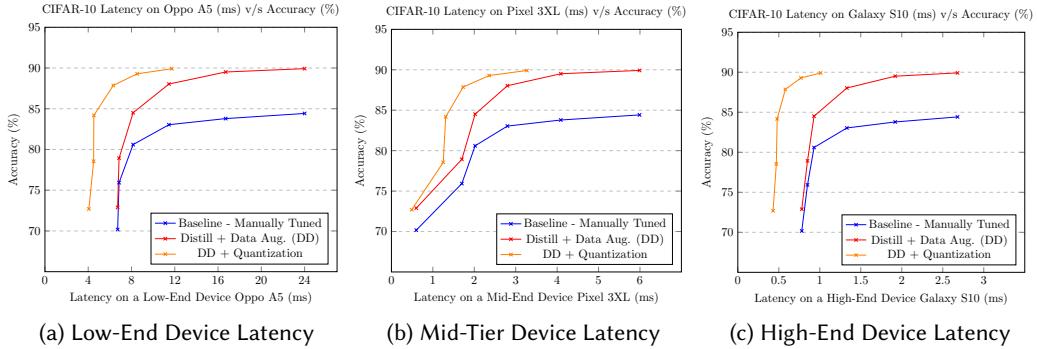


Fig. 27. Average latency of models on different devices (low-, mid-, and high-end smartphones). The orange curve denotes the quantized models in addition to being trained with distillation and data augmentation.

4.2 Discussion

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

Shrink-and-Improve for Footprint-Sensitive Models: Refer to Table 5 and Figure 26. If 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). 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 & 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 \times smaller, and the latency is 8.15 ms, which is 2.94 \times 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 \times smaller) and has a latency of 4.52 ms (5.31 \times faster), while retaining an accuracy of 84.18%. It is possible to do this for other pairs of points on the curves.

Grow-Improve-Shrink for Quality-Sensitive Models: Assuming 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 \times bigger), and latency to 1.33 ms (1.43 \times slower). However, we ignore that for a bit and improve our model's quality to 88.03% with Data Augmentation & 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 \times faster), with an accuracy of 87.86%, an absolute 7.10% increase in accuracy while keeping the model size approximately same and making it 1.6 \times faster. It is also possible to apply this strategy to other pairs of models.

Thus, we've verified that the above 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 26 and 27 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

& Distillation). Hence, it would be useful to explore other techniques as well such as Automation (for Hyper-Parameter Tuning to further improve on results), and Efficient Layers & Models (Separable Convolution as illustrated in MobileNet [133] could be used in place of larger convolutional layers). Finally, we would also like to emphasize paying attention to performance of Deep Learning models (optimized or not) on underrepresented classes and out-of-distribution data to ensure model fairness, since quality metrics alone might not be sufficient for discovering deeper issues with models [76].

5 CONCLUSION

In this paper, 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 modelling process, and/or use them as a starting point for exploration. The infrastructure section also lays out the software libraries and hardware which make training and inference of efficient models possible.

Finally, we presented a section of explicit and actionable insights supplemented by code, for a practitioner to use as a guide in this space. This section will hopefully give concrete and actionable takeaways, as well as tradeoffs to think about when optimizing a model for training and deployment. To conclude, we feel that with this survey we have equipped the reader with the necessary understanding to break-down the steps required to go from a sub-optimal model to one that meets their constraints for both quality as well as footprint.

REFERENCES

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.
- [2] Apoorv Agnihotri and Nipun Batra. 2020. Exploring bayesian optimization. *Distill* 5, 5 (2020), e26.
- [3] Jay Alammar. 2021. The Illustrated Transformer. <https://jalammar.github.io/illustrated-transformer> [Online; accessed 3. Jun. 2021].
- [4] Android Developers. 2021. Neural Networks API | Android NDK | Android Developers. <https://developer.android.com/ndk/guides/neuralnetworks> [Online; accessed 3. Jun. 2021].
- [5] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13, 3 (2017), 1–18.
- [6] Apple Authors. 2021. Accelerate | Apple Developer Documentation. <https://developer.apple.com/documentation/accelerate> [Online; accessed 3. Jun. 2021].
- [7] PyTorch Authors. 2021. Automatic Mixed Precision examples – PyTorch 1.8.1 documentation. https://pytorch.org/docs/stable/notes/amp_examples.html [Online; accessed 3. Jun. 2021].
- [8] PyTorch Authors. 2021. Performance Tuning Guide – PyTorch Tutorials 1.8.1+cu102 documentation. https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html [Online; accessed 3. Jun. 2021].
- [9] PyTorch Authors. 2021. PyTorch Mobile. <https://pytorch.org/mobile/home> [Online; accessed 3. Jun. 2021].
- [10] PyTorch Authors. 2021. Quantization Recipe – PyTorch Tutorials 1.8.1+cu102 documentation. <https://pytorch.org/tutorials/recipes/quantization.html> [Online; accessed 3. Jun. 2021].
- [11] PyTorch Authors. 2021. torch.jit.script – PyTorch 1.8.1 documentation. <https://pytorch.org/docs/stable/generated/torch.jit.script.html> [Online; accessed 3. Jun. 2021].
- [12] Tensorflow Authors. 2020. TensorFlow Model Optimization. https://www.tensorflow.org/model_optimization [Online; accessed 3. Jun. 2021].
- [13] Tensorflow Authors. 2021. Post-training quantization | TensorFlow Lite. https://www.tensorflow.org/lite/performance/post_training_quantization [Online; accessed 3. Jun. 2021].
- [14] Tensorflow Authors. 2021. TensorFlow. <https://www.tensorflow.org> [Online; accessed 3. Jun. 2021].

- [15] Tensorflow Authors. 2021. TensorFlow Lite converter. <https://www.tensorflow.org/lite/convert> [Online; accessed 3. Jun. 2021].
- [16] Tensorflow Authors. 2021. TensorFlow Lite | ML for Mobile and Edge Devices. <https://www.tensorflow.org/lite> [Online; accessed 3. Jun. 2021].
- [17] Tensorflow Authors. 2021. XLA: Optimizing Compiler for Machine Learning | TensorFlow. <https://www.tensorflow.org/xla> [Online; accessed 3. Jun. 2021].
- [18] XNNPACK Authors. 2021. XNNPACK. <https://github.com/google/XNNPACK> [Online; accessed 3. Jun. 2021].
- [19] XNNPACK Authors. 2021. XNNPACK backend for TensorFlow Lite. <https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/delegates/xnnpack/README.md/#sparse-inference> [Online; accessed 3. Jun. 2021].
- [20] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: A framework for self-supervised learning of speech representations. *arXiv preprint arXiv:2006.11477* (2020).
- [21] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
- [22] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432* (2013).
- [23] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, 2 (2012).
- [24] Avrim Blum and Tom Mitchell. 1998. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*. 92–100.
- [25] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. 2016. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576* (2016).
- [26] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
- [27] Cristian Bucilă, 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.
- [28] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 380–388.
- [29] Sneha Chaudhari, Varun Mithal, Gungor Polatkan, and Rohan Ramanath. 2019. An attentive survey of attention models. *arXiv preprint arXiv:1904.02874* (2019).
- [30] 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.
- [31] Dihao Chen. 2021. advisor. <https://github.com/tobegin3hub/advisor> [Online; accessed 3. Jun. 2021].
- [32] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.
- [33] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey Hinton. 2020. Big self-supervised models are strong semi-supervised learners. *arXiv preprint arXiv:2006.10029* (2020).
- [34] François Fleuret. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1251–1258.
- [35] Francois Fleuret. 2020. The Keras Blog. <https://blog.keras.io> [Online; accessed 4. Jun. 2021].
- [36] Dan C Cireşan, Ueli Meier, Jonathan Masci, Luca M Gambardella, and Jürgen Schmidhuber. 2011. High-performance neural networks for visual object classification. *arXiv preprint arXiv:1102.0183* (2011).
- [37] Contributors to Wikimedia projects. 2021. AVX-512 - Wikipedia. <https://en.wikipedia.org/w/index.php?title=AVX-512&oldid=1025044245> [Online; accessed 3. Jun. 2021].
- [38] Contributors to Wikimedia projects. 2021. CUDA - Wikipedia. <https://en.wikipedia.org/w/index.php?title=CUDA&oldid=1025500257> [Online; accessed 3. Jun. 2021].
- [39] Contributors to Wikimedia projects. 2021. Hyperparameter optimization - Wikipedia. https://en.wikipedia.org/w/index.php?title=Hyperparameter_optimization&oldid=1022309479 [Online; accessed 3. Jun. 2021].
- [40] Contributors to Wikimedia projects. 2021. Multiply-accumulate operation - Wikipedia. https://en.wikipedia.org/w/index.php?title=Multiply-accumulate_operation&oldid=1026461481 [Online; accessed 3. Jun. 2021].
- [41] Contributors to Wikimedia projects. 2021. SSE4 - Wikipedia. <https://en.wikipedia.org/w/index.php?title=SSE4&oldid=1023092035> [Online; accessed 3. Jun. 2021].
- [42] Contributors to Wikimedia projects. 2021. WebGL - Wikipedia. <https://en.wikipedia.org/w/index.php?title=WebGL&oldid=1026775533> [Online; accessed 3. Jun. 2021].
- [43] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. 2019. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 113–123.

- [44] 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.
- [45] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. <https://doi.org/10.1109/CVPR.2009.5206848>
- [46] Tim Dettmers and Luke Zettlemoyer. 2019. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840* (2019).
- [47] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [48] Thomas G Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*. Springer, 1–15.
- [49] 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.
- [50] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. 2017. Learning to prune deep neural networks via layer-wise optimal brain surgeon. *arXiv preprint arXiv:1705.07565* (2017).
- [51] Marat Dukhan, Yiming Wu Wu, and Hao Lu. 2020. QNNPACK: Open source library for optimized mobile deep learning - Facebook Engineering. <https://engineering.fb.com/2018/10/29/ml-applications/qnnpack> [Online; accessed 3. Jun. 2021].
- [52] 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.
- [53] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. 2019. Neural architecture search: A survey. *J. Mach. Learn. Res.* 20, 55 (2019), 1–21.
- [54] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. 2020. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*. PMLR, 2943–2952.
- [55] Angela Fan, Pierre Stock, Benjamin Graham, Edouard Grave, Rémi Gribonval, Hervé Jégou, and Armand Joulin. 2020. Training with quantization noise for extreme model compression. *arXiv e-prints* (2020), arXiv–2004.
- [56] Alhussein Fawzi, Horst Samulowitz, Deepak Turaga, and Pascal Frossard. 2016. Adaptive data augmentation for image classification. In *2016 IEEE international conference on image processing (ICIP)*. Ieee, 3688–3692.
- [57] Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Training pruned neural networks. *arXiv preprint arXiv:1803.03635* 2 (2018).
- [58] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The State of Sparsity in Deep Neural Networks. *CoRR* abs/1902.09574 (2019). [arXiv:1902.09574](https://arxiv.org/abs/1902.09574) <http://arxiv.org/abs/1902.09574>
- [59] Spyros Gidaris, Praveer Singh, and Nikos Komodakis. 2018. Unsupervised representation learning by predicting image rotations. *arXiv preprint arXiv:1803.07728* (2018).
- [60] James Glass. 2012. Towards unsupervised speech processing. In *2012 11th International Conference on Information Science, Signal Processing and their Applications (ISSPA)*. IEEE, 1–4.
- [61] 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.
- [62] Google. 2021. Cloud TPU | Google Cloud. <https://cloud.google.com/tpu> [Online; accessed 3. Jun. 2021].
- [63] Google. 2021. Coral. <https://coral.ai> [Online; accessed 4. Jun. 2021].
- [64] Google. 2021. Edge TPU performance benchmarks | Coral. <https://coral.ai/docs/edgetpu/benchmarks> [Online; accessed 3. Jun. 2021].
- [65] Google. 2021. TensorFlow models on the Edge TPU | Coral. <https://coral.ai/docs/edgetpu/models-intro/#supported-operations> [Online; accessed 3. Jun. 2021].
- [66] google research. 2021. google-research. <https://github.com/google-research/google-research/tree/master/fastconvnets> [Online; accessed 3. Jun. 2021].
- [67] 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.
- [68] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
- [69] Song Han, Jeff Pool, John Tran, and William J Dally. 2015. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626* (2015).
- [70] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint*

arXiv:1412.5567 (2014).

- [71] Lars Kai Hansen and Peter Salamon. 1990. Neural network ensembles. *IEEE transactions on pattern analysis and machine intelligence* 12, 10 (1990), 993–1001.
- [72] Babak Hassibi, David G Stork, and Gregory J Wolff. 1993. Optimal brain surgeon and general network pruning. In *IEEE international conference on neural networks*. IEEE, 293–299.
- [73] 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.
- [74] 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)*. 784–800.
- [75] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).
- [76] Sara Hooker, Nyalleng Moorosi, Gregory Clark, Samy Bengio, and Emily Denton. 2020. Characterising bias in compressed models. *arXiv preprint arXiv:2010.03058* (2020).
- [77] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1314–1324.
- [78] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146* (2018).
- [79] 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).
- [80] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. 4114–4122.
- [81] Hiroshi Inoue. 2018. Data augmentation by pairing samples for images classification. *arXiv preprint arXiv:1801.02929* (2018).
- [82] 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.
- [83] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. 2017. Population based training of neural networks. *arXiv preprint arXiv:1711.09846* (2017).
- [84] Kevin Jamieson and Ameet Talwalkar. 2016. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*. PMLR, 240–248.
- [85] Jeremy Jordan. 2020. Setting the learning rate of your neural network. *Jeremy Jordan* (Aug 2020). <https://www.jeremyjordan.me/mn-learning-rate>
- [86] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, 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.
- [87] 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)*. 5012–5021.
- [88] 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).
- [89] Pankaj Kanwar, Peter Brandt, and Zongwei Zhou. 2021. TensorFlow 2 MLPerf submissions demonstrate best-in-class performance on Google Cloud. <https://blog.tensorflow.org/2020/07/tensorflow-2-mlperf-submissions.html> [Online; accessed 3. Jun. 2021].
- [90] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv* (Jun 2018). *arXiv:1806.08342* <https://arxiv.org/abs/1806.08342v1>
- [91] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
- [92] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.
- [93] Anders Krogh and Jesper Vedelsby. 1994. Neural network ensembles, cross validation and active learning. In *NIPS'94: Proceedings of the 7th International Conference on Neural Information Processing Systems*. MIT Press, Cambridge, MA, USA, 231–238. <https://doi.org/10.5555/2998687.2998716>
- [94] HT Kung and CE Leiserson. 1980. Introduction to VLSI systems. *Mead, C. A., and Conway, L.,(Eds), Addison-Wesley, Reading, MA* (1980), 271–292.
- [95] Hsiang-Tsung Kung. 1982. Why systolic architectures? *IEEE computer* 15, 1 (1982), 37–46.

- [96] Yann LeCun. 2018. Yann LeCun @EPFL - "Self-supervised learning: could machines learn like humans?". <https://www.youtube.com/watch?v=7l0Qt7GALV&t=316s> [Online; accessed 3. Jun. 2021].
- [97] Yann Lecun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov 1998), 2278–2324. <https://doi.org/10.1109/5.726791>
- [98] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*. 598–605.
- [99] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).
- [100] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning Filters for Efficient ConvNets. In *ICLR (Poster)*.
- [101] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research* 18, 1 (2017), 6765–6816.
- [102] Sharon Y. Li. 2020. Automating Data Augmentation: Practice, Theory and New Direction. *SAIL Blog* (Apr 2020). <http://ai.stanford.edu/blog/data-augmentation>
- [103] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118* (2018).
- [104] 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)*. 19–34.
- [105] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).
- [106] 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.
- [107] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the Value of Network Pruning. *CoRR* abs/1810.05270 (2018). arXiv:1810.05270 <http://arxiv.org/abs/1810.05270>
- [108] Arm Ltd. 2021. SIMD ISAs | Neon – Arm Developer. <https://developer.arm.com/architectures/instruction-sets/simd-isas/neon> [Online; accessed 3. Jun. 2021].
- [109] Gaurav Menghani and Sujith Ravi. 2019. Learning from a Teacher using Unlabeled Data. *arXiv preprint arXiv:1911.05275* (2019).
- [110] 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).
- [111] Jonas Močkus. 1975. On Bayesian methods for seeking the extremum. In *Optimization techniques IFIP technical conference*. Springer, 400–404.
- [112] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning. *CoRR* abs/1611.06440 (2016). arXiv:1611.06440 <http://arxiv.org/abs/1611.06440>
- [113] Node.js Authors. 2021. Node.js. <https://nodejs.org/en> [Online; accessed 3. Jun. 2021].
- [114] NVIDIA. 2020. GTC 2020: Accelerating Sparsity in the NVIDIA Ampere Architecture. <https://developer.nvidia.com/gtc/2020/video/s22085-vid> [Online; accessed 3. Jun. 2021].
- [115] NVIDIA. 2020. Inside Volta: The World’s Most Advanced Data Center GPU | NVIDIA Developer Blog. <https://developer.nvidia.com/blog/inside-volta> [Online; accessed 3. Jun. 2021].
- [116] NVIDIA. 2021. NVIDIA Embedded Systems for Next-Gen Autonomous Machines. <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems> [Online; accessed 4. Jun. 2021].
- [117] Rina Panigrahy. 2021. Matrix Compression Operator. <https://blog.tensorflow.org/2020/02/matrix-compression-operator-tensorflow.html> [Online; accessed 5. Jun. 2021].
- [118] PapersWithCode.com. 2021. Papers with Code - The latest in Machine Learning. <https://paperswithcode.com> [Online; accessed 3. Jun. 2021].
- [119] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).
- [120] Mandela Patrick, Yuki M Asano, Polina Kuznetsova, Ruth Fong, João F Henriques, Geoffrey Zweig, and Andrea Vedaldi. 2020. Multi-modal self-supervision from generalized data transformations. *arXiv preprint arXiv:2003.04298* (2020).
- [121] 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)*. 1532–1543.

- [122] Valerio Perrone, Huibin Shen, Aida Zolic, Iaroslav Shcherbatyi, Amr Ahmed, Tanya Bansal, Michele Donini, Fela Winkelmolen, Rodolphe Jenatton, Jean Baptiste Faddoul, et al. 2020. Amazon SageMaker Automatic Model Tuning: Scalable Black-box Optimization. *arXiv preprint arXiv:2012.08489* (2020).
- [123] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*. PMLR, 4095–4104.
- [124] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668* (2018).
- [125] 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.
- [126] Brian Rakowski. 2019. Pixel 4 is here to help. *Google* (Oct 2019). <https://blog.google/products/pixel/pixel-4>
- [127] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*. Springer, 525–542.
- [128] Sujith Ravi. 2017. Projectionnet: Learning efficient on-device deep networks using neural projections. *arXiv preprint arXiv:1708.00630* (2017).
- [129] 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.
- [130] 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.
- [131] Microsoft Research. 2019. Neural Network Intelligence - Microsoft Research. <https://www.microsoft.com/en-us/research/project/neural-network-intelligence> [Online; accessed 3. Jun. 2021].
- [132] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenshtein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).
- [133] 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.
- [134] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).
- [135] Chinnadhurai Sankar, Sujith Ravi, and Zornitsa Kozareva. 2019. Transferable neural projection representations. *arXiv preprint arXiv:1906.01605* (2019).
- [136] Chinnadhurai Sankar, Sujith Ravi, and Zornitsa Kozareva. 2020. ProFormer: Towards On-Device LSH Projection Based Transformers. *arXiv preprint arXiv:2004.05801* (2020).
- [137] Kaz Sato. 2021. What makes TPUs fine-tuned for deep learning? | Google Cloud Blog. <https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning> [Online; accessed 3. Jun. 2021].
- [138] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (2020), 604–609.
- [139] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Edinburgh neural machine translation systems for wmt 16. *arXiv preprint arXiv:1606.02891* (2016).
- [140] Patrice Y Simard, David Steinhaus, John C Platt, et al. 2003. Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, Vol. 3. Citeseer.
- [141] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
- [142] Dusan Stosic. 2020. Training Neural Networks with Tensor Cores - Dusan Stosic, NVIDIA. https://www.youtube.com/watch?v=jF4_ZK_tyC [Online; accessed 3. Jun. 2021].
- [143] 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.
- [144] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. Mobilebert: a compact task-agnostic bert for resource-limited devices. *arXiv preprint arXiv:2004.02984* (2020).
- [145] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215* (2014).
- [146] 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.
- [147] 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.

- [148] TensorFlow. 2019. TensorFlow Model Optimization Toolkit – Post-Training Integer Quantization. *Medium* (Nov 2019). <https://medium.com/tensorflow/tensorflow-model-optimization-toolkit-post-training-integer-quantization-b4964a1ea9ba>
- [149] TensorFlow. 2021. Model optimization | TensorFlow Lite. https://www.tensorflow.org/lite/performance/model_optimization [Online; accessed 3. Jun. 2021].
- [150] Sik-Ho Tsang. 2019. Review: Xception – With Depthwise Separable Convolution, Better Than Inception-v3 (Image Classification). *Medium* (Mar 2019). <https://towardsdatascience.com/review-xception-with-depthwise-separable-convolution-better-than-inception-v3-image-dc967dd42568>
- [151] Gregor Urban, Krzysztof J Geras, Samira Ebrahimi Kahou, Ozlem Aslan, Shengjie Wang, Rich Caruana, Abdelrahman Mohamed, Matthai Philipose, and Matt Richardson. 2016. Do deep convolutional nets really need to be deep and convolutional? *arXiv preprint arXiv:1603.05691* (2016).
- [152] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. (2011).
- [153] 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).
- [154] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).
- [155] Peisong Wang, Qiang Chen, Xiangyu He, and Jian Cheng. 2020. Towards Accurate Post-training Network Quantization via Bit-Split and Stitching. In *International Conference on Machine Learning*. PMLR, 9847–9856. <http://proceedings.mlr.press/v119/wang20c.html>
- [156] Shibo Wang and Pankaj Kanwar. 2021. BFLOAT16: The secret to high performance on Cloud TPUs | Google Cloud Blog. <https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus> [Online; accessed 3. Jun. 2021].
- [157] Xinyi Wang, Hieu Pham, Zihang Dai, and Graham Neubig. 2018. Switchout: an efficient data augmentation algorithm for neural machine translation. *arXiv preprint arXiv:1808.07512* (2018).
- [158] Pete Warden and Daniel Situnayake. 2019. *TinyML: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. " O'Reilly Media, Inc.".
- [159] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. Fbnets: 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.
- [160] 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.
- [161] 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).
- [162] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V Le. 2018. Qanet: Combining local convolution with global self-attention for reading comprehension. *arXiv preprint arXiv:1804.09541* (2018).
- [163] Tong Yu and Hong Zhu. 2020. Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689* (2020).
- [164] Sergey Zagoruyko and Nikos Komodakis. 2016. Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer. *arXiv preprint arXiv:1612.03928* (2016).
- [165] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. 2017. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412* (2017).
- [166] Michael Zhu and Suyog Gupta. 2018. To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net. <https://openreview.net/forum?id=SyiiIDkPM>
- [167] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).
- [168] 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.