Chapter 1 - Introduction to Efficient Deep Learning

Welcome to the book! This chapter is a preview of what to expect in the book. We start off by providing an overview of the state of deep learning, its applications, and rapid growth. We will establish our motivation behind seeking efficiency in deep learning models. We will also introduce core areas of efficiency techniques (compression techniques, learning techniques, automation, efficient models & layers, infrastructure).

Our hope is that even if you just read this chapter, you would be able to appreciate why we need efficiency in deep learning models today, how to think about it in terms of metrics that you care about, and finally the tools at your disposal to achieve what you want. The subsequent chapters will delve deeper into techniques, infrastructure, and other helpful topics where you can get your hands dirty with practical projects.

With that being said, let's start off on our journey to more efficient deep learning models.

Introduction to Deep Learning

Machine learning is being used in countless applications today. It is a natural fit in domains where there might not be a single algorithm that works perfectly, and there is a large amount of unseen data that the algorithm needs to process. Unlike traditional algorithm problems where we expect exact optimal answers, machine learning applications can often tolerate approximate responses, since often there are no exact answers. Machine learning algorithms help build models, which as the name suggests is an approximate mathematical model of what outputs correspond to a given input.

To illustrate, when you visit Netflix's homepage, the recommendations that show up are based on your past interests, what is popular with other users at that time, and so on. If you have seen 'The Office' many times over like me, there are chances you might like 'Seinfeld' too, which might be popular with other users too. If we train a model to predict the probability based on your behavior and currently trending content, the model will assign a high probability to Seinfeld. While there is no way of predicting with absolute certainty the exact content that you would end up clicking on, at that particular moment, with more data and sophisticated algorithms, these models can be trained to be fairly accurate over a longer term.

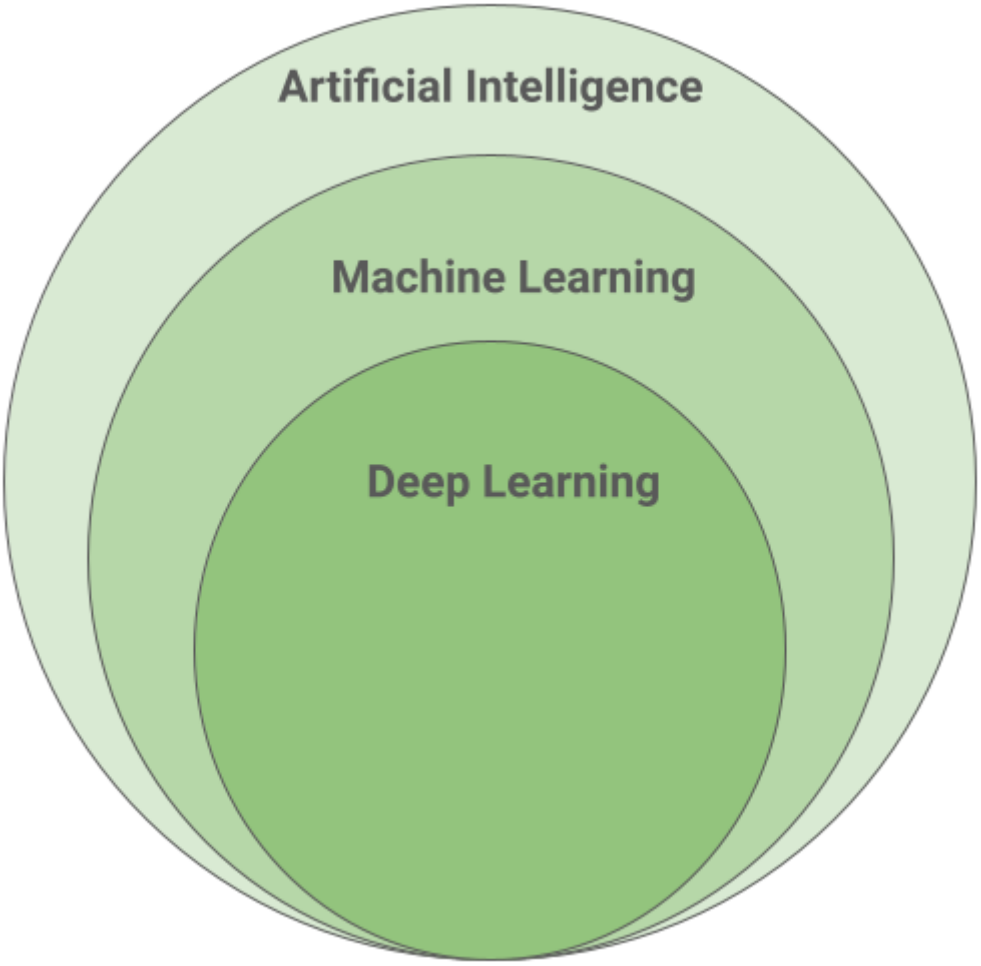


Figure 1-1: Relation between Artificial Intelligence, Machine Learning, and Deep Learning. Deep learning is one possible way of solving machine learning problems. Machine learning in turn is one approach towards artificial intelligence.

Deep learning with neural networks has been the dominant methodology of training new machine learning models for the past decade (Refer to Figure 1-1 for the connection between deep learning and machine learning). Deep Learning models have beaten previous baselines significantly in many tasks in computer vision, natural language understanding, speech, and so on. Their rise can be attributed to a combination of things:

Faster compute through GPUs

Historically, machine learning models were trained on regular CPUs. While CPUs progressively became faster, thanks to Moore's law, they were not optimized for the heavy number-crunching at the heart of deep learning. AlexNet[1] was one of the earliest models to rely on Graphics Processing Units (GPUs) for training, which could

[1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012): 1097-1105.

do linear algebra operations such as multiplying two matrices together much faster than traditional CPUs.

Advances in the training algorithms

There has been substantial progress in machine learning algorithms over the past two decades. Stochastic Gradient Descent (SGD) and Backpropagation were the well-known algorithms designed for training deep networks. However, one of the critical improvements in the past decade was the ReLU activation function.

ReLU[2] allowed the gradients to back-propagate deeper in the networks. Previous iterations of deep networks used the sigmoid or the tanh activation functions, which saturate at either 1.0 or -1.0 except a very small range of input. As a result, changing the input variable leads to a very tiny gradient (if any), and when there are a large number of layers the gradient essentially vanishes.

Availability of labelled data

Even if one has enough compute, and sophisticated algorithms, solving classical machine learning problems relies on the presence of sufficient labeled data. With deep learning models, the performance of the model scaled well with the number of labeled examples, since the network had a large number of parameters. Thus to extract the most out of the setup, the model needed a large number of labeled examples.

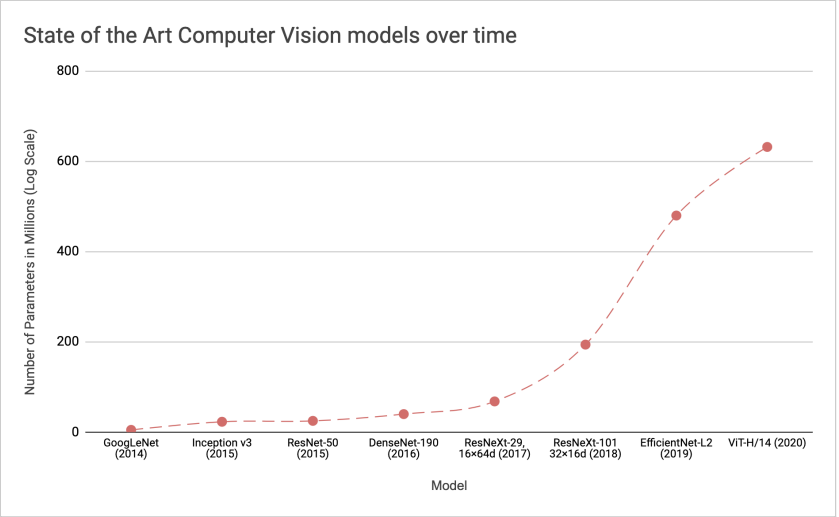
Collecting labeled data is expensive, since it requires training and paying humans to do the arduous job of going through each example and matching it to the given guidelines. The ImageNet dataset was a big boon in this aspect. It has more than 1 million labeled images, where each image belongs to 1 out of 1000 possible classes. This helped with creating a testbed for researchers to experiment with. Along with techniques like Transfer Learning to adapt such models for the real world, and a rapid growth in data collected via websites and apps (Facebook, Instagram, Google, etc.), it became possible to train models that performed well on unseen data (in other words, the models generalized well).

As a result of this trailblazing work, there has been a race to create deeper networks with an ever larger number of parameters and increased complexity. In Computer Vision, several model architectures such as VGGNet, Inception, ResNet etc. (refer to Figure 1-2). have successively beat previous records at the annual ImageNet competitions[3]. These models have been deployed in the real world in various computer vision applications.

---

[2] Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks." *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2011.

[3] The ImageNet competition (officially called the ImageNet Large Scale Visual Recognition Challenge (ILSVRC)) [https://www.image-net.org/challenges/LSVRC/index.php] was an annual competition with the goal of evaluating and discovering state-of-the-art solutions for image classification and object detection, and relied on the ImageNet dataset.



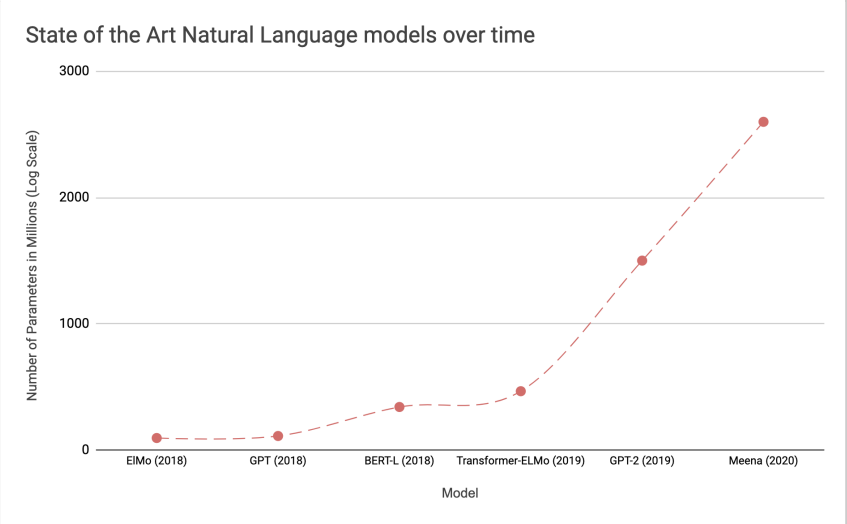


Figure 1-2: Growth of parameters in Computer Vision and NLP models over time. (Data Source)

We have seen a similar effect in the world of Natural Language Processing (NLP) (see Figure 1-2), where the Transformer architecture significantly beat previous benchmarks such as the General Language Understanding Evaluation (GLUE) benchmark. Subsequently models like BERT[4] and GPT[5] models have demonstrated additional improvements on NLP-related tasks. BERT spawned several related model architectures optimizing its various aspects. GPT-3 has captured the attention by being able to generate realistic text accompanying the given prompts. Both these models have been deployed in production. BERT is used in Google Search to improve relevance of results, and GPT-3 is available as an API for interested users to consume. Having demonstrated the rapid growth of deep learning models, let us now move on to how we can make this growth sustainable with efficient deep learning.

[4] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805* (2018).
[5] Brown, Tom B., et al. "Language models are few-shot learners." *arXiv preprint arXiv:2005.14165* (2020).

Efficient Deep Learning

Deep learning research has been focused on improving on the State of the Art, and as a result we have seen progressive improvements on benchmarks like image classification, text classification. Each new breakthrough in neural networks has led to an increase in the network complexity, number of parameters, the amount of training resources required to train the network, prediction latency, etc.

Natural language models such as GPT-3 now cost millions of dollars to train just one iteration. This does not include the cost of trying combinations of different hyper-parameters (tuning), or experimenting with the architecture manually or automatically. These models also often have billions (or trillions) of parameters.

At the same time, the incredible performance of these models also drives the demand for applying them on new tasks which were earlier bottlenecked by the available technology. This creates an interesting problem, where the spread of these models is rate-limited by their efficiency.

While efficiency can be an overloaded term, let us investigate two primary aspects:

Training Efficiency

Training Efficiency involves benchmarking the model training process in terms of computation cost, memory cost, amount of training data, and the training latency. It addresses questions like:

- How long does the model take to train?
- How many devices are needed for the training?
- Can the model fit in memory?
- How much data would the model need to achieve the desired performance on the given task that the model is solving?

For example, when a model is trained to predict if a given tweet contains offensive text, the user should be aware of how many GPUs / TPUs are needed and for how long to converge to a good accuracy. Figure 1-3 shows examples of metrics that measure training efficiency.

Inference Efficiency

By inference, we mean when the model is deployed and is in the prediction mode. Hence, inference efficiency primarily deals with questions that someone deploying a model 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. Using the sensitive tweet classifier example, during the deployment phase the user will be concerned about the inference efficiency and should be aware of what is the inference latency per tweet, peak RAM consumption,

and other requirements that are to be met if the given model is deployed. Refer to Figure 1-3 for examples.

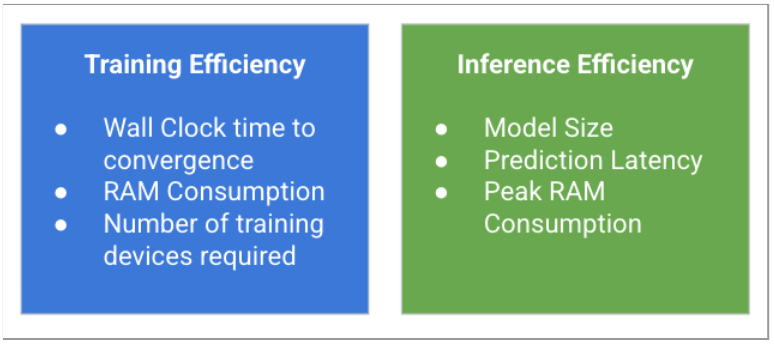


Figure 1-3: Some examples of training and iInference efficiency metrics.

If we have two models performing equally well on a given task, we would choose the one which does better on training or inference efficiency metrics, or both (depending on the use case).

For example, if you are 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. As an illustration, let's say there are two models that achieve comparable classification accuracy. Model A takes 1.5 ms to classify a tweet, while model B takes 10 ms, we would naturally prefer model A since it generates the prediction faster.

Similarly, if you are training a large model from scratch on either with limited or costly training resources, developing models that are designed for Training Efficiency would help. For example, if model A takes 100 GPU hours, while model B takes 5 GPU hours, it might be worth preferring model B if training efficiency is a more important characteristic.

A general guiding principle is illustrated in figure 1-4. Say, we were trying to find models that optimize for both accuracy and latency (in practice, there might be more than two objectives that we might want to optimize for). Naturally, there is a trade-off between the two metrics. It is likely that higher quality models are deeper, hence will have a higher inference latency.

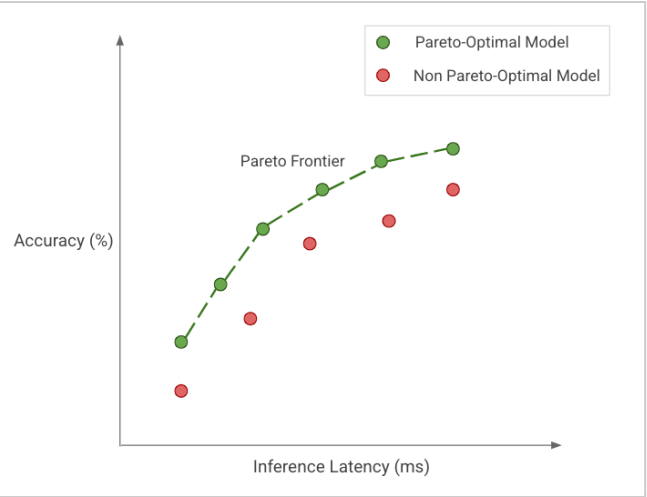


Figure 1-4: Pareto Optimal Models & Pareto Frontier. The green dots are models that have the best tradeoffs for the given objectives, such that there is no other candidate model which gets a better accuracy while keeping the latency the same (and vice versa). They are pareto-optimal models and together make the pareto-frontier.

However, certain models might offer better trade-offs than others. In case we find models where we cannot get a better quality while holding the latency constant, or we cannot get better latency while holding quality constant, we call just models pareto-optimal, and the set of these pareto-optimal models is called the pareto-frontier. All other models are non pareto-optimal. If we care about efficiency, then we would want to deploy models belonging to the pareto-frontier.

Our goal with efficient deep learning is to have a collection of algorithms, techniques, tools, and infrastructure that work together to allow users to train and deploy pareto-optimal models that simply cost less resources to train and/or deploy. This means going from the red dots in Figure 3 to the green dots on the pareto-frontier.

Having such a toolbox to make our models pareto-optimal has the following benefits:

Sustainable Server-Side Scaling

Training and deploying large deep learning models is costly. While training is 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. There is also a very real concern around the carbon footprint of datacenters that are used for training and deploying these large models. Large organizations like Google, Facebook, Amazon, etc. spend several billion dollars each per year in capital expenditure on their data-centers, hence any efficiency gains are very significant.

Enabling On-Device Deployment

With the advent of smartphones, Internet-of-Things (IoT) devices (refer to Figure 1-5 for the trend), and the applications deployed on them have to be realtime, hence there is a need for on-device ML models (where the model inference happens directly on

the device). Which makes it imperative to optimize the models for the device they will run on.

Privacy & Data Sensitivity

Being able to use as little data for training is critical when the user-data might be sensitive to handling / subject to various restrictions such as the General Data Protection Regulation (GDPR) law[6] in Europe. Hence, efficiently training models with a fraction of the data means lesser data-collection required. Similarly, enabling on-device models would imply that the model inference can be run completely on the user's device without the need to send the input data to the server-side.

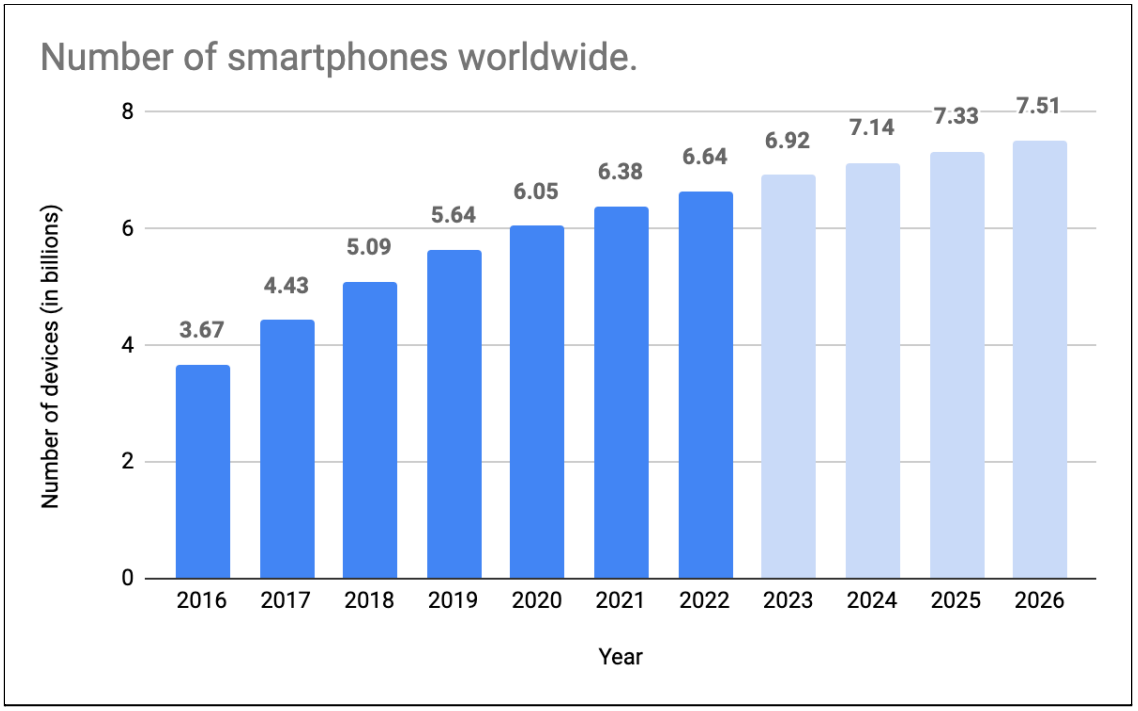
New Applications

Efficiency would also enable applications that couldn't have otherwise been feasible with the existing resource constraints. Similarly, having models directly on-device would also support new offline applications of these models. As an example, the Google Translate application supports offline mode which improves the user experience in low or no-connectivity areas. This is made possible with an efficient on-device translation model.

Explosion of Models

Often there might be multiple ML models being served concurrently on the same device. This further reduces the available resources for a single model. This could happen on the server-side where multiple models are co-located on the same machine, or could be in an app where different models are used for different functionalities.

[6] The GDPR law (https://en.wikipedia.org/wiki/General_Data_Protection_Regulation) introduced regulations for those who collect data of European citizens, such that they are responsible for the safe-keeping of the data and are held legally liable for data breaches. The law went into effect in 2018.



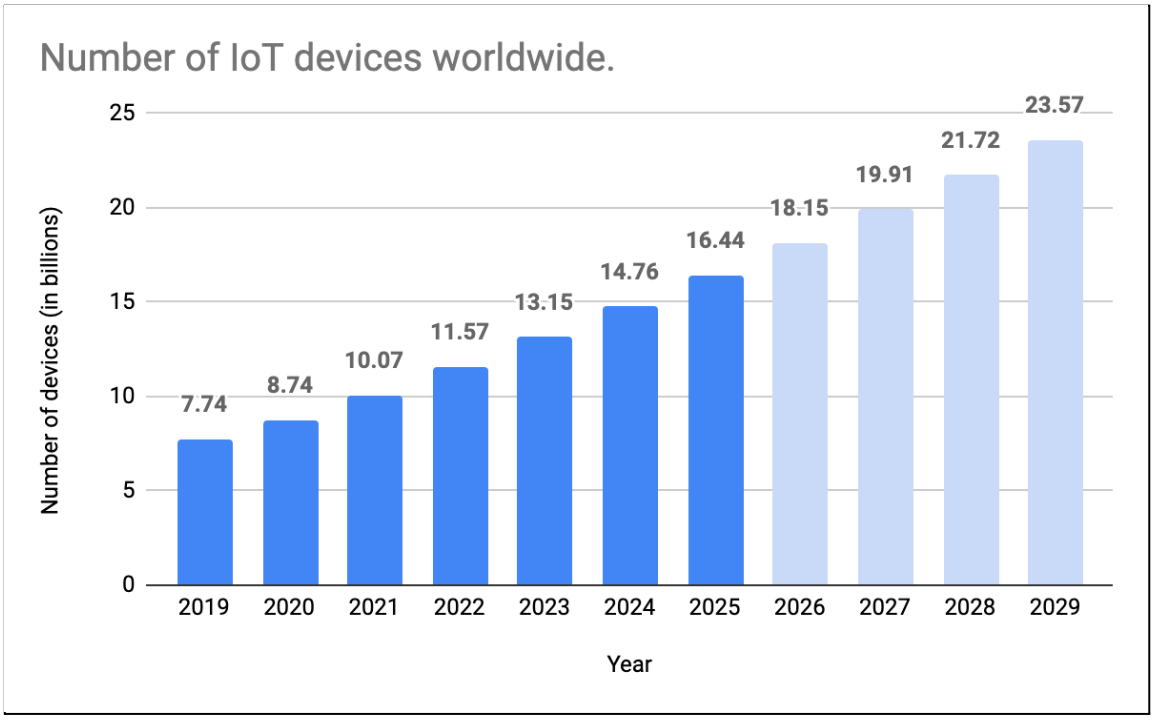


Figure 1-5: Growth in the number of mobile and IoT devices over time. The lighter blue bars represent forecasts. (Data Source: 1, 2)

In this book, we will primarily focus on efficiency for both training and deploying efficient deep learning models from large servers to tiny microcontrollers. Let us start building a mental model of efficient deep learning in the next section.

A Mental Model of Efficient Deep Learning

Before we dive deeper, let's visualize two sets of closely connected metrics that we care about. First, we have quality metrics like accuracy, precision, recall, F1, AUC, etc. Then we have footprint metrics like model size, latency, RAM, etc. Empirically, we have seen that larger deep learning models have better quality, but they are also costly to train and deploy hence worse footprint. On the other hand, smaller and shallower models might have suboptimal quality.

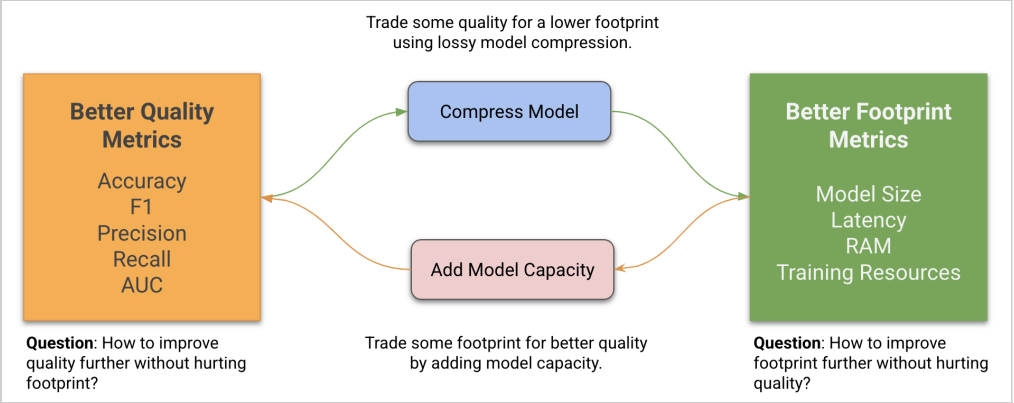


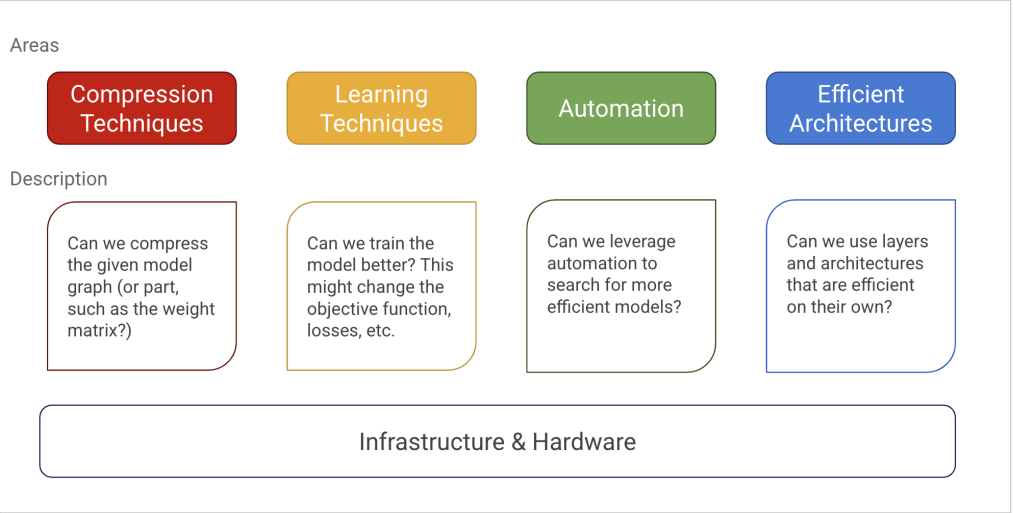
Figure 1-6: Trade-offs between quality metrics and footprint metrics.

In case we have a model where we have some leeway in model quality, we can trade off some of it for a smaller footprint by using lossy model compression techniques[7]. For example, when compressing a model naively we might reduce the model size, RAM, latency etc., but we should expect to lose some model quality too.

Similarly, we can trade off some legroom in footprint if our model is already small, by adding more layers / making the existing layers wider. This might improve model quality but it will increase model size, latency, etc. The question becomes: how can we improve one without hurting the other? This is illustrated in Figure 1-6.

As mentioned earlier, with this book we'll strive to build a set of tools and techniques that can help us make models pareto-optimal and let the user pick the right tradeoff. To that end, we can think of work on efficient deep learning to be categorized in roughly four core areas, with infrastructure and hardware forming the foundation (see Figure 1-7).

[7] Lossy compression techniques allow you to compress data very well, but you lose some information too when you try to recover the data. An example could be reading the summary of a book. You can get an idea of the book's main points, but you will lose the finer details. We cover these in more detail in Chapter 2.



(Figure 1-7: A mental model of Efficient Deep Learning, which comprises the core areas and relevant techniques as well as the foundation of infrastructure, hardware and tools.)

Let us go over each of these areas individually.

Compression Techniques

These are general techniques and algorithms that look at optimizing the architecture itself, typically by compressing its layers, while trading off some quality in return. Often, these approaches are generic enough to be used across architectures.

A classical example is Quantization (see Figure 1-8), which tries to compress the weight matrix of a layer, by reducing its precision (eg., from 32-bit floating point values to 8-bit unsigned / signed integers). Quantization can generally be applied to any network which has a weight matrix. It can often help reduce the model size 2 - 8x, while also speeding up the inference latency.

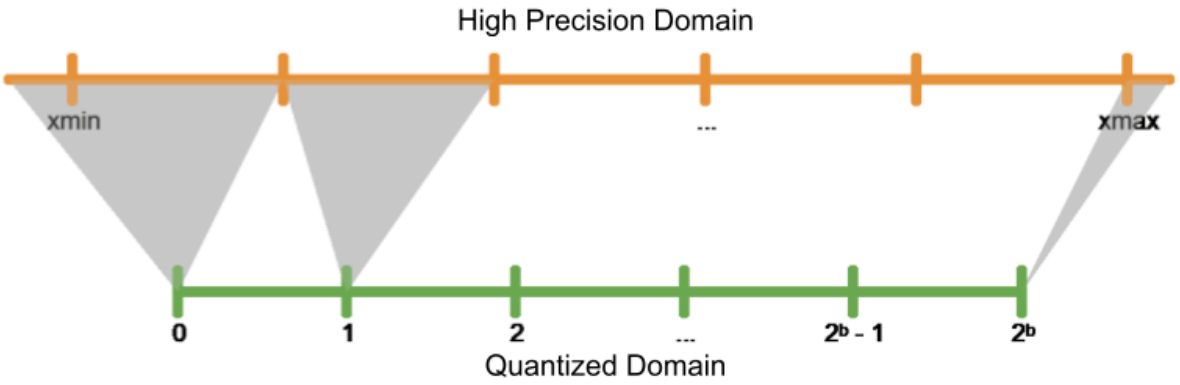


Figure 1-8: An illustration of the quantization process: mapping of continuous high-precision values to discrete fixed-point integer values.

Another example is Pruning (see Figure 1-9), where weights that are not important for the network's quality are removed / pruned. The core idea is to remove redundant weights while the network is being trained. Once the training concludes, the network has fewer connections which helps in reducing the network size and improving the latency. There are many criteria for removing a certain weight, including removing weights with magnitude below a certain threshold t, removing weights which have very little impact on the validation loss of the network, etc.

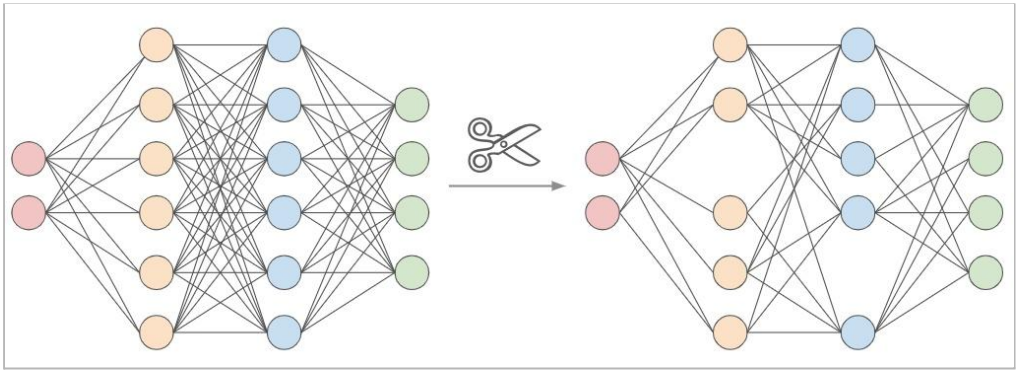


Figure 1-9: Illustration of the pruning process. On the left is the unpruned graph, and on the right is a pruned graph with the unimportant connections and neurons removed.

Learning Techniques

Learning techniques are training phase techniques to improve quality metrics such as accuracy, precision, recall etc. without impacting the model footprint. Improved accuracy 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 (see Figure 1-10), which helps a smaller model (student) that can be deployed, to learn from a larger more accurate model (teacher) which might not be suitable for deployment. The larger model is used to generate soft labels on the training data, and the student model learns to copy both the ground-truth labels as well as the soft labels generated by the teacher model. While the ground-truth labels simply assign a '1' to the correct class, and a '0' to the incorrect class, the soft labels consist of probabilities for each of the possible classes according to the teacher model.

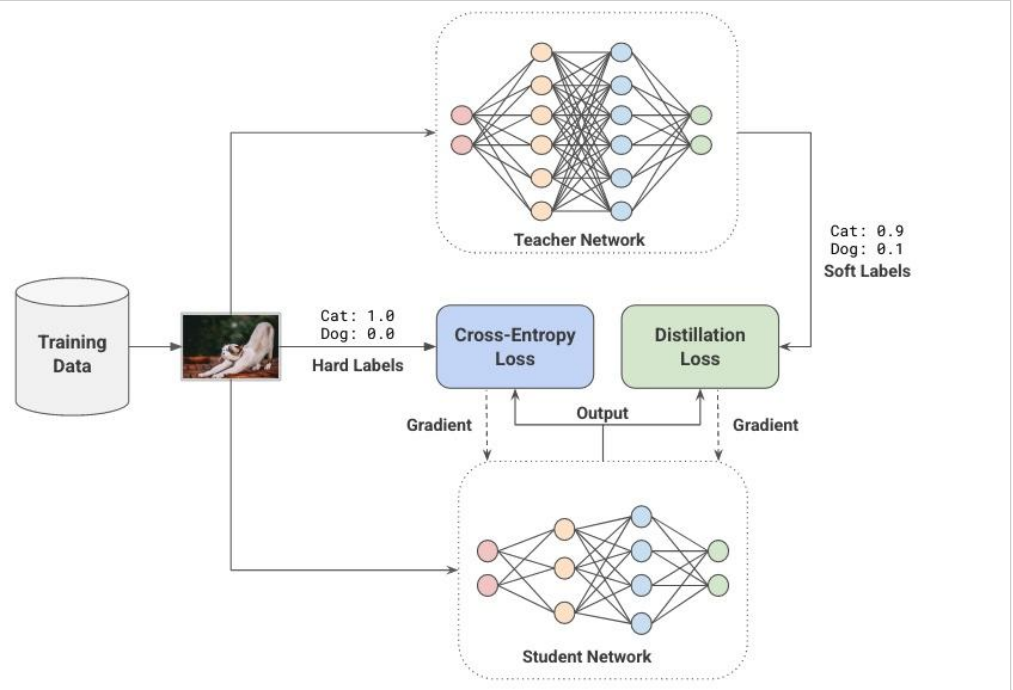


Figure 1-10: Distillation of a smaller student model from a larger pre-trained teacher model. Both the teacher's weights are frozen and the student learns to copy both the ground-truth and the teacher's outputs on the given training data.

The intuition is that the soft labels from the teacher can help the student capture how wrong/right the prediction is. For example, given an image of a truck if the student model predicts that it is a car, the mistake would be penalized less, as compared to predicting that the image is an apple, when using soft labels. Hard labels would penalize both mistakes the same way.

In the original paper which proposed distillation, Hinton et al. replicated performance of an ensemble of 10 models with one model when using distillation. For vision datasets like CIFAR-10, an accuracy improvement in the range of 1-5% has been reported based on the model size. Similarly, the large BERT model was distilled down to a 40% smaller model (DistillBERT), while retaining 97% of the performance.

Another learning technique is Data Augmentation. It is a nifty way of addressing the scarcity of labeled data during training. It is a collection of transformations that can be applied on the given input such that it is trivial to compute the label for the transformed input. For example, if we are classifying the sentiment of a given long piece of text, introducing a single typo is not going to change the sentiment. Similarly, given an image of a dog, rotating the image by 30 degrees, or adding some blur, etc. should still be classified by the model as a dog.

This forces the classifier to learn a representation of the input that generalizes better across the transformations, as it should. Figure 1-11 gives some examples of these transformations. On the CIFAR-10 dataset, typically a 1-4% accuracy improvement is seen with data augmentation techniques.

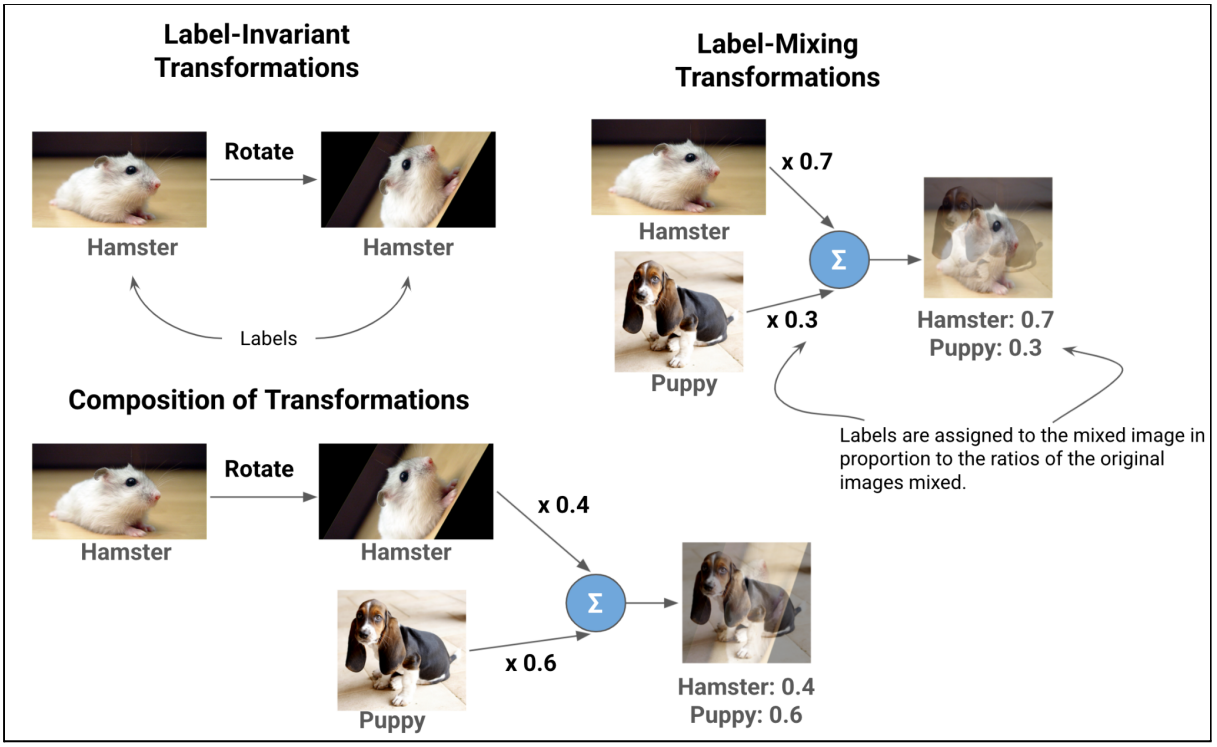


Figure 1-11: Some common types of data augmentations for images.

We will cover learning techniques in more detail in Chapter 3.

Automation

Automation techniques automate some of the tedious tuning that ML practitioners have to do. Apart from saving humans time, it also helps by reducing the bias that manual decisions might introduce when designing efficient networks. Automation techniques can help improve footprint and/or quality. However the trade-off is that it requires large computational resources, so they have to be carefully used.

Automated Hyper-Param Optimization (HPO) is one such technique that can be used to replace / supplement manual tweaking of hyper-parameters like learning rate, regularization, dropout, etc. This relies on search methods that can range from Random Search to methods that smartly allocate resources to promising ranges of hyper-parameters like Bayesian Optimization (Figure 1-12 illustrates Bayesian Optimization). These algorithms construct 'trials' of hyper-parameters, where each trial is a set of values for the respective hyper-parameters. What varies across them is how future trials are constructed based on past results.

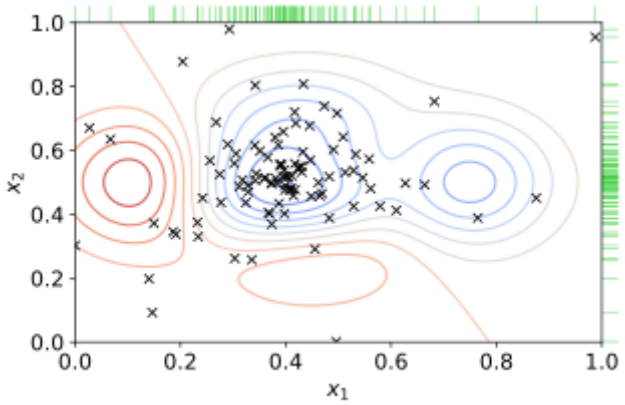


Figure 1-12: Bayesian Optimization over two dimensions x_1 and x_2. Red contour lines denote a high loss value, and blue contour lines denote a low loss value. The contours are unknown to the algorithm. Each cross is a trial (pair of x_1 and x_2 values) that the algorithm evaluated. Bayesian Optimization picks future trials in regions that were more favorable. Source.

As an extension to HPO, Neural Architecture Search (NAS) can help go beyond just learning hyper-parameters, and instead search for efficient architectures (layers, blocks, end-to-end models) automatically. A simplistic architecture search could involve just learning the number of fully connected layers, or the number of filters in a convolutional layer, etc. This could be done with any of the algorithms being used for HPO. A more sophisticated problem would be to learn larger blocks and full networks, where one standard way to do so is described in Figure 1-13.

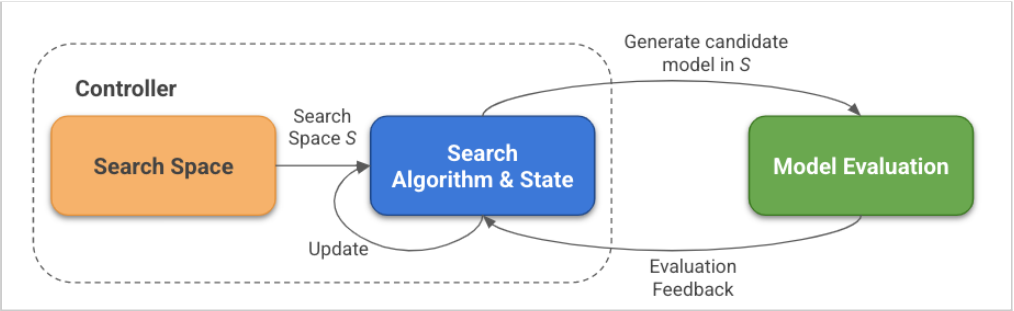


Figure 1-13: The controller can be thought of as a unit that generates candidate models. These candidate models are evaluated and is used to update the state, and generate better candidate models

A controller unit which is provided the possible search space (allowed layers, blocks, configurations), generates valid networks based on this information and past performances of the network using a search algorithm. These candidate networks are then evaluated based on the criteria that needs to be optimized (accuracy, precision, recall, etc.), and the feedback is passed back to the controller to make better suggestions in the future.

NAS has been used to generate State of the Art networks for common datasets like CIFAR-10, ImageNet, WMT etc. An example network for machine translation is shown in

Figure 1-14, where using Neural Architecture Search, the authors improve over the Transformer Encoder architecture that is the leading architecture being used for complex NLP tasks such as translation. The NAS generated architecture, which is named Evolved Transformer[8], achieves better quality at the same footprint, and smaller footprint at the same quality. NAS has also been used to explicitly optimize these multiple objectives directly, like finding networks that get the best quality, while incurring the least latency during inference.

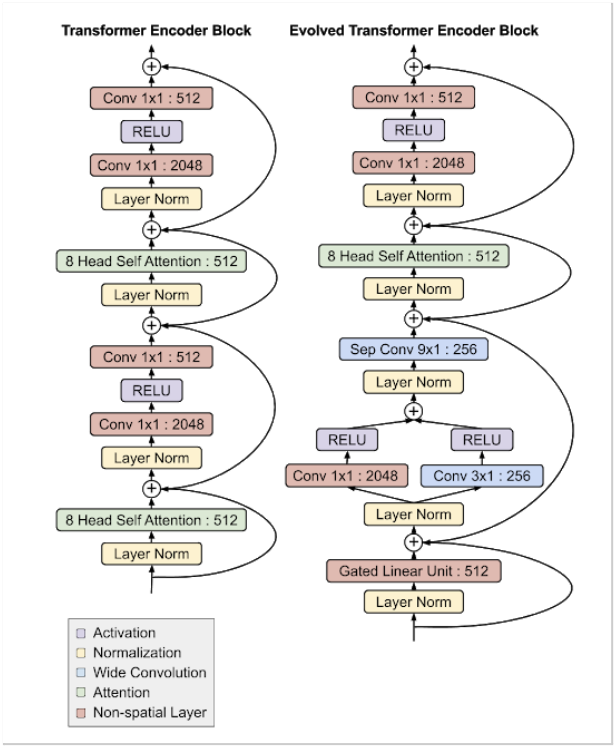


Figure 1-14: Standard Transformer Encoder block (left), and an Evolved Transformer Encoder block (right). While the former was designed manually, the latter was learnt through Neural Architecture Search (NAS). It achieves better quality at the same footprint, and better footprint at the same quality. Source.

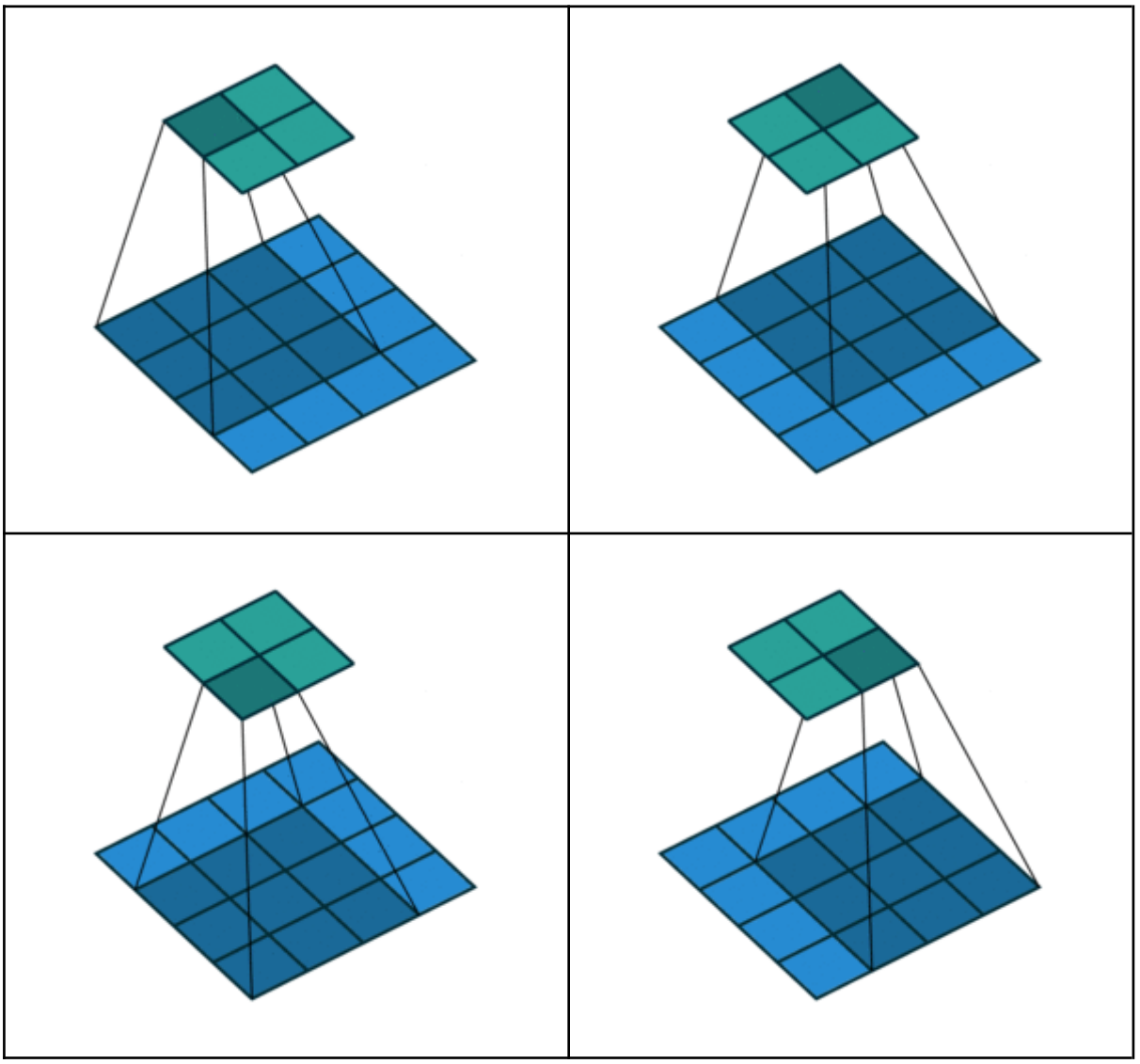
We will explore automation techniques further in Chapter 5.

Efficient Model Architectures & Layers

Now we get to efficient model architectures and layers, which forms the crux of efficient deep learning. These are fundamental blocks that were designed from scratch (Convolutional Layers (see Figure 1-15), Attention, etc.), that are a significant leap over the baseline methods used before them. As an example, convolutional layers introduce parameter sharing and filters for use in image models, which avoids having to learn separate

[8] So, David, Quoc Le, and Chen Liang. "The evolved transformer." *International Conference on Machine Learning*. PMLR, 2019.

weights for each input pixel. This clearly saves the number of parameters when you compare it to a standard multi-layer perceptron (MLP) network. Avoiding over-parameterization further helps in making the networks more robust. Within these sets of techniques, we would look at layers and architectures that have been designed specifically with efficiency in mind.



(Figure 1-15: An illustration of the convolutional process, with the bigger 5x5 representing the input, and the smaller 2x2 square representing the filter. Source)

Similarly for natural language models, one of the costlier parts of the models are Embedding Tables, where each input token that is likely to be seen during inference, is assigned a learned *embedding vector* that encodes a semantic representation of that token in a

fixed-dimensional floating point vector. These embedding tables are very useful, because they help us convert abstract concepts hidden in natural language into a mathematical representation that our models can use. The quality of these models scales with the number of tokens we learn an embedding for (the size of our vocabulary), and the size of the embedding (known as the dimensionality). However, this also leads to a direct increase in model size and memory consumption.

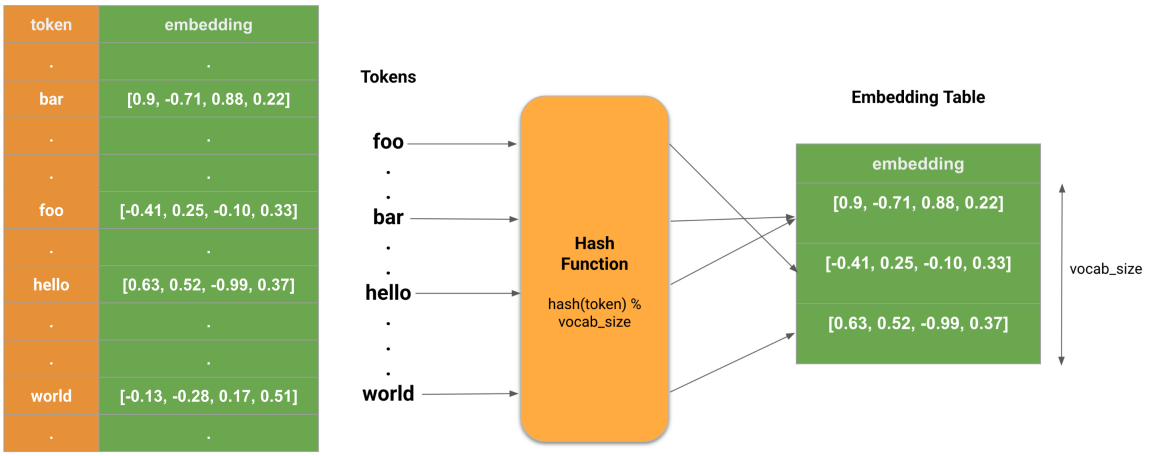


Figure 1-16: A regular embedding table on the left with an embedding for each token. Hashing Trick on the right, where multiple tokens map to the same slot and share embeddings, and thus helps with saving space.

To remedy this problem, there are approaches like the Hashing Trick (refer to Figure 1-16), which offers a trade-off that could be used in case the model size and memory is a bottleneck for deploying the model. With the Hashing Trick, instead of learning one embedding vector for each token, many tokens can share a single embedding vector. The sharing can be done by computing the hash of the token modulo the number of possible vectors that are desired (vocabulary size). This trade-off allows exchanging some quality for model size. We will explore this approach in detail in Chapter 4.

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, PyTorch, etc.. Often these frameworks will be paired with the tools required specifically for deploying efficient models. For example, tensorflow has a tight integration with Tensorflow Lite (TFLite) and related libraries, which allow exporting and running models on mobile devices. Similarly, TFLite Micro helps in running these models on DSPs. PyTorch offers PyTorch Mobile for quantizing and exporting models for inference on mobile and embedded devices.

As another example, to get size and latency improvements with quantized models, we need the inference platform to support common neural net layers in quantized mode. TFLite supports quantized models, by allowing export of models with 8-bit unsigned int weights, and having integration with libraries like GEMMLOWP and XNNPACK for fast inference.

Similarly, PyTorch uses QNNPACK to support quantized operations. Refer to Figure 1-17 for an illustration of how infrastructure fits in training and inference.

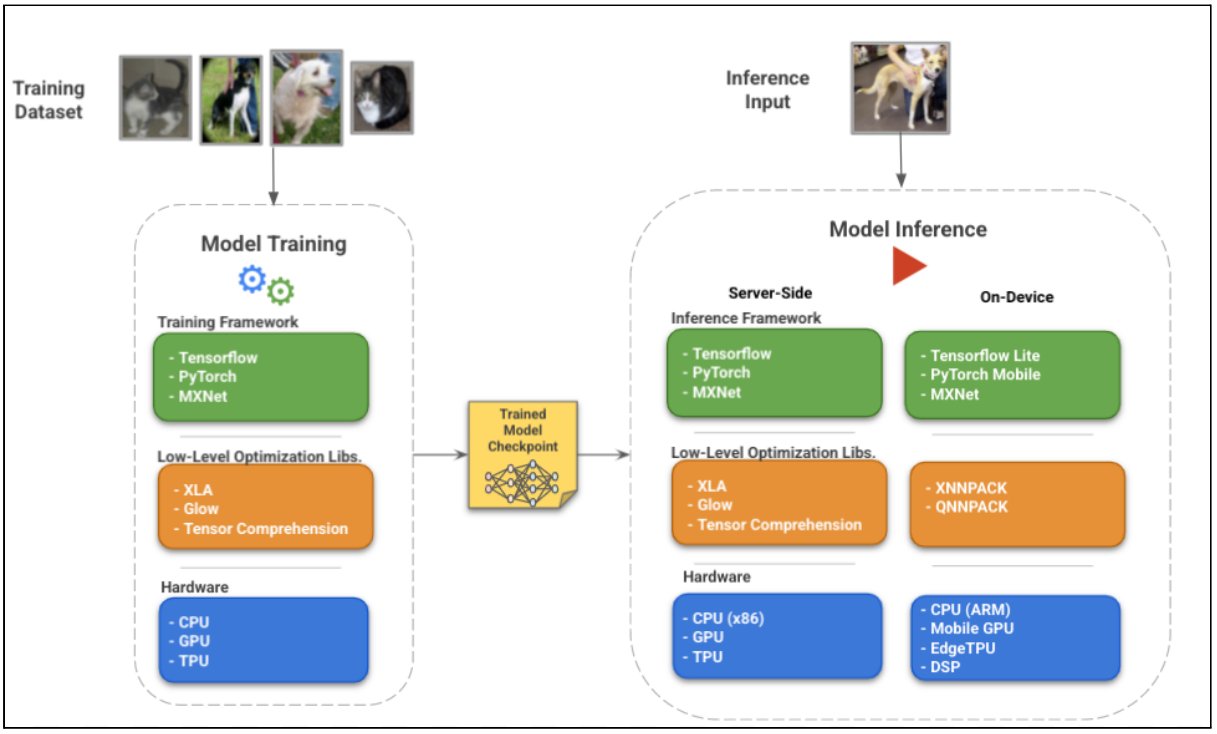


Figure 1-17: Model Training & Inference stages, along with the constituent infrastructure components.

Advances in hardware are significantly responsible for the deep learning revolution, specifically the GPU (Graphics Processing Unit), since they made it possible to train deep models many times faster than CPU. Since GPUs were not originally designed with such applications in mind but rather are general purpose hardware that cater to a large number of applications, new hardware that optimizes for model training and inference have been springing up. For example, Google's Tensor Processing Units (TPUs) optimize for accelerating large matrix multiplications (specifically they allow massively parallelizing the Multiply-Add-Accumulate operation while minimizing memory access). TPUs have been used for speeding up training as well as inference, apart from being used in production they have also been used in the famous AlphaGo and AlphaZero projects, where DL models beat the best humans as well as other computer bots in games like chess, shogi, and go.

For the purpose of deployment in IoT and edge devices, both Google and NVidia have come up with accelerators that can be used for fast inference on-devices. The EdgeTPU (see Figure 1-18 for reference) , like the TPU, specialized in accelerating linear algebra operations, but only for inference and with a much lower compute budget. It uses about 2 watts of power, and operates in quantized mode with a restricted set of operations. It is available in various form-factors ranging from a Raspberry-Pi like Dev Board to an independent solderable module. It has also been shipped directly on phones, such as Pixel 4.

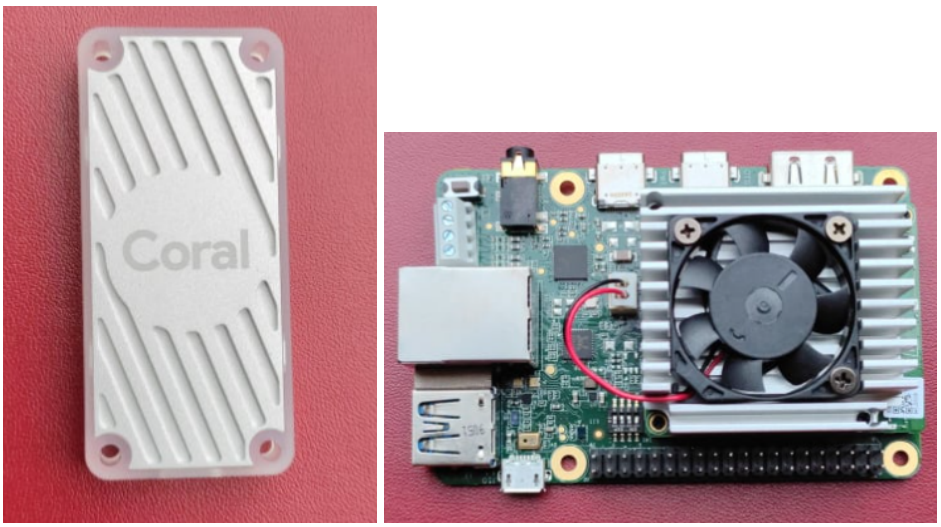


Figure 1-18: Approximate size of the EdgeTPU, Coral, and the Dev Board
(Courtesy: Bhuwan Chopra)

Jetson (see Figure 1-19) is Nvidia's equivalent family of accelerators for edge devices. It comprises 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.

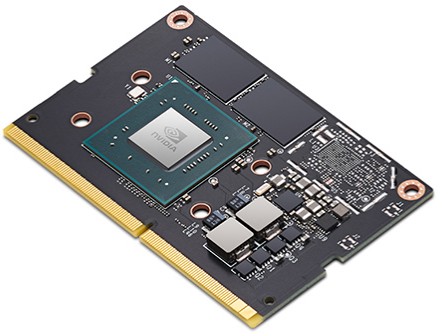


Figure 1-19: Jetson Nano module (Source)

Hardware platforms like these are crucial because they enable our efficient models and algorithms. Hence, often we need to keep their strengths and limitations in mind when optimizing models. We will cover deep learning hardware in detail in the later chapters.

Summary

What we want the reader to take away from this chapter is that the industry and academia have already been investing and accelerating their investment in ML efficiency.

Freeing up finite resources is a no-brainer for everyone from a budding startup to a large corporation that invests billions in data-centers, therefore paying attention to training and deployment efficiency is critical to be competitive.

We can measure efficiency in terms of footprint and quality metrics. The former is associated with metrics like size, memory, latency, and so on. While the latter deals with the model's performance such as accuracy, precision, recall etc. It is also possible to exchange some improvement in one kind of metric for the other. How you trade-off one for the other depends on your usecase.

We introduced efficiency techniques to improve the metrics that you care about. Compression techniques for instance, can be used to improve the footprint of your model (size, memory, latency, etc.) while trading off some quality as needed. Learning techniques can be used to improve the model quality without hurting the footprint. Similarly, automation techniques help us leverage automation to search for efficient models and layers.

Apart from the above techniques, we gave a quick introduction to efficient models and layers, which are designed with efficiency in mind and can be used as building blocks in your usecase. Finally, we went over infrastructure, both software and hardware, which is a crucial foundation for us to be able to leverage the efficiency gains in practice.

We hope that this chapter would have given you an idea of why efficiency matters in your models, how to think about it in terms of tangible metrics, and the tools available at your disposal. In the following chapters, we would go over each of the above areas in more detail along with projects, to get your hands dirty with optimizing machine learning models.