images in the batch that we have to mix up. In the function we will also choose a weighting factor, `wt`, of between 0.1 and 0.4 to interpolate the pair of images.

To interpolate, we need pairs of images. The first set of images will be the first `fracn` images in the batch:

```
img1, label1 = img[:fracn], label[:fracn]
```

How about the second image in each pair? We'll do something quite simple: we'll pick the next image, so that the first image gets interpolated with the second, the second with the third, and so on. Now that we have the pairs of images/labels, interpolating can be done as follows:

```
def _interpolate(b1, b2, wt):
    return wt*b1 + (1-wt)*b2
interp_img = _interpolate(img1, img2, wt)
interp_label = _interpolate(label1, label2, wt)
```

The results are shown in Figure 6-13. The top row is the original batch of five images. The bottom row is the result of mixup: 40% of 5 is 2, so the first two images are the ones that are mixed up, and the last three images are left as-is. The first mixed-up image is obtained by interpolating the first and second original images, with a weight of 0.63 to the first and 0.37 to the second. The second mixed-up image is obtained by mixing up the second and third images from the top row. Note that the labels (the array above each image) show the impact of the mixup as well.

```
img2, label2 = img[1:fracn+1], label[1:fracn+1] # offset by one
```

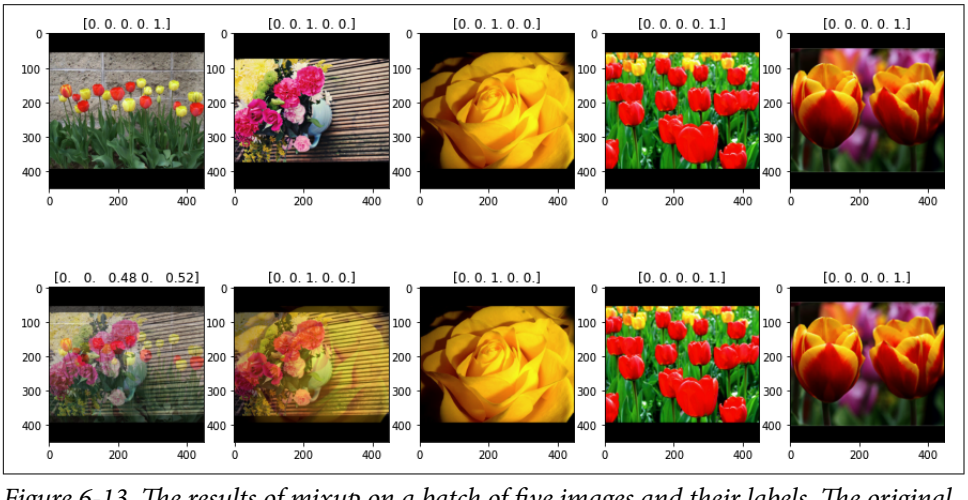

Figure 6-13. The results of mixup on a batch of five images and their labels. The original images are in the top row, and the first two images (40% of the batch) in the bottom row are the ones that are mixed up.

At this point, we have `fracn` interpolated images built from the first `fracn+1` images (we need `fracn+1` images to get `fracn` pairs, since the `fracn`th image is interpolated with the `fracn+1`th one). We then stack the interpolated images and the remaining unaltered images to get back a `batch_size` of images:

```
img = tf.concat([interp_img, img[fracn:]], axis=0)
label = tf.concat([interp_label, label[fracn:]], axis=0)
```

The `augment_mixup()` method can be passed into the `tf.data` pipeline that is used to create the training dataset:

```
train_dataset = create_preproc_dataset(...) \
    .shuffle(8 * batch_size) \
    .batch(batch_size, drop_remainder=True) \
    .map(augment_mixup)
```

There are a couple of things to notice in this code. First, we have added a `shuffle()` step to ensure that the batches are different in each epoch (otherwise, we won't get any variety in our mixup). We ask `tf.data` to drop any leftover items in the last batch, because computation of the parameter `n` could run into problems on very small batches. Because of the shuffle, we'll be dropping different items each time, so we're not too bothered about this.

`shuffle()` works by reading records into a buffer, shuffling the records in the buffer, and then providing the records to the next step of the data pipeline. Because we want the records in a batch to be different during each epoch, we will need the size of the shuffle buffer to be much larger than the batch size—shuffling the records within a batch won't suffice. Hence, we use:

```
.shuffle(8 * batch_size)
```

Interpolating labels is not possible if we keep the labels as sparse integers (e.g., 4 for tulips). Instead, we have to one-hot encode the labels (see Figure 6-13). Therefore, we make two changes to our training program. First, our `read_from_tfr()` method does the one-hot encoding instead of simply returning `label_int`:

```
def read_from_tfr(self, proto):
    ...
    rec = tf.io.parse_single_example(
        proto, feature_description
    )
    shape = tf.sparse.to_dense(rec['shape'])
    img = tf.reshape(tf.sparse.to_dense(rec['image']), shape)
    label_int = rec['label_int']
    return img, tf.one_hot(label_int, len(CLASS_NAMES))
```

Second, we change the loss function from `SparseCategoricalCrossentropy()` to `CategoricalCrossentropy()` since the labels are now one-hot encoded:

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lrate),
              loss=tf.keras.losses.CategoricalCrossentropy(
                  from_logits=False),
              metrics=['accuracy'])
```

On the 5-flowers dataset, mixup doesn't improve the performance of the model—we got the same accuracy (0.88, see Figure 6-14) with mixup as without it. However, it might help in other situations. Recall that information dropping helps the model learn to disregard uninformative parts of the image and mixup works by linearly interpolating pairs of training images. So, information dropping via mixup would work well in situations where only a small section of the image is informative, and where the pixel intensity is informative—think, for example, of remotely sensed imagery where we are trying to identify deforested patches of land.

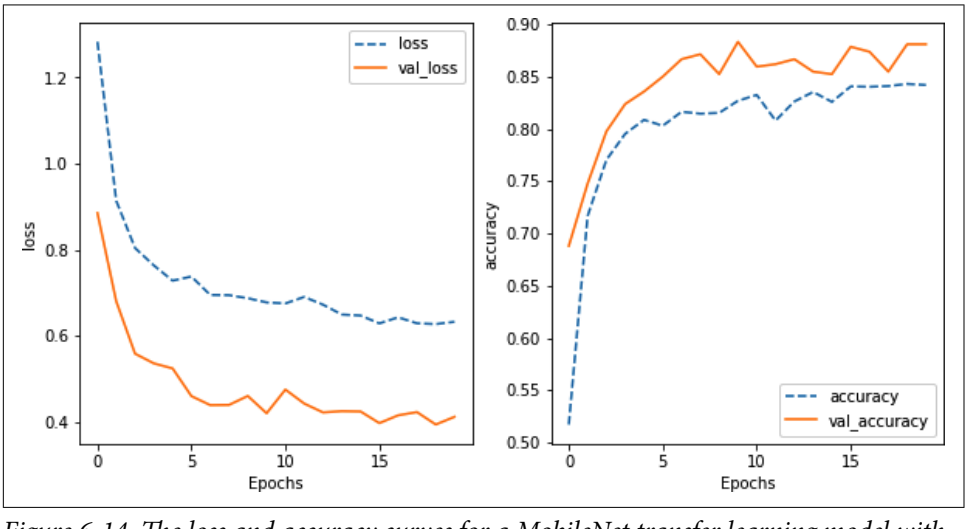

Figure 6-14. The loss and accuracy curves for a MobileNet transfer learning model with mixup. Compare to Figure 6-12.

Interestingly, the validation accuracy and loss are now better than the training accuracy. This is logical when we recognize that the training dataset is "harder" than the validation dataset—there are no mixed-up images in the validation set.

Forming Input Images

The preprocessing operations we have looked at so far are one-to-one, in that they simply modify the input image and provide a single image to the model for every image that is input. This is not necessary, however. Sometimes, it can be helpful to use the preprocessing pipeline to break down each input into multiple images that are then fed to the model for training and inference (see Figure 6-15).

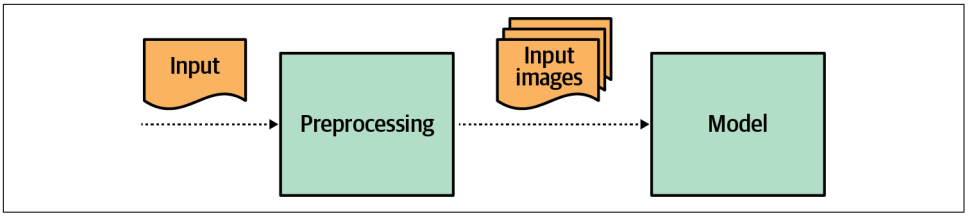

Figure 6-15. Breaking down a single input into component images that are used to train the model. The operation used to break an input into its component images during training also has to be repeated during inference.

One method of forming the images that are input to a model is *tiling*. Tiling is useful in any field where we have extremely large images and where predictions can be carried out on parts of the large image and then assembled. This tends to be the case for geospatial imagery (identifying deforested areas), medical images (identifying cancerous tissue), and surveillance (identifying liquid spills on a factory floor).

Imagine that we have a remotely sensed image of the Earth and would like to identify forest fires (see Figure 6-16). To do this, a machine learning model would have to predict whether an individual pixel contains a forest fire or not. The input to such a model would be a *tile*, the part of the original image immediately surrounding the pixel to be predicted. We can preprocess geospatial images to yield equal-sized tiles that are used to train ML models and obtain predictions from them.

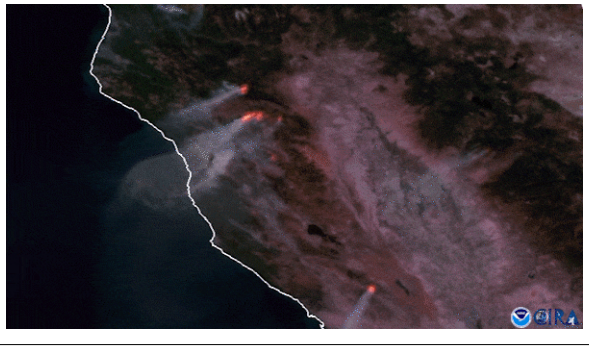

Figure 6-16. Remotely sensed image of wildfires in California. Image courtesy of NOAA.

For each of the tiles, we'll need a label that signifies whether or not there is fire within the tile. To create these labels, we can take fire locations called in by fire lookout towers and map them to an image the size of the remotely sensed image (the full code is in *06g_tiling.ipynb* on GitHub):

```
fire_label = np.zeros((338, 600))
for loc in fire_locations:
    fire_label[loc[0]][loc[1]] = 1.0
```

To generate the tiles, we will extract patches of the desired tile size and stride forward by half the tile height and width (so that the tiles overlap):

```
tiles = tf.image.extract_patches(
    images=images,
    sizes=[1, TILE_HT, TILE_WD, 1],
    strides=[1, TILE_HT//2, TILE_WD//2, 1],
    rates=[1, 1, 1, 1],
    padding='VALID')
```

The result, after a few reshaping operations, is shown in Figure 6-17. In this figure, we are also annotating each tile by its label. The label for an image tile is obtained by looking for the maximum value within the corresponding label tile (this will be 1.0 if the tile contains a `fire_location` point):

```
labels = tile_image(labels)
labels = tf.reduce_max(labels, axis=[1, 2, 3])
```

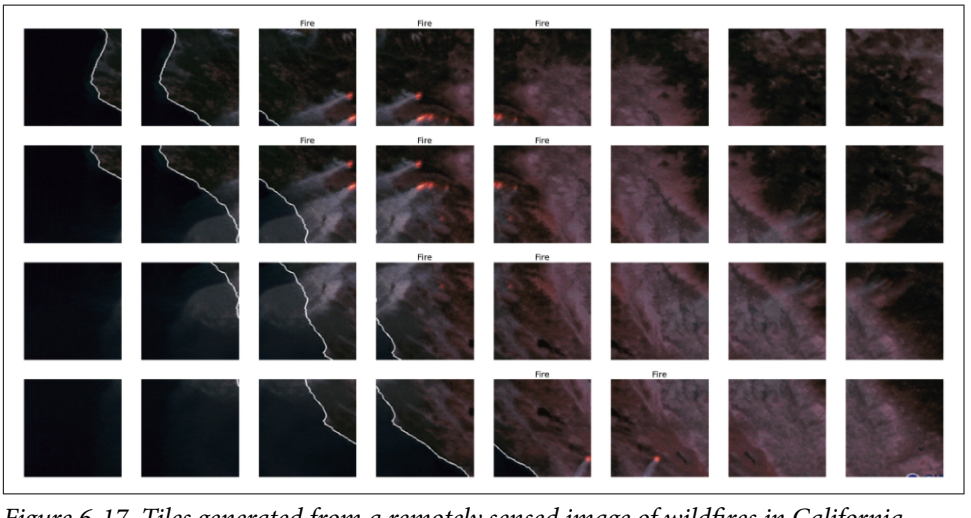

Figure 6-17. Tiles generated from a remotely sensed image of wildfires in California. Tiles with fire are labeled "Fire." Image courtesy of NOAA.

These tiles and their labels can now be used to train an image classification model. By reducing the stride by which we generate tiles, we can augment the training dataset.

Summary

In this chapter, we looked at various reasons why the preprocessing of images is needed. It could be to reformat and reshape the input data into the data type and shape required by the model, or to improve the data quality by carrying out operations such as scaling and clipping. Another reason to do preprocessing is to perform data augmentation, which is a set of techniques to increase the accuracy and

resilience of a model by generating new training examples from the existing training dataset. We also looked at how to implement each of these types of preprocessing, both as Keras layers and by wrapping TensorFlow operations into Keras layers.

In the next chapter, we will delve into the training loop itself.

CHAPTER 7
Training Pipeline

The stage after preprocessing is model training, during which the machine learning model will read in the training data and use that data to adjust its weights (see Figure 7-1). After training, the model is saved or exported so that it can be deployed.

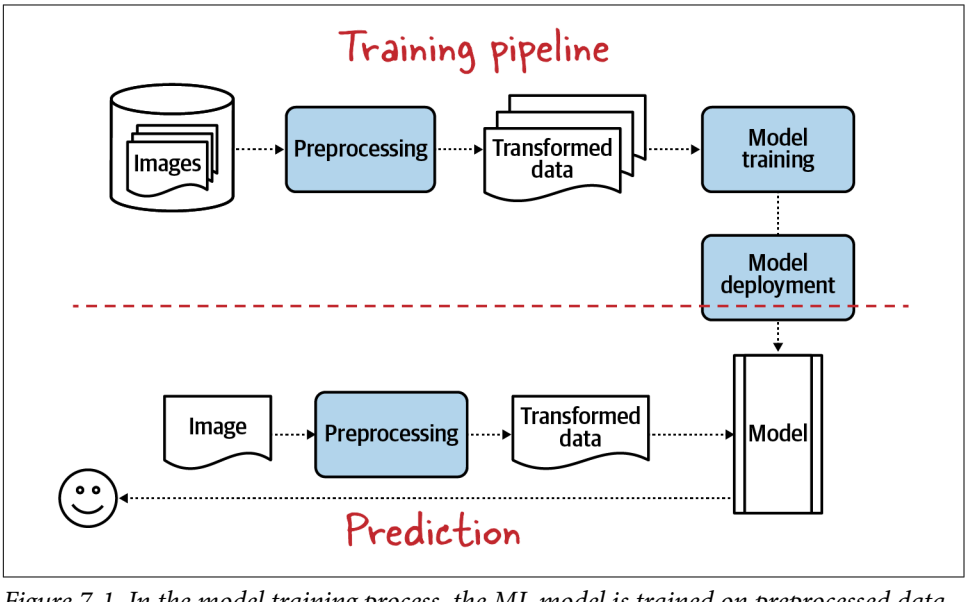

Figure 7-1. In the model training process, the ML model is trained on preprocessed data and then exported for deployment. The exported model is used to make predictions.

In this chapter, we will look at ways to make the ingestion of training (and validation) data into the model more efficient. We will take advantage of time slicing between the different computational devices (CPUs and GPUs) available to us, and examine how to make the whole process more resilient and reproducible.

The code for this chapter is in the *07_training* folder of the book's GitHub repository. We will provide file names for code samples and notebooks where applicable.

Efficient Ingestion

A significant part of the time it takes to train machine learning models is spent on ingesting data—reading it and transforming it into a form that is usable by the model. The more we can do to streamline and speed up this stage of the training pipeline, the more efficient we can be. We can do this by:

Storing data efficiently
: We should preprocess the input images as much as possible, and store the preprocessed values in a way that is efficient to read.

Parallelizing the reading of data
: When ingesting data, the speed of storage devices tends to be a bottleneck. Different files may be stored on different disks, or can be read via different network connections, so it is often possible to parallelize the reading of data.

Preparing images in parallel with training
: If we can preprocess the images on the CPU in parallel with training on the GPU, we should do so.

Maximizing GPU utilization
: As much as possible, we should try to carry out matrix and mathematical operations on the GPU, since it is many orders of magnitude faster than a CPU. If any of our preprocessing operations involve these operations, we should push them to the GPU.

Let's look at each of these ideas in more detail.

Storing Data Efficiently

Storing images as individual JPEG files is not very efficient from a machine learning perspective. In Chapter 5, we discussed how to convert JPEG images into TensorFlow Records. In this section, we will explain why TFRecords are an efficient storage mechanism, and consider trade-offs between flexibility and efficiency in terms of the amount of preprocessing that is carried out before the data is written out.

TensorFlow Records

Why store images as TensorFlow Records? Let's consider what we're looking for in a file format.

We know that we are going to be reading these images in batches, so it will be best if we can read an entire batch of images using a single network connection rather than open up one connection per file. Reading a batch all at once will also provide greater throughput to our machine learning pipeline and minimize the amount of time the GPU is waiting for the next batch of images.

Ideally, we would like the files to be around 10–100 MB in size. This allows us to balance the ability to read the images from multiple workers (one for every GPU) and the need to have each file open long enough to amortize the latency of reading the first byte over many batches.

Also, we would like the file format to be such that bytes read from the file can be mapped immediately to an in-memory structure without the need to parse the file or handle storage layout differences (such as endianness) between different types of machines.

The file format that meets all these criteria is TensorFlow Records. We can store the image data for training, validation, and testing into separate TFRecord files, and shard the files at around 100 MB each. Apache Beam has a handy TFRecord writer that we used in Chapter 5.

Storing preprocessed data

We can improve the performance of our training pipeline if we don't have to do the preprocessing in the training loop. We might be able to carry out the desired preprocessing on the JPEG images and then write out the preprocessed data rather than the raw data.

In practice, we will have to split the preprocessing operations between the ETL pipeline that creates the TensorFlow Records and the model code itself. Why not do it all in the ETL pipeline or all in the model code? The reason is that preprocessing operations applied in the ETL pipeline are done only once instead of in each epoch of the model training. However, there will always be preprocessing operations that are specific to the model that we are training or that need to be different during each epoch. These cannot be done in the ETL pipeline—they must be done in the training code.

In Chapter 5, we decoded our JPEG files, scaled them to lie between [0, 1], flattened out the array, and wrote the flattened array out to TensorFlow Records:

```
def create_tfrecord(filename, label, label_int):
    img = tf.io.read_file(filename)
    img = tf.image.decode_jpeg(img, channels=IMG_CHANNELS)
    img = tf.image.convert_image_dtype(img, tf.float32)
    img = tf.reshape(img, [-1]) # flatten to 1D array
    return tf.train.Example(features=tf.train.Features(feature={
        'image': _float_feature(img),
        ...
    })).SerializeToString()
```

The operations that we did before writing the TensorFlow Records were chosen explicitly.

We could have done less if we'd wanted—we could have simply read the JPEG files and written out the contents of each file as a string into the TensorFlow Records:

```
def create_tfrecord(filename, label, label_int):
    img = tf.io.read_file(filename)
    return tf.train.Example(features=tf.train.Features(feature={
        'image': _bytes_feature(img),
        ...
    })).SerializeToString()
```

Had we been concerned about potentially different file formats (JPEG, PNG, etc.) or image formats not understood by TensorFlow, we could have decoded each image, converted the pixel values into a common format, and written out the compressed JPEG as a string.

We could also have done much more. For example, we could have created an embedding of the images and written out not the image data, but only the embeddings:

```
embedding_encoder = tf.keras.Sequential([
    hub.KerasLayer(
        "https://tfhub.dev/.../mobilenet_v2/...",
        trainable=False,
        input_shape=(256, 256, IMG_CHANNELS),
        name='mobilenet_embedding'),
])

def create_tfrecord(filename, label, label_int):
    img = tf.io.read_file(filename)
    img = tf.image.decode_jpeg(img, channels=IMG_CHANNELS)
    img = tf.image.convert_image_dtype(img, tf.float32)
    img = tf.resize(img, [256, 256, 3])
    embed = embedding_encoder(filename)
    embed = tf.reshape(embed, [-1]) # flatten to 1D array
    return tf.train.Example(features=tf.train.Features(feature={
        'image_embedding': _float_feature(embed),
        ...
    })).SerializeToString()
```

The choice of what operations to perform comes down to a trade-off between efficiency and reusability. It's also affected by what types of reusability we envision. Remember that ML model training is a highly iterative, experimental process. Each training experiment will iterate over the training dataset multiple times (specified by the number of epochs). Therefore, each TFRecord in the training dataset will have to be processed multiple times. The more of the processing we can carry out before writing the TFRecords, the less processing has to be carried out in the training pipeline itself. This will result in faster and more efficient training and higher data throughput. This advantage is multiplied manyfold because we normally do not just

train a model once; we run multiple experiments with multiple hyperparameters. On the other hand, we have to make sure that the preprocessing that we are carrying out is desirable for all the ML models that we want to train using this dataset—the more preprocessing we do, the less reusable our dataset might become. We should also not enter the realm of micro-optimizations that improve speed minimally but make the code much less clear or reusable.

If we write out the image embeddings (rather than the pixel values) to TensorFlow Records, the training pipeline will be hugely efficient since the embedding computation typically involves passing the image through one hundred or more neural network layers. The efficiency gains can be considerable. However, this presupposes that we will be doing transfer learning. We cannot train an image model from scratch using this dataset. Of course, storage being much less expensive than compute, we might also find that it is advantageous to create two datasets, one of the embeddings and the other of the pixel values.

Because TensorFlow Records can vary in terms of how much preprocessing has been carried out, it is a good practice to document this in the form of metadata. Explain what data is present in the records, and how that data was generated. General-purpose tools like Google Cloud Data Catalog, Collibra, and Informatica can help here, as can custom ML frameworks like the Feast feature store.

Reading Data in Parallel

Another way to improve the efficiency of ingesting data into the training pipeline is to read the records in parallel. In Chapter 6, we read the written-out TFRecords and preprocessed them using:

```
preproc = _Preprocessor()
trainds = tf.data.TFRecordDataset(pattern)
            .map(preproc.read_from_tfr)
            .map(_preproc_img_label)
```

In this code, we are doing three things:

1. Creating a `TFRecordDataset` from a pattern
2. Passing each record in the files to `read_from_tfr()`, which returns an (img, label) tuple
3. Preprocessing the tuples using `_preproc_img_label()`

Parallelizing

There are a couple of improvements that we can make to our code, assuming that we are running on a machine with more than one virtual CPU (most modern machines

have at least two vCPUs, often more). First, we can ask TensorFlow to automatically interleave reading when we create the dataset:

```
tf.data.TFRecordDataset(pattern, num_parallel_reads=AUTO)
```

Second, the two `map()` operations can be parallelized using:

```
.map(preproc.read_from_tfr, num_parallel_calls=AUTOTUNE)
```

Measuring performance

In order to measure the performance impact of these changes, we need to go through the dataset and carry out some mathematical operations. Let's compute the mean of all the images. To prevent TensorFlow from optimizing away any calculations (see the following sidebar), we'll compute the mean only of pixels that are above some random threshold that is different in each iteration of the loop:

```
def loop_through_dataset(ds, nepochs):
    lowest_mean = tf.constant(1.)
    for epoch in range(nepochs):
        thresh = np.random.uniform(0.3, 0.7)  # random threshold
        ...
        for (img, label) in ds:
            ...
            mean = tf.reduce_mean(tf.where(img > thresh, img, 0))
            ...
```

Measuring Performance Impacts

When carrying out measurements of performance, we have to make sure that the optimizer doesn't realize that our code is unnecessary to compute. For example, in the following code, the optimizer would realize that we are not using the mean at all and simply optimize away the calculations:

```
for iter in range(100):
    mean = tf.reduce_mean(img)
```

One way to prevent this is to use the mean somehow. For example, we can find the lowest mean found in the loop and make sure to print or return it:

```
    lowest_mean = tf.constant(1.)
for iter in range(100):
    mean = tf.reduce_mean(img)
    lowest_mean = mean if mean < lowest_mean
return(lowest_mean)
```

However, the optimizer is capable of recognizing that `img` doesn't change and that `reduce_mean()` will return the same value, so it will get moved out of the loop. That's why we are adding a random threshold to our code.

Table 7-1 shows the result of measuring the performance of the preceding loop when ingesting the first 10 TFRecord files using different mechanisms. It is clear that while the additional parallelization increases the overall CPU time, the actual wall-clock time reduces with each bout of parallelization. We get a 35% reduction in the time spent by making the maps parallel and by interleaving two datasets.

Table 7-1. Time taken to loop through a small dataset when the ingestion is done in different ways

Method	CPU time	Wall time
Plain	7.53 s	7.99 s
Parallel map	8.30 s	5.94 s
Interleave	8.60 s	5.47 s
Interleave + parallel map	8.44 s	5.23 s

Will this performance gain carry over to machine learning models? To test this, we can try training a simple linear classification model instead of using the `loop_through_dataset()` function:

```
def train_simple_model(ds, nepochs):
    model = tf.keras.Sequential([
        tf.keras.layers.Flatten(
            input_shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS)),
        tf.keras.layers.Dense(len(CLASS_NAMES), activation='softmax')
    ])
    model.compile(optimizer=tf.keras.optimizers.Adam(),
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(
                      from_logits=False),
                  metrics=['accuracy'])
    model.fit(ds, epochs=nepochs)
```

The result, shown in Table 7-2, illustrates that the performance gains do hold up—we get a 25% speedup between the first and last rows. As the complexity of the model increases, the I/O plays a smaller and smaller role in the overall timing, so it makes sense that the improvement is less.

Table 7-2. Time taken to train a linear ML model on a small dataset when the ingestion is done in different ways

Method	CPU time	Wall time
Plain	9.91 s	9.39 s
Parallel map	10.7 s	8.17 s
Interleave	10.5 s	7.54 s
Interleave + parallel map	10.3 s	7.17 s

Looping through a dataset is faster than training an actual ML model on the full training dataset. Use it as a lightweight way to exercise your ingestion code for the purposes of tuning the performance of the I/O part.

Maximizing GPU Utilization

Because GPUs are more efficient at doing machine learning model operations, our goal should be to maximize their utilization. If we rent GPUs by the hour (as we do in a public cloud), maximizing GPU utilization will allow us to take advantage of their increased efficiency to get an overall lower cost per training run than we would get if we were to train on CPUs.

There are three factors that will affect our model's performance:

1. Every time we move data between the CPU and the GPU, that transfer takes time.
2. GPUs are efficient at matrix math. The more operations we do on single items, the less we are taking advantage of the performance speedup offered by a GPU.
3. GPUs have limited memory.

These factors play a role in the optimizations that we can do to improve the performance of our training loop. In this section, we will look at three core ideas in maximizing GPU utilization: efficient data handling, vectorization, and staying in the graph.

Efficient data handling

When we are training our model on a GPU, the CPU will be idle while the GPU is calculating gradients and doing weight updates.

It can be helpful to give the CPU something to do—we can ask it to *prefetch* the data, so that the next batch of data is ready to pass to the GPU:

```
ds = create_preproc_dataset(
    'gs://practical-ml-vision-book/flowers_tfr/train' + PATTERN_SUFFIX
).prefetch(AUTOTUNE)
```

If we have a small dataset, especially one where the images or TensorFlow Records have to be read across a network, it can also be helpful to cache them locally:

```
ds = create_preproc_dataset(
    'gs://practical-ml-vision-book/flowers_tfr/train' + PATTERN_SUFFIX
).cache()
```

Table 7-3 shows the impact of prefetching and caching on how long it takes to train a model.

Table 7-3. Time taken to train a linear ML model on a small dataset when the input records are prefetched and/or cached

Method	CPU time	Wall time
Interleave + parallel	9.68 s	6.37 s
Cache	6.16 s	4.36 s
Prefetch + cache	5.76 s	4.04 s

In our experience, caching tends to work only for small (toy) datasets. For large datasets, you are likely to run out of local storage.

Vectorization

Because GPUs are good at matrix manipulation, we should attempt to give the GPU the maximum amount of data it can handle at one time. Instead of passing images one at a time, we should send in a batch of images—this is called *vectorization*.

To batch records, we can do:

```
ds = create_preproc_dataset(
    'gs://practical-ml-vision-book/flowers_tfr/train' + PATTERN_SUFFIX
).prefetch(AUTOTUNE).batch(32)
```

It's important to realize the entire Keras model operates on batches. Therefore, the `RandomFlip` and `RandomColorDistortion` preprocessing layers that we added do not process one image at a time; they process batches of images.

The larger the batch size is, the faster the training loop will be able to get through an epoch. There are diminishing returns, however, to increasing the batch size. Also, there is a limit imposed by the memory limit of the GPU. It's worth doing a cost–benefit analysis of using larger, more expensive machines with more GPU memory and training for a shorter period of time versus using smaller, less expensive machines and training for longer.

When training on Google's Vertex AI, GPU memory usage and utilization are automatically reported for every job. Azure allows you to configure containers for GPU monitoring. Amazon CloudWatch provides GPU monitoring on AWS. If you are managing your own infrastructure, use GPU tools like `nvidia-smi` or AMD System Monitor. You can use these to diagnose how effectively your GPUs are being used, and whether there is headroom in GPU memory to increase your batch size.

In Table 7-4, we show the impact of changing the batch size on a linear model. Larger batches are faster, but there are diminishing returns and we'll run out of on-board GPU memory beyond a certain point. The faster performance with increasing batch size is one of the reasons why TPUs, with their large on-board memory and interconnected cores that share the memory, are so cost-effective.

Table 7-4. Time taken to train a linear ML model at different batch sizes

Method	CPU time	Wall time
Batch size 1	11.4 s	8.09 s
Batch size 8	9.56 s	6.90 s
Batch size 16	9.90 s	6.70 s
Batch size 32	9.68 s	6.37 s

A key reason that we implemented the random flip, color distortion, and other preprocessing and data augmentation steps as Keras layers in Chapter 6 has to do with batching. We could have done the color distortion using a `map()` as follows:

```
trainds = tf.data.TFRecordDataset(
        [filename for filename in tf.io.gfile.glob(pattern)]
    ).map(preproc.read_from_tfr).map(_preproc_img_label
    ).map(color_distort).batch(32)
```

where `color_distort()` is:

```
def color_distort(image, label):
    contrast = np.random.uniform(0.5, 1.5)
    brightness = np.random.uniform(-0.2, 0.2)
    image = tf.image.adjust_contrast(image, contrast)
    image = tf.image.adjust_brightness(image, brightness)
    image = tf.clip_by_value(image, 0, 1)
    return image, label
```

But this would have been inefficient since the training pipeline would have to do color distortion one image at a time. It is much more efficient if we carry out preprocessing operations in Keras layers. This way, the preprocessing is done on the whole batch in one step. An alternative would be to vectorize the color distortion operation by writing the code as:

```
    ).batch(32).map(color_distort)
```

This would also cause the color distortion to happen on a batch of data. Best practice, however, is to write preprocessing code that follows the `batch()` operation in a Keras layer. There are two reasons for this. First, the separation between ingestion code and model code is cleaner and more maintainable if we consistently make the call to `batch()` a hard boundary. Second, keeping preprocessing in a Keras layer (see Chapter 6) makes it easier to reproduce the preprocessing functionality in the inference pipeline since all the model layers are automatically exported.

Staying in the graph

Because executing mathematical functions is much more efficient on a GPU than on a CPU, TensorFlow reads the data using the CPU, transfers the data to the GPU, then runs all our code that belongs to the `tf.data` pipeline (the code in the `map()` calls, for example) on the GPU. It also runs all the code in the Keras model layers on the GPU. Since we are sending the data directly from the `tf.data` pipeline to the Keras input layer, there is no need to transfer the data—the data stays within the TensorFlow graph. The data and model weights all remain in the GPU memory.

This means that we have to be extremely careful to make sure that we don't do anything that would involve moving the data out of the TensorFlow graph once the CPU has delivered the data to the GPU. Data transfers carry extra overhead, and any code executed on the CPU will tend to be slower.

Iteration. As an example, suppose we are reading a satellite image of California wildfires and wish to apply a specific formula based on photometry to the RGB pixel values to transform them into a single "grayscale" image (see Figure 7-2 and the full code in *07b_gpumax.ipynb* on GitHub):

```
def to_grayscale(img):
    rows, cols, _ = img.shape
    result = np.zeros([rows, cols], dtype=np.float32)
    for row in range(rows):
        for col in range(cols):
            red = img[row][col][0]
            green = img[row][col][1]
            blue = img[row][col][2]
            c_linear = 0.2126 * red + 0.7152 * green + 0.0722 * blue
            if c_linear > 0.0031308:
                result[row][col] = 1.055 * pow(c_linear, 1/2.4) - 0.055
            else:
                result[row][col] = 12.92 * c_linear
    return result
```

There are three problems with this function:

- It needs to iterate through the image pixels:

```
rows, cols, _ = img.shape
    for row in range(rows):
    for col in range(cols):
```

- It needs to read individual pixel values:

```
green = img[row][col][1]
```

- It needs to change output pixel values:

```
result[row][col] = 12.92 * c_linear
```

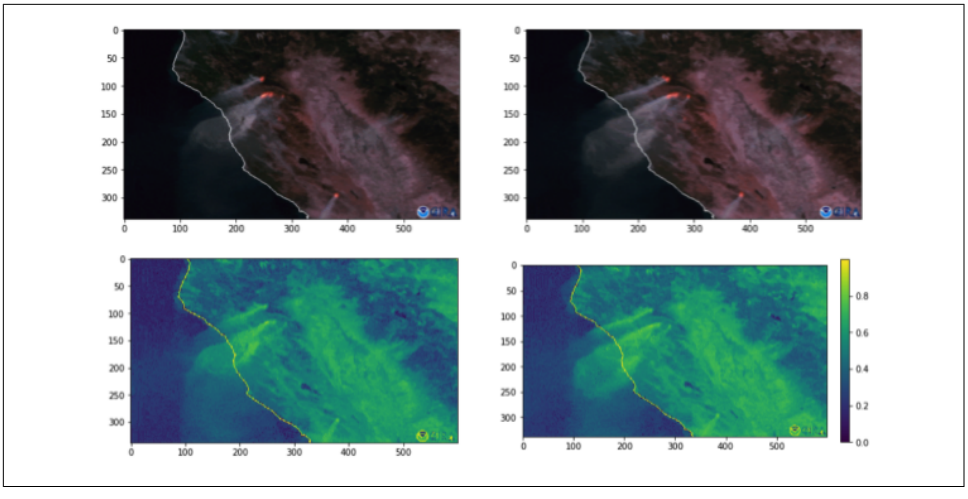

Figure 7-2. Top: original images with three channels. Bottom: transformed images with only one channel. Image of wildfires in California courtesy of NOAA.

These operations cannot be done in the TensorFlow graph. Therefore, to call the function, we need to bring it out of the graph using `.numpy()`, do the transformation, and then push the result back into the graph as a tensor (`gray` is converted into a tensor for the `reduce_mean()` operation).

Using tf.py_function() to Call Pure Python Code

There are times when we will have to invoke pure Python functionality from our TensorFlow programs. Maybe we have to do some time zone conversions and we need access to the `pytz` library, or maybe we get some data in JSON format and we need to invoke `json.loads()`.

In order to drop out of the TensorFlow graph, run a Python function, and get its result back into the TensorFlow graph, use `tf.py_function()`:

```
def to_grayscale(img):
    return tf.py_function(to_grayscale_numpy, [img],
                          tf.float32)
```

Now, `to_grayscale()` can be used as if it were implemented using TensorFlow operations:

```
ds = tf.data.TextLineDataset(...).map(to_grayscale)
```

There are three parameters to `py_function()`: the name of the function to wrap, the input tensor(s), and the output type. In our case, the wrapped function will be:

```
def to_grayscale_numpy(img):
    # The conversion from a tensor happens here.
    img = img.numpy()
    rows, cols, _ = img.shape
    result = np.zeros([rows, cols], dtype=np.float32)
    ...
    # The conversion back happens here.
    return tf.convert_to_tensor(result)
```

Note that we are still calling `numpy()` to bring the `img` tensor out of the graph, so that we can iterate through it, and we are taking the result (a `numpy` array) and converting it to a tensor so that it can be used in the remainder of the TensorFlow code.

Using `py_function()` is merely a way to call out to Python functions from TensorFlow. It does not do any optimizations or acceleration.

Slicing and conditionals. We can avoid the explicit iteration and pixel-wise read/write by using TensorFlow's slicing functionality:

```
def to_grayscale(img):
    # TensorFlow slicing functionality
    red = img[:, :, 0]
    green = img[:, :, 1]
    blue = img[:, :, 2]
    c_linear = 0.2126 * red + 0.7152 * green + 0.0722 * blue
```

Note that the last line of this code snippet is actually operating on tensors (`red` is a tensor, not a scalar) and uses operator overloading (the `+` is actually `tf.add()`) to invoke TensorFlow functions.

But how do we do the `if` statement in the original?

```
if c_linear > 0.0031308:
    result[row][col] = 1.055 * pow(c_linear, 1 / 2.4) - 0.055
else:
    result[row][col] = 12.92 * c_linear
```

The `if` statement assumes that `c_linear` is a single floating-point value, whereas now `c_linear` is a 2D tensor.

To push a conditional statement into the graph and avoid setting pixel values individually, we can use `tf.cond()` and/or `tf.where()`:

```
gray = tf.where(c_linear > 0.0031308,
                1.055 * tf.pow(c_linear, 1 / 2.4) - 0.055,
                12.92 * c_linear)
```

One key thing to realize is that all the three parameters to `tf.where()` in this example are actually 2D tensors. Note also the use of `tf.pow()` rather than `pow()`. Given the choice between `tf.cond()` and `tf.where()`, use `tf.where()` as it is faster.

This results in a more than 10x speedup.

Matrix math. The computation of `c_linear` can be optimized further. This is what we had:

```
red = img[:, :, 0]
green = img[:, :, 1]
blue = img[:, :, 2]
c_linear = 0.2126 * red + 0.7152 * green + 0.0722 * blue
```

If we look carefully at this calculation, we'll see that we don't need the slicing. Instead, we can write the computation as a matrix multiplication if we take the constants and put them into a 3x1 tensor:

```
def to_grayscale(img):
    wt = tf.constant([[0.2126], [0.7152], [0.0722]]) # 3x1 matrix
    c_linear = tf.matmul(img, wt)  # (ht,wd,3) x (3x1) -> (ht, wd)
    gray = tf.where(c_linear > 0.0031308,
                    1.055 * tf.pow(c_linear, 1 / 2.4) - 0.055,
                    12.92 * c_linear)
    return gray
```

With this optimization, we get an additional 4x speedup.

Batching. Once we have written the calculation of `c_linear` using matrix math, we also realize that we don't need to process the data one image at a time. We can process a batch of images all at once. We can do the calculations on a batch of images using either a custom Keras layer or a `Lambda` layer.

Let's wrap the grayscale calculation into the `call()` statement of a custom Keras layer:

```
class Grayscale(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super(Grayscale, self).__init__(kwargs)

    def call(self, img):
        wt = tf.constant([[0.2126], [0.7152], [0.0722]]) # 3x1 matrix
        c_linear = tf.matmul(img, wt)  #(N, ht,wd,3)x(3x1)->(N, ht, wd)
        gray = tf.where(c_linear > 0.0031308,
                        1.055 * tf.pow(c_linear, 1 / 2.4) - 0.055,
                        12.92 * c_linear)
        return gray # (N, ht, wd)
```

An important thing to note is that the input matrix is now a 4D tensor, with the first dimension being the batch size. The result is therefore a 3D tensor.

Clients calling this code can compute the mean of each image to get back a 1D tensor of means:

```
tf.keras.layers.Lambda(lambda gray: tf.reduce_mean(gray, axis=[1, 2]))
```

We can combine these two layers into a Keras model, or prepend them to an existing model:

```
preproc_model = tf.keras.Sequential([
    Grayscale(input_shape=(336, 600, 3)),
    tf.keras.layers.Lambda(lambda gray: tf.reduce_mean(
                               gray, axis=[1, 2]))  # note axis change
])
```

The timings of all the methods we discussed in this section are shown in Table 7-5.

Table 7-5. Time taken when the grayscale computation is carried out in different ways

Method	CPU time	Wall time
Iterate	39.6 s	41.1 s
Pyfunc	39.7 s	41.1 s
Slicing	4.44 s	3.07 s
Matmul	1.22 s	2.29 s
Batch	1.11 s	2.13 s

Saving Model State

So far in this book, we have been training a model and then using the trained model to immediately make a few predictions. This is highly unrealistic—we will want to train our model, and then keep the trained model around to continue making predictions with it. We will need to save the model's state so that we can quickly read in the trained model (its structure and its final weights) whenever we want.

We will want to save the model not just to predict from it, but also to resume training. Imagine that we have trained a model on one million images and are carrying out predictions with that model. If a month later we receive one thousand new images, it would be good to continue the training of the original model for a few steps with the new images instead of training from scratch. This is called fine-tuning (and was discussed in Chapter 3).

So, there are two reasons to save model state:

- To make inferences from the model
- To resume training

What these two use cases require are quite different. It's easiest to understand the difference between the two use cases if we consider the `RandomColorDistortion` data augmentation layer that is part of our model. For the purposes of inference, this layer can be removed completely. However, in order to resume training, we may need to know the full state of the layer (consider, for example, that we lower the amount of distortion the longer we train).

Saving the model for inference is called *exporting* the model. Saving the model in order to resume training is called *checkpointing*. Checkpoints are much larger in size than exports because they include a lot more internal state.

Exporting the Model

To export a trained Keras model, use the `save()` method:

```
os.mkdir('export')
model.save('export/flowers_model')
```

The output directory will contain a protobuf file called *saved_model.pb* (which is why this format is often referred to as the TensorFlow SavedModel format), the variable weights, and any assets such as vocabulary files that the model needs for prediction.

An alternative to SavedModel is Open Neural Network Exchange (ONNX), an open source, framework-agnostic ML model format that was introduced by Microsoft and Facebook. You can use the `tf2onnx` tool to convert a TensorFlow model to ONNX.

Invoking the model

We can interrogate the contents of a SavedModel using the command-line tool `saved_model_cli` that comes with TensorFlow:

```
saved_model_cli show --tag_set all --dir export/flowers_model
```

This shows us that the prediction signature (see the following sidebar) is:

```
inputs['random/center_crop_input'] tensor_info:
    dtype: DT_FLOAT
    shape: (-1, 448, 448, 3)
    name: serving_default_random/center_crop_input:0
```

The given SavedModel `SignatureDef` contains the following output(s):

```
outputs['flower_prob'] tensor_info:
    dtype: DT_FLOAT
    shape: (-1, 5)
    name: StatefulPartitionedCall:0
Method name is: tensorflow/serving/predict
```

Signature of a TensorFlow Function

The signature of a function is the name of the function, the parameters it takes, and what it returns. The typical Python function is written to be *polymorphic* (i.e., applicable to values of different types). For example, this will work when `a` and `b` are floats as well as when they are both strings:

```
def myfunc(a, b):
    return (a + b)
```

This is because Python is an interpreted (as opposed to a compiled) language—the code gets executed when it's called, and at runtime, the interpreter knows whether you are passing it floats or strings. Indeed, when we inspect the signature of this function using Python's reflection capability:

```
from inspect import signature
print(signature(myfunc).parameters)
print(signature(myfunc).return_annotation)
```

we get back simply:

```
OrderedDict([('a', <Parameter "a">), ('b', <Parameter "b">)])
<class 'inspect._empty'>
```

meaning the parameters can be of any type and the return type is unknown.

It is possible to provide type hints to Python3 by explicitly specifying the input and output types:

```
def myfunc(a: int, b: float) -> float:
    return (a + b)
```

Note that these type hints are not checked by the runtime—it is still possible to pass strings to this function. The type hints are meant for use by code editors and linting tools. However, Python's reflection capability does read the type hints and tell us more details about the signature:

```
OrderedDict([('a', <Parameter "a: int">), ('b', <Parameter "b: float">)])
<class 'float'>
```

While this is nice, type hints are not sufficient for TensorFlow programs. We also need to specify the shapes of tensors. We do that by adding an annotation to our function:

```
@tf.function(input_signature=[
    tf.TensorSpec([3,5], name='a'),
    tf.TensorSpec([5,8], name='b')
])
def myfunc(a, b):
    return (tf.matmul(a,b))
```

The `@tf.function` annotation *auto-graphs* the function by walking through it and figures out the shape and type of the output tensor. We can examine the information that TensorFlow now has about the signature by calling `get_concrete_function()` and passing in an eager tensor (a tensor that is immediately evaluated):

```
print(myfunc.get_concrete_function(tf.ones((3,5)), tf.ones((5,8))))
```

This results in:

```
ConcreteFunction myfunc(a, b)
    Args:
```

```
            a: float32 Tensor, shape=(3, 5)
            b: float32 Tensor, shape=(5, 8)
        Returns:
            float32 Tensor, shape=(3, 8)
```

Note that the full signature includes the name of the function (`myfunc`), the parameters (`a, b`), the parameter types (`float32`), the parameter shapes (`(3, 5)` and `(5, 8)`), and the output tensor's type and shape.

Therefore, to invoke this model we can load it and call the `predict()` method, passing in a 4D tensor with the shape `[num_examples, 448, 448, 3]`, where num_examples is the number of examples we want to predict on at once:

```
serving_model = tf.keras.models.load_model('export/flowers_model')
img = create_preproc_image('../dandelion/9818247_e2eac18894.jpg')
batch_image = tf.reshape(img, [1, IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS])
batch_pred = serving_model.predict(batch_image)
```

The result is a 2D tensor with the shape [num_examples, 5] which represents the probability for each type of flower. We can look for the maximum of these probabilities to obtain the prediction:

```
pred = batch_pred[0]
pred_label_index = tf.math.argmax(pred).numpy()
pred_label = CLASS_NAMES[pred_label_index]
prob = pred[pred_label_index]
```

All this is still highly unrealistic, however. Do we really expect that a client who needs the prediction for an image will know enough to do the `reshape()`, `argmax()`, and so on? We need to provide a much simpler signature for our model to be usable.

Usable signature

A more usable signature for our model is one that doesn't expose all the internal details of the training (such as the size of the images the model was trained on).

What kind of signature would be easiest for a client to use? Instead of asking them to send us a tensor with the image contents, we can simply ask them for a JPEG file. And instead of returning a tensor of logits, we can send back easy-to-understand information extracted from the logits (the full code is in *07c_export.ipynb* on GitHub):

```
@tf.function(input_signature=[tf.TensorSpec([None,], dtype=tf.string)])
def predict_flower_type(filenames):
    ...
    return {
        'probability': top_prob,
        'flower_type_int': pred_label_index,
        'flower_type_str': pred_label
    }
```

Note that while we are at it, we might as well make the function more efficient—we can take a batch of filenames and do the predictions for all the images at once. Vectorizing brings efficiency gains at prediction time as well, not just during training!

Given a list of filenames, we can get the input images using:

```
input_images = [create_preproc_image(f) for f in filenames]
```

However, this involves iterating through the list of filenames and moving data back and forth from accelerated TensorFlow code to unaccelerated Python code. If we have a tensor of filenames, we can achieve the effect of iteration while keeping all the data in the TensorFlow graph by using `tf.map_fn()`. With that, our prediction function becomes:

```
input_images = tf.map_fn(
    create_preproc_image,
    filenames,
    fn_output_signature=tf.float32
)
```

Next, we invoke the model to get the full probability matrix:

```
batch_pred = model(input_images)
```

We then find the maximum probability and the index of the maximum probability:

```
top_prob = tf.math.reduce_max(batch_pred, axis=1)
pred_label_index = tf.math.argmax(batch_pred, axis=1)
```

Note that we are being careful to specify the `axis` as 1 (`axis=0` is the batch dimension) when finding the maximum probability and the argmax. Finally, where in Python we could simply do:

```
pred_label = CLASS_NAMES[pred_label_index]
```

the TensorFlow in-graph version is to use `tf.gather()`:

```
pred_label = tf.gather(params=tf.convert_to_tensor(CLASS_NAMES),
                       indices=pred_label_index)
```

This code converts the `CLASS_NAMES` array into a tensor and then indexes into it using the `pred_label_index` tensor. The resulting values are stored in the `pred_label` tensor.

You can often replace Python iterations by `tf.map_fn()` and dereference arrays (read the *n*th element of an array) by using `tf.gather()`, as we have done here. Slicing using the [:, :, 0] syntax is very useful as well. The difference between `tf.gather()` and slicing is that `tf.gather()` can take a tensor as the index, whereas slices are constants. In really complex situations, `tf.dynamic_stitch()` can come in handy.

Using the signature

With the signature defined, we can specify our new signature as the serving default:

```
model.save('export/flowers_model',
           signatures={
               'serving_default': predict_flower_type
           })
```

Note that the API allows us to have multiple signatures in the model—this is useful if we want to add versioning to our signature, or support different signatures for different clients. We will explore this further in Chapter 9.

With the model exported, the client code to do a prediction now becomes simplicity itself:

```
serving_fn = tf.keras.models.load_model('export/flowers_model'
                                        ).signatures['serving_default']
filenames = [
    'gs://.../9818247_e2eac18894.jpg',
     ...
    'gs://.../8713397358_0505cc0176_n.jpg'
]
pred = serving_fn(tf.convert_to_tensor(filenames))
```

The result is a dictionary and can be used as follows:

```
print(pred['flower_type_str'].numpy().decode('utf-8'))
```

A few input images and their predictions are shown in Figure 7-3. The point to note is that the images are all different sizes. The client doesn't need to know any of the internal details of the model in order to invoke it. It's also worth noting that the "string" type in TensorFlow is only an array of bytes. We have to pass these bytes into a UTF-8 decoder to get proper strings.

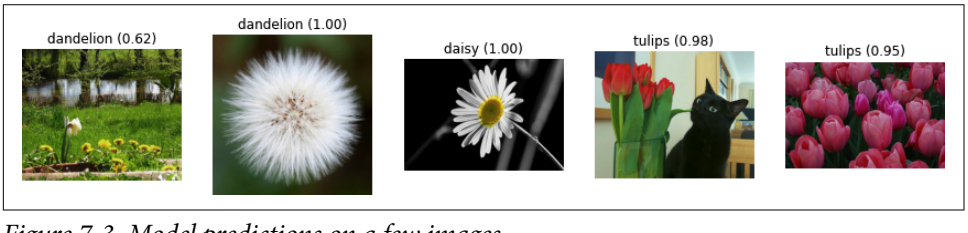

Figure 7-3. Model predictions on a few images.

Checkpointing

So far, we have focused on how to export the model for inference. Now, let's look at how to save the model in order to resume training. Checkpointing is typically done not only at the end of training, but also in the middle of training. There are two reasons for this:

- It might be helpful to go back and select the model at the point where the validation accuracy is highest. Recall that the training loss keeps decreasing the longer we train, but at some epoch, the validation loss starts to rise because of overfitting. When we observe that, we have to pick the checkpoint of the previous epoch because it had the lowest validation error.
- Machine learning on production datasets can take several hours to several days. The chances that a machine will crash during such a long time period are uncomfortably high. Therefore, it's a good idea to have periodic backups so that we can resume training from an intermediate point rather than starting from scratch.

Checkpointing is implemented in Keras by means of *callbacks*—functionality that is invoked during the training loop by virtue of being passed in as a parameter to the `model.fit()` function:

```
model_checkpoint_cb = tf.keras.callbacks.ModelCheckpoint(
    filepath='./chkpts',
    monitor='val_accuracy', mode='max',
    save_best_only=True)
history = model.fit(train_dataset,
                    validation_data=eval_dataset,
                    epochs=NUM_EPOCHS,
                    callbacks=[model_checkpoint_cb])
```

Here, we are setting up the callback to overwrite a previous checkpoint if the current validation accuracy is higher.

While we are doing this, we might as well set up early stopping—even if we initially start out thinking that we need to train for 20 epochs, we can stop the training once the validation error hasn't improved for 2 consecutive epochs (specified by the `patience` parameter):

```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(
    monitor='val_accuracy', mode='max',
    patience=2)
```

The callbacks list now becomes:

```
callbacks=[model_checkpoint_cb, early_stopping_cb]
```

When we train using these callbacks, training stops after eight epochs, as shown in Figure 7-4.

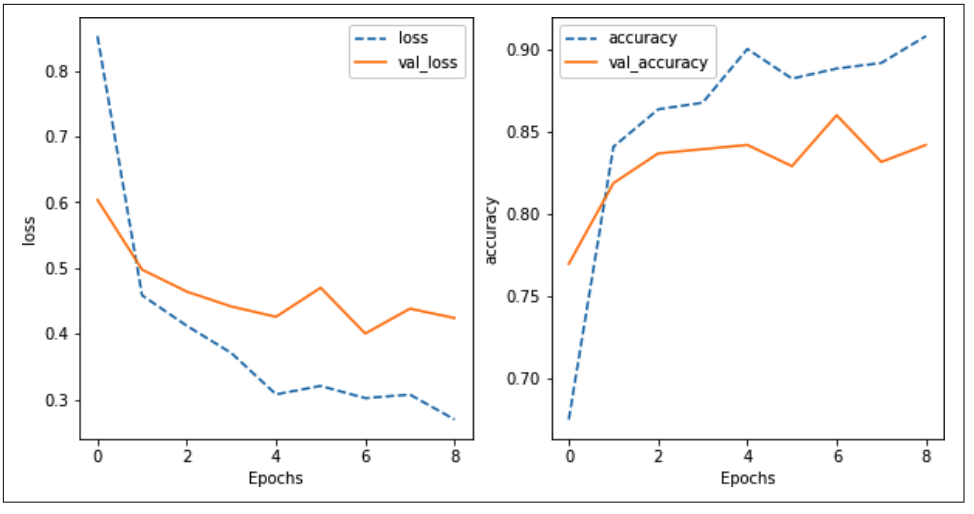

Figure 7-4. With early stopping, model training stops once the validation accuracy no longer increases.

To start from the last checkpoint in the output directory, call:

```
model.load_weights(checkpoint_path)
```

Full fault resilience is provided by the `BackupAndRestore` callback which, at the time of writing, was experimental.

Distribution Strategy

To distribute processing among multiple threads, accelerators, or machines, we need to parallelize it. We have looked at how to parallelize the ingestion. However, our Keras model is not parallelized; it runs on only one processor. How do we run our model code on multiple processors?

To distribute the model training, we need to set up a *distribution strategy*. There are several available, but they're all used in a similar way—you first create a strategy using its constructor, then create the Keras model within the scope of that strategy (here, we're using `MirroredStrategy`):

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    layers = [
        ...
    ]
    model = tf.keras.Sequential(layers)
model.compile(...)
history = model.fit(...)
```

What is `MirroredStrategy`? What other strategies are available, and how do we choose between them? We will answer each of these questions in the next sections.

What device does the code run on? All TensorFlow instructions that create trainable variables (such as Keras models or layers) must be created within the `strategy.scope()`, with the exception of `model.compile()`. You can call the `compile()` method wherever you want. Even though this method technically creates variables such as optimizer slots, it has been implemented to use the same strategy as the model. Also, you can create your ingestion (`tf.data`) pipeline wherever you want. It will always run on the CPU and it will always distribute data to workers appropriately.

Choosing a Strategy

Image ML models tend to be deep, and the input data is dense. For such models, there are three contending distribution strategies:

`MirroredStrategy`
: Makes mirrors of the model structure on each of the available GPUs. Each weight in the model is mirrored across all the replicas and kept in sync through identical updates that happen at the end of each batch. Use `MirroredStrategy` whenever you have a single machine, whether that machine has one GPU or multiple GPUs. This way, your code will require no changes when you attach a second GPU.

`MultiWorkerMirroredStrategy`
: Extends the `MirroredStrategy` idea to GPUs spread across multiple machines. In order to get the multiple workers communicating, you need to set up the `TF_CONFIG` variable correctly—we recommend using a public cloud service (such as Vertex Training) where this is automatically done for you.

`TPUStrategy`
: Runs the training job on TPUs, which are specialized application-specific integrated chips (ASICs) that are custom-designed for machine learning workloads. TPUs get their speedup through a custom matrix multiplication unit, high-speed on-board networking to connect up to thousands of TPU cores, and a large shared memory. They are available commercially only on Google Cloud Platform. Colab offers free TPUs with some limitations, and Google Research provides academic researchers access to TPUs through the TensorFlow Research Cloud program.

All three of these strategies are forms of *data parallelism*, where each batch is split among the workers, and then an all-reduce operation is carried out. Other available

distribution strategies, like `CentralStorage` and `ParameterServer`, are designed for sparse/massive examples and are not a good fit for image models where an individual image is dense and small.

We recommend maximizing the number of GPUs on a single machine with `MirroredStrategy` before moving on to multiple workers with `MultiWorkerMirroredStrategy` (more on this in the following section). TPUs are usually more cost-effective than GPUs, especially when you move to larger batch sizes. The current trend in GPUs (such as with the 16xA100) is to provide multiple powerful GPUs on a single machine so as to make this strategy work for more and more models.

Creating the Strategy

In this section, we will cover the specifics of the three strategies commonly used to distribute the training of image models.

MirroredStrategy

To create a `MirroredStrategy` instance, we can simply call its constructor (the full code is in *07d_distribute.ipynb* on GitHub):

```
def create_strategy():
    return tf.distribute.MirroredStrategy()
```

To verify whether we are running on a machine with GPUs set up, we can use:

```
if (tf.test.is_built_with_cuda() and
    len(tf.config.experimental.list_physical_devices("GPU")) > 1)
```

This is not a requirement; `MirroredStrategy` will work on a machine with only CPUs.

Starting a Jupyter notebook on a machine with two GPUs and using `MirroredStrategy`, we see an immediate speedup. Where an epoch took about 100 s to process on a CPU, and 55 s on a single GPU, it takes only 29 s when we have two GPUs.

When training in a distributed manner, you must make sure to increase the batch size. This is because a batch is split between the GPUs, so if a single GPU has the resources to process a batch size of 32, two GPUs will be able to easily handle 64. Here, 64 is the global batch size, and each of the two GPUs will have a local batch size of 32. Larger batch sizes are typically associated with better behaved training curves. We will experiment with different batch sizes in "Hyperparameter tuning" on page 42.

Sometimes, it is helpful for consistency and for debugging purposes to have a strategy even if you are not distributing the training code or using GPUs. In such cases, use `OneDeviceStrategy`:

```
tf.distribute.OneDeviceStrategy('/cpu:0')
```

MultiWorkerMirroredStrategy

To create a `MultiWorkerMirroredStrategy` instance, we can again simply call its constructor:

```
def create_strategy():
    return tf.distribute.MultiWorkerMirroredStrategy()
```

To verify that the `TF_CONFIG` environment variable is set up correctly, we can use:

```
tf_config = json.loads(os.environ["TF_CONFIG"])
```

and check the resulting config.

If we use a managed ML training system like Google's Vertex AI or Amazon SageMaker, these infrastructure details will be taken care of for us.

When using multiple workers, there are two details that we need to take care of: shuffling and virtual epochs.

Shuffling. When all the devices (CPUs, GPUs) are on the same machine, each batch of training examples is split among the different device workers and the resulting gradient updates are made *synchronously*—each device worker returns its gradient, the gradients are averaged across the device workers, and the computed weight update is sent back to the device workers for the next step.

When the devices are spread among multiple machines, having the central loop wait for all the workers on every machine to finish with a batch will lead to significant wastage of compute resources, as all the workers will have to wait for the slowest one. Instead, the idea is to have workers process data in parallel and for gradient updates to be averaged if they are available—a late-arriving gradient update is simply dropped from the calculation. Each worker receives the weight update that is current as of this time.

When we apply gradient updates *asynchronously* like this, we cannot split a batch across the different workers because then our batches would be incomplete, and our model will want equal-sized batches. So, we will have to have each worker reading full batches of data, computing the gradient, and sending in a gradient update for each full batch. If we do that, there is no use having all the workers reading the same data—we want every worker's batches to contain different examples. By shuffling the dataset, we can ensure that the workers are all working on different training examples at any point in time.

Even if we are not doing distributed training, it's a good idea to randomize the order in which the data is read by the `tf.data` pipeline. This will help reduce the chances that, say, one batch contains all daisies and the next batch contains all tulips. Such bad batches can play havoc with the gradient descent optimizer.

We can randomize the data that is read in two places:

- When we obtain the files that match the pattern, we shuffle these files:

```
files = [filename for filename
    # shuffle so that workers see different orders
    in tf.random.shuffle(tf.io.gfile.glob(pattern))
]
```

- After we preprocess the data, and just before we batch, we shuffle the records within a buffer that is larger than the batch size:

```
trainds = (trainds
    .shuffle(8 * batch_size)  # Shuffle for distribution ...
    .map(preproc.read_from_tfr, num_parallel_calls=AUTOTUNE)
    .map(_preproc_img_label, num_parallel_calls=AUTOTUNE)
    .prefetch(AUTOTUNE)
)
```

- The more ordered your dataset is, the larger your shuffle buffer needs to be. If your dataset is initially sorted by label, only a buffer size covering the entire dataset will work. In that case, it's better to shuffle the data ahead of time, when preparing the training dataset.

Virtual epochs. We often wish to train for a fixed number of training examples, not a fixed number of epochs. Since the number of training steps in an epoch depends on the batch size, it is easier to key off the total number of training examples in the dataset and compute what the number of steps per epoch ought to be:

```
num_steps_per_epoch = None
if (num_training_examples > 0):
    num_steps_per_epoch = (num_training_examples // batch_size)
```

We call a training cycle consisting of this number of steps a *virtual epoch* and train for the same number of epochs as before.

We specify the number of steps per virtual epoch as a parameter to `model.fit()`:

```
history = model.fit(train_dataset,
                    validation_data=eval_dataset,
                    epochs=num_epochs,
                    steps_per_epoch=num_steps_per_epoch)                    )
```

What if we get the number of training examples in the dataset wrong? Suppose we specify the number as 4,000, but there are actually 3,500 examples? We will have a problem, because the dataset will finish before 4,000 examples are encountered. We

can prevent that from happening by making the training dataset repeat upon itself indefinitely:

```
if (num_training_examples > 0):
    train_dataset = train_dataset.repeat()
```

This also works when we underestimate the number of training examples in the dataset—the next set of examples simply carry over to the next epoch. Keras knows that when a dataset is infinite, it should use the number of steps per epoch to decide when the next epoch starts.

TPUStrategy

While `MirroredStrategy` is meant for one or more GPUs on a single machine, and `MultiWorkerMirroredStrategy` is meant for GPUs on multiple machines, `TPUStrategy` allows us to distribute to a custom ASIC chip called the TPU, shown in Figure 7-5.

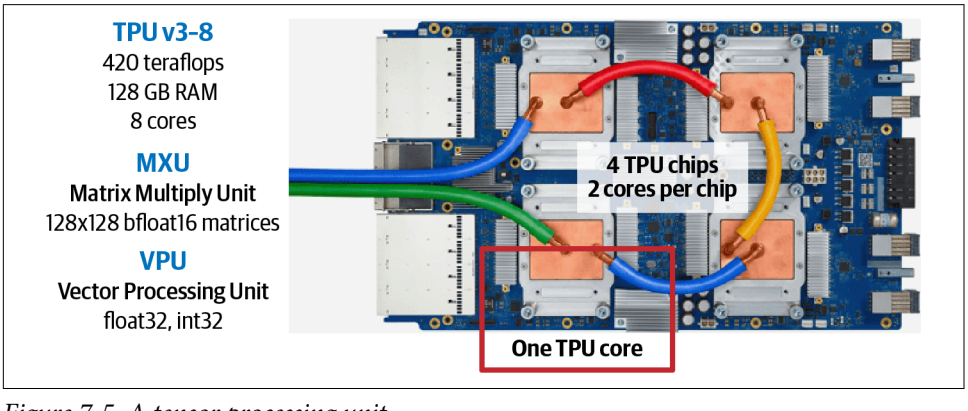

Figure 7-5. A tensor processing unit.

To create a `TPUStrategy` instance, we can call its constructor, but we have to pass a parameter to this constructor:

```
tpu = tf.distribute.cluster_resolver.TPUClusterResolver().connect()
return tf.distribute.TPUStrategy(tpu)
```

Because TPUs are multiuser machines, the initialization will wipe out the existing memory on the TPU, so we have to make sure to initialize the TPU system before we do any work in our program.

In addition, we add an extra parameter to `model.compile()`:

```
model.compile(steps_per_execution=32)
```

This parameter instructs Keras to send multiple batches to the TPU at once. In addition to lowering communication overhead, this gives the compiler the opportunity to

optimize TPU hardware utilization across multiple batches. With this option, it is no longer necessary to push batch sizes to very high values to optimize TPU performance.

It is worth noting what the user does not need to worry about—in TensorFlow/Keras, the complicated code to distribute the data is taken care of for you automatically in `strategy.distribute_dataset()`. At the time of writing, this is code you have to write by hand in PyTorch.

It's not enough to simply write the software, though; we also need to set up the hardware. For example, to use `MultiWorkerMirroredStrategy`, we will also need to launch a cluster of machines that coordinate the task of training an ML model.

To use `TPUStrategy`, we will need to launch a machine with a TPU attached to it. We can accomplish this using:

```
gcloud compute tpus execution-groups create \
    --accelerator-type v3-32 --no-forward-ports --tf-version 2.4.1 \
    --name somevmname --zone europe-west4-a \
    --metadata proxy-mode=project_editors
```

Distribution strategies are easier to implement if we use a service that manages the hardware infrastructure for us. We'll defer the hardware setup to the next section.

Serverless ML

While Jupyter notebooks are good for experimentation and training, it's a lot easier for ML engineers to maintain code in production if it's organized into Python packages. It is possible to use a tool like Papermill to directly execute a notebook. We recommend, however, that you treat notebooks as expendable, and keep your production-ready code in standalone Python files with associated unit tests.

By organizing code into Python packages, we also make it easy to submit the code to a fully managed ML service such as Google's Vertex AI, Azure ML, or Amazon SageMaker. Here, we'll demonstrate Vertex AI, but the others are similar in concept.

Creating a Python Package

To create a Python package, we have to organize files in a folder structure where each level is marked by an *__init__.py* file. The *__init__.py* file, which runs any initialization code the package needs, is required, but it can be empty. The simplest structure that would be sufficient is to have:

```
trainer/
       __init__.py
       07b_distribute.py
```

Reusable modules

How do we get code in a notebook into the file *07b_distribute.py*? An easy way to reuse code between Jupyter notebooks and the Python package is to export the Jupyter notebook to a *.py* file and then remove code whose only purpose is to display graphs and other output in the notebook. Another possibility is to base all the code development in the standalone files and simply `import` the necessary modules from the notebook cells as needed.

The reason that we create a Python package is that packages make it much easier to make our code reusable. However, it is unlikely that this model is the only one that we will train. For maintainability reasons, we suggest that you have an organizational structure like this (the full code is in *serverlessml* on GitHub):

```
flowers/                    Top-level package
      __init__.py           Initialize the flowers package
      classifier/           Subpackage for the classification model
              __init__.py
              model.py      Most of the code in the Jupyter notebook
              train.py      argparse and then launches model training
              ...
      ingest/               Subpackage for reading data
              __init__.py
              tfrecords.py  Code to read from TensorFlow Records
              ...
      utils/                Subpackage for code reusable across models
              __init__.py
              augment.py    Custom layers for data augmentation
              plots.py      Various plotting functions
              ...
```

Use Jupyter notebooks for experimentation, but at some point, move the code into a Python package and maintain that package going forward. From then on, if you need to experiment, call the Python package from the Jupyter notebook.

Invoking Python modules

Given files in the structure outlined in the previous section, we can invoke the training program using:

```
python3 -m flowers.classifier.train --job-dir /tmp/flowers
```

This is also a good time at which to make all the hyperparameters to the module settable as command-line parameters. For example, we will want to experiment with different batch sizes, so we make the batch size a command-line parameter:

```
python3 -m flowers.classifier.train --job-dir /tmp/flowers \
        --batch_size 32 --num_hidden 16 --lrate 0.0001 ...
```

Within the entrypoint Python file, we'll use Python's `argparse` library to pass the command-line parameters to the `create_model()` function.

It's best to try to make every aspect of your model configurable. Besides the L1 and L2 regularizations, it's a good idea to make data augmentation layers optional as well.

Because the code has been split across multiple files, you will find yourself needing to call functions that are now in a different file. So, you will have to add import statements of this form to the caller:

```
from flowers.utils.augment import *
from flowers.utils.util import *
from flowers.ingest.tfrecords import *
```

Installing dependencies

While the package structure we've shown is sufficient to create and run a module, it is quite likely that you will need the training service to `pip install` Python packages that you need. The way to specify that is to create a *setup.py* file in the same directory as the package, so that the overall structure becomes:

```
serverlessml/                Top-level directory
      setup.py               File to specify dependencies
      flowers/               Top-level package
            __init__.py
```

The *setup.py* file looks like this:

```
from setuptools import setup, find_packages
setup(
    name='flowers',
    version='1.0',
    packages=find_packages(),
    author='Practical ML Vision Book',
    author_email='abc@nosuchdomain.com',
    install_requires=['python-package-example']
)
```

Verify that you got the packaging and imports correct by doing two things from within the top-level directory (the directory that contains *setup.py*):

```
python3 ./setup.py dist
python3 -m flowers.classifier.train \
        --job-dir /tmp/flowers \
        --pattern '-00000-*'--num_epochs 1
```

Also look at the generated *MANIFEST.txt* file to ensure that all the desired files are there. If you need ancillary files (text files, scripts, and so on), you can specify them in *setup.py*.

Submitting a Training Job

Once we have a locally callable module, we can put the module source in Cloud Storage (e.g., `gs://${BUCKET}/flowers-1.0.tar.gz`) and then submit jobs to Vertex Training to have it run the code for us on the cloud hardware of our choice.

For example, to run on a machine with a single CPU, we'd create a configuration file (let's call it *cpu.yaml*) specifying the CustomJobSpec:

```
workerPoolSpecs:
  machineSpec:
    machineType: n1-standard-4
  replicaCount: 1
  pythonPackageSpec:
    executorImageUri: us-docker.pkg.dev/vertex-ai/training/tf-cpu.2-4:latest
    packageUris: gs://{BUCKET}/flowers-1.0.tar.gz
    pythonModule: flowers.classifier.train
    args:
    - --pattern="-*"
    - --num_epochs=20
    - --distribute="cpu"
```

We'd then provide that configuration file when starting the training program:

```
gcloud ai custom-jobs create \
  --region=${REGION} \
  --project=${PROJECT} \
  --python-package-uris=gs://${BUCKET}/flowers-1.0.tar.gz \
  --config=cpu.yaml \
  --display-name=${JOB_NAME}
```

A key consideration is that if we have developed the code using Python 3.7 and TensorFlow 2.4, we need to ensure that Vertex Training uses the same versions of Python and TensorFlow to run our training job. We do this using the `executorImageUri` setting. Not all combinations of runtimes and Python versions are supported, since some versions of TensorFlow may have had issues that were subsequently fixed. If you are developing on Vertex Notebooks, there will be a corresponding runtime on Vertex Training and Vertex Prediction (or an upgrade path to get to a consistent state). If you are developing in a heterogeneous environment, it's worth verifying that your development, training, and deployment environments support the same environment in order to prevent nasty surprises down the line.

Container or Python Package?

The purpose of putting our training code into a Python package is to make it easier to install on ephemeral infrastructure. Because we won't have the ability to log in to such machines and install software packages interactively, we want to make the installation of our training software completely automated. Python packages provide us that ability. Another way to capture dependencies, that works beyond just Python, is to containerize your training code. A container is a lightweight bundle of software that includes everything needed to run an application—besides our Python code, it will also include the Python installation itself, any system tools and system libraries that we require (such as video decoders), and all our settings (such as configuration files, authentication keys, and environment variables).

Because Vertex Training accepts both Python modules and container images, we can capture the Python and TensorFlow dependencies by building a container image that contains our training code as well as the versions of Python and TensorFlow that we need. In the container, we'd also install any extra packages our code requires. To make container creation easier, TensorFlow provides a base container image. If you are developing using Vertex Notebooks, every Notebook instance has a corresponding container image that you can use as your base (we will do this in the next section). Therefore, creating a container to run your training code is quite straightforward.

Given that both Python packages and containers are relatively easy to create, which one should you use?

Using Python packages for training can help you organize your code better and fosters reuse and maintainability. Furthermore, if you provide a Python package to Vertex Training, it will install the package on a *machine-* and *framework-optimized* container, something that you would be hard pressed to do if you were building your own containers.

On the other hand, using a container for training is much more flexible. For example, a container is a good option in a project where you need to use old or unsupported versions of a runtime or need to install proprietary software components such as database connectors to internal systems.

So, the choice comes down to what you value more: efficiency (in which case you'd choose a Python package) or flexibility (in which case you'd choose a container).

In the training code, a `OneDeviceStrategy` should be created:

```
strategy = tf.distribute.OneDeviceStrategy('/cpu:0')
```

Using the `gcloud` command to launch a training job makes it easy to incorporate model training in scripts, invoke the training job from Cloud Functions, or schedule the training job using Cloud Scheduler.

Next, let's walk through the hardware setups corresponding to the different distribution scenarios that we have covered so far. Each scenario here corresponds to a different distribution strategy.

Running on multiple GPUs

To run on a single machine with one, two, four, or more GPUs, we can add a snippet like this to the YAML configuration file:

```
workerPoolSpecs:
  machineSpec:
    machineType: n1-standard-4
    acceleratorType: NVIDIA_TESLA_T4
    acceleratorCount: 2
  replicaCount: 1
```

and launch the `gcloud` command as before, making sure to specify this configuration file in `--config`.

In the training code, a `MirroredStrategy` instance should be created.

Distribution to multiple GPUs

To run on multiple workers, each of which has several GPUs, the configuration YAML file should include lines similar to the following:

```
workerPoolSpecs:
  - machineSpec:
      machineType: n1-standard-4
      acceleratorType: NVIDIA_TESLA_T4
      acceleratorCount: 1
  - machineSpec:
      machineType: n1-standard-4
      acceleratorType: NVIDIA_TESLA_T4
      acceleratorCount: 1
  replicaCount: 1
```

Remember that if you are using multiple worker machines, you should use virtual epochs by declaring the number of training examples that you will term as an epoch. Shuffling is also required. The code example in *serverlessml* on GitHub does both these things.

In the training code, a `MultiWorkerMirroredStrategy` instance should be created.

Distribution to TPU

To run on a Cloud TPU, the configuration file YAML looks like this (choose the version of TPU that is most appropriate at the time you are reading this):

```
workerPoolSpecs:
        - machineSpec:
```

```
                machineType: n1-standard-4
                acceleratorType:TPU_V2
                acceleratorCount: 8
```

In the training code, a TPUStrategy instance should be created.

You can use Python's error handling mechanism to create a boilerplate method for creating the distribution strategy appropriate for the hardware configuration:

```
def create_strategy():
        try:
            # detect TPUs
            tpu = tf.distribute.cluster_resolver.TPUClusterResolver().connect()
            return tf.distribute.experimental.TPUStrategy(tpu)
        except ValueError:
            # detect GPUs
            return tf.distribute.MirroredStrategy()
```

Now that we have looked at how to train a single model, let's consider how to train a family of models and pick the best one.

Hyperparameter Tuning

In the process of creating our ML model, we have made many arbitrary choices: the number of hidden nodes, the batch size, the learning rate, the L1/L2 regularization amounts, and so on. The overall number of possible combinations is massive, so it's preferable to take an optimization approach where we specify a budget (e.g., "try 30 combinations") and instead ask a hyperparameter optimization technique to choose the best settings.

In Chapter 2, we looked at the in-built Keras Tuner. However, that only works if your model and dataset are small enough that the entire training process can be carried out wrapped within the tuner. For more realistic ML datasets, it's better to use a fully managed service.

Fully managed hyperparameter training services provide a combination of parameter values to the training program, which then trains the model and reports on performance metrics (accuracy, loss, etc.). So, the hyperparameter tuning service requires that we:

- Specify the set of parameters to tune, the search space (the range of values each parameter can take, for example that the learning rate has to be between 0.0001 and 0.1), and the search budget.
- Incorporate a given combination of parameters into the training program.
- Report how well the model performed when using that combination of parameters.

In this section, we'll discuss hyperparameter tuning on Vertex AI as an example of how this works.

Specifying the search space

We specify the search space in the YAML configuration provided to Vertex AI. For example, we might have:

```
displayName: "FlowersHpTuningJob"
maxTrialCount: 50
parallelTrialCount: 2
studySpec:
  metrics:
  - metricId: accuracy
    goal: MAXIMIZE
  parameters:
  - parameterId: l2
    scaleType: UNIT_LINEAR_SCALE
    doubleValueSpec:
      minValue: 0
      maxValue: 0.2
  - parameterId: batch_size
    scaleType: SCALE_TYPE_UNSPECIFIED
    discreteValueSpec:
      values:
      - 16
      - 32
      - 64
  algorithm: ALGORITHM_UNSPECIFIED
```

In this YAML listing, we are specifying (see if you can find the corresponding lines):

- The goal, which is to maximize the accuracy that is reported by the trainer
- The budget, which is a total of 50 trials carried out 2 at a time
- That we want to stop a trial early if it looks unlikely to do better than we have already seen
- Two parameters, `l2` and `batch_size`:
  - The possible L2 regularization strengths (between 0 and 0.2)
  - The batch size, which can be one of 16, 32, or 64
- The algorithm type, which if unspecified uses Bayesian Optimization

Using parameter values

Vertex AI will invoke our trainer, passing specific values for `l2` and `batch_size` as command-line parameters. So, we make sure to list them in the `argparse`:

```
parser.add_argument(
    '--l2',
```

```
        help='L2 regularization', default=0., type=float)
    parser.add_argument(
        '--batch_size',
        help='Number of records in a batch', default=32, type=int)
```

We have to incorporate these values into the training program. For example, we'll use the batch size as:

```
train_dataset = create_preproc_dataset(
    'gs://...' + opts['pattern'],
    IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS
).batch(opts['batch_size'])
```

It's helpful at this point to step back and think carefully about all the implicit choices that we have made in the model. For example, our `CenterCrop` augmentation layer was:

```
tf.keras.layers.experimental.preprocessing.RandomCrop(
    height=IMG_HEIGHT // 2, width=IMG_WIDTH // 2,
    input_shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS),
    name='random/center_crop'
)
```

The number 2 is baked in, yet the truly fixed thing is the size of the image (224x224x3) that the MobileNet model requires. It's worth experimenting with whether we should center crop the images to 50% the original size, or use some other ratio. So, we make `crop_ratio` one of the hyperparameters:

```
- parameterName: crop_ratio
  type: DOUBLE
  minValue: 0.5
  maxValue: 0.8
  scaleType: UNIT_LINEAR_SCALE
```

and use it as follows:

```
IMG_HEIGHT = IMG_WIDTH = round(MODEL_IMG_SIZE / opts['crop_ratio'])
tf.keras.layers.experimental.preprocessing.RandomCrop(
    height=MODEL_IMG_SIZE, width=MODEL_IMG_SIZE,
    input_shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS),
    name='random/center_crop'
)
```

Reporting accuracy

After we train the model using the hyperparameters that were supplied to the trainer on the command line, we need to report back to the hyperparameter tuning service. What we report back is whatever we specified as the `hyperparameterMetricTag` in the YAML file:

```
hpt = hypertune.HyperTune()
accuracy = ...
hpt.report_hyperparameter_tuning_metric(
```

```
        hyperparameter_metric_tag='accuracy',
        metric_value=accuracy,
        global_step=nepochs)
```

Result

On submitting the job, hyperparameter tuning is launched and 50 trials are carried out, 2 at a time. The hyperparameters for these trials are chosen using a Bayesian optimization approach, and because we specified two parallel trials, the optimizer starts with two random initial starting points. Whenever a trial finishes, the optimizer determines which part of the input space needs further exploration and a new trial is launched.

The cost of the job is determined by the infrastructure resources used to train the model 50 times. Running the 50 trials 2 at a time causes the job to finish twice as fast as if we'd run them only one at a time. If we were to run the 50 trials 10 at a time, the job would finish 10 times faster but cost the same—however, the first 10 trials wouldn't get much of a chance to incorporate information from the previously finished trials, and future trials, on average, are unable to take advantage of the information from 9 already-started trials. We recommend using as many total trials as your budget allows and as few prallel trials as your patience allows! You can also resume an already completed hyperparameter job (specify `resumePreviousJobId` in the YAML) so you can continue the search if you find more budget or more patience.

The results are shown in the web console (see Figure 7-6).

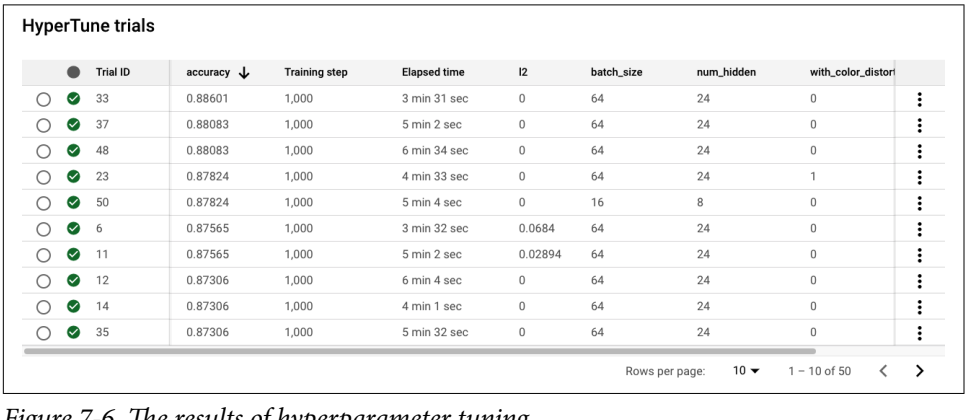

Figure 7-6. The results of hyperparameter tuning.

Based on the tuning, the highest accuracy (0.89) is obtained with the following settings: `l2=0`, `batch_size=64`, `num_hidden=24`, `with_color_distort=0`, `crop_ratio=0.70706`.

Continuing tuning

Looking at these results, it is striking that the optimal values for `num_hidden` and `batch_size` are the highest values we tried. Given this, it might be a good idea to continue the hyperparameter tuning process and explore even higher values. At the same time, we can reduce the search space for the `crop_ratio` by making it a set of discrete values (0.70706 should probably be just 0.7).

This time, we don't need Bayesian optimization. We just want the hyperparameter service to carry out a grid search of 45 possible combinations (this is also the budget):

```
    - parameterId: batch_size
      scaleType: SCALE_TYPE_UNSPECIFIED
      discreteValueSpec:
        values:
        - 48
        - 64
        - 96
    - parameterId: num_hidden
      scaleType: SCALE_TYPE_UNSPECIFIED
      discreteValueSpec:
        values:
        - 16
        - 24
        - 32
    - parameterId: crop_ratio
      scaleType: SCALE_TYPE_UNSPECIFIED
      discreteValueSpec:
        values:
        - 0.65
        - 0.70
        - 0.75
        - 0.80
        - 0.85
```

After this new training run, we get a report as before, and we can select the best set of parameters. When we did this, it turned out that `batch_size=64`, `num_hidden=24` was indeed the best—better than choosing 96 for the batch size or 32 for the number of hidden nodes—but with `crop_ratio=0.8`.

Deploying the Model

Now that we have a trained model, let's deploy it for online predictions. The TensorFlow SavedModel format is supported by a serving system called TensorFlow Serving. A Docker container for TensorFlow Serving is available for you to deploy this in a container orchestration system like Google Kubernetes Engine, Google Cloud Run, Amazon Elastic Kubernetes Service, AWS Lambda, Azure Kubernetes Service, or on premises using Kubernetes. Managed versions of TensorFlow Serving are available in

all the major clouds. Here, we'll show you how to deploy the SavedModel into Google's Vertex AI.

Vertex AI also provides model management and versioning capabilities. In order to use these functionalities, we'll create an endpoint called *flowers* to which we will deploy multiple model versions:

```
gcloud ai endpoints create --region=us-central1 --display-name=flowers
```

Suppose, for example, that hyperparameter tuning trial #33 was the best and contains the model we want to deploy. This command will create a model called `txf` (for transfer learning) and deploy it into the flowers endpoint:

```
MODEL_LOCATION="gs://...}/33/flowers_model"
gcloud ai models upload ---display-name=txf \
           --container-image-uri=".../tf2-cpu.2-1:latest" -artifact-uri=$MODEL_LOCATION
gcloud ai endpoints deploy-model $ENDPOINT_ID  --model=$MODEL_ID \
            ... --region=us-central1  --traffic-split=
```

Once the model is deployed, we can do an HTTP POST of a JSON request to the model to obtain predictions. For example, posting:

```
{"instances": [
    {"filenames": "gs://cloud-ml-data/.../9853885425_4a82356f1d_m.jpg"},
    {"filenames": "gs://cloud-ml-data/../8713397358_0505cc0176_n.jpg"}
]}
```

returns:

```
{
    "predictions": [
        {
            "probability": 0.9999885559082031,
            "flower_type_int": 1,
            "flower_type_str": "dandelion"
        },
        {
            "probability": 0.9505964517593384,
            "flower_type_int": 4,
            "flower_type_str": "tulips"
        }
    ]
}
```

Of course, we could post this request from any program capable of sending an HTTP POST request (see Figure 7-7).

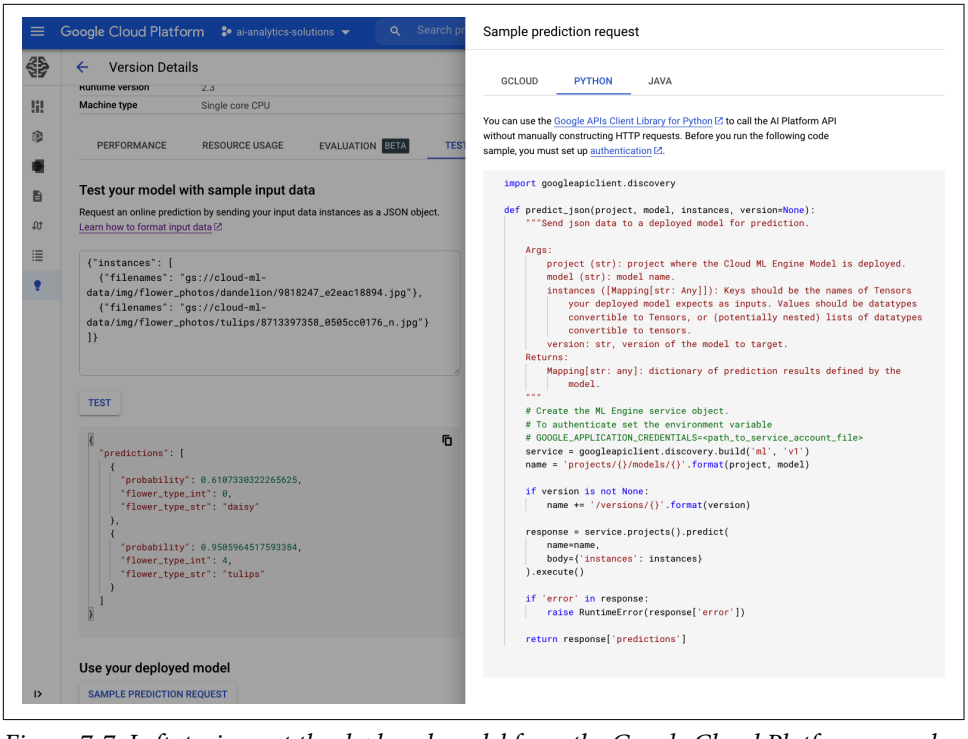

Figure 7-7. Left: trying out the deployed model from the Google Cloud Platform console. Right: example code for replicating in Python.

How would someone use this model? They would have to upload an image file to the cloud, and send the path to the file to the model for predictions. This process is a bit onerous. Can the model not directly accept the contents of an image file? We'll look at how to improve the serving experience in Chapter 9.

Summary

In this chapter, we covered various aspects of building a training pipeline. We started by considering efficient storage in TFRecords files, and how to read that data efficiently using a `tf.data` pipeline. This included parallel execution of map functions, interleaved reading of datasets, and vectorization. The optimization ideas carried over into the model itself, where we looked at how to parallelize model execution across multiple GPUs, multiple workers, and on TPUs.

We then moved on to operationalization considerations. Rather than managing infrastructure, we looked at how to carry out training in a serverless way by submitting a training job to Vertex AI, and how to use this paradigm to carry out distributed training. We also looked at how to use Vertex AI's hyperparameter tuning service to

achieve better model performance. For predictions, we need autoscaling infrastructure, so we looked at how to deploy a SavedModel into Vertex AI. Along the way, you learned about signatures, how to customize them, and how to get predictions out of a deployed model.

In the next chapter, we will look at how to monitor the deployed model.

CHAPTER 8

Model Quality and Continuous Evaluation

So far in this book, we have covered the design and implementation of vision models. In this chapter, we will dive into the important topic of monitoring and evaluation. In addition to beginning with a high-quality model, we also want to maintain that quality. In order to ensure optimal operation, it is important to obtain insights through monitoring, calculate metrics, understand the quality of the model, and continuously evaluate its performance.

Monitoring

So, we've trained our model on perhaps millions of images, and we are very happy with its quality. We've deployed it to the cloud, and now we can sit back and relax while it makes great predictions forever into the future… Right? Wrong! Just as we wouldn't leave a small child alone to manage him or herself, we also don't want to leave our models alone out in the wild. It's important that we constantly monitor their quality (using metrics like accuracy) and computational performance (queries per second, latency, etc.). This is especially true when we're constantly retraining models on new data that may contain distribution changes, errors, and other issues that we'll want to be aware of.

TensorBoard

Often ML practitioners train their models without fully considering all the details. They submit a training job and check it every now and then until the job is finished. Then they make predictions using the trained model to see how it's performing. This may not seem like a big deal if the training jobs take a few minutes. However, many computer vision projects, especially with datasets that contain millions of images, have training jobs that take days or weeks. It would be terrible if something went

wrong with the training early on and we didn't notice until training was complete, or until we tried to use the model to make some predictions.

There is a great monitoring tool called TensorBoard that is distributed with TensorFlow that we can use to avoid just this scenario. TensorBoard is an interactive dashboard (see Figure 8-1) that displays summaries saved during model training and evaluation. You can use it as a historical record of experiments run, for comparing different versions of your model or code, and for analyzing training jobs.

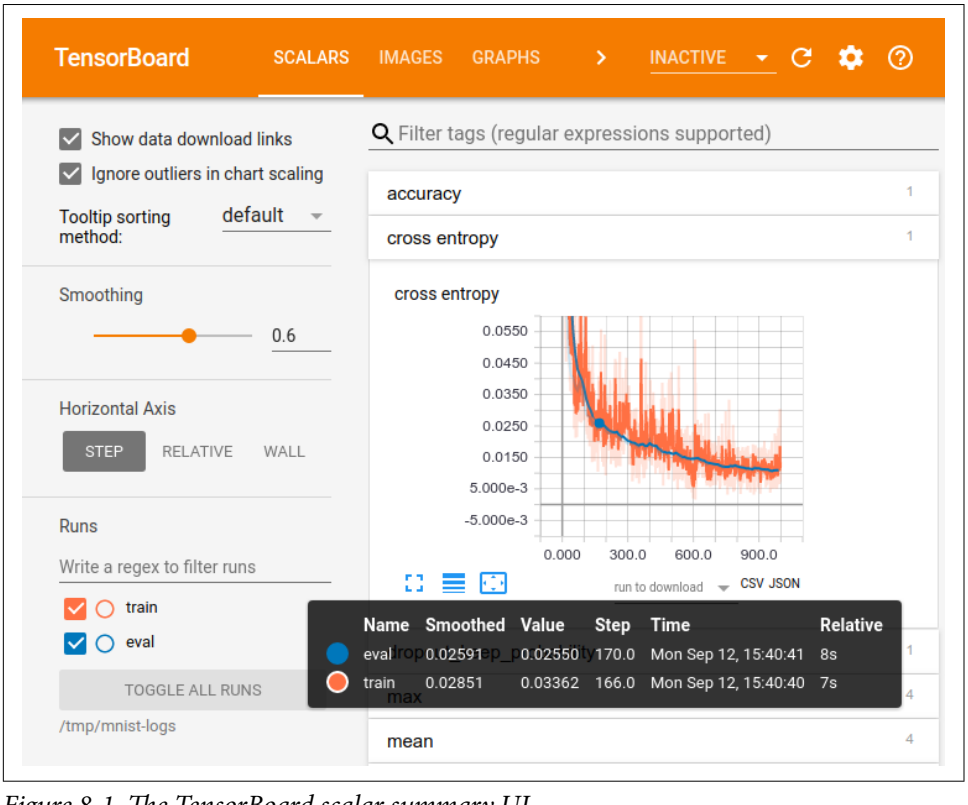

Figure 8-1. The TensorBoard scalar summary UI.

TensorBoard allows us to monitor loss curves to make sure that the model training is still progressing, and it hasn't stopped improving. We can also display and interact with any other evaluation metrics we have in our model, such as accuracy, precision, or AUC—for example, we can perform filtering across multiple series, smoothing, and outlier removal, and we're able to zoom in and out.

Weight Histograms

We can also explore histograms in TensorBoard, as shown in Figure 8-2. We can use these to monitor weights, gradients, and other scalar quantities that have too many values to inspect individually.

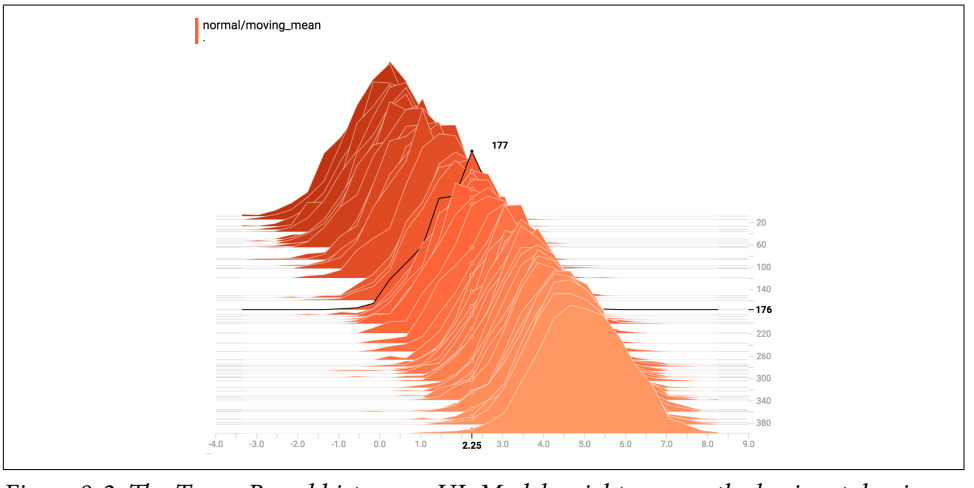

Figure 8-2. The TensorBoard histogram UI. Model weights are on the horizontal axis and the training step number is on the vertical axis.

What Should the Weight Distributions Look Like?

Each layer of a neural network can have thousands or millions of weights. As with any large collection of values, they form a distribution. The distribution of weights at the beginning of training and at the end can be very different. Usually the initial distribution is based on our weight initialization strategy, whether that be samples from a random normal distribution or a random uniform distribution, with or without normalization factors, and so on.

As the model gets trained and converges, however, the central limit theorem (CLT) tells us that the weight distribution should ideally start to look more Gaussian. It states that if we have a population with mean μ and standard deviation σ, then given a large enough number of random samples, the distribution of the sample means should be approximately Gaussian. But if we have a systematic problem in our model, then the weights will reflect that problem and may be skewed toward zero values (an indication of the presence of "dead layers," which happen if input values are poorly scaled) or very large ones (which happens as a result of overfitting). Thus, by looking at the weight distribution, we can diagnose whether there is a problem.

If our weight distribution is not looking Gaussian, there are a few things we can do. We can scale our input values from [0, 1] to [–1, 1] if the distribution is skewed toward zero values. If the problem is in intermediate layers, try adding batch

normalization. If the distribution tends toward large values, we can try adding regularization or increasing the dataset size. Other issues can be resolved through trial and error. Perhaps we chose our initialization strategy poorly, and it made it hard for the weights to move to the standard regime of small, normally distributed weights. In that case, we can try changing the initialization method.

We may also be having gradient issues that could be moving the weights away from a Gaussian distribution. We can fix this by adding gradient clipping, constraints, or penalties (for example, adding a loss term that grows the farther gradients stray from a set value). Additionally, the order and distribution of examples in our mini-batches can affect the evolution of our weight distribution, so experimenting with those parameters might help make the weight distribution more Gaussian.

Device Placement

We can output the TensorFlow model graph to TensorBoard for visualization and exploration, as shown in Figure 8-3. .

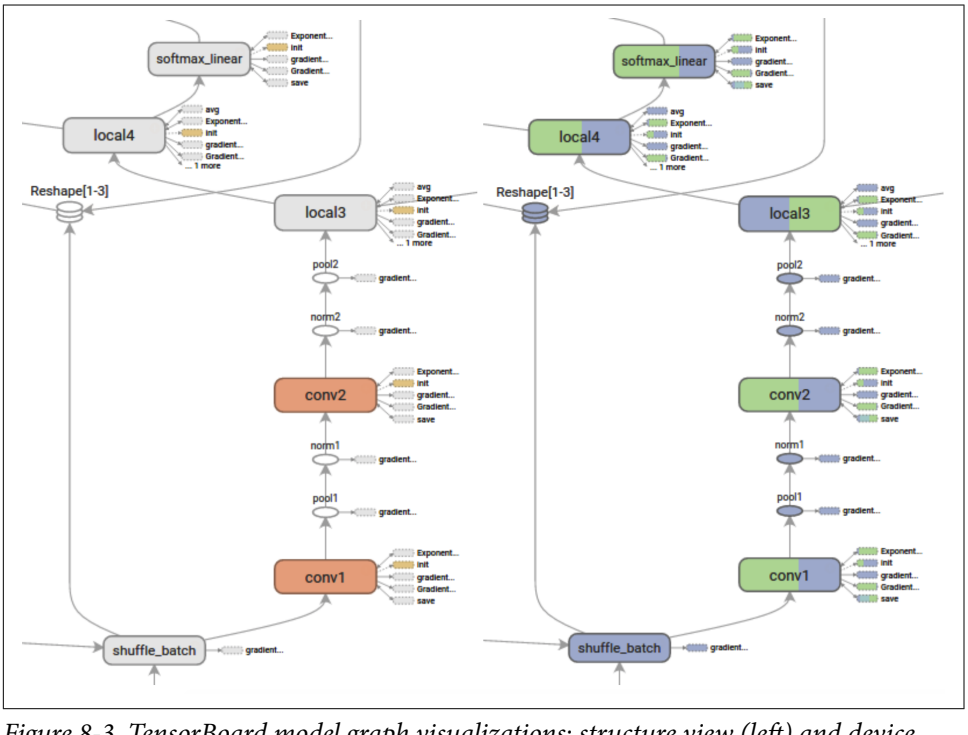

Figure 8-3. TensorBoard model graph visualizations: structure view (left) and device view (right).

The default structure view shows which nodes share the same structure, and the device view shows which nodes are on which device(s), with a color per device. We can also see TPU compatibility and more. This can allow us to ensure that our model code is being accelerated properly.

Data Visualization

TensorBoard can display examples of specific types of data, such as images (on the Images tab, shown on the left in Figure 8-4) or audio (on the Audio tab). This way, we can get feedback as training is progressing; for example, with image generation, we can see live how our generated images are looking. For classification problems, TensorBoard also has the ability to display confusion matrices, as seen on the right in Figure 8-4, so we can monitor metrics per class throughout a training job (more on this in "Metrics for Classification" on page 287).

Figure 8-4. The TensorBoard Images tab allows you to visualize training images (left) and view a confusion matrix (right) to see where the classifier is making most of its mistakes.

Training Events

We can add a TensorBoard callback to our model using code like the following:

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir='logs', histogram_freq=0, write_graph=True,
    write_images=False, update_freq='epoch', profile_batch=2,
    embeddings_freq=0, embeddings_metadata=None, **kwargs
)
```

We specify the directory path where the TensorBoard event logs will get written to disk using the `log_dir` argument. `histogram_freq` and `embeddings_freq` control how often (in epochs) those two types of summaries are written; if you specify a value of zero they are not computed or displayed. Note that validation data, or at least a

split, needs to be specified when fitting the model for histograms to show. Furthermore, for embeddings, we can pass a dictionary to the argument `embeddings_metadata` that maps layer names to a filename where the embedding metadata will be saved.

If we want to see the graph in TensorBoard, we can set the `write_graph` argument to `True`; however, the event log files can get quite sizable if our model is large. The update frequency is specified through the `update_freq` argument. Here it is set to update every epoch or batch, but we can set it to an integer value to have it update after that number of batches. We can visualize model weights as images in TensorBoard using the Boolean argument `write_images`. Lastly, if we want to profile the performance of our compute characteristics, such as the contributions to the step time, we can set `profile_batch` to an integer or tuple of integers and it will profile that batch or range of batches. Setting the value to zero disables profiling.

Once defined, we can add the TensorBoard callback to `model.fit()`'s callbacks list as shown here:

```
history = model.fit(
    train_dataset,
    epochs=10,
    batch_size=1,
    validation_data=validation_dataset,
    callbacks=[tensorboard_callback]
)
```

The simplest way to run TensorBoard is to open a terminal and run the following bash command:

```
tensorboard --logdir=<path_to_your_logs>
```

You can provide other arguments, for example to change the default port TensorBoard uses, but to quickly get it up and running you simply need to specify the `log dir`.

The summaries typically include loss and evaluation metric curves. However, we can use callbacks to emit other potentially useful summaries, like images and weight histograms, depending on our use case. We can also print out and/or log the loss and eval metrics while training is taking place, as well as have periodic evaluations of generated images or other model outputs that we can then inspect for diminishing returns in improvement. Lastly, if training locally with `model.fit()`, we can inspect the history output and look at the loss and eval metrics and how they change over time.

Model Quality Metrics

Even if you are using a validation set, looking at the validation loss doesn't really give a clear picture of how well the model is performing. Enter the evaluation metrics! These are metrics that are calculated based on the model's predictions on unseen data that allow us to evaluate how the model is doing in terms that are related to the use case.

Metrics for Classification

As you learned in previous chapters, image classification involves assigning labels to images that indicate which class they belong to. Labels can be mutually exclusive, with only a single label applying to any given image, or it may be possible for multiple labels to describe an image. In both the single-label and multilabel cases, we typically predict a probability across each of the classes for an image. Since our predictions are probabilities and our labels are usually binary (0 if the image is not that class and 1 if it is), we need some way to convert predictions into a binary representation so we can compare them with the actual labels. To do that, we typically set a threshold: any predicted probabilities below the threshold become a 0, and any predicted probabilities above it become a 1. In binary classification the default threshold is normally 0.5, giving an equal chance for both choices.

Binary classification

There are many metrics for single-label classification that are used in practice, but the best choice depends on our use case. In particular, different evaluation metrics are appropriate for binary and multiclass classification. Let's begin with binary classification.

The most common evaluation metric is accuracy. This is a measure of how many predictions our model got right. To figure that out, it's also useful to calculate four other metrics: true positives, true negatives, false positives, and false negatives. True positives are when the label is 1, indicating that the example belongs to a certain class, and the prediction is also 1. Similarly, true negatives are when the label is 0, indicating that the example does not belong to that class, and the prediction is also 0. Conversely, a false positive is when the label is 0 but the prediction is 1, and a false negative is when the label is 1 but the prediction is 0. Taken together, these create something called a *confusion matrix* for the set of predictions, which is a 2x2 grid that counts the number of each of these four metrics, as can be seen in Figure 8-5.

		Actual class	
		Yes	No
Predicted class	Yes	TP	FP
	No	FN	TN

Figure 8-5. A binary classification confusion matrix.

We can add these four metrics to our Keras model as follows:

```
model.compile(
    optimizer="sgd",
    loss="mse",
    metrics=[
        tf.keras.metrics.TruePositives(),
        tf.keras.metrics.TrueNegatives(),
        tf.keras.metrics.FalsePositives(),
        tf.keras.metrics.FalseNegatives(),
    ]
)
```

Classification accuracy is the percentage of correct predictions, so it's calculated by dividing the number of predictions the model got right by the total number of predictions it made. Using the four confusion matrix metrics, this can be expressed as:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

In TensorFlow, we can add an accuracy metric to our Keras model like this:

```
model.compile(optimizer="sgd", loss="mse",
    metrics=[tf.keras.metrics.Accuracy()]
)
```

This counts the number of predictions that matched the labels and then divides by the total number of predictions.

If our predictions and labels are all either 0 or 1, as in the case of binary classification, then we could instead add the following TensorFlow code:

```
model.compile(optimizer="sgd", loss="mse",
    metrics=[tf.keras.metrics.BinaryAccuracy()]
)
```

In this case the predictions are most likely probabilities that are thresholded to 0 or 1 and then compared with the actual labels to see what percentage of them match.

If our labels are categorical, one-hot encoded, then we could instead add the following TensorFlow code:

```
model.compile(optimizer="sgd", loss="mse",
    metrics=[tf.keras.metrics.CategoricalAccuracy()]
)
```

This is more common for the multiclass case and usually involves comparing a vector of predicted probabilities for each class to a one-hot-encoded vector of labels for each example.

A problem with accuracy, however, is that it works well only when the classes are balanced. For example, suppose our use case is to predict whether a retinal image depicts eye disease. Let's say we have screened one thousand patients, and only two of them actually have eye disease. A biased model that predicts that every image shows a healthy eye would be correct 998 times and wrong only twice, thus achieving 99.8% accuracy. While that might sound impressive, this model is actually useless to us because it will completely fail to detect the cases we're actually looking for. For this specific problem, accuracy is not a useful evaluation metric. Thankfully, there are other combinations of the confusion matrix values that can be more meaningful for imbalanced datasets (and also for balanced ones).

If, instead, we were interested in the percentage of positive predictions that our model got right, then we would be measuring the *prediction precision*. In other words, how many patients really have eye disease out of all the ones the model predicted to have eye disease? Precision is calculated as follows:

$$precision = \frac{TP}{TP + FP}$$

Similarly, if we wanted to know the percentage of positive examples that our model was able to correctly identify, then we would be measuring the *prediction recall*. In other words, of the patients who really had the eye disease, how many did the model find? Recall is calculated as:

$$recall = \frac{TP}{TP + FN}$$

In TensorFlow, we can add these two metrics to our Keras model using:

```
model.compile(optimizer="sgd", loss="mse",
    metrics=[tf.keras.metrics.Precision(), tf.keras.metrics.Recall()]
)
```

We could also add a `thresholds` argument either as a float in the range [0, 1] or a list or tuple of float values if we want the metrics calculated at thresholds other than 0.5.

As you can see, with precision and recall the numerators are identical and the denominators only differ in whether they include false positives or false negatives.

Therefore, typically when one goes up, the other goes down. So how do we find a good balance point between the two? We can add another metric, the F1 score:

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

The F1 score is simply the harmonic mean between precision and recall. Like accuracy, precision, and recall, it has a range between 0 and 1. An F1 score of 1 indicates a model with perfect precision and recall and thus perfect accuracy. An F1 score of 0 means either the precision or the recall is 0, which means that there are no true positives. This indicates either that we have a terrible model or that our evaluation dataset contains no positive examples at all, denying our model the chance to learn how to predict positive examples well.

A more general metric known as the F_β score adds a real-valued constant between 0 and 1, β, which allows us to scale the importance of precision or recall in the F-score equation:

$$F_\beta = \left(1 + \beta^2\right) * \frac{precision * recall}{\beta^2 * precision + recall}$$

This is useful if we want to use a more aggregate measure than precision or recall alone, but the costs associated with false positives and false negatives are different; it allows us to optimize for the one we care about the most.

All the evaluation metrics we've looked at so far require us to choose a classification threshold which determines whether the probabilities are high enough to become positive class predictions or not. But how do we know where to set the threshold? Of course, we could try many possible threshold values and then choose the one that optimizes the metric we care most about.

However, if we're using multiple thresholds, there is another way to compare models across all thresholds at once. This first involves building curves out of metrics across a grid of thresholds. The two most popular curves are the *receiver operating characteristic* (ROC) and *precision-recall* curves. ROC curves have the true positive rate, also known as the sensitivity or recall, on the y-axis; and the false positive rate, also known as 1-specificity (the true negative rate) or fallout, along the x-axis. The false positive rate is defined as:

$$FPR = \frac{FP}{FP + TN}$$

Precision-recall curves have precision on the y-axis and recall on the x-axis.

Suppose we've chosen a grid of two hundred equally spaced thresholds, and calculated the thresholded evaluation metrics for both the horizontal and vertical axes for either type of curve. Of course, plotting these points will create a line that extends across all two hundred thresholds.

Generating curves like these can help us with threshold selection. We want to choose a threshold that optimizes the metric of interest. It could be one of these statistical metrics, or, better yet, a metric relevant to the business or use case at hand, such as the economic cost of missing a patient who has eye disease versus carrying out additional unnecessary screening of a patient who doesn't have eye disease.

We can summarize this information into a single number by calculating the *area under the curve* (AUC). As we can see on the left side of Figure 8-6, a perfect classifier would have an AUC of 1 because there would be a 100% true positive rate and a 0% false positive rate. A random classifier would have an AUC of 0.5 because the ROC curve would fall along the y = x-axis, which shows that the numbers of true positives and false positives grow at equal rates. If we calculate an AUC less than 0.5, then that means our model is performing *worse* than a random classifier; an AUC of 0 means the model was perfectly wrong about every prediction. All else being equal, a higher AUC is usually better, with the possible range being between 0 and 1.

The precision-recall (PR) curve is similar, as we can see on the right of Figure 8-6; however, not every point in PR space may be obtained, and thus the range is less than [0, 1]. The actual range depends on how skewed the data's class distributions are.

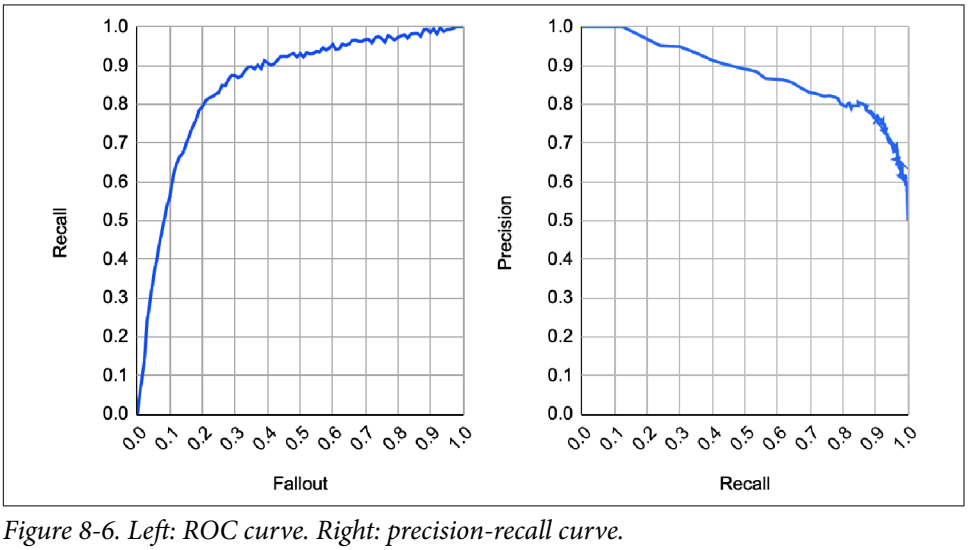

Figure 8-6. Left: ROC curve. Right: precision-recall curve.

So, which curve should we use when comparing classification models? If the classes are well sampled and balanced, then calculating the AUC-ROC is recommended.

Otherwise, if the classes are imbalanced or skewed, then AUC-PR is the recommended choice. Here is the TensorFlow code to add the AUC evaluation metric:

```
tf.keras.metrics.AUC(
    num_thresholds=200, curve="ROC",
    summation_method="interpolation",
    thresholds=None, multi_label=False
)
```

We can set the number of thresholds to calculate the four confusion metrics via the `num_thresholds` argument, which will create that number of equally spaced thresholds between 0 and 1. Alternatively, we can provide a list of float thresholds within the range [0, 1] that `tf.keras.metrics.AUC()` will use instead to calculate the AUC.

We can also set the type of curve via the `curve` argument to either `"ROC"` or `"PR"` to use an ROC or precision-recall curve, respectively.

Lastly, since we are performing binary classification, we set `multi_label` to `False`. Otherwise, it would calculate the AUC for each class and then average.

Multiclass, single-label classification

If we instead have a multiclass classification problem, let's say with three classes (dog, cat, and bird), then the confusion matrix will look like Figure 8-7. Notice that instead of a 2x2 matrix we now have a 3x3 matrix; thus, in general, it will be an *nxn* matrix where *n* is the number of classes. A key difference between the binary classification problem and the multiclass classification problem is that we don't have true negatives anymore, because those are now the "true positives" of the other classes.

		Actual classes		
		Dog	Cat	Bird
Predicted classes	Dog			
	Cat			
	Bird			

Figure 8-7. A multiclass confusion matrix with three classes.

Remember, for multiclass, single-label classification, even though we have multiple classes, each instance still belongs to one and only one class. The labels are mutually exclusive. It is either a picture of a dog, a cat, or a bird, not more than one of these things.

How can we fit our binary classification confusion matrix metrics into our multiclass version? Let's walk through an example. If we have an image that is labeled a dog and we predict correctly that it is a dog, then the count in the dog-dog cell of the matrix

gets incremented by one. This is what we called a true positive in the binary classification version. But what if instead our model predicted "cat"? It's obviously a false something, but it doesn't really fit into the false positive or false negative camps. It's just…false, wrong.

Thankfully, we don't have to leap too far to get our multiclass confusion matrix to work for us. Let's look at the confusion matrix again, with values filled in this time (Figure 8-8).

		Actual classes		
		Dog	**Cat**	**Bird**
Predicted classes	**Dog**	150	50	50
	Cat	30	125	60
	Bird	20	25	90

Figure 8-8. A dog, cat, bird multiclass classification confusion matrix example.

We can see that this is a balanced dataset, because each class has two hundred examples. However, it is not a perfect model since it is not a purely diagonal matrix; it has gotten many examples wrong, as evidenced by the off-diagonal counts. If we want to be able to calculate the precision, recall, and other metrics, then we must look at each class individually.

Looking only at the dog class, our confusion matrix contracts to what we see in Figure 8-9. We can see in this figure that our true positives are where the image was actually a dog and we predicted a dog, which was the case for 150 examples. The false positives are where we predicted the image was a dog but it was not (i.e., it was a cat or a bird). Therefore, to get this count we add together the 50 examples from the dog-cat cell and the 50 examples from the dog-bird cell. To find the count of false negatives, we do the opposite: these are cases where we should have predicted a dog but didn't, so to get their total we add together the 30 examples from the cat-dog cell and the 20 examples from the bird-dog cell. Lastly, the true negative count is the sum of the rest of the cells, where we correctly said that those images were not pictures of dogs. Remember, even though the model might have gotten cats and birds mixed up with each other in some cases, because for now we are only looking at the dog class those values all get lumped together in the true negative count.

		Actual dog	
		Yes	No
Predicted dog	Yes	TP = 150	FP = 50 + 50
	No	FN = 30 + 20	TN = 125 + 25 + 60 + 90

Figure 8-9. The dog classification confusion matrix.

Once we've done this for every class, we can calculate the composite metrics (precision, recall, F1 score, etc.) for each class. We can then take the unweighted average of each of these to get the macro versions of these metrics—for example, averaging the precisions across all classes would give the macro-precision. There is also a micro version, where instead we add up all of the true positives from each of the individual class confusion matrices into a global true positive count and do the same for the other three confusion metrics. However, since this was done globally, the micro-precision, micro-recall, and micro-F1 score will all be the same. Lastly, instead of using an unweighted average as we did in the macro versions, we could weight each class's individual metric by the total number of samples of that class. This would then give us the weighted precision, weighted recall, and so on. This can be useful if we have imbalanced classes.

Since these all still used thresholds to convert the predicted class probabilities into a 1 or 0 for the winning class, we can use these combined metrics for various thresholds to make ROC or precision-recall curves to find the AUC for comparing threshold-agnostic model performance.

Multiclass, multilabel classification

In binary (single-class, single-label) classification, the probabilities are mutually exclusive and each example either is the positive class or is not. In multiclass single-label classification the probabilities are again mutually exclusive, so each example can belong to one and only one class, but there are no positive and negative classes. The third type of classification problem is multiclass multilabel classification, where the probabilities are no longer mutually exclusive. An image doesn't necessarily have to be of just a dog or just a cat. If both are in the image, then the labels for both dog and cat can be 1, and therefore a good model should predict a value close to 1 for each of those classes, and a value close to 0 for any other classes.

What evaluation metrics can we use for the multilabel case? We have several options, but first let's define some notation. We'll define Y to be the set of actual labels, Z to be the set of predicted labels, and the function I to be the indicator function.

A harsh and challenging metric to maximize is the *exact match ratio* (EMR), also known as the *subset accuracy*:

$$EMR = \frac{1}{n}\sum_{i=1}^{n} I(Y_i = Z_i)$$

This measures the percentage of examples where we got *all* of the labels exactly right. Note that this does not give partial credit. If we were supposed to predict that one hundred classes are in an image but we only predict 99 of them, then that example isn't counted as an exact match. The better the model, the higher the EMR should be.

A less strict metric we could use is the *Hamming score*, which is effectively the multilabel accuracy:

$$HS = \frac{1}{n}\sum_{i=1}^{n} \frac{|Y_i \cap Z_i|}{|Y_i \cup Z_i|}$$

Here we are measuring the ratio of predicted correct labels to the total number of labels, predicted and actual, for each example, averaged across all examples. We want to maximize this quantity. This is similar to the Jaccard index or intersection over union (IOU), which we looked at in Chapter 4.

There is also a *Hamming loss* that can be used, which has a range of [0, 1]:

$$HL = \frac{1}{kn}\sum_{i=1}^{n}\sum_{l=1}^{k}\left[I(l \in Z_i \wedge l \notin Y_i) + I(l \notin Z_i \wedge l \in Y_i)\right]$$

Different from the Hamming score, the Hamming loss measures the relevance of an example to a class label that is incorrectly predicted and then averages that measure. Therefore, we are able to capture two kinds of errors: in the first term of the sum we are measuring the prediction error where we predicted an incorrect label, and for the second term we are measuring the missing error where a relevant label was not predicted. This is similar to an exclusive or (XOR) operation. We sum over the number of examples n and the number of classes k and normalize the double sum by those two numbers. If we only had one class, this would simplify to essentially 1 – accuracy for binary classification. Since this is a loss, the smaller the value, the better.

We also have multilabel forms of precision, recall, and F1 score. For precision, we average the ratio of predicted correct labels to the total number of actual labels.

$$precision = \frac{1}{n}\sum_{i=1}^{n} \frac{|Y_i \cap Z_i|}{|Z_i|}$$

Similarly for recall, where instead we average the ratio of predicted correct labels to the total number of predicted labels:

$$recall = \frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap Z_i|}{|Z_i|}$$

For F1 score, it is similar to before as the harmonic mean of precision and recall:

$$F_1 = \frac{1}{n}\sum_{i=1}^{n}\frac{2|Y_i \cap Z_i|}{|Y_i| + |Z_i|}$$

Of course we can also calculate the AUC of a ROC curve or precision-recall curve using the macro version, where we calculate the AUCs per class and then average them.

Metrics for Regression

For image regression problems, there are also evaluation metrics that we can use to see how well our model is performing on data outside of training. For all of the following regression metrics, our goal is to minimize them as much as possible.

The most well-known and standard metric is *mean squared error* (MSE):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \widehat{Y}_i\right)^2$$

MSE, as its name suggests, is the mean of the squared error between the predicted and actual continuous labels. This is a mean-unbiased estimator which has great sensitivity due to the quadratic term, but this sensitivity means a few outliers can unduly influence it.

The *root mean squared error* (RMSE), which is just the square root of the mean squared error, is also used:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \widehat{Y}_i\right)^2}$$

A slightly simpler and more interpretable metric is the *mean absolute error* (MAE):

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|Y_i - \widehat{Y}_i\right|$$

The MAE is just the absolute difference between continuous predictions and labels. Compared to the MSE/RMSE with their squared exponents, the MAE is not as prone to being skewed by a few outliers. Also, unlike MSE, which is a mean-unbiased estimator, where the estimator's sample mean is the same as the distributional mean, MAE is instead a median-unbiased estimator, where the estimator overestimates as frequently as it underestimates.

In an effort to make regression more robust, we can also try the Huber loss metric. This is also less sensitive to outliers than having a squared error loss:

$$HL_{\delta}\left(Y, \widehat{Y}\right) = \frac{1}{n}\sum_{i=1}^{n}\begin{cases}\frac{1}{2}\left(Y_i - \widehat{Y}_i\right)^2 & for\left|Y_i - \widehat{Y}_i\right| \leq \delta \\ \delta\left|Y_i - \widehat{Y}_i\right| - \frac{1}{2}\delta^2 & otherwise\end{cases}$$

As you can see, we get the best of both worlds with this metric. We declare a constant threshold, δ; if the absolute residual is less than this value we use the squared term, and otherwise we use the linear term. This way we can benefit from the sensitivity of the squared mean–unbiased estimator of the quadratic term for values close to zero and the robustness of the median-unbiased estimator of the linear term for values further from zero.

Metrics for Object Detection

Essentially, most of the usual object detection evaluation metrics are the same as the classification metrics. However, instead of comparing predicted and actual labels for an entire image, we are comparing the objects detected versus the objects that are actually there using bounding boxes, as we saw in Chapter 4.

One of the most common object detection metrics is the intersection over union:

$$IOU = \frac{area\left(\widehat{B} \cap B\right)}{area\left(\widehat{B} \cup B\right)}$$

The numerator is the area of the intersection of our predicted bounding box and the actual bounding box. The denominator is the union of our predicted bounding box and the actual bounding box. We can see this graphically in Figure 8-10.

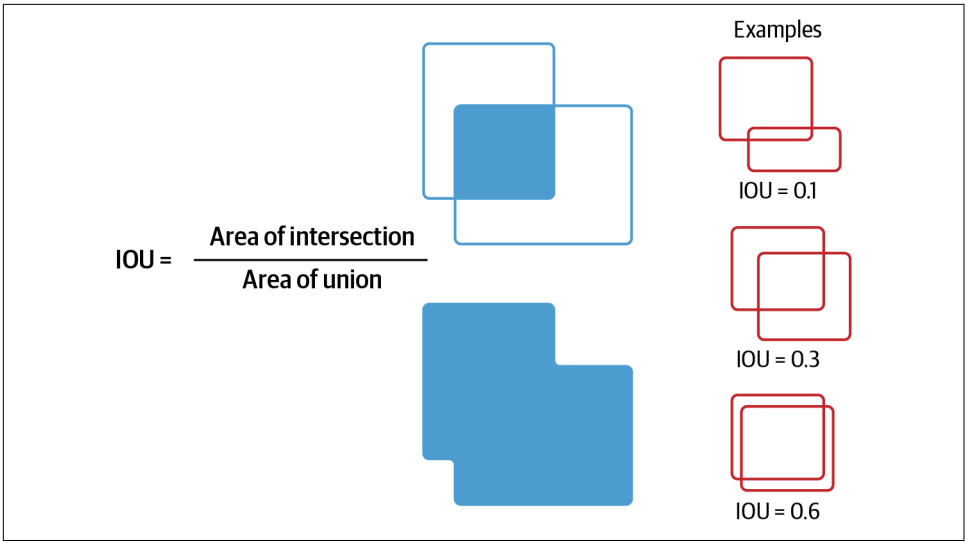

Figure 8-10. Intersection over union is the area of overlap divided by the area of union.

With perfect overlap, the two areas will be equal and thus the IOU will be 1. With no overlap, there will be 0 in the numerator and therefore the IOU will be 0. Thus, the bounds of IOU are [0, 1].

We can also use a form of the classification confusion metrics, such as true positives. As with classification, calculating these requires a threshold, but instead of thresholding a predicted probability, we threshold the IOU. In other words, if a bounding box's IOU is over a certain value, then we declare that that object has been detected. Threshold values are typically 50, 75, or 95%.

A true positive in this case would be considered a correct detection. This occurs when the predicted and actual bounding boxes have an IOU greater than or equal to the threshold. A false positive, on the other hand, would be considered a wrong detection. This occurs when the predicted and actual bounding boxes have an IOU less than the threshold. A false negative would be considered a missed detection, where an actual bounding box was not detected at all.

Lastly, true negatives don't apply for object detection. A true negative is a correct missed detection. If we remember our per-class multiclass confusion matrices, the true negative was the sum of all of the other cells not used in the other three confusion metrics. Here, the true negatives would be all of the bounding boxes that we could have placed on the image and not triggered one of the other three confusion metrics. Even for small images the number of permutations of these kinds of not-placed bounding boxes would be enormous, so it doesn't make sense to use this confusion metric.

Precision in this case equals the number of true positives divided by the number of all detections. This measures the model's ability to identify only the relevant objects within the image:

$$precision = \frac{TP}{alldetections}$$

In object detection, recall measures the model's ability to find all of the relevant objects within the image. Therefore, it equals the number of true positives divided by the number of all actual bounding boxes:

$$recall = \frac{TP}{allactualboundingboxes}$$

Just like with classification, these composite metrics can be used to create curves using different threshold values. Some of the most common are precision-recall curves (like the ones we've seen before) and recall-IOU curves, which typically plot IOU in the range [0.5, 1.0].

We can also calculate the average precision and average recall using the precision-recall and recall-IOU curves. In order to smooth out any perturbations in the curve, we typically interpolate the precision at multiple recall levels before performing the actual average precision calculation, as shown in Figure 8-11.

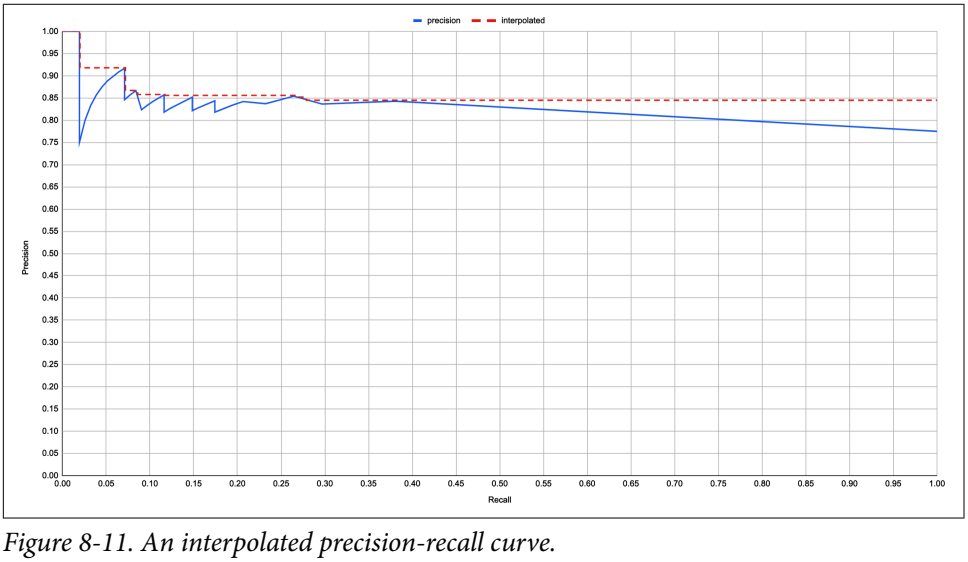

Figure 8-11. An interpolated precision-recall curve.

We do something similar for average recall. In the formula, the interpolated precision at a chosen recall level *r* is the maximum of the precision *p* found for any recall level *r'* that is greater than or equal to *r*:

$$pinterpolated = max_{r' \geq r}[p(r')]$$

The traditional interpolation method is to choose 11 equally spaced recall levels; however, more recently practitioners have been experimenting with choosing all unique recall levels for interpolation. The average precision is thus the area under the interpolated precision-recall curve:

$$AP = \frac{1}{n}\sum_{i=1}^{n-1}(r_{i+1} - r_i)pinterpolated(r_{i+1})$$

This is the end of the story for precision if we only have one class, but often in object detection we have many classes, all of which have different detection performances. Therefore, it can be useful to calculate the *mean average precision* (mAP), which is just the mean of each class's average precision:

$$mAP = \frac{1}{k}\sum_{l=1}^{k} AP_l$$

To calculate average recall, as mentioned previously, we use the recall-IOU curve instead of the precision-recall curve used for average precision. It is essentially the recall averaged over all IOUs (specifically, IOUs that are at least 50%) and thus becomes two times the area under the recall-IOU curve:

$$AR = 2\int_{0.5}^{1} recall(u)du$$

As we did for the multiclass objection detection case for average precision, we can find the *mean average recall* (mAR) by averaging the average recalls across all classes:

$$mAR = \frac{1}{k}\sum_{l=1}^{k} AR_l$$

For instance segmentation tasks, the metrics are exactly the same as for detection. IOU can equally well be defined for boxes or masks.

Now that we have explored the available evaluation metrics for models, let's look at how we use them for understanding model bias and for continuous evaluation.

Quality Evaluation

The evaluation metrics computed on the validation dataset during training are computed in aggregate. Such aggregate metrics miss a number of subtleties that are needed to truly gauge a model's quality. Let's take a look at sliced evaluations, a technique to catch these subtleties, and how to use sliced evaluations to identify bias in a model.

Sliced Evaluations

Evaluation metrics are usually calculated based on a holdout dataset that is similar to the training dataset in distribution. This typically gives us a good overall view of model health and quality. However, the model may perform much worse on some slices of the data than others, and these deficiencies can be lost in the ocean of calculating on the entire dataset.

Therefore, it can often be a good idea to analyze model quality on a more granular level. We can do this by taking slices of the data based on classes or other separating characteristics and calculating the usual evaluation metrics on each one of those subsets. Of course, we should still calculate the evaluation metrics using all of the data so that we can see how individual subsets vary from the superset. You can see an example of these sliced evaluation metrics in Figure 8-12.

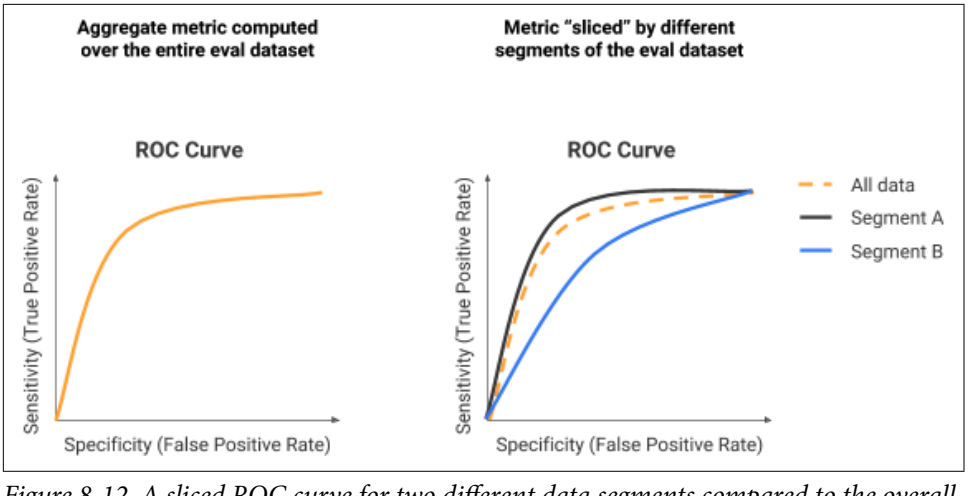

Figure 8-12. A sliced ROC curve for two different data segments compared to the overall ROC curve.

Some use cases place special importance on certain segments of the data, so these are prime targets to apply sliced evaluation metrics to in order to keep a close eye on them.

This doesn't just have to be a passive monitoring exercise, though! Once we know the sliced evaluation metrics, we can then make adjustments to our data or model to bring each of the sliced metrics in line with our expectations. This could be as simple as augmenting the data more for a particular class or adding some more complexity to the model to be able to understand those problematic slices better.

Next, we'll look at a specific example of a segment that we may need to do sliced evaluations on.

Fairness Monitoring

Image ML models have been shown to perform poorly on some segments of the population. For example, a 2018 study showed that commercially available facial analysis programs had dramatically higher error rates at identifying the gender of darker-skinned women as compared to lighter-skinned men. In 2020, many Twitter users reported that Twitter's photo preview feature appears to favor white faces over Black faces. Meanwhile, Zoom's facial recognition appeared to remove Black faces when using a virtual background. And in 2015, Google Photos mistakenly labeled a selfie of a Black couple as being an image of gorillas.

Considering these high-profile and distressing mistakes by highly capable engineering teams, it is clear that if our computer vision problems involve human subjects, we should attempt to safeguard against such errors by carrying out sliced evaluations where the segments consist of individuals belonging to different races and genders. This will allow us to diagnose whether there is a problem.

Poor model performance on subjects of different genders and races cannot be addressed simply by ensuring that all races and genders are present in the training and evaluation datasets. There may be deeper problems. Photographic filters and processing techniques were historically optimized to best represent lighter skin tones, and this causes problems with lighting effects on darker-toned individuals. Therefore, preprocessing and data augmentation methods may have to be incorporated into our model-training pipelines in order to correct for this effect. Also, ML model training focuses initially on common cases, and only later on rarer examples. This means that techniques such as early stopping, pruning, and quantization might amplify biases against minorities. It is not, in other words, "just" a data problem. Addressing fairness issues requires examination of the entire machine learning pipeline.

Sliced evaluations are an invaluable tool to diagnose whether such biases exist in the models that have been trained. This means that we should perform these evaluations for any segment of the population that we are concerned might be treated unfairly.

Continuous Evaluation

How often should we carry out sliced evaluations? It's important to constantly be evaluating our models even after we deploy them. This can help us catch things that could be going wrong early. For instance, we might have prediction drift because the inference input distribution is slowly shifting over time. There also could be a sudden event that causes a major change in the data, which in turn causes the model's behavior to change.

Continuous evaluation typically consists of seven steps:

1. Randomly sample and save the data being sent for model predictions. For example, we might choose to save 1% of all the images sent to the deployed model.
2. Carry out predictions with the model as usual and send them back to the client—but make sure to also save the model's prediction for each of the sampled images.
3. Send the samples for labeling. We can use the same labeling approach as was used for the training data—for example, we can use a labeling service, or label the data a few days later based on the eventual outcome.
4. Compute evaluation metrics over the sampled data, including sliced evaluation metrics.
5. Plot moving averages of the evaluation metrics. For example, we might plot the average Hubert loss over the past seven days.
6. Look for changes in the averaged evaluation metrics over time, or specific thresholds that are exceeded. We might choose to send out an alert, for example, if the accuracy for any monitored segment drops below 95% or if the accuracy this week is more than 1% lower than the accuracy the previous week.
7. We might also choose to periodically retrain or fine-tune the model after adding the sampled and subsequently labeled data to the training dataset.

When to retrain is a decision that we need to make. Some common choices include retraining whenever the evaluation metric falls below a certain threshold, retraining every *X* days, or retraining once we have *X* new labeled examples.

Whether to train from scratch or just fine tune is another decision that we need to make. The typical choice is to fine tune the model if the new samples are a small fraction of the original training data and to train from scratch once the sampled data starts to approach about 10% of the number of examples in the original dataset.

Summary

In this chapter, we discussed the importance of monitoring our models during training. We can use the amazing graphical UI of TensorBoard to watch our loss and other metrics throughout training, and to verify that the model is converging and getting better over time. Additionally, since we don't want to overtrain our models, by creating checkpoints and enabling early stopping we can halt training at the best moment.

We also discussed many quality metrics that we can use to evaluate our models on unseen data to get a better measure of how well they're doing. There are different metrics for image classification, image regression, and object detection, although some of them resurface in slightly different forms among the various problem types. In fact, image classification has three different subfamilies of classification metrics, depending on both the number of classes and the number of labels per image.

Finally, we looked at performing sliced evaluations on subsets of our data to not only be aware of our model's gaps but also to help us brainstorm fixes to close those gaps. This practice can help us monitor for bias, to make sure that we are being as fair as possible and understand the inherent risks of using our model.

CHAPTER 9

Model Predictions

The primary purpose of training machine learning models is to be able to use them to make predictions. In this chapter, we will take a deep dive into several considerations and design choices involved in deploying trained ML models and using them to make predictions.

The code for this chapter is in the *09_deploying* folder of the book's GitHub repository. We will provide file names for code samples and notebooks where applicable.

Making Predictions

To *invoke* a trained model—i.e., to use it to make predictions—we have to load the model from the directory into which it was exported and call the serving signature. In this section, we will look at how to do this. We will also look at how to improve the maintainability and performance of invoked models.

Exporting the Model

To obtain a serving signature to invoke, we must export our trained model. Let's quickly recap these two topics—exporting and model signatures—which were covered in much greater detail in "Saving Model State" on page 253 in Chapter 7. Recall that a Keras model can be exported (see the notebook *07c_export.ipynb* on GitHub) using code like this:

```
model.save('gs://practical-ml-vision-book/flowers_5_trained')
```

This saves the model in TensorFlow SavedModel format. We discussed examining the signature of the prediction function using the command-line tool `saved_model_cli`.

By default the signature matches the input layer of the Keras model that was saved, but it is possible to export the model with a different function by explicitly specifying it (see Figure 9-1):

```
model.save('export/flowers_model',
           signatures={
               'serving_default': predict_flower_type
           })
```

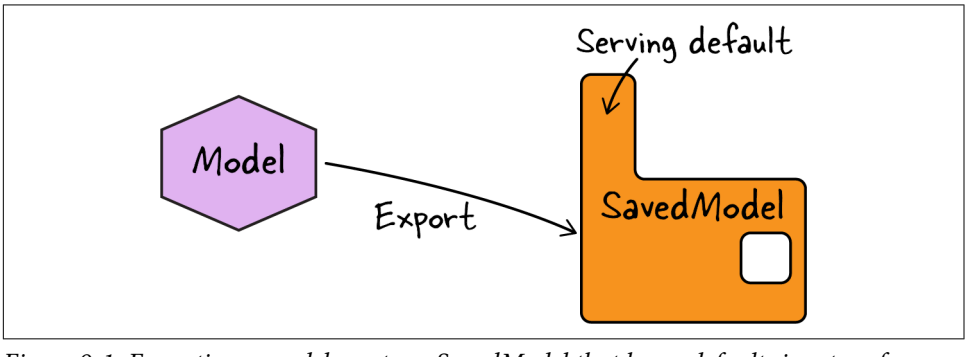

Figure 9-1. Exporting a model creates a SavedModel that has a default signature for serving predictions. In this case, the model on the left is the Python object in memory, and the SavedModel is what is persisted to disk.

The `predict_flower_type()` function carries a `@tf.function` annotation, as explained in "Signature of a TensorFlow Function" on page 254 in Chapter 7:

```
@tf.function(input_signature=[tf.TensorSpec([None,], dtype=tf.string)])
def predict_flower_type(filenames):
    ...
```

Suppose, for the examples in the first part of this chapter, that we have exported the model with the `predict_flower_type()` function as its default serving function.

Using In-Memory Models

Imagine we are programming a client that needs to call this model and obtain predictions from it for some input. The client could be a Python program from which we wish to invoke the model. We would then load the model into our program and obtain the default serving function as follows (full code in *09a_inmemory.ipynb* on GitHub):

```
serving_fn = tf.keras.models.load_model(MODEL_LOCATION
                                        ).signatures['serving_default']
```

If we pass in a set of filenames to the serving function, we obtain the corresponding predictions:

```
filenames = [
    'gs://.../9818247_e2eac18894.jpg',
     ...
    'gs://.../8713397358_0505cc0176_n.jpg'
]
pred = serving_fn(tf.convert_to_tensor(filenames))
```

The result is a dictionary. The maximum likelihood prediction can be obtained from the tensor by looking in the dictionary for the specific key and calling `.numpy()`:

```
pred['flower_type_str'].numpy()
```

In this prediction situation, the model was loaded and invoked directly within the client program (see Figure 9-2). The input to the model had to be a tensor, and so the client program had to create a tensor out of the filename strings. Because the output of the model was also a tensor, the client program had to obtain a normal Python object using `.numpy()`.

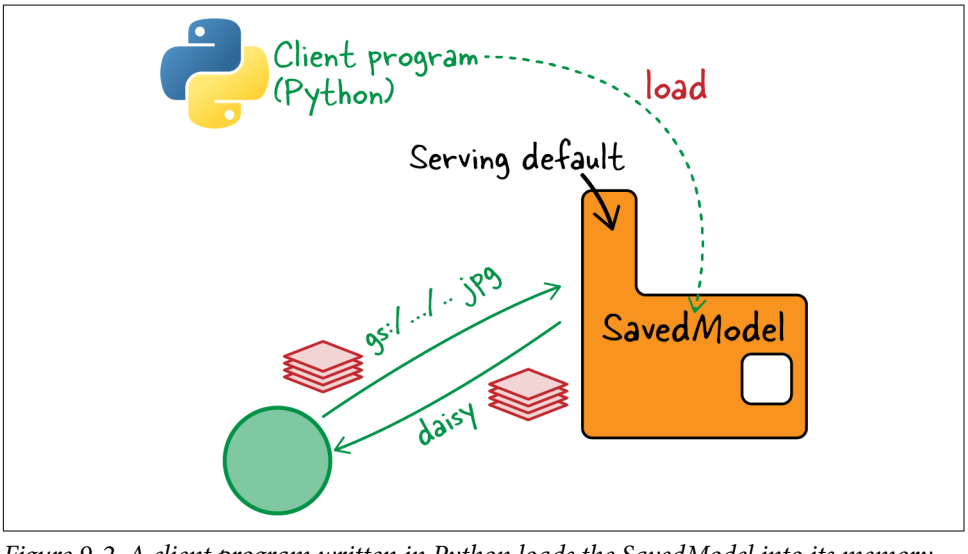

Figure 9-2. A client program written in Python loads the SavedModel into its memory, sends a tensor containing filenames to the in-memory model, and receives a tensor containing the predicted labels.

A few input images and their predictions are shown in Figure 9-3. Note that because of the care we took in Chapters 5 and 7 to replicate the preprocessing operations in the serving function, clients can send us images of any size—the server will resize the images to what the model requires.

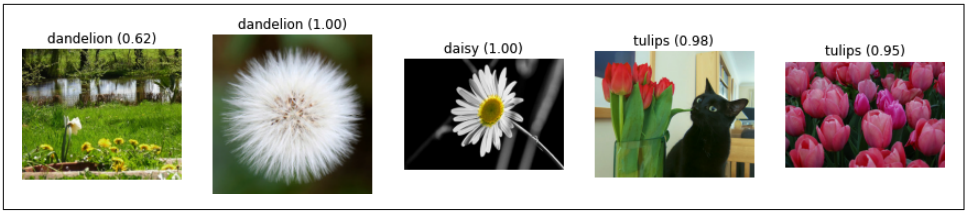

Figure 9-3. A selection of images and their corresponding predictions.

There are, nevertheless, two key problems with this in-memory approach: abstraction and performance. Let's look at what these problems are and how to address them.

Improving Abstraction

It is usually the case that the machine learning engineers and data scientists who develop an ML model have different tools and skills at their disposal than the application developers who are integrating the ML predictions into user-facing applications. You want the ML prediction API to be such that it can be used by someone without any knowledge of TensorFlow or programming in React, Swift, or Kotlin. This is why abstraction is necessary.

We have abstracted away the model's details to some extent—the client doesn't need to know the required size of the images (indeed, note in Figure 9-3 that the images are all of different sizes) or the architecture of the ML model being used for classification. However, the abstraction is not complete. We do have some requirements for the client programmer:

- The client machine will need to have the TensorFlow libraries installed.
- At the time of writing, TensorFlow APIs are callable only from Python, C, Java, Go, and JavaScript. Therefore, the client will have to be written in one of those languages.
- Because the client programmer has to call functions like `tf.convert_to_tensor()` and `.numpy()`, they must understand concepts like tensor shapes and eager execution.

To improve the abstraction, it would be better if we could invoke the model using a protocol such as HTTPS that can be used from many languages and environments. Also, it would be better if we could supply the inputs in a generic format such as JSON, and obtain the results in the same format.

Improving Efficiency

In the in-memory approach, the model is loaded and invoked directly within the client program. So, the client will need:

- Considerable on-board memory, since image models tend to be quite large
- Accelerators such as GPUs or TPUs, as otherwise the computation will be quite slow

As long as we make sure to run the client code on machines with enough memory and with accelerators attached, are we OK? Not quite.

Performance problems tend to manifest themselves in four scenarios:

Online prediction
: We may have many concurrent clients that need the predictions in near real time. This is the case if we are building interactive tools, such as one that offers the ability to load product photographs onto an ecommerce website. Since there may be many thousands of simultaneous users, we need to ensure that the predictions are carried out at a low latency for all these concurrent users.

Batch prediction
: We might need to carry out inference on a large dataset of images. If each image takes 300 ms to process, the inference on 10,000 images will take nearly an hour. We might need the results faster.

Stream prediction
: We might need to carry out inference on images as they stream into our system. If we receive around 10 images a second, and it takes 100 ms to process each image, we will barely be able to keep up with the incoming stream, so any traffic spikes will cause the system to start falling behind.

Edge prediction
: Low-connectivity clients might need the predictions in near real time. For example, we might need to identify defects in the parts on a factory conveyor belt even as it is moving. For this to happen, we need the image of the belt to get processed as quickly as possible. We may not have the network bandwidth to send that image to a powerful machine in the cloud and get the results back within the time budget imposed by the moving conveyor belt. This is also the situation in cases where an app on a mobile phone needs to make a decision based on what the phone camera is being pointed at. Because the factory or mobile phone sits on the edge of the network, where network bandwidth isn't as high as it would be between two machines in a cloud data center, this is called *edge prediction*.

In the following sections, we'll dive into each of these scenarios and look at techniques for dealing with them.

Online Prediction

For online prediction, we require a microservices architecture—model inference will need to be carried out on powerful servers with accelerators attached. Clients will request model inference by sending HTTP requests and receiving HTTP responses. Using accelerators and autoscaling infrastructure addresses the performance problem, while using HTTP requests and responses addresses the abstraction problem.

TensorFlow Serving

The recommended approach for online prediction is to deploy the model using TensorFlow Serving as a web microservice that responds to POST requests. The request and response will not be tensors, but abstracted into a web-native message format such as JSON.

Deploying the model

TensorFlow Serving is just software, so we also need some infrastructure. User requests will have to be dynamically routed to different servers, which will need to autoscale to deal with traffic peaks. You can run TensorFlow Serving on managed services like Google Cloud's Vertex AI, Amazon SageMaker, or Azure ML (see Figure 9-4). Acceleration on these platforms is available both via GPUs and through custom-built accelerators like AWS Inferentia and Azure FPGA. Although you can install the TensorFlow Serving module or Docker container into your favorite web application framework, we don't recommend this approach since you won't get the benefits of the optimized ML serving systems and infrastructure management that the cloud providers' ML platforms offer.

To deploy the SavedModel as a web service on Google Cloud, we'd point `gcloud` at the Google Cloud Storage location to which the model was exported and deploy the resulting model to a Vertex AI endpoint. Please see the code in GitHub for details.

When deploying the model, we can also specify the machine type, type of accelerators, and minimum and maximum replica counts.

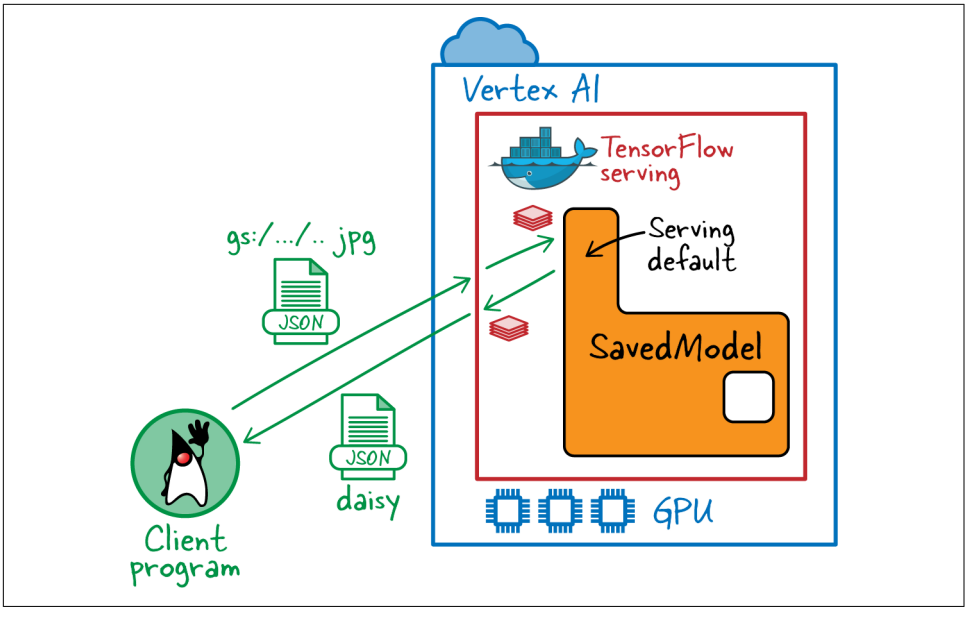

Figure 9-4. Online model predictions served through a REST API.

Making predictions

Predictions can be obtained from any machine that is capable of making an HTTPS call to the server on which the model is deployed (see Figure 9-4). The data is sent back and forth as JSON messages, and TensorFlow Serving converts the JSON into tensors to send to the SavedModel.

We can try out the deployed model by creating a JSON request:

```
{
    "instances": [
        {
            "filenames": "gs://.../9818247_e2eac18894.jpg"
        },
        {
            "filenames": "gs://.../9853885425_4a82356f1d_m.jpg"
        },
     ]
}
```

and sending it to the server using `gcloud`:

```
gcloud ai endpoints predict ${ENDPOINT_ID} \
    --region=${REGION} \
    --json-request=request.json
```

One key thing to note is that the JSON request consists of a set of instances, each of which is a dictionary. The items in the dictionary correspond to the inputs specified

in the model signature. We can view the model signature by running the command-line tool `saved_model_cli` on the SavedModel:

```
saved_model_cli show --tag_set serve \
    --signature_def serving_default --dir ${MODEL_LOCATION}
```

In the case of the flowers model, this returns:

```
inputs['filenames'] tensor_info:
    dtype: DT_STRING
    shape: (-1)
    name: serving_default_filenames:0
```

That's how we knew that each instance in the JSON needed a string element called `filenames`.

Because this is just a REST API, it can be invoked from any programming language that is capable of sending an HTTPS POST request. Here's how to do it in Python:

```
api = ('https://{}-aiplatform.googleapis.com/v1/projects/' +
       '{}/locations/{}/endpoints/{}:predict'.format(
       REGION, PROJECT, REGION, ENDPOINT_ID))
```

The header contains the client's authentication token. This can be retrieved programmatically using:

```
token = (GoogleCredentials.get_application_default()
         .get_access_token().access_token)
```

We have seen how to deploy the model and obtain predictions from it, but the API is whatever signature the model was exported with. Next, let's look at how to change this.

Modifying the Serving Function

Currently, the flowers model has been exported so that it takes a filename as input and returns a dictionary consisting of the most likely class (e.g., daisy), the index of this class (e.g., 2), and the probability associated with this class (e.g., 0.3). Suppose we wish to change the signature so that we also return the filename that the prediction is associated with.

This sort of scenario is quite common because it is impossible to anticipate the exact signature that we will need in production when a model is exported. In this case, we want to pass input parameters from the client through to the response. A need for such *pass-through parameters* is quite common, and different clients will want to pass through different things.

While it is possible to go back, change the trainer program, retrain the model, and re-export the model with the desired signature, it is more convenient to simply change the signature of the exported model.

Changing the default signature

To change the signature, first we load the exported model:

```
model = tf.keras.models.load_model(MODEL_LOCATION)
```

Then we define a function with the desired new signature, making sure to invoke the old signature on the model from within the new function:

```
@tf.function(input_signature=[tf.TensorSpec([None,], dtype=tf.string)])
def pass_through_input(filenames):
    old_fn = model.signatures['serving_default']
    result = old_fn(filenames) # has flower_type_int etc.
    result['filename'] = filenames # pass through
    return result
```

If the client instead wanted to supply a sequence number and asked us to pass this through in the response, we could do that as follows:

```
@tf.function(input_signature=[tf.TensorSpec([None,], dtype=tf.string),
                              tf.TensorSpec([], dtype=tf.int64)])
def pass_through_input(filenames, sequenceNumber):
    old_fn = model.signatures['serving_default']
    result = old_fn(filenames) # has flower_type_int etc.
    result['filename'] = filenames # pass through
    result['sequenceNumber'] = sequenceNumber # pass through
    return result
```

Finally, we export the model with our new function as the serving default:

```
model.save(NEW_MODEL_LOCATION,
           signatures={
               'serving_default': pass_through_input
           })
```

We can verify the resulting signature using `saved_model_cli` and ensure that the filename is included in the output:

```
outputs['filename'] tensor_info:
      dtype: DT_STRING
      shape: (-1)
      name: StatefulPartitionedCall:0
```

Multiple signatures

What if you have multiple clients, and each of them wants a different signature? TensorFlow Serving allows you to have multiple signatures in a model (although only one of them will be the serving default).

For example, suppose we want to support both the original signature and the pass-through version. In this case, we can export the model with two signatures (see Figure 9-5):

```
model.save('export/flowers_model2',
           signatures={
               'serving_default': old_fn,
               'input_pass_through': pass_through_input
           })
```

where `old_fn` is the original serving signature that is obtained via:

```
model = tf.keras.models.load_model(MODEL_LOCATION)
old_fn = model.signatures['serving_default']
```

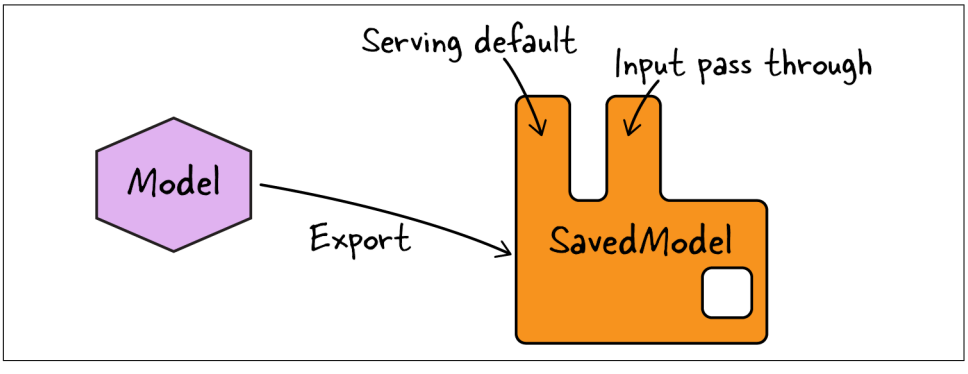

Figure 9-5. Exporting a model with multiple signatures.

Clients who wish to invoke the nondefault serving signature will have to specifically include a signature name in their requests:

```
{
    "signature_name": "input_pass_through",
    "instances": [
        {
            "filenames": "gs://.../9818247_e2eac18894.jpg"
        },
        ...
}
```

Others will get the response corresponding to the default serving function.

Handling Image Bytes

We have, so far, been sending a filename to the service and asking for the classification result. This works well for images that have been uploaded to the cloud already, but can introduce friction if that's not the case. If the image is not already in the cloud, it would be ideal for the client code to send us the JPEG bytes corresponding to the file contents. That way, we can avoid an intermediate step of uploading the image data to the cloud before invoking the prediction model.

Loading the model

To change the model in this situation, we could load the exported model and change the input signature to be:

```
@tf.function(input_signature=[tf.TensorSpec([None,], dtype=tf.string)])
def predict_bytes(img_bytes):
```

But what would this implementation do? In order to invoke the existing model signature, we will need the user's file to be available to the server. So, we'd have to take the incoming image bytes, write them to a temporary Cloud Storage location, and then send it to the model. The model would then read this temporary file back into memory. This is pretty wasteful—how can we get the model to directly use the bytes we are sending it?

To do this, we need to decode the JPEG bytes, preprocess them the same way that we did during model training, and then invoke `model.predict()`. For that, we need to load the last (or best) checkpoint saved during model training:

```
CHECK_POINT_DIR='gs://.../chkpts'
model = tf.keras.models.load_model(CHECK_POINT_DIR)
```

We can also load the exported model using the same API:

```
EXPORT_DIR='gs://.../export'
model = tf.keras.models.load_model(EXPORT_DIR)
```

Adding a prediction signature

Having loaded the model, we use this model to implement the prediction function:

```
@tf.function(input_signature=[tf.TensorSpec([None,], dtype=tf.string)])
def predict_bytes(img_bytes):
    input_images = tf.map_fn(
        preprocess, # preprocessing function used in training
        img_bytes,
        fn_output_signature=tf.float32
    )
    batch_pred = model(input_images) # same as model.predict()
    top_prob = tf.math.reduce_max(batch_pred, axis=[1])
    pred_label_index = tf.math.argmax(batch_pred, axis=1)
    pred_label = tf.gather(tf.convert_to_tensor(CLASS_NAMES),
                           pred_label_index)
    return {
        'probability': top_prob,
        'flower_type_int': pred_label_index,
        'flower_type_str': pred_label
    }
```

In that code snippet, note that we need to get access to the preprocessing functions used in training, perhaps by importing a Python module. The preprocessing function has to be the same as what was used in training:

```
def preprocess(img_bytes):
    img = tf.image.decode_jpeg(img_bytes, channels=IMG_CHANNELS)
    img = tf.image.convert_image_dtype(img, tf.float32)
    return tf.image.resize_with_pad(img, IMG_HEIGHT, IMG_WIDTH)
```

We might as well also implement another method to predict from the filename:

```
@tf.function(input_signature=[tf.TensorSpec([None,], dtype=tf.string)])
def predict_filename(filenames):
    img_bytes = tf.map_fn(
        tf.io.read_file,
        filenames
    )
    result = predict_bytes(img_bytes)
    result['filename'] = filenames
    return result
```

This function simply reads in the file (using `tf.io.read_file()`) and then invokes the other prediction method.

Exporting signatures

Both of these functions can be exported, so that clients have the choice of supplying either the filename or the byte contents:

```
model.save('export/flowers_model3',
           signatures={
               'serving_default': predict_filename,
               'from_bytes': predict_bytes
           })
```

Base64 encoding

In order to provide the contents of a local image file to the web service, we read the file contents into memory and send them over the wire. Because it is very possible that the JPEG files will contain special characters that will confuse the JSON parser on the server side, it is necessary to base64-encode the file contents before sending them (the full code is available in *09d_bytes.ipynb* on GitHub):

```
def b64encode(filename):
    with open(filename, 'rb') as ifp:
        img_bytes = ifp.read()
        return base64.b64encode(img_bytes)
```

The base64-encoded data can then be incorporated into the JSON message that is sent as follows:

```
data = {
    "signature_name": "from_bytes",
    "instances": [
        {
            "img_bytes": {"b64": b64encode('/tmp/test1.jpg')}
        },
```

```
            {
                "img_bytes": {"b64": b64encode('/tmp/test2.jpg')}
            },
        ]
    }
```

Note the use of the special `b64` element to denote base64 encoding. TensorFlow Serving understands this and decodes the data on the other end.

Batch and Stream Prediction

Doing batch prediction one image at a time is unacceptably slow. A better solution is to carry out the predictions in parallel. Batch prediction is an embarrassingly parallel problem—predictions on two images can be performed entirely in parallel because there is no data to transfer between the two prediction routines. However, attempts to parallelize the batch prediction code on a single machine with many GPUs often run into memory issues because each of the threads will need to have its own copy of the model. Using Apache Beam, Apache Spark, or any other big data processing technology that allows us to distribute data processing across many machines is a good way to improve batch prediction performance.

We also need multiple machines for streaming prediction (such as in response to a clickstream of events through Apache Kafka, Amazon Kinesis, or Google Cloud Pub/Sub), for the same reasons that we require them for batch prediction—to carry out inference on the images as they arrive in parallel without causing out-of-memory problems. However, because streaming workloads tend to be spiky, we also require this infrastructure to autoscale—we should provision more machines at traffic peaks and scale down to a minimal number of machines at traffic lows. Apache Beam on Cloud Dataflow provides this capability. Therefore, we suggest using Beam for improving the performance of streaming prediction. Happily, the same code that is used for batch prediction in Beam will also work unchanged for streaming prediction.

The Apache Beam Pipeline

The solution to both batch and streaming prediction involves Apache Beam. We can write a Beam transform to carry out inference as part of the pipeline:

```
    | 'pred' >> beam.Map(ModelPredict(MODEL_LOCATION))
```

We can reuse the model prediction code that we used in in-memory prediction by loading the serving function from the exported model:

```
class ModelPredict:
    def __init__(self, model_location):
        self._model_location = model_location
```

```
    def __call__(self, filename):
        serving_fn = (tf.keras.models.load_model(self._model_location)
                      .signatures['serving_default'])
        result = serving_fn(tf.convert_to_tensor([filename]))
        return {
            'filenames': filename,
            'probability': result['probability'].numpy()[0],
            'pred_label': result['flower_type_str'].numpy()[0]
        }
```

However, there are two issues with this code. First, we are processing the files one at a time. TensorFlow graph operations are faster if we can carry them out in batches, so we'll want to batch up the filenames. Second, we are loading the model for each element. Ideally, we'd load the model once and reuse it. Because Beam is a distributed system, however, we actually have to load the model once *on each worker* (see Figure 9-6). To do that, we must use a shared *handle* (essentially a shared connection to the service) that is acquired by each worker. This handle has to be acquired through a weak reference so that if a worker is decommissioned (due to low traffic) and then reactivated (due to a traffic peak), Beam does the right thing and reloads the model in that worker.

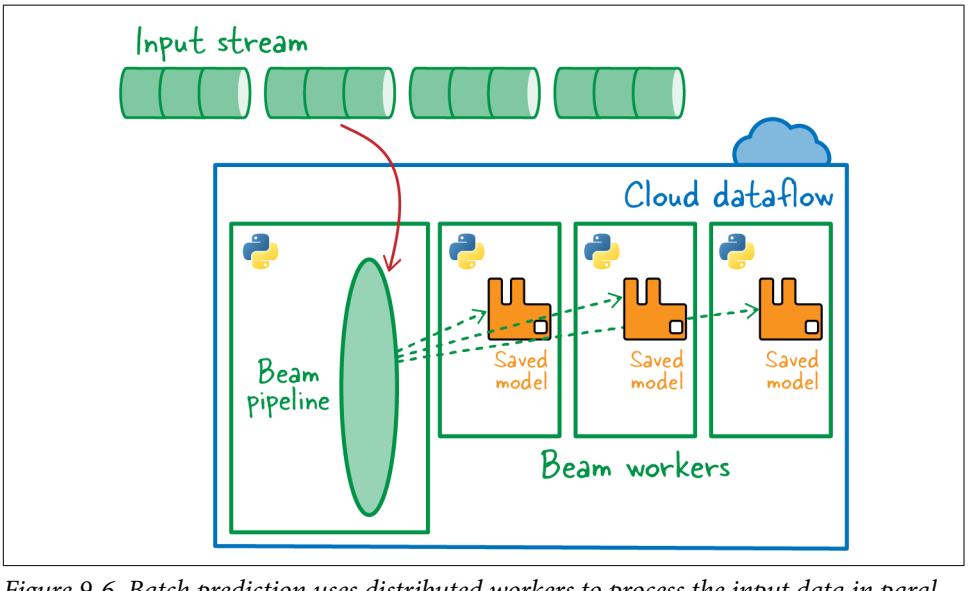

Figure 9-6. Batch prediction uses distributed workers to process the input data in parallel. This architecture also works for stream prediction.

To use the shared handle, we modify the model prediction code as follows:

```
class ModelPredict:
    def __init__(self, shared_handle, model_location):
        self._shared_handle = shared_handle
```

```
        self._model_location = model_location

    def __call__(self, filenames):
        def initialize_model():
            logging.info('Loading Keras model from ' +
                         self._model_location)
            return (tf.keras.models.load_model(self._model_location)
                    .signatures['serving_default'])

        serving_fn = self._shared_handle.acquire(initialize_model)
        result = serving_fn(tf.convert_to_tensor(filenames))
        return {
            'filenames': filenames,
            'probability': result['probability'].numpy(),
            'pred_label': result['flower_type_str'].numpy()
        }
```

The shared handle, whose capability is provided by Apache Beam, ensures that connections are reused within a worker and reacquired after passivation. In the pipeline, we create the shared handle and make sure to batch the elements before calling model prediction (you can see the full code in *09a_inmemory.ipynb* on GitHub):

```
with beam.Pipeline() as p:

    shared_handle = Shared()

    (p
     | ...
     | 'batch' >> beam.BatchElements(
                    min_batch_size=1, max_batch_size=32)
     | 'addpred' >> beam.Map(
                    ModelPredict(shared_handle, MODEL_LOCATION) )
    )
```

The same code works for both batch and streaming predictions.

If you are grouping the images, then the groups are already a batch of images and so there is no need to explicitly batch them:

```
    | 'groupbykey' >> beam.GroupByKey()  # (usr, [files])
    | 'addpred' >> beam.Map(lambda x:
                           ModelPredict(shared_handle,
                                        MODEL_LOCATION)(x[1]))
```

We can run the Apache Beam code at scale using Cloud Dataflow.

Managed Service for Batch Prediction

If we have deployed the model as a web service to support online prediction, then an alternative to using the Beam on Dataflow batch pipeline is to also use Vertex AI to carry out batch prediction:

```
gcloud ai custom-jobs create \
    --display_name=flowers_batchpred_$(date -u +%y%m%d_%H%M%S) \
    --region ${REGION} \
    --project=${PROJECT} \

--worker-pool-spec=machine-type='n1-highmem-2',container-image-uri=${IMAGE}
```

Performance-wise, the best approach depends on the accelerators that are available in your online prediction infrastructure versus what is available in your big data infrastructure. Since online prediction infrastructure can use custom ML chips, this approach tends to be better. Also, Vertex AI batch prediction is easier to use because we don't have to write code to handle batched requests.

Invoking Online Prediction

Writing our own batch prediction pipeline in Apache Beam is more flexible because we can do additional transformations in our pipeline. Wouldn't it be great if we could combine the Beam and REST API approaches?

We can do this by invoking the deployed REST endpoint from the Beam pipeline instead of invoking the model that is in memory (the full code is in *09b_rest.ipynb* on GitHub):

```
class ModelPredict:
    def __init__(self, project, model_name, model_version):
        self._api = ('https://ml.googleapis.com/...:predict'
            .format(project, model_name, model_version))

    def __call__(self, filenames):
        token = (GoogleCredentials.get_application_default()
                 .get_access_token().access_token)
        data = {
            "instances": []
        }
        for f in filenames:
            data['instances'].append({
                "filenames" : f
            })
        headers = {'Authorization': 'Bearer ' + token }
        response = requests.post(self._api, json=data, headers=headers)
        response = json.loads(response.content.decode('utf-8'))
        for (a,b) in zip(filenames, response['predictions']):
            result = b
            result['filename'] = a
            yield result
```

If we combine the Beam approach with the REST API approach as shown here, we will be able to support streaming predictions (something the managed service doesn't do). We also gain a couple of performance advantages:

- A deployed online model can be scaled according to the computational needs of the model. Meanwhile, the Beam pipeline can be scaled on the data rate. This ability to independently scale the two parts can lead to cost savings.
- A deployed online model can make more effective use of GPUs because the entire model code is on the TensorFlow graph. Although you can run the Dataflow pipeline on GPUs, GPU use is less effective because the Dataflow pipeline does many other things (like reading data, grouping keys, etc.) that do not benefit from GPU acceleration.

These two performance benefits have to be balanced against the increased networking overhead, however—using an online model adds a network call from the Beam pipeline to the deployed model. Measure the performance to determine whether the in-memory model is better for your needs than the REST model. In practice, we have observed that the larger the model is, and the more instances there are in the batch, the greater are the performance advantages of invoking the online model from Beam rather than hosting the model in memory.

Edge ML

Edge ML is becoming increasingly important because the number of devices with computational capabilities has been growing dramatically in recent years. These include smartphones, connected appliances in homes and factories, and instruments placed outdoors. If these edge devices have a camera, then they are candidates for machine learning use cases on images.

Constraints and Optimizations

Edge devices tend to have a few constraints:

- They may have no connectivity to the internet, and even if they do have connectivity, the connection might be spotty and have low bandwidth. It is, therefore, necessary to carry out ML model inference on the device itself so that we do not wait for the duration of a round trip to the cloud.
- There may be privacy constraints, and it may be desired that image data never leaves the device.
- Edge devices tend to have limited memory, storage, and computing power (at least compared to typical desktop or cloud machines). Therefore, the model inference has to be done in an efficient way.
- The use case often requires that the device have low cost, be small, use very little power, and not get too hot.

For edge prediction, therefore, we need a low-cost, efficient, on-device ML accelerator. In some cases, the accelerator will already be built in. For example, modern mobile phones tend to have an on-board GPU. In other cases, we will have to incorporate an accelerator into the design of the instrument. We can buy edge accelerators to attach or incorporate into instruments (such as cameras and X-ray scanners) when they are being built.

In conjunction with selecting fast hardware, we also need to ensure that we do not overtax the device. We can do that by taking advantage of approaches that reduce the computational requirements of image models so that they operate efficiently on the edge.

TensorFlow Lite

TensorFlow Lite is a software framework for carrying out TensorFlow model inference on edge devices. Note that TensorFlow Lite is not a version of TensorFlow—we cannot train models using TensorFlow Lite. Instead, we train a model using regular TensorFlow, and then convert the SavedModel into an efficient form for use on edge devices (see Figure 9-7).

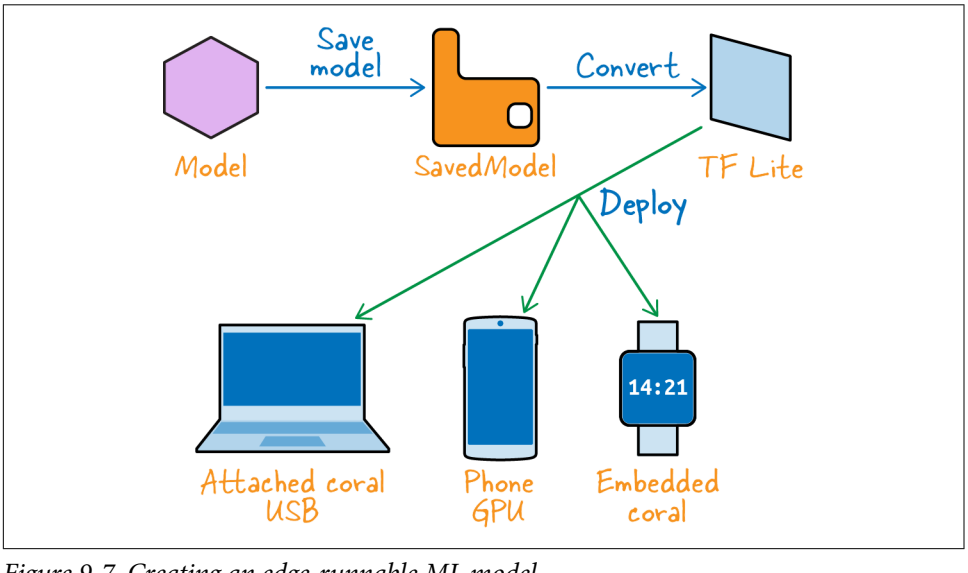

Figure 9-7. Creating an edge-runnable ML model.

To convert a SavedModel file into a TensorFlow Lite file, we need to use the `tf.lite` converter tool. We can do so from Python as follows:

```
converter = tf.lite.TFLiteConverter.from_saved_model(MODEL_LOCATION)
tflite_model = converter.convert()
with open('export/model.tflite', 'wb') as ofp:
    ofp.write(tflite_model)
```

In order to get efficient edge predictions, we need to do two things. First, we should make sure to use an edge-optimized model such as MobileNet. MobileNet tends to be about 40x faster than models like Inception thanks to optimizations such as pruning connections during training and using a piecewise linear approximation of the activation function (see Chapter 3).

Second, we should carefully select how to quantize the model weights. The appropriate choice for quantization depends on the device to which we are deploying the model. For example, the Coral Edge TPU works best if we quantize the model weights to integers. We can do quantization to integers by specifying some options on the converter:

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = training_dataset.take(100)
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 = converter.convert()
```

In this code, we ask the optimizer to look at one hundred representative images (or whatever the model input is) from our training dataset to determine how best to quantize the weights without losing the model's predictive power. We also ask the conversion process to use only int8 arithmetic, and specify that the input and output types for the model will be int8.

Quantizing the model weights from float32 to int8 allows the Edge TPU to use one-fourth the memory and to accelerate the arithmetic by carrying it out on integers, which is an order of magnitude faster than using floats. Quantization tends to incur about a 0.2 to 0.5% loss in accuracy, although this depends on the model and dataset.

Once we have a TensorFlow Lite model file, we download the file to the edge device or package the model file with the application that is installed onto the device.

Running TensorFlow Lite

To obtain predictions from the model, the edge devices need to run a TensorFlow Lite interpreter. Android comes with an interpreter written in Java. To do inference from within an Android program, we can do:

```
try (Interpreter tflite = new Interpreter(tf_lite_file)) {
    tflite.run(inputImageBuffer.getBuffer(), output);
}
```

Similar interpreters for iOS are available in Swift and Objective-C.

The ML Kit framework supports many common edge uses, like text recognition, barcode scanning, face detection, and object detection. ML Kit is well integrated with Firebase, a popular software development kit (SDK) for mobile applications. Before you roll your own ML solution, check that it is not already available in ML Kit.

For non-phone devices, use the Coral Edge TPU. At the time of writing, the Coral Edge TPU is available in three forms:

- A dongle that can be attached via USB3 to an edge device such as a Raspberry Pi
- A baseboard with Linux and Bluetooth
- A standalone chip that is small enough to be soldered onto an existing board

The Edge TPU tends to provide a 30–50x speedup over a CPU.

Using the TensorFlow Lite interpreter on Coral involves setting and retrieving the interpreter state:

```
interpreter = make_interpreter(path_to_tflite_model)
interpreter.allocate_tensors()
common.set_input(interpreter, imageBuffer)
interpreter.invoke()
result = classify.get_classes(interpreter)
```

To run models on microcontrollers like Arduino, use TinyML,[1] not TensorFlow Lite. A microcontroller is a small computer on a single circuit board and does not require any operating system. TinyML provides a customized TensorFlow library designed to run on embedded devices without an operating system and only tens of kilobytes of memory. TensorFlow Lite, on the other hand, is a set of tools that optimize ML models to run on edge devices that do have an operating system.

Processing the Image Buffer

On the edge device we will have to process the image in the camera buffer directly, so we will be processing only one image at a time. Let's change the serving signature appropriately (full code in *09e_tflite.ipynb* on GitHub):

1 Pete Warden and Daniel Situnayake, *TinyML* (O'Reilly, 2019).

```
@tf.function(input_signature=[
       tf.TensorSpec([None, None, 3], dtype=tf.float32)])
def predict_flower_type(img):
    img = tf.image.resize_with_pad(img, IMG_HEIGHT, IMG_WIDTH)
    ...
    return {
        'probability': tf.squeeze(top_prob, axis=0),
        'flower_type': tf.squeeze(pred_label, axis=0)
    }
```

We then export it:

```
model.save(MODEL_LOCATION,
           signatures={
               'serving_default': predict_flower_type
           })
```

and convert this model to TensorFlow Lite.

Federated Learning

With TensorFlow Lite, we trained the model on the cloud and converted the cloud-trained model to a file format that was copied over to the edge device. Once the model is on the edge device, it is no longer retrained. However, data drift and model drift will occur on edge ML models just as they do on cloud models. Therefore, we will have to plan on saving at least a sample of the images to a disk on the device, and periodically retrieving the images to a centralized location.

Recall, though, that one of the reasons to carry out inference on the edge is to support privacy-sensitive use cases. What if we don't want the image data to ever leave the device?

One solution to this privacy concern is *federated learning*. In federated learning, devices collaboratively learn a shared prediction model while each of the devices keeps its training data on-device. Essentially, each device computes a gradient update and shares only the gradient (not the original image) with its neighbors, or *federation*. The gradient updates across multiple devices are averaged by one or more members of the federation, and only the aggregate is sent to the cloud. It is also possible for a device to further fine-tune the shared prediction model based on interactions that happen on the device (see Figure 9-8). This allows for privacy-sensitive personalization to happen on each device.

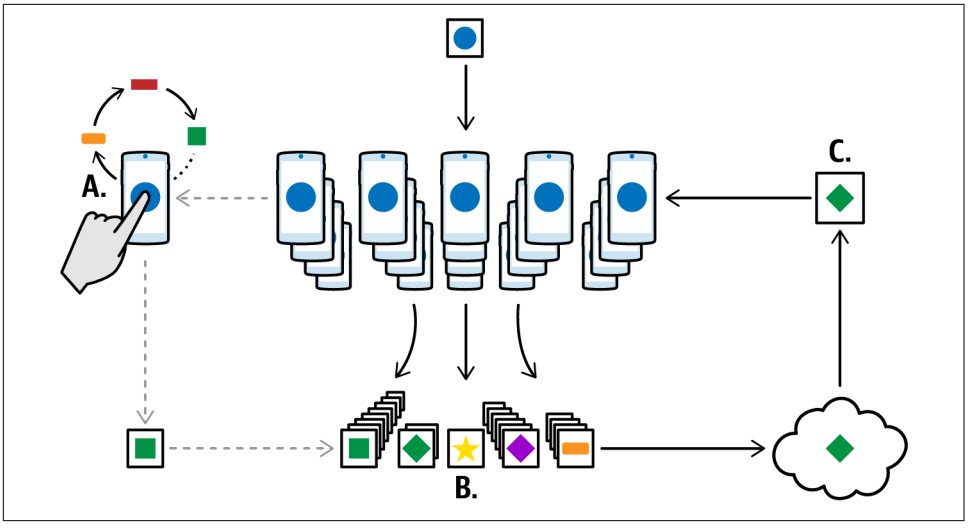

Figure 9-8. In federated learning, the model on the device (A) is improved based on both its own interactions and data from many other devices, but data never leaves the device. Many users' updates are aggregated (B) to form a consensus change (C) to the shared model, after which the procedure is repeated. Image courtesy of the Google AI Blog.

Even with this approach, model attacks could still extract some sensitive information out of the trained model. To further boost privacy protection, federated learning can be combined with *differential privacy*. An open source framework to implement federated learning is available in the TensorFlow repository.

Summary

In this chapter, we looked at how to invoke a trained model. We improved the abstraction provided by the prediction API and discussed how to improve the inference performance. For batch predictions, we suggested using a big data tool like Apache Beam and distributing the predictions over many machines.

For scaled concurrent, real-time predictions, we suggested deploying the model as a microservice using TensorFlow Serving. We also discussed how to change the signature of the model to support multiple requirements, and to accept image byte data sent directly over the wire. We also demonstrated making the model more efficient for deploying to the edge using TensorFlow Lite.

At this point, we have covered all the steps of the typical machine learning pipeline, from dataset creation to deployment for predictions. In the next chapter, we will look at a way to tie them all together into a pipeline.

CHAPTER 10
Trends in Production ML

So far in this book, we have looked at computer vision as a problem to be solved by data scientists. Because machine learning is used to solve real-world business problems, however, there are other roles that interface with data scientists to carry out machine learning—for example:

ML engineers
: ML models built by data scientists are put into production by ML engineers, who tie together all the steps of a typical machine learning workflow, from dataset creation to deployment for predictions, into a machine learning pipeline. You will often hear this being described as *MLOps*.

End users
: People who make decisions based on ML models tend to not trust black-box AI approaches. This is especially true in domains such as medicine, where end users are highly trained specialists. They will often require that your AI models are *explainable*—explainability is widely considered a prerequisite for carrying out AI responsibly.

Domain experts
: Domain experts can develop ML models using code-free frameworks. As such, they often help with data collection, validation, and problem viability assessment. You may hear this being described as ML being "democratized" through *no-code* or *low-code* tools.

In this chapter, we'll look at how the needs and skills of people in these adjacent roles increasingly affect the ML workflow in production settings.

The code for this chapter is in the *10_mlops* folder of the book's GitHub repository. We will provide file names for code samples and notebooks where applicable.

Machine Learning Pipelines

Figure 10-1 shows a high-level view of the machine learning pipeline. In order to create a web service that takes an image file and identifies the flower in it, as we have depicted throughout this book, we need to perform the following steps:

- Create our dataset by converting our JPEG images into TensorFlow Records, with the data split into training, validation, and test datasets.
- Train an ML model to classify flowers (we carried out hyperparameter tuning to select the best model, but let's assume that we can predetermine the parameters).
- Deploy the model for serving.

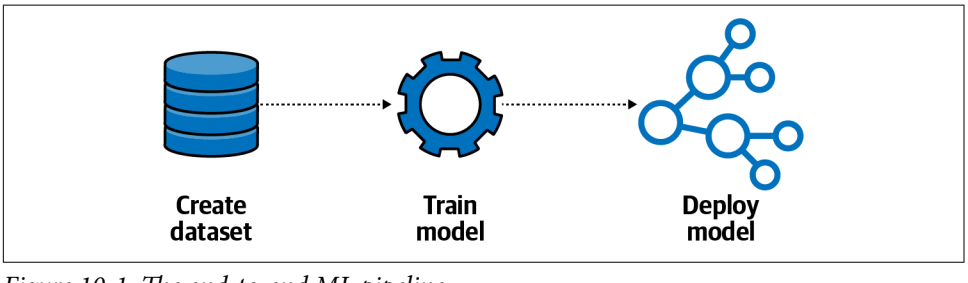

Figure 10-1. The end-to-end ML pipeline.

As you'll see in this section, in order to complete these steps in an ML pipeline we have to:

- Set up a cluster on which to execute the pipeline.
- Containerize our codebase, since the pipeline executes containers.
- Write pipeline components corresponding to each step of the pipeline.
- Connect the pipeline components so as to run the pipeline in one go.
- Automate the pipeline to run in response to events such as the arrival of new data.

First, though, let's discuss why we need an ML pipeline in the first place.

The Need for Pipelines

After we have trained our model on the original dataset, what happens if we get a few hundred more files to train on? We need to carry out the same set of operations to process those files, add them to our datasets, and retrain our model. In a model that depends heavily on having fresh data (say, one used for product identification rather than flower classification), we might need to perform these steps on a daily basis.

As new data arrives for a model to make predictions on, it is quite common for the model's performance to start to degrade because of *data drift*—that is, the newer data might be different from the data it was trained on. Perhaps the new images are of a higher resolution, or from a season or place we don't have in our training dataset. We can also anticipate that a month from now, we'll have a few more ideas that we will want to try. Perhaps one of our colleagues will have devised a better augmentation filter that we want to incorporate, or a new version of MobileNet (the architecture we are doing transfer learning from) might have been released. Experimentation to change our model's code will be quite common and will have to be planned for.

Ideally, we'd like a framework that will help us schedule and operationalize our ML pipelines and allow for constant experimentation. Kubeflow Pipelines provides a software framework that can represent any ML pipeline we choose in its domain-specific language (DSL). It runs on Kubeflow, a Kubernetes framework optimized for running TensorFlow models (see Figure 10-2). The managed Kubeflow Pipelines executor on Google Cloud is called Vertex Pipelines. The pipeline itself can execute the steps on the Kubernetes cluster (for on-premises work) or call out to Vertex Training, Vertex Prediction, and Cloud Dataflow on Google Cloud. Metadata about experiments and steps can be stored in the cluster itself, or in Cloud Storage and Cloud SQL.

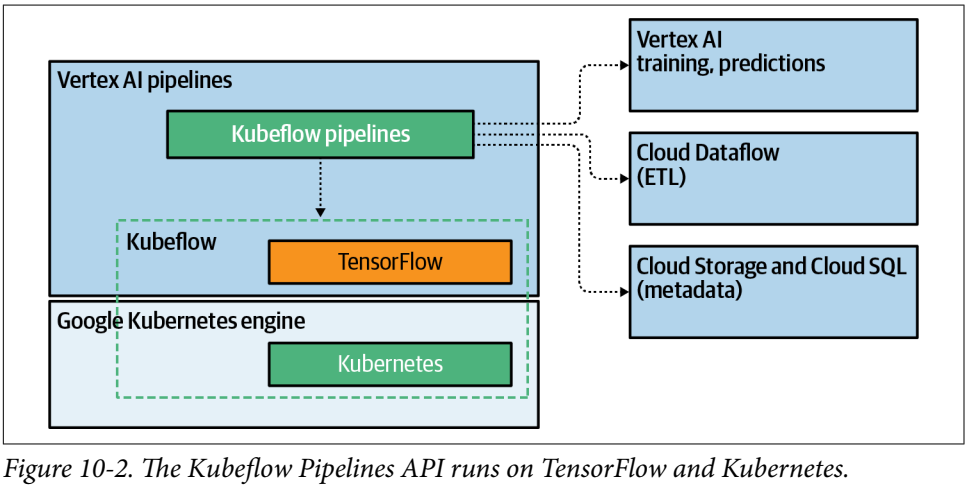

Figure 10-2. The Kubeflow Pipelines API runs on TensorFlow and Kubernetes.

Most ML pipelines follow a pretty standard set of steps: data validation, data transformation, model training, model evaluation, model deployment, and model monitoring. If your pipeline follows these steps, you can take advantage of the higher-level abstractions TensorFlow Extended (TFX) provides in the form of Python APIs. This way, you don't need to work at the level of the DSL and containerized steps. TFX is beyond the scope of this book.

Kubeflow Pipelines Cluster

To execute Kubeflow pipelines, we need a cluster. We can set one up on Google Cloud by navigating to the AI Platform Pipelines console and creating a new instance. Once started, we will get a link to open up the Pipelines dashboard and a Settings icon that provides the host URL (see Figure 10-3).

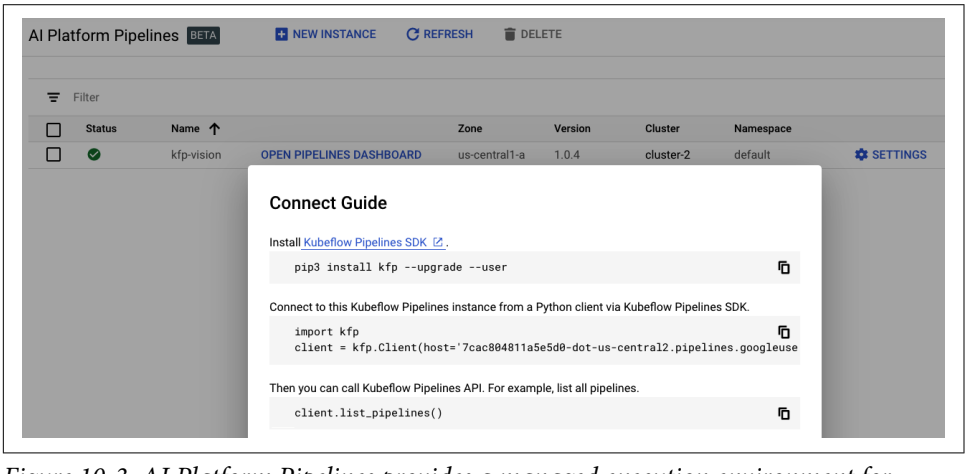

Figure 10-3. AI Platform Pipelines provides a managed execution environment for Kubeflow Pipelines.

We can develop pipelines in a Jupyter notebook, then deploy them to the cluster. Follow along with the full code in *07e_mlpipeline.ipynb* on GitHub.

Containerizing the Codebase

Once we have our cluster, the first step of our pipeline needs to transform the JPEG files into TensorFlow Records. Recall that we wrote an Apache Beam program called *jpeg_to_tfrecord.py* in Chapter 5 to handle this task. In order to make this repeatable, we need to make it a container where all the dependencies are captured.

We developed it in a Jupyter notebook, and fortunately the Notebooks service on Vertex AI releases a container image corresponding to each Notebook instance type.

Therefore, to build a container that is capable of executing that program, we need to do the following:

- Get the container image corresponding to the Notebook instance.
- Install any additional software dependencies. Looking through all our notebooks, we see that we need to install two additional Python packages: `apache-beam[gcp]` and `cloudml-hypertune`.
- Copy over the script. Because we will probably need other code from the repository as well for other tasks, it's probably better to copy over the entire repository.

This Dockerfile (the full code is in *Dockerfile* on GitHub) performs those three steps:

```
FROM gcr.io/deeplearning-platform-release/tf2-gpu
RUN python3 -m pip install --upgrade apache-beam[gcp] cloudml-hypertune
RUN mkdir -p /src/practical-ml-vision-book
COPY . /src/practical-ml-vision-book/
```

Those of you familiar with Dockerfiles will recognize that there is no `ENTRYPOINT` in this file. That's because we will set up the entry point in the Kubeflow component—all the components for our pipeline will use this same Docker image.

We can push the Docker image to a container registry using standard Docker functionality:

```
full_image_name=gcr.io/${PROJECT_ID}/practical-ml-vision-book:latest
docker build -t "${full_image_name}" .
docker push "$full_image_name"
```

Writing a Component

For every component that we need, we'll first load its definition from a YAML file and then use it to create the actual component.

The first component we need to create is the dataset (see Figure 10-1). From Chapter 5, we know that the step involves running *jpeg_to_tfrecord.py*. We define the component in a file named *create_dataset.yaml*. It specifies these input parameters:

```
inputs:
- {name: runner, type: str, default: 'DirectRunner', description: 'DirectRunner…'}
- {name: project_id, type: str, description: 'Project to bill Dataflow job to'}
- {name: region, type: str, description: 'Region to run Dataflow job in'}
- {name: input_csv, type: GCSPath, description: 'Path to CSV file'}
- {name: output_dir, type: GCSPath, description: 'Top-level directory…'}
- {name: labels_dict, type: GCSPath, description: 'Dictionary file…'}
```

It also specifies the implementation, which is to call a script called *create_dataset.sh* that you'll find in *create_dataset.sh* on GitHub. The arguments to the script are constructed from the inputs to the component:

```
implementation:
    container:
        image: gcr.io/[PROJECT-ID]/practical-ml-vision-book:latest
        command: [
            "bash",
            "/src/practical-ml-vision-book/.../create_dataset.sh"
        ]
        args: [
            {inputValue: output_dir},
            {outputPath: tfrecords_topdir},
            "--all_data", {inputValue: input_csv},
            "--labels_file", {inputValue: labels_dict},
            "--project_id", {inputValue: project_id},
            "--output_dir", {inputValue: output_dir},
            "--runner", {inputValue: runner},
            "--region", {inputValue: region},
        ]
```

The *create_dataset.sh* script simply forwards everything to the Python program:

```
cd /src/practical-ml-vision-book/05_create_dataset
python3 -m jpeg_to_tfrecord $@
```

Why do we need the extra level of indirection here? Why not simply specify `python3` as the command (instead of the bash call to the shell script)? That's because, besides just calling the converter program, we also need to perform additional functionality like creating folders, passing messages to subsequent steps of our Kubeflow pipelines, cleaning up intermediate files, and so on. Rather than update the Python code to add unrelated Kubeflow Pipelines functionality to it, we'll wrap the Python code within a bash script that will do the setup, message passing, and teardown. More on this shortly.

We will call the component from the pipeline as follows:

```
create_dataset_op = kfp.components.load_component_from_file(
    'components/create_dataset.yaml'
)
create_dataset = create_dataset_op(
    runner='DataflowRunner',
    project_id=project_id,
    region=region,
    input_csv='gs://cloud-ml-data/img/flower_photos/all_data.csv',
    output_dir='gs://{}/data/flower_tfrecords'.format(bucket),
    labels_dict='gs://cloud-ml-data/img/flower_photos/dict.txt'
)
```

If we pass in `DirectRunner` instead of `DataflowRunner`, the Apache Beam pipeline executes on the Kubeflow cluster itself (albeit slowly and on a single machine). This is useful for executing on premises.

Given the `create_dataset_op` component that we have just created, we can create a pipeline that runs this component as follows:

```
create_dataset_op = kfp.components.load_component_from_file(
    'components/create_dataset.yaml'
)

@dsl.pipeline(
    name='Flowers Transfer Learning Pipeline',
    description='End-to-end pipeline'
)
def flowerstxf_pipeline(
    project_id=PROJECT,
    bucket=BUCKET,
    region=REGION
):
    # Step 1: Create dataset
    create_dataset = create_dataset_op(
        runner='DataflowRunner',
        project_id=project_id,
        region=region,
        input_csv='gs://cloud-ml-data/img/flower_photos/all_data.csv',
        output_dir='gs://{}/data/flower_tfrecords'.format(bucket),
        labels_dict='gs://cloud-ml-data/img/flower_photos/dict.txt'
    )
```

We then compile this pipeline into a *.zip* file:

```
pipeline_func = flowerstxf_pipeline
pipeline_filename = pipeline_func.__name__ + '.zip'
import kfp.compiler as compiler
compiler.Compiler().compile(pipeline_func, pipeline_filename)
```

and submit that file as an experiment:

```
import kfp
client = kfp.Client(host=KFPHOST)
experiment = client.create_experiment('from_notebook')
run_name = pipeline_func.__name__ + ' run'
run_result = client.run_pipeline(
    experiment.id,
    run_name,
    pipeline_filename,
    {
        'project_id': PROJECT,
        'bucket': BUCKET,
        'region': REGION
```

```
    }
)
```

We can also upload the *.zip* file, submit the pipeline, and carry out experiments and runs using the Pipelines dashboard.

Connecting Components

We now have the first step of our pipeline. The next step (see Figure 10-1) is to train an ML model on the TensorFlow Records created in the first step.

The dependency between the `create_dataset` step and the `train_model` step is expressed as follows:

```
create_dataset = create_dataset_op(...)
train_model = train_model_op(
    input_topdir=create_dataset.outputs['tfrecords_topdir'],
    region=region,
    job_dir='gs://{}/trained_model'.format(bucket)
)
```

In this code, notice that one of the inputs to `train_model_op()` depends on the output of `create_dataset`. Connecting the two components in this manner makes Kubeflow Pipelines wait for the `create_dataset` step to complete before starting the `train_model` step.

The underlying implementation involves the `create_dataset` step writing out the value for the `tfrecords_topdir` into a local temporary file whose name will be automatically generated by Kubeflow Pipelines. So, our `create_dataset` step will have to take this additional input and populate the file. Here's how we write the output directory name to the file in *create_dataset.sh* (the parameters for Kubeflow to provide to this script are specified in the YAML file):

```
#!/bin/bash -x
OUTPUT_DIR=$1
shift
COMPONENT_OUT=$1
shift

# run the Dataflow pipeline
cd /src/practical-ml-vision-book/05_create_dataset
python3 -m jpeg_to_tfrecord $@

# for subsequent components
mkdir -p $(dirname $COMPONENT_OUT)
echo "$OUTPUT_DIR" > $COMPONENT_OUT
```

The script writes the name of the output directory to the component output file, removes the two parameters from the command-line arguments (that's what the

`shift` does in bash), and passes along the rest of the command-line parameters to `jpeg_to_tfrecord`.

The `train_model` step is similar to the `create_dataset` step in that it uses the codebase container and invokes a script to train the model:

```
name: train_model_caip
...
implementation:
    container:
        image: gcr.io/[PROJECT-ID]/practical-ml-vision-book:latest
        command: [
            "bash",
            "/src/practical-ml-vision-book/.../train_model_caip.sh",
        ]
        args: [
            {inputValue: input_topdir},
            {inputValue: region},
            {inputValue: job_dir},
            {outputPath: trained_model},
        ]
```

We can turn this into a local training run on the cluster by replacing the call to Vertex AI Training by a call to `gcloud ai-platform local`. See *train_model_kfp.sh* in the book's GitHub repository for details.

The script writes out the directory in which the trained model is stored:

```
echo "${JOB_DIR}/flowers_model" > $COMPONENT_OUT
```

The deploy step does not require any custom code. To deploy the model, we can use the deploy operator that comes with Kubeflow Pipelines:

```
deploy_op = kfp.components.load_component_from_url(
    'https://.../kubeflow/pipelines/.../deploy/component.yaml')
deploy_model = deploy_op(
    model_uri=train_model.outputs['trained_model'],
    model_id='flowers',
    version_id='txf',
    ...)
```

As the pipeline is run, the logs, steps, and artifacts passed between steps show up in the console (see Figure 10-4).

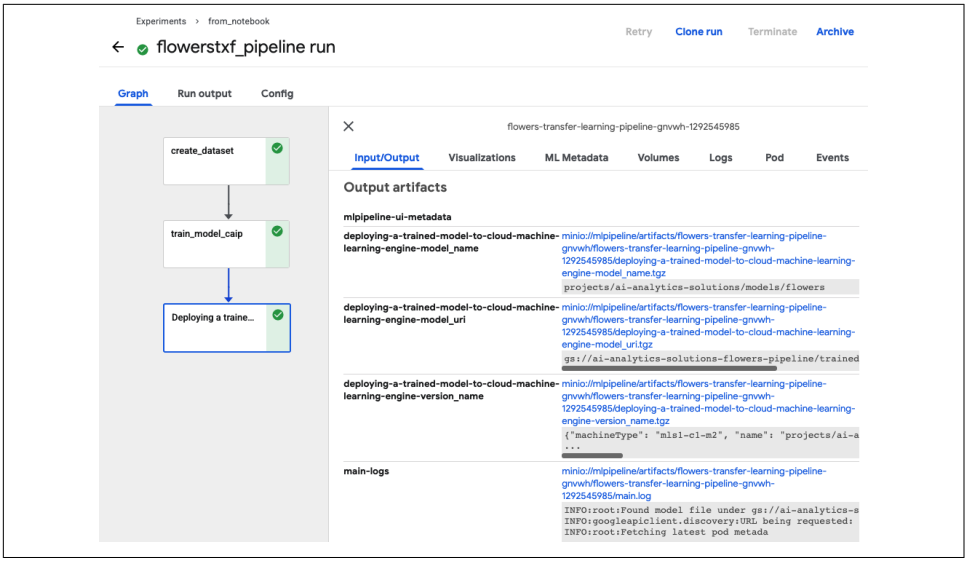

Figure 10-4. Information about a pipeline that has been run is displayed in the Vertex Pipelines console.

Automating a Run

Because we have a Python API to submit new runs of an experiment, it is quite straightforward to incorporate this Python code into a Cloud Function or a Cloud Run container. The function will then get invoked in response to a Cloud Scheduler trigger, or whenever new files are added to a storage bucket.

Caching in Kubeflow

The results of previous runs are cached and simply returned back if a component is run again with the same set of inputs and output strings. Unfortunately, Kubeflow Pipelines doesn't check the contents of Google Cloud Storage directories, so it doesn't know that the contents of a bucket may have changed even if the input parameter (the bucket) remains the same. Therefore, the caching tends to be of limited use. Because of that, you might want to explicitly set *staleness* criteria for the cache:

```
create_dataset.execution_options.caching_strategy.max_cache_staleness = "P7D"
```

The time duration for which data should be cached is represented in ISO 8601 format. P7D, for example, indicates that the output should be cached for 7 days. To effectively use caching, you have to incorporate timestamps into the names of your input and output directories.

The experiment-launching code can also be invoked in response to continuous integration (CI) triggers (such as GitHub/GitLab Actions) to carry out retraining any time new code is committed. The necessary continuous integration, continuous deployment (CD), permission management, infrastructure authorization, and authentication together form the realm of *MLOps*. MLOps is beyond the scope of this book, but the ML Engineering on Google Cloud Platform, MLOps on Azure, and Amazon SageMaker MLOps Workshop GitHub repositories contain instructions to help get you started on the respective platforms.

We have seen how pipelines address the need of ML engineers to tie together all the steps of a typical machine learning workflow into an ML pipeline. Next, let's look at how explainability meets the needs of decision makers.

Explainability

When we present an image to our model, we get a prediction. But why do we get that prediction? What is the model using to decide that a flower is a daisy or a tulip? Explanations of how AI models work are useful for several reasons:

Trust
: Human users may not trust a model that doesn't explain what it is doing. If an image classification model says that an X-ray depicts a fracture but does not point out the exact pixels it used to make its determination, few doctors will trust the model.

Troubleshooting
: Knowing what parts of an image are important to make a determination can be useful to diagnose why the model is making an error. For example, if a dog is identified as a fox, and the most relevant pixels happen to be of snow, it is likely that the model has wrongly learned to associate the background (snow) with foxes. To correct this error we will have to collect examples of foxes in other seasons or dogs in snow, or augment the dataset by pasting foxes and dogs into each others' scenes.

Bias busting
: If we are using image metadata as input to our model, examining the importance of features associated with sensitive data can be very important to determining sources of bias. For example, if a model to identify traffic violations treats potholes in the road as an important feature, this might be because the model is learning the biases in the training dataset (perhaps more tickets were handed out in poorer/less well maintained areas than in wealthy ones).

There are two types of explanations: global and instance-level. The term *global* here highlights that these explanations are a property of the whole model after training, as opposed to each individual prediction at inference time. These methods rank the

inputs to the model by the extent to which they *explain* the variance of the predictions. For example, we may say that `feature1` explains 36% of the variance, `feature2` 23%, and so on. Because global feature importance is based on the extent to which different features contribute to the variance, these methods are calculated on a dataset consisting of many examples, such as the training or the validation dataset. However, global feature importance methods are not that useful in computer vision because there are no explicit, human-readable features when images are directly used as inputs to models. We will therefore not consider global explanations any further.

The second type of explanation is a measure of *instance-level* feature importance. These explanations attempt to explain each individual prediction and are invaluable in fostering user trust and for troubleshooting errors. These methods are more common in image models, and will be covered next.

Techniques

There are four methods that are commonly employed to interpret or explain the predictions of image models. In increasing order of sophistication, they are:

- Local Interpretable Model-agnostic Explanations (LIME)
- Kernel Shapley Additive Explanations (KernelSHAP)
- Integrated Gradients (IG)
- Explainable Representations through AI (xRAI)

Let's look at each of these in turn.

LIME

LIME perturbs the input image by first identifying patches of the image that consist of contiguous similar pixels (see Figure 10-5), then replacing some of the patches with a uniform value, essentially removing them. It then asks the model to make a prediction on the perturbed image. For each perturbed image, we get a classification probability. These probabilities are spatially weighted based on how similar the perturbed image is to the original. Finally, LIME presents the patches with the highest positive weights as the explanation.

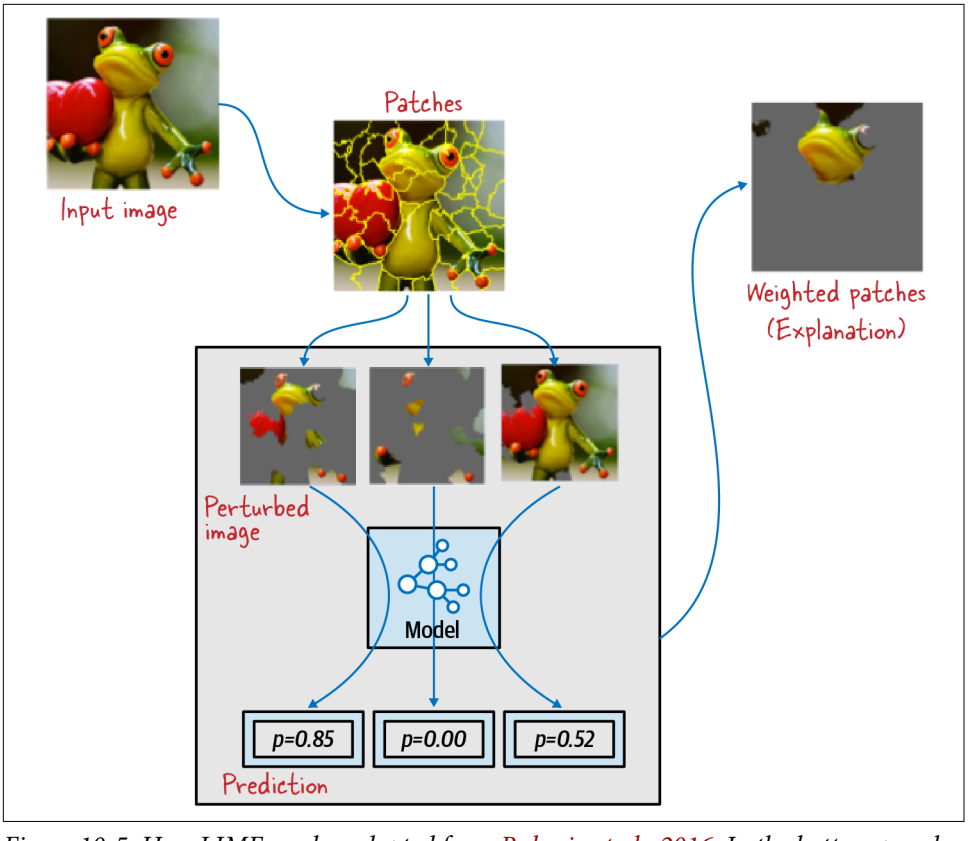

Figure 10-5. How LIME works, adapted from Ruberio et al., 2016. In the bottom panel, p represents the predicted probability of the image being that of a frog.

KernelSHAP

KernelSHAP is similar to LIME, but it weights the perturbed instances differently. LIME weights instances that are similar to the original image very low, on the theory that they possess very little extra information. KernelSHAP, on the other hand, weights the instances based on a distribution derived from game theory. The more patches are included in a perturbed image, the less weight the instance gets because theoretically any of those patches could have been important. In practice, KernelSHAP tends to be computationally much more expensive than LIME but provides somewhat better results.

Integrated Gradients

IG uses the gradient of the model to identify which pixels are important. A property of deep learning is that the training initially focuses on the most important pixels because the error rate can be reduced the most by using their information in the output. Therefore, high gradients are associated with important pixels at the start of training. Unfortunately, neural networks *converge* during training, and during convergence the network keeps the weights corresponding to the important pixels unchanged and focuses on rarer situations. This means that the gradients corresponding to the most important pixels are actually close to zero at the end of training! Therefore, IG needs the gradient not at the end of training, but over the entire training process. However, the only weights that are available in the SavedModel file are the final weights. So, how can IG use the gradient to identify important pixels?

IG is based on the intuition that the model will output the a priori class probability if provided a baseline image that consists of all 0s, all 1s, or random values in the range [0, 1]. The overall gradient change is computed numerically by changing each pixel's value from the baseline value to the actual input in several steps and computing the gradient for each such change. The pixels with the highest gradients integrated over the change from the baseline value to the actual pixel value are then depicted on top of the original image (see Figure 10-6).

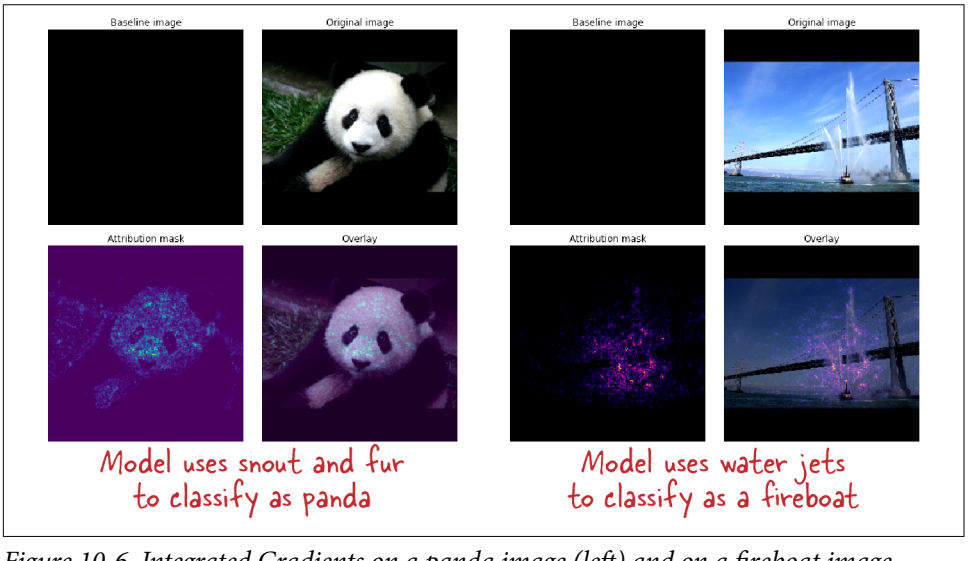

Figure 10-6. Integrated Gradients on a panda image (left) and on a fireboat image (right). Images from the IG TensorFlow tutorial.

Choosing an appropriate baseline image is critical when using IG. The explanation is relative to the baseline, so you should not use an all-white or all-black image as a baseline if your training data contains a lot of black (or white) regions that convey meaning in the images. As an example, black areas in X-rays correspond to tissue. You should use a baseline of random pixels in that case. On the other hand, if your training data contains a lot of high-variance patches that convey meaning in the image, you might not want to use random pixels as a baseline. It's worth trying different baselines, as this can significantly affect the quality of your attributions.

The output of IG on two images was shown in Figure 10-6. In the first image, IG identifies the snout and fur texture of the panda's face as the pixels that play the most important part in determining that the image is of a panda. The second image, of a fireboat, shows how IG can be used for troubleshooting. Here, the fireboat is correctly identified as a fireboat, but the method uses the jets of water from the boat as the key feature. This indicates that we may need to collect images of fireboats that are not actively shooting water up in the air.

However, in practice (as we will see shortly), IG tends to pick up on high-information areas in the images regardless of whether that information is used by the model for classifying the specific image.

xRAI

In xRAI, the weights and biases of the trained neural network are used to train an interpretation network. The interpretation network outputs a choice among a family of algebraic expressions (such as Booleans and low-order polynomials) that are well understood. Thus, xRAI aims to find a close approximation to the original trained model from within the family of simple functions. This approximation, rather than the original model, is then interpreted.

The xRAI method combines the benefits of the preprocessing method of LIME and KernelSHAP to find patches in the image with the pixel-level attribution against a baseline image that IG provides (see Figure 10-7). The pixel-level attribution is integrated among all the pixels that form a patch, and these patches are then combined into regions based on having similar levels of integrated gradients. The regions are then removed from the input image and the model is invoked in order to determine how important each region is, and the regions are ordered based on how important they are to the given prediction.

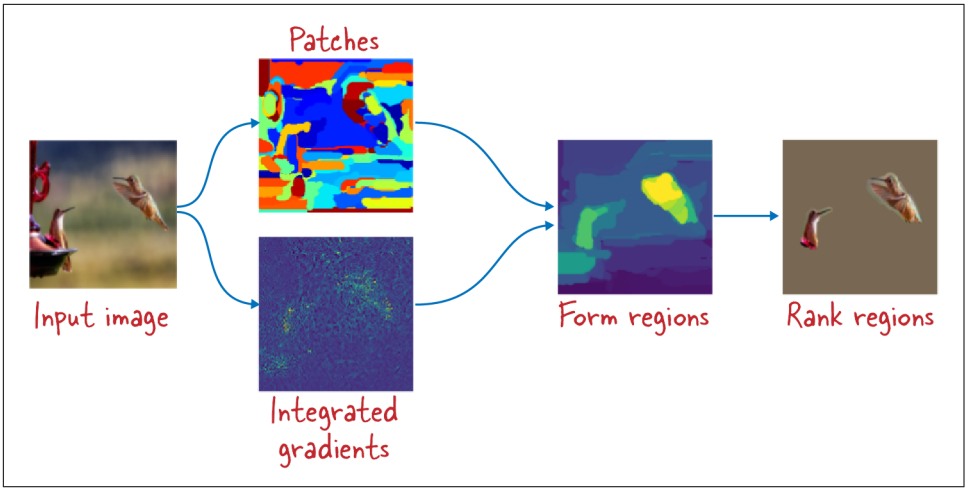

Figure 10-7. xRAI combines the patch-based preprocessing of LIME and KernelSHAP with the pixel-wise attributions of IG and ranks regions based on their effect on the prediction. Image adapted from the Google Cloud documentation.

IG provides pixel-wise attributions. xRAI provides region-based attributions. Both have their uses. In a model identifying diseased regions of an eye (the diabetic retinopathy use case), for example, knowing the specific pixels that caused the diagnosis is very useful, so use IG. IG tends to work best on low-contrast images like X-rays or scientific images taken in a lab.

In natural images where you're detecting the type of animal depicted, for example, region-based attributions are preferred, so use xRAI. We do not recommend using IG on natural images like pictures taken in nature or around the house.

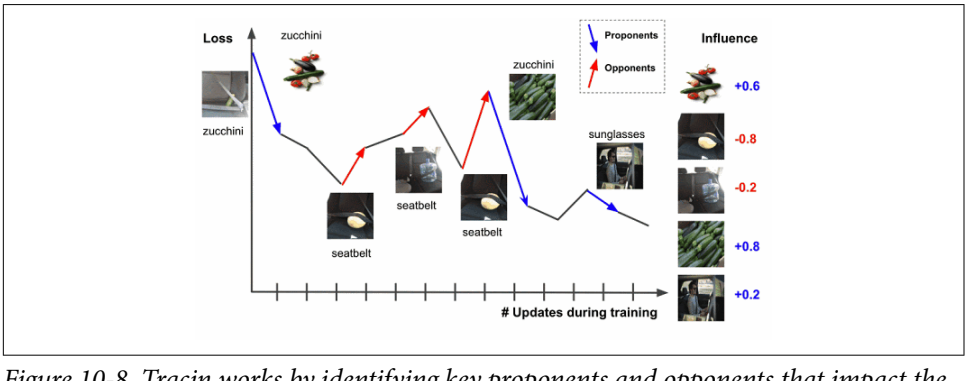

Figure 10-8. Tracin works by identifying key proponents and opponents that impact the training loss on a selected training example. Proponents are associated with a reduction in loss. Image courtesy of the Google AI Blog.

Let's now look at how to get explanations for our flowers' model's predictions using these techniques.

Tracing ML Training for Proponents and Opponents

All the explainability methods covered in the main text are about explaining *predictions* after the model is deployed. Recently, researchers at Google published an interesting paper that describes a method called Tracin that can be used to explain the model's behavior on *training examples* during the *training* process.

As shown in Figure 10-8, the underlying idea is to pick a single training example (such as the zucchini image to the left of the y-axis) and look for changes in the loss on that training example as weights are updated. Tracin identifies individual training examples that cause changes in the predicted class of individual training examples or changes in the direction of the loss. Examples that cause reduction in loss (i.e., improve the predictions) are called *proponents*, and those that cause the loss to increase are called *opponents*. Opponents tend to be similar images that belong to another class, and proponents tend to be similar images that belong to the same class as the selected training example. Exceptions to this rule tend to be mislabeled examples and outliers.

Adding Explainability

Because image explainability is associated with individual predictions, we recommend that you use an ML deployment platform that carries out one or all of the explainability techniques mentioned in the previous section for every prediction presented to it. Explainability methods are computationally expensive, and a deployment platform that can distribute and scale the computation can help you do your prediction analysis more efficiently.

In this section, we will demonstrate obtaining explanations using Integrated Gradients and xRAI from a model deployed on Google Cloud's Vertex AI.

At the time of writing, Azure ML supports SHAP, as does Amazon SageMaker Clarify. Conceptually, the services are used similarly even if the syntax is slightly different. Please consult the linked documentation for specifics.

Explainability signatures

The explainability methods all need to invoke the model with perturbed versions of the original image. Let's say that our flowers model has the following export signature:

```
@tf.function(input_signature=[tf.TensorSpec([None,], dtype=tf.string)])
def predict_filename(filenames):
    ...
```

It accepts a filename and returns the predictions for the image data in that file.

In order to provide the Explainable AI (XAI) module the ability to create perturbed versions of the original images and obtain predictions for them, we will need to add two signatures:

- A preprocessing signature, to obtain the image that is input to the model. This method will take one or more filenames as input (like the original exported signature) and produce a 4D tensor of the shape required by the model (the full code is in *09f_explain.ipynb* on GitHub):

  ```
  @tf.function(input_signature=[
               tf.TensorSpec([None,], dtype=tf.string)])
  def xai_preprocess(filenames):
      input_images = tf.map_fn(
          preprocess, # preprocessing function from Ch 6
          filenames,
          fn_output_signature=tf.float32
      )
      return {
          'input_images': input_images
      }
  ```

  Note that the return value is a dictionary. The key values of the dictionary (`input_images`, here) have to match the parameter names in the second signature that is described next so that the two methods can be called one after the other in a third model signature that we will discuss shortly.

- A model signature, to send in the 4D image tensor (XAI will send in perturbed images) and obtain predictions:

  ```
  @tf.function(input_signature=[
       tf.TensorSpec([None, IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS],
                     dtype=tf.float32)])
  def xai_model(input_images):
      batch_pred = model(input_images) # same as model.predict()
      top_prob = tf.math.reduce_max(batch_pred, axis=[1])
      pred_label_index = tf.math.argmax(batch_pred, axis=1)
      pred_label = tf.gather(tf.convert_to_tensor(CLASS_NAMES),
                             pred_label_index)
      return {
          'probability': top_prob,
          'flower_type_int': pred_label_index,
          'flower_type_str': pred_label
     }
  ```

This code invokes the model and then pulls out the highest-scoring label and its probability.

Given the preprocessing and model signatures, the original signature (that most clients will use) can be refactored into:

```
@tf.function(input_signature=[tf.TensorSpec([None,], dtype=tf.string)])
def predict_filename(filenames):
    preproc_output = xai_preprocess(filenames)
    return xai_model(preproc_output['input_images'])
```

Now, we save the model with all three export signatures:

```
model.save(MODEL_LOCATION,
           signatures={
               'serving_default': predict_filename,
               'xai_preprocess': xai_preprocess, # input to image
               'xai_model': xai_model # image to output
           })
```

At this point, the model has the signatures it needs to apply XAI, but there is some additional metadata needed in order to compute explanations.

Explanation metadata

Along with the model, we need to supply XAI a baseline image and some other metadata. This takes the form of a JSON file that we can create programmatically using the Explainability SDK open-sourced by Google Cloud.

We start by specifying which exported signature is the one that takes a perturbed image as input, and which of the output keys (`probability`, `flower_type_int`, or `flower_type_str`) needs to be explained:

```
from explainable_ai_sdk.metadata.tf.v2 import SavedModelMetadataBuilder
builder = SavedModelMetadataBuilder(
    MODEL_LOCATION,
    signature_name='xai_model',
    outputs_to_explain=['probability'])
```

Then we create the baseline image that will be used as the starting point for gradients. Common choices here are all zeros (`np.zeros`), all ones (`np.ones`), or random noise. Let's do the third option:

```
random_baseline = np.random.rand(IMG_HEIGHT, IMG_WIDTH, 3)
builder.set_image_metadata(
    'input_images',
    input_baselines=[random_baseline.tolist()])
```

Note that we specified the name of the input parameter to the `xai_model()` function, `input_images`.

Finally, we save the metadata file:

```
builder.save_metadata(MODEL_LOCATION)
```

This creates a file named *explanation_metadata.json* that lives along with the SavedModel files.

Deploying the model

The SavedModel and associated explanation metadata are deployed to Vertex AI as before, but with a couple of extra parameters to do with explainability. To deploy a model version that provides IG explanations, we'd do:

```
gcloud beta ai-platform versions create \
    --origin=$MODEL_LOCATION --model=flowers ig ... \
    --explanation-method integrated-gradients --num-integral-steps 25
```

whereas to get xRAI explanations we'd do:

```
gcloud beta ai-platform versions create \
    --origin=$MODEL_LOCATION --model=flowers xrai ... \
    --explanation-method xrai --num-integral-steps 25
```

The `--num-integral-steps` argument specifies the number of steps between the baseline image and input image for the purposes of numerical integration. The more steps there are, the more accurate (and computationally intensive) the gradient computation is. A value of 25 is typical.

The explanation response contains an approximation error for each prediction. Check the approximation error for a representative set of inputs, and if this error is too high, increase the number of steps.

For this example, let's employ both image explainability methods—we'll deploy a version that provides IG explanations with the name `ig` and a version that provides xRAI explanations with the name `xrai`.

Either deployed version can be invoked as normal with a request whose payload looks like this:

```
{
    "instances": [
        {
            "filenames": "gs://.../9818247_e2eac18894.jpg"
        },
        {
            "filenames": "gs://.../9853885425_4a82356f1d_m.jpg"
        },
        ...
    ]
}
```

It returns the label and associated probability for each of the input images:

```
FLOWER_TYPE_INT  FLOWER_TYPE_STR  PROBABILITY
1                dandelion        0.398337
1                dandelion        0.999961
0                daisy            0.994719
4                tulips           0.959007
4                tulips           0.941772
```

The XAI versions can be used for normal serving with no performance impact.

Obtaining explanations

There are three ways to get the explanations. The first is through `gcloud` and the second through the Explainable AI SDK. Both of these end up invoking the third way—a REST API—which we can use directly as well.

We'll look at the `gcloud` method, as it is the simplest and most flexible. We can send in a JSON request and obtain a JSON response using:

```
gcloud beta ai-platform explain --region=$REGION \
    --model=flowers --version=ig \
    --json-request=request.json > response.json
```

To get explanations using IG, we'll deploy this version (`ig`) with the option:

```
--explanation-method integrated-gradients
```

The JSON response contains the attribution image in base64-encoded form. We can decode it using:

```
with open('response.json') as ifp:
    explanations = json.load(ifp)['explanations']
    for expln in explanations:
        b64bytes = (expln['attributions_by_label'][0]
                    ['attributions']['input_images']['b64_jpeg'])
        img_bytes = base64.b64decode(b64bytes)
        img = tf.image.decode_jpeg(img_bytes, channels=3)
        attribution = tf.image.convert_image_dtype(img, tf.float32)
```

The IG results for five images are shown in Figure 10-9. The *10b_explain.ipynb* notebook on GitHub has the necessary plotting code.

Figure 10-9. Integrated Gradients explanation of the flowers model. The input images are in the top row, and the attributions returned by the XAI routine are in the second row.

For the first image, it appears that the model uses the tall white flower, as well as parts of the white pixels in the background, to decide that the image is a daisy. In the second image, the yellow-ish center and white petals are what the model relies on. Worryingly, in the fourth image, the cat seems to be an important part of the determination. Interestingly, the tulips' determination seems to be driven more by the green stalks than the bulb-like flowers. Again, as we will see shortly, this attribution is misleading, and this misleading attribution demonstrates the limitations of the IG approach.

To get xRAI explanations, we invoke `gcloud explain` on the model endpoint for the version we deployed with the name `xrai`. The attributions from xRAI for the same flower images are shown in Figure 10-10.

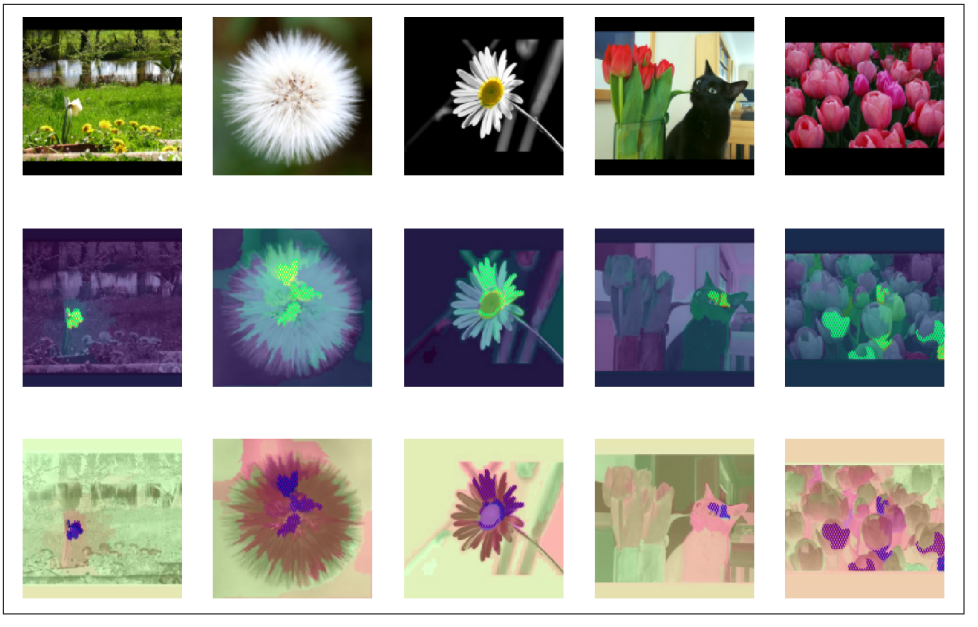

Figure 10-10. xRAI explanation of the flowers model. The input images are in the top row, and the attributions returned by the XAI routine are in the second row. The bottom row contains the same information as the second row, except that the attribution images have been recolored for easier visualization on the pages of this book.

Recall that xRAI uses the IG approach to identify salient regions, and then invokes the model with perturbed versions of the images to determine how important each of the regions is. It is clear that the attributions from xRAI in Figure 10-10 are much more precise than those obtained with IG in Figure 10-9.

For the first flower image, the model focuses on the tall white flower and only that flower. It is clear that the model has learned to ignore the smaller flowers in the background. And while IG seemed to indicate that the background was important, the xRAI results show that the model discards that information in favor of the most prominent flower in the image. In the second image, the yellow-ish center and white petals are what the model keys off of (IG got this right too). The precision of the xRAI approach is clear for the third image—the model picks up on the narrow band of bright yellow where the petals join the center. That is unique to daisies, and helps it distinguish them from similarly colored dandelions. In the fourth image, we can see that the tulip bulbs are what the model uses for its classification, although the cat confuses its attention. The final classification as tulips seems to be driven by the presence of so many flowers. The IG method led us astray—the stalks are prominent, but it is the bulbs that drive the prediction probability.

IG is useful in certain situations. Had we considered radiology images where pixel-wise attributions (rather than regions) are important, IG would have performed better. However, in images that depict objects, xRAI tends to perform better.

In this section, we have looked at how to add explainability to our prediction services in order to meet the need of decision makers to understand what a machine learning model is relying on. Next, let's look at how no-code tools help democratize ML.

No-Code Computer Vision

The computer vision problems that we have considered so far in this book—image classification, object detection, and image segmentation—are supported out of the box by low-code and no-code machine learning systems. For example, Figure 10-11 shows the starting console for Google Cloud AutoML Vision.

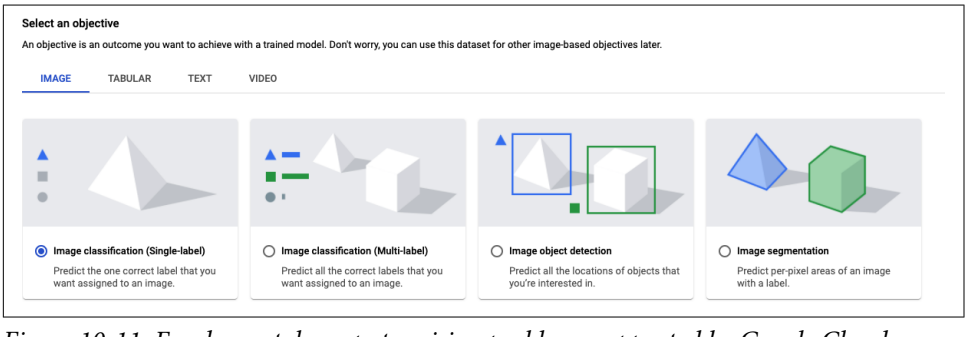

Figure 10-11. Fundamental computer vision problems supported by Google Cloud AutoML Vision, a machine learning tool that you can use without having to write any code.

Other no-code and low-code tools that work on images include Create ML by Apple, DataRobot, and H2O.

Why Use No-Code?

In this book, we have focused on implementing machine learning models using code. However, it is worth incorporating a no-code tool into your overall workflow.

No-code tools are useful when embarking on a computer vision project for several reasons, including:

Problem viability
: Tools such as AutoML serve as a sanity check on the kind of accuracy that you can expect. If the accuracy that is achieved is far from what would be acceptable in context, this allows you to avoid wasting your time on futile ML projects. For example, if identifying a counterfeit ID achieves only 98% precision at the desired recall, you know that you have a problem—wrongly rejecting 2% of your customers might be an unacceptable outcome.

Data quality and quantity
: No-code tools provide a check on the quality of your dataset. After data collection, the correct next step in many ML projects is to go out and collect more/better data, not to train an ML model; and the accuracy that you get from a tool like AutoML can help you make that decision. For example, if the confusion matrix the tool produces indicates that the model frequently classifies all flowers in water as lilies, that might be an indication that you need more photographs of water scenes.

Benchmarking
: Starting out with a tool like AutoML gives you a benchmark against which you can compare the models that you build.

Many machine learning organizations arm their domain experts with no-code tools so that they can examine problem viability and help collect high-quality data before bringing the problem to the data science team.

In the rest of this section, we'll quickly run through how to use AutoML on the 5-flowers dataset, starting with loading data.

Loading Data

The first step is to load the data into the system. We do that by pointing the tool at the *all_data.csv* file in the Cloud Storage bucket (see Figure 10-12).

Once the data is loaded, we see that there are 3,667 images with 633 daisies, 898 dandelions, and so on (see Figure 10-13). We can verify that all the images are labeled, and correct the labels if necessary. If we had loaded a dataset without labels, we could label the images ourselves in the user interface or farm the task out to a labeling service (labeling services were covered in Chapter 5).

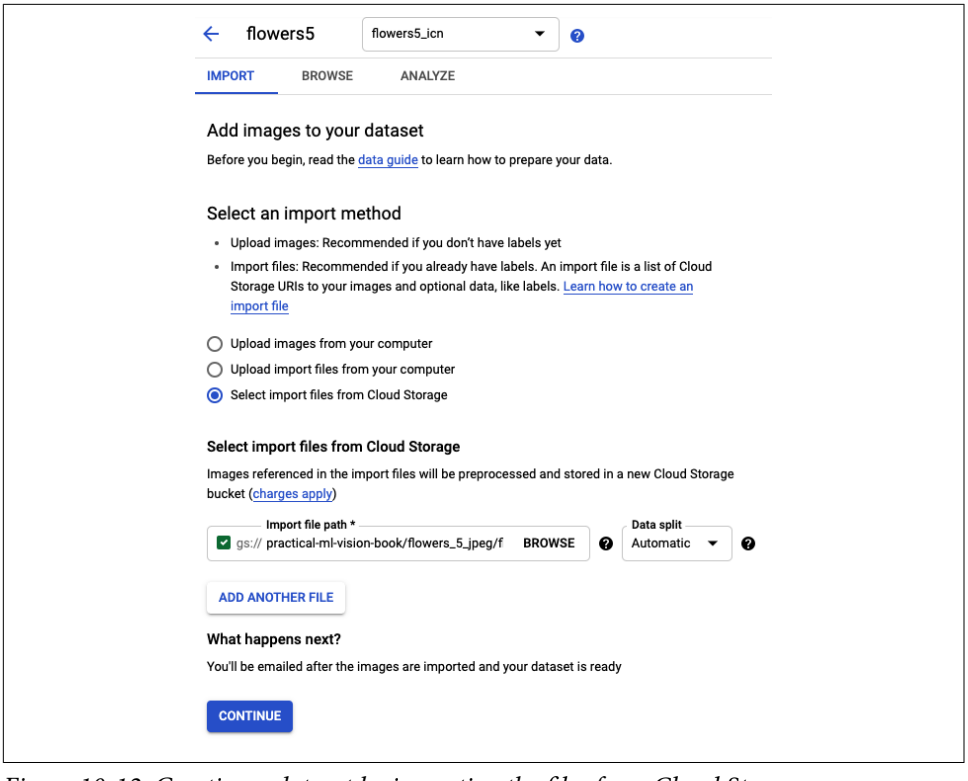

Figure 10-12. Creating a dataset by importing the files from Cloud Storage.

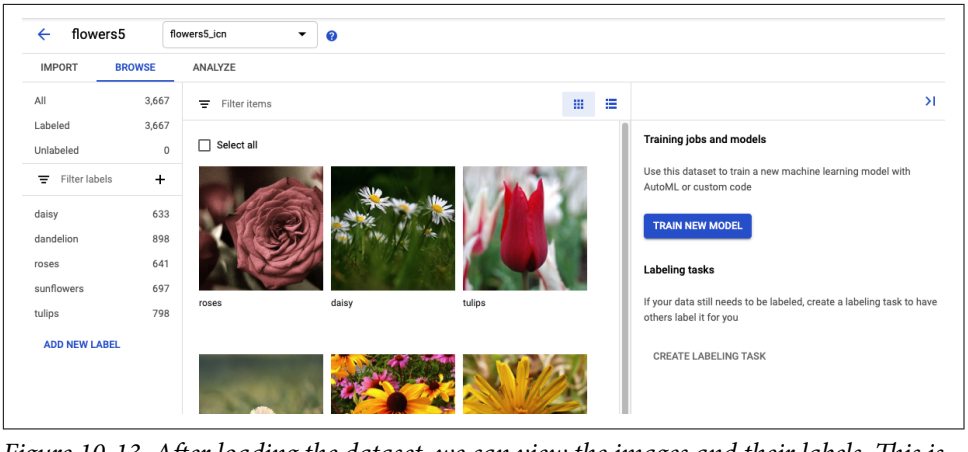

Figure 10-13. After loading the dataset, we can view the images and their labels. This is also an opportunity to add or correct labels if necessary.

Training

Once we are happy with the labels, we can click the Train New Model button to train a new model. This leads us through the set of screens shown in Figure 10-14, where we select the model type, the way to split the dataset, and our training budget. At the time of writing, the 8-hour training budget we specified would have cost about $25.

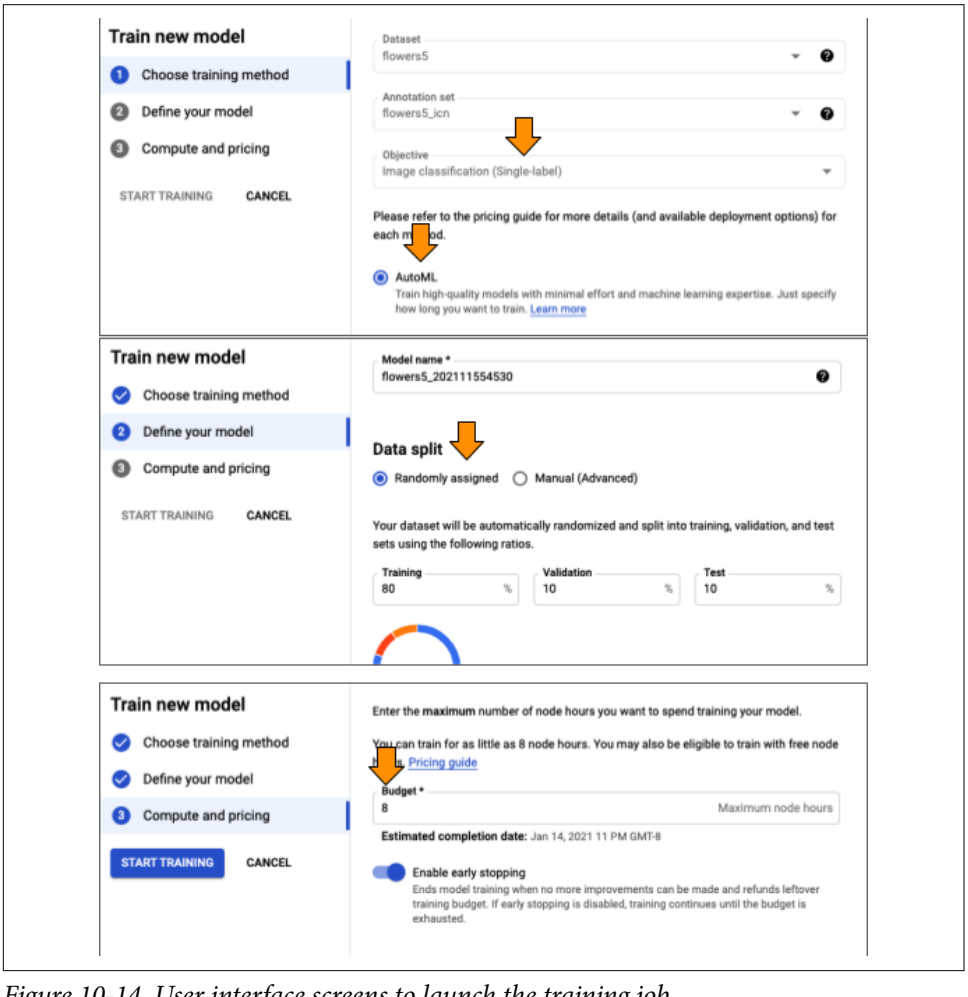

Figure 10-14. User interface screens to launch the training job.

Note that in the last screen we enabled early stopping, so that AutoML can decide to stop early if it doesn't see any more improvement in validation metrics. With this option the training finished in under 30 minutes (see Figure 10-15), meaning that the entire ML training run cost us around $3. The result was 96.4% accuracy, comparable

to the accuracy we got with the most sophisticated models we created in Chapter 3 after a lot of tuning and experimentation.

Figure 10-15. AutoML finished training in well under an hour at a cost of less than $3, and it achieved an accuracy of 96.4% on the 5-flowers dataset.

We should caution you that not all no-code systems are the same—the Google Cloud AutoML system we used in this section performs data preprocessing and augmentation, employs state-of-the-art models, and carries out hyperparameter tuning to build a very accurate model. Other no-code systems might not be as sophisticated: some train only one model (e.g., ResNet50), some train a single model but do hyperparameter tuning, and some others search among a family of models (ResNet18, ResNet50, and EfficientNet). Check the documentation so that you know what you are getting.

Evaluation

The evaluation results indicate that the most misclassifications were roses wrongly identified as tulips. If we were to continue our experimentation, we would examine some of the mistakes (see Figure 10-16) and attempt to gather more images to minimize the false positives and negatives.

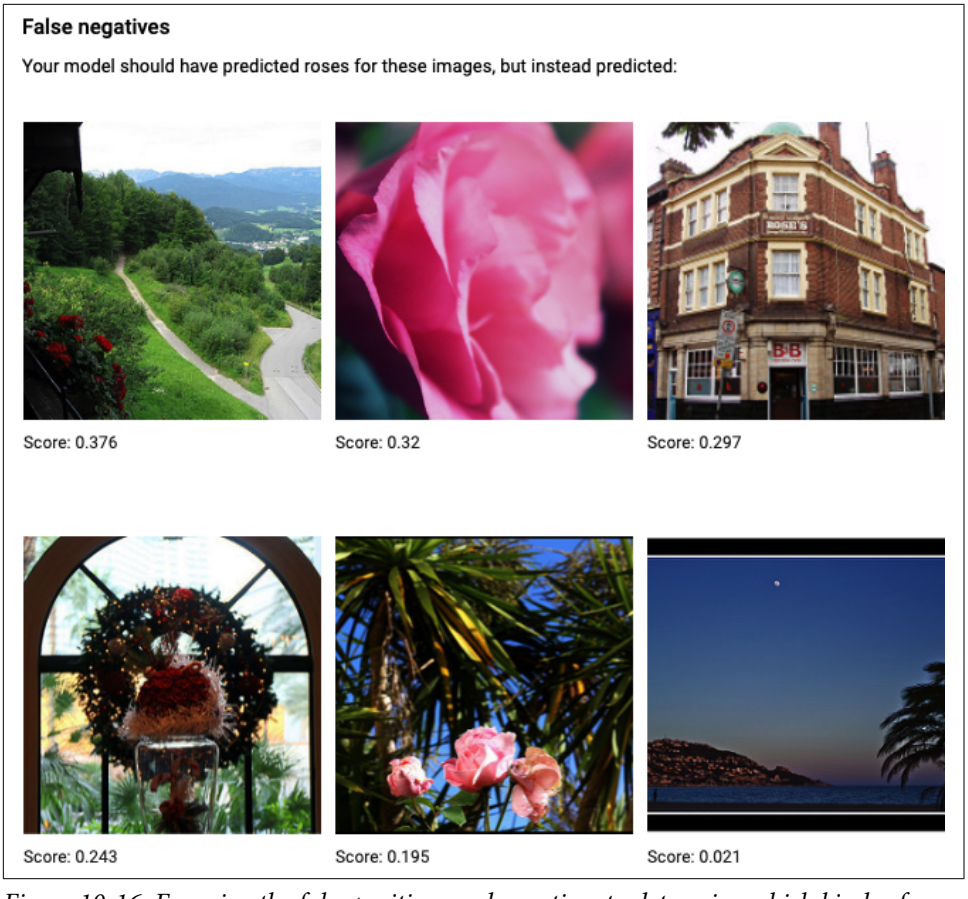

Figure 10-16. Examine the false positives and negatives to determine which kinds of examples to collect more of. This can also be an opportunity to remove unrepresentative images from the dataset.

Once we are satisfied with the model's performance, we can deploy it to an endpoint, thus creating a web service through which clients can ask the model to make predictions. We can then send sample requests to the model and obtain predictions from it.

For basic computer vision problems, the ease of use, low cost, and high accuracy of no-code systems are extremely compelling. We recommend that you incorporate these tools as a first step in your computer vision projects.

Summary

In this chapter, we looked at how to operationalize the entire ML process. We used Kubeflow Pipelines for this purpose and took a whirlwind tour of the SDK, creating Docker containers and Kubernetes components and stringing them into a pipeline using data dependencies.

We explored several techniques that allow us to understand what signals the model is relying on when it makes a prediction. We also looked at what no-code computer vision frameworks are capable of, using Google Cloud's AutoML to illustrate the typical steps.

No-code tools are used by domain experts to validate problem viability, while machine learning pipelines are used by ML engineers in deployment, and explainability is used to foster adoption of machine learning models by decision makers. As such, these usually form the bookends of many computer vision projects and are the points at which data scientists interface with other teams.

This concludes the main part of this book, where we have built and deployed an image classification model from end to end. In the remainder of the book, we will focus on advanced architectures and use cases.

CHAPTER 11

Advanced Vision Problems

So far in this book, we have looked primarily at the problem of classifying an entire image. In Chapter 2 we touched on image regression, and in Chapter 4 we discussed object detection and image segmentation. In this chapter, we will look at more advanced problems that can be solved using computer vision: measurement, counting, pose estimation, and image search.

The code for this chapter is in the *11_adv_problems* folder of the book's GitHub repository. We will provide file names for code samples and notebooks where applicable.

Object Measurement

Sometimes we want to know the measurements of an object within an image (e.g., that a sofa is 180 cm long). While we can simply use pixel-wise regression to measure something like ground precipitation using aerial images of cloud cover, we will need to do something more sophisticated for the object measurement scenario. We can't simply count the number of pixels and infer a size from that, because the same object could be represented by a different number of pixels due to where it is within the image, its rotation, aspect ratio, etc. Let's walk through the four steps needed to measure an object from a photograph of it, following an approach suggested by Imaginea Labs.

Reference Object

Suppose we're an online shoe store, and we want to help customers find the best shoe size by using photographs of their footprints. We ask customers to get their feet wet and step onto a paper material, then upload a photo of their footprint like the one shown in Figure 11-1. We can then obtain the appropriate shoe size (based on length and width) and arch type from the footprint using an ML model.

Figure 11-1. Left: Photograph of wet footprint on paper. Right: Photograph of the same footprint taken with the camera a few inches closer to the paper. Identifying the high-pressure areas is helpful to identify the type of arch the person has. Photographs in this section are by the author.

The ML model should be trained using different paper types, different lighting, rotations, flips, etc. to anticipate all of the possible variations of footprint images the model might receive at inference time to predict foot measurements. But an image of the footprint alone is insufficient to create an effective measurement solution, because (as you can see in Figure 11-1) the size of foot in the image will vary depending on factors such as the distance between the camera and the paper.

A simple way to address the scale problem is to include a reference object that virtually all customers should have. Most customers have credit cards, which have standard dimensions, so this can be used as a reference or calibration object to help the model determine the relative size of the foot in the image. As shown in Figure 11-2, we simply ask each customer to place a credit card next to their footprint before taking the photo. Having a reference object simplifies the measurement task to one of comparing the foot against that object.

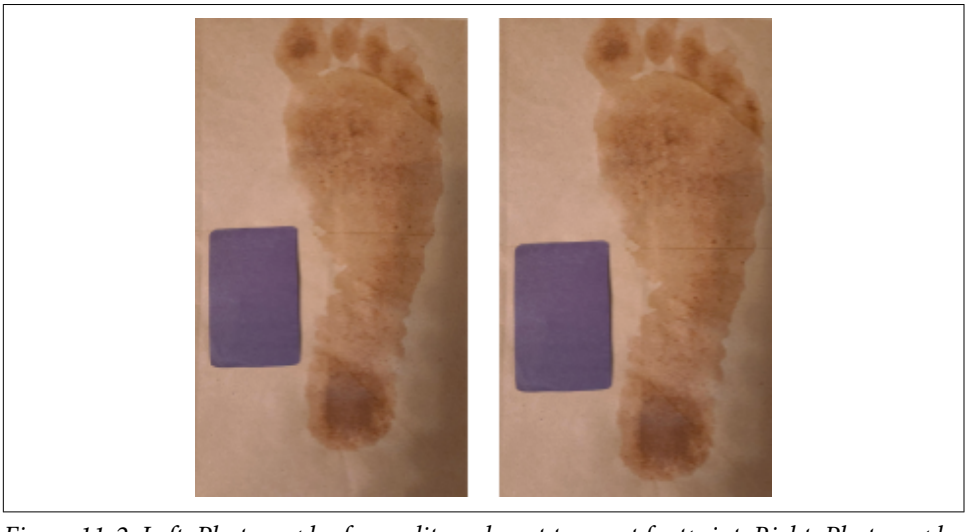

Figure 11-2. Left: Photograph of a credit card next to a wet footprint. Right: Photograph of the same objects, taken with the camera a few inches closer to the paper.

Building our training dataset of different footprints on various backgrounds of course may require some cleaning, such as rotating the images to have all footprints oriented the same way. Otherwise, for some images we would be measuring the projected length and not the true length. As for the reference credit card, we won't perform any corrections before training and will align the generated foot and reference masks at prediction time.

At the beginning of the training we can perform data augmentation, such as rotating, blurring, and changing the brightness, scaling, and contrast, as shown in Figure 11-3. This can help us increase the size of our training dataset as well as teaching the model to be flexible enough to receive many different real-world variations of the data.

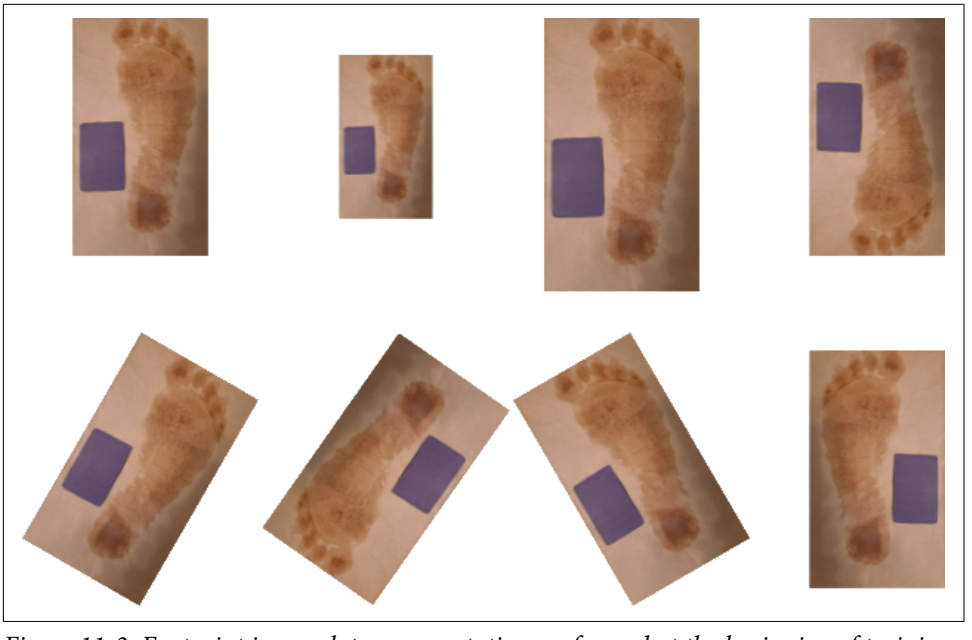

Figure 11-3. Footprint image data augmentation performed at the beginning of training.

Segmentation

The machine learning model first needs to segment out the footprint from the credit card in the image and identify those as the two correct objects extracted. For this we will be using the Mask R-CNN image segmentation model, as discussed in Chapter 4 and depicted in Figure 11-4.

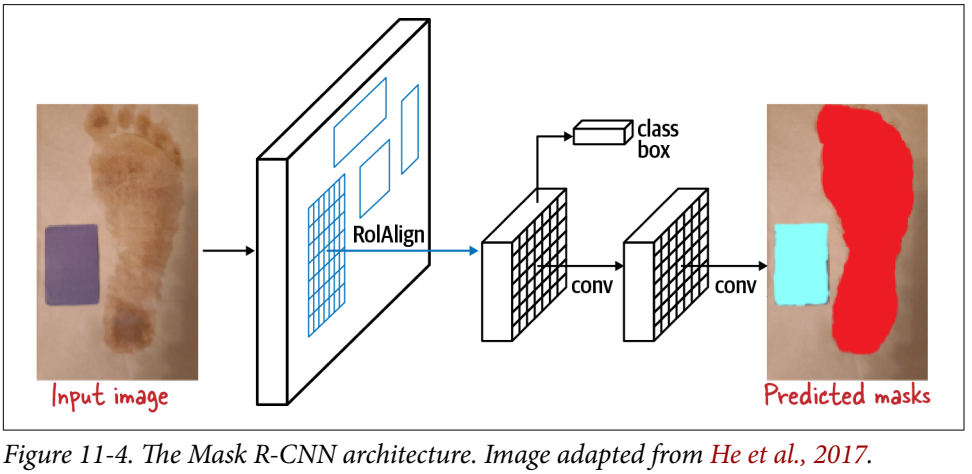

Figure 11-4. The Mask R-CNN architecture. Image adapted from He et al., 2017.

Through the mask branch of the architecture we will predict a mask for the footprint and a mask for the credit card, obtaining a result similar to that on the right in Figure 11-4.

Remember our mask branch's output has two channels: one for each object, footprint and credit card. Therefore, we can look at each mask individually, as shown in Figure 11-5.

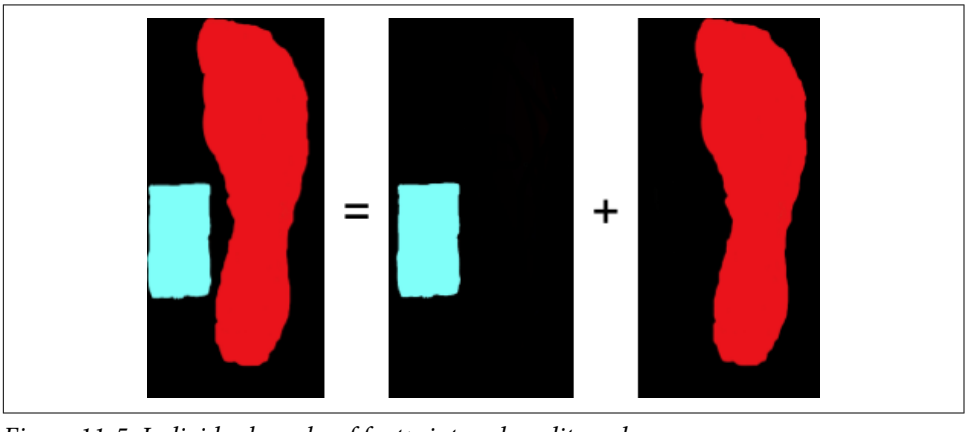

Figure 11-5. Individual masks of footprint and credit card.

Next, we have to align the masks so that we can obtain the correct measurements.

Rotation Correction

Once we have obtained the masks of the footprint and the credit card, they have to be normalized with respect to rotation, for users who may have placed the credit card in slightly different orientations when taking the photograph.

To correct for the rotation, we can use principal component analysis (PCA) on each of the masks to get the *eigenvectors*—the size of the object in the direction of the largest eigenvector, for example, is the length of the object (see Figure 11-6). The eigenvectors obtained from PCA are orthogonal to each other and each subsequent component's eigenvector has a smaller and smaller contribution to the variance.

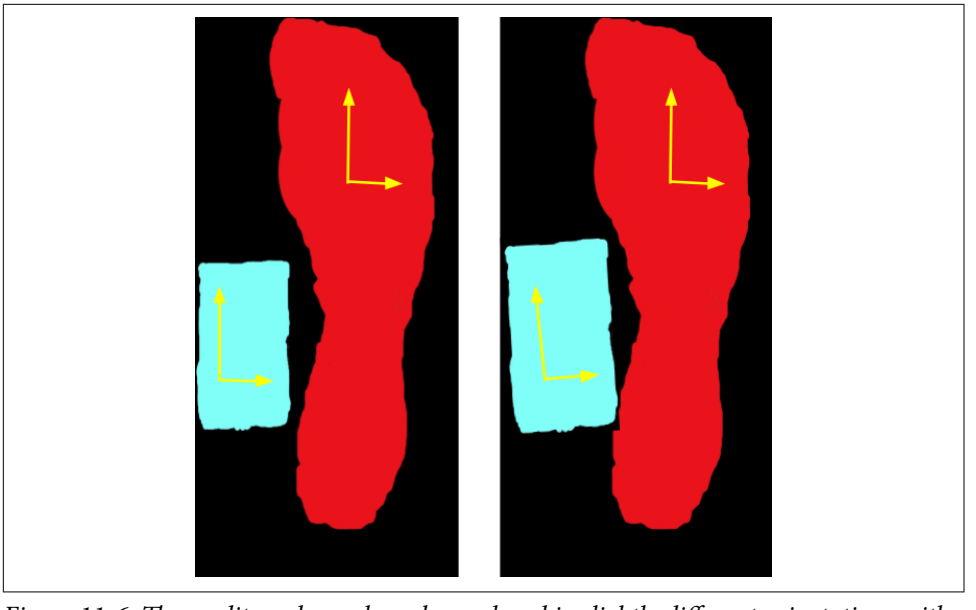

Figure 11-6. The credit card may have been placed in slightly different orientations with respect to the foot. The directions of the two largest eigenvectors in each object are marked by the axes.

Before PCA, the mask dimensions were in a vector space that had dimension axes with respect to the original image, as shown on the left side of Figure 11-6. Using the fact that the eigenvectors are in a different vector space basis after PCA, with the axes now along the direction of greatest variance (as shown on the right in Figure 11-6), we can use the angle between the original coordinate axis and the first eigenvector to determine how much of a rotation correction to make.

Ratio and Measurements

With our rotation-corrected masks, we can now calculate the footprint measurements. We first project our masks onto a two-dimensional space and look along the x- and y-axes. The length is found by measuring the pixel distance between the smallest and largest y-coordinate values, and likewise for the width in the x-dimension. Remember, the measurements for both the footprint and the credit card are in units of pixels, not centimeters or inches.

Next, knowing the precise measurements of the credit card, we can find the ratio between the pixel dimensions and the real dimensions of the card. This ratio can then be applied to the pixel dimensions of the footprint to ascertain its true measurements.

Determining the arch type is slightly more complicated, but it still requires counting in pixels after finding high-pressure areas (see Su et al., 2015, and Figure 11-1). With

the correct measurements, as shown in Figure 11-7, our store will be able to find the perfect shoe to fit each customer.

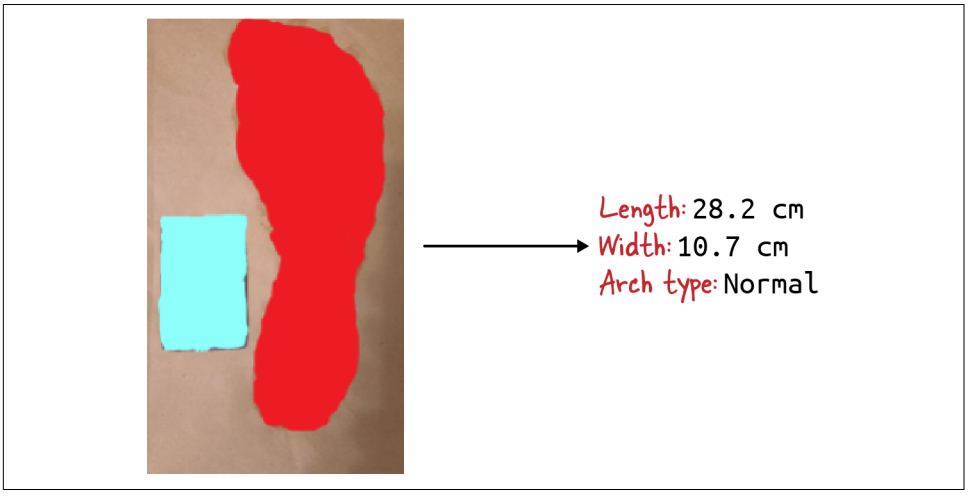

Figure 11-7. We can obtain the final measurements of the PCA-corrected masks using the reference pixel/centimeter ratio.

Counting

Counting the number of objects in an image is a problem with widespread applications, from estimating crowd sizes to identifying the potential yield of a crop from drone imagery. How many berries are in the photograph shown in Figure 11-8?

Figure 11-8. Berries on a plant. Photograph by the author.

Based on the techniques we have covered so far, you might choose one of the following approaches:

1. Train an object detection classifier to detect berries, and count the number of bounding boxes. However, the berries tend to overlap one another, and detection approaches might miss or combine berries.
2. Treat this as a segmentation problem. Find segments that contain berries and then, based on the properties of each cluster (for example, its size), determine the number of berries in each. The problem with this method is that it is not scale-invariant, and will fail if our berries are smaller or larger than typical. Unlike the foot-size measurement scenario discussed in the previous section, a reference object is difficult to incorporate into this problem.
3. Treat this as a regression problem, and estimate the number of berries from the entire image itself. This method has the same scale problems as the segmentation approach and it is difficult to find enough labeled images, although it has been successfully employed in the past for counting crowds and wildlife.

There are additional drawbacks to these approaches. For example, the first two methods require us to correctly classify berries, and the regression method ignores location, which we know is a significant source of information about the contents of images.

A better approach is to use density estimation on simulated images. In this section, we will discuss the technique and step through the method.

Density Estimation

For counting in situations like this where the objects are small and overlapping, there is an alternative approach, introduced in a 2010 paper by Victor Lempitsky and Andrew Zisserman, that avoids having to do object detection or segmentation and does not lose the location information. The idea is to teach the network to estimate the *density* of objects (here, berries) in patches of the image.[1]

In order to do density estimation, we need to have labels that indicate density. So, we take the original image and break it into smaller nonoverlapping patches, and we label each patch by the number of berry centers that lie in it, as shown in Figure 11-9. It is this value that the network will learn to estimate. In order to make sure that the total number of berries in the patches equals the number of berries in the image, we make sure to count a berry as being in a patch only if its center point is in the patch. Because some berries may be only partially in a patch, the grid input to the model has

1 Lempitsky and Zisserman introduced a custom loss function that they call the MESA distance, but the technique works well with good old mean squared error, so that's what we show.

to be larger than a patch. The input is shown by the dashed lines. Obviously, this makes the border of the image problematic, but we can simply pad the images as shown on the right in Figure 11-9 to deal with this.

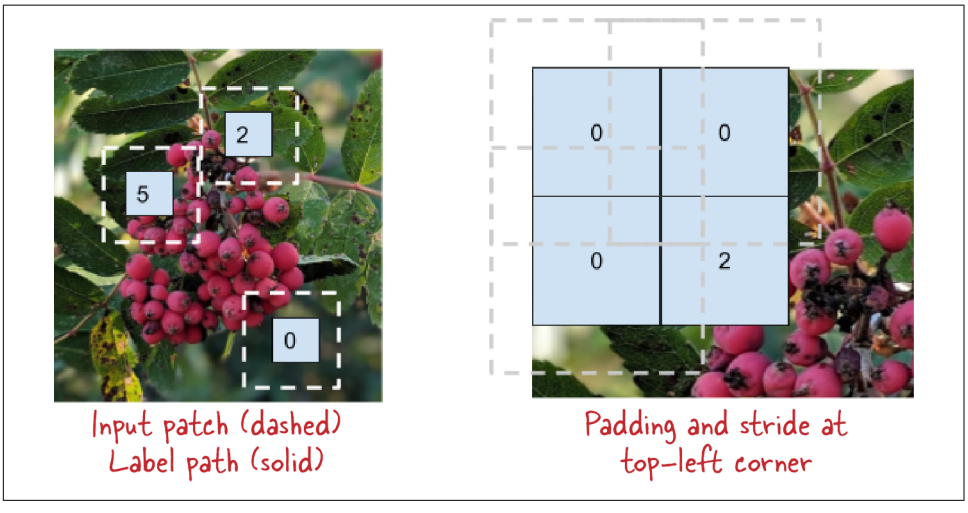

Figure 11-9. The model is trained on patches of the original image: the inputs and labels for three such patches are shown in the left panel. The labels consist of the number of berries whose center points lie within the inner square of each patch. The input patches require "same" padding on all sides whereas the label patches consist of valid pixels only.

This method is applicable beyond just counting berries, of course—it tends to work better than the alternatives at estimating crowd sizes, counting cells in biological images, and other such applications where there are lots of objects and some objects may be partially occluded by others. This is just like image regression, except that by using patches we increase the dataset size and teach the model to focus on density.

Extracting Patches

Given an image of berries and a label image consisting of 1s corresponding to the center point of each berry, the easiest way to generate the necessary input and label patches is to employ the TensorFlow function `tf.image.extract_patches()`. This function requires us to pass in a batch of images. If we have only one image, then we can expand the dimension by adding a batch size of 1 using `tf.expand_dims()`. The label image will have only one channel since it is Boolean, so we'll also have to add a depth dimension of 1 (the full code is in *11a_counting.ipynb* on GitHub):

```
def get_patches(img, label, verbose=False):
    img = tf.expand_dims(img, axis=0)
    label = tf.expand_dims(tf.expand_dims(label, axis=0), axis=-1)
```

Now we can call `tf.image.extract_patches()` on the input image. Notice in the following code that we ask for patches of the size of the dashed box (`INPUT_WIDTH`) but stride by the size of the smaller label patch (`PATCH_WIDTH`). If the dashed boxes are 64x64 pixels, then each of the boxes will have 64 * 64 * 3 pixel values. These values will be 4D, but we can reshape the patch values to a flattened array for convenience:

```
num_patches = (FULL_IMG_HEIGHT // PATCH_HEIGHT)**2
patches = tf.image.extract_patches(img,
    =[1, INPUT_WIDTH, INPUT_HEIGHT, 1],
    =[1, PATCH_WIDTH, PATCH_HEIGHT, 1],
    =[1, 1, 1, 1],
    ='SAME',
    ='get_patches')
patches = tf.reshape(patches, [num_patches, -1])
```

Next, we repeat the same operation on the label image:

```
labels = tf.image.extract_patches(label,
    =[1, PATCH_WIDTH, PATCH_HEIGHT, 1],
    =[1, PATCH_WIDTH, PATCH_HEIGHT, 1],
    =[1, 1, 1, 1],
    ='VALID',
    ='get_labels')
labels = tf.reshape(labels, [num_patches, -1])
```

There are two key differences in the code for the label patches versus that for the image patches. First, the size of the label patches is the size of the inner box only. Note also the difference in the padding specification. For the input image, we specify `padding=SAME`, asking TensorFlow to pad the input image with zeros and then extract all patches of the larger box size from it (see Figure 11-9). For the label image we ask only for fully valid boxes, so the image will not be padded. This ensures that we get the corresponding outer box of the image for every valid label patch.

The label image will now have 1s corresponding to the centers of all the objects we want to count. We can find the total number of such objects, which we will call the density, by summing up the pixel values of the label patch:

```
# the "density" is the number of points in the label patch
patch_labels = tf.math.reduce_sum(labels, axis=[1], name='calc_density')
```

Simulating Input Images

In their 2017 paper on yield estimation, Maryam Rahnemoor and Clay Sheppard showed that it is not even necessary to have real labeled photographs to train a neural network to count. To train their neural network to count tomatoes on a vine, the authors simply fed it simulated images consisting of red circles on a brown and green background. Because the method requires only simulated data, it is possible to quickly create a large dataset. The resulting trained neural network performed well on actual tomato plants. It is this approach, called *deep simulated learning*, that we

show next. Of course, if you actually have labeled data where each berry (or person in a crowd, or antibody in a sample) is marked, you can use that instead.

We will generate a blurred green background, simulate 25–75 "berries," and add them to the image (see Figure 11-10).

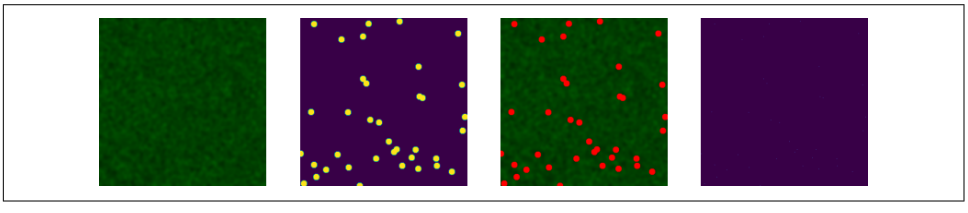

Figure 11-10. Simulating input images for counting "berries" on a green background. The first image is the background, the second the simulated berries, and the third the actual input image.

The key pieces of code are to randomly position a few berries:

```
num_berries = np.random.randint(25, 75)
berry_cx = np.random.randint(0, FULL_IMG_WIDTH, size=num_berries)
berry_cy = np.random.randint(0, FULL_IMG_HEIGHT, size=num_berries)
label = np.zeros([FULL_IMG_WIDTH, FULL_IMG_HEIGHT])
label[berry_cx, berry_cy] = 1
```

At each berry location in the label image, a red circle is drawn:

```
berries = np.zeros([FULL_IMG_WIDTH, FULL_IMG_HEIGHT])
for idx in range(len(berry_cx)):
    rr, cc = draw.circle(berry_cx[idx], berry_cy[idx],
                         radius=10,
                         shape=berries.shape)
    berries[rr, cc] = 1
```

The berries are then added to the green background:

```
img = np.copy(backgr)
img[berries > 0] = [1, 0, 0] # red
```

Once we have an image, we can generate image patches from it and obtain the density by adding up the berry centers that fall within the label patch. A few example patches and the corresponding densities are shown in Figure 11-11.

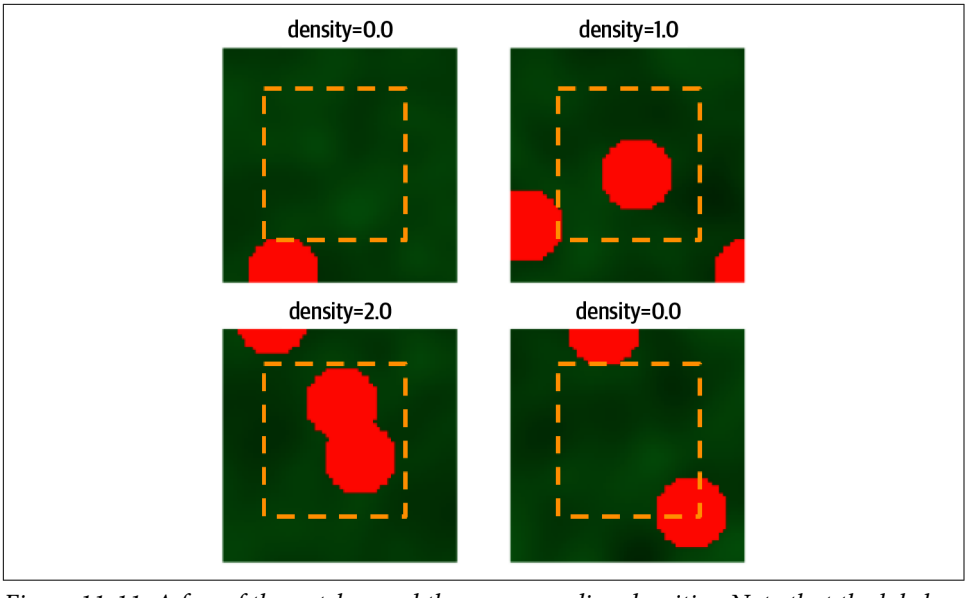

Figure 11-11. A few of the patches and the corresponding densities. Note that the label patch consists only of the center 50% of the input patch, and only red circles whose centers are in the label patch are counted in the density calculation.

Regression

Once we have the patch creation going, we can train a regression model on the patches to predict the density. First, we set up our training and evaluation datasets by generating simulated images:

```
def create_dataset(num_full_images):
    def generate_patches():
        for i in range(num_full_images):
            img, label = generate_image()
            patches, patch_labels = get_patches(img, label)
        for patch, patch_label in zip(patches, patch_labels):
            yield patch, patch_label

    return tf.data.Dataset.from_generator(
            generate_patches,
            (tf.float32, tf.float32), # patch, patch_label
            (tf.TensorShape([INPUT_HEIGHT*INPUT_WIDTH*IMG_CHANNELS]),
             tf.TensorShape([]))
    )
```

We can use any of the models we discussed in Chapter 3. For illustration purposes, let's use a simple ConvNet (the full code is available in *11a_counting.ipynb* on GitHub):

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
reshape (Reshape)            (None, 64, 64, 3)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 62, 62, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 31, 31, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 29, 29, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 12, 12, 64)        36928
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0
_________________________________________________________________
dense (Dense)                (None, 64)                589888
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65
=================================================================
Total params: 646,273
Trainable params: 646,273
Non-trainable params: 0
```

The key aspects to note about the architecture shown here are:

- The output is a single numeric value (density).
- The output node is a linear layer (so that the density can take any numeric value).
- The loss is mean square error.

These aspects make the model a regression model capable of predicting the density.

Prediction

Remember that the model takes a patch and predicts the density of berries in the patch. Given an input image, we have to break it into patches exactly as we did during training and carry out model prediction on all the patches, then sum up the predicted densities, as shown below:

```
def count_berries(model, img):
    num_patches = (FULL_IMG_HEIGHT // PATCH_HEIGHT)**2
    img = tf.expand_dims(img, axis=0)
    patches = tf.image.extract_patches(img,
        sizes=[1, INPUT_WIDTH, INPUT_HEIGHT, 1],
        strides=[1, PATCH_WIDTH, PATCH_HEIGHT, 1],
        rates=[1, 1, 1, 1],
        padding='SAME',
        name='get_patches')
```

```
    patches = tf.reshape(patches, [num_patches, -1])
    densities = model.predict(patches)
    return tf.reduce_sum(densities)
```

Predictions on some independent images are shown in Figure 11-12. As you can see, the predictions are within 10% of the actual numbers.

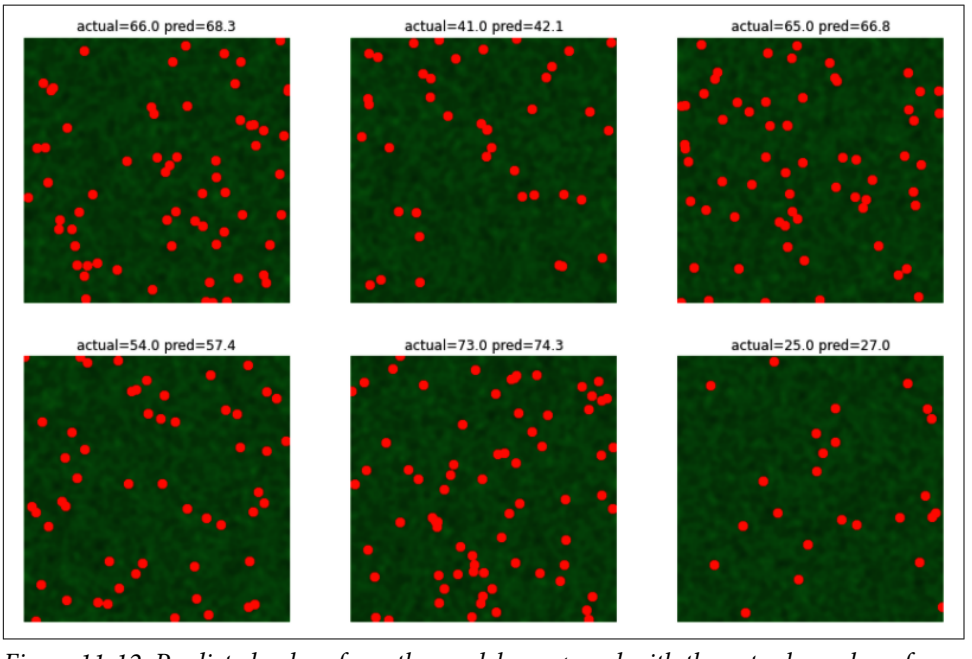

Figure 11-12. Predicted values from the model, compared with the actual number of objects in each image.

When we tried this on the real berry image with which we started this section, however, the estimate was considerably off. Addressing this might require simulating berries of different sizes, not just placing equally sized berries at random positions.

Pose Estimation

There are a variety of situations where we may desire to identify key parts of an object. A very common situation is to identify elbows, knees, face, and so on in order to identify the pose of a person. Therefore, this problem is termed *pose estimation* or *pose detection*. Pose detection can be useful to identify whether a subject is sitting, standing, dancing, or lying down or to provide advice on posture in sports and medical settings.

Given a photograph like the one in Figure 11-13, how can we identify the feet, knees, elbows, and hands in the image?

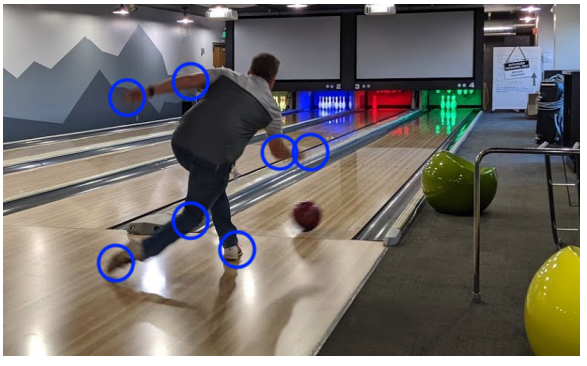

Figure 11-13. Identifying the relative position of key body parts is useful to provide coaching on improving a player's form. Photograph by the author.

In this section, we will discuss the technique and point you toward an already trained implementation. It is rarely necessary to train a pose estimation model from scratch—instead, you will use the output of an already trained pose estimation model to determine what the subjects in the images are doing.

PersonLab

The state-of-the-art approach was suggested in a 2018 paper by George Papandreou et al. They called it PersonLab, but the models that implement their approach now go by the name *PoseNet*. Conceptually, PoseNet consists of the steps depicted in Figure 11-14:

1. Use an object detection model to identify a heatmap of all the points of interest in the skeleton. These typically include the knees, elbows, shoulders, eyes, nose, and so on. For simplicity, we'll refer to these as *joints*. The heatmap is the score that is output from the classification head of the object detection model (i.e., before thresholding).
2. Anchored at each detected joint, identify the most likely location of nearby joints. The offset location of the elbow given a detected wrist is shown in the figure.
3. Use a voting mechanism to detect human poses based on the joints chosen based on steps 1 and 2.

In reality, steps 1 and 2 are carried out simultaneously by means of an object detection model (any of the models discussed in Chapter 4 may be used) that predicts a joint, its location, and the offset to nearby joints.

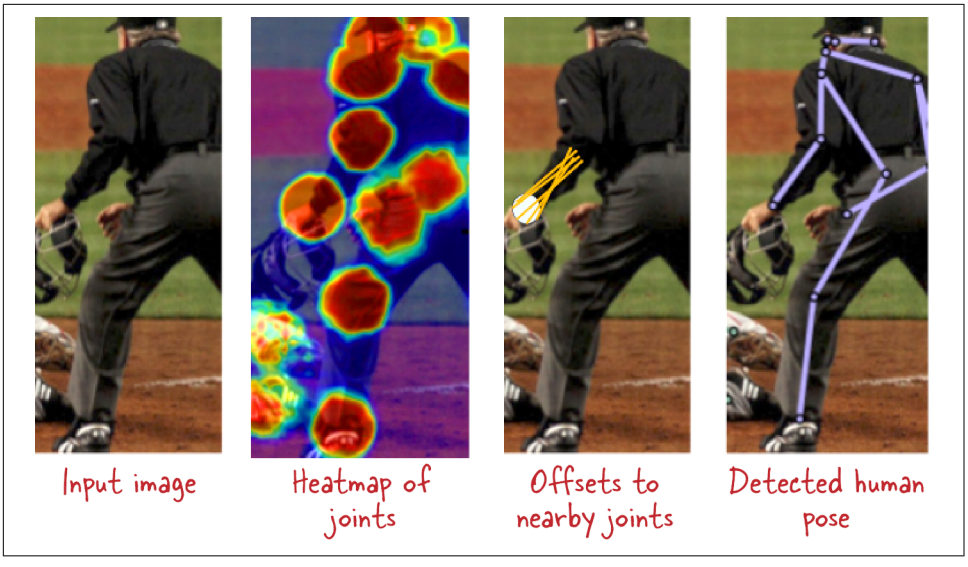

Figure 11-14. Identifying the relative position of key joints is useful to identify human poses. Image adapted from Papandreou et al., 2018.

We need steps 2 and 3 because it is not sufficient to simply run an object detection model to detect the various joints—it is possible that the model will miss some joints and identify spurious joints. That's why the PoseNet model also predicts offsets to nearby joints from the detected joints. For example, if the model detects a wrist, the wrist detection comes with an offset prediction for the location of the elbow joint. This helps in cases where, for some reason, the elbow was not detected. If the elbow was detected, we might now have three candidate locations for that joint—the elbow location from the heatmap and the elbow locations from the offset predictions of the wrist and the shoulder. Given all these candidate locations, a weighted voting mechanism called the Hough transform is used to determine the final location of the joint.

The PoseNet Model

PoseNet implementations are available in TensorFlow for Android and for the web browser. The TensorFlow JS implementation runs in a web browser and uses MobileNet or ResNet as the underlying architecture, but continues to refer to itself as PoseNet. An alternate implementation is provided by OpenPose.

The TensorFlow JS PoseNet model was trained to identify 17 body parts that include facial features (nose, leftEye, rightEye, leftEar, rightEar) and key limb joints (shoulder, elbow, wrist, hip, knee, and ankle) on both the left and the right side.

To try it out, you'll need to run a local web server—*11b_posenet.html* in the GitHub repository provides details. Load the `posenet` package and ask it to estimate a single pose (as opposed to an image with multiple people in it):

```
posenet.load().then(function(net) {
    const pose = net.estimateSinglePose(imageElement, {
        flipHorizontal: false
    });
    return pose;
})
```

Note that we ask that the image not be flipped. However, if you are processing selfie images, you might want to flip them horizontally to match the user experience of seeing mirrored images.

We can display the returned value as a JSON element using:

```
document.getElementById('output_json').innerHTML =
    "<pre>" + JSON.stringify(pose, null, 2) + "</pre>";
```

The JSON has the key points identified, along with their positions in the image:

```
{
    "score": 0.5220872163772583,
    "part": "leftEar",
    "position": {
        "x": 342.9179292671411,
        "y": 91.27406275411522
    }
},
```

We can use these to directly annotate the image as shown in Figure 11-15.

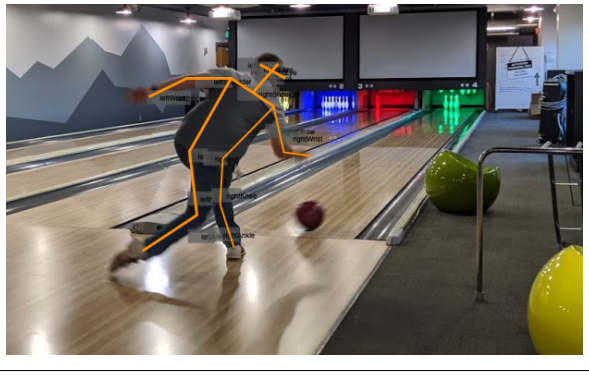

Figure 11-15. An annotated image, with the annotations derived from the output of PoseNet. Each of the light gray boxes contains a marker (e.g., rightWrist), and they have been connected by the skeleton.

The accuracy of PoseNet is determined by the accuracy of the underlying classification model (ResNet tends to be both larger and slower but more accurate than MobileNet, for example) and by the size of the output strides—the larger the stride, the larger the patches, so the precision of the output locations suffers.

These factors can be changed when PoseNet is loaded:

```
posenet.load({
    architecture: 'ResNet50',
    outputStride: 32, # default 257
    inputResolution: { width: 500, height: 900 },
    quantBytes: 2
});
```

A smaller output stride results in a more accurate model, at the expense of speed. The input resolution specifies the size the image is resized and padded to before it is fed into the PoseNet model. The larger the value, the more accurate it is, again at the cost of speed.

The MobileNet architecture takes a parameter called `multiplier` that specifies the depth multiplier for convolution operations. The larger the multiplier, the more accurate but slower the model is. The `quantBytes` parameter in ResNet specifies the number of bytes used for weight quantization. Using a value of `4` leads to a higher accuracy and larger models than using `1`.

Identifying Multiple Poses

To estimate the poses of multiple people in a single image, we use the same technique outlined in the previous section, with a few additional steps:

1. Use an image segmentation model to identify all the pixels that correspond to persons in the image.
2. Using the combination of joints, identify the most likely location of a specific body part, such as the nose.
3. Using the pixels from the segmentation mask found in step 1, and the likely connections identified in step 2, assign the person pixels to individual persons.

An example is shown in Figure 11-16. Again, any of the image segmentation models discussed in Chapter 4 may be used here.

Figure 11-16. Identifying the poses of multiple people in the image. Adapted from Papandreou et al., 2018.

When running PoseNet, you can ask it to estimate multiple poses using:

```
net.estimateMultiplePoses(image, {
    flipHorizontal: false,
    maxDetections: 5,
    scoreThreshold: 0.5,
    nmsRadius: 20
});
```

The key parameters here are the maximum number of people in the image (`maxDetections`), the confidence threshold for a person detection (`scoreThreshold`), and the distance within which two detections should suppress each other (`nmsRadius`, in pixels).

Next, let's look at the problem of supporting image search.

Image Search

eBay uses image search to improve the shopping experience (e.g., find the eyeglasses that a specific celebrity is wearing) and the listing experience (e.g., here are all the relevant technical specifications of this gadget you are trying to sell).

The crux of the problem in both cases is to find the image in the dataset that is most similar to a newly uploaded image. To provide this capability, we can use embeddings. The idea is that two images that are similar to each other will have embeddings that are also close to each other. So, to search for a similar image, we can simply search for a similar embedding.

Distributed Search

To enable searching for similar embeddings, we will have to create a search index of embeddings of the images in our dataset. Suppose we store this embedding index in a large-scale, distributed data warehouse such as Google BigQuery.

If we have embeddings of weather images in the data warehouse, then it becomes easy to search for "similar" weather situations in the past to some scenario in the present. Here's a SQL query that would do it:

```
WITH ref1 AS (
    SELECT time AS ref1_time, ref1_value, ref1_offset
    FROM `ai-analytics-solutions.advdata.wxembed`,
        UNNEST(ref) AS ref1_value WITH OFFSET AS ref1_offset
    WHERE time = '2019-09-20 05:00:00 UTC'
)
SELECT
    time,
    SUM( (ref1_value - ref[OFFSET(ref1_offset)])
        * (ref1_value - ref[OFFSET(ref1_offset)]) ) AS sqdist
FROM ref1, `ai-analytics-solutions.advdata.wxembed`
GROUP BY 1
ORDER By sqdist ASC
LIMIT 5
```

We are computing the Euclidean distance between the embedding at the specified timestamp (`ref1`) and every other embedding, and displaying the closest matches. The result, shown here:

<0xa0>	time	sqdist
0	2019-09-20 05:00:00+00:00	0.000000
1	2019-09-20 06:00:00+00:00	0.519979
2	2019-09-20 04:00:00+00:00	0.546595
3	2019-09-20 07:00:00+00:00	1.001852
4	2019-09-20 03:00:00+00:00	1.387520

makes a lot of sense. The image from the previous/next hour is the most similar, then images from +/– 2 hours, and so on.

Fast Search

In the SQL example in the previous section we searched the entire dataset, and we were able to do it efficiently because BigQuery is a massively scaled cloud data warehouse. A drawback of data warehouses, however, is that they tend to have high latency. We will not be able to get millisecond response times.

For real-time serving, we need to be smarter about how we search for similar embeddings. Scalable Nearest Neighbors (ScaNN), which we use in our next example, does search space pruning and provides an efficient way to find similar vectors.

Let's build a search index of the first hundred images of our 5-flowers dataset (normally, of course, we'd build a much larger dataset, but this is an illustration). We can create MobileNet embeddings by creating a Keras model:

```
layers = [
    hub.KerasLayer(
        "https://.../mobilenet_v2/...",
        input_shape=(IMG_WIDTH, IMG_HEIGHT, IMG_CHANNELS),
        trainable=False,
        name='mobilenet_embedding'),
    tf.keras.layers.Flatten()
]
model = tf.keras.Sequential(layers, name='flowers_embedding')
```

To create an embeddings dataset, we loop through the dataset of flower images and invoke the model's `predict()` function (the full code is in *11c_scann_search.ipynb* on GitHub):

```
def create_embeddings_dataset(csvfilename):
    ds = (tf.data.TextLineDataset(csvfilename).
          map(decode_csv).batch(BATCH_SIZE))
    dataset_filenames = []
    dataset_embeddings = []
    for filenames, images in ds:
        embeddings = model.predict(images)
        dataset_filenames.extend(
            [f.numpy().decode('utf-8') for f in filenames])
        dataset_embeddings.extend(embeddings)
    dataset_embeddings = tf.convert_to_tensor(dataset_embeddings)
    return dataset_filenames, dataset_embeddings
```

Once we have the training dataset, we can initialize the ScaNN searcher, specifying that the distance function to use is the cosine distance (we could also use Euclidean distance):

```
searcher = scann.scann_ops.builder(
    dataset_embeddings,
    NUM_NEIGH, "dot_product").score_ah(2).build()
```

This builds a tree for fast searching.

To search for the neighbors for some images, we obtain their embeddings and invoke the searcher:

```
_, query_embeddings = create_embeddings_dataset(
    "gs://cloud-ml-data/img/flower_photos/eval_set.csv"
)
neighbors, distances = searcher.search_batched(query_embeddings)
```

If you have only one image, call `searcher.search()`.

Some results are shown in Figure 11-17. We are looking for images similar to the first image in each row; the three closest neighbors are shown in the other panels. The results aren't too impressive. What if we used a better approach to create embeddings, rather than using the embeddings from MobileNet that are meant for transfer learning?

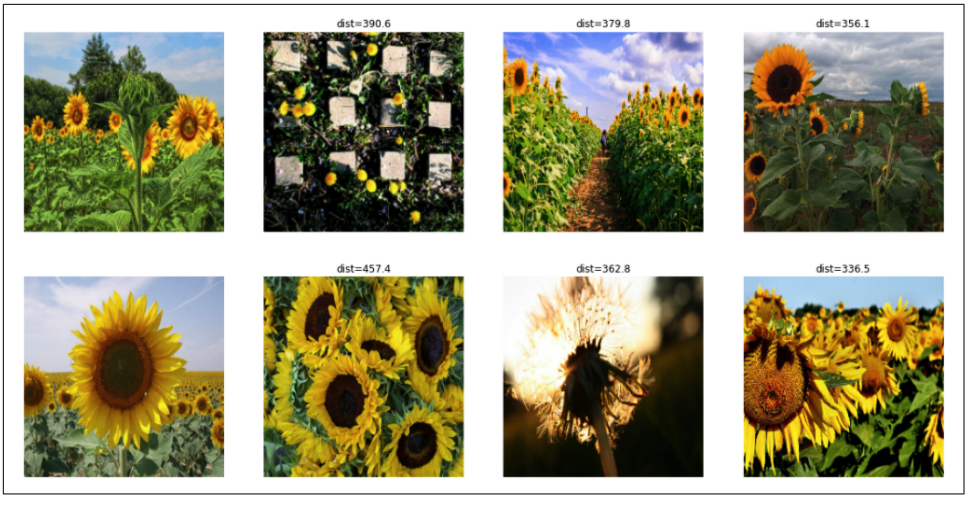

Figure 11-17. Searching for images similar to the first image in each row.

Better Embeddings

In the previous section we used MobileNet embeddings, which are derived from an intermediate bottleneck layer obtained by training a large image classification model. It is possible to use more customized embeddings. For example, when searching for face similarity, embeddings from a model trained to identify and verify faces will perform better than a generic embedding.

To optimize the embeddings for the purposes of facial search, a system called FaceNet uses triplets of matching/nonmatching face patches that are aligned based on facial features. The triplets consist of two matching and one nonmatching face thumbnails. A *triplet loss* function is used that aims to separate the positive pair from the negative one by the maximum possible distance. The thumbnails themselves are tight crops of the face area. The difficulty of the triplets shown to the network increases as the network trains.

Because of the ethical sensitivities that surround facial search and verification, we are not demonstrating an implementation of facial search in our repository or covering this topic any further. Code that implements the FaceNet technique is readily available online. Please make sure that you use AI responsibly and in a way that doesn't run afoul of governmental, industry, or company policies.

The triplet loss can be used to create embeddings that are clustered together by label such that two images with the same label have their embeddings close together and two images with different labels have their embeddings far apart.

The formal definition of triplet loss uses three images: the anchor image, another image with the same label (so that the second image and the anchor form a positive pair), and a third image with a different label (so that the third image and the anchor form a negative pair). Given three images, the loss of a triplet (a, p, n) is defined such that the distance $d(a, p)$ is pushed toward zero and the distance $d(a, n)$ is at least some margin greater than $d(a, p)$:

$$L - max(d(a, p) - d(a, n) + margin, 0)$$

Given this loss, there are three categories of negatives:

- Hard negatives, which are negatives that are closer to the anchor than the positive
- Easy negatives, which are negatives that are very far away from the anchor
- Semi-hard negatives, which are further away than the positive, but within the margin distance

In the FaceNet paper, Schroff et al. found that focusing on the semi-hard negatives yielded embeddings where images with the same label clustered together and were distinct from images with a different label.

We can improve the embeddings for our flower images by adding a linear layer and then training the model to minimize the triplet loss on those images, focusing on the semi-hard negatives:

```
layers = [
    hub.KerasLayer(
        "https://tfhub.dev/.../mobilenet_v2/feature_vector/4",
        input_shape=(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS),
        trainable=False,
        name='mobilenet_embedding'),
    tf.keras.layers.Dense(5, activation=None, name='dense_5'),
    tf.keras.layers.Lambda(lambda x: tf.math.l2_normalize(x, axis=1),
                           name='normalize_embeddings')
]
model = tf.keras.Sequential(layers, name='flowers_embedding')
model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
              loss=tfa.losses.TripletSemiHardLoss())
```

In the preceding code, the architecture ensures that the resulting embedding is of dimension 5 and that the embedding values are normalized.

Note that the definition of the loss means that we have to somehow ensure that each batch contains at least one positive pair. Shuffling and using a large enough batch size tends to work. In the 5-flowers example we used a batch size of 32, but it is a number

you have to experiment with. Assuming the k classes are equally distributed, the odds of a batch of size B containing at least one positive pair is:

$$1 - \frac{k - 1^B}{k}$$

For 5 classes and a batch size of 32, this works out to 99.9%. 0.1% is not zero, however, so in the ingest pipeline we have to discard batches that don't meet this criterion.

After training this model and plotting the embeddings on a test dataset (the full code is in *11c_scann_search.ipynb* on GitHub), we see that the resulting embeddings cluster with similar labels (see Figure 11-18).

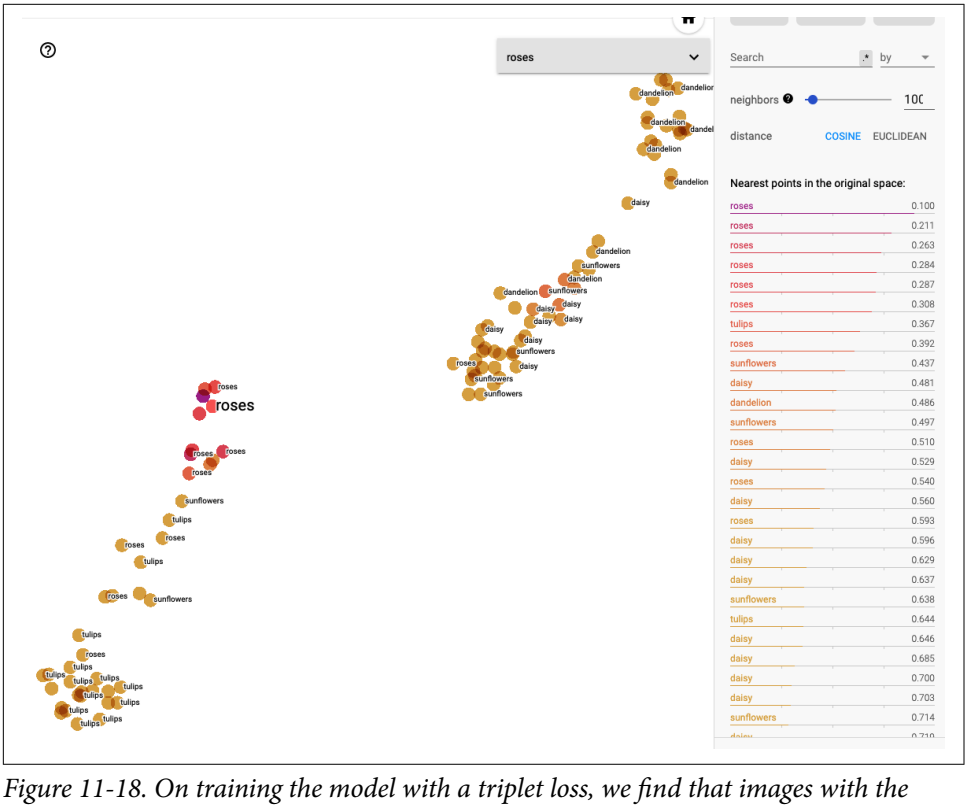

Figure 11-18. On training the model with a triplet loss, we find that images with the same labels cluster together in the embedding space.

This is also apparent in the results obtained when we search for similar images (see Figure 11-19)—the distances are smaller, and the images look much more similar than the ones in Figure 11-17.

Figure 11-19. On training the embedding with a triplet loss, the distances become smaller, and the close-by images are truly similar. Compare with Figure 11-17.

Summary

In this chapter, we explored a variety of use cases that build on the fundamental computer vision techniques. Object measurement can be done using reference objects, masks, and some image correction. Counting can be done through postprocessing of object detections. However, in some situations, a density estimate is better. Pose estimation is done by predicting the likelihood of the different joints at coarse-grained blocks within the image. Image search can be improved by training an embedding with a triplet loss and using a fast search method such as ScaNN.

In the next chapter, we will explore how to generate images, not just process them.

CHAPTER 12

Image and Text Generation

So far in this book, we have focused on computer vision methods that act on images. In this chapter, we will look at vision methods that can *generate* images. Before we get to image generation, though, we have to learn how to train a model to understand what's in an image so that it knows what to generate. We will also look at the problem of generating text (captions) based on the content of an image.

The code for this chapter is in the *12_generation* folder of the book's GitHub repository. We will provide file names for code samples and notebooks where applicable.

Image Understanding

It's one thing to know what components are in an image, but it's quite another to actually understand what is happening in the image and to use that information for other tasks. In this section, we will quickly recap embeddings and then look at various methods (autoencoders and variational autoencoders) to encode an image and learn about its properties.

Embeddings

A common problem with deep learning use cases is lack of sufficient data, or data of high enough quality. In Chapter 3 we discussed transfer learning, which provides a way to extract embeddings that were learned from a model trained on a larger dataset, and apply that knowledge to train an effective model on a smaller dataset.

With transfer learning, the embeddings we use were created by training the model on the same task, such as image classification. For instance, suppose we have a ResNet50 model that was trained on the ImageNet dataset, as shown in Figure 12-1.

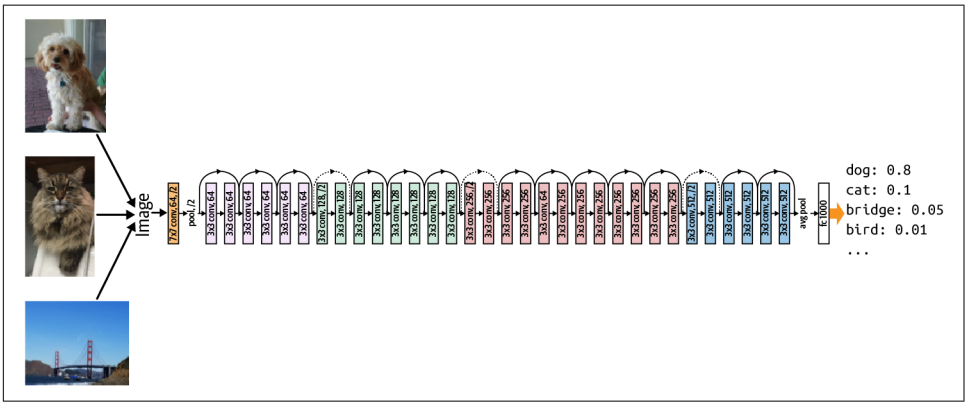

Figure 12-1. Training a ResNet model to classify images.

To extract the learned image embeddings, we would choose one of the model's intermediate layers—usually the last hidden layer—as the numerical representation of the input image (see Figure 12-2).

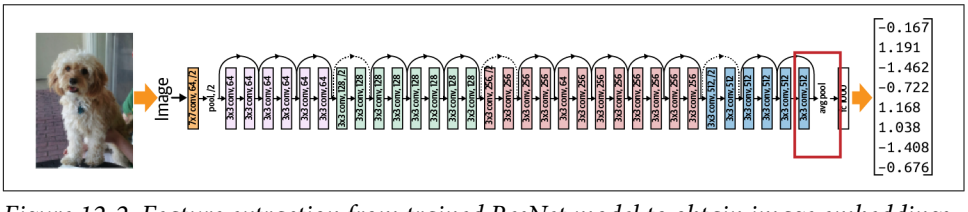

Figure 12-2. Feature extraction from trained ResNet model to obtain image embeddings.

There are two problems with this approach of creating embeddings by training a classification model and using its penultimate layer:

- To create these embeddings, we need a large, labeled dataset of images. In this example, we trained a ResNet50 on ImageNet to get the image embeddings. However, these embeddings will work well only for the types of images found in ImageNet—photographs found on the internet. If you have a different type of image (such as diagrams of machine parts, scanned book pages, architectural drawings, or satellite imagery), the embeddings learned from the ImageNet dataset may not work so well.
- The embeddings reflect the information that is relevant to determining the label of the image. By definition, therefore, many of the details of the input images that

are not relevant to this specific classification task may not be captured in the embeddings.

What if you want an embedding that works well for images other than photographs, you don't have a large labeled dataset of such images, and you want to capture as much of the information content in the image as possible?

Auxiliary Learning Tasks

Another way to create embeddings is to use an *auxiliary learning task*. An auxiliary task is a task other than the actual supervised learning problem we are trying to solve. This task should be one for which large amounts of data are readily available. For example, in the case of text classification we can create the text embeddings using an unrelated problem, such as predicting the next word of a sentence, for which there is already copious and readily available training data. The weight values of some intermediate layer can then be extracted from the auxiliary model and used to represent text for various other unrelated tasks. Figure 12-3 shows an example of this kind of text or word embedding where a model is trained to predict the next word in a sentence. Using the words "the cat sat" as input, such a model would be trained to predict the word "on." The input words are first one-hot encoded, but the penultimate layer of the prediction model, if it has four nodes, will learn to represent the input words as a four-dimensional embedding.

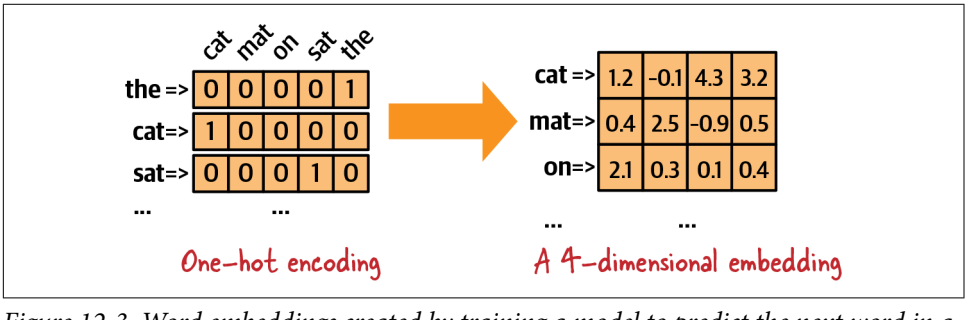

Figure 12-3. Word embeddings created by training a model to predict the next word in a sentence can be used for an unrelated task, such as text classification. The illustration shows the word encoding before (left) and after (right) the auxiliary task.

Autoencoders take advantage of an auxiliary learning task for images, similar to the predict-the-next-word model in the case of text. We'll look at these next.

Autoencoders

A great auxiliary learning task to learn image embeddings is to use autoencoders. With an autoencoder, we take the image data and pass it through a network that bottlenecks it into a smaller internal vector, and then expands it back out into the

dimensionality of the original image. When we train the autoencoder, the input image itself functions as its own label. This way we are essentially learning *lossy compression*, or how to recover the original image despite squeezing the information through a constrained network. The hope is that we are squeezing out the noise from the data and learning an efficient map of the signal.

With embeddings that are trained through a supervised task, any information in the inputs that isn't useful or related to the label usually gets pruned out with the noise. On the other hand, with autoencoders, since the "label" is the entire input image, every part of the input is relevant to the output, and therefore hopefully much more of the information is retained from the inputs. Because autoencoders are self-supervised (we don't need a separate step to label the images), we can train on much more data and get a greatly improved encoding.

Typically, the encoder and decoder form an hourglass shape as each progressive layer in the encoder shrinks in dimensionality and each progressive layer in the decoder expands in dimensionality, as seen in Figure 12-4. With the shrinking dimensionality in the encoder and the expanding dimensionality in the decoder, at some point the dimensionality reaches a minimum size at the end of the encoder and the start of the decoder, represented by the two-pixel, single-channel block in the middle of Figure 12-4. This latent vector is a concise representation of the inputs, where the data is being forced through a bottleneck.

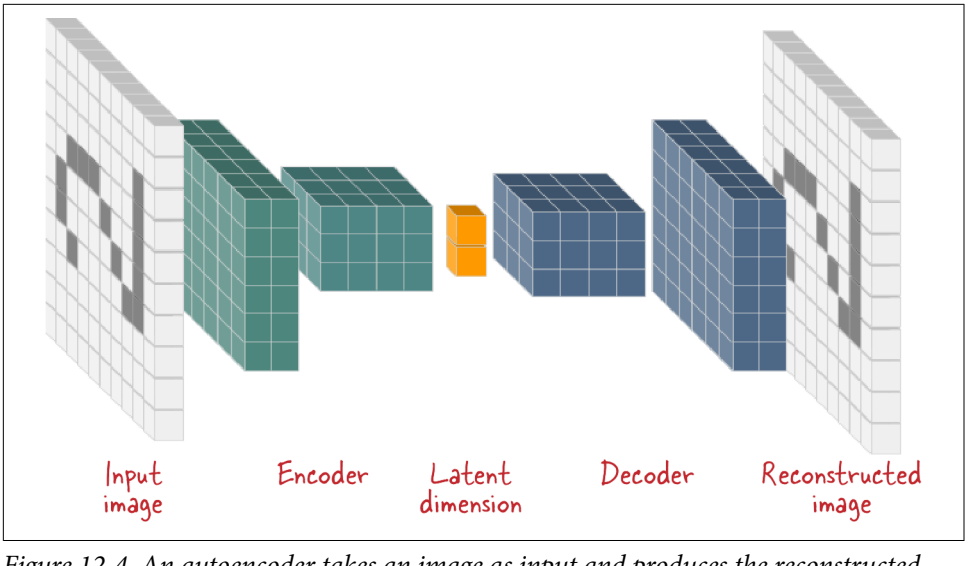

Figure 12-4. An autoencoder takes an image as input and produces the reconstructed image as output.

So, how big should the latent dimension be? As with other types of embeddings, there is a trade-off between compression and expressivity. If the dimensionality of the

latent space is too small, there won't be enough expressive power to fully represent the original data—some of the signal will be lost. When this representation is decompressed back to the original size, too much information will be missing to get the desired outputs. Conversely, if the dimensionality of the latent space is too large, then even though there will be ample space to store all of the desired information there will also be space to encode some of the unwanted information (i.e., the noise). The ideal size for the latent dimension is thus something to tune through experimentation. Typical values are 128, 256, or 512, though of course this depends on the sizes of the layers of the encoder and the decoder.

Next we'll look at implementing an autoencoder, starting with its architecture.

Architecture

To simplify our discussion and analysis of the autoencoder architecture, we'll pick a simple dataset of handwritten digits called MNIST to apply the autoencoder to (the full code is in *12a_autoencoder.ipynb* on GitHub). The input images are of size 28x28 and consist of a single grayscale channel.

The encoder starts with these 28x28 inputs and then progressively squeezes the information into fewer and fewer dimensions by passing the inputs through convolutional layers to end up at a latent dimension of size 2:

```
encoder = tf.keras.Sequential([
    keras.Input(shape=(28, 28, 1), name="image_input"),
    layers.Conv2D(32, 3, activation="relu", strides=2, padding="same"),
    layers.Conv2D(64, 3, activation="relu", strides=2, padding="same"),
    layers.Flatten(),
    layers.Dense(2) # latent dim
], name="encoder")
```

The decoder will have to reverse these steps using `Conv2DTranspose` layers (also known as deconvolution layers, covered in Chapter 4) wherever the encoder has a `Conv2D` (convolution) layer:

```
decoder = tf.keras.Sequential([
    keras.Input(shape=(latent_dim,), name="d_input"),
    layers.Dense(7 * 7 * 64, activation="relu"),
    layers.Reshape((7, 7, 64)),
    layers.Conv2DTranspose(32, 3, activation="relu",
                           strides=2, padding="same"),
    layers.Conv2DTranspose(1, 3, activation="sigmoid",
                           strides=2, padding="same")
], name="decoder")
```

Once we have the encoder and decoder blocks, we can tie them together to form a model that can be trained.

Reverse Operations for Common Keras Layers

When writing autoencoders, it is helpful to know the "reverse" operations for common Keras layers. The reverse of a `Dense` layer that takes inputs of shape `s1` and produces outputs of shape `s2`:

```
Dense(s2)(x) # x has shape s1
```

is a `Dense` layer that takes `s2`s and produces `s1`s:

```
Dense(s1)(x) # x has shape s2
```

The reverse of a `Flatten` layer will be a `Reshape` layer with the input and output shapes similarly swapped.

The reverse of a `Conv2D` layer is a `Conv2DTranspose` layer. Instead of taking a neighborhood of pixels and downsampling them into one pixel, it expands one pixel into a neighborhood to upsample the image. Keras also has an `Upsampling2D` layer. Upsampling is a cheaper operation since it involves no trainable weights and just repeats the source pixel values, or does bilinear interpolation. On the other hand, `Conv2DTranspose` deconvolves with a kernel, and thus weights, which are learned during model training to get a superior upsampled image.

Training

The model to be trained consists of the encoder and decoder blocks chained together:

```
encoder_inputs = keras.Input(shape=(28, 28, 1))
x = encoder(encoder_inputs)
decoder_output = decoder(x)
autoencoder = keras.Model(encoder_inputs, decoder_output)
```

The model has to be trained to minimize the reconstruction error between the input and output images—for example, we could compute the mean squared error between the input and output images and use that as the loss function. This loss function can be used in backpropagation to calculate the gradients and update the weights of the encoder and decoder subnetworks:

```
autoencoder.compile(optimizer=keras.optimizers.Adam(), loss='mse')
history = autoencoder.fit(mnist_digits, mnist_digits,
                          epochs=30, batch_size=128)
```

Latent vectors

Once the model is trained, we can drop the decoder and use the encoder to convert images into latent vectors:

```
z = encoder.predict(img)
```

If the autoencoder has successfully learned how to reconstruct the image, the latent vectors for similar images will tend to cluster, as shown in Figure 12-5. Notice that the 1s, 2s, and 0s occupy different parts of the latent vector space.

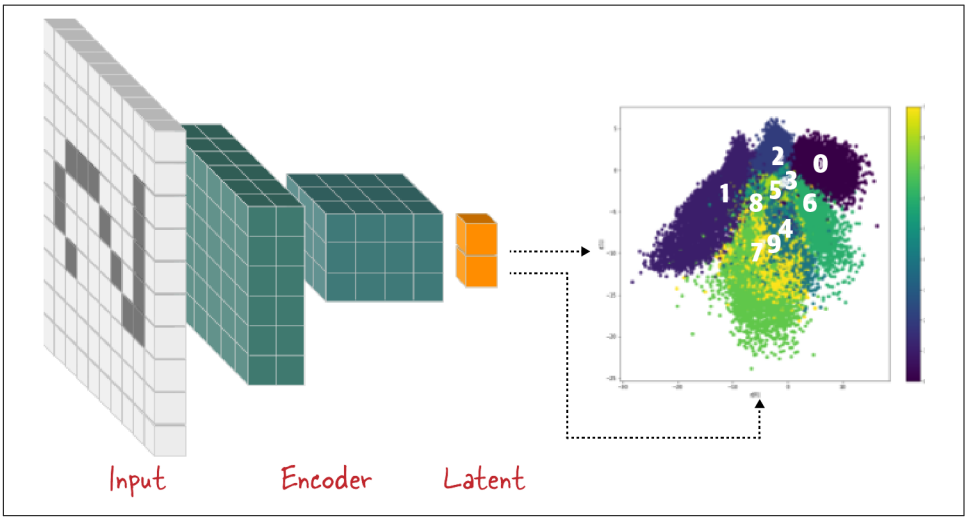

Figure 12-5. The encoder compresses input images into the latent representation. The latent representations for specific digits cluster together. We are able to represent the latent representations as points on a 2D graph because each latent representation is two numbers (x, y).

Because the entire information content of the input image has to flow through the bottleneck layer, the bottleneck layer after training will retain enough information for the decoder to be able to reconstruct a close facsimile of the input image. Therefore, we can train autoencoders for dimensionality reduction. The idea, as in Figure 12-5, is to drop the decoder and use the encoder to convert images into latent vectors. These latent vectors can then be used for downstream tasks such as classification and clustering, and the results may be better than those achieved with classical dimensionality reduction techniques like principal component analysis. If the autoencoder uses nonlinear activations, the encoding can capture nonlinear relationships between the input features, unlike PCA, which is solely a linear method.

A different application might be to use the decoder to turn latent vectors provided not by an encoded image but by a user into generated images, as shown in Figure 12-6.

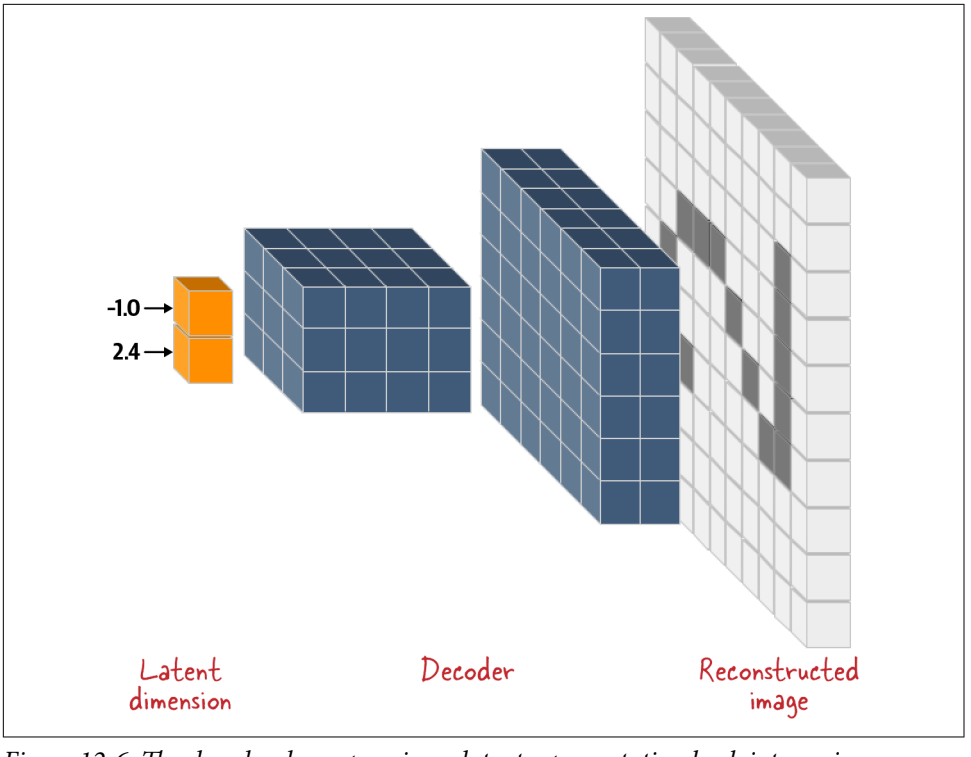

Figure 12-6. The decoder decompressing a latent representation back into an image.

While this works tolerably well for the very simple MNIST dataset, it doesn't work in practice on more complex images. Indeed, in Figure 12-7 we can see some of the shortcomings even on the handwritten digits in MNIST—while the digits look realistic in some parts of the latent space, in other places they are unlike any digits that we know. For example, look at the center left of the image, where 2s and 8s have been interpolated into something nonexistent. The reconstructed images are completely meaningless. Notice also that there is a preponderance of 0s and 2s, but not as many 1s as one would expect looking at the overlapping clusters in Figure 12-5. While it will be relatively easy to generate 0s and 2s, it will be quite difficult to generate 1s—we'd have to get the latent vector just right to get a 1 that doesn't look like a 2 or a 5!

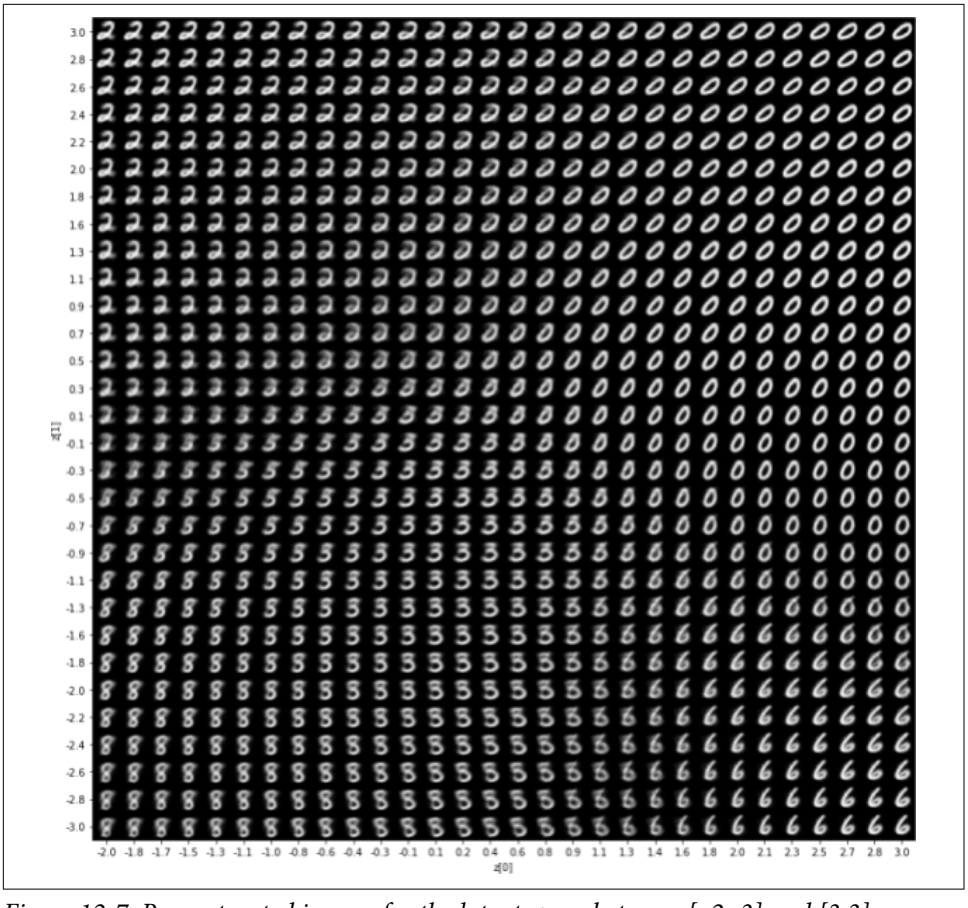

Figure 12-7. Reconstructed images for the latent space between [–2,-3] and [3,3].

What could be the matter? There are vast regions of the latent space (note the whitespace in Figure 12-5) that do not correspond to valid digits. The training task does not care about the whitespace at all, but has no incentive for minimizing it. The trained model does a great job when using both the encoder and decoder together, which is the original task we asked the autoencoder to learn. We passed images into the encoder subnetwork, which compressed the data down into vectors (the learned latent representations of those images). We then decompressed those representations with the decoder subnetwork to reconstruct the original images. The encoder has learned a mapping from image to latent space using a highly nonlinear combination of perhaps millions of parameter weights. If we naively try to create our own latent vector, it won't conform to the more nuanced latent space that the encoder has created. Therefore, without the encoder, our randomly chosen latent vector isn't very likely to result in a good encoding that the decoder can use to generate a quality image.

We've seen now that we can use autoencoders to reconstruct images by using their two subnetworks: the encoder and decoder. Furthermore, by dropping the decoder and using only the encoder, we now can encode images nonlinearly into latent vectors that we can then use as embeddings in a different task. However, we've also seen that naively trying the converse of dropping the encoder and using the decoder to generate an image from a user-provided latent vector doesn't work.

If we truly want to use the decoder of an autoencoder-type structure to generate images, then we need to develop a way to organize or regularize the latent space. This can be achieved by mapping points that are close in latent space to points that are close in image space and filling the latent space map so that all points make something sensical rather than creating small islands of reasonable outputs in an ocean of unmapped noise. This way we can generate our own latent vectors, and the decoder will be able to use those to make quality images. This forces us to leave classic autoencoders behind and takes us to the next evolution: the variational autoencoder.

Variational Autoencoders

Without an appropriately organized latent space, there are usually two main problems with using autoencoder-type architectures for image generation. First, if we were to generate two points that are close in the latent space, we would expect the outputs corresponding to those points to be similar to each other after they've been passed through the decoder. For instance, as Figure 12-8 depicts, if we have trained our autoencoder on geometric shapes such as circles, squares, and triangles, if we create two points that are close in the latent space we assume they should both be latent representations of either circles, squares, triangles, or some interpolation in between. However, since the latent space hasn't been explicitly regularized, one of the latent points might generate a triangle whereas the other latent point might generate a circle.

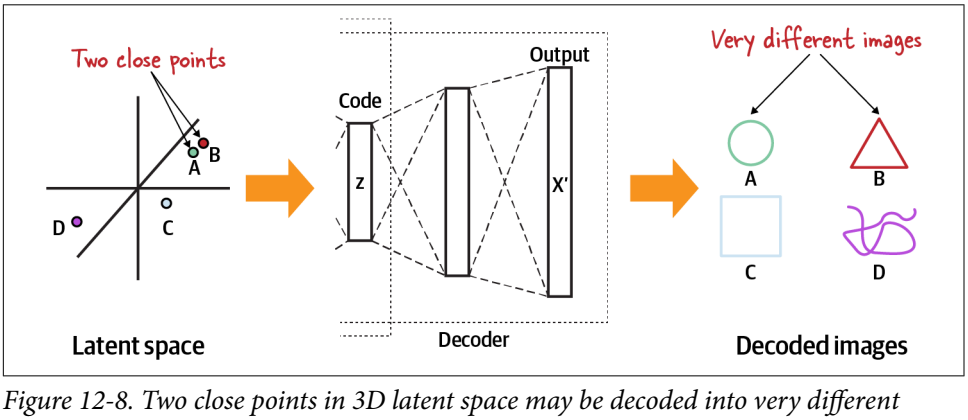

Figure 12-8. Two close points in 3D latent space may be decoded into very different images.

Second, after training an autoencoder for a long time and observing good reconstructions, we would expect that the encoder will have learned where each archetype of our images (for instance, circles or squares in the shapes dataset) best fits within the latent space, creating *n*-dimensional subdomains for each archetype that can overlap with other subdomains. For example, there may be a region of the latent space where the square-type images mostly reside, and another region of the latent space where circle-like images have been organized. Furthermore, where they overlap, we'll get shapes that lie somewhere in between on some square–circle spectrum.

Yet, because the latent space is not explicitly organized in autoencoders, random points in the latent space can return completely meaningless, unrecognizable images after being passed through the decoder, as shown in Figure 12-9. Instead of the imagined vast overlapping spheres of influence, small isolated islands are formed.

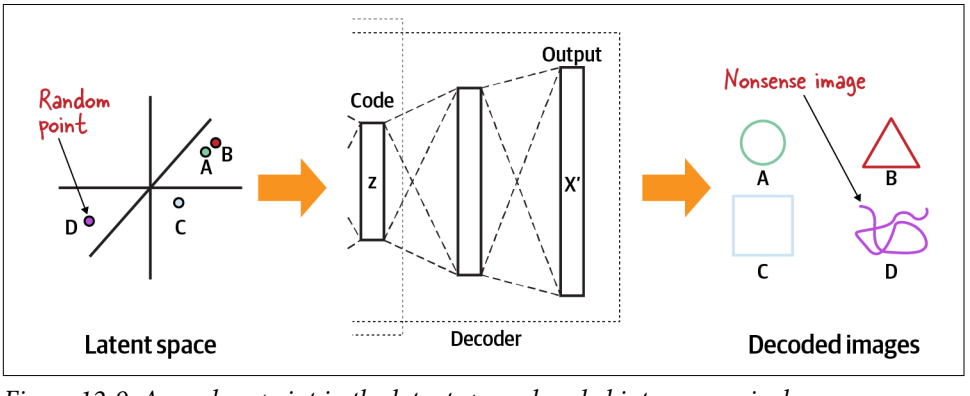

Figure 12-9. A random point in the latent space decoded into a meaningless, nonsense image.

Now, we can't blame autoencoders too much. They are doing exactly what they were designed to do, namely reconstructing their inputs with the goal of minimizing a reconstruction loss function. If having small isolated islands achieves that, then that is what they'll learn to do.

Variational autoencoders (VAEs) were developed in response to classic autoencoders being unable to use their decoders to generate quality images from user-generated latent vectors.

VAEs are generative models that can be used when, instead of just wanting to discriminate between classes, such as in a classification task where we create a decision boundary in a latent space (possibly satisfied with barely separating classes), we want to create *n*-dimensional bubbles that encompass similar training examples. This distinction is visualized in Figure 12-10.

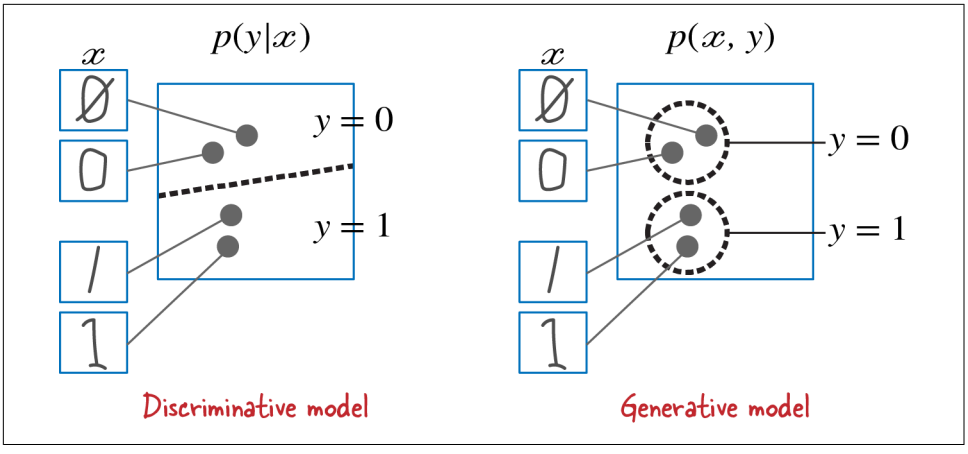

Figure 12-10. Discriminative model conditional versus generative model joint probability distributions.

Discriminative models, including popular image ML models for tasks such as classification and object detection, learn a conditional probability distribution that models the decision boundary between the classes. Generative models, on the other hand, learn a joint probability distribution that explicitly models the distribution of each class. The conditional probability distribution isn't lost—we can still do classification using Bayes' theorem, for example.

Fortunately, most of the architecture of variational autoencoders is the same as that of classic autoencoders: the hourglass shape, reconstruction loss, etc. However, the few additional complexities allow VAEs to do what autoencoders can't: image generation.

This is illustrated in Figure 12-11, which shows that both of our latent space regularity problems have been solved. The first problem of close latent points generating very different decoded images has been fixed, and now we are able to create similar images that smoothly interpolate through the latent space. The second problem of points in the latent space generating meaningless, nonsense images has also been fixed, and now we can generate plausible images. Remember, these images may not actually be exactly like the images the model was trained on, but may be in between some of the main archetypes because of the smooth overlapping regions within the learned organized latent space.

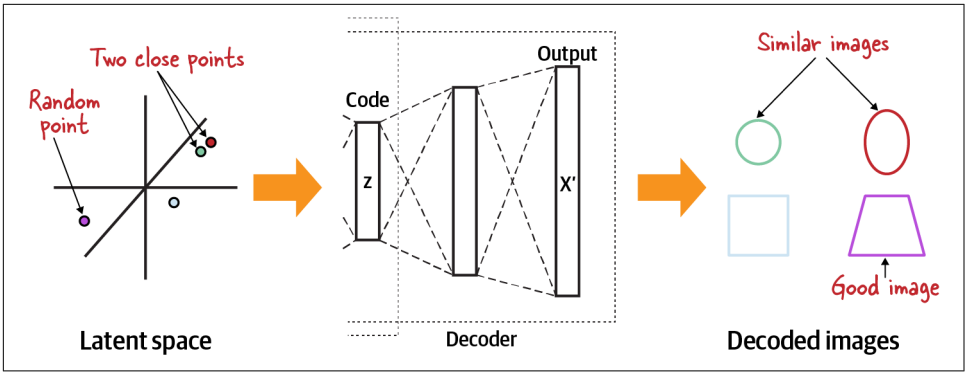

Figure 12-11. Both problems have been solved with an organized, regularized latent space.

Rather than having just a latent vector in between the encoder and decoder networks, which is essentially a point within the latent space, variational autoencoders train the encoder to produce parameters of a probability distribution and make the decoder randomly sample using them. The encoder no longer outputs a vector that describes a point within the latent space, but the parameters of a probability distribution. We can then sample from that distribution and pass those sampled points to our decoder to decompress them back into images.

In practice, the probability distribution is a standard normal distribution. This is because a mean close to zero will help prevent encoded distributions being too far apart and appearing as isolated islands. Also, a covariance close to the identity helps prevent the encoded distributions from being too narrow. The left side of Figure 12-12 shows what we are trying to avoid, with small, isolated distributions surrounded by voids of meaningless nonsense.

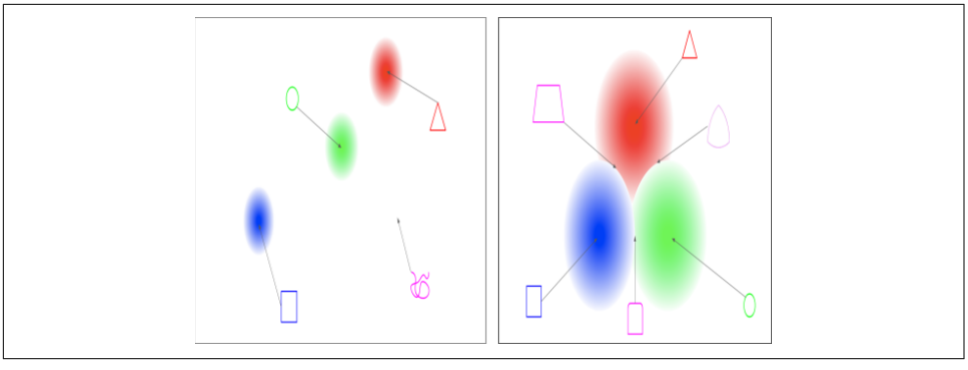

Figure 12-12. What we're trying to avoid (left) and what we're trying to achieve (right). We don't want small, isolated island distributions surrounded by vast voids of nonsense; we want the entire space covered by n-dimensional bubbles, as shown on the right. The goal is to have smoothly overlapping distributions without large gaps.

The image on the right in Figure 12-12 shows smoothly overlapping distributions without large gaps, which is exactly what we want for great image generation. Notice the interpolation where two distributions intersect. There is in fact a smooth gradient encoded over the latent space. For instance, we could start at the deep heart of the triangle distribution and move directly toward the circle distribution. We would begin with a perfect triangle, and with every step we took toward the circle distribution our triangle would get rounder and rounder until we reached the deep heart of the circle distribution, where we would now have a perfect circle.

To be able to sample from these distributions, we need both the mean vector and the covariance matrix. Therefore, the encoder network will output a vector for the distribution's mean and a vector for the distribution's covariance. To simplify things, we assume that these are independent. Therefore, the covariance matrix is diagonal, and we can simply use that instead of having an n^2-long vector of mostly zeros.

Now let's look at how to implement a VAE, starting with its architecture.

Architecture

In TensorFlow, a VAE encoder has the same structure as in a classic autoencoder, except instead of a single latent vector we now have a vector with two components, mean and variance (the full code is in *12b_vae.ipynb* on GitHub):

```
encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(
    64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten(name="e_flatten")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x) # same as autoencoder
```

However, in addition to the `z_mean`, we need two additional outputs from the encoder. And because our model now has multiple outputs, we can no longer use the Keras Sequential API; instead, we have to use the Functional API. The Keras Functional API is more like standard TensorFlow, where inputs are passed into a layer and the layer's outputs are passed to another layer as its inputs. Any arbitrarily complex directed acyclic graph can be made using the Keras Functional API:

```
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
z = Sampling()(z_mean, z_log_var)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z], name="encoder")
```

The sampling layer needs to sample from a normal distribution parameterized by the outputs of our encoder layers rather than using a vector from the final encoder layer as in non-variational autoencoders. The code for the sampling layer in TensorFlow looks like this:

```
class Sampling(tf.keras.layers.Layer):
    """Uses (z_mean, z_log_var) to sample z, the vector encoding a digit.
    """
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(input=z_mean)[0]
        dim = tf.shape(input=z_mean)[1]
        epsilon = tf.random.normal(shape=(batch, dim))
        return z_mean + tf.math.exp(x=0.5 * z_log_var) * epsilon
```

The VAE decoder is identical to that in a non-variational autoencoder—it takes the latent vector `z` produced by the encoder and decodes it into an image (specifically, the reconstruction of the original image input):

```
z_mean, z_log_var, z = encoder(encoder_inputs) # 3 outputs now
decoder_output = decoder(z)
vae = keras.Model(encoder_inputs, decoder_output, name="vae")
```

Loss

A variational autoencoder's loss function contains an image reconstruction term, but we cannot just use the mean squared error (MSE). In addition to the reconstruction error, the loss function also contains a regularization term called the Kullback–Leibler divergence, which is essentially a penalty for the encoder's normal distribution (parameterized by mean μ and standard deviation σ) not being a perfect standard normal distribution (with mean 0 and a standard deviation of identity):

$$L = \left\|x - \hat{x}\right\|^2 + KL\left(N\left(\mu_x, \sigma_x\right), N(0, I)\right)$$

We therefore modify the encoder loss function as follows:

```
def kl_divergence(z_mean, z_log_var):
    kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) -
```

```
                        tf.exp(z_log_var))
    return tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
encoder.add_loss(kl_divergence(z_mean, z_log_var))
```

The overall reconstruction loss is the sum of the per-pixel losses:

```
def reconstruction_loss(real, reconstruction):
    return tf.reduce_mean(
        tf.reduce_sum(
            keras.losses.binary_crossentropy(real, reconstruction),
            axis=(1, 2)
        )
    )
vae.compile(optimizer=keras.optimizers.Adam(),
            loss=reconstruction_loss, metrics=["mse"])
```

We then train the encoder/decoder combination with the MNIST images functioning as both the input features and the labels:

```
history = vae.fit(mnist_digits, mnist_digits, epochs=30,
                  batch_size=128)
```

Because the variational encoder has been trained to include the binary cross-entropy in its loss function, it takes into account the separability of the different classes in the images. The resulting latent vectors are more separable, occupy the entire latent space, and are better suited to generation (see Figure 12-13).

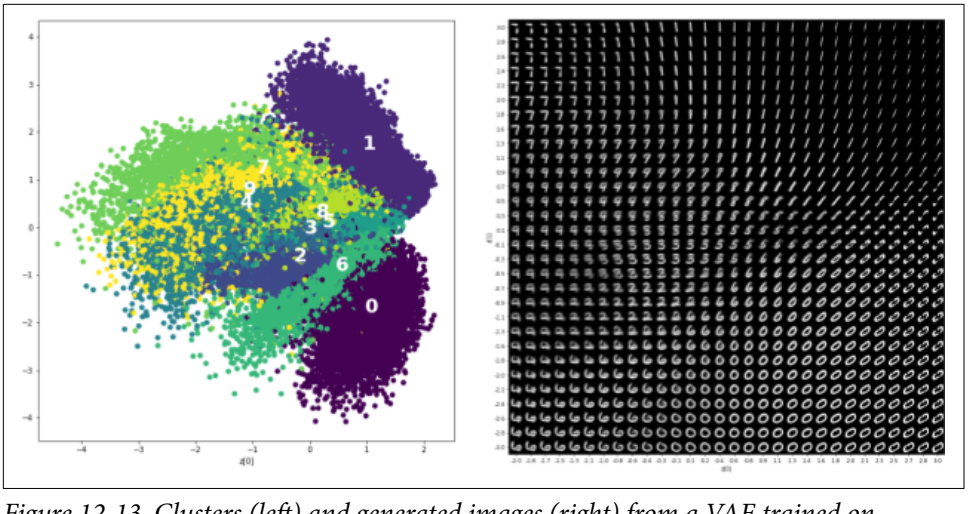

Figure 12-13. Clusters (left) and generated images (right) from a VAE trained on MNIST.

Variational autoencoders are able to create images that look just like their inputs. But what if we want to generate entirely new images? We'll look at image generation next.

Image Generation

Image generation is an important and rapidly growing field that goes well beyond just generating numbers and faces for fun; it has many important use cases for individuals and businesses alike, such as image restoration, image translation, super-resolution, and anomaly detection.

We saw previously how VAEs re-create their inputs; however, they aren't particularly successful at creating entirely new images that are similar to but different from the images in the input dataset. This is especially true if the generated images need to be perceptually real—for example, if, given a dataset of images of tools, we want a model to generate new pictures of tools that have different characteristics than the tools in the training images. In this section, we will discuss methods to generate images in cases like these (GANs, cGANs), and some of the uses (such as translation, super-resolution, etc.) to which image generation can be put.

Generative Adversarial Networks

The type of model most often used for image generation is a *generative adversarial network* (GAN). GANs borrow from game theory by pitting two networks against each other until an equilibrium is reached. The idea is that one network, the *generator*, will constantly try to create better and better reproductions of the real images, while the other network, the *discriminator*, will try to get better and better at detecting the difference between the reproductions and the real images. Ideally, the generator and discriminator will establish a Nash equilibrium so that neither network can completely dominate the other one. If one of the networks does begin to dominate the other, not only will there be no way for the "losing" network to recover, but also this unequal competition will prevent the networks from improving each other.

The training alternates between the two networks, each one becoming better at what it does by being challenged by the other. This continues until convergence, when the generator has become so good at creating realistic fake images that the discriminator is only randomly guessing which images are real (coming from the dataset) and which images are fake (coming from the generator).

The generator and discriminator are both neural networks. Figure 12-14 shows the overall training architecture.

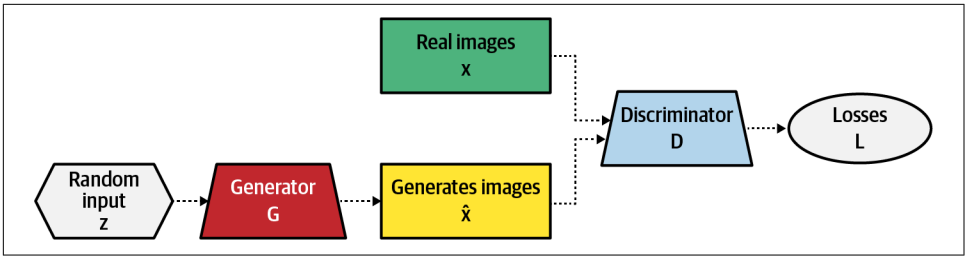

Figure 12-14. The standard GAN architecture, consisting of a generator and a discriminator.

For instance, imagine that a criminal organization wants to create realistic-looking money to deposit at the bank. In this scenario the criminals would be the generator, since they are trying to create realistic fake bills. The bankers would be the discriminator, examining the bills and trying to ensure the bank doesn't accept any counterfeit money.

A typical GAN training would begin with both the generator and the discriminator initialized with random weights. In our scenario that would mean that the counterfeiters have no clue how to generate realistic money: the generated outputs at the beginning simply look like random noise. Likewise, the bankers would begin not knowing the difference between a real and a generated bill, so they would be making terribly ill-informed random guesses as to what is real and what is fake.

The discriminator (the group of bankers) is presented with the first set of legitimate and generated bills, and has to classify them as real or fake. Because the discriminator starts off with random weights, initially it can't "see" easily that one bill is random noise and the other is a good bill. It's updated based on how well (or poorly) it performs, so over many iterations the discriminator will start becoming better at predicting which bills are real and which are generated. While the discriminator's weights are being trained, the generator's weights are frozen. However, the generator (the group of counterfeiters) is also improving during its turns, so it creates a moving target for the discriminator, progressively increasing the difficulty of the discrimination task. It's updated during each iteration based on how well (or not) its bills fooled the discriminator, and while it's being trained the discriminator's weights are frozen.

After many iterations, the generator is beginning to create something resembling real money because the discriminator was getting good at separating real and generated bills. This further pushes the discriminator to get even better at separating the now decent-looking generated bills from the real ones when it is its turn to train.

Eventually, after many iterations of training, the algorithm converges. This happens when the discriminator has lost its ability to separate generated bills from real bills and essentially is randomly guessing.

To see some of the finer details of the GAN training algorithm, we can refer to the pseudocode in Figure 12-15.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 12-15. A vanilla GAN training algorithm. Image from Goodfellow et al., 2014.

As we can see in the first line of the pseudocode, there is an outer `for` loop over the number of alternating discriminator/generator training iterations. We'll look at each of these updating phases in turn, but first we need to set up our generator and our discriminator.

Creating the networks

Before we do any training, we need to create our networks for the generator and discriminator. In a vanilla GAN, typically this is just a neural network composed of `Dense` layers.

The generator network takes a random vector of some latent dimension as input and passes it through some (possibly several) `Dense` layers to generate an image. For this example, we'll be using the MNIST handwritten digit dataset, so our inputs are 28x28 images. `LeakyReLU` activation functions usually work very well for GAN training because of their nonlinearity, not having vanishing gradient problems while also not losing information for any negative inputs or having the dreaded dying ReLU problem. Alpha is the amount of negative signal we want to leak through where a value of 0 would be the same as a ReLU activation and a value of 1 would be a linear activation. We can see this in the following TensorFlow code:

```
latent_dim = 512
vanilla_generator = tf.keras.Sequential(
    [
        tf.keras.Input(shape=(latent_dim,)),
        tf.keras.layers.Dense(units=256),
        tf.keras.layers.LeakyReLU(alpha=0.2),
        tf.keras.layers.Dense(units=512),
        tf.keras.layers.LeakyReLU(alpha=0.2),
        tf.keras.layers.Dense(units=1024),
        tf.keras.layers.LeakyReLU(alpha=0.2),
        tf.keras.layers.Dense(units=28 * 28 * 1, activation="tanh"),
        tf.keras.layers.Reshape(target_shape=(28, 28, 1))
    ],
    name="vanilla_generator"
)
```

The discriminator network in a vanilla GAN is also made up of `Dense` layers, but instead of generating images, it takes images as input, as shown here. The outputs are vectors of logits:

```
vanilla_discriminator = tf.keras.Sequential(
    [
        tf.keras.Input(shape=(28, 28, 1)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(units=1024),
        tf.keras.layers.LeakyReLU(alpha=0.2),
        tf.keras.layers.Dense(units=512),
        tf.keras.layers.LeakyReLU(alpha=0.2),
        tf.keras.layers.Dense(units=256),
        tf.keras.layers.LeakyReLU(alpha=0.2),
        tf.keras.layers.Dense(units=1),
    ],
    name="vanilla_discriminator"
)
```

Discriminator training

Within the outer loop is an inner loop for updating the discriminator. First, we sample a mini-batch of noise, typically random samples from a standard normal distribution. The random noise latent vector is passed through the generator to create generated (fake) images, as shown in Figure 12-16.

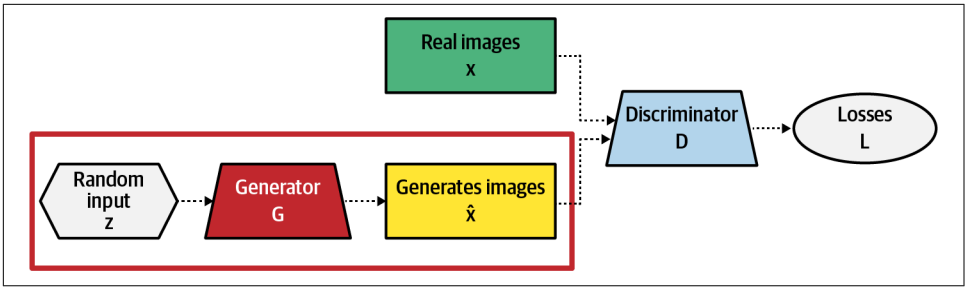

Figure 12-16. The generator creates its first batch of generated images by sampling from the latent space, and passes them to the discriminator.

In TensorFlow, we could for instance sample a batch of random normals using the following code:

```
# Sample random points in the latent space.
random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
```

We also sample a mini-batch of examples from our dataset—in our case, real images—as shown in Figure 12-17.

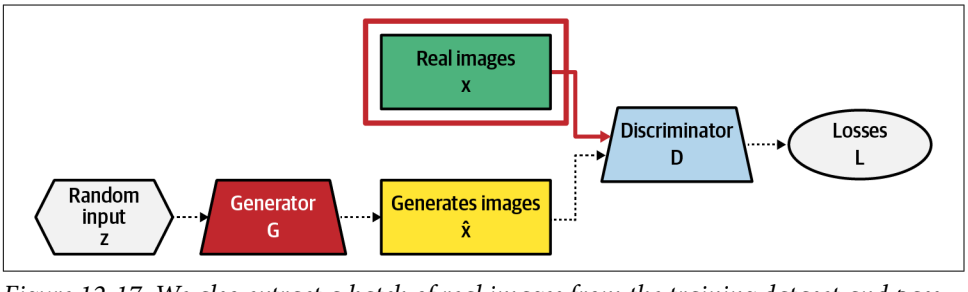

Figure 12-17. We also extract a batch of real images from the training dataset and pass this to the discriminator.

The generated images from the generator and the real images from the dataset are each passed through the discriminator, which makes its predictions. Loss terms for the real images and generated images are then calculated, as shown in Figure 12-18. Losses can take many different forms: binary cross-entropy (BCE), the average of the final activation map, second-order derivative terms, other penalties, etc. In the sample code, we'll be using BCE: the larger the real image loss, the more the discriminator thought that the real images were fake; the larger the generated image loss, the more the discriminator thought that the generated images were real.

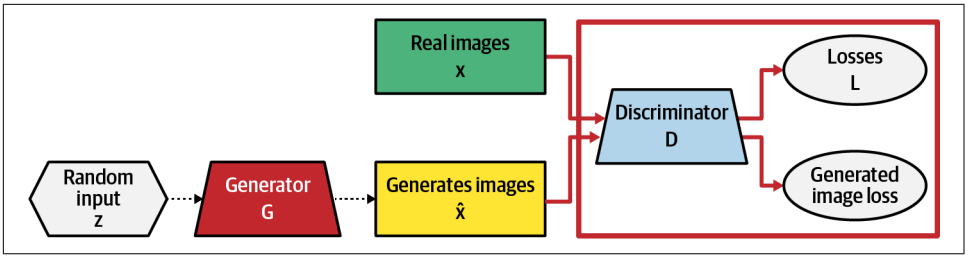

Figure 12-18. Real and generated samples pass through the discriminator to calculate the losses.

We do this in the following TensorFlow code (as usual, the complete code is in *12c_gan.ipynb* on GitHub). We can concatenate our generated and real images together and do the same with the corresponding labels so that we can do one pass through the discriminator:

```
# Generate images from noise.
generated_images = self.generator(inputs=random_latent_vectors)

# Combine generated images with real images.
combined_images = tf.concat(
    values=[generated_images, real_images], axis=0
)

# Create fake and real labels.
fake_labels = tf.zeros(shape=(batch_size, 1))
real_labels = tf.ones(shape=(batch_size, 1))

# Smooth real labels to help with training.
real_labels *= self.one_sided_label_smoothing

# Combine labels to be inline with combined images.
labels = tf.concat(
    values=[fake_labels, real_labels], axis=0
)

# Calculate discriminator loss.
self.loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
predictions = self.discriminator(inputs=combined_images)
discriminator_loss = self.loss_fn(y_true=labels, y_pred=predictions)
```

We first pass our random latent vectors through the generator to obtain a batch of generated images. This is concatenated with our batch of real images so we have both sets of images together in one tensor.

We then generate our labels. For the generated images we make a vector of 0s, and for the real images a vector of 1s. This is because with our BCE loss we are essentially just doing binary image classification (where the positive class is that of the real images), and therefore we are getting the probability that an image is real. Labeling the real

images with 1s and the fake images with 0s encourages the discriminator model to output probabilities as close to 1 as possible for real images and as close to 0 as possible for fake images.

It can be helpful sometimes to add one-sided label smoothing to our real labels, which involves multiplying them by a float constant in the range [0.0, 1.0]. This helps the discriminator avoid becoming overconfident in its predictions based on only a small set of features within the images, which the generator may then exploit (causing it to become good at beating the discriminator but not at image generation).

Since this is a discriminator training step, we use a combination of these losses to calculate the gradients with respect to the discriminator weights and then update the aforementioned weights as shown in Figure 12-19. Remember, during the discriminator training phase the generator's weights are frozen. This way each network gets its own chance to learn, independent of the other.

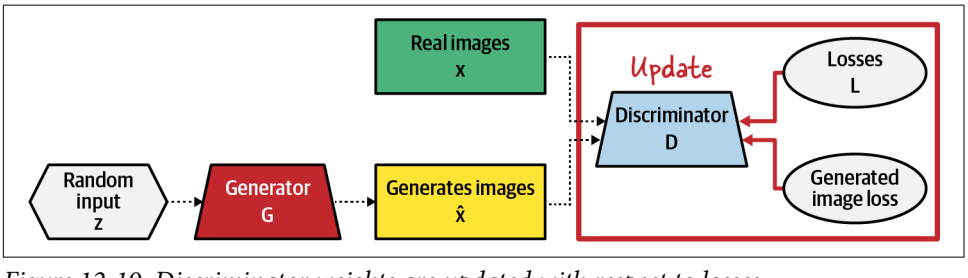

Figure 12-19. Discriminator weights are updated with respect to losses.

In the following code we can see this discriminator update being performed:

```
# Train ONLY the discriminator.
with tf.GradientTape() as tape:
    predictions = self.discriminator(inputs=combined_images)
    discriminator_loss = self.loss_fn(
        y_true=labels, y_pred=predictions
    )

grads = tape.gradient(
    target=discriminator_loss,
    sources=self.discriminator.trainable_weights
)

self.discriminator_optimizer.apply_gradients(
    grads_and_vars=zip(grads, self.discriminator.trainable_weights)
)
```

Generator training

After a few steps of applying gradient updates to the discriminator, it's time to update the generator (this time with the discriminator's weights frozen). We can do this in an

inner loop too. This is a simple process where we again take a mini-batch of random samples from our standard normal distribution and pass them through the generator to obtain fake images.

In TensorFlow the code would look like this:

```
# Sample random points in the latent space.
random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))

# Create labels as if they're real images.
labels = tf.ones(shape=(batch_size, 1))
```

Notice that even though these will be generated images, we will label them as real. Remember, we want to trick the discriminator into thinking our generated images are real. We can provide the generator with the gradients produced by the discriminator on images it was not fooled by. The generator can use these gradients to update its weights so that the next time it can do a better job of fooling the discriminator.

The random inputs pass through the generator as before to create generated images; however, there are no real images needed for generator training, as you can see in Figure 12-20.

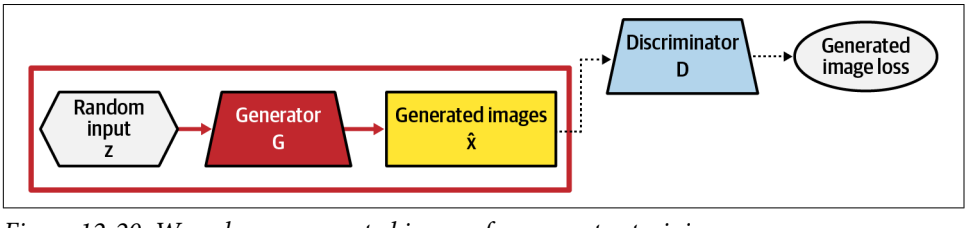

Figure 12-20. We only use generated images for generator training.

The generated images are then passed through the discriminator as before and a generator loss is calculated, as seen in Figure 12-21.

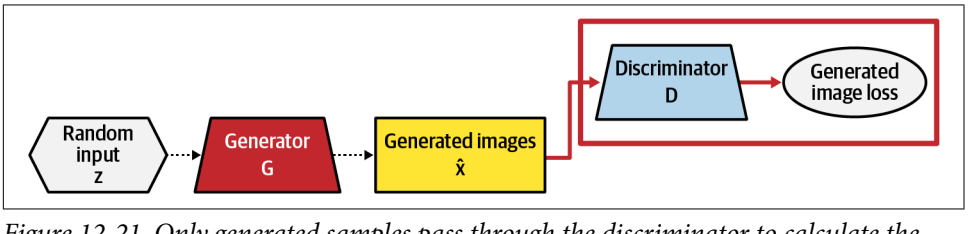

Figure 12-21. Only generated samples pass through the discriminator to calculate the loss.

Notice that no real images from the dataset are used in this phase. The loss is used to update only the generator's weights, as shown in Figure 12-22; even though the discriminator was used in the generator's forward pass, its weights are frozen during this phase so it does not learn anything from this process.

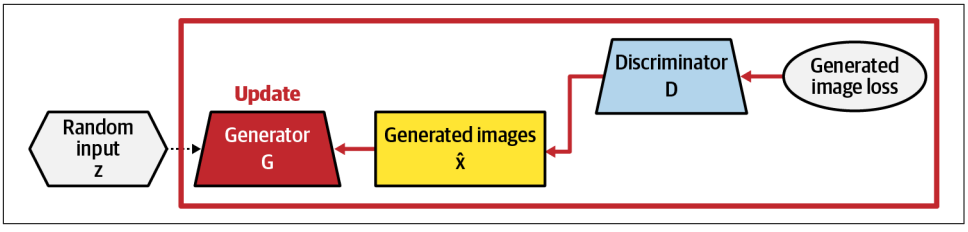

Figure 12-22. The generator's weights are updated with respect to the loss.

Here's the code that performs the generator update:

```
# Train ONLY the generator.
with tf.GradientTape() as tape:
    predictions = self.discriminator(
        inputs=self.generator(inputs=random_latent_vectors)
    )
    generator_loss = self.loss_fn(y_true=labels, y_pred=predictions)

grads = tape.gradient(
    target=generator_loss, sources=self.generator.trainable_weights
)

self.generator_optimizer.apply_gradients(
    grads_and_vars=zip(grads, self.generator.trainable_weights)
)
```

Once this is complete we go back to the discriminator's inner loop, and so on and so forth until convergence.

We can call the following code from our vanilla GAN generator TensorFlow model to see some of the generated images:

```
gan.generator(
    inputs=tf.random.normal(shape=(num_images, latent_dim))
)
```

Of course, if the model hasn't been trained, the images coming out will be random noise (produced by random noise coming in and going through multiple layers of random weights). Figure 12-23 shows what our GAN has learned once training on the MNIST handwritten digit dataset is complete.

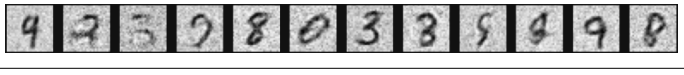

Figure 12-23. MNIST digits generated by a vanilla GAN generator.

Distribution changes

GANs definitely have an interesting training procedure compared to more traditional machine learning models. They may even seem a bit mysterious in terms of how they

work mathematically to learn the things they do. One way of trying to understand them a little more deeply is to observe the dynamics of the learned distributions of the generator and discriminator as they each try to outdo the other. Figure 12-24 shows how the generator's and discriminator's learned distributions change throughout the GAN training.

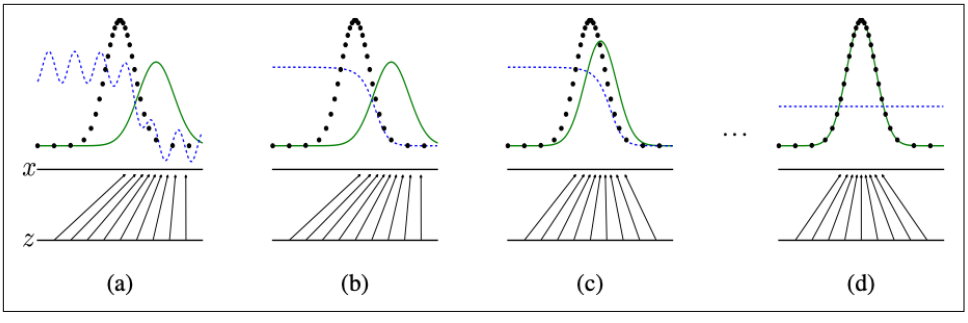

Figure 12-24. Learned distribution evolution during GAN training. The dashed line is the discriminator distribution, the solid line is the generator distribution, and the dotted line is the data generating (true data) distribution. The lower horizontal line is the domain that z is sampled from for the latent space and the upper horizontal line is a portion of the domain of x for the image space. The arrows show how z maps to x by x = G(z). The generator distribution shrinks in regions of low density and expands in regions of high density of the z to x mapping. Image from Goodfellow et al., 2014.

In Figure 12-24(a) we can see that the generator is not amazing, but is doing a decent job at generating some of the data distribution. The solid-lined generator distribution overlaps somewhat with the dotted-lined true data distribution (what we're trying to learn to generate from). Likewise, the discriminator does a fairly decent job of classifying real versus fake samples: it shows a strong signal (dashed line) when overlapping the dotted-lined data distribution and to the left of the peak of the generator distribution. The discriminative signal greatly shrinks in the region where the solid-lined generator distribution peaks.

The discriminator is then trained on another batch of real and generated images from the fixed generator within the inner discriminator training loop, over some number of iterations. In Figure 12-24(b) we can see that the dashed-lined discriminator distribution smooths out, and on the right it follows along the dotted-lined data distribution under the solid-lined generator distribution. On the left the distribution is much higher, and closer to the data distribution. Notice that the solid-lined generator distribution does not change at this step since we haven't updated the generator yet.

Figure 12-24(c) shows the results after the generator has been trained for some number of iterations. The performance of the newly updated discriminator helps guide it to shift its network weights and thus fill in some of the gaps it was missing from the

data distribution, so it gets better at generating fake samples. We can see this as the solid-lined generator distribution is now much closer to the dotted curve of the data distribution.

Figure 12-24(d) shows the results after many more iterations alternating between training the discriminator and the generator. If both networks have enough capacity, the generator will have converged: its distribution will closely match with the data distribution, and it will be generating great-looking samples. The discriminator has also converged because it is no longer able to tell what is a real sample from the data distribution and what is a generated sample from the generator distribution. Thus, the discriminator's distribution flatlines to random guesses with 50/50 odds, and training of the GAN system is complete.

GAN Improvements

This looks great on paper, and GANs are extremely powerful for image generation—however, in practice they can be extremely hard to train due to hypersensitivity to hyperparameters, unstable training, and many failure modes.

If either network gets too good at its job too fast, then the other network will be unable to keep up, and the generated images will never get to look very realistic. Another problem is mode collapse, where the generator loses most of its diversity in creating images and only generates the same few outputs. This happens when it stumbles upon a generated output that for whatever reason is very good at stumping the discriminator. This can go on for quite some time during training until, by chance, the discriminator finally is able to detect that those few images are generated and not real.

In a GAN, the first network (the generator) has an expanding layer size from its input layer to its output layer. The second network (the discriminator) has a shrinking layer size from its input layer to its output layer. Our vanilla GAN architecture used dense layers, like an autoencoder. However, convolutional layers tend to perform better on tasks involving images.

A *deep convolutional GAN* (DCGAN) is more or less just a vanilla GAN with the dense layers swapped out for convolutional layers. In the following TensorFlow code, we define a DCGAN generator:

```
def create_dcgan_generator(latent_dim):
    dcgan_generator = [
        tf.keras.Input(shape=(latent_dim,)),
        tf.keras.layers.Dense(units=7 * 7 * 256),
        tf.keras.layers.LeakyReLU(alpha=0.2),
        tf.keras.layers.Reshape(target_shape=(7, 7, 256)),
    ] + create_generator_block(
        filters=128, kernel_size=4, strides=2, padding="same", alpha=0.2
    ) + create_generator_block(
```

```
            filters=128, kernel_size=4, strides=2, padding="same", alpha=0.2
        ) + [
            tf.keras.layers.Conv2DTranspose(
                filters=1,
                kernel_size=3,
                strides=1,
                padding="same",
                activation="tanh"
            )
        ]

        return tf.keras.Sequential(
            layers=dcgan_generator, name="dcgan_generator"
        )
```

Our templated generator block then looks like the following:

```
    def create_generator_block(filters, kernel_size, strides, padding, alpha):
        return [
            tf.keras.layers.Conv2DTranspose(
                filters=filters,
                kernel_size=kernel_size,
                strides=strides,
                padding=padding
            ),
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.LeakyReLU(alpha=alpha)
        ]
```

Likewise, we can define a DCGAN discriminator like this:

```
    def create_dcgan_discriminator(input_shape):
        dcgan_discriminator = [
            tf.keras.Input(shape=input_shape),
            tf.keras.layers.Conv2D(
                filters=64, kernel_size=3, strides=1, padding="same"
            ),
            tf.keras.layers.LeakyReLU(alpha=0.2)
        ] + create_discriminator_block(
            filters=128, kernel_size=3, strides=2, padding="same", alpha=0.2
        ) + create_discriminator_block(
            filters=128, kernel_size=3, strides=2, padding="same", alpha=0.2
        ) + create_discriminator_block(
            filters=256, kernel_size=3, strides=2, padding="same", alpha=0.2
        ) + [
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(units=1)
        ]

        return tf.keras.Sequential(
            layers=dcgan_discriminator, name="dcgan_discriminator"
        )
```

And here's our templated discriminator block:

```
def create_discriminator_block(filters, kernel_size, strides, padding, alpha):
    return [
        tf.keras.layers.Conv2D(
            filters=filters,
            kernel_size=kernel_size,
            strides=strides,
            padding=padding
        ),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.LeakyReLU(alpha=alpha)
    ]
```

As you can see, the generator is upsampling the image using `Conv2DTranspose` layers whereas the discriminator is downsampling the image using `Conv2D` layers.

We can then call the trained DCGAN generator to see what it has learned:

```
dcgan.generator(
    inputs=tf.random.normal(shape=(num_images, latent_dim))
)
```

The results are shown in Figure 12-25.

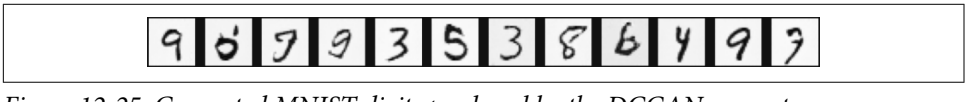

Figure 12-25. Generated MNIST digits produced by the DCGAN generator.

There are many other improvements that can be made to vanilla GANs, such as using different loss terms, gradients, and penalties. Since this is an active area of research, those are beyond the scope of this book.

Conditional GANs

The basic GAN that we discussed previously is trained in a completely unsupervised way on images that we want to learn how to generate. Latent representations, such as a random noise vector, are then used to explore and sample the learned image space. A simple enhancement is to add an external flag to our inputs with a label. For instance, consider the MNIST dataset, which consists of handwritten digits from 0 to 9. Normally, the GAN just learns the distribution of digits, and when the generator is given random noise vectors it generates different digits, as shown in Figure 12-26. However, which digits are generated cannot be controlled.

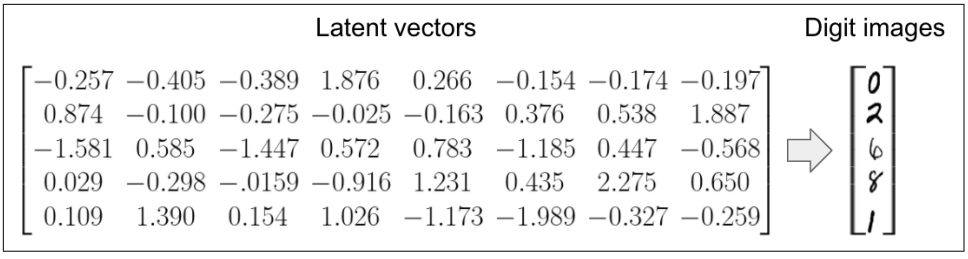

Figure 12-26. Unconditional GAN output.

During training, as with MNIST, we may know the actual label or class designation for each image. That extra information can be included as a feature in our GAN training that can then be used at inference time. With *conditional GANs* (cGANs), image generation can be conditional on the label, so we are able to home in on the specific digit of interest's distribution. Then, at inference time we can create an image of a specific digit by passing in the desired label instead of receiving a random digit, as seen in Figure 12-27.

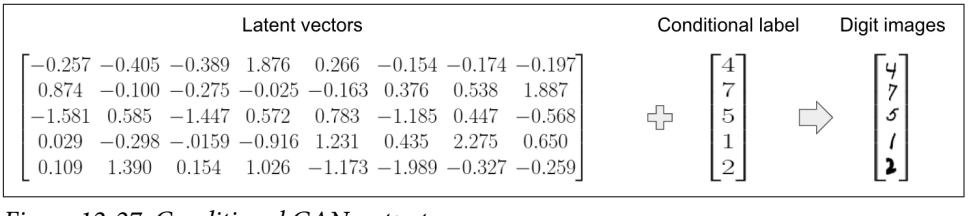

Figure 12-27. Conditional GAN output.

The cGAN generator. We need to make some changes to our vanilla GAN generator code from earlier so that we can incorporate the label. Essentially, we'll be concatenating our latent vector with a vector representation of our labels, as you can see in the following code:

```
# Create the generator.
def create_label_vectors(labels, num_classes, embedding_dim, dense_units):
    embedded_labels = tf.keras.layers.Embedding(
        input_dim=num_classes, output_dim=embedding_dim
    )(inputs=labels)
    label_vectors = tf.keras.layers.Dense(
        units=dense_units
    )(inputs=embedded_labels)

    return label_vectors
```

Here, we use an `Embedding` layer to transform our integer labels into a dense representation. We'll later be combining the embedding of the label with our typical random noise vector to create a new latent vector that is a mixture of the input latent space and the class labels. We then use a `Dense` layer to further mix the components.

Next, we utilize our standard vanilla GAN generator from before. However, this time we are using the Keras Functional API, as we did for our variational autoencoder earlier, because we now have multiple inputs to our generator model (the latent vector and labels):

```
def standard_vanilla_generator(inputs, output_shape):
    x = tf.keras.layers.Dense(units=64)(inputs=inputs)
    x = tf.keras.layers.LeakyReLU(alpha=0.2)(inputs=x)
    x = tf.keras.layers.Dense(units=128)(inputs=x)
    x = tf.keras.layers.LeakyReLU(alpha=0.2)(inputs=x)
    x = tf.keras.layers.Dense(units=256)(inputs=x)
    x = tf.keras.layers.LeakyReLU(alpha=0.2)(inputs=x)
    x = tf.keras.layers.Dense(
        units=output_shape[0] * output_shape[1] * output_shape[2],
        activation="tanh"
    )(inputs=x)

    outputs = tf.keras.layers.Reshape(target_shape=output_shape)(inputs=x)

    return outputs
```

Now that we have a way to embed our integer labels, we can combine this with our original standard generator to create a vanilla cGAN generator:

```
def create_vanilla_generator(latent_dim, num_classes, output_shape):
    latent_vector = tf.keras.Input(shape=(latent_dim,))

    labels = tf.keras.Input(shape=())
    label_vectors = create_label_vectors(
        labels, num_classes, embedding_dim=50, dense_units=50
    )

    concatenated_inputs = tf.keras.layers.Concatenate(
        axis=-1
    )(inputs=[latent_vector, label_vectors])

    outputs = standard_vanilla_generator(
        inputs=concatenated_inputs, output_shape=output_shape
    )

    return tf.keras.Model(
        inputs=[latent_vector, labels],
        outputs=outputs,
        name="vanilla_generator"
    )
```

Notice we now have two sets of inputs using the Keras `Input` layer. Remember, this is the main reason we are using the Keras Functional API instead of the Sequential API: it allows us to have an arbitrary number of inputs and outputs, with any type of network connectivity in between. Our first input is the standard latent vector, which is our generated random normal noise. Our second input is the integer labels that we

will condition on later, so we can target certain classes of generated images via labels provided at inference time. Since, in this example, we are using MNIST handwritten digits, the labels will be integers between 0 and 9.

Once we've created our dense label vectors, we combine them with our latent vectors using a Keras `Concatenate` layer. Now we have a single tensor of vectors, each of shape `latent_dim + dense_units`. This is our new "latent vector," which gets sent into the standard vanilla GAN generator. This isn't the original latent vector of our original vector space that we sampled random points from, but is now a new higher-dimensional vector space due to the concatenation of the encoded label vector.

This new latent vector helps us target specific classes for generation. The class label is now embedded in the latent vector and therefore will be mapped to a different point in image space than with the original latent vector. Furthermore, given the same latent vector, each label pairing will map to a different point in image space because it is using a different learned mapping due to the different concatenated latent–label vectors. Therefore, when the GAN is trained it learns to map each latent point to a point in image space corresponding to an image belonging to one of the 10 classes.

As we can see at the end of the function, we simply instantiate a Keras `Model` with our two input tensors and output tensor. Looking at the conditional GAN generator's architecture diagram, shown in Figure 12-28, should make clear how we are using the two sets of inputs, the latent vector and the label, to generate images.

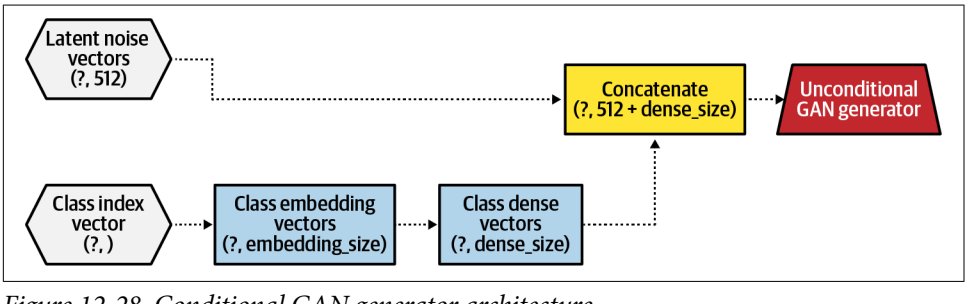

Figure 12-28. Conditional GAN generator architecture.

Now that we've seen the generator, let's take a look at the code for the conditional GAN discriminator.

The cGAN discriminator. For the generator, we created label vectors that we concatenated with our latent vectors. For the discriminator, which has image inputs, we instead convert the labels into images and concatenate the images created from the labels with the input images. This allows the label information to be embedded into our images to help the discriminator differentiate between real and generated images. The label will help warp the latent space to image space mapping such that each input will be associated with its label's bubble within image space. For example, for MNIST,

if the model is given the digit 2, the discriminator will generate something within the bubble of 2s in the learned image space.

To accomplish the conditional mapping for the discriminator, we once again pass our integer labels through an `Embedding` and a `Dense` layer. However, each example in the batch is now just a vector `num_pixels` long. Thus, we use a `Reshape` layer to transform the vector into an image with just one channel. Think of it as a grayscale image representation of our label. In the following code, we can see the labels being embedded into images:

```
def create_label_images(labels, num_classes, embedding_dim, image_shape):
    embedded_labels = tf.keras.layers.Embedding(
        input_dim=num_classes, output_dim=embedding_dim)(inputs=labels)
    num_pixels = image_shape[0] * image_shape[1]
    dense_labels = tf.keras.layers.Dense(
        units=num_pixels)(inputs=embedded_labels)
    label_image = tf.keras.layers.Reshape(
        target_shape=(image_shape[0], image_shape[1], 1))(inputs=dense_labels)

    return label_image
```

As we did for the generator, we will reuse our standard vanilla GAN discriminator from the previous section that maps images into logit vectors that will be used for loss calculations with binary cross-entropy. Here's the code for the standard discriminator, using the Keras Functional API:

```
def standard_vanilla_discriminator(inputs):
    """Returns output of standard vanilla discriminator layers.

    Args:
        inputs: tensor, rank 4 tensor of shape (batch_size, y, x, channels).

    Returns:
        outputs: tensor, rank 4 tensor of shape
            (batch_size, height, width, depth).
    """
    x = tf.keras.layers.Flatten()(inputs=inputs)
    x = tf.keras.layers.Dense(units=256)(inputs=x)
    x = tf.keras.layers.LeakyReLU(alpha=0.2)(inputs=x)
    x = tf.keras.layers.Dense(units=128)(inputs=x)
    x = tf.keras.layers.LeakyReLU(alpha=0.2)(inputs=x)
    x = tf.keras.layers.Dense(units=64)(inputs=x)
    x = tf.keras.layers.LeakyReLU(alpha=0.2)(inputs=x)

    outputs = tf.keras.layers.Dense(units=1)(inputs=x)

    return outputs
```

Now we'll create our conditional GAN discriminator. It has two inputs: the first is the standard image input, and the second is the class labels that the images will be conditioned on. Just like for the generator, we convert our labels into a usable

representation to use with our images—namely, into grayscale images—and we use a `Concatenate` layer to combine the input images with the label images. We send those combined images into our standard vanilla GAN discriminator and then instantiate a Keras `Model` using our two inputs, the outputs, and a name for the discriminator `Model`:

```
def create_vanilla_discriminator(image_shape, num_classes):
    """Creates vanilla conditional GAN discriminator model.

    Args:
        image_shape: tuple, the shape of the image without batch dimension.
        num_classes: int, the number of image classes.

    Returns:
        Keras Functional Model.
    """
    images = tf.keras.Input(shape=image_shape)

    labels = tf.keras.Input(shape=())
    label_image = create_label_images(
        labels, num_classes, embedding_dim=50, image_shape=image_shape
    )

    concatenated_inputs = tf.keras.layers.Concatenate(
        axis=-1
    )(inputs=[images, label_image])

    outputs = standard_vanilla_discriminator(inputs=concatenated_inputs)

    return tf.keras.Model(
        inputs=[images, labels],
        outputs=outputs,
        name="vanilla_discriminator"
    )
```

Figure 12-29 shows the full conditional GAN discriminator architecture.

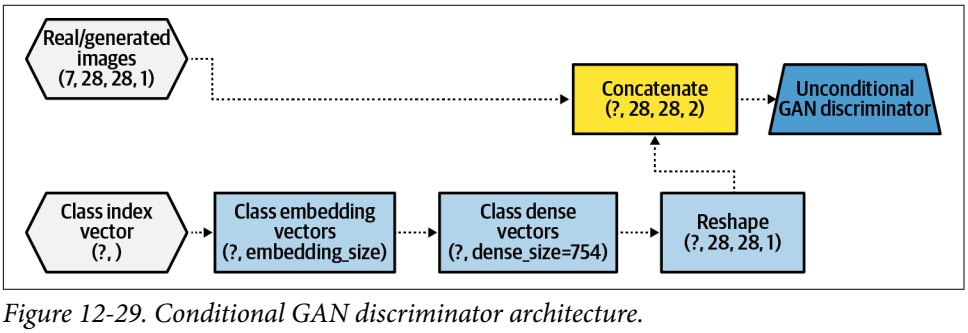

Figure 12-29. Conditional GAN discriminator architecture.

The rest of the conditional GAN training process is virtually the same as the non-conditional GAN training process, except for the fact we now pass in the labels from our dataset to use both for the generator and the discriminator.

Using a `latent_dim` of 512 and training for 30 epochs, we can use our generator to produce images like the ones in Figure 12-30. Note that for each row the label used at inference was the same, hence why the first row is all zeros, the second row is all ones, and so on. This is great! Not only can we generate handwritten digits, but we can specifically generate the digits we want.

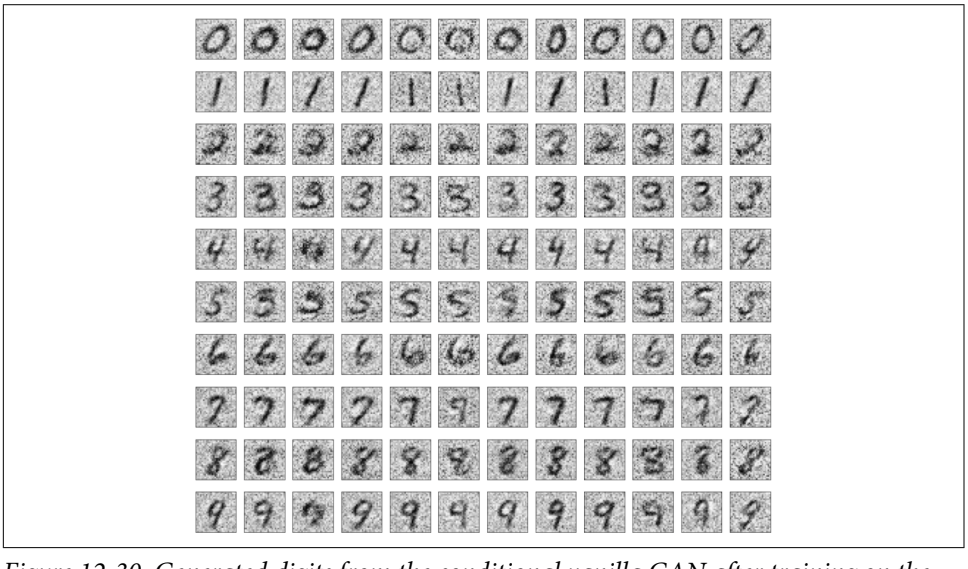

Figure 12-30. Generated digits from the conditional vanilla GAN after training on the MNIST dataset.

We can get even cleaner results if, instead of using our standard vanilla GAN generator and discriminator, we use the DCGAN generator and discriminator shown earlier. Figure 12-31 shows some of the images generated after training the conditional DCGAN model with a `latent_dim` of 512 for 50 epochs.

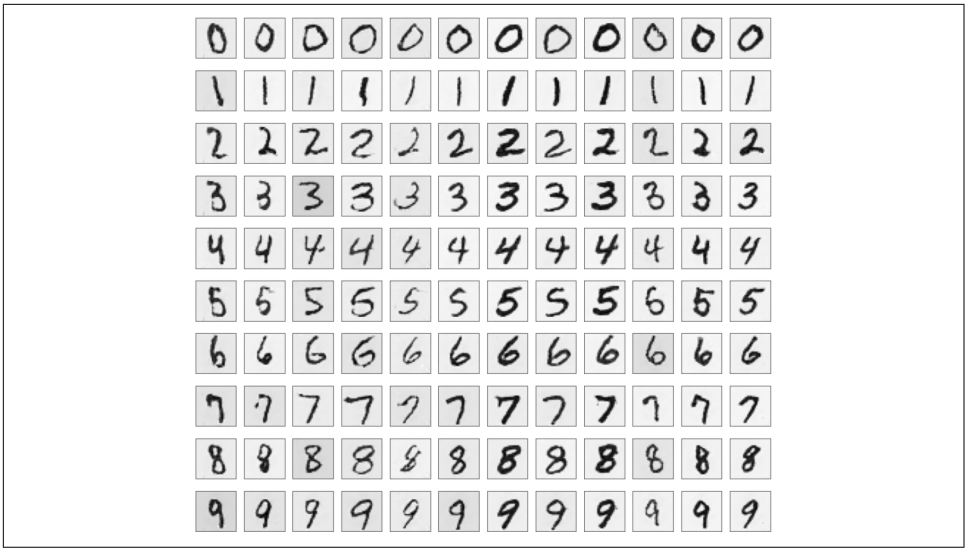

Figure 12-31. Generated digits from the conditional DCGAN after training on the MNIST dataset.

GANs are powerful tools for generating data. We focused on image generation here, but other types of data (such as tabular, time series, and audio data) can also be generated via GANs. However, GANs are a bit finicky and often require the use of the tricks we've covered here, and many more, to improve their quality and stability. Now that you've added GANs as another tool in your toolbox, let's look at some advanced applications that use them.

Image-to-Image Translation

Image generation is one of the simpler applications that GANs are great at. We can also combine and manipulate the essential components of GANs to put them to other uses, many of which are state of the art.

Image-to-image translation is when an image is translated from one (source) domain into another (target) domain. For instance, in Figure 12-32, an image of a horse is translated so that it looks like the horses are zebras. Of course, since finding paired images (e.g., the same scene in winter and summer) can be quite difficult, we can instead create a model architecture that can perform the image-to-image translation using unpaired images. This might not be as performant as a model working with paired images, but it can get very close. In this section we will explore how to perform image translation first if we have unpaired images, the more common situation, and then if we have paired images.

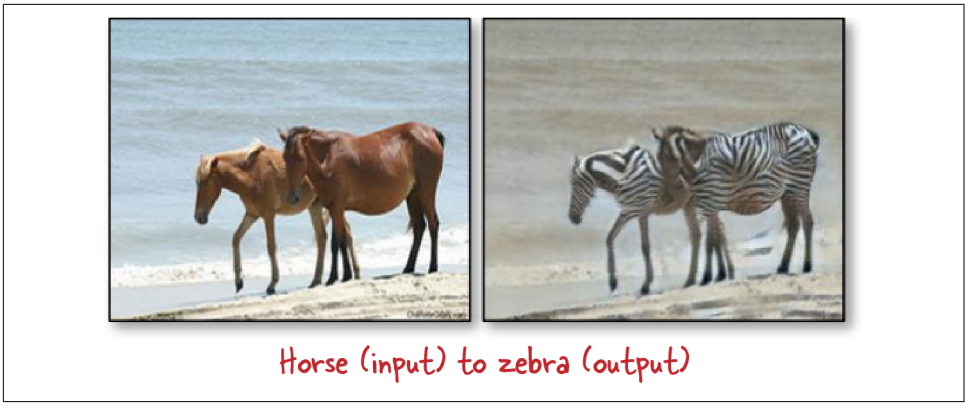

Figure 12-32. Results of using CycleGAN to translate an image of horses into an image of zebras. Image from Zhu et al., 2020.

The CycleGAN architecture used to perform the translation in Figure 12-32 takes GANs one step further and has two generators and two discriminators that cycle back and forth, as illustrated in Figure 12-33. Continuing with the previous example, let's say that horse images belong to image domain X, and zebra images belong to image domain Y. Remember, these are unpaired images; therefore, there isn't a matching zebra image for each horse image, and vice versa.

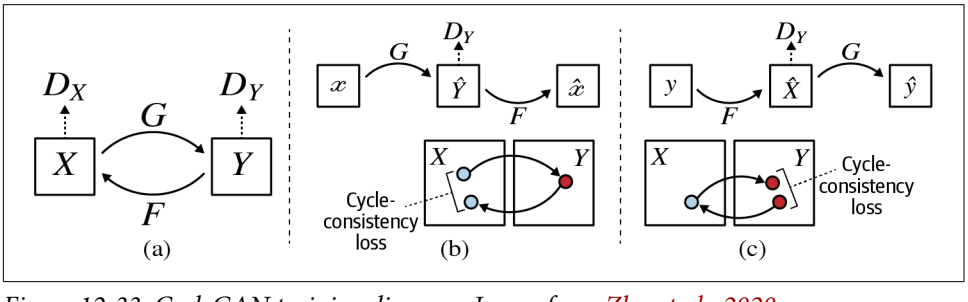

Figure 12-33. CycleGAN training diagram. Image from Zhu et al., 2020.

In Figure 12-33(a), generator G maps image domain X (horses) to image domain Y (zebras) while another generator, F, maps the reverse, Y (zebras) to X (horses). This means that generator G learns weights that map an image of horses, in this example, to an image of zebras, and vice versa for generator F. Discriminator D_X leads generator F to have a great mapping from Y (zebras) to X (horses), while discriminator D_Y leads generator G to have a great mapping from X (horses) to Y (zebras). We then perform cycles to add a little more regularization to the learned mappings, as described in the next panel.

In Figure 12-33(b), the forward cycle consistency loss—X (horses) to Y (zebras) to X (horses)—is from the comparison of an X (horses) domain image mapped to Y

(zebras) using generator G and then mapped back to X (horses) using generator F with the original X (horses) image.

Likewise, in Figure 12-33(c), the backward cycle consistency loss—Y (zebras) to X (horses) to Y (zebras)—is from the comparison of a Y (zebras) domain image mapped to X (horses) using generator F and then mapped back to Y (zebras) using generator G with the original Y (zebras) image.

Both the forward and backward cycle consistency loss compare the original image for a domain with the cycled image for that domain so that the network can learn to reduce the difference between them.

By having these multiple networks and ensuring cycle consistency we're able to get impressive results despite having unpaired images, such as when translating between horses and zebras or between summer images and winter images, as in Figure 12-34.

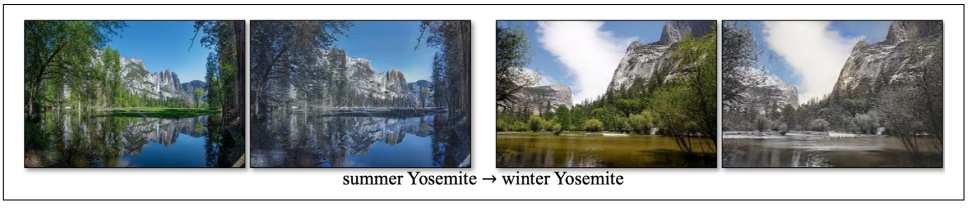

Figure 12-34. Results of using CycleGAN to translate summer images to winter images. Image from Zhu et al., 2020.

Now, if instead we did have paired examples, then we could take advantage of supervised learning to get even more impressive image-to-image translation results. For instance, as shown in Figure 12-35, an overhead map view of a city can be translated into the satellite view and vice versa.

Figure 12-35. Results of using Pix2Pix to translate a map view to satellite view and vice versa. Image from Isola et al., 2018.

The Pix2Pix architecture uses paired images to create the forward and reverse mappings between the two domains. We no longer need cycles to perform the image-to-image translation, but instead have a U-Net (previously seen in Chapter 4) with skip connections as our generator and a PatchGAN as our discriminator, which we discuss further next.

The U-Net generator takes a source image and tries to create the target image version, as shown in Figure 12-36. This generated image is compared to the actual paired target image via the L1 loss or MAE, which is then multiplied by lambda to weight the loss term. The generated image (source to target domain) and the input source image then go to the discriminator with labels of all 1s with a binary/sigmoid cross-entropy loss. The weighted sum of these losses is used for the gradient calculation for the generator to encourage the generator to improve its weights for domain translation in order to fool the discriminator. The same is done for the other generator/discriminator set with the source and target domains reversed for the reverse translation.

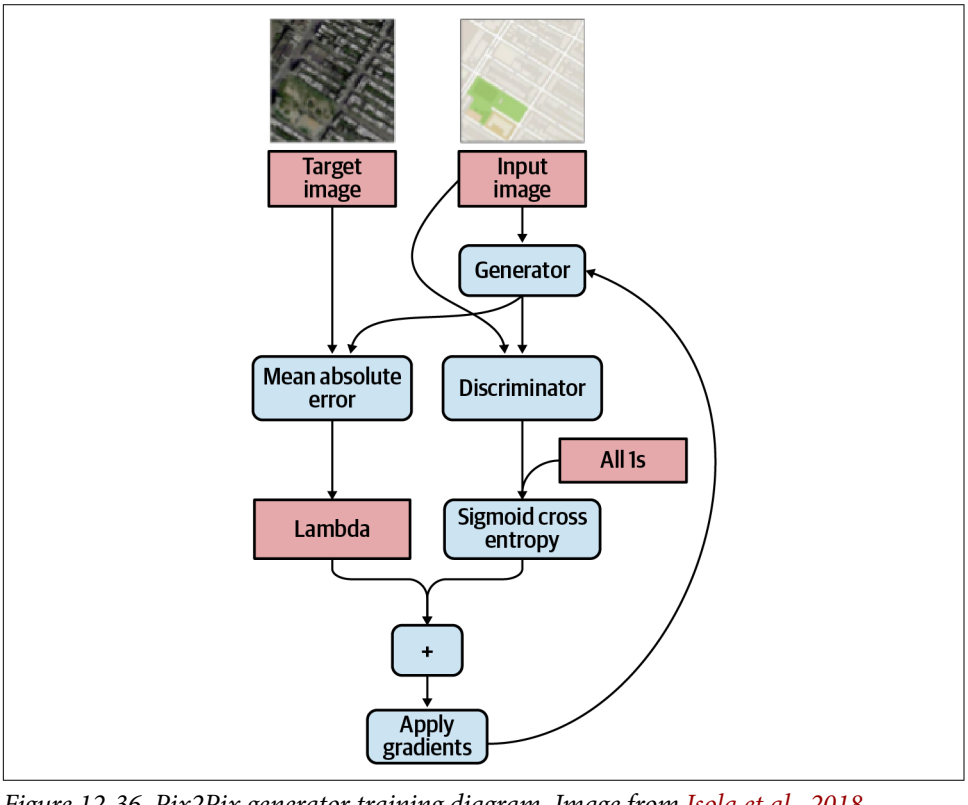

Figure 12-36. Pix2Pix generator training diagram. Image from Isola et al., 2018.

The PatchGAN discriminator classifies portions of the input image using smaller-resolution patches. This way each patch is classified as either real or fake using the local information in that patch rather than the entire image. The discriminator is passed two sets of input pairs, concatenated along channels, as shown in Figure 12-37.

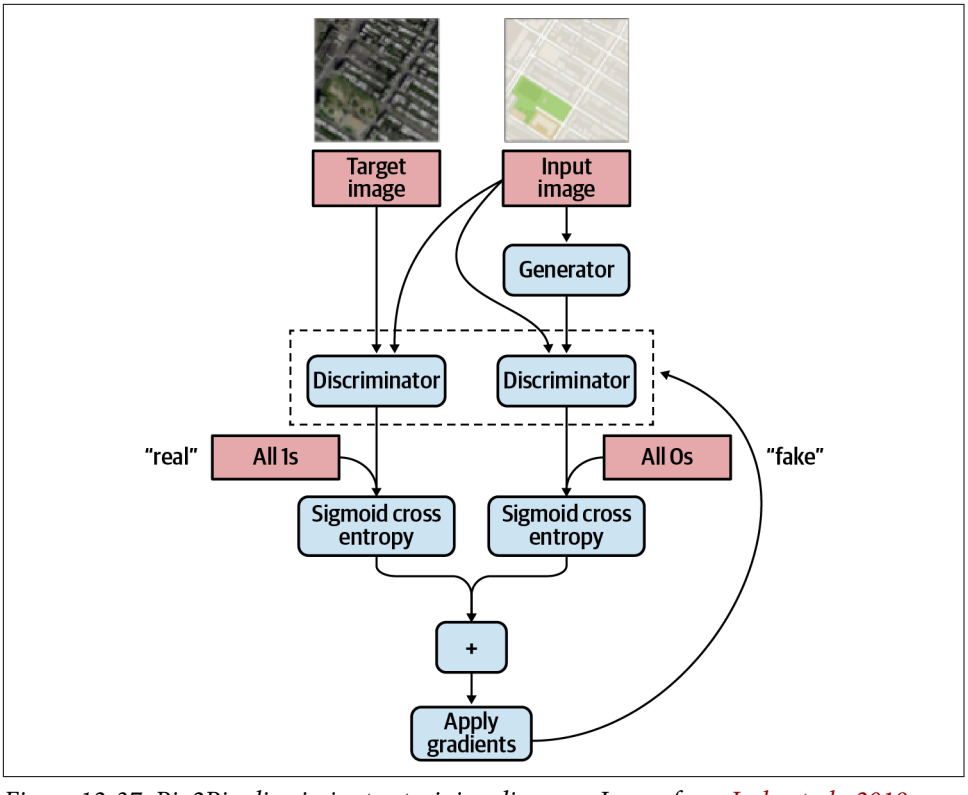

Figure 12-37. Pix2Pix discriminator training diagram. Image from Isola et al., 2018.

The first pair is made up of the input source image and the generated "source to target" image from the generator, which the discriminator should classify as fake by labeling them with all 0s. The second pair is made up of the input source image with the target image concatenated with it instead of the generated image. This is the real branch and hence this pair is labeled with all 1s. If we think back to simpler image generation this is following the same discriminator training pattern where the generated images are in the fake, all 0 labels branch and the real images we want to generate are in the real, all 1 labels branch. Therefore, the only change compared to image generation is that we are essentially conditioning the discriminator with the input image from the source domain, similar to what we did with conditional GANs.

This can lead to amazing use cases such as Figure 12-38, where hand-drawn objects can be filled in to look like real objects with photographic quality.

Figure 12-38. Results of using Pix2Pix on a drawing to transform it to an image of photographic quality. Image from Isola et al., 2018.

Just like how we can translate text and speech between languages, we can also translate images between different domains. We can use architectures like CycleGAN with (much more common) datasets of unpaired images, or more specialized architectures like Pix2Pix that can take full advantage of paired image datasets. This is still a very active area of research with many improvements being discovered.

Super-Resolution

For most of the use cases we've seen so far we've been both training and predicting with pristine images. However, we know in reality there can often be many defects in images, such as blur or the resolution being too low. Thankfully, we can modify some of the techniques we've already learned to fix some of those image issues.

Super-resolution is the process of taking a degraded or low-resolution image and upscaling it, transforming it into a corrected, high-resolution image. Super-resolution itself has been around for a long time as part of general image processing, yet it wasn't until more recently that deep learning models were able to produce state-of-the-art results with this technique.

The simplest and oldest methods of super-resolution use various forms of pixel interpolation, such as nearest neighbor or bicubic interpolation. Remember that we're starting with a low-resolution image that, when upscaled, has more pixels than the original image. These pixels need to be filled in through some means, and in a way that looks perceptually correct and doesn't just produce a smoothed, blurry larger image.

Figure 12-39 shows a sample of the results from a 2017 paper by Christian Ledig et al. The original high-resolution image is on the far right. It's from this image that a lower-resolution one is created for the training procedure. On the far left is the bicubic interpolation—it's a quite smooth and blurry recreation of the starting image from a smaller, lower-resolution version of the original. For most applications, this is not of high enough quality.

The second image from the left in Figure 12-39 is an image created by SRResNet, which is a residual convolutional block network. Here, a version of the original image that is low resolution, due to Gaussian noise and downsampling, is passed through 16 residual convolutional blocks. The output is a decent super-resolution image that's fairly close to the original—however, there are some errors and artifacts. The loss function used in SRResNet is the mean squared error between each pixel of the super-resolution output image and the original high-resolution image. Although the model is able to get pretty good results using MSE loss alone, it's not quite enough to encourage the model to make truly photorealistic and perceptually similar images.

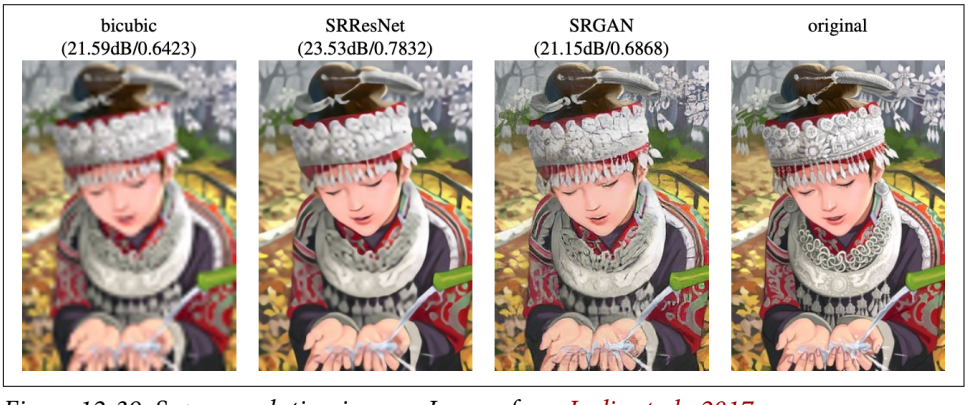

Figure 12-39. Super-resolution images. Images from Ledig et al., 2017.

The third image from the left in Figure 12-39 shows the best (in terms of perceptual quality) results, obtained using a model called SRGAN. The idea to utilize a GAN came from what the SRResNet image was lacking: high perceptual quality, which we can quickly judge by looking at the image. This is due to the MSE reconstruction loss, which aims to minimize the average error of the pixels but doesn't attempt to ensure the individual pixels combine to form a perceptually convincing image.

As we saw earlier, GANs generally have both a generator for creating images and a discriminator to discern whether the images being passed to it are real or generated. Rather than trying to slowly and painfully manually tune models to create convincing images, we can use GANs to do this tuning for us automatically. Figure 12-40 shows the generator and discriminator network architectures of SRGAN.

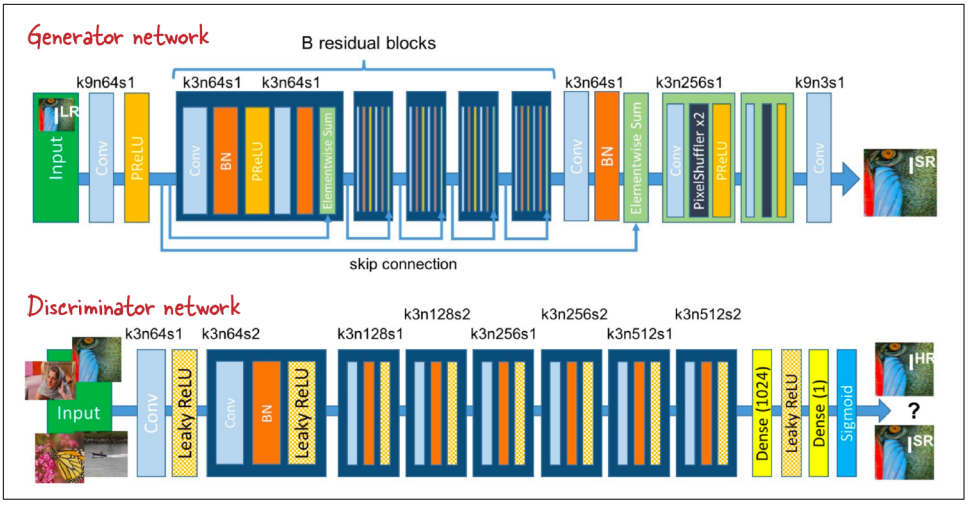

Figure 12-40. SRGAN generator and discriminator architectures. Image from Ledig et al., 2017.

In the SRGAN generator architecture, we begin by taking a high-resolution (HR) image and applying a Gaussian filter to it, then downsampling the image by some factor. This creates a low-resolution (LR) version of the image, which then gets convolved and passed through several residual blocks, like we saw in Chapter 4 with ResNet. The image is upsampled along the way, since we need to get back to its original size. The network also includes skip connections so that more detail from the earlier layers can condition the later layers and for better gradient backpropagation during the backward pass. After a few more convolutional layers, a super-resolution (SR) image is generated.

The discriminator takes images as input and determines whether they are SR or HR. The input is passed through several convolutional blocks, ending with a dense layer that flattens the intermediate images and finally another dense layer that produces the logits. Just like with a vanilla GAN, these logits are optimized on binary cross-entropy and so the result is a probability that the image is HR or SR, which is the adversarial loss term.

With SRGAN, there is another loss term that is weighted together with the adversarial loss to train the generator. This is the *contextual loss*, or how much of the original content remains within the image. Minimizing the contextual loss will ensure that the output image looks similar to the original image. Typically this is the pixel-wise MSE, but since that is a form of averaging it can create overly smooth textures that don't look realistic. Therefore, SRGAN instead uses what its developers call the *VGG loss*, using the activation feature maps for each of the layers of a pretrained 19-layer VGG network (after each activation, before the respective max-pooling layer). They

calculate the VGG loss as the sum of the Euclidean distance between the feature maps of the original HR images and the feature maps of the generated SR images, summing those values across all VGG layers, and then normalizing by the image height and width. The balance of these two loss terms will create images that not only look similar to the input images but also are correctly interpolated so that, perceptually, they look like real images.

Modifying Pictures (Inpainting)

There can be other reasons to fix an image, such as a tear in a photograph or a missing or obscured section, as shown in Figure 12-41(a). This hole-filling is called *inpainting*, where we want to literally paint in the pixels that should be in the empty spot. Typically, to fix such an issue an artist would spend hours or days restoring the image by hand, as shown in Figure 12-41(b), which is a laborious process and unscalable. Thankfully, deep learning with GANs can make scalably fixing images like this a reality—Figure 12-41(c) and (d) show some sample results.

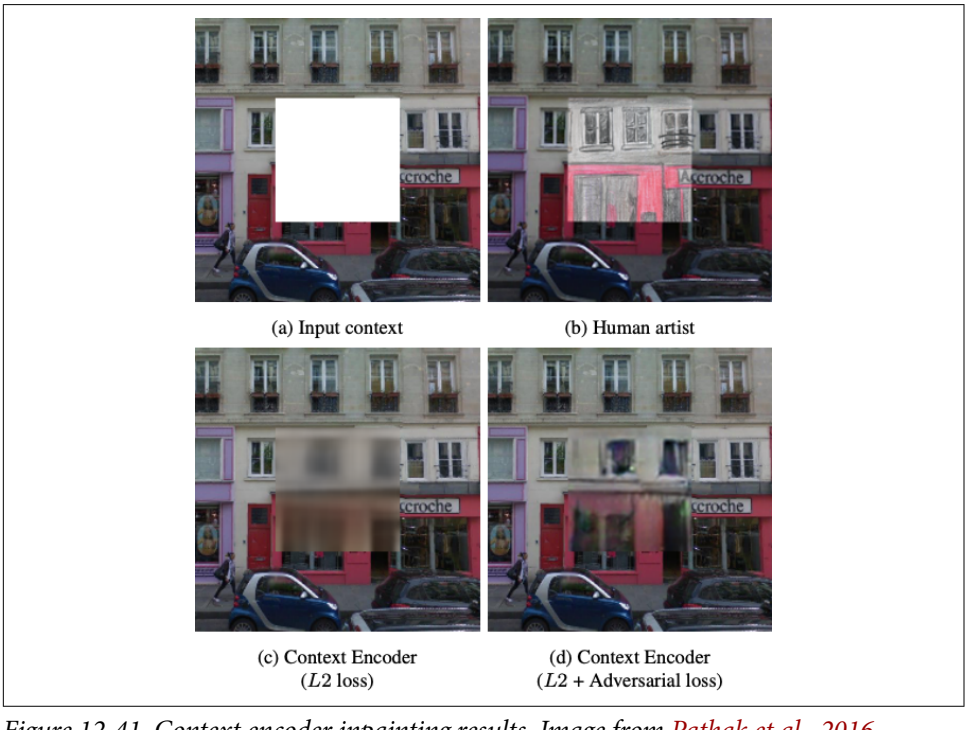

Figure 12-41. Context encoder inpainting results. Image from Pathak et al., 2016.

Here, unlike with SRGAN, instead of adding noise or a filter and downsampling a high-resolution image for training, we extract an area of pixels and set that region aside. We then pass the remaining image through a simple encoder/decoder network,

as shown in Figure 12-42. This forms the generator of the GAN, which we hope will generate content similar to what was in the pixel region we extracted.

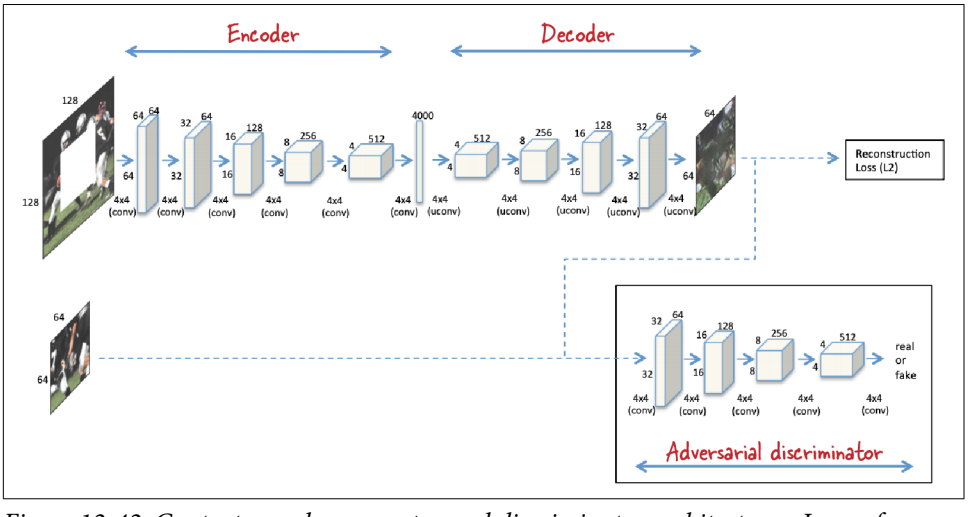

Figure 12-42. Context encoder generator and discriminator architectures. Image from Pathak et al., 2016.

The discriminator then compares the generated region of pixels with the region we extracted from the original image and tries to determine whether the image was generated or came from the real dataset.

Similar to SRGAN, and for the same reasons, the loss function has two terms: a reconstruction loss and an adversarial loss. The reconstruction loss isn't the typical L2 distance between the extracted image patch and the generated image patch, but rather the normalized masked L2 distance. The loss function applies a mask to the overall image so that we only aggregate the distances of the patch that we reconstructed, and not the border pixels around it. The final loss is the aggregated distance normalized by the number of pixels in the region. Alone, this usually does a decent job of creating a rough outline of the image patch; however, the reconstructed patch is usually lacking high-frequency detail and ends up being blurry due to the averaged pixel-wise error.

The adversarial loss for the generator comes from the discriminator, which helps the generated image patch appear to come from the manifold of natural images and therefore look realistic. These two loss functions can be combined in a weighted sum joint loss.

The extracted image patches don't just have to come from the central region, like in Figure 12-43(a)—in fact, that approach can be detrimental to the training as a result of poor generalization of the learned low-level image features to images without

patches extracted. Instead, taking random blocks, as in Figure 12-43(b), or random regions of pixels, as shown in Figure 12-43(c), produces more general features and greatly outperforms the approach of using a central region mask.

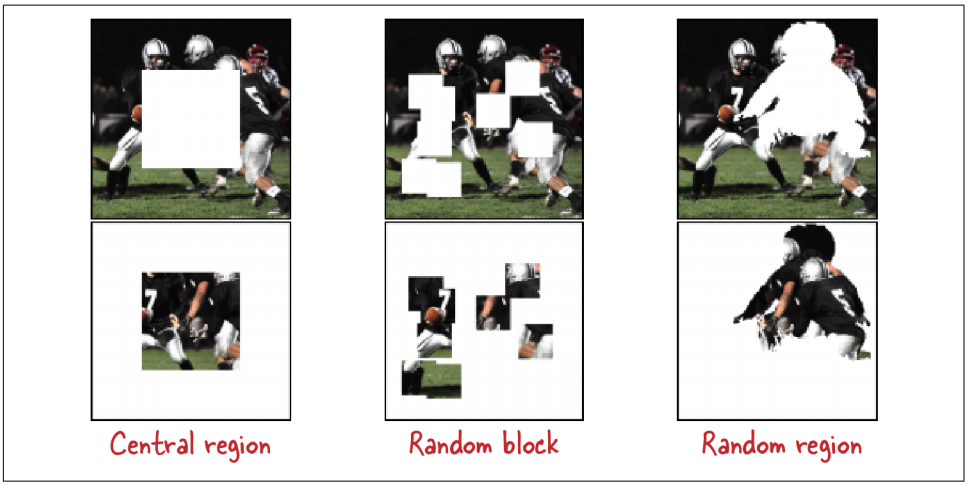

Figure 12-43. Different extracted patch region masks. Image from Pathak et al., 2016.

Anomaly Detection

Anomaly detection is another application that can benefit from the use of GANs—images can be passed to a modified GAN model and flagged as anomalous or not. This can be useful for tasks such as counterfeit currency detection, or looking for tumors in medical scans.

Typically there is a lot more unlabeled data available for deep learning use cases than labeled, and often the process of labeling is extremely laborious and may require deep subject matter expertise. This can make a supervised approach infeasible, which means we need an unsupervised approach.

To perform anomaly detection, we need to learn what "normal" looks like. If we know what normal is, then when an image doesn't fit within that distribution, it may contain anomalies. Therefore, when training anomaly detection models, it is important to train the model only on normal data. Otherwise, if the normal data is contaminated with anomalies, then the model will learn that those anomalies are normal. At inference time, this would lead to actual anomalous images not being correctly flagged, thus generating many more false negatives than may be acceptable.

The standard method of anomaly detection is to first learn how to reconstruct normal images, then learn the distribution of reconstruction errors of normal images, and finally learn a distance threshold where anything above that threshold is flagged as anomalous. There are many different types of models we can use to minimize the

reconstruction error between the input image and its reconstruction. For example, using an autoencoder, we can pass normal images through the encoder (which compresses an image down to a more compact representation), possibly through a layer or two in the bottleneck, then through the decoder (which expands the image back to its original representation), and finally generate an image that should be a reconstruction of the original image. The reconstruction is never perfect; there is always some error. Given a large collection of normal images, the reconstruction error will form a distribution of "normal errors." Now, if the network is given an anomalous image—one that it has not seen anything like during training—it will not be able to compress and reconstruct it correctly. The reconstruction error will be way out of the normal error distribution. The image can thus be flagged as an anomalous image.

Taking the anomaly detection use case one step further, we could instead perform *anomaly localization*, where we are flagging individual pixels as anomalous. This is like an unsupervised segmentation task rather than an unsupervised classification task. Each pixel has an error, and this can form a distribution of errors. In an anomalous image, many pixels will exhibit a large error. Reconstructed pixels above a certain distance threshold from their original versions can be flagged as anomalous, as shown in Figure 12-44.

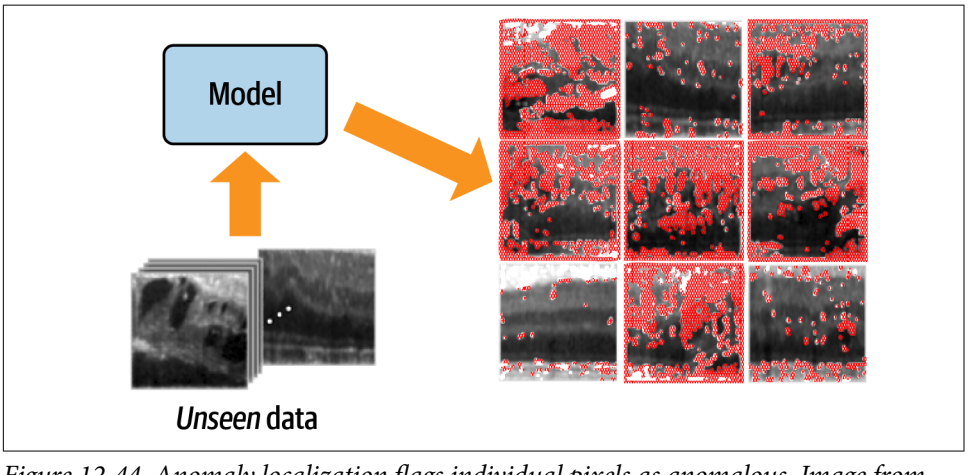

Figure 12-44. Anomaly localization flags individual pixels as anomalous. Image from Schlegl et al., 2019.

However, for many use cases and datasets this isn't the end of the story. With just the autoencoder and reconstruction loss, the model may learn how to map any image to itself instead of learning what normal looks like. Essentially, reconstruction dominates the combined loss equation and therefore the model learns the best way to compress *any* image, rather than learning the "normal" image manifold. For anomaly detection this is very bad because the reconstruction loss for both normal and

anomalous images will be similar. Therefore, as with super-resolution and inpainting, using a GAN can help.

This is an active area of research, so there are many competing variations of model architectures, loss functions, training procedures, etc., but they all have several components in common, as seen in Figure 12-45. Typically they consist of a generator and discriminator, sometimes with additional encoder and decoder networks depending on the use case.

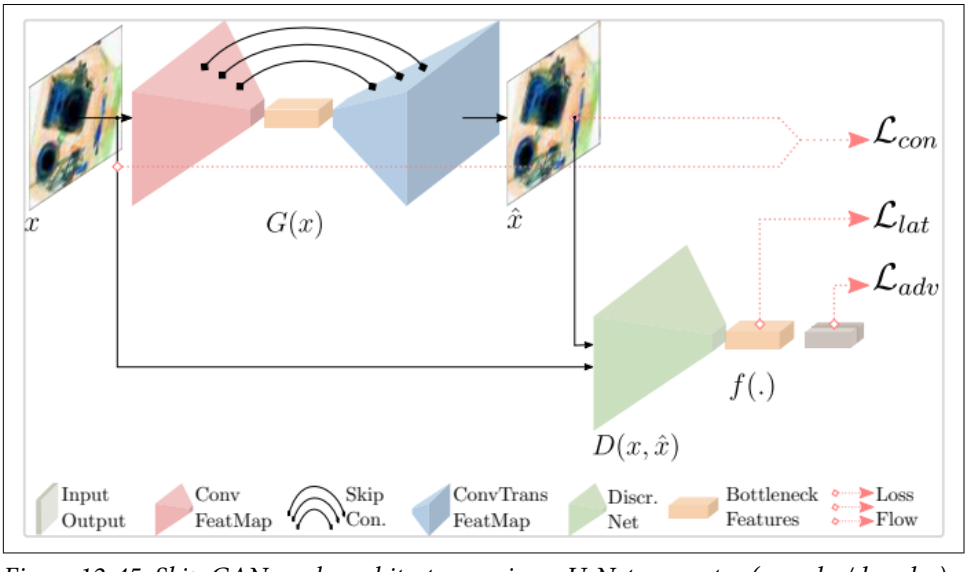

Figure 12-45. Skip-GANomaly architecture, using a U-Net generator (encoder/decoder) with skip connections, discriminator, and multiple loss terms. Image from Akçay et al., 2019.

The generator can be an autoencoder or U-Net, if the input and output are images, as in Figure 12-45's *G*, or the generator can just be a decoder that takes as input a user-provided random latent vector. This image autoencoder, since it's part of a GAN, is also sometimes called an adversarial autoencoder.

The discriminator, such as Figure 12-45's *D*, is used to adversarially train the generator. This is typically an encoder-type network, compressing an image down into a vector of logits to then be used for loss calculations.

As mentioned previously, sometimes there's an additional encoder or multiple generator/discriminator pairs. If the generator is an autoencoder, the additional encoder can be used for regularizing the intermediate bottleneck vector of the generator. If the generator is just a decoder, then the encoder can encode the generator's generated image into a feature vector to reconstruct the noise prior, essentially acting as the inverse of the generator.

As with SRGAN and inpainting, there are usually multiple loss terms: namely a reconstruction loss such as L_{con} and an adversarial loss such as L_{adv} in the example architecture in Figure 12-45. Additionally, there can be other loss terms like Figure 12-45's L_{lat}, which is a latent loss that sums the Euclidean distance between two feature maps in an intermediate layer from the discriminator. The weighted sum of these losses is designed to encourage the desired inference behavior. The adversarial loss ensures that the generator has learned the manifold of normal images.

The three-phase training procedure of image reconstruction, calculating the normal prediction error distribution, and applying the distance threshold will produce different results depending on whether normal or anomalous images are passed through the trained generator. Normal images will look very similar to the original images, so their reconstruction error will be low; therefore, when compared to the learned parameterized error distribution, they will have distances that are below the learned threshold. However, when the generator is passed an anomalous image, the reconstruction will no longer just be slightly worse. Therefore, the anomalies should be painted out, generating what the model thinks the image would look like without anomalies. The generator will essentially hallucinate what it thinks should be there based on its learned normal image manifold. Obviously this should result in a very large error when compared to the original anomalous image, allowing us to correctly flag the anomalous images or pixels for anomaly detection or localization, respectively.

Deepfakes

A popular technique that has recently exploded into the mainstream is the making of so-called *deepfakes*. Deepfakes replace objects or people in existing images or videos with different objects or people. The typical models used to create these deepfake images or videos are autoencoders or, with even better performance, GANs.

One method of creating deepfakes is to create one encoder and two decoders, A and B. Let's say we are trying to swap person X's face with person Y's. First, we distort an image of person X's face and pass that through the encoder to get the embedding, which is then passed through decoder A. This encourages the two networks to learn how to reconstruct person X's face from the noisy version. Next, we pass a warped version of person Y's face through the same encoder, and pass it through decoder B. This encourages these two networks to learn how to reconstruct person Y's face from the noisy version. We repeat this process over and over again until decoder A is great at producing clean images of person X and decoder B is great for person Y. The three networks have learned the essence of the two faces.

At inference time, if we now pass an image of person X through the encoder and then through decoder B, which was trained on the other person (person Y) instead of person X, the networks will think that the input is noisy and "denoise" the image into

person Y's face. Adding a discriminator for an adversarial loss can help improve the image quality.

There have been many other advancements in the creation of deepfakes, such as requiring only a single source image (often demonstrated by running the deepfake on a work of art such as the *Mona Lisa*). Remember, though, that to achieve great results, a lot of data is required to sufficiently train the networks.

Deepfakes are something that we need to keep a close eye on due to their possible abuse for political or financial gain—for instance, making a politician appear to say something they never did. There is a lot of active research looking into methods to detect deepfakes.

Image Captioning

So far in this chapter, we have looked at how to represent images (using encoders) and how to generate images from those representations (using decoders). Images are not the only thing worth generating from image representations, though—we might want to generate text based on the content of the images, a problem known as *image captioning*.

Image captioning is an asymmetric transformation problem. The encoder here operates on images, whereas the decoder needs to generate text. A typical approach is to use standard models for the two tasks, as shown in Figure 12-46. For example, we could use the Inception convolutional model to encode images into image embeddings, and a language model (marked by the gray box) for the sequence generation.

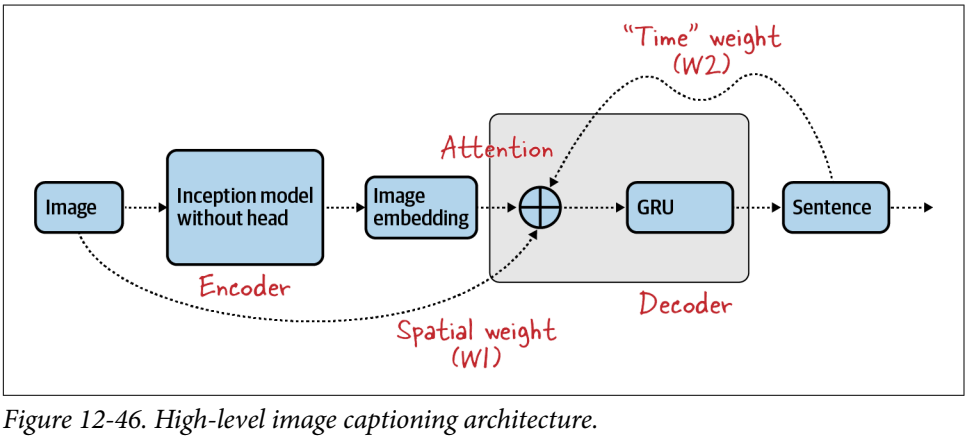

Figure 12-46. High-level image captioning architecture.

There are two important concepts that are necessary to understand what's happening in the language model: attention and gated recurrent units (GRUs).

Attention is important for the model to learn the relationships between specific parts in the image and specific words in the caption. This is accomplished by training the network such that it learns to focus its attention on specific parts of the image for specific words in the output sequence (see Figure 12-47). Therefore, the decoder incorporates a mechanism that *attends* over the image to predict the next word.

Figure 12-47. The model learns to predict the next word in the sequence by focusing its attention on the relevant part of the input image. The attention of the network at the time the word "frisbee" is to be predicted is shown in this figure. Image from Xu et al., 2016.

A *GRU cell* is the basic building block of a sequence model. Unlike the image models that we have seen in this book, language models need to remember what words they have already predicted. In order for a language model to take an English input sentence ("I love you") and translate it into French ("Je t'aime"), it is insufficient for the model to translate the sentence word-by-word. Instead, the model needs to have some memory. This is accomplished through a GRU cell that has an input, an output, an input state, and an output state. In order to predict the next word, the state is passed around from step to step, and the output of one step becomes the input to the next.

In this section we will build an end-to-end captioning model, starting with creating the dataset and preprocessing the captions and moving on to building the captioning model, training it, and using it to make predictions.

Dataset

To train a model to predict captions, we need a training dataset that consists of images and captions for those images. The COCO captions dataset is a large corpus of such captioned images. We will use a version of the COCO dataset that is part of TensorFlow Datasets—this version contains images, bounding boxes, labels, and captions from COCO 2014, split into the subsets defined by Karpathy and Li (2015), and

takes care of some data quality issues with the original dataset (for example, some of the images in the original dataset did not have captions).

We can create a training dataset using the following code (the full code is in *02e_image_captioning.ipynb* on GitHub):

```
def get_image_label(example):
    captions = example['captions']['text'] # all the captions
    img_id = example['image/id']
    img = example['image']
    img = tf.image.resize(img, (IMG_WIDTH, IMG_HEIGHT))
    img = tf.keras.applications.inception_v3.preprocess_input(img)
    return {
        'image_tensor': img,
        'image_id': img_id,
        'captions': captions
    }

    trainds = load_dataset(...).map(get_image_label)
```

This code applies the `get_image_label()` function to each of the examples that are read. This method pulls out the captions and the image tensor. The images are all different sizes, but we need them to be of shape (299, 299, 3) in order to use the pretrained Inception model. Therefore, we resize each image to the desired size.

Each image has multiple captions. A few example images and the first caption of each are shown in Figure 12-48.

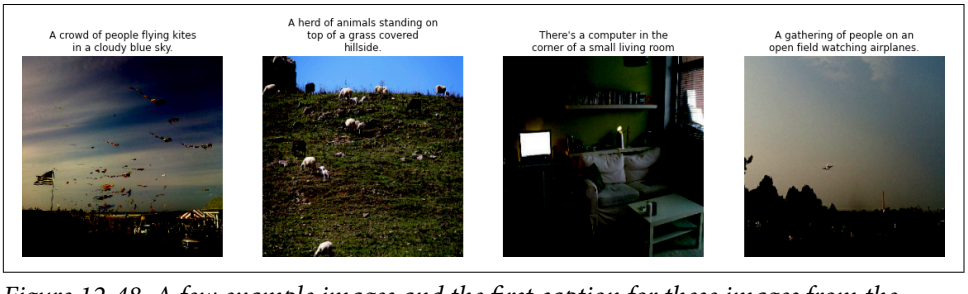

Figure 12-48. A few example images and the first caption for these images from the COCO dataset.

Tokenizing the Captions

Given a caption such as:

```
A toilet and sink in a tiled bathroom.
```

we need to remove punctuation, lowercase it, split into words, remove unusual words, add special start and stop tokens, and pad it to a consistent length:

```
['<start>', 'a', 'toilet', 'and', 'sink', 'in', 'a', 'tiled', 'bathroom', '<end>'
, '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>'
, '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>'
, '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>'
, '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>'
, '<pad>', '<pad>', '<pad>']
```

We start by adding the `<start>` and `<end>` tokens to each caption string:

```
train_captions = []
for data in trainds:
    str_captions = ["<start> {} <end>".format(
        t.decode('utf-8')) for t in data['captions'].numpy()]
    train_captions.extend(str_captions)
```

Then we use the Keras tokenizer to create the word-to-index lookup table:

```
tokenizer = tf.keras.layers.experimental.preprocessing.TextVectorization(
    max_tokens=VOCAB_SIZE, output_sequence_length=MAX_CAPTION_LEN)
tokenizer.adapt(train_captions)
```

The tokenizer can now be used to do the lookups in both directions:

```
padded = tokenizer(str_captions)
predicted_word = tokenizer.get_vocabulary()[predicted_id]
```

Batching

Each image in the COCO dataset can have up to five captions. So, given an image, we can actually generate up to five feature/label pairs (the image is the feature, the caption is the label). Because of this, creating a batch of training features is not as easy as:

```
trainds.batch(32)
```

since these 32 examples will expand into anywhere from 32 to 32 * 5 potential examples. We need batches to be of consistent size, so we will have to use the training dataset to generate the necessary examples before batching them:

```
def create_batched_ds(trainds, batchsize):
    # generator that does tokenization, padding on the caption strings
    # and yields img, caption
    def generate_image_captions():
        for data in trainds:
            captions = data['captions']
            img_tensor = data['image_tensor']
            str_captions = ["starttoken {} endtoken".format(
                t.decode('utf-8')) for t in captions.numpy()]
            # Pad each vector to the max_length of the captions
            padded = tokenizer(str_captions)
            for caption in padded:
                yield img_tensor, caption # repeat image
        return tf.data.Dataset.from_generator(
            generate_image_captions,
            (tf.float32, tf.int32)).batch(batchsize)
```

Note that we are reading the caption strings, and applying the same processing to each of these strings that we did in the previous section. That was for the purpose of creating the word-to-index lookup tables and computing the maximum caption length over the entire dataset so that captions can be padded to the same length. Here, we simply apply the lookup tables and pad the captions based on what was calculated over the full dataset.

We can then create batches of 193 image/caption pairs by:

```
create_batched_ds(trainds, 193)
```

Captioning Model

The model consists of an image encoder followed by a caption decoder (see Figure 12-46). The caption decoder incorporates an attention mechanism that focuses on different parts of the input image.

Image encoder

The image encoder consists of the pretrained Inception model followed by a `Dense` layer:

```
class ImageEncoder(tf.keras.Model):
    def __init__(self, embedding_dim):
    inception = tf.keras.applications.InceptionV3(
        include_top=False,
        weights='imagenet'
    )
    self.model = tf.keras.Model(inception.input,
                                inception.layers[-1].output)
    self.fc = tf.keras.layers.Dense(embedding_dim)
```

Invoking the image encoder applies the Inception model, flattens the result from the [batch, 8, 8, 2048] that Inception returns to [batch, 64, 2048], and passes it through the `Dense` layer:

```
def call(self, x):
    x = self.model(x)
    x = tf.reshape(x, (x.shape[0], -1, x.shape[3]))
    x = self.fc(x)
    x = tf.nn.relu(x)
    return x
```

Attention mechanism

The attention component is complicated—look at the following description in conjunction with Figure 12-46 and the complete code in *02e_image_captioning.ipynb* on GitHub.

Recall that attention is how the model learns the relationships between specific parts in the image and specific words in the caption. The attention mechanism consists of two sets of weights—`W1` is a dense layer meant for the spatial component (`features`, where in the image to focus on), and `W2` is a dense layer for the "temporal" component (indicating which word in the input sequence to focus on):

```
attention_hidden_layer = (tf.nn.tanh(self.W1(features) +
                                     self.W2(hidden_with_time_axis)))
```

The weighted attention mechanism is applied to the hidden state of the recurrent neural network to compute a score:

```
score = self.V(attention_hidden_layer)
attention_weights = tf.nn.softmax(score, axis=1)
```

`V` here is a dense layer that has a one output node that is passed through a softmax layer to obtain a final combined weight that adds up to 1 across all the words. The features are weighted by this value, and this is an input to the decoder:

```
context_vector = attention_weights * features
```

This attention mechanism is part of the decoder, which we'll look at next.

Caption decoder

Recall that the decoder needs to have some memory of what it has predicted in the past, and so the state is passed around from step to step, with the output of one step becoming the input to the next. Meanwhile, during training, the caption words are fed into the decoder one word at a time.

The decoder takes the caption words one a time (x in the following listing) and converts each word into its word embedding. The embedding is then concatenated with the context output of the attention mechanism (which specifies where in the image the attention mechanism is currently focused) and passed into a recurrent neural network cell (a GRU cell is used here):

```
x = self.embedding(x)
x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)
output, state = self.gru(x)
```

The output of the GRU cell is then passed through a set of dense layers to obtain the decoder output. The output here would normally be a softmax because the decoder is a multilabel classifier—we need the decoder to tell which of the five thousand words the next word needs to be. However, for reasons that will become apparent in the section on predictions, it is helpful to keep the output as logits.

Putting these pieces together, we have:

```
encoder = ImageEncoder(EMBED_DIM)
decoder = CaptionDecoder(EMBED_DIM, ATTN_UNITS, VOCAB_SIZE)
optimizer = tf.keras.optimizers.Adam()
```

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')
```

The loss function of the captioning model is a bit tricky. It's not simply the mean cross-entropy over the entire output, because we need to ignore the padded words. Therefore, we define a loss function that masks out the padded words (which are all zeros) before computing the mean:

```
def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)
    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask
    return tf.reduce_mean(loss_)
```

Training Loop

Now that our model has been created, we can move on to training it. You might have noticed that we don't have a single Keras model—we have an encoder and a decoder. That is because it is not enough to call `model.fit()` on the entire image and caption—we need to pass in the caption words to the decoder one by one because the decoder needs to learn how to predict the next word in the sequence.

Given an image and a target caption, we initialize the loss and reset the decoder state (so that the decoder doesn't continue with the words of the previous caption):

```
def train_step(img_tensor, target):
    loss = 0
    hidden = decoder.reset_state(batch_size=target.shape[0])
```

The decoder input starts with a special start token:

```
dec_input = ... tokenizer(['starttoken'])...
```

We invoke the decoder and compute the loss by comparing the decoder's output against the next word in the caption:

```
for i in range(1, target.shape[1]):
    predictions, hidden, _ = decoder(dec_input, features, hidden)
    loss += loss_function(target[:, i], predictions)
```

We are adding the *i*th word to the decoder input each time so that the model learns based on the correct caption, not based on whatever the predicted word is:

```
dec_input = tf.expand_dims(target[:, i], 1)
```

This is called *teacher forcing*. Teacher forcing swaps the target word in the input with the predicted word from the last step.

The whole set of operations just described has to be captured for the purpose of computing gradient updates, so we wrap it in a `GradientTape`:

```
with tf.GradientTape() as tape:
    features = encoder(img_tensor)
    for i in range(1, MAX_CAPTION_LENGTH):
        predictions, hidden, _ = decoder(dec_input, features, hidden)
        loss += loss_function(target[:, i], predictions)
        dec_input = tf.expand_dims(target[:, i], 1)
```

We can then update the loss, and apply gradients:

```
    total_loss = (loss / MAX_CAPTION_LENGTH)
    trainable_variables = \
        encoder.trainable_variables + decoder.trainable_variables
    gradients = tape.gradient(loss, trainable_variables)
    optimizer.apply_gradients(zip(gradients, trainable_variables))
```

Now that we have defined what happens in a single training step, we can loop through it for the desired number of epochs:

```
batched_ds = create_batched_ds(trainds, BATCH_SIZE)
for epoch in range(EPOCHS):
    total_loss = 0
    num_steps = 0
    for batch, (img_tensor, target) in enumerate(batched_ds):
        batch_loss, t_loss = train_step(img_tensor, target)
        total_loss += t_loss
        num_steps += 1
    # storing the epoch end loss value to plot later
    loss_plot.append(total_loss / num_steps)
```

Prediction

For the purpose of prediction, we are given an image and need to generate a caption. We start the caption string with the token `<start>` and feed the image and the initial token to the decoder. The decoder returns a set of logits, one for each of the words in our vocabulary.

Now we need to use the logits to get the next word. There are several approaches we could follow:

- A greedy approach where we pick the word with the maximum log-likelihood. This essentially means that we do `tf.argmax()` on the logits. This is fast but tends to overemphasize uninformative words like "a" and "the."
- A *beam search* method where we pick the top three or five candidates. We will then force the decoder with each of these words, and pick the next word in the sequence. This creates a tree of output sequences, from which the highest-probability sequence is selected. Because this optimizes the probability of the sequence rather than of individual words, it tends to give the best results, but it's computationally quite expensive and can lead to high latencies.

- A probabilistic method where we choose the word in proportion to its likelihood—in TensorFlow, this is achieved using `tf.random.categorical()`. If the word following "crowd" is 70% likely to be "people" and 30% likely to be "watching," then the model chooses "people" with a 70% likelihood, and "watching" with a 30% probability, so that the less likely phrase is also explored. This is a reasonable trade-off that achieves both novelty and speed at the expense of being nonreproducible.

Let's try out the third approach.

We start by applying all the preprocessing to the image, and then send it to the image encoder:

```
def predict_caption(filename):
    attention_plot = np.zeros((max_caption_length, ATTN_FEATURES_SHAPE))
    hidden = decoder.reset_state(batch_size=1)
    img = tf.image.decode_jpeg(tf.io.read_file(filename),
                               channels=IMG_CHANNELS)
    img = tf.image.resize(img, (IMG_WIDTH, IMG_HEIGHT)) # inception size
    img_tensor_val = tf.keras.applications.inception_v3.preprocess_input(img)

    features = encoder(tf.expand_dims(img_tensor_val, axis=0))
```

We then initialize the decoder input with the token `<start>` and invoke the decoder repeatedly until an `<end>` caption is received or the maximum caption length is reached:

```
dec_input = tf.expand_dims([tokenizer(['starttoken'])], 0)
result = []
for i in range(max_caption_length):
    predictions, hidden = decoder(dec_input, features, hidden)
    # draws from log distribution given by predictions
    predicted_id = tf.random.categorical(predictions, 1)[0][0].numpy()
    result.append(tokenizer.vocabulary()[predicted_id])
    if tokenizer.vocabulary()[predicted_id] == 'endtoken':
        return result
    dec_input = tf.expand_dims([predicted_id], 0)

return img, result, attention_plot
```

An example image and captions generated from it are shown in Figure 12-49. The model seems to have captured that this is a group of people on a field playing baseball. However, the model believes that there is a high likelihood that the white line in the center is the median divider of a street, and that this game could have been played on the street. Stopwords (*of*, *in*, *and*, *a*, etc.) are not generated by the model because we removed them from the training dataset. Had we had a larger dataset, we could have tried to generate proper sentences by keeping those stopwords in.

Figure 12-49. An example image, courtesy of the author, and a few of the captions generated by the model.

At this point, we now have an end-to-end image captioning model. Image captioning is an important way to make sense of a large corpus of images and is starting to find use in a number of applications, such as generating image descriptions for the visually impaired, meeting accessibility requirements in social media, generating audio guides like those used in museums, and performing cross-language annotation of images.

Summary

In this chapter, we looked at how to generate images and text. To generate images, we first create latent representations of images using an autoencoder (or variational autoencoder). A latent vector passed through a trained decoder functions as an image generator. In practice, however, the generated images are too obviously fake. To improve the realism of the generated images, we can use GANs, which use a game theoretic approach to train a pair of neural networks. Finally, we looked at how to implement image captioning by training an image encoder and a text decoder along with an attention mechanism.

Afterword

In 1966, MIT professor Seymour Papert launched a summer project for his students. The final goal of the project was to name objects in images by matching them with a vocabulary of known objects. He helpfully broke the task down for them into subprojects, and expected the group to be done in a couple of months. It's safe to say that Dr. Papert underestimated the complexity of the problem a little.

We started this book by looking at naive machine learning approaches like fully connected neural networks that do not take advantage of the special characteristics of images. In Chapter 2, trying the naive approaches allowed us to learn how to read in images, and how to train, evaluate, and predict with machine learning models.

Then, in Chapter 3, we introduced many of the innovative concepts—convolutional filters, max-pooling layers, skip connections, modules, squeeze activation, and so on—that enable modern-day machine learning models to work well at extracting information from images. Implementing these models, practically speaking, involves using either a built-in Keras model or a TensorFlow Hub layer. We also covered transfer learning and fine-tuning in detail.

In Chapter 4, we looked at how to use the computer vision models covered in Chapter 3 to solve two more fundamental problems in computer vision: object detection and image segmentation.

The next few chapters of the book covered, in depth, each of the stages involved in creating production computer vision machine learning models:

- In Chapter 5, we covered how to create a dataset in a format that will be efficient for machine learning. We also discussed the options available for label creation and for keeping an independent dataset for model evaluation and hyperparameter tuning.

- In Chapter 6, we did a deep dive into preprocessing and preventing training-serving skew. Preprocessing can be done in the `tf.data` input pipeline, in Keras layers, in `tf.transform`, or using a mix of these methods. We covered both the implementation details and the pros and cons of each approach.
- In Chapter 7, we discussed model training, including how to distribute the training across GPUs and workers.
- In Chapter 8, we explored how to monitor and evaluate models. We also looked at how to carry out sliced evaluations to diagnose unfairness and bias in our models.
- In Chapter 9, we discussed the options available for deploying models. We implemented batch, streaming, and edge prediction. We were able to invoke our models locally, and across the web.
- In Chapter 10, we showed you how to tie together all these steps into a machine learning pipeline. We also tried out a no-code image classification system to take advantage of the ongoing democratization of machine learning.

In Chapter 11, we widened our lens beyond image classification. We looked at how the basic building blocks of computer vision can be used to solve a variety of problems including counting, pose detection, and other use cases. Finally, in Chapter 12, we looked at how to generate images and captions.

Throughout the book, the concepts, models, and processes discussed are accompanied by implementations in GitHub. We strongly recommend that you not just read this book, but also work through the code and try it out. The best way to learn machine learning is to do it.

Computer vision is at an exciting stage. The underlying technologies work well enough that today, more than 50 years after Dr. Papert posed the problem to his students, we are finally at the point where image classification *can* be a two-month project! We wish you much success in applying this technology to better human lives and hope that it brings you as much joy to use computer vision to solve real-world problems as it has brought us.

Index

Symbols

A

B

C

D

E

F

G

J

K

L

M

N

O

P

T

U

V

W

X

Y

About the Authors

Valliappa (Lak) Lakshmanan is the director of analytics and AI solutions at Google Cloud, where he leads a team building cross-industry solutions to business problems. His mission is to democratize machine learning so that it can be done by anyone anywhere.

Martin Görner is a product manager for Keras/TensorFlow focused on improving the developer experience when using state-of-the-art models. He's passionate about science, technology, coding, algorithms, and everything in between.

Ryan Gillard is an AI engineer in Google Cloud's Professional Services organization, where he builds ML models for a wide variety of industries. He started his career as a research scientist in the hospital and healthcare industry. With degrees in neuroscience and physics, he loves working at the intersection of those disciplines exploring intelligence through mathematics.

Colophon

The bird on the cover of *Practical Machine Learning for Computer Vision* is an emerald toucanet (*Aulacorhynchus prasinus*), the smallest species of toucan. Central and South America have large populations from the cloud forests of Costa Rica to Venezuela.

Vibrant green feathers camouflage emerald toucanets in the tropics. Adults typically measure 12–13 inches long, weigh just over 5 ounces, and live 10–11 years in the wild. Their beaks are colorful: yellow on top, a white outline, and red or black on the bottom. They eat fruit and insects, as well as small lizards and the eggs and young of other birds. Groups of about eight will hunt and forage together. Emerald toucanets build their nests by enlarging the nests of smaller birds. The male and female trade off shifts in the nest, incubating, feeding, and cleaning their chicks.

Deforestation has driven emerald toucanets into shade coffee farms. Overall, their population is decreasing. Many of the animals on O'Reilly's covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Shaw's Zoology*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.