Chapter 6. Model Resource Management Techniques

The compute, storage, and I/O systems that your model requires will determine how much it will cost to put your model into production and maintain it during its entire lifetime. In this chapter, we'll take a look at some important techniques that can help us manage model resource requirements. We'll focus on three key areas that are the primary ways to optimize models in both traditional ML and generative AI (GenAI):

- Dimensionality reduction
- Quantizing model parameters and pruning model graphs
- Knowledge distillation to capture knowledge contained in large models

Dimensionality Reduction: Dimensionality Effect on Performance

We'll begin by discussing dimensionality and how it affects our model's performance and resource requirements.

In the not-so-distant past, data generation and, to some extent, data storage were a lot more costly than they are today. Back then, a lot of domain experts would carefully consider which features or variables to measure

before designing their experiments and feature transforms. As a consequence, datasets were expected to be well designed and to potentially contain only a small number of relevant features.

Today data science tends to be more about integrating everything end to end. Generating and storing data is becoming faster, easier, and less expensive to do. So there's a tendency for people to measure everything they can and to include ever more complex feature transformations. As a result, datasets are often high dimensional, containing a large number of features, although the relevancy of each feature for analyzing the data is not always clear.

Before going too deep, let's discuss a common misconception about neural networks. Many developers correctly assume that when they train their neural network models, the model itself, as part of the training process, will learn to ignore features that don't provide predictive information, by reducing their weights to zero or close to zero. While this is true, the result is not an efficient model.

Much of the model can end up being "shut off" when running inference to generate predictions, but those unused parts of the model are still there. They take up space, and they consume compute resources as the model server traverses the computation graph.

Those unwanted features can also introduce unwanted noise into the data, which can often degrade model performance. In fact, high dimensionality can even cause overfitting. And outside of the model itself, each extra feature still requires systems and infrastructure to collect that data, store it, and manage updates, which adds cost and complexity to the overall system. That includes monitoring for problems with the data, and the effort to fix those problems if and when they happen. Those costs continue for the lifetime of the product or service that you're deploying, which could easily be years.

There are techniques for optimizing models with weights that are zero or close to zero. But in general, you shouldn't just throw everything at your model and rely on your training process to determine which features are actually useful.

In ML, high-dimensional data is a common challenge. For instance, tracking 60 different metrics per shopper results in a 60-dimensional space. Analyzing 50 × 50 pixel grayscale images involves 2,500 dimensions, while RGB images have 7,500 dimensions, with each pixel's color channels contributing a dimension.

Some feature representations such as one-hot encoding are problematic for working with text in high-dimensional spaces, as they tend to produce very sparse representations that do not scale well. One way to overcome this problem is to use an embedding layer that tokenizes the sentences and

assigns a float value to each word. This leads to a more powerful vector representation that respects the timing and sequence of the words in a given sentence. This representation can be automatically learned during training.

Example: Word Embedding Using Keras

Let's look at a concrete example of word embedding using Keras. First, we'll train a model with a high-dimensional embedding. Then, we'll reduce the embedding dimension, retrain the model, and compare its accuracy against the high-dimensional version:

```
!pip install -U "jax[cpu]" -f
    https://storage.googleapis.com/jax-releases/l
!pip install --upgrade keras
import numpy as np
import os
os.environ["KERAS_BACKEND"] = "jax"
import keras
from keras.datasets import reuters
from keras.preprocessing import sequence
from keras.utils import to_categorical
from keras import layers
num_words = 1000
print(f'Keras version: {keras.__version__}\n\n')
(reuters_train_x, reuters_train_y), (reuters_test
    reuters.load_data(num_words=num_words)
n_labels = np.unique(reuters_train_y).shape[0]
```

```
reuters_train_y = to_categorical(reuters_train_y,
reuters_test_y = to_categorical(reuters_test_y, 4
```

The Reuters news dataset contains 11,228 newswires labeled over 46 topics. The documents are already encoded in such a way that each word is indexed by an integer (its overall frequency in the dataset). While loading the dataset, we specify the number of words we'll work with (1,000) so that the least-repeated words are considered unknown.

Let's further preprocess the data so that it's ready for training a model. First, the following code converts target vectors `*_y` into categorical variables, for both `train` and `test`. Next, the code segments the input text `*_x` into text sequences that are 20 words long:

```
reuters_train_x = sequence.pad_sequences(reuters_
reuters_test_x = sequence.pad_sequences(reuters_t
```

Building the network is the next logical step. Here, the choice is to embed a 1,000-word vocabulary using all the dimensions in the data. The last layer is dense, with dimension 46, since the target variable is a 46-dimensional vector of categories.

With the model structure ready, let's compile the model by specifying the loss, optimizer, and output metric. For this problem, the natural choices are

categorical cross-entropy loss, rmsprop optimization, and accuracy as the metric:

```
model1 = keras.Sequential(
    [
        layers.Embedding(num_words, 1000),
        layers.Flatten(),
        layers.Dense(256),
        layers.Dropout(0.25),
        layers.Activation('relu'),
        layers.Dense(46),
        layers.Activation('softmax')
    ]
)
model1.compile(loss="categorical_crossentropy", c
               metrics=["accuracy"])
```

We’re ready to actually do a model fitting. We’ll specify the validation set, batch size, and number of epochs for training. We’ll also add a callback for TensorBoard:

```
tensorboard_callback =
          keras.callbacks.TensorBoard(log_dir="
model_1 = model1.fit(reuters_train_x, reuters_tra
                     validation_data=(reuters_test_
```

```
                    batch_size=128, epochs=20, ver
                    callbacks=[tensorboard_callbac
```

Now let's plot our results using TensorBoard. Note that this code is running in a Colab notebook:

```
# Load the TensorBoard notebook extension
%load_ext tensorboard
# Open an embedded TensorBoard viewer
%tensorboard --logdir ./logs_model1
```

Figure 6-1 shows the training accuracy and loss as a function of training epochs. Notice that after about two epochs our training set results in significantly higher accuracies and lower losses compared to the validation set. This is a clear indication that the model is severely overfitting. This may be the result of using all the dimensions of the data, and therefore, the model is picking up nuances in the training set that do not generalize well.

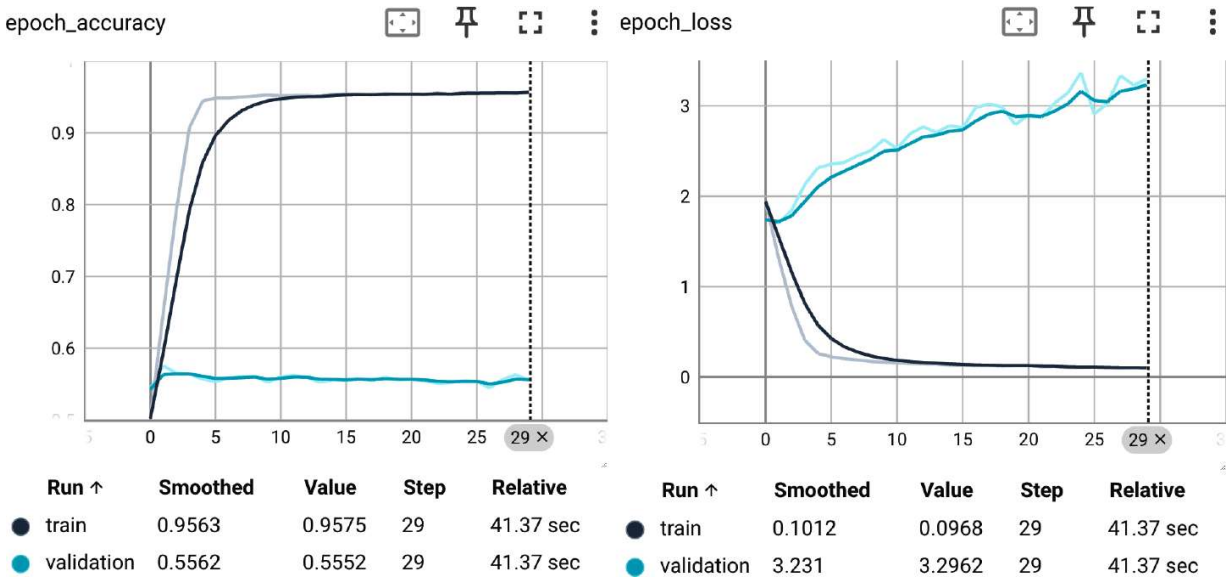

Figure 6-1. Model training and validation metrics

Let's try reducing the dimensionality and see how this affects model performance. Let's take our 1,000-word vocabulary and embed it into 6 dimensions instead of the 1,000 dimensions that we used in Figure 6-1. This is roughly a reduction by a fourth root factor. The model remains unchanged otherwise:

```
model2 = keras.Sequential(
    [
        layers.Embedding(num_words, 10),
        layers.Flatten(),
        layers.Dense(256),
        layers.Dropout(0.25),
        layers.Activation('relu'),
        layers.Dense(46),
        layers.Activation('softmax')
```

```
    ]
)
model2.compile(loss="categorical_crossentropy", c
               metrics=["accuracy"])
tensorboard_callback =
          keras.callbacks.TensorBoard(log_dir="
model_2 = model2.fit(reuters_train_x, reuters_tra
                     validation_data=(reuters_test_
                     batch_size=128, epochs=20, ver
                     callbacks=[tensorboard_callbac
# Open an embedded TensorBoard viewer
%tensorboard --logdir ./logs_model2
```

Figure 6-2 shows that there may still be some overfitting, but with that one change this model performs significantly better than the 1,000-dimension version.

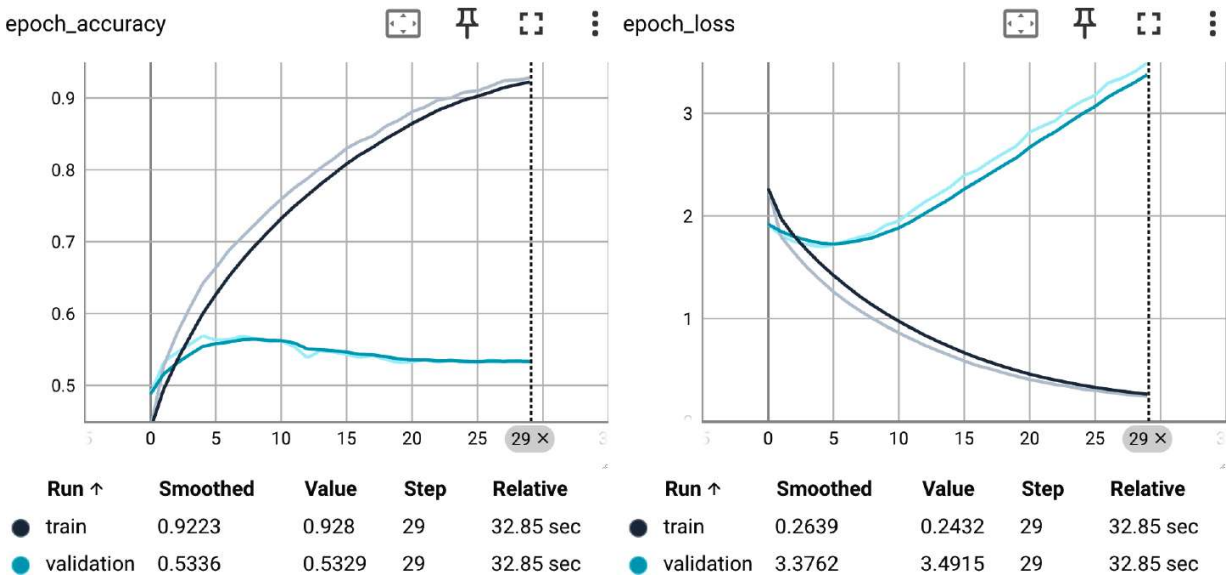

Figure 6-2. Training metrics after reducing embedding dimensions

Curse of Dimensionality

Let’s talk about the curse of dimensionality, and why this is a very important topic when building models.

Many common ML tasks, such as segmentation and clustering, rely on computing distances between observations. For example, supervised classification uses the distance between observations to assign a class. K-nearest neighbors is an example of this. Support vector machines (SVMs) deal with projecting observations using kernels based on the distance between the observations after projection. Another example is recommendation systems that use a distance-based similarity measure between the user and the item attribute vectors. Other forms of distance could also be used. One of the most common distance metrics is Euclidean distance, which is simply a linear distance between two points in a

multidimensional hyperspace. The Euclidean distance between two-dimensional vectors with Cartesian coordinates is calculated using this familiar formula:

$$d_{ij} = \sqrt{\sum_{k=1}^{n} (x_{ik} - x_{jk})^2}$$

But why is distance important? Let’s look at some issues with measuring distance in high-dimensional spaces.

You might be wondering why data being high dimensional can be an issue. In extreme cases where we have more features (dimensions) than observations, we run the risk of massively overfitting our model. But in more general cases when we have too many features, observations become harder to cluster. An abundance of dimensions can lead to a situation where all data points seem equally far apart. This poses a significant challenge for clustering algorithms that depend on distance metrics; it makes all observations appear similar, hindering the creation of meaningful clusters. This phenomenon, known as the *curse of dimensionality*, causes the dissimilarity between data points to diminish as the number of dimensions increases. In essence, the distances between points tend to become more concentrated, resulting in unexpected outcomes when working in high-dimensional spaces.

The *curse of dimensionality* was coined by Richard Bellman in his 1961 book *Adaptive Control Processes: A Guided Tour* (Princeton University Press) and describes the unintuitive behavior of data in high-dimensional

spaces. This primarily affects our ability to understand and use distances and volumes. The curse of dimensionality has two key implications:

- ML excels at high-dimensional analysis. ML algorithms have a distinct advantage over humans in handling high-dimensional data. They can effectively uncover patterns within datasets containing a large number of dimensions, even when those dimensions have intricate relationships.
- Increased dimensionality demands more resources: As the number of dimensions increases, so does the computational power and training data needed to build effective models.

So, although there is sometimes a tendency to add as many features as possible to our data, adding more features can easily create problems. This could include redundant or irrelevant features appearing in data. Moreover, noise is added when features don't provide predictive power for our models. On top of that, more features make it harder for one to interpret and visualize data. Finally, more features mean more data, so you need to have more storage and more processing power to process it. Ultimately, having more dimensions often means our model is less efficient.

When we have problems getting our models to perform, we are often tempted to try adding more and more features. But as we add more features, we reach a certain point where our model's performance degrades, as shown in Figure 6-3.

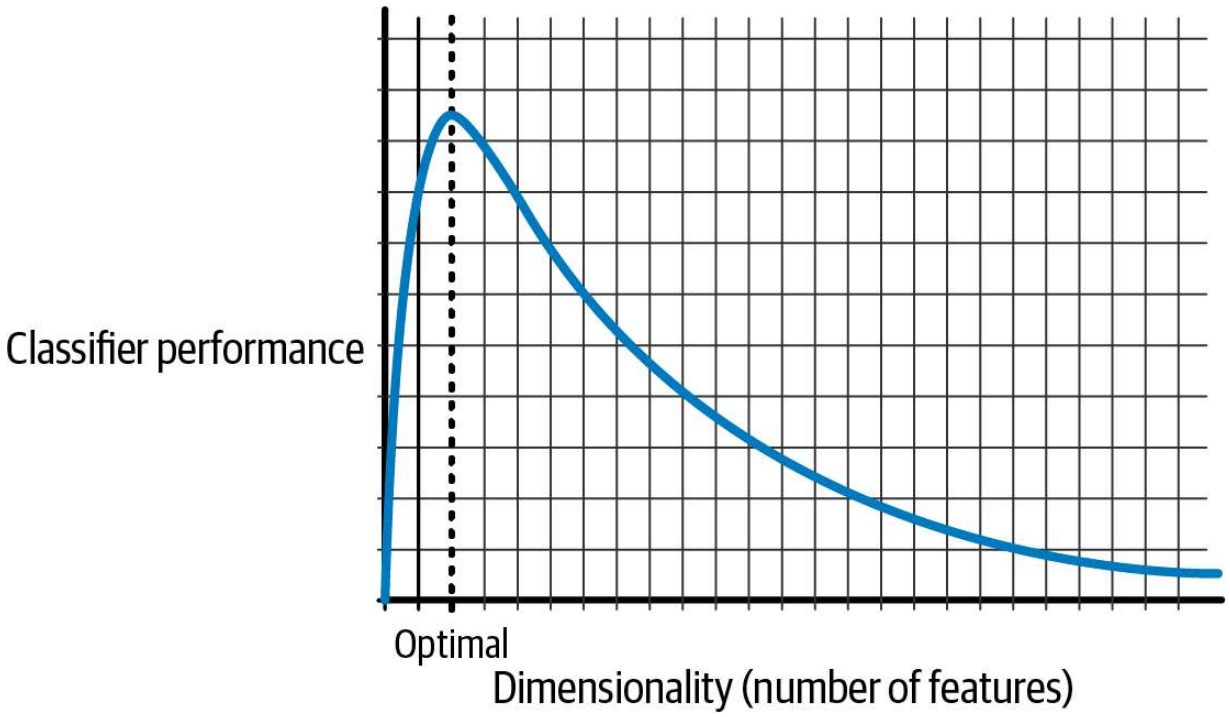

Figure 6-3. The curse of dimensionality

Figure 6-3 demonstrates that the classifier's performance improves as the number of dimensions increases up to a point where the optimal number of features is reached. Beyond that point, with a fixed number of training examples, adding more dimensions leads to a gradual decline in performance.

Let's explore another issue related to dimensionality to better understand the cause of this behavior.

Adding Dimensions Increases Feature Space Volume

An increase in the number of dimensions of a dataset (the number of features) means there are more entries in the feature vector representing each training example. For instance, in terms of a Euclidean space and Euclidean distance measure, each new dimension adds a nonnegative term to the sum. That tends to increase the distance measure as we add more features. As a result, the examples get farther apart.

In other words, as the number of features grows for a given number of training examples, the feature space becomes increasingly sparse, with more distance between training examples. Because of that, the lower data density requires more training examples to keep the average distance between examples the same. It’s also important that the examples added are significantly different from the examples already present in the sample.

As the distance between data points increases, supervised learning becomes more challenging because predictions for new instances are less likely to be informed by similar training examples. The feature space expands rapidly with the addition of more features, making effective generalization increasingly difficult. The model’s variance also increases, raising the risk of overfitting to noise present in higher-dimensional spaces. In practice, features can also often be correlated or do not exhibit much variation. For these reasons, there’s a need to reduce dimensionality.

The goal is to keep as much of the predictive information as possible, using as few features as possible, to make the model as efficient as possible.

Regardless of which modeling approach you're using, increasing dimensionality has another problem, especially for classification. The *Hughes effect* is a phenomenon that demonstrates the improvement in classification performance as the number of features increases until we reach a Goldilocks optimum where we have just the right number of features. As shown in Figure 6-3, adding more features while keeping the training set the same size will degrade the classifier's performance.

In classification, the goal is to find a function that discriminates between two or more classes. You could do this by searching for hyperplanes in space that separate these categories. The more dimensions you have, the easier it is to find a hyperplane during training, but at the same time, the harder it is to match that performance when generalizing to unseen data. And the less training data you have, the less sure you are that you identified the dimensions that matter for discriminating between the categories.

Dimensionality Reduction

Unfortunately, there's no one-size-fits-all answer to the question of how many features are ideal for an ML problem. The optimal number depends on various factors, including the volume of training data, the variability

within that data, the intricacy of the decision boundary, and the specific model being employed.

There is a connection between dimensionality reduction and feature selection, since the number of features you include in the input to your model has a large impact on the overall dimensionality of your model. Essentially, you want enough data, with the best features, enough variety in the values of those features, and enough predictive information in those features, to maximize the performance of your model while simplifying it as much as possible.

Therefore, when preprocessing a set of features to create a new feature set, it’s important to retain as much predictive information as possible. Without predictive information, all the data in the world won’t help your model learn. This information also needs to be in a form that will help your model learn.

Achieving optimal outcomes with ML often depends on the practitioner’s expertise in crafting effective features. This aspect of ML engineering involves a degree of artistry; feature importance and selection tools can only provide objective insights about existing features. You often need to manually create them. This requires spending a lot of time with actual sample data and thinking about the underlying form of the problem, the structures in the data, and how to best express them for predictive modeling algorithms.

Three approaches

There are basically three different ways to select the right features. This is before you get into any feature engineering to try to improve the features you've selected.

The first approach is manual feature selection, which is usually based on domain knowledge and/or previous experience with similar models in a similar domain.

The second approach is to apply feature selection algorithms, of which there are many. Feature selection tries to analyze your data by creating a search space of your features and trying to determine the optimal set of features that meet your criteria, which is often the number of features you want to have or the model you want to train. Dimensionality reduction can also be done, and is often discussed independent of feature selection.

The third approach is algorithmic dimensionality reduction, which tries to project your features from the space they define into a lower-dimensional space. Principal component analysis (PCA) is the most commonly used example of this. By representing your data in a lower-dimensional space, the number of dimensions is decreased. However, this usually also means the intuitive understanding of the different dimensions of your data is lost, and humans have a hard time interpreting what a particular example represents.

These approaches are not mutually exclusive, and in many cases you'll end up using some combination of them. Which ones you decide to use in any particular situation is part of the art form of ML engineering. As a rule of thumb, it's best to start simple, and to progressively add complexity only as you need it and only when doing so continues to improve the results.

Algorithmic dimensionality reduction

There are several algorithms for doing dimensionality reduction. First, let's build some intuition on how linear dimensionality reduction actually works. In this approach, you linearly project *n*-dimensional data onto a smaller *k*-dimensional subspace. Here, *k* is usually much smaller than *n*. There are infinitely many dimensional subspaces that we can project the original data onto. So, which subspace should you choose?

To understand how subspaces are chosen, let's take a step backward and look at how we can project data onto a line. To start, let's think of examples as vectors existing in a high-dimensional space. Visualizing them would reveal a lot about the distribution of the data, though it's impossible for us humans to see only so many dimensions at once.

Instead, we need to project data onto a lower dimension. This kind of projection is called an *embedding*. In the extreme case where we want to have only one dimension, we take each example and calculate a single number to describe it. A benefit of reducing to one dimension is that the

numbers and the examples can be sorted on a line, which is easy to visualize. In practice, though, we will rarely want just one dimension for data we're going to use to train a model.

Coming back to subspaces, there are several ways to choose these *k*-dimensional subspaces. For example, for a classification task we typically want to maximize the separation among classes. Linear discriminant analysis (LDA) generally works well for that. For regression, we want to maximize the correlation between the projected data and the output, and partial least squares (PLS) works well. Finally, in unsupervised tasks, we typically want to retain as much of the variance as possible. PCA is the most widely used technique for doing that.

Principal component analysis

PCA is called principal component analysis because it learns the "principal components" of the data. These are the directions in which the samples vary the most, depicted in Figure 6-4 as a dashed line. It is the principal components that PCA aligns with the coordinate axes.

PCA is available in scikit-learn and in TF Transform, which is especially useful in a production pipeline using TFX.

PCA, an unsupervised method, constructs new features through linear combinations of the original ones. It achieves dimensionality reduction in two steps, starting with a decorrelation process that maintains the original

number of dimensions. In this first step, PCA rotates the data points to align them with the coordinate axes and centers them by shifting their mean to zero.

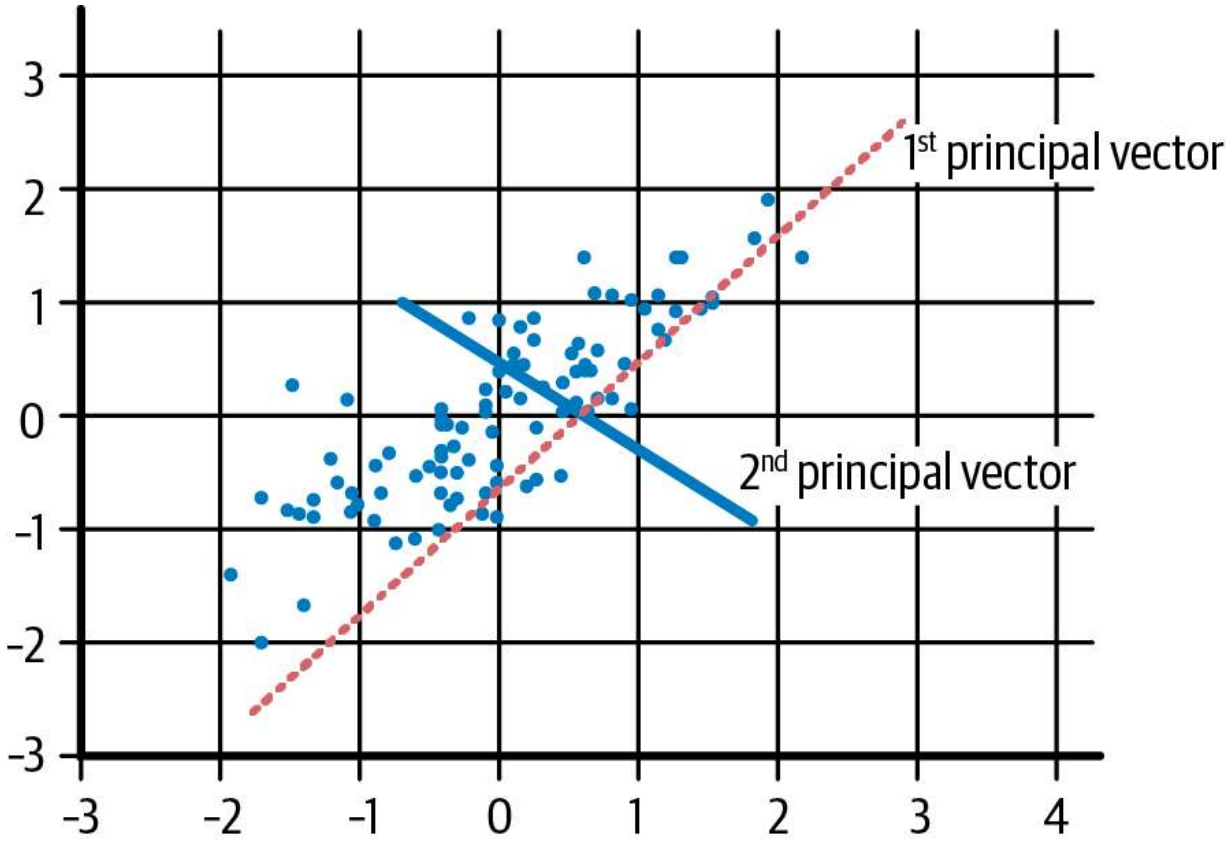

Figure 6-4. An example of PCA

The goal of PCA is to find a lower-dimensional surface onto which to project the data so that it minimizes the squared projection error—or in other words, to minimize the square of the distance between each point and the location where it gets projected. The result will be to maximize the variance of the projections. The initial principal component represents the projection direction that yields the highest variance in the projected data.

Subsequently, the second principal component is identified as the projection direction perpendicular to the first, while also maximizing the remaining variance within the projected data.

We won’t go into more detail here on how PCA works, but if you’re interested, there are many good resources available. PCA is a practical and effective technique known for its speed and ease of implementation. This allows for convenient comparison of algorithm performance with and without PCA. Furthermore, PCA boasts various adaptations and extensions, such as kernel PCA and sparse PCA, to address specific challenges. However, the resulting principal components are often not readily interpretable, which can be a significant drawback in scenarios where interpretability is crucial. Additionally, you must still manually determine or adjust a threshold for cumulative explained variance.

PCA is especially useful when visually studying clusters of observations in high dimensions. This could be when you are still exploring the data. For example, you may have reason to believe that the data is inherently *low rank*, which means there are many attributes but only a few attributes that mostly determine the rest, through a linear association. PCA can help you test that theory.

Quantization and Pruning

Model optimization is another area of focus where you can further optimize performance and resource requirements. The goal is to create models that are as efficient and accurate as possible in order to achieve the highest performance at the least cost. Let’s look at two advanced techniques: quantization and pruning. We’ll start by looking at some of the issues around mobile, Internet of Things (IoT), and embedded applications.

Mobile, IoT, Edge, and Similar Use Cases

ML is increasingly becoming part of more and more devices and products. This includes the rapid growth of mobile and IoT applications, including devices that are situated everywhere from farmers’ fields to train tracks. Businesses are using the data these devices generate to train ML models to improve their business processes, products, and services. Even digital advertisers spend more on mobile than desktop. There are already billions of mobile and edge computing devices, and that number will continue to grow rapidly in the next decade.

Quantization

Quantization is a process in which a model is converted into a functionally equivalent representation that uses parameters and computations with

reduced precision, meaning fewer bits are used. This technique enhances the model's execution speed and efficiency, but it may lead to a decrease in overall model accuracy.

Benefits and process of quantization

Let's use an analogy to understand this better. Think of an image. As you might know, a picture is a grid of pixels, where each pixel has a certain number of bits. If you try reducing the continuous color spectrum of real life to discrete colors, you are quantizing or approximating the image. Quantization, in essence, lessens the number of bits needed to represent information. However, you may notice that as you reduce the number of possible colors beyond a certain point, depending on the image, the quality of the image may suffer. Generally speaking, quantization will always reduce model accuracy, so there is a trade-off between the benefits of quantization and the amount of accuracy lost as a result.

Neural networks comprise activation nodes, their interconnections, weight parameters assigned to each connection, and bias terms. In the context of quantization, the primary focus is on quantizing these weight parameters and the computations performed within the activation nodes.

Neural network models often occupy a significant amount of storage space, primarily due to the numerous model parameters (weights associated with neural connections), which can number in the millions or even billions

within a single model. These parameters, being distinct floating-point numbers, are not easily compressed through conventional methods like zipping unless the model's density is reduced.

However, model parameters can also be quantized to turn them from floating-point to integer values. This reduces the model size, and also usually speeds inference since integer operations are usually faster than floating-point operations. Even quantizing a 16-bit floating-point model down to 4-bit integers has been shown to deliver acceptable results.

Quantization inherently involves some loss of information. However, weights and activations within a given layer often cluster within a narrow, predictable range. This allows us to allocate our limited bits within a smaller, predetermined range (e.g., –3 to +3), thus optimizing precision. Accurate estimation of this range is critical. When executed correctly, quantization results in minimal precision loss, typically with negligible impacts on the output.

The most straightforward motivation for quantization is to shrink file sizes and memory requirements. For mobile apps especially, it's often impractical to store a 200 MB model on a phone just to run a single app. So, compressing higher-precision models is necessary.

Another reason to quantize is to minimize the computational resources required for inference calculations by performing them exclusively with

low-precision inputs and outputs. Although this is a lot more challenging, necessitating modifications throughout the calculation process, it can yield substantial benefits. For example, it can help you run your models faster and use less power, which is especially important on mobile devices. It even opens the door to a lot of embedded systems that can’t run floating-point code efficiently, enabling many applications in the IoT world.

However, optimizations can sometimes impact model accuracy, a factor you must account for during application development. These accuracy changes are specific to the model and data you’re optimizing and are challenging to foresee. Generally, it’s reasonable to expect some level of accuracy degradation in models optimized for size or latency. Depending on your application, this may or may not impact your users’ experience. In rare cases, certain models may actually gain some accuracy as a result of the optimization process.

You will need to make a trade-off between model accuracy and model complexity. If your task requires high accuracy, you may need a large and complex model. For tasks that require less precision, it’s better to use a smaller, less complex model because it not only will use less disk space and memory but also will generally be faster and more energy efficient. So, once you have selected a candidate model that is right for your task, it’s a good practice to profile and benchmark your model.

MobileNets

MobileNets are a family of architectures that achieve a state-of-the-art trade-off between on-device latency and ImageNet classification accuracy. A study from Google Research demonstrated how integer-only quantization could further improve the trade-off on common hardware. The authors of the paper benchmarked the MobileNet architecture with varying-depth multipliers and resolutions on ImageNet on three types of Qualcomm cores. Figure 6-5 shows results for the Snapdragon 835 chip. You can see that for any given level of accuracy, latency time (runtime) is lower for the 8-bit version of the model than for the float version (shifted to the left).

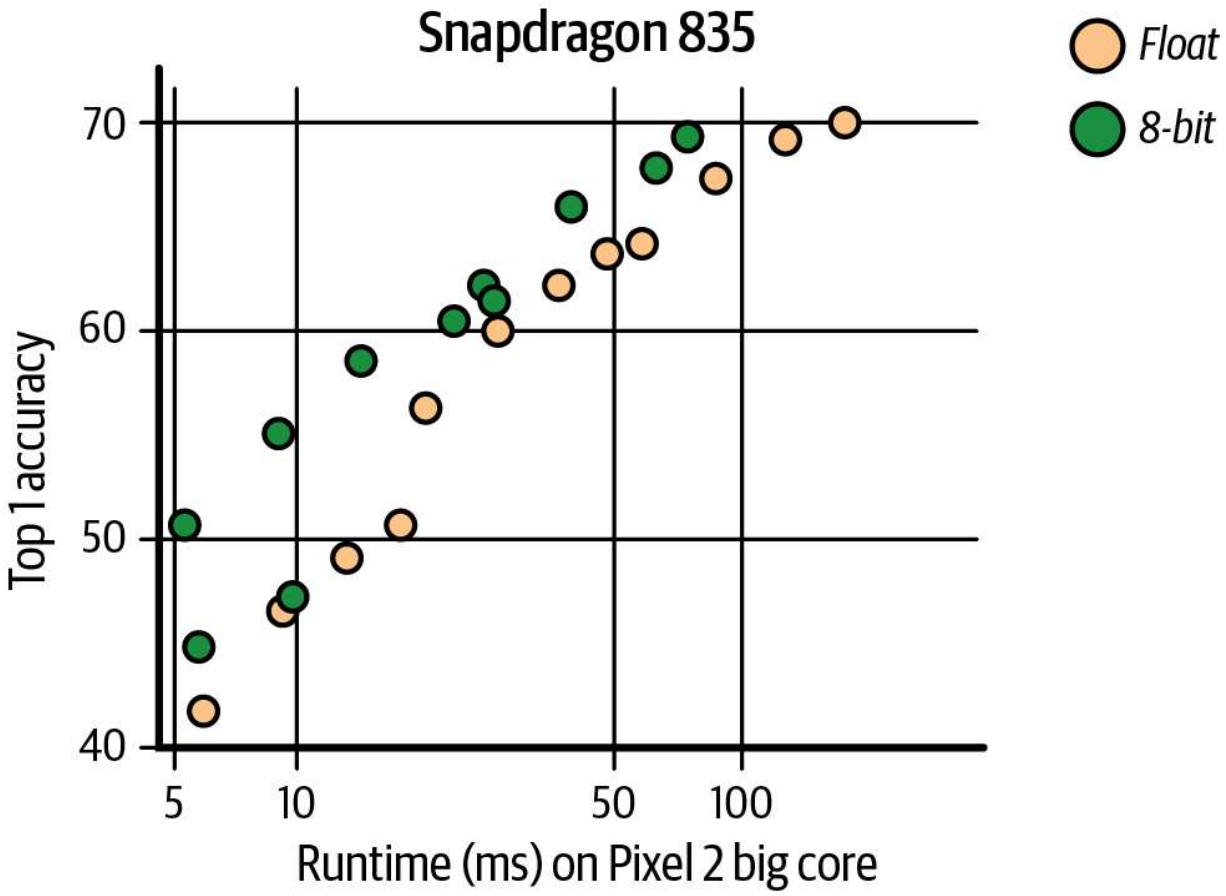

Figure 6-5. Accuracy versus runtime (ms) for 8-bit and float MobileNet models (sources: Yang et al., 2019; Jacob et al., 2017)

Arithmetic operations performed with reduced bit depth tend to be faster, provided the hardware supports it. While modern CPUs have largely bridged the performance gap between floating-point and integer computation, operations involving 32-bit floating-point numbers will almost generally still be slower than, for example, 8-bit integers.

In moving from 32 bits to 8 bits, we usually get speedups and a 4x reduction in memory. Smaller models use less storage space, are easier to share over smaller bandwidths, and are easier to update. Lower bit depths

also mean we can squeeze more data into the same caches and registers. This makes it possible to build applications with better caching capabilities, which reduces power usage and increases speed.

Floating-point arithmetic is hard, which is why it may not always be supported on microcontrollers and on some ultra-low-power embedded devices, such as drones, watches, or IoT devices. Integer support, on the other hand, is always available.

Post-training quantization

The most straightforward method for quantizing a neural network involves training it initially with full precision and subsequently quantizing the weights to fixed points. This is known as *post-training quantization*. You can perform quantization either during training (quantization-aware training), or after the model has been trained (post-training quantization). Let’s begin by examining post-training quantization.

Post-training quantization aims to decrease the size of an already-trained model, with the objective of enhancing CPU and hardware accelerator latency, ideally without significantly impacting model accuracy. For example, you can readily quantize a pretrained float TensorFlow model when you convert it to TensorFlow Lite (TF Lite) format using the TensorFlow Lite Converter.

At a basic level, what post-training quantization does is convert, or more precisely, quantize the weights, from floating-point numbers to integers in an efficient way. By doing that, you can often gain up to three times lower latency without taking a major hit on accuracy. With TF Lite's default optimization strategy, the converter will do its best to apply post-training quantization, trying to optimize the model both for size and latency. This is recommended, but you can also customize this behavior.

There are several post-training quantization options to choose from. Table 6-1 summarizes the choices and the benefits they provide.

Table 6-1. Post-training quantization techniques and benefits

Technique	Benefits
Dynamic range quantization	4x smaller, 2x–3x speedup
Full integer quantization	4x smaller, 3x+ speedup
Float16 quantization	2x smaller, GPU acceleration

If you're looking for a decent speedup, such as two to three times faster while being two times smaller, you can consider dynamic range quantization. With *dynamic range quantization*, during inference the weights are converted from 8 bits to floating point and the activations are computed using floating-point kernels. This conversion is done once, and

cached to reduce latency. This optimization provides latencies that are close to fully fixed-point inference.

Using dynamic range quantization, you can reduce the model size and/or latency. But this comes with a limitation, as it requires inference to be done with floating-point numbers. This may not always be ideal, since some hardware accelerators only support integer operations (e.g., Edge TPUs).

On the other hand, if you want to squeeze even more performance from your model, full integer quantization or float16 quantization may result in faster performance. Float16 is especially useful when you plan to use a GPU.

The TF Lite optimization toolkit also supports full integer quantization. This enables users to take an already-trained floating-point model and fully quantize it to only use 8-bit signed integers, which enables fixed-point hardware accelerators to run these models. When targeting greater CPU improvements or fixed-point accelerators, this is often a better option.

Full integer quantization works by gathering calibration data, which it does by running inferences on a small set of inputs to determine the right scaling parameters needed to convert the model to an integer-quantized model.

Post-training quantization can result in a loss of accuracy, particularly for smaller networks, but the loss is often fairly negligible. On the plus side, this will speed up execution of the heaviest computations by using lower

precision, and by using the most sensitive computations with higher precision, thus typically resulting in little to no final loss of accuracy.

Pretrained fully quantized models are also available for specific networks in the TF Lite model repository. It is important to check the accuracy of the quantized model to verify that any degradation in accuracy is within acceptable limits. TF Lite includes a tool to evaluate model accuracy.

Quantization-aware training

Alternatively, if the loss of accuracy from post-training quantization is too great, consider using quantization-aware training. However, doing so requires modifications during model training to add fake quantization nodes.

Quantization-aware training applies quantization to the model while it is being trained. The core idea is that quantization-aware training simulates low-precision inference-time computation in the forward pass of the training process.

By introducing simulated quantization nodes, the rounding effects that would typically happen during real-world inference due to quantization are replicated during the forward pass. The intention here is to fine-tune the weights to compensate for any loss of precision. So, if these simulated quantization nodes are incorporated into the model graph at the specific locations where quantization is expected to occur (e.g., at convolutions),

then in the forward pass the float values will be rounded to the specified number of levels to simulate the effects of quantization.

This approach incorporates quantization error as noise during the training process, treating it as part of the overall loss that the optimization algorithm seeks to minimize. Consequently, the model learns parameters that are more resilient to quantization. In quantization-aware training, you start by constructing a model in the standard way and then use the TensorFlow Model Optimization toolkit's APIs to make it quantization-aware. Then, you train this model with the quantization emulation operations to obtain a fully quantized model that operates solely with integers.

Comparing results

Table 6-2 shows the loss of accuracy on a few models. This should give you a feel for what to expect in your own models.

Table 6-2. Comparing resulting accuracy from post-training quantization with quantization-aware training

Model	Top-1 accuracy (original)	Top-1 accuracy (post-training quantized)	Top-1 accuracy (quantization-aware training)
Mobilenet-v1-1-224	0.709	0.657	0.70
Mobilenet-v2-1-224	0.719	0.637	0.709
Inception_v3	0.78	0.772	0.775
Resnet_v2_101	0.770	0.768	N/A

Table 6-3 shows the change in latency for a few models. Remember that for latency, lower numbers are better.

Table 6-3. Comparing resulting latency (ms) from post-training quantization with quantization-aware training

Model	Latency (original) (ms)	Latency (post-training quantized) (ms)	Latency (quantization-aware training) (ms)
Mobilenet-v1-1-224	124	112	64
Mobilenet-v2-1-224	89	98	54
Inception_v3	1130	845	543
Resnet_v2_101	3973	2868	N/A

Table 6-4 compares model size. Both post-training and quantization-aware training give approximately the same size reduction. Again, lower numbers are better.

Table 6-4. Comparing resulting model sizes from quantization

Model	Size (original) (MB)	Size (optimized) (MB)
Mobilenet-v1-1-224	16.9	4.3
Mobilenet-v2-1-224	14	3.6
Inception_v3	95.7	23.9
Resnet_v2_101	178.3	44.9

Example: Quantizing models with TF Lite

The TensorFlow ecosystem provides a number of libraries to export models to different platforms such as mobile devices or web browsers. Usually those devices come with hardware constraints; for example, mobile devices are limited in accessible memory.

TensorFlow lets you optimize ML models for such devices through the TF Lite library. There are a few caveats to consider when optimizing with TF Lite. For example, not all TensorFlow operations can be converted to TF Lite. But the list of supported operations is continually growing.

You can deploy models converted to TF Lite with TensorFlow Serving, which we'll show you in Chapter 20.

Optimizing Your TensorFlow Model with TF Lite

At the time of this writing, TF Lite supported the following model formats:

- TensorFlow's SavedModel format
- Keras models
- TensorFlow's concrete functions
- JAX models

TF Lite provides a variety of optimization options and tools. You can convert your model through command-line tools or through the Python library. The starting point is always your trained and exported ML model in one of the formats in the preceding list.

In the following example, we load a Keras model:

```
import tensorflow as tf
model = tf.keras.models.load_model("model.h5")
```

Next, we create a converter object in which we'll hold all the optimization parameters:

```
converter = tf.lite.TFLiteConverter.from_keras_mo
```

After creating the converter object, we can define our optimization parameters. This can be the objective of the optimization, the supported TensorFlow ops, or the input/output types:

```
converter.optimizations = [tf.lite.Optimize.DEFAU
converter.target_spec.supported_ops = [tf.lite.Op
converter.inference_input_type = tf.int8  # or tf
converter.inference_output_type = tf.int8  # or t
```

After defining all the parameters, we can convert the model by calling `converter.convert()` and then save the returned object:

```
tflite_quantized_model = converter.convert()
with open('your_quantized_model.tflite', 'wb') as
    f.write(tflite_quantized_model)
```

We can now consume the quantized model, either by integrating the TF Lite model `your_quantized_model.tflite` in a mobile application or by consuming it with TensorFlow Serving (we will discuss this in more detail in Chapter 11).

Optimization Options

Older TF Lite documentation offered two optimization options:

- `OPTIMIZE_FOR_SIZE`
- `OPTIMIZE_FOR_LATENCY`

Those two options have been deprecated and are now replaced by a new optimization option: `EXPERIMENTAL_SPARSITY`. This option inspects the model for sparsity patterns of the model parameters and improves the model's size and latency accordingly. It can be combined with the `DEFAULT` option:

```
...
converter.optimizations = [
    tf.lite.Optimize.DEFAULT,
    tf.lite.EXPERIMENTAL_SPARSITY]
tflite_model = converter.convert()
...
```

If your model includes a TensorFlow operation that is not supported by TF Lite at the time of exporting your model, the conversion step will fail with an error message. You can enable an additional set of selected TensorFlow operations to be available for the conversion process. However, this will increase the size of your TF Lite model by approximately 30 MB. The following code snippet shows how to enable the additional TensorFlow operations before the converter is executed:

```
...
converter.target_spec.supported_ops = [
    tf.lite.OpsSet.TFLITE_BUILTINS,
    tf.lite.OpsSet.SELECT_TF_OPS]
tflite_model = converter.convert()
...
```

If the conversion of your model still fails due to an unsupported TensorFlow operation, you can bring it to the attention of the TensorFlow community. The community is actively increasing the number of operations supported by TF Lite and welcomes suggestions for future operations to be included in TF Lite. TensorFlow ops can be nominated via the TF Lite Op Request form in GitHub.

Pruning

Another method to increase the efficiency of models is to remove parts of the model that do not contribute substantially to producing accurate results. This is referred to as *pruning*.

As ML models were pushed into embedded devices such as mobile phones, compressing neural networks grew in importance. Pruning in deep learning is a biologically inspired concept that mimics some of the behavior of neurons in the brain. Pruning strives to reduce the computational

complexity of a neural network by eliminating redundant connections, resulting in fewer parameters and potentially faster inference.

Networks generally look like the one on the left in Figure 6-6. Here, every neuron in a layer has a connection to the layer before it, but this means we have to multiply a lot of floats together.

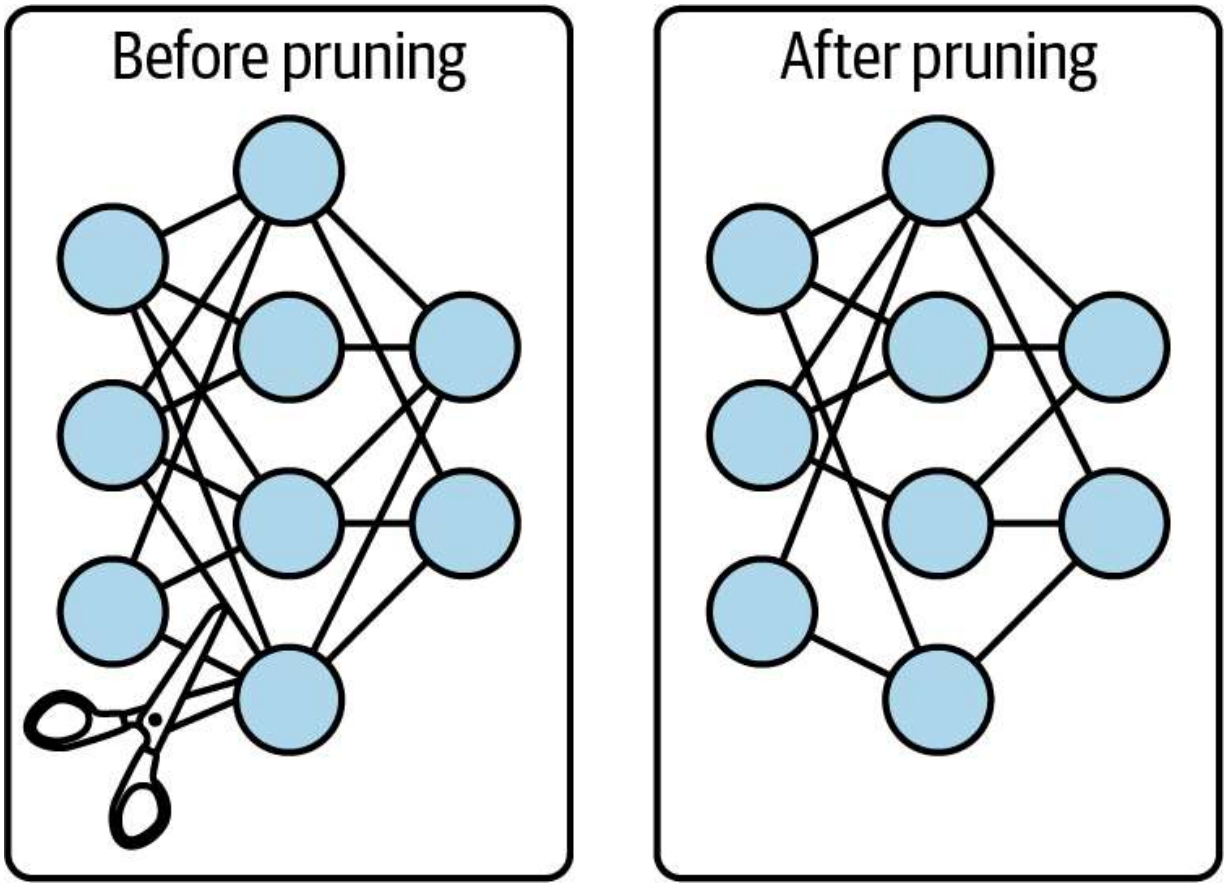

Figure 6-6. Before and after pruning

Ideally, we'd only connect each neuron to a few others and save on doing some of the multiplications, if we can find a way to do that without too much loss of accuracy. That's the motivation behind pruning.

Connection sparsity has long been a foundational principle in neuroscience research, as it is one of the critical observations about the neocortex. Everywhere you look in the brain, the activity of neurons is always sparse. But common neural network architectures have a lot of parameters that generally aren't sparse. Take, for example, ResNet50. It has almost 25 million connections. This means that during training, we need to adjust 25 million weights. Doing that is relatively costly, to say the least. So, there's a need to fix this somehow.

The story of sparsity in neural networks starts with pruning, which is a way to reduce the size of the neural network through compression. Where hardware is limited, such as in embedded devices or smartphones, speed and size can make or break a model. Also, more complex models are more prone to overfitting. So, in some sense, restricting the search space can also act as a regularizer. However, it's not a simple task, since reducing the model's capacity can also lead to a loss of accuracy. So, as in many other areas, there is a delicate balance between complexity and performance.

The first major paper advocating sparsity in neural networks dates back to 1990. Written by Yann Le Cun, John S. Denker, and Sara A. Solla, the paper has the rather provocative title of "Optimal Brain Damage." At the time, post-pruning neural networks to compress trained models was already a popular approach. Pruning was mainly done by using magnitude as an approximation for saliency to determine less useful connections—the intuition being that smaller-magnitude weights have a smaller effect in the

output, and hence are less likely to have an impact in the model outcome if pruned.

It was a sort of iterative pruning method. The first step was to train a model. Then, the saliency of each weight was estimated, which was defined by the change in the loss function upon applying perturbation to the weights in the network. The smaller the change, the less effect the weight would have on the training. Finally, the authors eliminated the weights with the lowest saliency (this is equivalent to setting them to zero), and then this pruned model was retrained.

One particular challenge with this method arises when the pruned network is retrained. It turned out that due to its decreased capacity, retraining was much more difficult. The solution to this problem arrived later, along with an insight called the Lottery Ticket Hypothesis.

The Lottery Ticket Hypothesis

The probability of winning the jackpot of a lottery is very low. For example, if you're playing Powerball, you have a probability p of 1 in about 3 million to win per ticket. What are your chances if you purchase N tickets?

For N tickets we have a probability of (1 – p) to the power of N. From this, it follows that the probability of at least one of the tickets winning is simply the complement again. What does this have to do with neural networks? Before training, the weights of a model are initialized randomly. Can it

happen that there is a subnetwork of a randomly initialized network that won the initialization lottery?

Some researchers set out to investigate the problem and answer that question. Most notably, Frankle and Carbin in 2019 found that fine-tuning the weights after training was not required for these new pruned networks. In fact, they showed that the best approach was to reset the weights to their original value, and then retrain the entire network. This would lead to models with even higher accuracy, compared to both the original dense model and the post-pruning plus fine-tuning approach.

This discovery led Frankle and Carbin to propose an idea considered wild at first, but now commonly accepted—that overparameterized dense networks contain several sparse subnetworks, with varying performances, and one of these subnetworks is the winning ticket that outperforms all the others.

However, there were significant limitations to this method. For one, it does not perform well for larger-scale problems and architectures. In the original paper, the authors stated that for more complex datasets like ImageNet, and for deeper architectures like ResNet, the method fails to identify the winners of the initialization lottery. In general, achieving a good sparsity–accuracy trade-off is a difficult problem. At the time of this writing, this is a very active research field, and the state of the art keeps improving.

Pruning in TensorFlow

TensorFlow includes a Keras-based weight pruning API that uses a straightforward yet broadly applicable algorithm designed to iteratively remove connections based on their magnitude during training. Fundamentally, a final target sparsity is specified, along with a schedule to perform the pruning.

During training, a pruning routine will be scheduled to execute, removing the weights with the lowest-magnitude values that are closest to zero until the current sparsity target is reached. Every time the pruning routine is scheduled to execute, the current sparsity target is recalculated, starting from 0%, until it reaches the final target sparsity at the end of the pruning schedule by gradually increasing it according to a smooth ramp-up function.

Just like the schedule, the ramp-up function can be tweaked as needed. For example, in certain cases, it may be convenient to schedule the training procedure to start after a certain step when some convergence level has been achieved, or to end pruning earlier than the total number of training steps in your training program, in order to further fine-tune the system at the final target sparsity level.

Sparsity increases as training proceeds, so you need to know when to stop. That means at the end of the training procedure, the tensors corresponding

to the pruned Keras layers will contain zeros where weights have been pruned, according to the final sparsity target for the layer.

An immediate benefit that you can get out of pruning is disk compression. That's because sparse tensors are compressible. Thus, by applying simple file compression to the pruned TensorFlow checkpoint or the converted TF Lite model, we can reduce the size of the model for storage and/or transmission. In some cases, you can even gain speed improvements in CPU and ML accelerators that exploit integer precision efficiencies. Moreover, across several experiments, we found that weight pruning is compatible with quantization, resulting in compound benefits.

Knowledge Distillation

So far we've discussed ways to optimize the implementation of models to make them more efficient. But you can also try to capture or "distill" the knowledge that has been learned by a model into a more efficient or compact model, by using a different style of training. This is known as *knowledge distillation*.

Teacher and Student Networks

Models tend to become larger and more complex as they try to capture more information, or knowledge, in order to learn complex tasks. A larger, more complex model requires more compute resources to generate

predictions, which is a disadvantage in any style of deployment, but especially in a mobile deployment where compute resources are limited.

But if we can express or represent this learning more efficiently, we might be able to create smaller models that are equivalent to these larger, more complex models, as shown in Figure 6-7.

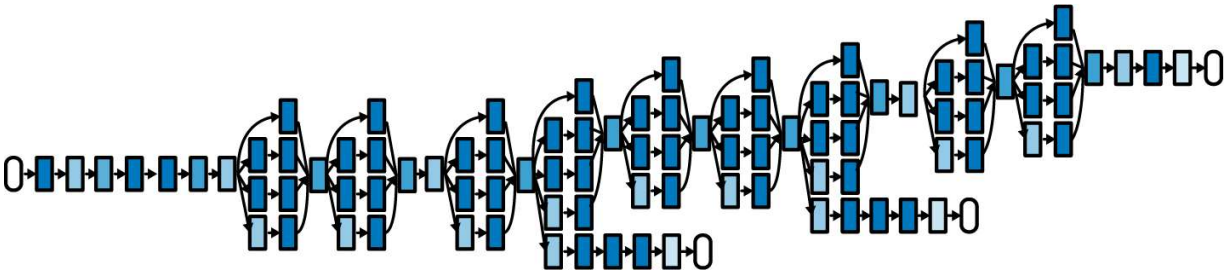

Figure 6-7. A complex model (source: Szegedy et al., 2014)

For example, consider GoogLeNet, depicted in Figure 6-7. Today it's considered a reasonably small or perhaps midsize network, but even so it's still deep and complex enough that it's hard to fit on the page. The fact that it is so deep gives it the ability to express complex relationships between features, which is the power that many applications need. But it's large enough that it's difficult or impossible to deploy it in many production environments, including mobile phones and edge devices.

So, can you have the best of both worlds, and capture the knowledge contained in a complex model like GoogLeNet in a much smaller, more efficient model?

That's the goal of *knowledge distillation*. Rather than optimizing the network implementation as we saw with quantization and pruning, knowledge distillation seeks to create a more efficient model that captures the same knowledge as a more complex model. If needed, further optimization can then be applied to the result.

Knowledge distillation is a way to train a small model to mimic a larger model, or even an ensemble of models. It starts by first training a complex model or model ensemble to achieve a high level of accuracy. As shown in Figure 6-8, it then uses that model as a "teacher" for the simpler "student" model, which will be the actual model that gets deployed to production. This teacher network can be either fixed or jointly optimized, and can even be used to train multiple student models of different sizes simultaneously.

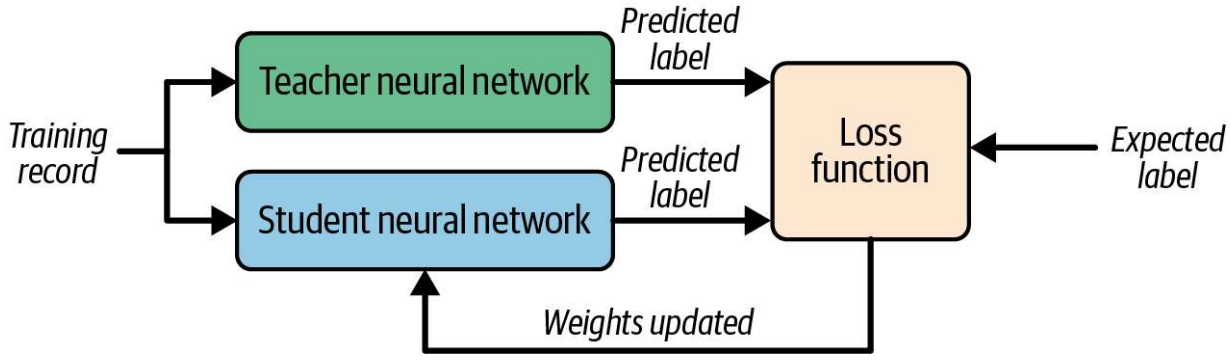

Figure 6-8. Teacher and student network

Knowledge Distillation Techniques

In model distillation, the training objective functions are different for the teacher and the student:

- The teacher will be trained first, using a standard objective function that seeks to maximize the accuracy (or a similar metric) of the model. This is normal model training.
- The student then seeks transferable knowledge. So, it uses an objective function that seeks to match the probability distribution of the predictions of the teacher.

Notice that the student isn't just mimicking the teacher's predictions, but rather internalizing the probabilities associated with those predictions. These probabilities serve as "soft targets," conveying richer insights into the teacher's knowledge than the mere predictions themselves.

Knowledge distillation operates by transferring knowledge from the teacher to the student by minimizing a loss function. Here, the target is the distribution of class probabilities as predicted by the teacher model. Typically, the teacher model's logits act as input to the final softmax layer because of the additional information they provide regarding the probabilities of all target classes for each example. However, in reality, this distribution often heavily favors the correct class, with negligible probabilities for others. Consequently, it may not offer much more information than the ground truth labels already present in the dataset.

To address this limitation, Hinton, Vinyals, and Dean introduced the concept of a *softmax temperature*. By increasing this temperature in the objective functions of both the student and teacher, you can enhance the

softness of the teacher's distribution, as illustrated by the following formula:

$$p_i = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_{j=1}^{n} \exp\left(\frac{z_j}{T}\right)}$$

In this formula, the probability P of class i is derived from the logits z as shown. T represents the temperature parameter. When T equals 1, you get the standard softmax function. However, as T increases, the softmax function produces a softer probability distribution, revealing more about which classes the teacher model perceived as similar to the predicted class. This nuanced information within the teacher model, which the authors call *dark knowledge*, is what you transfer to the student model during distillation. This captures the teacher's soft targets or soft logits, which the student aims to replicate.

Several techniques are used to train the student to match the teacher's soft targets. One approach involves training the student on both the teacher's logits and the target labels, using a standard objective function. These two objective functions are then weighted and combined during backpropagation. Another common method compares the distributions of the student's predictions and the teacher's predictions using a metric such as Kullback–Leibler (K–L) divergence.

When computing the loss function versus the teacher's soft targets, you use the same value of T to compute the softmax on the student's logits. This

loss is called the *distillation loss*. The authors also found another interesting behavior. It turns out that the distilled models are able to produce the correct labels in addition to the teacher's soft labels. This means you can calculate the "standard" loss between the student's predicted class probabilities and the ground truth labels. These are known as *hard labels* or *hard targets*. This loss is the *student loss*. So when you're calculating the probabilities for the student, you set the softmax temperature to 1:

$$L = (1 - \alpha)L_H + \alpha L_{KL}$$

In this approach, knowledge distillation is done by blending two loss functions, choosing a value for alpha of between 0 and 1. Here, $L_{\mathbf{H}}$ is the cross-entropy loss from the hard labels and $L_{\mathbf{KL}}$ is the K–L divergence loss from the teacher's logits. In case of heavy augmentation, you simply cannot trust the original hard labels due to the aggressive perturbations applied to the data.

The K–L divergence here is a metric of the difference between two probability distributions. You want those two probability distributions to be as close as possible, so the objective is to make the distribution over the classes predicted by the student as close as possible to the teacher.

The initial quantitative outcomes from applying knowledge distillation were encouraging. Hinton et al. trained 10 distinct models for an automatic speech recognition task, maintaining the same architecture and training procedure as the baseline. At that time, deep neural networks were

employed in automatic speech recognition to map a short temporal context of features. The models were initialized with different random weight values to ensure diversity in the trained models, allowing their ensemble predictions to easily surpass those of individual models. The models were initialized with different random weight values to ensure diversity in the trained models, allowing their ensemble predictions to easily surpass those of individual models. They considered varying the training data for each model, but found it had minimal impact on results, so they adopted a simpler strategy of comparing an ensemble against a single model.

For the distillation process, they tried different values for the softmax temperature, such as 1, 2, 5, and 10. They also used a relative weight of 0.5 on the cross-entropy for the hard targets.

Table 6-5 shows that distillation can indeed extract more useful information from the training set than merely using the hard labels to train a single model.

Table 6-5. Comparing accuracy and word error rate for a distilled model

Model	Accuracy	Word error rate
Baseline	58.9%	10.9%
10x ensemble	61.1%	10.7%
Distilled single model	60.8%	10.7%

Comparing the single baseline model to the 10x ensemble, we can see an improvement in accuracy. Then, comparing the ensemble to the distilled model, we can see that more than 80% of that improvement in accuracy is transferred to the single distilled model. The ensemble provides only a modest improvement in word error rate on a 23,000-word test set, likely due to the mismatch in the objective function. Nevertheless, the reduction in word error rate achieved by the ensemble is successfully transferred to the distilled model. This demonstrates that their model distillation strategy is effective and can be used to compress the ensemble of models into a single model that performs significantly better than a model of the same size trained directly from the same data.

That test was performed during research into knowledge distillation. In the real world, though, people are more interested in deploying a “low-resource” model, with close to state-of-the-art results, but a lot smaller and a lot faster. Hugging Face developed DistilBERT, a streamlined version of the BERT model that reduces parameters by 40% and increases speed by 60%, while still retaining 97% of BERT’s performance on the GLUE language understanding benchmark. Basically, it’s a smaller version of BERT where the token-type embeddings and the pooler layer typically used for the next sentence classification task are removed. To create DistilBERT, the researchers at Hugging Face applied knowledge distillation to BERT (hence, the name DistilBERT). They kept the rest of the architecture identical, while reducing the numbers of layers.

TMKD: Distilling Knowledge for a Q&A Task

Let’s look at how knowledge can be distilled for question answering. Applying these complex models to real business scenarios becomes challenging due to the vast number of model parameters. Older model compression methods generally suffer from information loss during the model compression procedure, leading to inferior models compared to the original one.

To tackle this challenge, researchers at Microsoft proposed a Two-stage Multi-teacher Knowledge Distillation (TMKD) method for a Web Question Answering system. In this approach, they first develop a general Q&A distillation task for student model pretraining, and further fine-tune this pretrained student model with multiteacher knowledge distillation on downstream tasks like the Web Q&A task. This can be used to effectively reduce the overfitting bias in individual teacher models, and it transfers more general knowledge to the student model.

The basic knowledge distillation approach presented so far is known as a *1-on-1 model* because one teacher transfers knowledge to one student. Although this approach can effectively reduce the number of parameters and the time for model inference, due to the information loss during knowledge distillation, the performance of the student model is sometimes not on par with that of its teacher.

This was the driving force for the Microsoft researchers to create a different approach, called an *m-on-m ensemble model*, combining both ensembling and knowledge distillation. This involves first training multiple teacher models. The models could be BERT or GPT or other similarly powerful models, each having different hyperparameters. Then, a student model for each teacher model is trained. Finally, the student models trained from different teachers are ensembled to generate the final results. With this technique, you prepare and train each teacher for a particular learning objective. Different student models have different generalization capabilities, and they also overfit the training data in different ways, achieving performance close to the teacher model.

TMKD outperforms various state-of-the-art baselines and has been applied to real commercial scenarios. Since ensembling is employed here, these compressed models benefit from large-scale data, and they learn feature representations well. Results from experiments show that TMKD can considerably outperform baseline methods, and even achieve comparable results to the original teacher models, along with a substantial speedup of model inference (see Figure 6-9).

The authors performed experiments on several datasets using benchmarks that are public, and even large scale, to verify the method's effectiveness. To support these claims, let's look at TMKD's advantages one by one. A unique aspect of TMKD is that it uses a multiteacher distillation task for

student model pretraining to boost model performance. To analyze the impact of pretraining, the authors evaluated two models.

The first one (TKD) is a three-layer BERT-based model, which is first trained using basic knowledge distillation pretraining on the CommQA dataset and then fine-tuned on a task-specific corpus by using only one teacher for each task. The second model is a traditional knowledge distillation model, which is again the same model but without the distillation pretraining stage. TKD showed significant gains by leveraging large-scale unsupervised Q&A pairs for distillation pretraining.

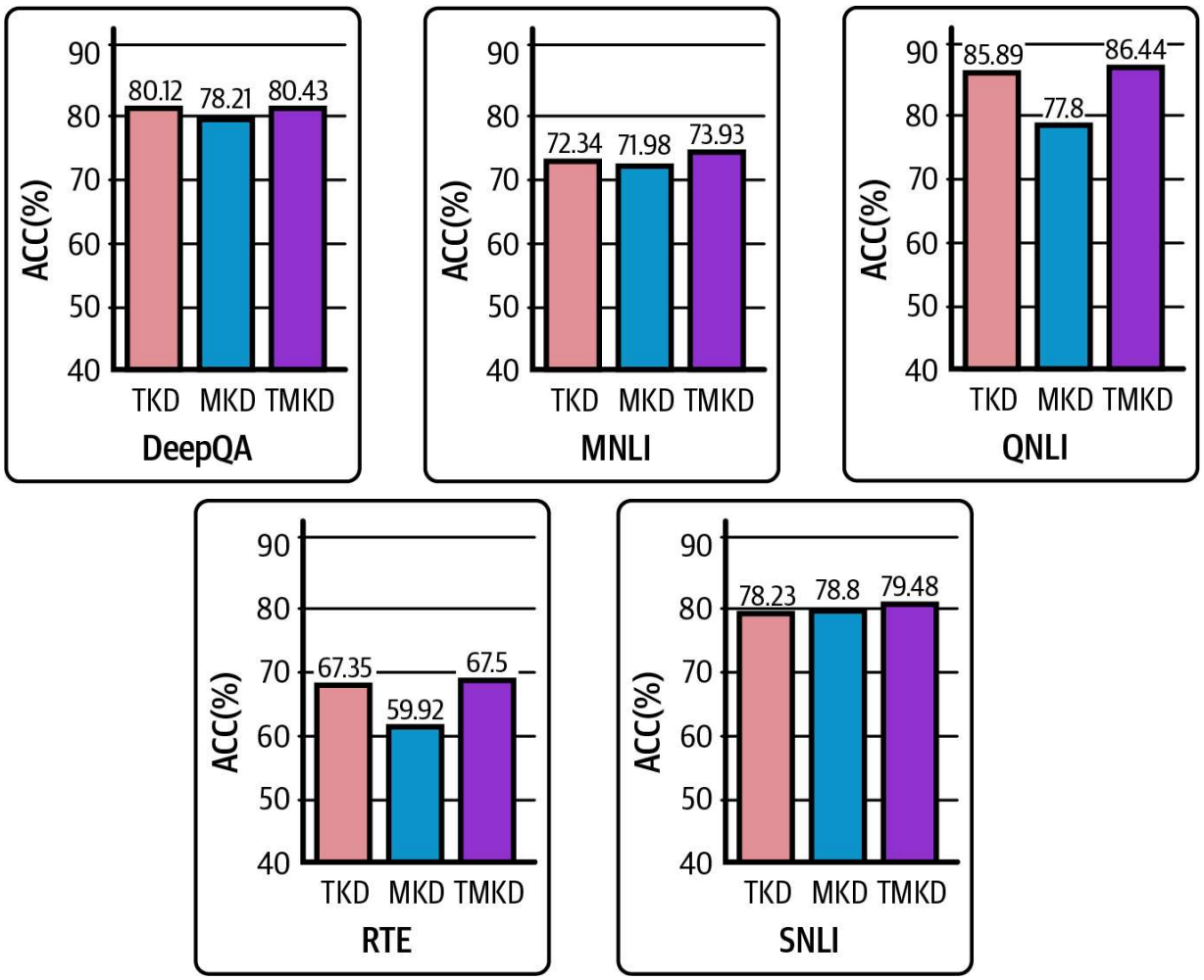

Figure 6-9. TMKD results on DeepQA, MNLI, SNLI, QNLI, and RTE datasets (source: Yang et al., 2019)

Another benefit of TMKD is its unified framework to learn from multiple teachers jointly. For this, the authors were able to compare the impact of multiteacher versus single-teacher knowledge distillation using two models—MKD, a three-layer BERT-based model trained by multiteacher distillation without a pretraining stage; and KD, a three-layer BERT-based model trained by single-teacher distillation without a pretraining stage, whose aim is to learn from the average score of the teacher models. MKD outperformed KD on the majority of tasks, demonstrating that a

multiteacher distillation approach can help the student model learn more generalized knowledge, fusing knowledge from different teachers.

Finally, they compared TKD, MKD, and TMKD with one another. As you can see in Figure 6-9, TMKD significantly outperformed TKD and MKD in all datasets, which verifies the complementary impact of the two stages—distillation pretraining and multiteacher fine-tuning.

Increasing Robustness by Distilling EfficientNets

In another example, researchers from Google Brain and Carnegie Mellon University trained models with a semi-supervised learning method called *noisy student*. In this approach, the knowledge distillation process is iterative. It uses a variation of the classic teacher–student paradigm, but here the student is purposefully kept larger than the teacher in terms of the number of parameters. This is done so that the model can attain robustness to noisy labels as opposed to traditional knowledge distillation patterns.

This works by first training an EfficientNet as the teacher model using labeled images, and then using the teacher to generate pseudolabels on a larger set of unlabeled images.

Subsequently, they trained a larger EfficientNet model as a student using both labeled and pseudo-labeled images and repeated the process multiple times; the student model was promoted to a teacher role to relabel the

unlabeled data and train a new student model. An important element of the approach was to ensure that noise was added to the student model using dropout, stochastic depth, and data augmentation via `RandAugment` during its training. This noising pushed it to learn harder from pseudolabels. Adding noise to a student model ensures that the task is much harder for the student (hence the name “noisy student”) and that it doesn’t merely learn the teacher’s knowledge. On a side note, the teacher model is not noised during the generation of pseudolabels, to ensure its accuracy isn’t altered in any way.

The loop closes by replacing the teacher with the optimized student network.

To compare the results of noisy student training, the authors used EfficientNets as their baseline models. Figure 6-10 shows different sizes of EfficientNet models along with some well-known state-of-the-art models for comparison. Note the results of the Noisy Student marked as NoisyStudentEfficientNet-B7. One key factor is that the datasets were balanced across different classes, which improved training, especially for smaller models. These results show that knowledge distillation isn’t just limited to creating smaller models like DistilBERT, but can also be used to increase the robustness of an already great model, using noisy student training.

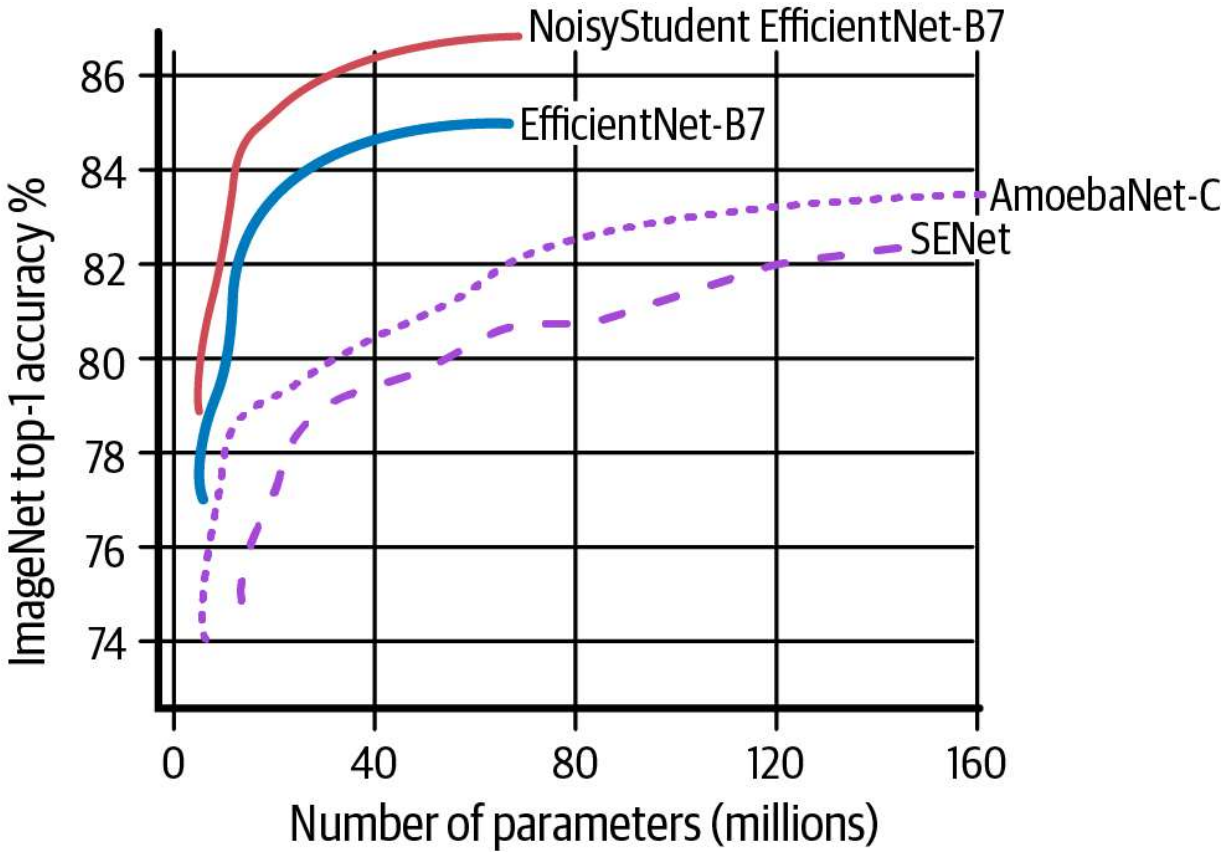

Figure 6-10. Noisy student accuracy (source: Xie et al., 2020)

As we've seen in this discussion, knowledge distillation is an important technique that you can use to make your models more efficient. The teacher-and-student approach is the most common way to use distillation, and we looked at some examples of how that can improve model efficiency.

Conclusion

The compute, storage, and I/O systems that your model requires will determine how much it will cost to put your model into production and maintain it during its entire lifetime. This chapter discussed some important

techniques that can help us manage model resource requirements, including reducing the dimensionality of our dataset, quantizing and pruning our models, and using knowledge distillation to train a smaller model with the knowledge captured in a larger model.

The approaches we discussed in this chapter were specific to ML, but we should also keep in mind that there are many ways to improve the efficiency and reduce the cost of any production software deployment. These include writing and deploying more efficient and scalable code for the various components of the production systems, and implementing more efficient infrastructure and scaling designs. Since this book is primarily focused on ML and not on software or systems engineering, we won't be discussing them here, but that doesn't mean you should ignore them. Always remember that a production ML system is still a production software and hardware system, so everything in those disciplines still applies.