12. Training Your First CNN

Now that we've reviewed the fundamentals of Convolutional Neural Networks, we are ready to implement our first CNN using Python and Keras. We'll start the chapter with a quick review of Keras configurations you should keep in mind when constructing and training your own CNNs.

We'll then implement ShallowNet, which as the name suggests, is a very shallow CNN with only a single `CONV` layer. However, don't let the simplicity of this network fool you – as our results will demonstrate, ShallowNet is capable of obtaining higher classification accuracy on both CIFAR-10 and the Animals dataset than *any other method* we've reviewed thus far in this book.

12.1 Keras Configurations and Converting Images to Arrays

Before we can implement `ShallowNet`, we first need to review the `keras.json` configuration file and how the settings inside this file will influence how you implement your own CNNs. We'll also implement a second image preprocessor called `ImageToArrayPreprocessor` which accepts an input image and then converts it to a NumPy array that Keras can work with.

12.1.1 Understanding the keras.json Configuration File

The first time you import the Keras library into your Python shell/execute a Python script that imports Keras, behind the scenes Keras generates a `keras.json` file in your home directory. You can find this configuration file in `~/.keras/keras.json`.

Go ahead and open the file up now and take a look at its contents:

```
{
    "epsilon": 1e-07,
    "floatx": "float32",
    "image_data_format": "channels_last",
    "backend": "tensorflow"
}
```

You'll notice that this JSON-encoded dictionary has four keys and four corresponding values. The `epsilon` value is used in a variety of locations throughout the Keras library to prevent division by zero errors. The default value of `1e-07` is suitable and should not be changed. We then have the `floatx` value which defines the floating point precision – it is safe to leave this value at `float32`.

The final two configurations, `image_data_format` and `backend`, are *extremely important*. By default, the Keras library uses the *TensorFlow* numerical computation backend. We can also use the *Theano* backend simply by replacing `tensorflow` with `theano`.

You'll want to keep these backends in mind when *developing* your own deep learning networks and when you *deploy* them to other machines. Keras does a fantastic job abstracting the backend, allowing you to write deep learning code that is compatible with *either* backend (and surely more backends to come in the future), and for the most part, you'll find that both computational backends will give you the same result. If you find your results are inconsistent or your code is returning strange errors, check your backend first and make sure the setting is what you expect it to be.

Finally, we have the `image_data_format` which can accept two values: `channels_last` or `channels_first`. As we know from previous chapters in this book, images loaded via OpenCV are represented in `(rows, columns, channels)` ordering, which is what Keras calls `channels_last`, as the channels are the last dimension in the array.

Alternatively, we can set `image_data_format` to be `channels_first` where our input images are represented as `(channels, rows, columns)` – notice how the number of channels is the first dimension in the array.

Why the two settings? In the Theano community, users tended to use *channels first* ordering. However, when TensorFlow was released, their tutorials and examples used *channels last* ordering. This discrepancy caused a bit of a problem when using Keras with code compatible with Theano because it may not be compatible with TensorFlow depending on how the programmer built their network. Thus, Keras introduced a special function called `img_to_array` which accepts an input image and then orders the channels correctly based on the `image_data_format` setting.

In general, you can leave the `image_data_format` setting as `channels_last` and Keras will take care of the dimension ordering for you regardless of backend; however, I do want to call this situation to your attention just in case you are working with legacy Keras code and notice that a different image channel ordering is used.

12.1.2 The Image to Array Preprocessor

As I mentioned above, the Keras library provides the `img_to_array` function that accepts an input image and then properly orders the channels based on our `image_data_format` setting. We are going to wrap this function inside a new class named `ImageToArrayPreprocessor`. Creating a class with a special `preprocess` function, just like we did in Chapter 7 when creating the `SimplePreprocessor` to resize images, will allow us to create "chains" of preprocessors to efficiently prepare images for training and testing.

To create our image-to-array preprocessor, create a new file named `imagetoarraypreprocessor.py` inside the `preprocessing` sub-module of `pyimagesearch`:

```
|--- pyimagesearch
|    |--- __init__.py
|    |--- datasets
|    |    |--- __init__.py
|    |    |--- simpledatasetloader.py
|    |--- preprocessing
|    |    |--- __init__.py
|    |    |--- imagetoarraypreprocessor.py
|    |    |--- simplepreprocessor.py
```

From there, open the file and insert the following code:

```
1  # import the necessary packages
2  from keras.preprocessing.image import img_to_array
3
4  class ImageToArrayPreprocessor:
5       def __init__(self, dataFormat=None):
6            # store the image data format
7            self.dataFormat = dataFormat
8
9       def preprocess(self, image):
10           # apply the Keras utility function that correctly rearranges
11           # the dimensions of the image
12           return img_to_array(image, data_format=self.dataFormat)
```

Line 2 imports the `img_to_array` function from Keras.

We then define the constructor to our `ImageToArrayPreprocessor` class on **Lines 5-7**. The constructor accepts an optional parameter named `dataFormat`. This value defaults to `None`, which indicates that the setting inside `keras.json` should be used. We could also explicitly supply a `channels_first` or `channels_last` string, but it's best to let Keras choose which image dimension ordering to used based on the configuration file.

Finally, we have the `preprocess` function on **Lines 9-12**. This method:

1. Accepts an `image` as input.
2. Calls `img_to_array` on the `image`, ordering the channels based on our configuration file/the value of `dataFormat`.
3. Returns a new NumPy array with the channels properly ordered.

The benefit of defining a *class* to handle this type of image preprocessing rather than simply calling `img_to_array` on every single image is that we can now *chain* preprocessors together as we load datasets from disk.

For example, let's suppose we wished to resize all input images to a fixed size of 32×32 pixels. To accomplish this, we would need to initialize our `SimpleProcessor` from Chapter 7:

```
sp = SimplePreprocessor(32, 32)
```

After the image is resized, we then need to apply the proper channel ordering – this can be accomplished using our `ImageToArrayPreprocessor` above:

```
iap = ImageToArrayPreprocessor()
```

Now, suppose we wished to load an image dataset from disk and prepare all images in the dataset for training. Using the `SimpleDatasetLoader` from Chapter 7, our task becomes very easy:

```
sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
(data, labels) = sdl.load(imagePaths, verbose=500)
```

Notice how our image preprocessors are *chained* together and will be applied in *sequential order*. For every image in our dataset, we'll first apply the `SimplePreprocessor` to resize it to

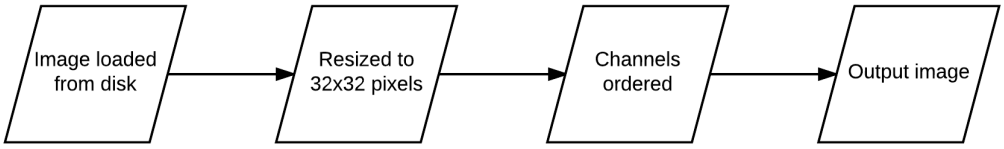

Figure 12.1: An example image pre-processing pipeline that (1) loads an image from disk, (2) resizes it to 32×32 pixels, (3) orders the channel dimensions, and (4) outputs the image.

32×32 pixels. Once the image is resized, the `ImageToArrayPreprocessor` is applied to handle ordering the channels of the image. This image processing pipeline can be visualized in Figure 12.1.

Chaining simple preprocessors together in this manner, where each preprocessor is responsible for *one, small job*, is an easy way to build an extendable deep learning library dedicated to classifying images. We'll make use of these preprocessors in the next section as well as define more advanced preprocessors in both the *Practitioner Bundle* and *ImageNet Bundle*.

12.2 ShallowNet

Inside this section, we'll implement the ShallowNet architecture. As the name suggests, the ShallowNet architecture contains only a few layers – the entire network architecture can be summarized as: `INPUT => CONV => RELU => FC`

This simple network architecture will allow us to get our feet wet implementing Convolutional Neural Networks using the Keras library. After implementing ShallowNet, I'll apply it to the Animals and CIFAR-10 datasets. As our results will demonstrate, CNNs are able to *dramatically outperform* the previous image classification methods discussed in this book.

12.2.1 Implementing ShallowNet

To keep our `pyimagesearch` package tidy, let's create a new sub-module inside `nn` named `conv` where all our CNN implementations will live:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- datasets
|    |--- nn
|    |    |--- __init__.py
...
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- shallownet.py
|    |--- preprocessing
```

Inside the `conv` sub-module, create a new file named `shallownet.py` to store our ShallowNet architecture implementation. From there, open up the file and insert the following code:

```
1 # import the necessary packages
2 from keras.models import Sequential
3 from keras.layers.convolutional import Conv2D
```

```
4 from keras.layers.core import Activation
5 from keras.layers.core import Flatten
6 from keras.layers.core import Dense
7 from keras import backend as K
```

Lines 2-7 import our required Python packages. The `Conv2D` class is the Keras implementation of the convolutional layer discussed in Section 11.1. We then have the `Activation` class, which as the name suggests, handles applying an activation function to an input. The `Flatten` classes takes our multi-dimensional volume and "flattens" it into a 1D array prior to feeding the inputs into the `Dense` (i.e, fully-connected) layers.

When implementing network architectures, I prefer to define them inside a class to keep the code organized – we'll do the same here:

```
9  class ShallowNet:
10     @staticmethod
11     def build(width, height, depth, classes):
12         # initialize the model along with the input shape to be
13         # "channels last"
14         model = Sequential()
15         inputShape = (height, width, depth)
16
17         # if we are using "channels first", update the input shape
18         if K.image_data_format() == "channels_first":
19             inputShape = (depth, height, width)
```

On **Line 9** we define the `ShallowNet` class and then define a `build` method on **Line 11**. Every CNN that we implement inside this book will have a `build` method – this function will accept a number of parameters, construct the network architecture, and then return it to the calling function. In this case, our `build` method requires four parameters:

- `width`: The width of the input images that will be used to train the network (i.e., number of columns in the matrix).
- `height`: The height of our input images (i.e., the number of rows in the matrix)
- `depth`: The number of channels in the input image.
- `classes`: The total number of classes that our network should learn to predict. For Animals, `classes=3` and for CIFAR-10, `classes=10`.

We then initialize the `inputShape` to the network on **Line 15** assuming "channels last" ordering. **Line 18 and 19** make a check to see if the Keras backend is set to "channels first", and if so, we update the `inputShape`. It's common practice to include **Lines 15-19** for nearly every CNN that you build, thereby ensuring that your network will work regardless of how a user is ordering the channels of their image.

Now that our `inputShape` is defined, we can start to build the ShallowNet architecture:

```
21         # define the first (and only) CONV => RELU layer
22         model.add(Conv2D(32, (3, 3), padding="same",
23             input_shape=inputShape))
24         model.add(Activation("relu"))
```

On **Line 22** we define the first (and only) convolutional layer. This layer will have 32 filters (K) each of which are 3×3 (i.e., square $F \times F$ filters). We'll apply `same` padding to ensure the size of output of the convolution operation matches the input (using `same` padding isn't strictly necessary

for this example, but it's a good habit to start forming now). After the convolution we apply an ReLU activation on **Line 24**.

Let's finish building ShallowNet:

```
26          # softmax classifier
27          model.add(Flatten())
28          model.add(Dense(classes))
29          model.add(Activation("softmax"))
30
31          # return the constructed network architecture
32          return model
```

In order to apply our fully-connected layer, we first need to flatten the multi-dimensional representation into a 1D list. The flattening operation is handled by the `Flatten` call on **Line 27**. Then, a `Dense` layer is created using the same number of nodes as our output class labels (**Line 28**). **Line 29** applies a softmax activation function which will give us the class label probabilities for each class. The ShallowNet architecture is returned to the calling function on **Line 32**.

Now that ShallowNet has been defined, we can move on to creating the actual "driver scripts" used to load a dataset, preprocess it, and then train the network. We'll look at two examples that leverage ShallowNet – Animals and CIFAR-10.

12.2.2 ShallowNet on Animals

To train ShallowNet on the Animals dataset, we need to create a separate Python file. Open up your favorite IDE, create a new file named `shallownet_animals.py`, ensuring that it is in the same directory level as our `pyimagesearch` module (or you have added `pyimagesearch` to the list of paths your Python interpreter/IDE will check when running a script).

From there, we can get to work:

```
1  # import the necessary packages
2  from sklearn.preprocessing import LabelBinarizer
3  from sklearn.model_selection import train_test_split
4  from sklearn.metrics import classification_report
5  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
6  from pyimagesearch.preprocessing import SimplePreprocessor
7  from pyimagesearch.datasets import SimpleDatasetLoader
8  from pyimagesearch.nn.conv import ShallowNet
9  from keras.optimizers import SGD
10 from imutils import paths
11 import matplotlib.pyplot as plt
12 import numpy as np
13 import argparse
```

Lines 2-13 import our required Python packages. Most of these imports you've seen from previous examples, but I do want to call your attention to **Lines 5-7** where we import our `ImageToArrayPreprocessor`, `SimplePreprocessor`, and `SimpleDatasetLoader` – these classes will form the actual *pipeline* used to process images before passing them through our network. We then import `ShallowNet` on **Line 8** along with `SGD` on **Line 9** – we'll be using Stochastic Gradient Descent to train ShallowNet.

Next, we need to parse our command line arguments and grab our image paths:

```
15 # construct the argument parser and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-d", "--dataset", required=True,
18      help="path to input dataset")
19 args = vars(ap.parse_args())
20
21 # grab the list of images that we'll be describing
22 print("[INFO] loading images...")
23 imagePaths = list(paths.list_images(args["dataset"]))
```

Our script requires only a single switch here, `--dataset`, which is the path to the directory containing our Animals dataset. **Line 23** then grabs the file paths to all 3,000 images inside Animals.

Remember how I was talking about creating a pipeline to load and process our dataset? Let's see how that is done now:

```
25 # initialize the image preprocessors
26 sp = SimplePreprocessor(32, 32)
27 iap = ImageToArrayPreprocessor()
28
29 # load the dataset from disk then scale the raw pixel intensities
30 # to the range [0, 1]
31 sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
32 (data, labels) = sdl.load(imagePaths, verbose=500)
33 data = data.astype("float") / 255.0
```

Line 26 defines the `SimpleProcessor` used to resize input images to 32×32 pixels. The `ImageToArrayPreprocessor` is then instantiated on **Line 27** to handle channel ordering.

We combine these preprocessors together on **Line 31** where we initialize the `SimpleDatasetLoader`. Take a look at the `preprocessors` parameter of the constructor – we are supplying a *list* of preprocessors that will be applied in *sequential order*. First, a given input image will be resized to 32×32 pixels. Then, the resized image will be have its channels ordered according to our `keras.json` configuration file. **Line 32** loads the images (applying the preprocessors) and the class labels. We then scale the images to the range $[0, 1]$.

Now that the data and labels are loaded, we can perform our training and testing split, along with one-hot encoding the labels:

```
35 # partition the data into training and testing splits using 75% of
36 # the data for training and the remaining 25% for testing
37 (trainX, testX, trainY, testY) = train_test_split(data, labels,
38      test_size=0.25, random_state=42)
39
40 # convert the labels from integers to vectors
41 trainY = LabelBinarizer().fit_transform(trainY)
42 testY = LabelBinarizer().fit_transform(testY)
```

Here we are using 75% of our data for training and 25% for testing.

The next step is to instantiate `ShallowNet`, followed by training the network itself:

```
44 # initialize the optimizer and model
45 print("[INFO] compiling model...")
46 opt = SGD(lr=0.005)
47 model = ShallowNet.build(width=32, height=32, depth=3, classes=3)
48 model.compile(loss="categorical_crossentropy", optimizer=opt,
49     metrics=["accuracy"])
50
51 # train the network
52 print("[INFO] training network...")
53 H = model.fit(trainX, trainY, validation_data=(testX, testY),
54     batch_size=32, epochs=100, verbose=1)
```

We initialize the `SGD` optimizer on **Line 46** using a learning rate of `0.005` (we'll discuss how to tune learning rates in a future chapter). The `ShallowNet` architecture is instantiated on **Line 47**, supplying a width and height of 32 pixels along with a depth of 3 – this implies that our input images are 32×32 pixels with three channels. Since the Animals dataset has three class labels, we set `classes=3`.

The `model` is then compiled on **Lines 48 and 49** where we'll use cross-entropy as our loss function and SGD as our optimizer. To actual train the network, we make a call to the `.fit` method of `model` on **Lines 53 and 54**. The `.fit` method requires us to pass in the training and testing data. We'll also supply our testing data so we can evaluate the performance of ShallowNet after each epoch. The network will be trained for 100 epochs using mini-batch sizes of 32 (meaning that 32 images will be presented to the network at a time, and a full forward and backward pass will be done to update the parameters of the network).

After training our network, we can evaluate its performance:

```
56 # evaluate the network
57 print("[INFO] evaluating network...")
58 predictions = model.predict(testX, batch_size=32)
59 print(classification_report(testY.argmax(axis=1),
60     predictions.argmax(axis=1),
61     target_names=["cat", "dog", "panda"]))
```

To obtain the output predictions on our testing data, we call `.predict` of the `model`. A nicely formatted classification report is displayed to our screen on **Lines 59-61**.

Our final code block handles plotting the accuracy and loss over time for *both* the training and testing data:

```
63 # plot the training loss and accuracy
64 plt.style.use("ggplot")
65 plt.figure()
66 plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
67 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
68 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
69 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
70 plt.title("Training Loss and Accuracy")
71 plt.xlabel("Epoch #")
72 plt.ylabel("Loss/Accuracy")
73 plt.legend()
74 plt.show()
```

To train ShallowNet on the Animals dataset, just execute the following command:

```
$ python shallownet_animals.py --dataset ../datasets/animals
```

Training should be quite fast as the network is *very* shallow and our image dataset is relatively small:

```
[INFO] loading images...
[INFO] processed 500/3000
[INFO] processed 1000/3000
[INFO] processed 1500/3000
[INFO] processed 2000/3000
[INFO] processed 2500/3000
[INFO] processed 3000/3000
[INFO] compiling model...
[INFO] training network...
Train on 2250 samples, validate on 750 samples
Epoch 1/100
0s - loss: 1.0290 - acc: 0.4560 - val_loss: 0.9602 - val_acc: 0.5160
Epoch 2/100
0s - loss: 0.9289 - acc: 0.5431 - val_loss: 1.0345 - val_acc: 0.4933
...
Epoch 100/100
0s - loss: 0.3442 - acc: 0.8707 - val_loss: 0.6890 - val_acc: 0.6947
[INFO] evaluating network...
             precision    recall  f1-score   support

        cat       0.58      0.77      0.67       239
        dog       0.75      0.40      0.52       249
      panda       0.79      0.90      0.84       262

avg / total       0.71      0.69      0.68       750
```

Due to the small amount of training data, epochs were quite speedy, taking less than one second on both my CPU and GPU.

As you can see from the output above, ShallowNet obtained 71% ***classification accuracy*** on our testing data, a massive improvement from our previous best of 59% using simple feedforward neural networks. Using more advanced training networks, as well as a more powerful architecture, we'll be able to boost classification accuracy even higher.

The loss and accuracy plotted over time is displayed in Figure 12.2. On the x-axis we have our epoch number and on the y-axis we have our loss and accuracy. Examining this figure, we can see that learning is a bit volatile with large spikes in loss around epoch 20 and epoch 60 – this result is likely due to our learning rate being too high, something we'll help resolve in Chapter 16.

Also take note that the training and testing loss diverge heavily past epoch 30, which implies that our network is modeling the training data *too closely* and overfitting. We can remedy this issue by obtaining more data or applying techniques like data augmentation (covered in the *Practitioner Bundle*).

Around epoch 60 our testing accuracy saturates – we are unable to get past $\approx 70\%$ classification accuracy, meanwhile our training accuracy continues to climb to over 85%. Again, gathering more training data, applying data augmentation, and taking more care to tune our learning rate will help us improve our results in the future.

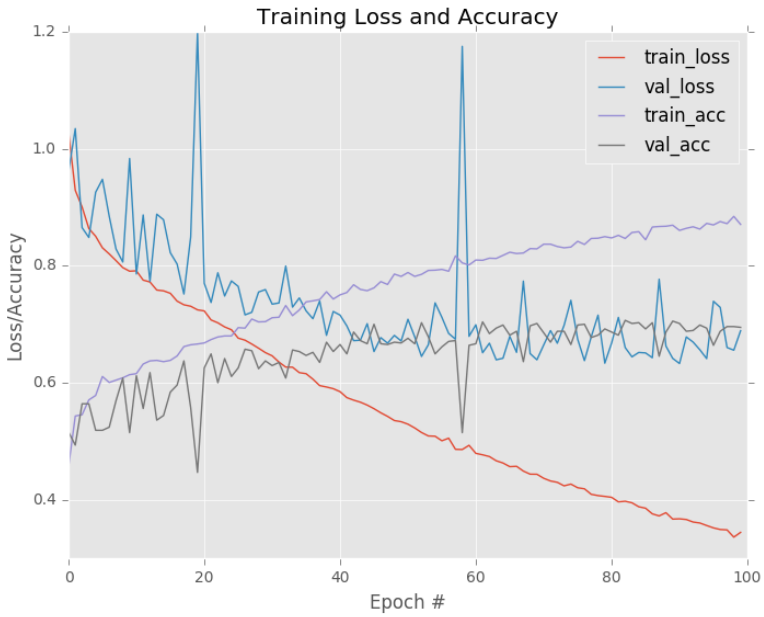

Figure 12.2: A plot of our loss and accuracy over the course of 100 epochs for the ShallowNet architecture trained on the Animals dataset.

The key point here is that an *extremely simple* Convolutional Neural Network was able to obtain 71% classification accuracy on the Animals dataset where our previous best was only 59% – that's an improvement of over 12%!

12.2.3 ShallowNet on CIFAR-10

Let's also apply the ShallowNet architecture to the CIFAR-10 dataset to see if we can improve our results. Open a new file, name it `shallownet_cifar10.py`, and insert the following code:

```
# import the necessary packages
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report
from pyimagesearch.nn.conv import ShallowNet
from keras.optimizers import SGD
from keras.datasets import cifar10
import matplotlib.pyplot as plt
import numpy as np

# load the training and testing data, then scale it into the
# range [0, 1]
print("[INFO] loading CIFAR-10 data...")
((trainX, trainY), (testX, testY)) = cifar10.load_data()
trainX = trainX.astype("float") / 255.0
testX = testX.astype("float") / 255.0

# convert the labels from integers to vectors
lb = LabelBinarizer()
```

```
19 trainY = lb.fit_transform(trainY)
20 testY = lb.transform(testY)
21
22 # initialize the label names for the CIFAR-10 dataset
23 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
24     "dog", "frog", "horse", "ship", "truck"]
```

Lines 2-8 import our required Python packages. We then load the CIFAR-10 dataset (pre-split into training and testing sets), followed by scaling the image pixel intensities to the range $[0, 1]$. Since the CIFAR-10 images are preprocessed and the channel ordering is handled *automatically* inside of `cifar10.load_data`, we do not need to apply any of our custom preprocessing classes.

Our labels are then one-hot encoded to vectors on **Lines 18-20**. We also initialize the label names for the CIFAR-10 dataset on **Lines 23 and 24**.

Now that our data is prepared, we can train ShallowNet:

```
26 # initialize the optimizer and model
27 print("[INFO] compiling model...")
28 opt = SGD(lr=0.01)
29 model = ShallowNet.build(width=32, height=32, depth=3, classes=10)
30 model.compile(loss="categorical_crossentropy", optimizer=opt,
31     metrics=["accuracy"])
32
33 # train the network
34 print("[INFO] training network...")
35 H = model.fit(trainX, trainY, validation_data=(testX, testY),
36     batch_size=32, epochs=40, verbose=1)
```

Line 28 initializes the SGD optimizer with a learning rate of `0.01`. ShallowNet is then constructed on **Line 29** using a width of 32, a height of 32, a depth of 3 (since CIFAR-10 images have three channels). We set `classes=10` since, as the name suggests, there are ten classes in the CIFAR-10 dataset. The model is compiled on **Lines 30 and 31** then trained on **Lines 35 and 36** over the course of 40 epochs.

Evaluating ShallowNet is done in the exact same manner as our previous example with the Animals dataset:

```
38 # evaluate the network
39 print("[INFO] evaluating network...")
40 predictions = model.predict(testX, batch_size=32)
41 print(classification_report(testY.argmax(axis=1),
42     predictions.argmax(axis=1), target_names=labelNames))
```

We'll also plot the loss and accuracy over time so we can get an idea how our network is performing:

```
44 # plot the training loss and accuracy
45 plt.style.use("ggplot")
46 plt.figure()
47 plt.plot(np.arange(0, 40), H.history["loss"], label="train_loss")
48 plt.plot(np.arange(0, 40), H.history["val_loss"], label="val_loss")
```

Figure 12.3: Loss and accuracy for ShallowNet trained on CIFAR-10. Our network obtains 60% classification accuracy; however, it is overfitting. Further accuracy can be obtained by applying regularization, which we'll cover later in this book.

```
49 plt.plot(np.arange(0, 40), H.history["acc"], label="train_acc")
50 plt.plot(np.arange(0, 40), H.history["val_acc"], label="val_acc")
51 plt.title("Training Loss and Accuracy")
52 plt.xlabel("Epoch #")
53 plt.ylabel("Loss/Accuracy")
54 plt.legend()
55 plt.show()
```

To train ShallowNet on CIFAR-10, simply execute the following command:

```
$ python shallownet_cifar10.py
[INFO] loading CIFAR-10 data...
[INFO] compiling model...
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
5s - loss: 1.8087 - acc: 0.3653 - val_loss: 1.6558 - val_acc: 0.4282
Epoch 2/40
5s - loss: 1.5669 - acc: 0.4583 - val_loss: 1.4903 - val_acc: 0.4724
...
Epoch 40/40
5s - loss: 0.6768 - acc: 0.7685 - val_loss: 1.2418 - val_acc: 0.5890
[INFO] evaluating network...
             precision    recall  f1-score   support
```

```
    airplane       0.62      0.68      0.65      1000
  automobile       0.79      0.64      0.71      1000
        bird       0.43      0.46      0.44      1000
         cat       0.42      0.38      0.40      1000
        deer       0.52      0.51      0.52      1000
         dog       0.44      0.57      0.50      1000
        frog       0.74      0.61      0.67      1000
       horse       0.71      0.61      0.66      1000
        ship       0.65      0.77      0.70      1000
       truck       0.67      0.66      0.66      1000

 avg / total       0.60      0.59      0.59     10000
```

Again, epochs are quite fast due to the shallow network architecture and relatively small dataset. Using my GPU, I obtained 5-second epochs while my CPU took 22 seconds for each epoch.

After 40 epochs ShallowNet is evaluated and we find that it obtains **60% accuracy** on the testing set, an increase from the previous 57% accuracy using simple neural networks.

More importantly, plotting our loss and accuracy in Figure 12.3 gives us some insight to the training process demonstrates that our validation loss does not skyrocket. Our training and testing loss/accuracy start to diverge past epoch 10. Again, this can be attributed to a larger learning rate and the fact we aren't using methods to help combat overfitting (regularization parameters, dropout, data augmentation, etc.).

It is also *notoriously easy* to overfit on the CIFAR-10 dataset due to the limited number of low-resolution training samples. As we become more comfortable building and training our own custom Convolutional Neural Networks, we'll discover methods to boost classification accuracy on CIFAR-10 while simultaneously reducing overfitting.

12.3 Summary

In this chapter, we implemented our first Convolutional Neural Network architecture, ShallowNet, and trained it on the Animals and CIFAR-10 dataset. ShallowNet obtained 71% classification accuracy on Animals, an increase of 12% from our previous best using simple feedforward neural networks.

When applied to CIFAR-10, ShallowNet reached 60% accuracy, an increase of the previous best of 57% using simple multi-layer NNs (and without the *significant* overfitting).

ShallowNet is an *extremely* simple CNN that uses only *one* `CONV` layer – further accuracy can be obtained by training deeper networks with multiple sets of `CONV => RELU => POOL` operations.

13. Saving and Loading Your Models

In our last chapter, you learned how to train your first Convolutional Neural Network using the Keras library. However, you might have noticed that each time you wanted to evaluate your network or test it on a set of images, you first needed to train it *before* you could do any type of evaluation. This requirement can be quite the nuisance.

We are only working with a shallow network on a small dataset which can be trained relatively quickly, but what if our network was deep and we needed to train it on a much larger dataset, thus taking many hours or even days to train? Would we have to invest this amount of time and resources to train our network *each and every time?* Or is there a way to *save* our model to disk after training is complete and then simply load it from disk when we want to classify new images?

You bet there's a way. The process of saving and loading a trained model is called **model serialization** and is the primary topic of this chapter.

13.1 Serializing a Model to Disk

Using the Keras library, model serialization is as simple as calling `model.save` on a trained model and then loading it via the `load_model` function. In the first part of this chapter, we'll modify our ShallowNet training script from the last chapter to serialize the network after it's been trained on the Animals dataset. We'll then create a second Python script that demonstrates how to load our serialized model from disk.

Let's get started with the training part – open up a new file, name it `shallownet_train.py`, and insert the following code:

```
1 # import the necessary packages
2 from sklearn.preprocessing import LabelBinarizer
3 from sklearn.model_selection import train_test_split
4 from sklearn.metrics import classification_report
5 from pyimagesearch.preprocessing import ImageToArrayPreprocessor
6 from pyimagesearch.preprocessing import SimplePreprocessor
7 from pyimagesearch.datasets import SimpleDatasetLoader
```

```
8  from pyimagesearch.nn.conv import ShallowNet
9  from keras.optimizers import SGD
10 from imutils import paths
11 import matplotlib.pyplot as plt
12 import numpy as np
13 import argparse
```

Lines 2-13 import our required Python packages. Much of the code in this example is identical to `shallownet_animals.py` from Chapter 12. We'll review the entire file, for the sake of completeness, and I'll be sure to call out the important changes made to accomplish model serialization, but for a detailed review of how to train ShallowNet on the Animals dataset, please refer Section 12.2.1.

Next, let's parse our command line arguments:

```
15 # construct the argument parse and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-d", "--dataset", required=True,
18     help="path to input dataset")
19 ap.add_argument("-m", "--model", required=True,
20     help="path to output model")
21 args = vars(ap.parse_args())
```

Our previous script only required a *single* switch, `--dataset`, which is the path to the input Animals dataset. However, as you can see, we've added another switch here – `--model` which is the path to where we would like to *save network after training is complete*.

We can now grab the paths to the images in our `--dataset`, initialize our preprocessors, and load our image dataset from disk:

```
23 # grab the list of images that we'll be describing
24 print("[INFO] loading images...")
25 imagePaths = list(paths.list_images(args["dataset"]))
26
27 # initialize the image preprocessors
28 sp = SimplePreprocessor(32, 32)
29 iap = ImageToArrayPreprocessor()
30
31 # load the dataset from disk then scale the raw pixel intensities
32 # to the range [0, 1]
33 sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
34 (data, labels) = sdl.load(imagePaths, verbose=500)
35 data = data.astype("float") / 255.0
```

The next step is to partition our data into training and testing splits, along with encoding our labels as vectors:

```
37 # partition the data into training and testing splits using 75% of
38 # the data for training and the remaining 25% for testing
39 (trainX, testX, trainY, testY) = train_test_split(data, labels,
40     test_size=0.25, random_state=42)
41
42 # convert the labels from integers to vectors
```

```
trainY = LabelBinarizer().fit_transform(trainY)
testY = LabelBinarizer().fit_transform(testY)
```

Training ShallowNet is handled via the code block below:

```
# initialize the optimizer and model
print("[INFO] compiling model...")
opt = SGD(lr=0.005)
model = ShallowNet.build(width=32, height=32, depth=3, classes=3)
model.compile(loss="categorical_crossentropy", optimizer=opt,
	metrics=["accuracy"])

# train the network
print("[INFO] training network...")
H = model.fit(trainX, trainY, validation_data=(testX, testY),
	batch_size=32, epochs=100, verbose=1)
```

Now that our network is trained, we need to save it to disk. This process is as simple as calling `model.save` and supplying the path to where our output network should be saved to disk:

```
# save the network to disk
print("[INFO] serializing network...")
model.save(args["model"])
```

The `.save` method takes the weights and state of the optimizer and serializes them to disk in HDF5 format. As we'll see in the next section, loading these weights from disk is just as easy as saving them.

From here we evaluate our network:

```
# evaluate the network
print("[INFO] evaluating network...")
predictions = model.predict(testX, batch_size=32)
print(classification_report(testY.argmax(axis=1),
	predictions.argmax(axis=1),
	target_names=["cat", "dog", "panda"]))
```

As well as plot our loss and accuracy:

```
# plot the training loss and accuracy
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

To run our script, simply execute the following command:

```
$ python shallownet_train.py --dataset ../datasets/animals \
    --model shallownet_weights.hdf5
```

After the network has finished training, list the contents of your directory:

```
$ ls
shallownet_load.py  shallownet_train.py  shallownet_weights.hdf5
```

And you will see a file named `shallownet_weights.hdf5` – this file is our serialized network. The next step is to take this saved network and load it from disk.

13.2 Loading a Pre-trained Model from Disk

Now that we've trained our model and serialized it, we need to load it from disk. As a practical application of model serialization, I'll be demonstrating how to classify *individual images* from the Animals dataset and then display the classified images to our screen.

Open a new file, name it `shallownet_load.py`, and we'll get our hands dirty:

```
1  # import the necessary packages
2  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
3  from pyimagesearch.preprocessing import SimplePreprocessor
4  from pyimagesearch.datasets import SimpleDatasetLoader
5  from keras.models import load_model
6  from imutils import paths
7  import numpy as np
8  import argparse
9  import cv2
```

We start off by importing our required Python packages. **Lines 2-4** import the classes uses to construct our standard pipeline of resizing an image to a fixed size, converting it to a Keras compatible array, and then using these preprocessors to load an entire image dataset into memory.

The actual function used to load our trained model from disk is `load_model` on **Line 5**. This function is responsible for accepting the path to our trained network (an HDF5 file), decoding the weights and optimizer inside the HDF5 file, and setting the weights inside our architecture so we can (1) continue training or (2) use the network to classify new images.

We'll import our OpenCV bindings on **Line 9** as well so we can draw the classification label on our images and display them to our screen.

Next, let's parse our command line arguments:

```
11 # construct the argument parse and parse the arguments
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-d", "--dataset", required=True,
14     help="path to input dataset")
15 ap.add_argument("-m", "--model", required=True,
16     help="path to pre-trained model")
17 args = vars(ap.parse_args())
18
19 # initialize the class labels
20 classLabels = ["cat", "dog", "panda"]
```

Just like in `shallownet_save.py`, we'll need two command line arguments:

1. `--dataset`: The path to the directory that contains images that we wish to classify (in this case, the Animals dataset).
2. `--model`: The path to the *trained network* serialized on disk.

Line 20 then initializes a list of class labels for the Animals dataset.

Our next code block handles randomly sampling ten image paths from the Animals dataset for classification:

```
22 # grab the list of images in the dataset then randomly sample
23 # indexes into the image paths list
24 print("[INFO] sampling images...")
25 imagePaths = np.array(list(paths.list_images(args["dataset"])))
26 idxs = np.random.randint(0, len(imagePaths), size=(10,))
27 imagePaths = imagePaths[idxs]
```

Each of these ten images will need to be preprocessed, so let's initialize our preprocessors and load the ten images from disk:

```
29 # initialize the image preprocessors
30 sp = SimplePreprocessor(32, 32)
31 iap = ImageToArrayPreprocessor()
32
33 # load the dataset from disk then scale the raw pixel intensities
34 # to the range [0, 1]
35 sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
36 (data, labels) = sdl.load(imagePaths)
37 data = data.astype("float") / 255.0
```

Notice how we are preprocessing our images in the *exact same manner* in which we preprocessed our images during training. Failing to do this procedure can lead to incorrect classifications since the network will be presented with patterns it cannot recognize. Always take special care to ensure your *testing images* were preprocessed in the same way as your *training images*.

Next, let's load our saved network from disk:

```
39 # load the pre-trained network
40 print("[INFO] loading pre-trained network...")
41 model = load_model(args["model"])
```

Loading our serialized network is as simple as calling `load_model` and supplying the path to model's HDF5 file residing on disk.

Once the model is loaded, we can make predictions on our ten images:

```
43 # make predictions on the images
44 print("[INFO] predicting...")
45 preds = model.predict(data, batch_size=32).argmax(axis=1)
```

Keep in mind that the `.predict` method of `model` will return a *list of probabilities* for every image in `data` – one probability for each class label, respectively. Taking the `argmax` on `axis=1` finds the index of the class label with the *largest probability* for each image.

Now that we have our predictions, let's visualize the results:

```
47 # loop over the sample images
48 for (i, imagePath) in enumerate(imagePaths):
49     # load the example image, draw the prediction, and display it
50     # to our screen
51     image = cv2.imread(imagePath)
52     cv2.putText(image, "Label: {}".format(classLabels[preds[i]]),
53         (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
54     cv2.imshow("Image", image)
55     cv2.waitKey(0)
```

On **Line 48** we start looping over our ten randomly sampled image paths. For each image, we load it from disk (**Line 51**) and draw the class label prediction on the image itself (**Lines 52 and 53**). The output image is then displayed to our screen on **Lines 54 and 55**.

To give `shallownet_load.py` a try, execute the following command:

```
$ python shallownet_load.py --dataset ../datasets/animals \
    --model shallownet_weights.hdf5
[INFO] sampling images...
[INFO] loading pre-trained network...
[INFO] predicting...
```

Based on the output, you can see that our images have been sampled, the pre-trained ShallowNet weights have been loaded from disk, and that ShallowNet has made predictions on our images. I have included a sample of predictions from the ShallowNet drawn on the images themselves in Figure 13.1.

Figure 13.1: A sample of images correctly classified by our ShallowNet CNN.

Keep in mind that ShallowNet is obtaining $\approx 70\%$ classification accuracy on the Animals dataset, meaning that nearly one in every three example images will be classified incorrectly. Furthermore, based on the `classification_report` from Section 12.2.2, we know that the network still struggles to consistently discriminate between dogs and cats. As we continue our journey applying deep learning to computer vision classification tasks, we'll look at methods to help us boost our classification accuracy.

13.3 Summary

In this chapter we learned how to:

1. Train a network.
2. Serialize the network weights and optimizer state to disk.
3. Load the trained network and classify images.

Later in Chapter 18 we'll discover how we can save our model's weights to disk after *every epoch*, allowing us to "checkpoint" our network and choose the best performing one. Saving model weights during the actual training process also enables us to *restart training from a specific point* if our network starts exhibiting signs of overfitting. The process of stopping training, tweaking parameters, and then restarting training again is covered in-depth inside the *Practitioner Bundle* and *ImageNet Bundle*.