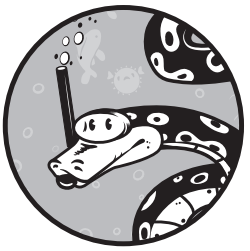
12

A CASE STUDY: CLASSIFYING AUDIO SAMPLES



Let's bring together everything that you've learned so far in the book. We'll be looking at a single case study. Here's our scenario: we are data scientists, and our boss has tasked us with building a classifier for audio samples stored as *.wav* files. We'll begin with the data itself. We first want to build some basic intuition for how it's structured. From there, we'll build augmented datasets for training models.

The first dataset uses the sound samples themselves, a 1D dataset. You'll see that this approach isn't as successful as we'd like it to be.

We'll then turn the audio data into images to allow us to explore 2D CNNs. This change of representation will lead to a big improvement in model performance. Finally, we'll use an ensemble of models to leverage the relative strengths and weaknesses of the individual models to boost overall performance still more.

Building the Dataset

There are 10 classes in our dataset, which consists of 400 samples total, 40 samples per class, each 5 seconds long. We'll assume we cannot get any more data because it's time-consuming and expensive to record the samples and label them. We must work with the data we are given and no more.

Throughout this book, I've consistently preached about the necessity of having a good dataset. We'll assume that the dataset we've been handed is complete in the sense that our system will encounter only types of sound samples in the dataset; there will be no unknown class or classes. Additionally, we'll assume that the balanced nature of the dataset is real, and all classes are indeed equally likely.

The audio dataset we'll use is called ESC-10. For a complete description, see "ESC: Dataset for Environmental Sound Classification" by Karol J. Piczak (2015). The dataset is available at *https://github.com/karoldvl/ESC-50*, but it needs to be extracted from the larger ESC-50 dataset, which doesn't have a license we can use. The ESC-10 subset does.

Let's do some preprocessing to extract the ESC-10 *.wav* files from the larger ESC-50 dataset. Download the single ZIP file version of the dataset from the preceding URL to the *src/chapter_14* directory and expand it. This will create a directory called *ESC-50-master*. Then, use the code in Listing 12-1 to build the ESC-10 dataset from it.

extract_esc10.py

```
import sys
import os
import shutil

classes = {
    "rain":0,
    "rooster":1,
    "crying_baby":2,
    "sea_waves":3,
    "clock_tick":4,
    "sneezing":5,
    "dog":6,
    "crackling_fire":7,
    "helicopter":8,
    "chainsaw":9,
}

with open("ESC-50-master/meta/esc50.csv") as f:
    lines = [i[:-1] for i in f.readlines()]
lines = lines[1:]

os.system("rm -rf ../data/audio/ESC-10")
os.system("mkdir ../data/audio/ESC-10")
os.system("mkdir ../data/audio/ESC-10/audio")
```

```
meta = []
for line in lines:
    t = line.split(",")
    if (t[-3] == 'True'):
        meta.append("../data/audio/ESC-10/audio/%s %d" % (t[0],classes[t[3]]))
        src = "ESC-50-master/audio/"+t[0]
        dst = "../data/audio/ESC-10/audio/"+t[0]
        shutil.copy(src,dst)

with open("../data/audio/ESC-10/filelist.txt","w") as f:
    for m in meta:
        f.write(m+"\n")
```

Listing 12-1: Building the ESC-10 dataset

The code uses the ESC-50 metadata to identify the sound samples that belong to the 10 classes of the ESC-10 dataset and then copies them to the *ESC-10/audio* directory. It also writes a list of the audio files to *filelist.txt*. After running this code, we'll use only the ESC-10 files.

If all is well, we should now have 400 five-second *.wav* files, 40 from each of the 10 classes: rain, rooster, crying baby, sea waves, clock tick, sneezing, dog, crackling fire, helicopter, and chainsaw. We'll politely refrain from asking our boss exactly why she wants to discriminate between these particular classes of sound.

Augmenting the Dataset

Our first instinct should be that our dataset is too small. After all, we have only 40 examples of each sound, and some of those will need to be held back for testing, leaving even fewer per class for training.

We could resort to *k*-fold validation, but in this case, we'll instead opt for data augmentation. So, how do we augment audio data?

Recall, the goal of data augmentation is to create new data samples that could plausibly come from the classes in the dataset. With images, we can make obvious changes like shifting, flipping left and right, and so on. With continuous vectors, we've seen how to use PCA to augment the data (see Chapter 2). To augment the audio files, we need to think of steps we can take to produce new files that still sound like the original class. Four thoughts come to mind.

First, we can shift the sample in time, much as we can shift an image to the left or right a few pixels. Second, we can simulate a noisy environment by adding a small amount of random noise to the sound itself. Third, we can shift the pitch of the sound, and make it higher or lower by a small amount; this is known as *pitch shifting*. Finally, we can lengthen or compress the sound in time. This is known as *time shifting*.

Doing all this sounds complicated, especially if we haven't worked with audio data before. In practice, being presented with unfamiliar data is a very real possibility; we don't all get to choose what we need to work with.

Fortunately for us, we're working in Python, and the Python community is vast and talented. It turns out that adding one library, librosa, to our system will allow us to easily perform time stretching and pitch shifting. Let's install the library:

```
$ sudo pip3 install librosa
```

With the necessary library installed, we can augment the ESC-10 dataset with the code in Listing 12-2 (*make_augmented_audio.py*).

```
import os
import random
import numpy as np
from scipy.io.wavfile import read, write
import librosa as rosa

N = 8
os.system("rm -rf ../data/audio/ESC-10/augmented; mkdir ../data/audio/ESC-10/augmented")
os.system("mkdir ../data/audio/ESC-10/augmented/train ../data/audio/ESC-10/augmented/test")

src_list = [i[:-1] for i in open("../data/audio/ESC-10/filelist.txt")]
z = [[] for i in range(10)]
for s in src_list:
    _,c = s.split()
    z[int(c)].append(s)

train = []
test = []
for i in range(10):
    p = z[i]
    random.shuffle(p)
    test += p[:8]
    train += p[8:]

random.shuffle(train)
random.shuffle(test)
augment_audio(train, "train")
augment_audio(test, "test")
```

Listing 12-2: Augmenting the ESC-10 dataset, part 1

This code loads the necessary modules, including the `librosa` module, which we'll just call `rosa`, and two functions from the SciPy `wavfile` module that let us read and write NumPy arrays as *.wav* files.

We set the number of samples per class that we'll hold back for testing (`N=8`) and create the output directory where the augmented sound files will reside (`augmented`). Then we read the file list we created with Listing 12-1. Next, we create a nested list (`z`) to hold the names of the audio files associated with each of the 10 classes.

Using the list of files per class, we pull it apart and create `train` and `test` file lists. Notice that we randomly shuffle the list of files per class and the final `train` and `test` lists. This code follows the convention we discussed in Chapter 1 of separating train and test first, then augmenting.

We can augment the train and test files by calling `augment_audio`. This function is in Listing 12-3.

```
def augment_audio(src_list, typ):
    flist = []
    for i,s in enumerate(src_list):
        f,c = s.split()
        wav = read(f) # (sample rate, data, type) ❶
        base = os.path.abspath("../data/audio/ESC-10/augmented/%s/%s"
                   % (typ, os.path.basename(f)[:-4]))
        fname = base+".wav"
        write(fname, wav[0], wav[1]) ❷
        flist.append("%s %s" % (fname,c))
        for j in range(19):
            d = augment(wav)
            fname = base+("_%04d.wav" % j)
            write(fname, wav[0], d.astype(wav[1].dtype)) ❸
            flist.append("%s %s" % (fname,c))
    random.shuffle(flist)
    with open("../data/audio/ESC-10/augmented_%s_filelist.txt" % typ,"w") as f:
        for z in flist:
            f.write("%s\n" % z)
```

Listing 12-3: Augmenting the ESC-10 dataset, part 2

The function loops over all the filenames in the given list (`src_list`), which will be either train or test. The filename is separated from the class label, and then the file is read from disk ❶. As indicated in the comment, `wav` is a list of two elements. The first is the sampling rate in Hz (cycles per second). This is how often the analog waveform was digitized to produce the *.wav* file. For ESC-10, the sampling rate is always 44,100 Hz, which is the standard rate for a compact disc. The second element is a NumPy array containing the actual digitized sound samples. These are the values we'll augment to produce new data files.

After setting up some output pathnames, we write the original sound sample to the augmented directory ❷. Then, we start a loop to generate 19 more augmented versions of the current sound sample. The augmented dataset, as a whole, will be 20 times larger, for a total of 8,000 sound files, 6,400 for training and 1,600 for testing. Note, the sound samples for an augmented source file are assigned to `d`. The new sound file is written to disk by using the sample rate of 44,100 Hz and the augmented data matching the data type of the source ❸.

As we create the augmented sound files, we also keep track of the filename and class and write them to a new file list. Here, `typ` is a string indicating train or test.

This function calls yet another function, `augment`. This is the function that generates an augmented version of a single sound file by randomly applying a subset of the four augmentation strategies mentioned previously: shifting, noise, pitch shifting, or time shifting. Some or all of these might be used for any call to `augment`. Listing 12-4 shows the `augment` function itself.

```
def augment(wav):
    sr = wav[0]
    d = wav[1].astype("float32")
❶   if (random.random() < 0.5):
        s = int(sr/4.0*(np.random.random()-0.5))
        d = np.roll(d,s)
        if (s < 0):
            d[s:] = 0
        else:
            d[:s] = 0
❷   if (random.random() < 0.5):
        d += 0.1*(d.max()-d.min())*np.random.random(d.shape[0])
❸   if (random.random() < 0.5):
        pf = 20.0*(np.random.random()-0.5)
        d = rosa.effects.pitch_shift(d, sr=sr, n_steps=pf)
❹   if (random.random() < 0.5):
        rate = 1.0 + (np.random.random()-0.5)
        d = rosa.effects.time_stretch(d, rate=rate)
        if (d.shape[0] > wav[1].shape[0]):
            d = d[:wav[1].shape[0]]
        else:
            w = np.zeros(wav[1].shape[0], dtype="float32")
            w[:d.shape[0]] = d
            d = w.copy()
    return d
```

Listing 12-4: Augmenting the ESC-10 dataset, part 3

This function separates the samples (`d`) from the sample rate (`sr`) and makes sure the samples are floating-point numbers. For ESC-10, the source samples are all of type `int16` (signed 16-bit integers). Next come four `if` statements. Each asks for a single random float, and if that float is less than 0.5, we execute the body of the `if`. This means that we apply each possible augmentation with a probability of 50 percent.

The first `if` shifts the sound samples in time ❶ by rolling the NumPy array, a vector, by a number of samples, `s`. This value amounts to at most an eighth of a second, `sr/4.0`. Note that the shift can be positive or negative. The quantity `sr/4.0` is the number of samples in a quarter of a second. However, the random float is in the range [−0.5, +0.5], so the ultimate shift is at most an eighth of a second. If the shift is negative, we need to zero samples at the end of the data; otherwise, we zero samples at the start.

Random noise is added by literally adding a random value of up to one-tenth of the range of the audio signal back in ❷. When played, this adds hiss, as you might hear on an old cassette tape.

Next comes shifting the pitch of the sample by using `librosa`. The pitch shift is expressed in musical steps, or fractions thereof. We randomly pick a float in the range [−10, +10] (`pf`) and pass it along with the data (`d`) and sampling rate (`sr`) to the `librosa` `pitch_shift` effect function ❸.

The last augmentation uses the `librosa` function to stretch or compress time (`time_stretch`) ❹. We adjust using an amount of time (`rate`) that is in the range [−0.5, +0.5]. If time was stretched, we need to chop off the extra samples to ensure that the sample length remains constant. If time was compressed, we need to add zero samples at the end.

Lastly, we return the new, augmented samples.

Running the code in Listing 12-2 takes about 11 minutes and creates a new *augmented* data directory in the *data/audio/ESC-10* directory with subdirectories *train* and *test*. These are the raw sound files that we'll work with going forward. I encourage you to listen to some of them to understand what the augmentations have done. The filenames should differentiate the originals from the augmentations.

Preprocessing the Data

Are we ready to start building models? Not yet. Our experience told us that the dataset was too small, and we augmented accordingly. However, we haven't yet turned the raw data into something we can pass to a model.

A first thought is to use the raw sound samples. These are already vectors representing the audio signal, with the time between the samples set by the sampling rate of 44,100 Hz. But we don't want to use them as they are. The samples are all exactly five seconds long. At 44,100 samples per second, that means each sample is a vector of $44{,}100 \times 5 = 220{,}500$ samples. That's too long for us to work with effectively.

With a bit more thought, we might be able to convince ourselves that distinguishing between a crying baby and a barking dog might not need such a high sampling rate. What if instead of keeping all the samples, we kept only every 100th sample? Moreover, do we really need five seconds' worth of data to identify the sounds? What if we kept only the first two seconds?

Let's keep only the first two seconds of each sound file; that's 88,200 samples. And let's keep only every 100th sample, so each sound file now becomes a vector of 882 elements. That's hardly more than an unraveled MNIST digit image, and we know we can work with those.

Listing 12-5 has the code to build the actual initial version of the dataset we'll use to build the models.

```
import os
import random
import numpy as np
from scipy.io.wavfile import read
```

```
sr = 44100 # Hz
N = 2*sr   # number of samples to keep
w = 100    # every 100 (0.01 s)

afiles = [i[:-1] for i in open("../data/audio/ESC-10/augmented_train_filelist.txt")]
trn = np.zeros((len(afiles),N//w,1), dtype="int16")
lbl = np.zeros(len(afiles), dtype="uint8")

for i,t in enumerate(afiles):
 ❶ f,c = t.split()
    trn[i,:,0] = read(f)[1][:N:w]
    lbl[i] = int(c)

np.save("../data/audio/ESC-10/esc10_raw_train_audio.npy", trn)
np.save("../data/audio/ESC-10/esc10_raw_train_labels.npy", lbl)

afiles = [i[:-1] for i in open("../data/audio/ESC-10/augmented_test_filelist.txt")]
tst = np.zeros((len(afiles),N//w,1), dtype="int16")
lbl = np.zeros(len(afiles), dtype="uint8")

for i,t in enumerate(afiles):
    f,c = t.split()
    tst[i,:,0] = read(f)[1][:N:w]
    lbl[i] = int(c)

np.save("../data/audio/ESC-10/esc10_raw_test_audio.npy", tst)
np.save("../data/audio/ESC-10/esc10_raw_test_labels.npy", lbl)
```

Listing 12-5: Building the reduced samples dataset

This code builds train and test NumPy files containing the raw data. The data is from the augmented sound files we built in Listing 12-2. The file list contains the file location and class label ❶. We load each file in the list and put it into an array, either the train or test array.

We have a 1D feature vector and a number of train or test files, so we might expect we need a 2D array to store our data, either 6,400×882 for the training set or 1,600×882 for the test set. However, we know we'll ultimately be working with Keras, and we know that Keras wants a dimension for the number of channels, so we define the arrays to be 6,400×882×1 and 1,600×882×1 instead. The most substantial line in this code is the following:

```
trn[i,:,0] = read(f)[1][:N:w]
```

It reads the current sound file, keeps only the sound samples (`[1]`), and from the sound samples keeps only the first two seconds' worth at every 100th sample, `[:N:w]`. Spend a little time with this code. If you're confused, I suggest experimenting with NumPy at the interactive Python prompt to understand what it's doing.

In the end, we have train and test files for the 882-element vectors and associated labels. We'll build our first models with these. Figure 12-1 shows the resulting vector for a crying baby.

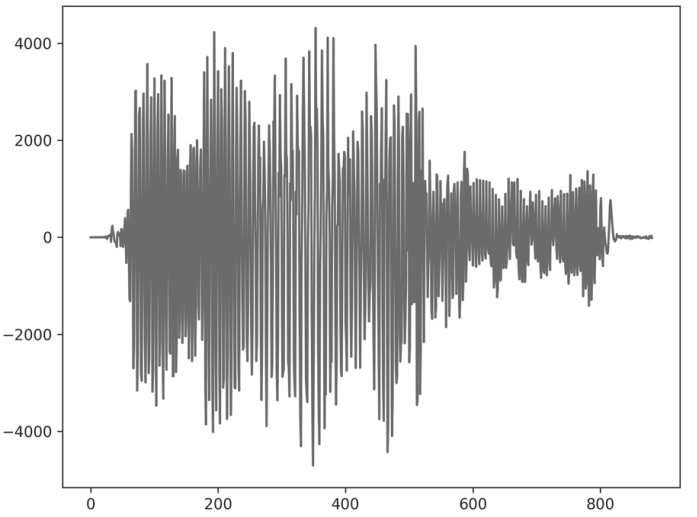


Figure 12-1: The feature vector for a crying baby

The x-axis is the sample number (think *time*), and the y-axis is the sample value.

Classifying the Audio Features

We have our training and test sets. Let's build some models and see how they do. Since we have feature vectors, we can start quickly with classical models. After those, we can build some 1D convolutional networks and see if they perform any better.

With Classical Models

We can test the same suite of classical models we used in Chapter 4 with the breast cancer dataset. Listing 12-6 has the setup code.

esc10_audio_classical.py

```
import numpy as np
from sklearn.neighbors import NearestCentroid
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC

x_train = np.load("../data/audio/ESC-10/esc10_raw_train_audio.npy")[:,:,0]
y_train = np.load("../data/audio/ESC-10/esc10_raw_train_labels.npy")
x_test  = np.load("../data/audio/ESC-10/esc10_raw_test_audio.npy")[:,:,0]
y_test  = np.load("../data/audio/ESC-10/esc10_raw_test_labels.npy")
```

```
❶ x_train = (x_train.astype('float32') + 32768) / 65536
  x_test = (x_test.astype('float32') + 32768) / 65536

  train(x_train, y_train, x_test, y_test)
```

Listing 12-6: Classifying the audio features with classical models, part 1

We import the necessary model types, load the dataset, scale it, and then call a `train` function that we'll introduce shortly.

Scaling is crucial here. Consider the y-axis range for Figure 12-1. It goes from about −4,000 to 4,000. We need to scale the data so that the range is smaller and the values are closer to being centered around 0. Recall, for the MNIST and CIFAR-10 datasets, we divided by the maximum value to scale to [0, 1].

The sound samples are 16-bit signed integers. This means the full range of values they can take on covers [−32,768, +32,767]. If we make the samples floats, add 32,768, and then divide by 65,536 (twice the lower value) ❶, we'll get samples in the range [0, 1), which is what we want.

Training and evaluating the classical models is straightforward, as Listing 12-7 shows.

```
def run(x_train, y_train, x_test, y_test, clf):
    clf.fit(x_train, y_train)
    score = 100.0*clf.score(x_test, y_test)
    print("score = %0.2f%%" % score)

def train(x_train, y_train, x_test, y_test):
    print("Nearest Centroid          : ", end='')
    run(x_train, y_train, x_test, y_test, NearestCentroid())
    print("k-NN classifier (k=3)     : ", end='')
    run(x_train, y_train, x_test, y_test, KNeighborsClassifier(n_neighbors=3))
    print("k-NN classifier (k=7)     : ", end='')
    run(x_train, y_train, x_test, y_test, KNeighborsClassifier(n_neighbors=7))
    print("Naive Bayes (Gaussian)    : ", end='')
    run(x_train, y_train, x_test, y_test, GaussianNB())
    print("Random Forest (trees=  5) : ", end='')
    run(x_train, y_train, x_test, y_test,
        RandomForestClassifier(n_estimators=5))
    print("Random Forest (trees= 50) : ", end='')
    run(x_train, y_train, x_test, y_test,
        RandomForestClassifier(n_estimators=50))
    print("Random Forest (trees=500) : ", end='')
    run(x_train, y_train, x_test, y_test,
        RandomForestClassifier(n_estimators=500))
    print("Random Forest (trees=1000): ", end='')
    run(x_train, y_train, x_test, y_test,
        RandomForestClassifier(n_estimators=1000))
    print("LinearSVM (C=0.01)        : ", end='')
```

```
    run(x_train, y_train, x_test, y_test, LinearSVC(C=0.01))
    print("LinearSVM (C=0.1)        : ", end='')
    run(x_train, y_train, x_test, y_test, LinearSVC(C=0.1))
    print("LinearSVM (C=1.0)        : ", end='')
    run(x_train, y_train, x_test, y_test, LinearSVC(C=1.0))
    print("LinearSVM (C=10.0)       : ", end='')
    run(x_train, y_train, x_test, y_test, LinearSVC(C=10.0))
```

Listing 12-7: Classifying the audio features with classical models, part 2

The `train` function creates the particular model instances and then calls `run`. We saw this same code structure in Chapter 4. The `run` function uses `fit` to train the model and `score` to score the model on the test set. For the time being, we'll evaluate the models based solely on their overall accuracy (the score). Running this code takes about eight minutes to produce output like this:

```
Nearest Centroid          : score = 11.9%
k-NN classifier (k=3)     : score = 12.1%
k-NN classifier (k=7)     : score = 10.5%
Naive Bayes (Gaussian)    : score = 28.1%
Random Forest (trees=  5) : score = 22.6%
Random Forest (trees= 50) : score = 30.8%
Random Forest (trees=500) : score = 32.8%
Random Forest (trees=1000): score = 34.4%
LinearSVM (C=0.01)        : score = 16.5%
LinearSVM (C=0.1)         : score = 17.5%
LinearSVM (C=1.0)         : score = 13.4%
LinearSVM (C=10.0)        : score = 10.2%
```

We can quickly see that the classical models have performed terribly. Many are essentially guessing the class label. There are 10 classes, so random chance guessing should have an accuracy of around 10 percent. The best-performing classical model is a random forest with 1,000 trees, but even that is performing at only 34.44 percent—far too low an overall accuracy to make the model one we'd care to use in most cases. The dataset is not a simple one, at least not for old-school approaches. Somewhat surprisingly, the Gaussian naive Bayes model is right 28 percent of the time. Recall that the Gaussian naive Bayes expects the samples to be independent of one another. Here the independence assumption between the sound samples for a particular test input is not valid. The feature vector, in this case, represents a signal evolving in time, not a collection of independent features.

The models that failed the most are nearest centroid, *k*-NN, and the linear SVMs. We have a reasonably high-dimensional input, 882 elements, but only 6,400 of them in the training set. That is likely too few samples for the nearest-neighbor classifiers to make use of; the feature space is too sparsely populated. Once again, the curse of dimensionality is rearing its ugly head. The linear SVM fails because the features seem not to be linearly separable. We did not try an RBF (Gaussian kernel) SVM, but we'll leave that as an

exercise for the reader. If you do try it, remember that there are now two hyperparameters to tune: C and γ.

With a Traditional Neural Network

We haven't yet tried a traditional neural network. We'll use the sklearn `MLPClassifier` class as we did before; Listing 12-8 has the code.

esc10_audio_mlp.py

```
from sklearn.neural_network import MLPClassifier
import numpy as np

num_classes = 10

x_train = np.load("../data/audio/ESC-10/esc10_raw_train_audio.npy")[:,:,0]
y_train = np.load("../data/audio/ESC-10/esc10_raw_train_labels.npy")
x_test  = np.load("../data/audio/ESC-10/esc10_raw_test_audio.npy")[:,:,0]
y_test  = np.load("../data/audio/ESC-10/esc10_raw_test_labels.npy")

x_train = (x_train.astype('float32') + 32768) / 65536
x_test = (x_test.astype('float32') + 32768) / 65536

model = MLPClassifier(hidden_layer_sizes=(512,128),
                      max_iter=200,
                      solver='lbfgs')
model.fit(x_train, y_train)
score = 100.0*model.score(x_test, y_test)
print("score = %0.2f%%" % score)
```

Listing 12-8: Using a traditional neural network

After loading the necessary modules, we load the data itself and scale it as we did for the classical models. Next, we build the model by using two hidden layers of 512 and 128 nodes, respectively, then train by calling `fit`.

The `MLPClassifier` constructor has a new keyword, `solver`, which is set to `lbfgs`. This is new. *Solver* is sklearn-speak for an optimizer. If not specified, it defaults to Adam. Here, I've set the solver to use the L-BFSG algorithm instead. The training set is small, 6,400 vectors, and that isn't enough to get a traditional neural network to learn anything when using SGD or Adam. In those cases, the final test-set accuracy is 10 percent, pure random guessing.

The label `lbfgs` stands for *limited-memory BFGS*, and *BFGS* stands for *Broyden–Fletcher–Goldfarb–Shanno*. Standard machine learning optimizers, as we know, use the gradient, which is the multivariable version of the derivative; think of it as the slope of a curve at a point. Optimization algorithms based on the gradient alone are known as *first-order algorithms*, as discussed in Chapter 6. Both SGD and Adam are first-order algorithms.

Ideally, optimization makes use of what we might call the *gradient of the gradient*—that is, information about how the gradient itself is changing in the vicinity of a point. This is the second derivative, so algorithms using this information are known as *second-order* optimization algorithms. The BFGS

algorithm falls into this class. The "limited" part approximates the second-order information making the algorithm useful for high-dimensional spaces, like the 882-element vectors we're working with in this example. Therefore, we might expect the L-BFGS algorithm to do a better job than SGD or Adam. Unfortunately, our expectation, while true, isn't anything to brag about; even with L-BFGS, the best we achieve is 19.1 percent accuracy, placing the traditional neural network near the bottom of the pack of models tested so far.

With a Convolutional Neural Network

Classical models and the traditional neural network don't cut it. We shouldn't be too surprised, but giving them a try was easy. Let's move on and apply a 1D convolutional neural network to this dataset to see whether it performs any better.

We haven't worked with 1D CNNs yet. Besides the structure of the input data, the only difference is that we replace calls to `Conv2D` and `MaxPooling2D` with calls to `Conv1D` and `MaxPooling1D`.

Listing 12-9 shows the code for the first model we'll try.

```
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv1D, MaxPooling1D
import numpy as np

batch_size = 32
num_classes = 10
epochs = 16
nsamp = (882,1)
x_train = np.load("../data/audio/ESC-10/esc10_raw_train_audio.npy")
y_train = np.load("../data/audio/ESC-10/esc10_raw_train_labels.npy")
x_test  = np.load("../data/audio/ESC-10/esc10_raw_test_audio.npy")
y_test  = np.load("../data/audio/ESC-10/esc10_raw_test_labels.npy")
x_train = (x_train.astype('float32') + 32768) / 65536
x_test = (x_test.astype('float32') + 32768) / 65536
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
model = Sequential()
model.add(Conv1D(32, kernel_size=3, activation='relu',
                 input_shape=nsamp))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
```

```
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
history = model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test accuracy:', score[1])
```

Listing 12-9: A 1D CNN in Keras

This model loads and preprocesses the dataset as before. This architecture, which we'll call the *shallow* architecture, has a single convolutional layer of 32 filters with a kernel size of 3. We'll vary this kernel size in the same way we tried different 2D kernel sizes for the MNIST models. Following the `Conv1D` layer is a single max-pooling layer with a pool kernel size of 3. `Dropout` and `Flatten` layers come next, before a single `Dense` layer of 512 nodes with dropout. A softmax layer completes the architecture.

We'll train for 16 epochs by using a batch size of 32. We'll keep the training history so we can examine the losses and validation performance as a function of epoch. We have 1,600 test samples, and we use all of them to monitor (but not alter) training and for performance testing. Finally, we'll vary the `Conv1D` kernel size from 3 to 33 in an attempt to find one that works well with the training data.

Let's define four other architectures. We'll refer to them as *medium*, *deep0*, *deep1*, and *deep2*. With no prior experience working with this data, trying multiple architectures makes sense. At present, there's no way to know the best architecture for a new dataset ahead of time. All we have is our previous experience.

Listing 12-10 lists the specific architectures, separated by comments.

```
# medium
model = Sequential()
model.add(Conv1D(32, kernel_size=3, activation='relu',
                 input_shape=nsamp))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

# deep0
model = Sequential()
model.add(Conv1D(32, kernel_size=3, activation='relu',
                 input_shape=nsamp))
```

```
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

# deep1
model = Sequential()
model.add(Conv1D(32, kernel_size=3, activation='relu',
                 input_shape=nsamp))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

# deep2
model = Sequential()
model.add(Conv1D(32, kernel_size=3, activation='relu',
                 input_shape=nsamp))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
```

```
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Listing 12-10: Various 1D CNN architectures

The files *esc10_audio_cnn_shallow.py* through *esc10_audio_cnn_deep2.py* contain the code. Each file expects an integer argument on the command line that it uses as the kernel size for the first convolutional layer. Combining this flexibility with the statements in the *train_all_1d_cnn* script trains and tests each combination of architecture and kernel size to produce the results in Table 12-1. The best-performing model for each architecture is bolded.

Table 12-1: Test-Set Accuracies by Convolutional Kernel Size and Model Architecture

Kernel size	Shallow	Medium	Deep0	Deep1	Deep2
3	**41.63**	38.75	**47.06**	10.00	10.00
5	38.13	**40.44**	46.82	10.00	10.00
7	38.19	38.69	44.69	**50.50**	**50.44**
9	40.13	38.25	42.94	49.19	50.19
11	38.00	39.44	43.19	46.88	10.00
13	37.00	37.06	45.44	49.38	49.63
15	35.25	40.38	41.13	10.00	10.00
33	32.81	36.81	43.06	46.50	10.00

Looking at Table 12-1, we see a general trend of accuracy improving as the model depth increases. However, at the deep2 model, things start to fall apart. Some of the models fail to converge, showing an accuracy equivalent to random guessing. The deep1 model is the best performing for larger kernel sizes. When looking across by kernel size, there is no clear winner among the five architectures, but deep1's highest overall accuracy for a kernel width of 7 implies this might be a good combination.

We trained the deep1 with kernel width 7 architecture for only 16 epochs. Will accuracy improve if we train for more? Let's train the deep1 model for 60 epochs and plot the training and validation loss and error to see how they converge (or don't). Doing this produces Figure 12-2, which shows the training and validation loss and error as a function of epoch.

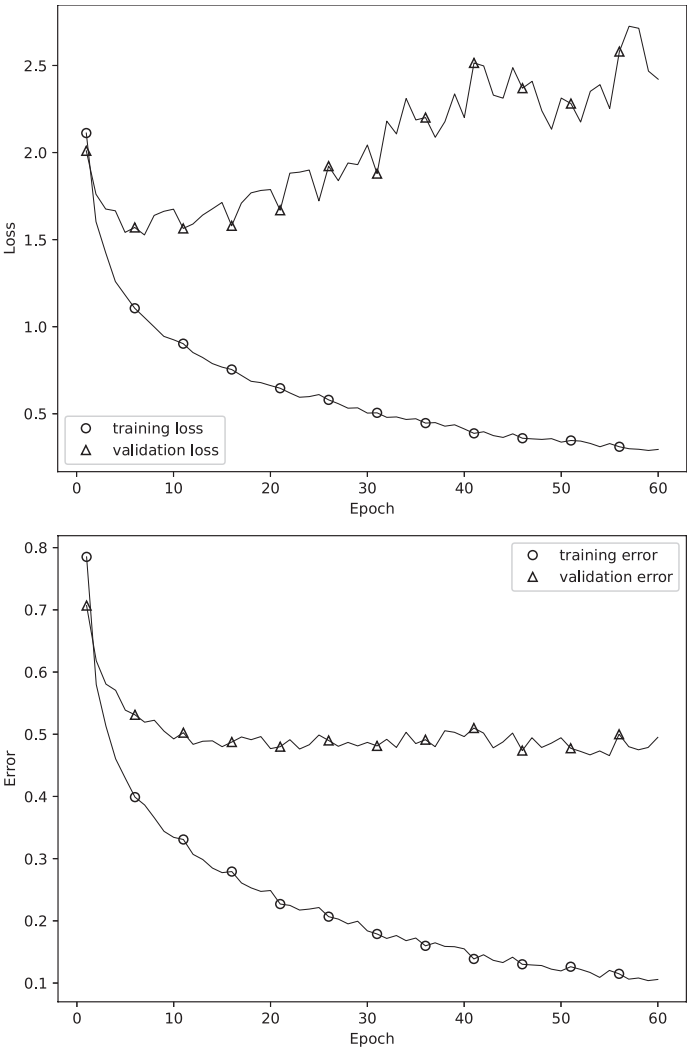


Figure 12-2: The training and validation loss (top) and error (bottom) for the deep1 architecture

Immediately, we should pick up on the explosion of the loss for the validation set. The validation loss is continually decreasing until after about epoch 15 or so; then it goes up and becomes oscillatory. This is a clear example of overfitting. The likely source of this overfitting is our limited training set size, only 6,400 samples, even after data augmentation. The validation error remains more or less constant after initially decreasing. The conclusion is that we cannot expect to do much better than an overall accuracy of about 50 percent for this dataset using 1D vectors.

If we want to improve, we need to be more expressive with our dataset. Fortunately for us, we have another preprocessing trick up our sleeves.

Spectrograms

Let's return to our augmented set of audio files. To build the dataset, we took the sound samples, keeping only two seconds' worth and only every 100th sample. The best we could do is an accuracy of a little more than 50 percent.

However, if we work with a small set of sound samples from an input audio file, say 200 milliseconds' worth, we can use the vector of samples to calculate the Fourier transform. The *Fourier transform* of a signal measured at regular intervals tells us the frequencies that went into building the signal. Any signal can be thought of as the sum of many sine and cosine waves. If the signal is composed of only a few waves, like the sound you might get from an instrument such as the ocarina, then the Fourier transform will have essentially a few peaks at those frequencies. If the signal is complex, like speech or music, then the Fourier transform will have many frequencies, leading to many peaks.

The Fourier transform itself is complex-valued: each element has both a real and an imaginary component. You can write it as $a + bi$, where a and b are real numbers and $i = \sqrt{-1}$. If we use the absolute value of these quantities, we'll get a real number representing the energy of a particular frequency. This is called the *power spectrum* of the signal. A simple tone might have energy in only a few frequencies, while something like a cymbal crash or white noise will have energy more or less evenly distributed among all frequencies. Figure 12-3 shows two power spectra.

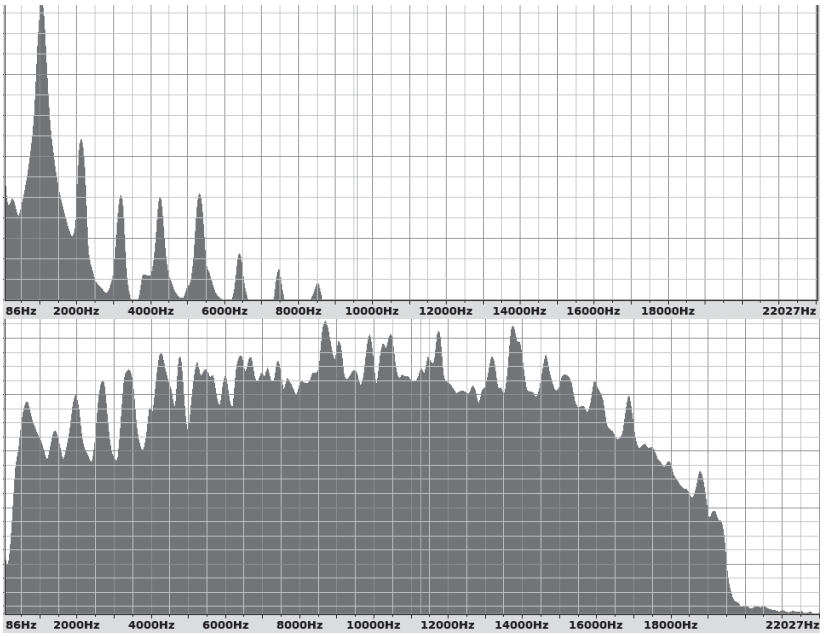


Figure 12-3: The power spectrum of an ocarina (top) and a cymbal (bottom)

On the top is the spectrum of an ocarina, and on the bottom is a cymbal crash. As expected, the ocarina has energy in only a few frequencies, while the cymbal uses all the frequencies. The important point for us is that *visually* the spectra are quite different from each other. (The spectra were made with Audacity, an excellent open source audio-processing tool.)

We could use these power spectra as feature vectors, but they represent only the spectra of tiny slices of time. The sound samples are five seconds long. Instead of using a spectrum, we will use a *spectrogram*. This is an image made up of columns that represent individual spectra. The x-axis represents time, and the y-axis represents frequency. The color of a pixel is proportional to the energy in that frequency at that time.

In other words, a spectrogram is what we get if we orient the power spectra vertically and use color to represent intensity at a given frequency. With this approach, we can turn an entire sound sample into an image. For example, Figure 12-4 shows the spectrogram of a crying baby. Compare this to the feature vector of Figure 12-1.

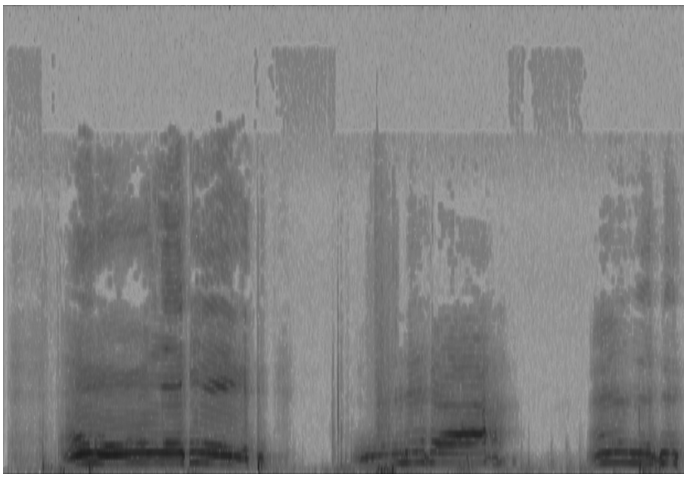


Figure 12-4: The spectrogram of a crying baby

To create spectrograms of the augmented audio files, we need a new tool and a bit of code. The tool we need is called `sox`. It's not a Python library, but a command line tool. Odds are that it is already installed if you're using our canonical Ubuntu Linux distribution. If not, you can install it:

```
$ sudo apt-get install sox
```

We'll use `sox` from inside a Python script to produce the spectrogram images we want. Each sound file becomes a new spectrogram image.

The source code to process the training images is in Listing 12-11 (see *make_augmented_spectrograms.py*).

```
import os
import numpy as np
from PIL import Image
```

```
rows = 100
cols = 160
flist = [i[:-1] for i in open("../data/audio/ESC-10/augmented_train_filelist.txt")] ❶
N = len(flist)
img = np.zeros((N,rows,cols,3), dtype="uint8")
lbl = np.zeros(N, dtype="uint8")
p = []

for i,f in enumerate(flist):
    src, c = f.split()
    os.system("sox %s -n spectrogram" % src) ❷
    im = np.array(Image.open("spectrogram.png").convert("RGB"))
    im = im[42:542,58:858,:] ❸
    im = Image.fromarray(im).resize((cols,rows))
    img[i,:,:,:] = np.array(im)
    lbl[i] = int(c)
    p.append(os.path.abspath(src))

os.system("rm -rf spectrogram.png")
p = np.array(p)
idx = np.argsort(np.random.random(N)) ❹
img = img[idx]
lbl = lbl[idx]
p = p[idx]
np.save("../data/audio/ESC-10/esc10_spect_train_images.npy", img)
np.save("../data/audio/ESC-10/esc10_spect_train_labels.npy", lbl)
np.save("../data/audio/ESC-10/esc10_spect_train_paths.npy", p)
```

Listing 12-11: Building the spectrograms

We start by defining the size of the spectrogram. This is the input to our model, and we don't want it to be too big because we're limited in the size of the inputs we can process. We'll settle for 100×160 pixels. We then load the training file list ❶ and create NumPy arrays to hold the spectrogram images and associated labels. The list `p` will hold the pathname of the source for each spectrogram in case we want to get back to the original sound file at some point. In general, it's a good idea to preserve information to get back to the source of derived datasets.

Then we loop over the file list. We get the filename and class label and then call `sox`, passing in the source sound filename ❷. The `sox` application is sophisticated. The syntax here turns the given sound file into a spectrogram image with the name *spectrogram.png*. We immediately load the output spectrogram into `im`, making sure it's an RGB file with no transparency layer (hence the call to `convert("RGB")`).

The spectrogram created by `sox` has a border with frequency and time information. We want only the spectrogram image portion, so we subset the image ❸. We determined the indices we're using empirically. It's possi-

ble, but somewhat unlikely, that a newer version of `sox` will require tweaking these to avoid including any border pixels.

Next, we resize the spectrogram so that it fits in our 100×160-pixel array. This is downsampling, true, but hopefully enough characteristic information is still present to allow a model to learn the difference between classes. We keep the downsampled spectrogram and the associated class label and sound filepath.

When we've generated all the spectrograms, the loop ends, and we remove the final extraneous spectrogram PNG file. We convert the list of sound filepaths to a NumPy array so we can store it in the same manner as the images and labels. Finally, we randomize the order of the images as a precaution against any implicit sorting that might group classes ❹. This is so that minibatches extracted sequentially are representative of the mix of classes as a whole. To conclude, we write the images, labels, and pathnames to disk. We repeat this entire process for the test set.

Are we able to visually tell the difference between the spectrograms of different classes? If we can do that easily, we have a good shot of getting a model to tell the difference too. Figure 12-5 shows 10 spectrograms of the same class in each row.

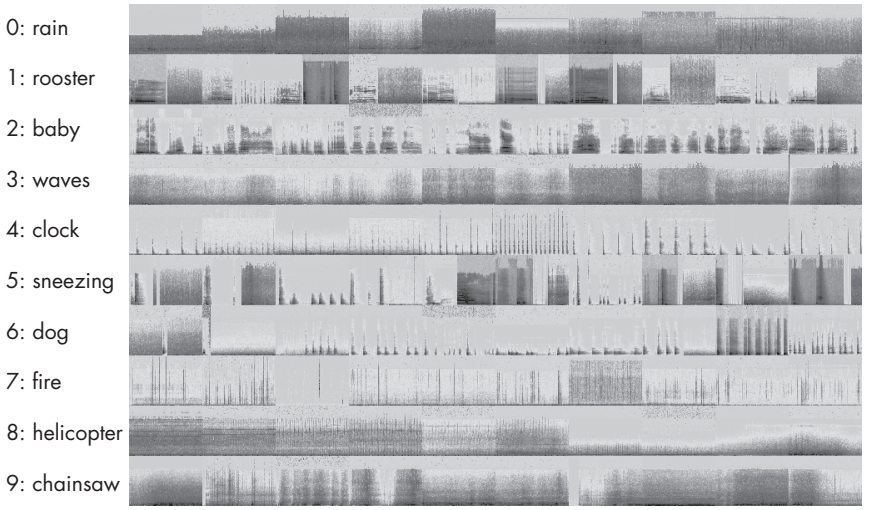


Figure 12-5: The sample spectrograms for each class in ESC-10. Each row shows 10 examples from the same class.

Visually, we can usually tell the spectra apart, which is encouraging. With our spectrograms in hand, we're ready to try some 2D CNNs to see if they do better than the 1D CNNs.

Classifying Spectrograms

To work with the spectrogram dataset, we need 2D CNNs. A possible starting point is to convert the shallow 1D CNN architecture to 2D by changing `Conv1D` to `Conv2D`, and `MaxPooling1D` to `MaxPooling2D`. However, if we do this, the

resulting model has 30.7 million parameters, which is many more than we want to work with. Instead, let's opt for a deeper architecture that has fewer parameters and then explore the effect of different kernel sizes for the first convolutional layer. The code is in Listing 12-12.

esc10_cnn_deep.py

```
import sys
import pickle
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
import numpy as np

#  kernel size -- 3 or 7
z = int(sys.argv[1])
batch_size = 16
num_classes = 10
epochs = 16
img_rows, img_cols = 100, 160
input_shape = (img_rows, img_cols, 3)

x_train = np.load("../data/audio/ESC-10/esc10_spect_train_images.npy")
y_train = np.load("../data/audio/ESC-10/esc10_spect_train_labels.npy")
x_test = np.load("../data/audio/ESC-10/esc10_spect_test_images.npy")
y_test = np.load("../data/audio/ESC-10/esc10_spect_test_labels.npy")

x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, kernel_size=(z, z),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
```

```
              metrics=['accuracy'])
history = model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=0,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('%dx%d: accuracy: %0.3f' % (z,z,100.0*score[1]))
```

Listing 12-12: Classifying spectrograms

We're using a minibatch size of 16 for 16 epochs along with the Adam optimizer. The model architecture has two convolutional layers, a max-pooling layer with dropout, another convolutional layer, and a second max-pooling layer with dropout. There is a single dense layer of 128 nodes before the softmax output.

We'll test two kernel sizes for the first convolutional layer: 3×3 and 7×7. The kernel size is expected on the command line.

All the initial 1D convolutional runs used a single training of the model for evaluation. Because of random initialization, we'll get slightly different results from training to training, even if nothing else changes. For the 2D CNNs, let's train each model six times and present the overall accuracy as a mean $\pm$ standard error of the mean. Doing just this gives us the following overall accuracies:

Kernel size	Score
3×3	$75.54 \pm 0.92\%$
7×7	$76.18 \pm 0.85\%$

The standard error ranges overlap, indicating this experiment finds no meaningful difference between using a 3×3 initial convolutional layer kernel size or a 7×7. Therefore, we'll stick with 3×3 going forward.

As an aside, just because the results from six runs of each kernel size aren't statistically significantly different doesn't mean there is, truly, no difference between 3×3 and 7×7 kernels. There might be a true difference, but the effect size is small enough that six trials isn't able to reveal it. What might we get with 100 trials of each kernel size? If those results are statistically significantly different, with a p-value well below 0.05, then we might come to believe one kernel size is superior to the other, but my money is on such a difference having a small Cohen's effect size (d), making the difference real but meaningless in practice. The "it's real but effectively meaningless in practice" scenario is more common than we might think.

Figure 12-6 shows the training and validation loss (top) and error (bottom) for one run of the 2D CNN trained on the spectrograms. As we saw in the 1D CNN case, after only a few epochs, the validation error starts to increase, but not as dramatically.

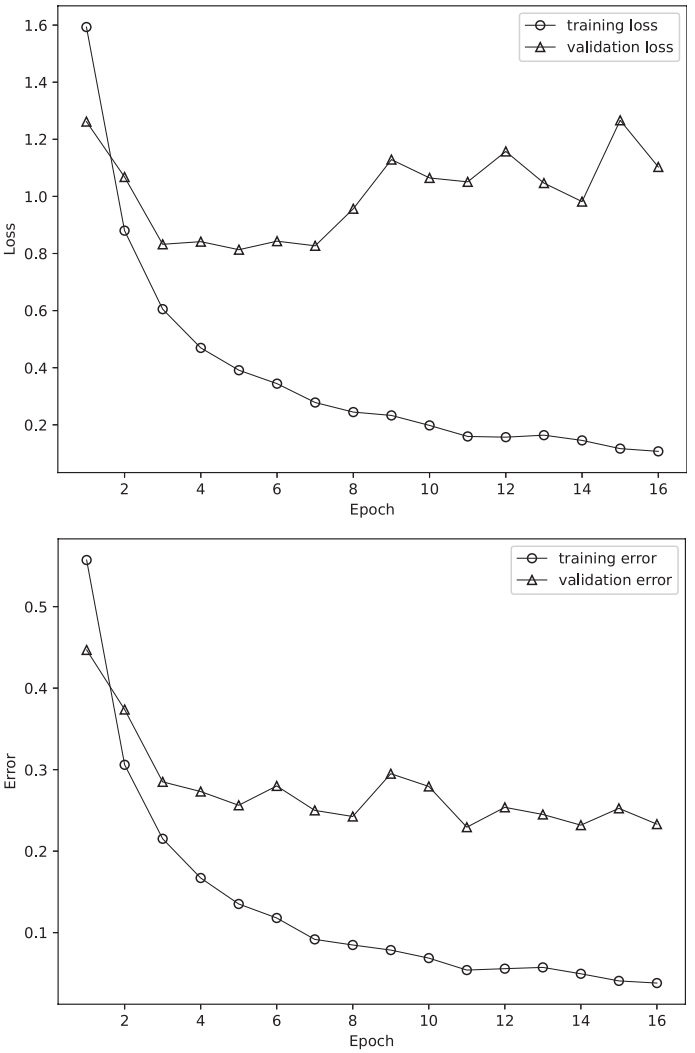


Figure 12-6: The training and validation loss (top) and error (bottom) for the 2D CNN architecture

The 2D CNN performs significantly better than the 1D CNN: 75 percent accuracy versus only 50 percent. This level of accuracy is still not particularly useful for many applications, but for others, it might be completely acceptable. Nevertheless, we'd like to do better if we can.

Let's take a quick look at the confusion matrix generated by one of the models using our chosen architecture. We've seen previously how to calculate the matrix; we'll show it here for discussion and for comparison with the confusion matrices we'll make in the next section. To generate it, run *stats.py* passing in a trained model (a *.keras* file).

Table 12-2 shows the matrix; as always, rows are the true class label, and columns are the model-assigned label. The elements have been scaled by the number of examples per class so that the entries in the matrix represent

percentages. For example, 93.8 percent of class 1 examples were correctly labeled by the model.

Table 12-2: The Confusion Matrix for the Spectrogram Model

Class	0	1	2	3	4	5	6	7	8	9
0	**67.5**	0.0	0.0	27.5	0.0	0.0	0.0	0.0	0.6	4.4
1	0.0	**93.8**	5.0	0.0	0.0	0.6	0.6	0.0	0.0	0.0
2	0.0	9.4	**79.4**	0.0	0.6	0.0	0.0	0.0	0.0	10.6
3	13.8	0.0	0.0	**68.8**	0.0	0.0	0.0	1.2	3.1	13.1
4	0.6	0.0	0.0	0.0	**83.1**	0.0	0.0	16.2	0.0	0.0
5	0.0	1.9	4.4	0.0	13.1	**60.6**	16.2	3.8	0.0	0.0
6	0.0	3.8	2.5	0.0	8.1	1.2	**82.5**	1.9	0.0	0.0
7	0.0	0.0	0.0	0.0	2.5	0.0	0.0	**89.4**	8.1	0.0
8	11.2	0.0	0.0	2.5	1.2	0.0	0.0	6.9	**77.5**	0.6
9	16.9	0.0	0.6	5.6	0.0	0.0	0.0	0.0	8.1	**68.8**

The two worst-performing classes are sneezing (5) and rain (0). Rain is most often confused with waves (3), which seems reasonable. Sneezing is roughly equally confused with a clock (4) or dog barking (6), with the edge going to the dogs. I can understand dogs, maybe, but confusing a sneeze for a ticking clock is a bit odd. Consider it an example of the mysterious ways of deep learning models. The two best-performing classes are a rooster (1) and a crackling fire (7).

Does this mean we're stuck at 76 percent accuracy? No, we have one more trick to try. We've been training and evaluating the performance of single models, but nothing is stopping us from training multiple models and combining their results. This is *ensembling*. We presented ensembles briefly in Chapter 3 and again in Chapter 6 when discussing dropout. Now, let's use the idea directly to see if we can improve our sound sample classifier.

Ensembles

The core idea of an ensemble is to take the output of multiple models trained on the same, or extremely similar, dataset(s) and combine them. It embodies the "wisdom of the crowds" concept: one model might be better at certain classes or types of inputs for a particular class than another, so it follows that if they work together, they might arrive at a final result better than either one could do on its own.

We'll use the same machine learning architecture we used in the previous section. Our different models will be separate trainings of this architecture, using the spectrograms as input. This is a weaker form of ensembling. Typically, the models in the ensemble are quite different from one another, either different architectures of neural networks, or completely different types of models like random forests and k-nearest neighbors. Of course, as we now understand, random forests are themselves ensembles of decision

trees. The variation between models here is due to the random initialization of the networks and the different parts of the loss landscape the network finds itself in when training stops. Our approach works like this:

1. Train multiple models ($n = 6$) by using the spectrogram dataset.
2. Combine the softmax output of these models on the test set in some manner.
3. Use the resulting output from the combination to predict the assigned class label.

We hope that the set of class labels assigned after combining the individual model outputs is superior to the set assigned by the model architecture used alone. Intuitively, we feel that this approach should buy us something. It makes sense.

However, a question immediately arises: how do we best combine the outputs of the individual networks? We have total freedom in the answer to that question. What we are looking for is an $f()$ such that:

$$y_{\text{predict}} = f(y_0, y_1, y_2, \ldots, y_n)$$

Here, $y_i, i = 0, 1, \ldots, n$ are the outputs of the n models in the ensemble, and $f()$ is a function, operation, or algorithm that best combines them into a single new prediction, y_{predict}.

Some combination approaches come readily to mind: we could average the outputs and select the largest, keep maximum per class output across the ensemble and then choose the largest of those, or use voting to decide which class label should be assigned.

We need the predictions of each model on the test set. To get those, run *train_six_cnn*, which is nothing more than a script that executes *esc10_cnn_deep.py* six times by using an additional command line argument to save the test-set predictions in a NumPy file:

```
python3 esc10_cnn_deep.py 3 prob_run0.npy 2>/dev/null
python3 esc10_cnn_deep.py 3 prob_run1.npy 2>/dev/null
python3 esc10_cnn_deep.py 3 prob_run2.npy 2>/dev/null
python3 esc10_cnn_deep.py 3 prob_run3.npy 2>/dev/null
python3 esc10_cnn_deep.py 3 prob_run4.npy 2>/dev/null
python3 esc10_cnn_deep.py 3 prob_run5.npy 2>/dev/null
```

Appending `2>/dev/null` to the command line directs TensorFlow output to the null device to prevent the output from cluttering the screen.

The script takes a little more than four hours to run on my test machine. The per model accuracy on the test set is displayed when each training run finishes. My run produced the following individual model accuracies:

```
model 0: 76.688%
model 1: 76.375%
model 2: 78.875%
model 3: 78.000%
```

```
model 4: 75.937%
model 5: 72.188%
```

These accuracies are in line with the results from earlier in the chapter, but, clearly, not all these models were created (trained) equal. Model 2 is the most accurate at 78.9 percent, while model 5 is rather poor, producing an accuracy of only 72.2 percent. Recall that the only difference between model 2 and model 5 is the random weight initialization—something worth remembering when a paper claims state-of-the-art accuracies yet provides no uncertainty estimates.

The file *ensemble.py* loads the six test-set prediction files, *prob_run0.npy* through *prob_run5.npy*, along with the true test-set class labels and calculates the average predictions, the maximum predictions, and voting among the six models. Listing 12-13 contains the code.

```
import sys
import numpy as np

def Accuracy(y,p):
    cm = np.zeros((10,10))
    for i in range(len(y)): cm[y[i],p[i]] += 1
    return 100.0*np.diag(cm).sum()/cm.sum()

p0 = np.load("prob_run0.npy"); p1 = np.load("prob_run1.npy")
p2 = np.load("prob_run2.npy"); p3 = np.load("prob_run3.npy")
p4 = np.load("prob_run4.npy"); p5 = np.load("prob_run5.npy")
y = np.load("../data/audio/ESC-10/esc10_spect_test_labels.npy")

prob = (p0+p1+p2+p3+p4+p5)/6.0
p = np.argmax(prob, axis=1)
print("Accuracy (average) = %0.2f%%" % Accuracy(y,p))

p = np.zeros(len(y), dtype="uint8")
for i in range(len(y)):
    t = np.array([p0[i],p1[i],p2[i],p3[i],p4[i],p5[i]])
    p[i] = np.argmax(t.reshape(60)) % 10
print("Accuracy (maximum) = %0.2f%%" % Accuracy(y,p))

t = np.zeros((6,len(y)), dtype="uint32")
t[0,:] = np.argmax(p0, axis=1); t[1,:] = np.argmax(p1, axis=1)
t[2,:] = np.argmax(p2, axis=1); t[3,:] = np.argmax(p3, axis=1)
t[4,:] = np.argmax(p4, axis=1); t[5,:] = np.argmax(p5, axis=1)
p = np.zeros(len(y), dtype="uint8")
for i in range(len(y)):
    q = np.bincount(t[:,i])
    p[i] = np.argmax(q)
print("Accuracy (voting)  = %0.2f%%" % Accuracy(y,p))
```

Listing 12-13: Ensembling the test-set predictions

The `Accuracy` function returns the overall accuracy by first creating the confusion matrix and then calculating the accuracy from it. The next code paragraph loads the six sets of test-set predictions. Remember, the predictions are softmax outputs, a set of 1,600 vectors of 10 elements stored in a $1{,}600 \times 10$ matrix. Finally, `y` holds the true class labels.

The final three code paragraphs calculate the average, maximum, and voting predicted class labels before printing the accuracy. The average is straightforward: sum the predictions and divide by 6 before using `np.argmax` to assign a class label. The maximum approach uses a loop to first form a vector of 60 elements for each prediction by appending the six softmax vectors. Then, `np.argmax` returns the index of the maximum, which modulo 10 (for the 10 classes) returns a class label corresponding to the largest value in the extended vector. Finally, voting uses `np.bincount` with `np.argmax` to select the class label that appears most often among the six models.

My run of *ensemble.py* produced the following:

```
Accuracy (average) = 78.88%
Accuracy (maximum) = 78.81%
Accuracy (voting)  = 78.50%
```

All of these are an improvement over an individual model, with averaging the softmax predictions providing the best performance on the test set, 78.9 percent.

In most cases, we'd stop here, claim victory on this assignment, and present our boss with the ensemble model. However, we're not that easily satisfied. Notice that simple averaging produced the best result from the ensemble approach. Why should we use simple averaging when know that the six models did not perform equally on the test set? Some were significantly better than the others. Might it be that simple averaging gives the poorer models too much say in the outcome?

Let's mix things up by using a *weighted average* of the models instead. That means, effectively, replacing

```
prob = (p0+p1+p2+p3+p4+p5)/6.0
```

with something like the following for scalar weights `w0` through `w5` that reflect the importance or belief we place on each individual model's quality:

```
prob = w0*p0 + w1*p1 + w2*p2 + w3*p3 + w4*p4 + w5*p5
```

Great! Weighted averaging is a small tweak to the existing approach, but we have a small problem: how do we pick the weights?

A first thought might be to build them from the individual model accuracies. After all, we have them, and they represent a relative measure of each model's quality. To turn them into weights, we divide each accuracy by the sum of all six, then use those weights to calculate the class label. This helps a tiny bit and produces an ensemble accuracy of 79.1 percent.

It's intuitively reasonable to make the weights based on the performance of the individual models, but that isn't likely to be ideal. It's an ad hoc substitute for doing what machine learning is always doing: learning the necessary parameter values. In other words, let's learn what the best set of weights is to maximize the ensemble's performance on the test set in the hopes that the weighting reflects true learning and will work equally well when the ensemble is used in the wild.

The next question is, of course, how do we learn the weights? Whenever I am presented with a situation where I'm looking for the best set of something, I think "optimization"—in this case, swarm intelligence optimization. As a topic, swarm intelligence and related evolutionary algorithms are far beyond what we can do justice to here. The short description is that *swarm intelligence* uses a collection of agents that move, somewhat intelligently, through the search space. Here, the search space is 6D, and each dimension corresponds to one of the weights we want to find. The best set of weights then becomes a best position in this 6D space. The swarm will locate this point (we hope), and when it does, we'll have the weights that give us the best possible ensemble.

The swarm search code is in *swarm_ensemble.py*. We won't walk through it, but the problem-specific code begins on line 1370. To learn more about swarm techniques, I recommend my book *The Art of Randomness: Randomized Algorithms in the Real World* (No Starch Press, 2024). Running *swarm_ensemble.py* tells us how to use it:

```
swarm_ensemble <npart> <niter> <alg>

  <npart>  - number of particles
  <niter>  - number of iterations
  <alg>    - PSO,JAYA,DE,RO,GWO,GA
```

Three arguments are required. The code is already configured to load the *prob_run0.npy* files along with the true test-set labels. The first argument is the size of the swarm. Good values are in the vicinity of 20. The second argument is the number of iterations, meaning the number of times the swarm updates it position in the search space. Empirically, for this experiment, a few hundred iterations is sufficient.

The final argument is the specific swarm algorithm to use. There are hundreds in the literature, but only six are available here: bare-bones particle swarm optimization (PSO), Jaya, differential evolution (DE), random optimization (RO), Grey Wolf Optimizer (GWO), and a genetic algorithm variant (GA). I recommend experimenting with all of them. Random optimization is the simplest algorithm and generally isn't the best choice. My experimenting indicates that Jaya, differential evolution, and bare-bones PSO work well. For example, a run using Jaya produced the following:

```
> python3 swarm_ensemble.py 20 500 Jaya
0001: accuracy = 80.00000000
0100: accuracy = 81.00000000
```

```
0200: accuracy = 81.00000000
0300: accuracy = 81.00000000
0400: accuracy = 81.06250000
0500: accuracy = 81.06250000

Accuracy: 81.062500000

Weights:
 0: 0.120140320
 1: 0.139839471
 2: 0.244902847
 3: 0.281249365
 4: 0.212761744
 5: 0.001106253

(11 best updates, 10020 function calls, time: 11.262 seconds)
```

The output shows the evolution of the ensemble accuracy as the weight vector is improved by the swarm as it searches. The final position leads to an ensemble accuracy of just over 81 percent, which is about 2 percent better than simple averaging.

The weights per model are listed, indicating the relative importance placed on each model's predictions. Models 2 and 3 are given the greatest weight. These were the best-performing models individually, so this weighting makes sense. Model 5 is given a tiny weight, thereby greatly minimizing its contribution to the ensemble. Model 5 was the worst-performing individual model, so, again the search results are sensible; a bad model was in a sense detected and removed from the ensemble.

This exercise demonstrates that ensemble techniques are helpful, but at a cost: we needed to train multiple models and then use each of them to classify new samples. Because of this, at least for large models, ensembling is not used as often as it might be.

Summary

This chapter presented a case study, a new dataset, and the steps we need to take to work through building a useful model. We started by working with the dataset as given to us, as raw sound samples, which we were able to augment successfully. We noticed that we had a feature vector and attempted to use classical models. From there, we moved on to 1D CNNs. Neither of these approaches was particularly successful.

Fortunately for us, our dataset allowed for a new representation, one that illustrated more effectively what composed the data and, especially important for us, introduced spatial elements so that we could work with 2D convolutional networks. With these networks, we improved quite a bit on the best 1D results, but we were still not at a level that was likely to be useful.

We then moved to ensembles of classifiers. With these, we discovered a modest improvement by using simple approaches to combining the base model outputs (for example, voting).

We can show the progression of models and their overall accuracies to see how our case study evolved. Table 12-3 shows the power of modern deep learning and the utility of combining it with well-proven classical approaches like ensembles.

Table 12-3: Case Study Model Performance

Model	Data source	Accuracy
Gaussian naive Bayes	1D sound sample	28.1%
Random forest (1,000 trees)	1D sound sample	34.4%
1D CNN	1D sound sample	50.5%
2D CNN	Spectrogram	75.5%
Ensemble (simple average)	Spectrogram	78.9%
Ensemble (weighted average)	Spectrogram	81.1%

The next chapter continues our exploration of convolutional networks by introducing us to advanced CNN architectures, specific model architectures that have become staples, especially for computer vision tasks.