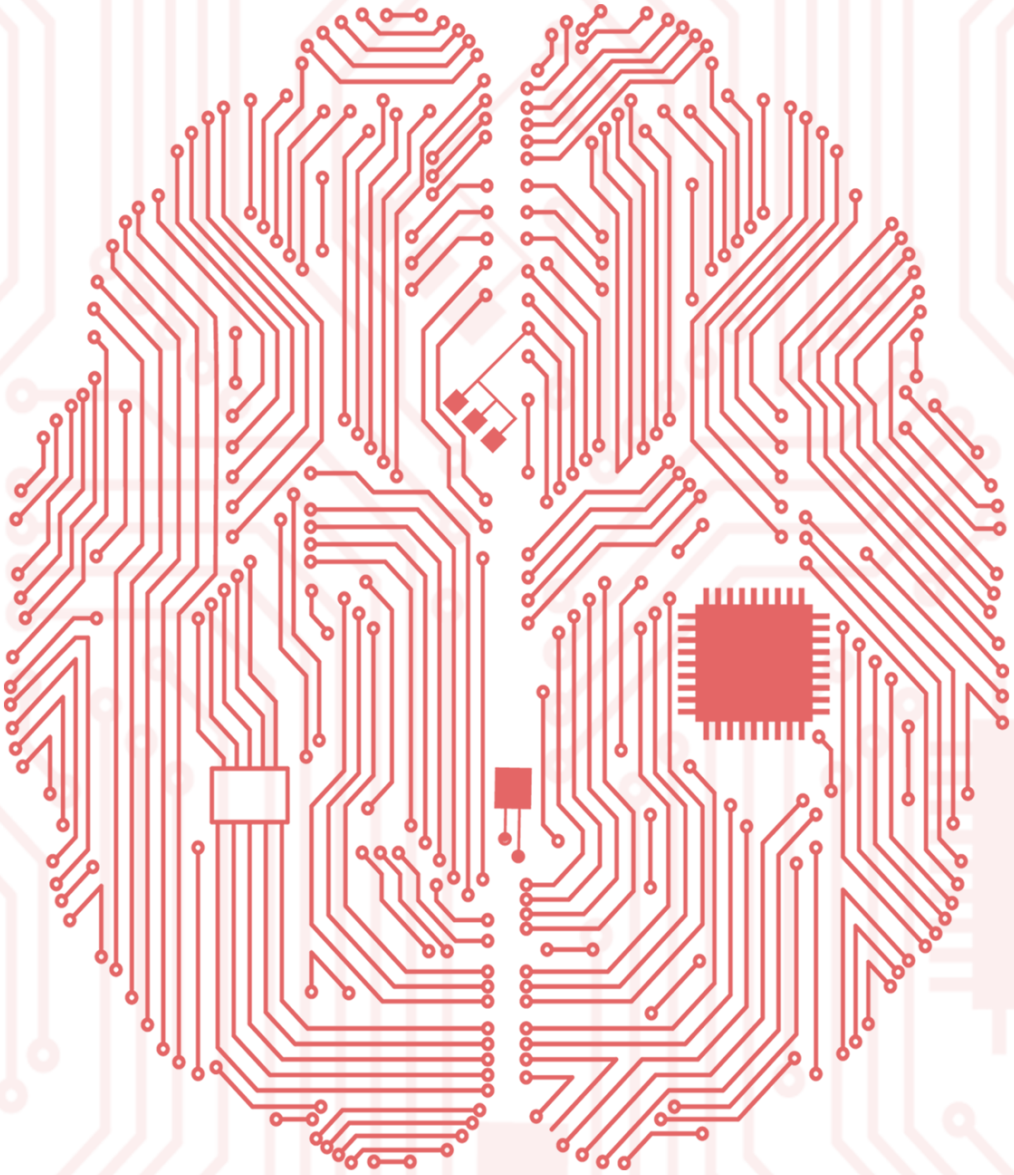
DEEP LEARNING FOR COMPUTER VISION



WITH PYTHON

Dr. Adrian Rosebrock

pyimagesearch

Deep Learning for Computer Vision with Python

Practitioner Bundle

Dr. Adrian Rosebrock

2nd Edition (2.1)

PUBLISHED BY PYIMAGESEARCH

PYIMAGESEARCH.COM

Second printing, November 2018

To my father, Joe; my wife, Trisha;
and the family beagles, Josie and Jemma.
Without their constant love and support,
this book would not be possible.

Contents

Companion Website

Thank you for picking up a copy of *Deep Learning for Computer Vision with Python!* To accompany this book I have created a companion website which includes:

- **Up-to-date installation instructions** on how to configure your development environment
- Instructions on how to use the **pre-configured Ubuntu VirtualBox virtual machine** and **Amazon Machine Image (AMI)**
- **Supplementary material** that I could not fit inside this book
- **Frequently Asked Questions (FAQs)** and their suggested fixes and solutions

Additionally, you can use the "Issues" feature inside the companion website to report any bugs, typos, or problems you encounter when working through the book. I don't expect many problems; however, this is a brand new book so myself and other readers would appreciate reporting any issues you run into. From there, I can keep the book updated and bug free.

To create your companion website account, just use this link:

http://pyimg.co/fnkxk

Take a second to create your account now so you'll have access to the supplementary materials as you work through the book.

1. Introduction

Welcome to the *Practitioner Bundle* of *Deep Learning for Computer Vision with Python!* This volume is meant to be the next logical step in your deep learning for computer vision education after completing the *Starter Bundle*.

At this point, you should have a strong understanding of the *fundamentals* of parameterized learning, neural networks, and Convolutional Neural Networks (CNNs). You should also feel relatively comfortable using the Keras library and the Python programming language to train your own custom deep learning networks.

The purpose of the *Practitioner Bundle* is to build on your knowledge gained from the *Starter Bundle* and introduce more advanced algorithms, concepts, and tricks of the trade — these techniques will be covered in three distinct parts of the book.

The first part will focus on methods that are used to boost your classification accuracy in one way or another. One way to increase your classification accuracy is to apply transfer learning methods such as fine-tuning or treating your network as a feature extractor.

We'll also explore *ensemble methods* (i.e., training *multiple networks* and combining the results) and how these methods can give you a nice classification boost with little extra effort. Regularization methods such as *data augmentation* are used to generate additional training data – in nearly all situations, data augmentation improves your model's ability to generalize. More advanced optimization algorithms such as Adam [1], RMSprop [2], and others can also be used on some datasets to help you obtain lower loss. After we review these techniques, we'll look at the *optimal pathway to apply these methods* to ensure you obtain the maximum amount of benefit with the least amount of effort.

We then move on to the second part of the *Practitioner Bundle* which focuses on *larger datasets* and *more exotic network architectures*. Thus far we have only worked with datasets that have fit into the main memory of our system – but what if our dataset is too large to fit into RAM? What do we do then? We'll address this question in Chapter 9 when we work with HDF5.

Given that we'll be working with larger datasets, we'll also be able to discuss more advanced network architectures using AlexNet, GoogLeNet, ResNet, and deeper variants of VGGNet. These network architectures will be applied to more challenging datasets and competitions, including the

Kaggle: Dogs vs. Cats recognition challenge [3] as well as the cs231n Tiny ImageNet challenge [4], the exact same task Stanford CNN students compete in. As we'll find out, we'll be able to obtain a top-25 position on the Kaggle Dogs vs. Cats leaderboard and top the cs231n challenge for our technique type.

The final part of this book covers applications of deep learning for computer vision *outside* of image classification, including basic object detection, deep dreaming and neural style, Generative Adversarial Networks (GANs), and Image Super Resolution. Again, the techniques covered in this volume are meant to be *much* more advanced than the *Starter Bundle* – this is where you'll start to separate yourself from a *deep learning novice* and transform into a true ***deep learning practitioner***. To start your transformation to deep learning expert, just flip the page.

2. Data Augmentation

According to Goodfellow et al., regularization is *"any modification we make to a learning algorithm that is intended to reduce its generalization error, but not its training error"* [5]. In short, regularization seeks to reduce our testing error perhaps at the expense of increasing training error slightly.

We've already looked at different forms of regularization in Chapter 9 of the *Starter Bundle*; however, these were *parameterized* forms of regularization, requiring us to update our loss/update function. In fact, there exist other types of regularization that either:

1. Modify the network architecture itself.
2. Augment the data passed into the network for training.

Dropout is a great example of modifying a network architecture by achieving greater generalizability. Here we insert a layer that randomly disconnects nodes from the *previous* layer to the *next* layer, thereby ensuring that *no single node* is responsible for learning how to represent a given class.

In the remainder of this chapter, we'll be discussing another type of regularization called ***data augmentation***. This method purposely perturbs training examples, changing their appearance slightly, before passing them into the network for training. The end result is that a network consistently sees "new" training data points generated from the *original training data*, partially alleviating the need for us to gather more training data (though in general, gathering more training data will rarely hurt your algorithm).

2.1 What Is Data Augmentation?

Data augmentation encompasses a wide range of techniques used to generate new training samples from the original ones by applying random jitters and perturbations such that the classes labels are not changed. Our goal when applying data augmentation is to increase the generalizability of the model. Given that our network is constantly seeing new, slightly modified versions of the input data points, it's able to learn more robust features. At testing time, we *do not* apply data augmentation and evaluate our trained network – in most cases, you'll see an increase in testing accuracy, perhaps at the expense at a slight dip in training accuracy.

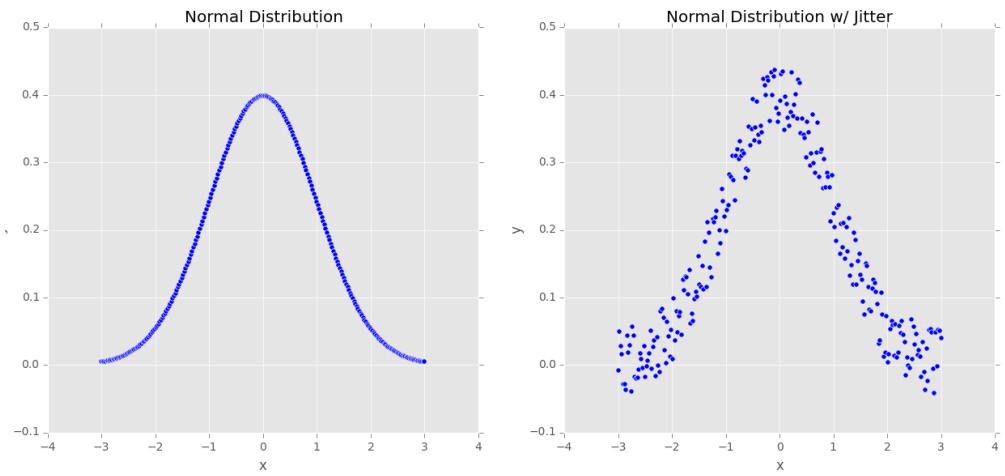


Figure 2.1: **Left:** A sample of 250 data points that follow a normal distribution exactly. **Right:** Adding a small amount of random "jitter" to the distribution. This type of data augmentation can increase the generalizability of our networks.

Let's consider the Figure 2.1 (*left*) of a normal distribution with zero mean and unit variance. Training a machine learning model on this data may result in us modeling the distribution *exactly* – however, in real-world applications, data rarely follows such a neat distribution.

Instead, to increase the generalizability of our classifier, we may first randomly jitter points along the distribution by adding some values ε drawn from a random distribution (*right*). Our plot still follows an *approximately* normal distribution, but it's not a *perfect* distribution as on the *left*. A model trained on this data is more likely to generalize to example data points not included in the training set.

In the context of computer vision, data augmentation lends itself naturally. For example, we can obtain additional training data from the original images by apply simple geometric transforms such as random:

1. Translations
2. Rotations
3. Changes in scale
4. Shearing
5. Horizontal (and in some cases, vertical) flips

Applying a (small) amount of these transformations to an input image will change its appearance slightly, but it *does not* change the class label – thereby making data augmentation a very natural, easy method to apply to deep learning for computer vision tasks. More advanced techniques for data augmentation applied to computer vision include random perturbation of colors in a given color space [6] and nonlinear geometric distortions [7].

2.2 Visualizing Data Augmentation

The best way to understand data augmentation applied to computer tasks is to simply visualize a given input being augmented and distorted. To accomplish this visualization, let's build a simple Python script that uses the built-in power of Keras to perform data augmentation. Create a new file, name it `augmentation_demo.py`. and insert the following code:

```
1  # import the necessary packages
2  from keras.preprocessing.image import ImageDataGenerator
3  from keras.preprocessing.image import img_to_array
4  from keras.preprocessing.image import load_img
5  import numpy as np
6  import argparse
```

Lines 2-6 import our required Python packages. Take note of **Line 2** where we import the `ImageDataGenerator` class from Keras – this code will be used for data augmentation and includes all relevant methods to help us transform our input image.

Next, we parse our command line arguments:

```
8  # construct the argument parse and parse the arguments
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-i", "--image", required=True,
11      help="path to the input image")
12 ap.add_argument("-o", "--output", required=True,
13      help="path to output directory to store augmentation examples")
14 ap.add_argument("-p", "--prefix", type=str, default="image",
15      help="output filename prefix")
16 args = vars(ap.parse_args())
```

Our script requires three command line arguments, each detailed below:

- `--image`: This is the path to the input image that we want to apply data augmentation to and visualize the results.
- `--output`: After applying data augmentation to a given image, we would like to store the result on disk so we can inspect it – this switch controls the output directory.
- `--prefix`: A string that will be prepended to the output image filename.

Now that our command line arguments are parsed, let's load our input image, convert it to a Keras-compatible array, and add an extra dimension to the image, just as we would do if we were preparing our image for classification:

```
18 # load the input image, convert it to a NumPy array, and then
19 # reshape it to have an extra dimension
20 print("[INFO] loading example image...")
21 image = load_img(args["image"])
22 image = img_to_array(image)
23 image = np.expand_dims(image, axis=0)
```

We are now ready to initialize our `ImageDataGenerator`:

```
25 # construct the image generator for data augmentation then
26 # initialize the total number of images generated thus far
27 aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
28      height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
29      horizontal_flip=True, fill_mode="nearest")
30 total = 0
```

The `ImageDataGenerator` class has a number of parameters, too many to enumerate in this book. For a full review of the parameters, please refer to the official Keras documentation (http://pyimg.co/j8ad8).

Instead, we'll be focusing on the augmentation parameters you will most likely use in your own applications. The `rotation_range` parameter controls the degree range of the random rotations. Here we'll allow our input image to be randomly rotated ± 30 degrees. Both the `width_shift_range` and `height_shift_range` are used for horizontal and vertical shifts, respectively. The parameter value is a *fraction* of the given dimension, in this case, 10%.

The `shear_range` controls the angle in counterclockwise direction as radians in which our image will allowed to be sheared. We then have the `zoom_range`, a floating point value that allows the image to be "zoomed in" or "zoomed out" according to the following uniform distribution of values: `[1 - zoom_range, 1 + zoom_range]`.

Finally, the `horizontal_flip` boolean controls whether or not a given input is allowed to be flipped horizontally during the training process. For most computer vision applications a horizontal flip of an image does not change the resulting class label – but there are applications where a horizontal (or vertical) flip does change the semantic meaning of the image. Take care when applying this type of data augmentation as our goal is to slightly modify the input image, thereby generating a new training sample, *without changing the class label itself*. For a more detailed review of image transformations, please refer to Module #1 in PyImageSearch Gurus ([8], PyImageSearch Gurus) as well as Szeliski [9].

Once `ImageDataGenerator` is initialized, we can actually generate new training examples:

```
32 # construct the actual Python generator
33 print("[INFO] generating images...")
34 imageGen = aug.flow(image, batch_size=1, save_to_dir=args["output"],
35      save_prefix=args["prefix"], save_format="jpg")
36
37 # loop over examples from our image data augmentation generator
38 for image in imageGen:
39      # increment our counter
40      total += 1
41
42      # if we have reached 10 examples, break from the loop
43      if total == 10:
44           break
```

Lines 34 and 35 initialize a Python generator used to construct our augmented images. We'll pass in our input `image`, a `batch_size` of 1 (since we are only augmenting one image), along with a few additional parameters to specify the output image file paths, the prefix for each file path, and the image file format. **Line 38** then starts looping over each `image` in the `imageGen` generator. Internally, `imageGen` is *automatically* generating a new training sample each time one is requested via the loop. We then increment the total number of data augmentation examples written to disk and stop the script from executing once we've reached ten examples.

To visualize data augmentation in action, we'll be using Figure 2.2 (*left*), an image of Jemma, my family beagle. To generate new training example images of Jemma, just execute the following command:

```
$ python augmentation_demo.py --image jemma.png --output output
```

After the script executes you should see ten images in the `output` directory:

```
$ ls output/
image_0_1227.jpg  image_0_2358.jpg  image_0_4205.jpg  image_0_4770.jpg
```

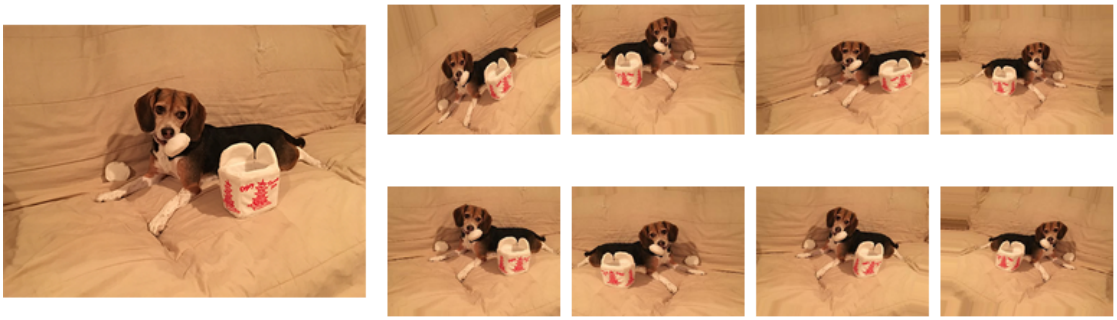


Figure 2.2: **Left:** The input image we are going to apply data augmentation to. **Right:** A montage of data augmentation examples. Notice how each image has been randomly rotated, sheared, zoomed, and horizontally flipped.

```
image_0_1933.jpg  image_0_2914.jpg  image_0_4657.jpg  image_0_6934.jpg
image_0_9197.jpg  image_0_953.jpg
```

I have constructed a montage of each of these images so you can visualize them in Figure 2.2 (*right*). Notice how each image has been randomly rotated, sheared, zoomed, and horizontally flipped. In each case the image retains the original class label: *dog*; however, each image has been modified slightly, thereby giving our neural network new patterns to learn from when training. Since the input images will constantly be changing (while the class labels remain the same), it's common to see our training accuracy decrease when compared to training *without* data augmentation.

However, as we'll find out later in this chapter, data augmentation can help *dramatically* reduce overfitting, all the while ensuring that our model generalizes better to new input samples. Furthermore, when working with datasets where we have *too few examples* to apply deep learning, we can utilize data augmentation to generate *additional training data*, thereby reducing the amount of hand-labeled data required to train a deep learning network.

2.3 Comparing Training With and Without Data Augmentation

In the first part of this section, we'll discuss the Flowers-17 dataset, a very small dataset (in terms of deep learning for computer vision tasks), and how data augmentation can help us artificially increase the size of this dataset by generating additional training samples. From there we'll perform two experiments:

1. Train MiniVGGNet on Flowers-17 *without* data augmentation.
2. Train MiniVGGNet on Flowers-17 *with* data augmentation.

As we'll find out, applying data augmentation dramatically reduces overfitting and allows MiniVGGNet to obtain substantially higher classification accuracy.

2.3.1 The Flowers-17 Dataset

The Flowers-17 dataset [10] is a fine-grained classification challenge where our task is to recognize 17 distinct species of flowers. The image dataset is quite small, having only 80 images per class for a total of 1,360 images. A general rule of thumb when applying deep learning to computer vision tasks is to have 1,000-5,000 examples *per class*, so we are certainly at a huge deficit here.

We call the Flowers-17 a fine-grained classification task because all categories are very similar (i.e., species of flower). In fact, we can think of each of these categories as *subcategories*. The categories are certainly different, but share a significant amount of common structure (e.x., petals,

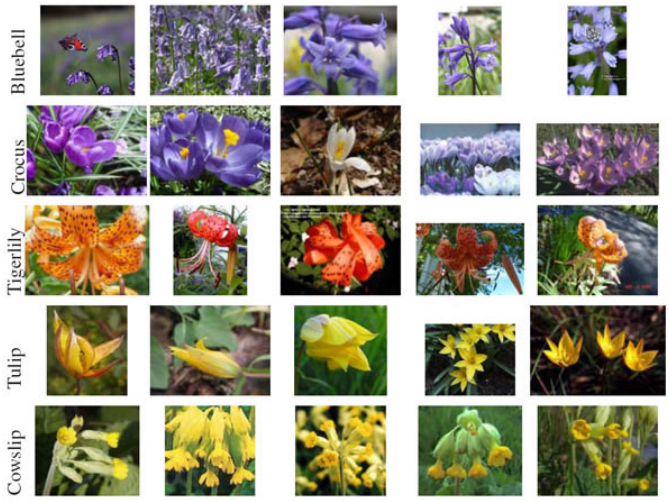


Figure 2.3: A sample of five (out of the seventeen total) classes in the Flowers-17 dataset where each class represents a *specific* flower species.

stamen, pistil, etc.). Fine-grained classification tasks tend to be the most challenging for deep learning practitioners as it implies that our machine learning models need to learn *extremely discriminating features* to distinguish between classes that are very similar. This fine-grained classification task becomes even more problematic given our limited training data.

2.3.2 Aspect-aware Preprocessing

Up until this point, we have only preprocessed images by resizing them to a fixed size, *ignoring the aspect ratio*. In some situations, especially for basic benchmark datasets, doing so is acceptable.

However, for more challenging datasets we should still seek to resize to a fixed size, *but maintain the aspect ratio*. To visualize this action, consider Figure 2.4.

On the *left*, we have an input image that we need to resize to a fixed width and height. Ignoring the aspect ratio, we resize the image to 256×256 pixels *(middle)*, effectively squishing and distorting the image such that it meets our desired dimensions. A better approach would be to take into account the aspect ratio of the image *(right)* where we first resize along the *shorter dimension* such that the width is 256 pixels and then *crop the image* along the height, such that the height is 256 pixels.

While we have effectively discarded part of the image during the crop, we have also maintained the original aspect ratio of the image. Maintaining a consistent aspect ratio allows our Convolutional Neural Network to learn more discriminative, consistent features. This is a common technique that we'll be applying when working with more advanced datasets throughout the rest of the *Practitioner Bundle* and *ImageNet Bundle*.

To see how aspect-aware preprocessing is implemented, let's update our `pyimagesearch` project structure to include a `AspectAwarePreprocessor`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |--- preprocessing
|    |    |--- __init__.py
|    |    |--- aspectawarepreprocessor.py
```

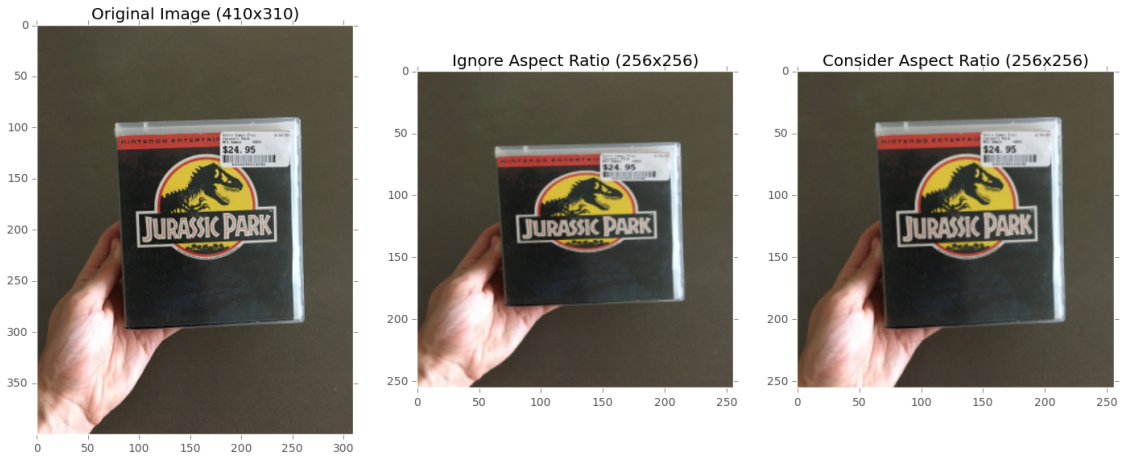


Figure 2.4: Left: The original input image (410×310). Middle: Resizing the image to 256×256 pixels, ignoring the aspect ratio. Notice how the image now appears squished and distorted. Right: Resizing the image to 256×256 while maintaining the aspect ratio.

```
|    |    |--- imagetoarraypreprocessor.py
|    |    |--- simplepreprocessor.py
|    |--- utils
```

Notice how we have added a new file named `aspectawarepreprocessor.py` inside the `preprocessing` sub-module – this location is where our new preprocessor will live. Open up `aspectawarepreprocessor.py` and insert the following code:

```
1  # import the necessary packages
2  import imutils
3  import cv2
4
5  class AspectAwarePreprocessor:
6      def __init__(self, width, height, inter=cv2.INTER_AREA):
7          # store the target image width, height, and interpolation
8          # method used when resizing
9          self.width = width
10         self.height = height
11         self.inter = inter
```

Just as in our `SimplePreprocessor`, our constructor requires two parameters (the desired `width` and `height` of the target output image) along with the interpolation method used when resizing the image. We can then define the `preprocess` function below:

```
13     def preprocess(self, image):
14         # grab the dimensions of the image and then initialize
15         # the deltas to use when cropping
16         (h, w) = image.shape[:2]
17         dW = 0
18         dH = 0
```

The `preprocess` function accepts a single argument, the `image` that we wish to preprocess. **Line 16** grabs the width and height of the input `image`, while **Lines 17 and 18** determine the delta offsets we'll be using when cropping along the larger dimension. Again, our aspect-aware preprocessor is a two step algorithm:

1. **Step #1:** Determine the shortest dimension and resize along it.
2. **Step #2:** Crop the image along the largest dimension to obtain the target width and height.

The following code block handles checking if the width is smaller than the height, and if so, resizes along the width:

```
20          # if the width is smaller than the height, then resize
21          # along the width (i.e., the smaller dimension) and then
22          # update the deltas to crop the height to the desired
23          # dimension
24          if w < h:
25               image = imutils.resize(image, width=self.width,
26                    inter=self.inter)
27               dH = int((image.shape[0] - self.height) / 2.0)
```

Otherwise, if the height is smaller than the width, then we resize along the height:

```
29          # otherwise, the height is smaller than the width so
30          # resize along the height and then update the deltas
31          # to crop along the width
32          else:
33               image = imutils.resize(image, height=self.height,
34                    inter=self.inter)
35               dW = int((image.shape[1] - self.width) / 2.0)
```

Now that our image is resized, we need to re-grab the width and height and use the deltas to crop the center of the image:

```
37          # now that our images have been resized, we need to
38          # re-grab the width and height, followed by performing
39          # the crop
40          (h, w) = image.shape[:2]
41          image = image[dH:h - dH, dW:w - dW]
42
43          # finally, resize the image to the provided spatial
44          # dimensions to ensure our output image is always a fixed
45          # size
46          return cv2.resize(image, (self.width, self.height),
47               interpolation=self.inter)
```

When cropping (due to rounding errors), our image target image dimensions may be off by $\pm$ one pixel; therefore, we make a call to `cv2.resize` to *ensure* our output image has the desired `width` and `height`. The preprocessed image is then returned to the calling function. Now that we've implemented our `AspectAwarePreprocessor`, let's put it to work when training the MiniVGGNet architecture on the Flowers-17 dataset.

2.3.3 Flowers-17: No Data Augmentation

To start, let's establish a baseline using *no data augmentation* when training the MiniVGGNet architecture (Chapter 15, *Starter Bundle*) on the Flowers-17 dataset. Open up a new file, name it `minivggnet_flowers17.py`, and we'll 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 AspectAwarePreprocessor
7  from pyimagesearch.datasets import SimpleDatasetLoader
8  from pyimagesearch.nn.conv import MiniVGGNet
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
14 import os
```

Lines 2-14 import our required Python packages. Most of these imports you've seen before, but I want to draw your attention to:

1. **Line 6:** Here we import our newly defined `AspectAwarePreprocessor`.
2. **Line 7**: Despite using a separate image preprocessor, we'll still be able to use `SimpleDatasetLoader` to load our dataset from disk.
3. **Line 8:** We'll be training the MiniVGGNet architecture on our dataset.

Next, we parse our command line arguments:

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

We only need a single switch here, `--dataset`, which is the path to our Flowers-17 dataset directory on disk.

Let's go ahead and extract the class labels from our input images:

```
22 # grab the list of images that we'll be describing, then extract
23 # the class label names from the image paths
24 print("[INFO] loading images...")
25 imagePaths = list(paths.list_images(args["dataset"]))
26 classNames = [pt.split(os.path.sep)[-2] for pt in imagePaths]
27 classNames = [str(x) for x in np.unique(classNames)]
```

Our Flowers-17 dataset has the following directory structure:

```
flowers17/{species}/{image}
```

An example of an image in the dataset follows:

```
flowers17/bluebell/image_0241.jpg
```

Therefore, to extract the class labels, we can simply extract the second to last index after splitting on the path separator (**Line 26**) yielding the text `bluebell`. If you struggle to see how this path and label extraction works, I would suggest opening a Python shell and playing around with file paths and path separators. In particular, notice how you can split a string based on the operating system's path separator and then use Python indexing to extract various parts of the array. **Line 27** then determines the unique set of class labels (in this case, 17 total classes) from the image paths.

Given our `imagePaths`, we can load the Flowers-17 dataset from disk:

```
29 # initialize the image preprocessors
30 aap = AspectAwarePreprocessor(64, 64)
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=[aap, iap])
36 (data, labels) = sdl.load(imagePaths, verbose=500)
37 data = data.astype("float") / 255.0
```

Line 30 initializes our `AspectAwarePreprocessor` such that every image it processes will be 64×64 pixels. The `ImageToArrayPreprocessor` is then initialized on **Line 31**, allowing us to convert images to Keras-compatible arrays. We then instantiate the `SimpleDatasetLoader` using these two preprocessors, respectively (**Line 35**).

The `data` and corresponding `labels` are loaded from disk on **Line 36**. All images in the `data` array are then normalized to the range $[0, 1]$ by dividing the raw pixel intensities by 255.

Now that our data is loaded we can perform a training and testing split (75 percent for training, 25 percent for testing) along with one-hot encoding our labels:

```
39 # partition the data into training and testing splits using 75% of
40 # the data for training and the remaining 25% for testing
41 (trainX, testX, trainY, testY) = train_test_split(data, labels,
42      test_size=0.25, random_state=42)
43
44 # convert the labels from integers to vectors
45 trainY = LabelBinarizer().fit_transform(trainY)
46 testY = LabelBinarizer().fit_transform(testY)
```

To train our flower classifier we'll be using the MiniVGGNet architecture along with the SGD optimizer:

```
48 # initialize the optimizer and model
49 print("[INFO] compiling model...")
50 opt = SGD(lr=0.05)
51 model = MiniVGGNet.build(width=64, height=64, depth=3,
52      classes=len(classNames))
53 model.compile(loss="categorical_crossentropy", optimizer=opt,
54      metrics=["accuracy"])
```

```
55
56 # train the network
57 print("[INFO] training network...")
58 H = model.fit(trainX, trainY, validation_data=(testX, testY),
59     batch_size=32, epochs=100, verbose=1)
```

The MiniVGGNet architecture will accept images with spatial dimensions $64 \times 64 \times 3$ (64 pixels wide, 64 pixels tall, and 3 channels). The total number of classes is `len(classNames)` which, in this case, equals seventeen, one for each of the categories in the Flowers-17 dataset.

We'll train MiniVGGNet using SGD with an initial learning rate of $\alpha = 0.05$. We'll purposely leave out learning rate decay so we can demonstrate the affect data augmentation has in the next section. **Lines 58 and 59** train MiniVGGNet for a total of 100 epochs.

We then evaluate our network and plot our loss and accuracy over time:

```
61 # evaluate the network
62 print("[INFO] evaluating network...")
63 predictions = model.predict(testX, batch_size=32)
64 print(classification_report(testY.argmax(axis=1),
65     predictions.argmax(axis=1), target_names=classNames))
66
67 # plot the training loss and accuracy
68 plt.style.use("ggplot")
69 plt.figure()
70 plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
71 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
72 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
73 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
74 plt.title("Training Loss and Accuracy")
75 plt.xlabel("Epoch #")
76 plt.ylabel("Loss/Accuracy")
77 plt.legend()
78 plt.show()
```

To obtain a baseline accuracy on Flowers-17 using MiniVGGNet, just execute the following command:

```
$ python minivggnet_flowers17.py --dataset ../datasets/flowers17/images
[INFO] loading images...
[INFO] processed 500/1360
[INFO] processed 1000/1360
[INFO] compiling model...
[INFO] training network...
Train on 1020 samples, validate on 340 samples
Epoch 1/100
...
Epoch 100/100
3s - loss: 0.0030 - acc: 1.0000 - val_loss: 1.7683 - val_acc: 0.6206
[INFO] evaluating network...
             precision    recall  f1-score   support

   bluebell       0.48      0.67      0.56        18
  buttercup       0.67      0.60      0.63        20
  coltsfoot       0.53      0.40      0.46        20
```

```
     cowslip       0.35      0.44      0.39        18
      crocus       0.62      0.50      0.56        20
    daffodil       0.43      0.33      0.38        27
       daisy       0.74      0.85      0.79        20
   dandelion       0.61      0.83      0.70        23
  fritillary       0.72      0.82      0.77        22
        iris       0.80      0.76      0.78        21
  lilyvalley       0.59      0.67      0.62        15
       pansy       0.82      0.74      0.78        19
    snowdrop       0.64      0.39      0.49        23
   sunflower       0.95      0.91      0.93        23
   tigerlily       0.88      0.74      0.80        19
       tulip       0.18      0.25      0.21        16
  windflower       0.67      0.62      0.65        16

 avg / total       0.64      0.62      0.62       340
```

As we can see from the output, we are able to obtain 64 percent classification accuracy, which is fairly reasonable given our limited amount of training data. However, what is concerning is our loss and accuracy plot (Figure (2.5). As the plot demonstrates, our network quickly starts overfitting past epoch 20. The reason for this behavior is because we only have 1,020 training examples with 60 images per class (the other images are used for testing). Keep in mind that we should *ideally* have anywhere between 1,000-5,000 examples *per class* when training a Convolutional Neural Network.

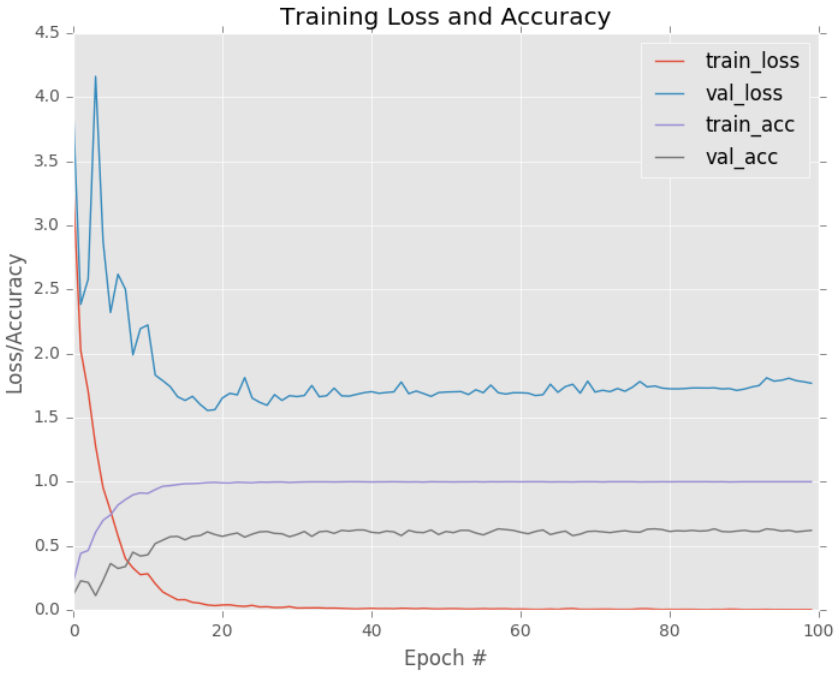


Figure 2.5: Learning plot for MiniVGGNet applied to the Flowers-17 dataset *without* data augmentation. Notice how overfitting starts to occur past epoch 25 as our validation loss increases.

Furthermore, training accuracy skyrockets past 95% in the first few epochs, eventually obtaining 100% accuracy in the later epochs – **this output is a clear case of overfitting**. Due to the lack of substantial training data, MiniVGGNet is modeling the underlying patterns in the training data

too closely and is unable to generalize to the test data. To combat the overfitting, we can apply regularization techniques – in the context of this chapter, our regularization method will be data augmentation. In practice, you would also include other forms of regularization (weight decay, dropout, etc.) to further reduce the effects of overfitting.

2.3.4 Flowers-17: With Data Augmentation

In this example we are going to apply the *exact same training process* as the previous section only with one addition: **we'll be applying data augmentation**. To see how data augmentation can increase our classification accuracy while preventing overfitting, open up a new file, name it `minivggnet_flowers17_data_aug.py`, and let's 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 AspectAwarePreprocessor
7  from pyimagesearch.datasets import SimpleDatasetLoader
8  from pyimagesearch.nn.conv import MiniVGGNet
9  from keras.preprocessing.image import ImageDataGenerator
10 from keras.optimizers import SGD
11 from imutils import paths
12 import matplotlib.pyplot as plt
13 import numpy as np
14 import argparse
15 import os
```

Our imports are the same as in `minivggnet_flowers17.py`, with the exception of **Line 9** where we import the `ImageDataGenerator` class used for data augmentation.

Next, let's parse our command line arguments and extract the class names from the image paths:

```
17 # construct the argument parse and parse the arguments
18 ap = argparse.ArgumentParser()
19 ap.add_argument("-d", "--dataset", required=True,
20     help="path to input dataset")
21 args = vars(ap.parse_args())
22
23 # grab the list of images that we'll be describing, then extract
24 # the class label names from the image paths
25 print("[INFO] loading images...")
26 imagePaths = list(paths.list_images(args["dataset"]))
27 classNames = [pt.split(os.path.sep)[-2] for pt in imagePaths]
28 classNames = [str(x) for x in np.unique(classNames)]
```

As well as load our dataset from disk, construct our training/testing splits, and encode our labels:

```
30 # initialize the image preprocessors
31 aap = AspectAwarePreprocessor(64, 64)
32 iap = ImageToArrayPreprocessor()
33
```

```
# load the dataset from disk then scale the raw pixel intensities
# to the range [0, 1]
sdl = SimpleDatasetLoader(preprocessors=[aap, iap])
(data, labels) = sdl.load(imagePaths, verbose=500)
data = data.astype("float") / 255.0

# partition the data into training and testing splits using 75% of
# the data for training and the remaining 25% for testing
(trainX, testX, trainY, testY) = train_test_split(data, labels,
    test_size=0.25, random_state=42)

# convert the labels from integers to vectors
trainY = LabelBinarizer().fit_transform(trainY)
testY = LabelBinarizer().fit_transform(testY)
```

Our next code block is *very important* as it initializes our `ImageDataGenerator`:

```
# construct the image generator for data augmentation
aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
    height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
    horizontal_flip=True, fill_mode="nearest")
```

Here we'll allow images to be:

1. Randomly rotated ± 30 degrees
2. Horizontally and vertically shifted by a factor of 0.1
3. Sheared by 0.2
4. Zoomed by uniformly sampling in the range `[0.8, 1.2]`
5. Randomly horizontally flipped

Depending on your exact dataset you'll want to tweak these data augmentation values. It's typical to see rotation ranges between $[10, 30]$ depending on your application. Horizontal and vertical shifts normally fall in the range $[0.1, 0.2]$ (same goes for zoom values). Unless horizontally flipping your image changes the class label, you should always include horizontal flipping as well.

Just as in the previous experiment we'll train MiniVGGNet using the SGD optimizer:

```
# initialize the optimizer and model
print("[INFO] compiling model...")
opt = SGD(lr=0.05)
model = MiniVGGNet.build(width=64, height=64, depth=3,
    classes=len(classNames))
model.compile(loss="categorical_crossentropy", optimizer=opt,
    metrics=["accuracy"])
```

However, the code used to train our network has to change slightly as we are now using an *image generator*:

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

Instead of calling the `.fit` method of `model`, we now need to call `.fit_generator`. The first parameter to `.fit_generator` is `aug.flow`, our data augmentation function used to generate new training samples from the training data. The `aug.flow` requires us to pass in our training data and corresponding labels. We also need to supply the batch size so the generator can construct appropriate batches when training the network.

We then supply the `validation_data` as a 2-tuple of `(testX, testY)` – this data is used for validation at the end of every epoch. The `steps_per_epoch` parameter controls *the number of batches per epoch* – we can programmatically determine the proper `steps_per_epoch` value by dividing the total number of training samples by our batch size and casting it to an integer. Finally, `epochs` controls the total number of epochs our network should be trained for, in this case, 100 epochs.

After training our network we'll evaluate it and plot the corresponding accuracy/loss plot:

```
68 # evaluate the network
69 print("[INFO] evaluating network...")
70 predictions = model.predict(testX, batch_size=32)
71 print(classification_report(testY.argmax(axis=1),
72      predictions.argmax(axis=1), target_names=classNames))
73
74 # plot the training loss and accuracy
75 plt.style.use("ggplot")
76 plt.figure()
77 plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
78 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
79 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
80 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
81 plt.title("Training Loss and Accuracy")
82 plt.xlabel("Epoch #")
83 plt.ylabel("Loss/Accuracy")
84 plt.legend()
85 plt.show()
```

Notice how we *do not* apply data augmentation to the validation data. We *only* apply data augmentation to the training set.

To train MiniVGGNet on Flowers-17 *with* data augmentation, just execute the following command:

```
$ python minivggnet_flowers17_data_aug.py \
     --dataset ../datasets/flowers17/images
[INFO] loading images...
[INFO] processed 500/1360
[INFO] processed 1000/1360
[INFO] compiling model...
[INFO] training network...
Epoch 1/100
3s - loss: 3.4073 - acc: 0.2108 - val_loss: 3.0306 - val_acc: 0.1882
...
Epoch 100/100
3s - loss: 0.2769 - acc: 0.9078 - val_loss: 1.3560 - val_acc: 0.6794
[INFO] evaluating network...
             precision    recall  f1-score   support
```

```
          bluebell       0.67      0.44      0.53        18
         buttercup       0.59      0.80      0.68        20
         coltsfoot       0.56      0.50      0.53        20
           cowslip       0.45      0.50      0.47        18
            crocus       0.82      0.45      0.58        20
          daffodil       0.67      0.30      0.41        27
             daisy       1.00      0.95      0.97        20
         dandelion       0.63      0.96      0.76        23
        fritillary       0.94      0.77      0.85        22
              iris       0.78      0.86      0.82        21
        lilyvalley       0.44      0.73      0.55        15
             pansy       1.00      0.74      0.85        19
           snowdrop       0.54      0.65      0.59        23
         sunflower       1.00      0.96      0.98        23
         tigerlily       0.80      0.84      0.82        19
             tulip       0.22      0.25      0.24        16
        windflower       0.72      0.81      0.76        16

       avg / total       0.71      0.68      0.68       340
```

Immediately after the network finishes training, you'll notice an increase of accuracy from 64% to 71%, a 7% improvement from our previous run. However, accuracy isn't everything – the *real* question is whether data augmentation has helped prevent overfitting. To answer that question, we'll need to examine the loss and accuracy plot in Figure 2.6.

Figure 2.6: Applying MiniVGGNet to Flowers-17 *with* data augmentation. Overfitting is still a concern; however, we are able to obtain *substantially* higher classification accuracy and lower loss.

While there is still overfitting occurring, the effect is *significantly dampened* by using data augmentation. Again, keep in mind that these two experiments are *identical* – the only changes

we made were whether or not data augmentation was applied. As a regularizer, you can also see data augmentation having an impact. We were able to *increase* our validation accuracy, thereby improving the generalizability of our model, despite having lowering training accuracy.

Further accuracy can be obtained by decaying the learning rate over time. Learning rate was specifically left out of this chapter so we could focus solely on the impact data augmentation as a regularizer has when training Convolutional Neural Networks.

2.4 Summary

Data augmentation is a type of regularization technique that operates on the training data. As the name suggests, data augmentation randomly jitters our training data by applying a series of random translations, rotations, shears, and flips. Applying these simple transformations does not change the class label of the input image; however, each augmented image can be considered a "new" image that the training algorithm has not seen before. Therefore, our training algorithm is being constantly presented with new training samples, allowing it to learn more robust and discriminative patterns.

As our results demonstrated, applying data augmentation increased our classification accuracy while helping mitigate the effects of overfitting. Furthermore, data augmentation also allowed us to train a Convolutional Neural Network on only *60 samples per class*, far below the suggested 1,000-5,000 samples per class.

While it's always better to gather "natural" training samples, in a pinch, data augmentation can be used to overcome small dataset limitations. When it comes to your own experiments, you should apply data augmentation to *nearly every experiment you run*. There is a slight performance hit you must take due to the fact that the CPU is now responsible for randomly transforming your inputs; however, this performance hit is mitigated by using threading and augmenting your data in the background *before* it is passed to the thread responsible for training your network.

Again, in nearly *all* remaining chapters inside the *Practitioner Bundle* and *ImageNet Bundle*, we'll be using data augmentation. Take the time to familiarize yourself with this technique now as it will help you obtain better performing deep learning models (using less data) quicker.

3. Networks as Feature Extractors

Over the next few chapters, we'll be discussing the concept of *transfer learning*, the ability to use a *pre-trained model* as a "shortcut" to learn patterns from data it was not originally trained on.

Consider a traditional machine learning scenario where we are given two classification challenges. In the first challenge, our goal is to train a Convolutional Neural Network to recognize dogs vs. cats in an image (as we'll do in Chapter 10).

Then, in the second project, we are tasked with recognizing three separate species of bears: *grizzly bears*, *polar bears*, and *giant pandas*. Using standard practices in machine learning, neural networks, and deep learning, we would treat these challenges as *two separate problems*. First, we would gather a sufficient labeled dataset of dogs and cats, followed by training a model on the dataset. We would then repeat the process a second time, only this time, gathering images of our bear breeds, and then training a model on top of the labeled bear dataset.

Transfer learning proposes a different training paradigm – *what if we could use an* ***existing*** *pre-trained classifier and use it as a* ***starting point*** *for a new classification task?* In context of the proposed challenges above, we would first train a Convolutional Neural Network to recognize dogs versus cats. Then, we would use *the same CNN trained on dog and cat data* to be used to distinguish between bear classes, *even though no bear data was mixed with the dog and cat data*.

Does this sound too good to be true? It's actually not. Deep neural networks trained on large-scale datasets such as ImageNet have demonstrated to be *excellent* at the task of transfer learning. These networks learn a set of rich, discriminating features to recognize 1,000 separate object classes. It makes sense that these filters can be *reused* for classification tasks other than what the CNN was originally trained on.

In general, there are two types of transfer learning when applied to deep learning for computer vision:

1. Treating networks as arbitrary feature extractors.
2. Removing the fully-connected layers of an existing network, placing new FC layer set on top of the CNN, and *fine-tuning* these weights (and optionally previous layers) to recognize object classes.

In this chapter, we'll be focusing primarily on the first method of transfer learning, treating

networks as feature extractors. We'll then discuss how to fine-tune the weights of a network to a *specific* classification task in Chapter 5.

3.1 Extracting Features with a Pre-trained CNN

Up until this point, we have treated Convolutional Neural Networks as end-to-end image classifiers:

1. We input an image to the network.
2. The image forward propagates through the network.
3. We obtain the final classification probabilities from the end of the network.

However, there is no "rule" that says we *must* allow the image to forward propagate through the *entire* network. Instead, we can stop the propagation at an arbitrary layer, such as an activation or pooling layer, extract the values from the network at this time, and then use them as feature vectors. For example, let's consider the VGG16 network architecture by Simonyan and Zisserman [11] (Figure 3.1, *left*).

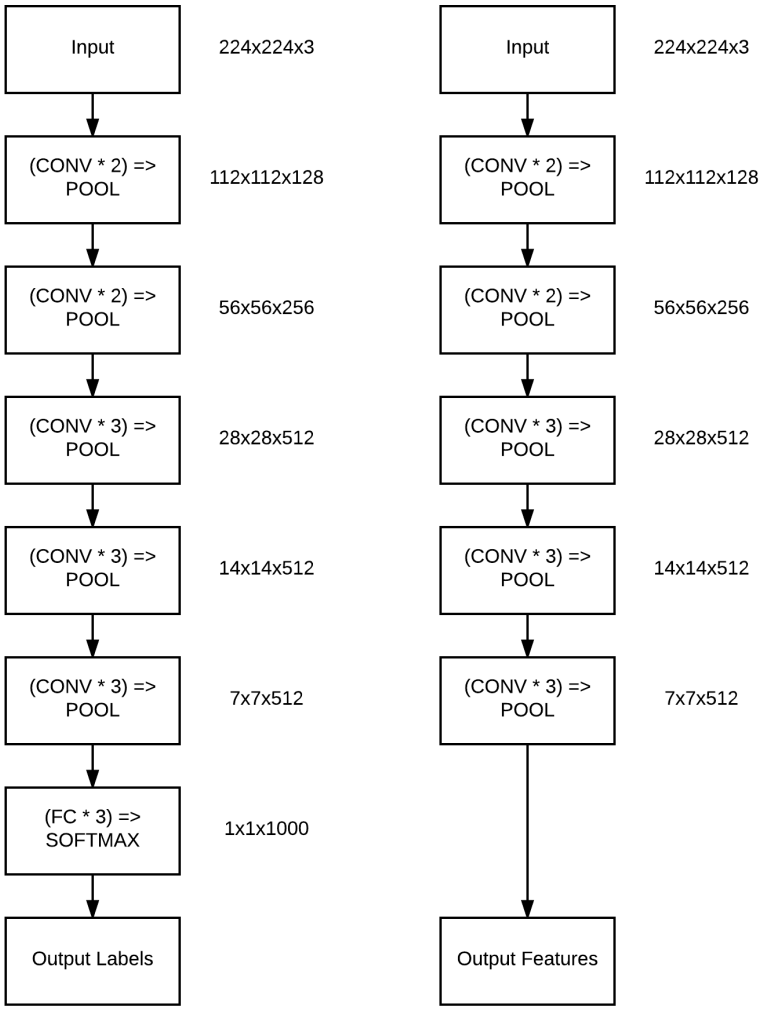


Figure 3.1: **Left:** The original VGG16 network architecture that outputs probabilities for each of the 1,000 ImageNet class labels. **Right:** Removing the `FC` layers from VGG16 and instead returning the output of the final `POOL` layer. This output will serve as our extracted features.

Along with the layers in the network, we have also included the input and output shapes of the

volumes for each layer. When treating networks as a feature extractor, we essentially "chop off" the network at an arbitrary point (normally prior to the fully-connected layers, but it really depends on your particular dataset).

Now the last layer in our network is a max pooling layer (Figure 3.1, *right*) which will have the output shape of $7 \times 7 \times 512$ implying there are 512 filters each of size 7×7. If we were to forward propagate an image through this network with its `FC` head removed, we would be left with 512, 7×7 activations that have either activated or not based on the image contents. Therefore, we can actually take these $7 \times 7 \times 512 = 25,088$ values and treat them as a feature vector that *quantifies the contents of an image*.

If we repeat this process for an entire dataset of images (including datasets that VGG16 was *not* trained on), we'll be left with a design matrix of N images, each with 25,088 columns used to quantify their contents (i.e., feature vectors). Given our feature vectors, we can train an off-the-shelf machine learning model such a Linear SVM, Logistic Regression classifier, or Random Forest on top of these features to obtain a classifier that recognizes new classes of images.

Keep in mind that the CNN *itself* is not capable of recognizing these new classes – instead, we are using the CNN as an intermediary feature extractor. The downstream machine learning classifier will take care of learning the underlying patterns of the features extracted from the CNN.

Later in this chapter, I'll be demonstrating how you can use pre-trained CNNs (specifically VGG16) and the Keras library to obtain $> 95\%$ classification accuracy on image datasets such as Animals, CALTECH-101, and Flowers-17. Neither of these datasets contain images that VGG16 was trained on, but by applying transfer learning, we are able to build *super accurate image classifiers* with little effort. The trick is *extracting* these features and storing them in an efficient manner. To accomplish this task, we'll need HDF5.

3.1.1 What Is HDF5?

HDF5 is binary data format created by the HDF5 group [12] to store gigantic numerical datasets on disk (far too large to store in memory) while facilitating easy access and computation on the rows of the datasets. Data in HDF5 is stored *hierarchically*, similar to how a file system stores data. Data is first defined in *groups*, where a group is a container-like structure which can hold *datasets* and *other groups*. Once a group has been defined, a *dataset* can be created within the group. A *dataset* can be thought of as a multi-dimensional array (i.e., a NumPy array) of a homogeneous data type (integer, float, unicode, etc.). An example of an HDF5 file containing a group with multiple datasets is displayed in Figure 3.2.

HDF5 is written in C; however, by using the `h5py` module (h5py.org), we can gain access to the underlying C API using the Python programming language. What makes `h5py` so awesome is the ease of interaction with data. We can store *huge* amounts of data in our HDF5 dataset and manipulate the data in a NumPy-like fashion. For example, we can use standard Python syntax to access and slice rows from *multi-terabyte datasets* stored on disk as if they were simple NumPy arrays loaded into memory. Thanks to specialized data structures, these slices and row accesses are *lighting quick*. When using HDF5 with `h5py`, you can think of your data as a gigantic NumPy array that is too large to fit into main memory but can still be accessed and manipulated just the same.

Perhaps best of all, the HDF5 format is *standardized*, meaning that datasets stored in HDF5 format are inherently portable and can be accessed by other developers using different programming languages such as C, MATLAB, and Java.

In the rest of this chapter, we'll be writing a custom Python class that allows us to efficiently accept input data and write it to an HDF5 dataset. This class will then serve two purposes:

1. Facilitate a method for us to apply transfer learning by taking our extracted features from VGG16 and writing them to an HDF5 dataset in an efficient manner.
2. Allow us to generate HDF5 datasets from *raw images* to facilitate faster training (Chapter 9).

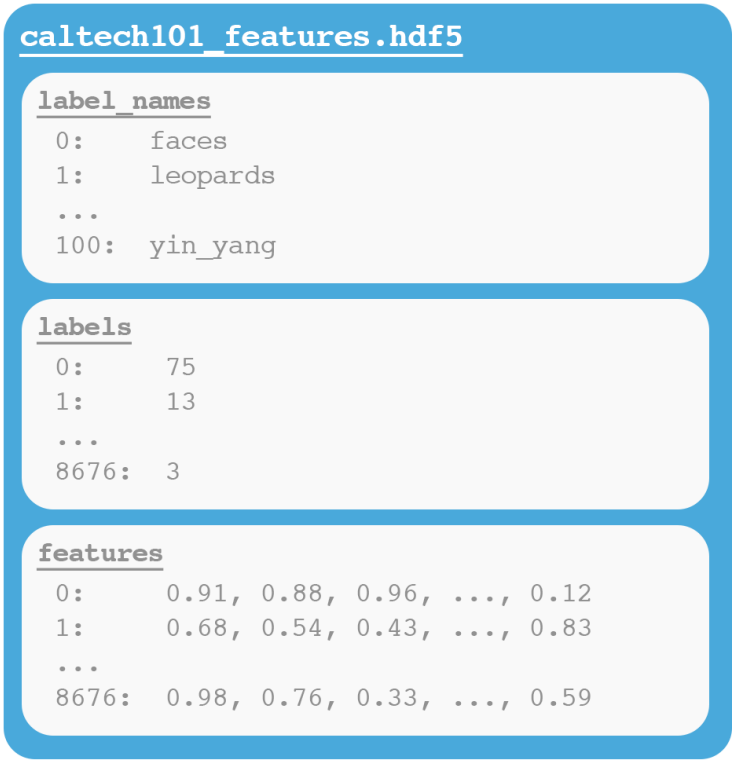


Figure 3.2: An example of a HDF5 file with three datasets. The first dataset contains the `label_names` for CALTECH-101. We then have `labels`, which maps the each image to its corresponding class label. Finally, the `features` dataset contains the image quantifications extracted by the CNN.

If you do not already have HDF5 and `h5py` installed on your system, please see the supplementary material for Chapter 6 of the *Starter Bundle* for instructions to configure your system.

3.1.2 Writing Features to an HDF5 Dataset

Before we can even think about treating VGG16 (or any other CNN) as a feature extractor, we first need to develop a bit of infrastructure. In particular, we need to define a Python class named `HDF5DatasetWriter`, which as the name suggests, is responsible for taking an input set of NumPy arrays (whether features, raw images, etc.) and writing them to HDF5 format. To do so, create a new sub-module in the `pyimagesearch` package named `io` and then place a file named `hdf5datasetwriter.py` inside of `io`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |    |--- __init__.py
|    |    |--- hdf5datasetwriter.py
|    |--- nn
|    |--- preprocessing
|    |--- utils
```

From there, open up `hdf5datasetwriter.py` and we'll get to work:

```
# import the necessary packages
import h5py
import os
```

We'll start off easy with our imports. We only need two Python packages to build the functionality inside this class – the built-in `os` module and `h5py` so we have access to the HDF5 bindings.

From there, let's define the constructor:

```
class HDF5DatasetWriter:
     def __init__(self, dims, outputPath, dataKey="images",
          bufSize=1000):
          # check to see if the output path exists, and if so, raise
          # an exception
          if os.path.exists(outputPath):
               raise ValueError("The supplied `outputPath` already "
                    "exists and cannot be overwritten. Manually delete "
                    "the file before continuing.", outputPath)

          # open the HDF5 database for writing and create two datasets:
          # one to store the images/features and another to store the
          # class labels
          self.db = h5py.File(outputPath, "w")
          self.data = self.db.create_dataset(dataKey, dims,
               dtype="float")
          self.labels = self.db.create_dataset("labels", (dims[0],),
               dtype="int")

          # store the buffer size, then initialize the buffer itself
          # along with the index into the datasets
          self.bufSize = bufSize
          self.buffer = {"data": [], "labels": []}
          self.idx = 0
```

The constructor to `HDF5DatasetWriter` accepts four parameters, two of which are optional. The `dims` parameter controls the *dimension* or *shape* of the data we will be storing in the dataset. Think of `dims` as the `.shape` of a NumPy array. If we were storing the (flattened) raw pixel intensities of the $28 \times 28 = 784$ MNIST dataset, then `dims=(70000, 784)` as there are 70,000 examples in MNIST, each with a dimensionality of 784. If we wanted to store the *raw CIFAR-10 images*, then we would set `dims=(60000, 32, 32, 3)` as there are 60,000 total images in the CIFAR-10 dataset, each represented by a $32 \times 32 \times 3$ RGB image.

In the context of transfer learning and feature extraction, we'll be using the VGG16 architecture and taking the outputs after the final `POOL` layer. The output of the final `POOL` layer is $512 \times 7 \times 7$ which, when flattened, yields a feature vector of length 25,088. Therefore, when using VGG16 for feature extraction, we'll set `dims=(N, 25088)` where `N` is the total number of images in our dataset.

The next parameter to the `HDF5DatasetWriter` constructor is the `outputPath` – this is the path to where our output HDF5 file will be stored on disk. The optional `dataKey` is the *name of the dataset* that will store the data our algorithm will learn from. We default this value to `"images"`,

since in most cases we'll be storing raw images in HDF5 format. However, for this example, when we instantiate the `HDF5DatasetWriter` we'll set `dataKey="features"` to indicate that we are storing features extracted from a CNN in the file.

Finally, `bufSize` controls the size of our in-memory buffer, which we default to 1,000 feature vectors/images. Once we reach `bufSize`, we'll flush the buffer to the HDF5 dataset.

Lines 10-13 then make a check to see if `outputPath` already exists. If it does, we raise an error to the end user (as we don't want to overwrite an existing database).

Line 18 opens the HDF5 file for writing using the supplied `outputPath`. **Lines 19 and 20** create a dataset with the `dataKey` name and the supplied `dims` – this is where we will store our raw images/extracted features. **Lines 21 and 22** create a second dataset, this one to store the (integer) class labels for each record in the dataset **Lines 25-28** then initialize our buffers.

Next, let's review the `add` method used to add data to our buffer:

```
30      def add(self, rows, labels):
31          # add the rows and labels to the buffer
32          self.buffer["data"].extend(rows)
33          self.buffer["labels"].extend(labels)
34
35          # check to see if the buffer needs to be flushed to disk
36          if len(self.buffer["data"]) >= self.bufSize:
37              self.flush()
```

The `add` method requires two parameters: the `rows` that we'll be adding to the dataset, along with their corresponding class `labels`. Both the `rows` and `labels` are added to their respective buffers on **Lines 32 and 33**. If the buffer fills up, we call the `flush` method to write the buffers to file and reset them.

Speaking of the `flush` method, let's define the function now:

```
39      def flush(self):
40          # write the buffers to disk then reset the buffer
41          i = self.idx + len(self.buffer["data"])
42          self.data[self.idx:i] = self.buffer["data"]
43          self.labels[self.idx:i] = self.buffer["labels"]
44          self.idx = i
45          self.buffer = {"data": [], "labels": []}
```

If we think of our HDF5 dataset as a big NumPy array, then we need to keep track of the current index into the next available row where we can store data (without overwriting existing data) – **Line 41** determines the next available row in the matrix. **Lines 42 and 43** then apply NumPy array slicing to store the data and labels in the buffers. **Line 45** then resets the buffers.

We'll also define a handy utility function named `storeClassLabels` which, if called, will store the raw string names of the class labels in a separate dataset:

```
47      def storeClassLabels(self, classLabels):
48          # create a dataset to store the actual class label names,
49          # then store the class labels
50          dt = h5py.special_dtype(vlen=unicode)
51          labelSet = self.db.create_dataset("label_names",
52              (len(classLabels),), dtype=dt)
53          labelSet[:] = classLabels
```

Finally, our last function `close` will be used to write any data left in the buffers to HDF5 as well as close the dataset:

```
55      def close(self):
56          # check to see if there are any other entries in the buffer
57          # that need to be flushed to disk
58          if len(self.buffer["data"]) > 0:
59              self.flush()
60
61          # close the dataset
62          self.db.close()
```

As you can see, the `HDF5DatasetWriter` doesn't have much to do with machine learning or deep learning at all – it's simply a class used to help us store data in HDF5 format. As you continue in your deep learning career, you'll notice that *much* of the initial labor when setting up a new problem is getting the data into a format you can work with. Once you have your data in a format that's straightforward to manipulate, it becomes *substantially easier* to apply machine learning and deep learning techniques to your data.

All that said, since the `HDF5DatasetWriter` class is a utility class non-specific to deep learning and computer vision, I've kept the explanation of the class shorter than the other code examples in this book. If you find yourself struggling to understand this class I would suggest you:

1. Finish reading the rest of this chapter so you can understand how we use it in context of feature extraction.
2. Take the time to educate yourself on some basic Python programming paradigms – I provide a list of Python programming sources I recommend here: http://pyimg.co/ida57.
3. Take apart this class and implement by hand, piece-by-piece, until you understand what is going on under the hood.

Now that our `HDF5DatasetWriter` is implemented, we can move on to actually *extracting features* using pre-trained Convolutional Neural Networks.

3.2 The Feature Extraction Process

Let's define a Python script that can be used to extract features from an arbitrary image dataset (provided the input dataset follows a specific directory structure). Open up a new file, name it `extract_features.py`, and we'll get to work:

```
1  # import the necessary packages
2  from keras.applications import VGG16
3  from keras.applications import imagenet_utils
4  from keras.preprocessing.image import img_to_array
5  from keras.preprocessing.image import load_img
6  from sklearn.preprocessing import LabelEncoder
7  from pyimagesearch.io import HDF5DatasetWriter
8  from imutils import paths
9  import numpy as np
10 import progressbar
11 import argparse
12 import random
13 import os
```

Lines 2-13 import our required Python packages. Notice how on **Line 2** we import the Keras implementation of the pre-trained VGG16 network – this is the architecture we'll be using as our feature extractor. The `LabelEncoder` class on **Line 6** will be used to convert our class labels from *strings* to *integers*. We also import our `HDF5DatasetWriter` on **Line 7** so we can write the features extracted from our CNN to a HDF5 dataset.

One import you haven't seen yet is `progressbar` on **Line 10**. This package has nothing to do with deep learning, but I like to use it for long running tasks as it displays a nicely formatted progress bar to your terminal, as well as provides approximate timings as to when your script will finish executing:

```
Extracting Features  30% |############                              | ETA: 0:00:18
```

If you do not already have `progressbar` installed on your system, you can install it via:

```
$ pip install progressbar
```

Otherwise, you can simply comment out all lines that use `progressbar` (the package is only used for fun, after all).

Let's move on to 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("-o", "--output", required=True,
20      help="path to output HDF5 file")
21 ap.add_argument("-b", "--batch-size", type=int, default=32,
22      help="batch size of images to be passed through network")
23 ap.add_argument("-s", "--buffer-size", type=int, default=1000,
24      help="size of feature extraction buffer")
25 args = vars(ap.parse_args())
```

Our `extract_features.py` script will require two command line arguments, followed by two optional ones. The `--dataset` switch controls the path to our input directory of images that we wish to extract features from. The `--output` switch determines the path to our output HDF5 data file.

We can then supply a `--batch-size` – this is the number of images in a batch that will be passed through VGG16 at a time. A value of 32 is reasonable here, but you can increase it if your machine has sufficient memory. The `--buffer-size` switch controls the number of extracted features we'll store in memory before writing the buffer for our HDF5 dataset. Again, if your machine has sufficient memory, you can increase the buffer size.

The next step is to grab our image paths from disk, shuffle them, and encode the labels:

```
27 # store the batch size in a convenience variable
28 bs = args["batch_size"]
29
30 # grab the list of images that we'll be describing then randomly
31 # shuffle them to allow for easy training and testing splits via
32 # array slicing during training time
```

```
33 print("[INFO] loading images...")
34 imagePaths = list(paths.list_images(args["dataset"]))
35 random.shuffle(imagePaths)
36
37 # extract the class labels from the image paths then encode the
38 # labels
39 labels = [p.split(os.path.sep)[-2] for p in imagePaths]
40 le = LabelEncoder()
41 labels = le.fit_transform(labels)
```

On **Line 34** we grab our `imagePaths`, the filenames for all images in our dataset. We then *purposely* shuffle them on **Line 35**. Why do we bother with the shuffling? Well, keep in mind that in previous examples in this book we computed a training and testing split prior to training our classifier. However, since we'll be working with datasets too large to fit into memory, we won't be able to perform this shuffle *in memory* – therefore, we shuffle the image paths *before* we extract the features. Then, at training time, we can compute the 75 percent index into the HDF5 dataset and use that index as the *end* of the training data and the *start* of the testing data (this point will become more clear in Section 3.3 below).

Line 39 then extracts the class label names from our file paths, assuming our file paths have the directory structure:

```
dataset_name/{class_label}/example.jpg
```

Provided that our dataset *does* follow this directory structure (as all examples in this book do), **Line 39** splits the path into an array based on the path separator (`'/'` on Unix machines and `'\'` on Windows), and then grabs the second-to-last entry in the array – this operation yields the class label of the particular image. Given the `labels`, we then encode them as integers on **Lines 40 and 41** (we'll perform one-hot encoding during the training process).

We can now load the VGG16 network weights and instantiate our `HDF5DatasetWriter`:

```
43 # load the VGG16 network
44 print("[INFO] loading network...")
45 model = VGG16(weights="imagenet", include_top=False)
46
47 # initialize the HDF5 dataset writer, then store the class label
48 # names in the dataset
49 dataset = HDF5DatasetWriter((len(imagePaths), 512 * 7 * 7),
50     args["output"], dataKey="features", bufSize=args["buffer_size"])
51 dataset.storeClassLabels(le.classes_)
```

Line 45 we load the pre-trained VGG16 network from disk; however, notice how we have included the parameter `include_top=False` – supplying this value indicates that the final fully-connected layers should *not* be included in the architecture. Therefore, when forward propagating an image through the network, we'll obtain the *feature values* after the final `POOL` layer rather than the probabilities produced by the softmax classifier in the `FC` layers.

Lines 49 and 50 instantiates the `HDF5DatasetWriter`. The first parameter is our dimensions of the dataset, where there will be `len(imagePaths)` total images, each with a feature vector of size $512 \times 7 \times 7 = 25,088$. **Line 51** then stores the string names of the class labels according to the label encoder.

Now it's time to perform the actual feature extraction:

```
53 # initialize the progress bar
54 widgets = ["Extracting Features: ", progressbar.Percentage(), " ",
55      progressbar.Bar(), " ", progressbar.ETA()]
56 pbar = progressbar.ProgressBar(maxval=len(imagePaths),
57      widgets=widgets).start()
58
59 # loop over the images in batches
60 for i in np.arange(0, len(imagePaths), bs):
61      # extract the batch of images and labels, then initialize the
62      # list of actual images that will be passed through the network
63      # for feature extraction
64      batchPaths = imagePaths[i:i + bs]
65      batchLabels = labels[i:i + bs]
66      batchImages = []
```

Lines 54-57 initialize our progress bar so we can visualize and estimate how long the feature extraction process is going to take. Again, using `progressbar` is optional, so feel free to comment these lines out.

On **Line 60** we start looping over our `imagePaths` in batches of `--batch-size`. **Lines 64 and 65** extract the image paths and labels for the corresponding batch, while **Line 66** initializes a list to store the images about to be loaded and fed into VGG16.

Preparing an image for feature extraction is exactly the same as preparing an image for classification via a CNN:

```
68      # loop over the images and labels in the current batch
69      for (j, imagePath) in enumerate(batchPaths):
70           # load the input image using the Keras helper utility
71           # while ensuring the image is resized to 224x224 pixels
72           image = load_img(imagePath, target_size=(224, 224))
73           image = img_to_array(image)
74
75           # preprocess the image by (1) expanding the dimensions and
76           # (2) subtracting the mean RGB pixel intensity from the
77           # ImageNet dataset
78           image = np.expand_dims(image, axis=0)
79           image = imagenet_utils.preprocess_input(image)
80
81           # add the image to the batch
82           batchImages.append(image)
```

On **Line 69** we loop over each image path in the batch. Each image is loaded from disk and converted to a Keras-compatible array (**Lines 72 and 73**). We then preprocess the `image` on **Lines 78 and 79**, followed by adding it to `batchImages` (**Line 82**).

To obtain our feature vectors for the images in `batchImages`, all we need to do is call the `.predict` method of `model`:

```
84      # pass the images through the network and use the outputs as
85      # our actual features
86      batchImages = np.vstack(batchImages)
87      features = model.predict(batchImages, batch_size=bs)
88
```

```
89        # reshape the features so that each image is represented by
90        # a flattened feature vector of the `MaxPooling2D` outputs
91        features = features.reshape((features.shape[0], 512 * 7 * 7))
92
93        # add the features and labels to our HDF5 dataset
94        dataset.add(features, batchLabels)
95        pbar.update(i)
```

We use the `.vstack` method of NumPy on **Lines 86** to "vertically stack" our images such that they have the shape `(N, 224, 224, 3)` where `N` is the size of the batch.

Passing `batchImages` through our network yields our actual feature vectors – remember, we chopped off the fully-connected layers at the head of VGG16, so now we are left with the values after the final max pooling operation (**Line 87**). However, the output of the `POOL` has the shape `(N, 512, 7, 7)`, implying there are 512 filters, each of size 7×7. To treat these values as a feature vector, we need to flatten them into an array with shape `(N, 25088)`, which is exactly what **Line 91** accomplishes. **Line 94** adds our `features` and `batchLabels` to our HDF5 dataset.

Our final code block handles closing our HDF5 dataset:

```
97  # close the dataset
98  dataset.close()
99  pbar.finish()
```

In the remainder of this section, we are going to practice extracting features using a pre-trained CNN from various datasets.

3.2.1 Extracting Features From Animals

The first dataset we are going to extract features from using VGG16 is our "Animals" dataset. This dataset consists of 3,000 images, of three classes: `dogs`, `cats`, and `pandas`. To utilize VGG16 to extract features from these images, simply execute the following command:

```
$ python extract_features.py --dataset ../datasets/animals/images \
      --output ../datasets/animals/hdf5/features.hdf5
[INFO] loading images...
[INFO] loading network...
Extracting Features: 100% |###################################| Time: 0:00:35
```

Using my Titan X GPU, I was able to extract features from all 3,000 images in approximately 35 seconds. After the script executes, take a look inside your `animals/hdf5` directory and you'll find a file named `features.hdf5`:

```
$ ls ../datasets/animals/hdf5/
features.hdf5
```

To investigate the `.hdf5` file for the Animals dataset, fire up a Python shell:

```
$ python
>>> import h5py
>>> p = "../datasets/animals/hdf5/features.hdf5"
>>> db = h5py.File(p)
>>> list(db.keys())
[u'features', u'label_names', u'labels']
```

Notice how our HDF5 file has three datasets: `features`, `label_names`, and `labels`. The `features` dataset is where our actual extracted features are stored. You can examine the shape of this dataset using the following commands:

```
>>> db["features"].shape
(3000, 25088)
>>> db["labels"].shape
(3000,)
>>> db["label_names"].shape
(3,)
```

Notice how the `.shape` is `(3000, 25088)` – this result implies that each of the 3,000 images in our Animals dataset is quantified via feature vector with length 25,088 (i.e., the values inside VGG16 after the final `POOL` operation). Later in this chapter, we'll learn how we can train a classifier on these features.

3.2.2 Extracting Features From CALTECH-101

Just as we extracted features from the Animals dataset, we can do the same with CALTECH-101:

```
$ python extract_features.py --dataset ../datasets/caltech-101/images \
      --output ../datasets/caltech-101/hdf5/features.hdf5
[INFO] loading images...
[INFO] loading network...
Extracting Features: 100% |###################################| Time: 0:01:27
```

We now have a file named `features.hdf5` in the `caltech-101/hdf5` directory:

```
$ ls ../datasets/caltech-101/hdf5/
features.hdf5
```

Examining this file you'll see that each of the 8,677 images is represented by a 25,088-dim feature vector.

3.2.3 Extracting Features From Flowers-17

Finally, let's apply CNN feature extraction to the Flowers-17 dataset:

```
$ python extract_features.py --dataset ../datasets/flowers17/images \
     --output ../datasets/flowers17/hdf5/features.hdf5
[INFO] loading images...
[INFO] loading network...
Extracting Features: 100% |###################################| Time: 0:00:19
```

Examining the `features.hdf5` file for Flowers-17 you'll see that each of the 1,360 images in the dataset is quantified via a 25,088-dim feature vector.

3.3 Training a Classifier on Extracted Features

Now that we've used a pre-trained CNN to extract features from a handful of datasets, let's see how *discriminative* these features really are, *especially* given that the VGG16 was trained on ImageNet and *not* Animals, CALTECH-101, or Flowers-17.

Take a second now and venture a guess on how good of a job a simple linear model might do at using these features to classify an image – would you guess better than 60% classification accuracy? Does 70% seem unreasonable? Surely 80% is unlikely? And 90% classification accuracy would be unfathomable, right?

Let's find out for ourselves. Open up a new file, name it `train_model.py`, and insert the following code:

```
1  # import the necessary packages
2  from sklearn.linear_model import LogisticRegression
3  from sklearn.model_selection import GridSearchCV
4  from sklearn.metrics import classification_report
5  import argparse
6  import pickle
7  import h5py
```

Lines 2-7 import our required Python packages. The `GridSearchCV` class will be used to help us turn the parameters to our `LogisticRegression` classifier. We'll be using `pickle` to serialize our `LogisticRegression` model to disk after training. Finally, `h5py` will be used so we can interface with our HDF5 dataset of features.

We can now parse our command line arguments:

```
9  # construct the argument parse and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-d", "--db", required=True,
12      help="path HDF5 database")
13 ap.add_argument("-m", "--model", required=True,
14      help="path to output model")
15 ap.add_argument("-j", "--jobs", type=int, default=-1,
16      help="# of jobs to run when tuning hyperparameters")
17 args = vars(ap.parse_args())
```

Our script requires two command line arguments, followed by a third optional one:

1. `--db`: The path to our HDF5 dataset containing our extracted features and class labels.
2. `--model`: Here we supply the path to our output Logistic Regression classifier.
3. `--jobs`: An optional integer used to specify the number of concurrent jobs when running a grid search to tune our hyperparameters to the Logistic Regression model.

Let's open our HDF5 dataset and determine where our training/testing split will be:

```
19 # open the HDF5 database for reading then determine the index of
20 # the training and testing split, provided that this data was
21 # already shuffled *prior* to writing it to disk
22 db = h5py.File(args["db"], "r")
23 i = int(db["labels"].shape[0] * 0.75)
```

As I mentioned earlier in this chapter, we *purposely* shuffled our image paths *prior* to writing the associated images/feature vectors to the HDF5 dataset – the reason why becomes clear on **Lines 22 and 23**.

Given that our dataset is too large to fit into memory, we need an efficient method to determine our training and testing split. Since we know how many entries there are in the HDF5 dataset (and we know we want to use 75% of the data for training and 25% for evaluation), we can simply compute the 75% index `i` into the database. Any data *before* the index `i` is considered training data – anything after `i` is testing data.

Given our training and testing splits, let's train our Logistic Regression classifier:

```
25 # define the set of parameters that we want to tune then start a
26 # grid search where we evaluate our model for each value of C
27 print("[INFO] tuning hyperparameters...")
28 params = {"C": [0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]}
29 model = GridSearchCV(LogisticRegression(solver="lbfgs",
30      multi_class="auto"), params, cv=3, n_jobs=args["jobs"])
31 model.fit(db["features"][:i], db["labels"][:i])
32 print("[INFO] best hyperparameters: {}".format(model.best_params_))
33
34 # evaluate the model
35 print("[INFO] evaluating...")
36 preds = model.predict(db["features"][i:])
37 print(classification_report(db["labels"][i:], preds,
38      target_names=db["label_names"]))
```

Lines 28-31 run a grid search over the parameter `C`, the strictness of the Logistic Regression classifier to determine what the optimal value is. A full detailed review of Logistic Regression is outside the scope of this book, so please see Andrew Ng's notes for a thorough review of the Logistic Regression classifier [13].

Take note of how we indicate the *training data* and *training labels* via array slices:

```
db["features"][:i]
db["labels"][:i]
```

Again, any data *before* index `i` is part of our training set. Once the best hyperparameters are found, we then evaluate the classifier on the testing data (**Lines 36-38**).

Notice here that our *testing data* and *testing labels* are accessed via the array slices:

```
db["features"][i:]
db["labels"][i:]
```

Anything *after* the index `i` is part of our testing set. Even though our HDF5 dataset resides on disk (and is too large to fit into memory), we can still treat it as if it was a NumPy array, which is one of the *huge* advantages of using HDF5 and `h5py` together for deep learning and machine learning tasks.

Finally, we save our `LogisticRegression` model to disk and close the database:

```
40 # serialize the model to disk
41 print("[INFO] saving model...")
```

```
42 f = open(args["model"], "wb")
43 f.write(pickle.dumps(model.best_estimator_))
44 f.close()
45
46 # close the database
47 db.close()
```

Also notice how there is *no specific code* related to *either* of the Animals, CALTECH-101, or Flowers-17 datasets – as long as our input dataset of images conforms to the directory structure detailed in Section 3.2 above, we can use both `extract_features.py` and `train_model.py` to rapidly build robust image classifiers based on features extracted from CNNs. How robust, you ask? Let's let the results do the talking.

3.3.1 Results on Animals

To train a Logistic Regression classifier on the features extracted via the VGG16 network on the Animals dataset, simply execute the following command:

```
$ python train_model.py --db ../datasets/animals/hdf5/features.hdf5 \
    --model animals.cpickle
[INFO] tuning hyperparameters...
[INFO] best hyperparameters: {'C': 0.1}
[INFO] evaluating...
             precision    recall  f1-score   support

       cats       0.96      0.98      0.97       252
       dogs       0.98      0.95      0.97       253
      panda       0.98      1.00      0.99       245

avg / total       0.98      0.98      0.98       750
```

Notice here that we are able to reach **98%** classification accuracy! This number is a *massive* improvement from our previous best of 71% in Chapter 12 of the *Starter Bundle*.

3.3.2 Results on CALTECH-101

These incredible results continue to the CALTECH-101 dataset as well. Execute this command to evaluate the performance of VGG16 features on CALTECH-101:

```
$ python train_model.py \
    --db ../datasets/caltech-101/hdf5/features.hdf5 \
    --model caltech101.cpickle
[INFO] tuning hyperparameters...
[INFO] best hyperparameters: {'C': 1000.0}
[INFO] evaluating...
                precision    recall  f1-score   support

         Faces       1.00      0.98      0.99       114
    Faces_easy       0.98      1.00      0.99       104
      Leopards       1.00      1.00      1.00        44
    Motorbikes       1.00      1.00      1.00       197
...
  windsor_chair       0.92      0.92      0.92        13
```

```
          wrench       0.88      0.78      0.82         9
       yin_yang       1.00      1.00      1.00        11

    avg / total       0.96      0.96      0.96      2170
```

This time we are able to obtain **96%** classification accuracy on 101 separate object categories with *minimal* effort!

3.3.3 Results on Flowers-17

Finally, let's apply the VGG16 features to the Flowers-17 dataset, where previously we struggled to break 71 percent accuracy, even when using data augmentation:

```
$ python train_model.py \
    --db ../datasets/flowers17/hdf5/features.hdf5 \
    --model flowers17.cpickle
[INFO] tuning hyperparameters...
[INFO] best hyperparameters: {'C': 0.1}
[INFO] evaluating...
             precision    recall  f1-score   support

   bluebell       1.00      1.00      1.00        25
  buttercup       0.90      0.78      0.84        23
  coltsfoot       1.00      1.00      1.00        20
    cowslip       0.67      0.95      0.78        19
      crocus       0.94      1.00      0.97        16
   daffodil       0.94      0.77      0.85        22
      daisy       1.00      0.95      0.97        20
  dandelion       1.00      1.00      1.00        18
 fritillary       1.00      0.96      0.98        23
       iris       1.00      0.94      0.97        16
 lilyvalley       0.73      0.94      0.82        17
      pansy       0.95      1.00      0.98        20
   snowdrop       0.95      0.72      0.82        29
  sunflower       0.96      1.00      0.98        24
  tigerlily       1.00      1.00      1.00        12
      tulip       0.76      0.84      0.80        19
 windflower       1.00      0.94      0.97        17

avg / total       0.93      0.92      0.92       340
```

This time we reach **93%** classification accuracy, *much* higher than the 71% before. Clearly, the networks such as VGG are capable of performing transfer learning, encoding their discriminative features into output activations that we can use to train our own custom image classifiers.

3.4 Summary

In this chapter, we started to explore transfer learning, the concept of using a *pre-trained Convolutional Neural Network* to classify class labels *outside* of what it was originally trained on. In general, there are two methods to perform transfer learning when applied to deep learning and computer vision:

1. Treat networks as feature extractors, forward propagating the image until a given layer, and then taking these activations and treating them as feature vectors.

2. Fine-tuning networks by adding a brand-new set of fully-connected layers to the head of the network and tuning these FC layers to recognize new classes (while still using the same underlying CONV filters).

We focused strictly on the *feature extraction* component of transfer learning in this chapter, demonstrating that deep CNNs such as VGG, Inception, and ResNet are capable of acting as *powerful* feature extraction machines, even more powerful than hand-designed algorithms such as HOG [14], SIFT [15], and Local Binary Patterns [16], just to name a few. Whenever approaching a new problem with deep learning and Convolutional Neural Networks, always consider if applying feature extraction will obtain reasonable accuracy – if so, you can skip the network training process entirely, saving you *a ton* of time, effort, and headache.

We'll go through my optimal pathway to apply deep learning techniques such as feature extraction, fine-tuning, and training from scratch in Chapter 8. Until then, let's continue studying transfer learning.

4. Understanding rank-1 & rank-5 Accuracies

Before we get too far in our discussion of advanced deep learning topics (such as transfer learning), let's first take a step back and discuss the concept of **rank-1**, **rank-5**, and **rank-*N*** accuracy. When reading deep learning literature, especially in the computer vision and image classification space, you'll likely encounter the concept of *ranked accuracy*. For example, nearly all papers that present machine learning methods evaluated on the ImageNet dataset present their results in terms of both rank-1 and rank-5 accuracy (we'll find out why both rank-1 and rank-5 accuracy are reported later in this chapter).

What exactly is rank-1 and rank-5 accuracy? And how do they differ from the traditional accuracy (i.e., precision)? In this chapter, we'll discuss ranked accuracy, learn how to implement it, and then apply it to machine learning models trained on the Flowers-17 and CALTECH-101 datasets.

4.1 Ranked Accuracy

Figure 4.1: **Left:** An input image of a frog that our neural network will try to classify. **Right:** An input image of a car.

Ranked accuracy is best explained in terms of an example. Let's suppose we are are evaluating a neural network trained on the CIFAR-10 dataset which includes ten classes: *airplane*, *automobile*,

Class Label	Probability
Airplane	0.0%
Automobile	0.0%
Bird	2.1%
Cat	0.03%
Deer	0.01%
Dog	0.56%
Frog	**97.3%**
Horse	0.0%
Ship	0.0%
Truck	0.0%

Class Label	Probability
Airplane	**1.1%**
Automobile	**38.7%**
Bird	0.0%
Cat	0.5%
Deer	0.0%
Dog	0.4%
Frog	0.11%
Horse	**1.4%**
Ship	**2.39%**
Truck	**55.4%**

Table 4.1: **Left:** Class label probabilities returned by our neural network for Figure 4.1 (*left*). **Right:** Class label probabilities returned by our network for Figure 4.1 (*right*).

bird, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*. Given the following input image (Figure 4.1, *left*) we ask our neural network to compute the *probabilities* for each class label – the neural network then returns the class label probabilities listed in Table 4.1 (*left*).

The class label with the largest probability is *frog* (97.3%) which is indeed the correct prediction. If we were to repeat this process of:

1. **Step #1:** Computing the class label probabilities for each input image in the dataset.
2. **Step #2:** Determining if the ground-truth label is equal to the *predicted class label* with the largest probability.
3. **Step #3:** Tallying the number of times where Step #2 is true.

We would arrive at our rank-1 accuracy. Rank-1 accuracy is, therefore, the percentage of predictions where the top prediction matches the ground-truth label – this is the "standard" type of accuracy we are used to computing: take the total number of correct predictions and divide it by the number of data points in the dataset.

We can then extend this concept to rank-5 accuracy. Instead of caring only about the number one prediction, *we care about the top-5 predictions*. Our evaluation process now becomes:

1. **Step #1:** Compute the class label probabilities for each input image in the dataset.
2. **Step #2:** Sort the *predicted* class label probabilities in descending order, such that labels with *higher probability* are placed at the front of the list.
3. **Step #3:** Determine if the *ground-truth label* exists in the top-5 predicted labels from Step #2.
4. **Step #4:** Tally the number of times where Step #3 is true.

Rank-5 is simply an *extension* to rank-1 accuracy: instead of caring about *only* the #1 prediction from the classifier, we'll take into account the *top-5* predictions from the network. For example, let's again consider an input image that is to be categorized into a CIFAR-10 category based on an arbitrary neural network (Figure 4.1, *right*). After being passed through our network, we obtain the class label probabilities detailed in Table 4.1 (*right*).

Our image is clearly of a *car*; however, our network has reported *truck* as the top prediction – this would be considered an incorrect prediction for rank-1 accuracy. But if we examine the *top-5* predictions by the network, we see that *automobile* is actually the number two prediction, which would be accurate when computing rank-5 accuracy. This approach can easily be extended to arbitrary rank-*N* accuracy as well; however, we normally only compute rank-1 and rank-5 accuracies – which raises the question, *why bother computing rank-5 accuracy at all?*

For the CIFAR-10 dataset, computing the rank-5 accuracy is a bit silly, but for large, challenging

datasets, especially for fine-grained classification, it's often helpful to look at the *top-5* predictions from a given CNN. Perhaps the best example of why we compute rank-1 and rank-5 accuracy can be found in Szegedy et al. [17] where we can see a Siberian husky on the *left* and an Eskimo dog on the *right* (Figure 4.2). Most humans would fail to recognize the difference between the two animals; however, both of these classes are valid labels in the ImageNet dataset.

Figure 4.2: **Left:** Siberian husky. **Right:** Eskimo dog.

When working with large datasets that cover many class labels with similar characteristics, we often examine the rank-5 accuracy as an extension to the rank-1 accuracy to see how our network is performing. In an ideal world our rank-1 accuracy would increase at the same rate as our rank-5 accuracy, but on challenging datasets, this is not always the case.

Therefore, we examine the rank-5 accuracy as well to ensure that our network is still "learning" in later epochs. It may be the case where rank-1 accuracy stagnates towards the end of training, but rank-5 accuracy continues to improve as our network learns more discriminating features (but not discriminative enough to overtake the top #1 predictions). Finally, depending on the image classification challenge (ImageNet being the canonical example), you are *required* to report both your rank-1 and rank-5 accuracies together.

4.1.1 Measuring rank-1 and rank-5 Accuracies

Computing rank-1 and rank-5 accuracy can be accomplished by building a simple utility function. Inside our `pyimagesearch` module we'll add this functionality to the `utils` sub-module by adding a file named `ranked.py`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |--- nn
|    |--- preprocessing
|    |--- utils
|    |    |--- __init__.py
|    |    |--- captchahelper.py
|    |    |--- ranked.py
```

Open up `ranked.py` and we'll define the `rank5_accuracy` function:

```
1  # import the necessary packages
2  import numpy as np
3
4  def rank5_accuracy(preds, labels):
```

```
5       # initialize the rank-1 and rank-5 accuracies
6       rank1 = 0
7       rank5 = 0
```

Line 4 defines our `rank5_accuracy` function. This method accepts two parameters:

- `preds`: An $N \times T$ matrix where N, the number of rows, contains the *probabilities* associated with each class label T.
- `labels`: The ground-truth labels for the images in the dataset.

We then initialize the rank-1 and rank-5 accuracies on **Lines 6 and 7**, respectively.

Let's go ahead and compute the rank-1 and rank-5 accuracies:

```
9       # loop over the predictions and ground-truth labels
10      for (p, gt) in zip(preds, labels):
11          # sort the probabilities by their index in descending
12          # order so that the more confident guesses are at the
13          # front of the list
14          p = np.argsort(p)[::-1]
15
16          # check if the ground-truth label is in the top-5
17          # predictions
18          if gt in p[:5]:
19              rank5 += 1
20
21          # check to see if the ground-truth is the #1 prediction
22          if gt == p[0]:
23              rank1 += 1
```

On **Line 10** we start looping over the predictions and ground-truth class labels for each example in the dataset. **Line 14** sorts the probabilities of the predictions `p` in descending order, such that the indices of the largest probabilities are placed at the *front* of the list. If the ground-truth label exists in the top-5 predictions, we increment our `rank5` variable (**Lines 18 and 19**). If the ground-truth label is equal to the number one position, we increment our `rank1` variable (**Lines 22 and 23**).

Our final code block handles converting `rank1` and `rank5` to percentages by dividing by the total number of `labels`:

```
25      # compute the final rank-1 and rank-5 accuracies
26      rank1 /= float(len(preds))
27      rank5 /= float(len(preds))
28
29      # return a tuple of the rank-1 and rank-5 accuracies
30      return (rank1, rank5)
```

Line 30 returns a 2-tuple of the rank-1 and rank-5 accuracies to the calling function.

4.1.2 Implementing Ranked Accuracy

To demonstrate how to compute rank-1 and rank-5 accuracy for a dataset, let's go back to Chapter 3 where we used a pre-trained Convolutional Neural Network on the ImageNet dataset as a feature extractor. Based on these extracted features we trained a Logistic Regression classifier on the data and evaluated the model. We'll now extend our accuracy reports to include rank-5 accuracy as well.

While we are computing rank-1 and rank-5 accuracy for our Logistic Regression model, keep in mind that both rank-1 and rank-5 accuracy can be computed for *any* machine learning, neural

network, or deep learning model – it is common to run into both of these metrics *outside* of the deep learning community. With all that said, open up a new file, name it `rank_accuracy.py`, and insert the following code:

```
1  # import the necessary packages
2  from pyimagesearch.utils.ranked import rank5_accuracy
3  import argparse
4  import pickle
5  import h5py
```

Lines 2-5 import our required Python packages. We'll be using our newly defined `rank5_accuracy` function to compute the rank-1 and rank-5 accuracies of our predictions, respectively. The `pickle` package is used to load our pre-trained scikit-learn classifier from disk. Finally, `h5py` will be used to interface with our HDF5 database of features extracted from our CNN in Chapter 3.

The next step is to parse our command line arguments:

```
7  # construct the argument parse and parse the arguments
8  ap = argparse.ArgumentParser()
9  ap.add_argument("-d", "--db", required=True,
10     help="path HDF5 database")
11 ap.add_argument("-m", "--model", required=True,
12     help="path to pre-trained model")
13 args = vars(ap.parse_args())
```

Our script will require two arguments: `--db`, which is the path to our HDF5 database of extracted features and `--model`, the path to our pre-trained Logistic Regression classifier.

The next code block handles loading the pre-trained model from disk as well as determining the index of the training and testing split into the HDF5 dataset, assuming that 75% of the data was used for training and 25% for testing:

```
15 # load the pre-trained model
16 print("[INFO] loading pre-trained model...")
17 model = pickle.loads(open(args["model"], "rb").read())
18
19 # open the HDF5 database for reading then determine the index of
20 # the training and testing split, provided that this data was
21 # already shuffled *prior* to writing it to disk
22 db = h5py.File(args["db"], "r")
23 i = int(db["labels"].shape[0] * 0.75)
```

Finally, let's compute our rank-1 and rank-5 accuracies:

```
25 # make predictions on the testing set then compute the rank-1
26 # and rank-5 accuracies
27 print("[INFO] predicting...")
28 preds = model.predict_proba(db["features"][i:])
29 (rank1, rank5) = rank5_accuracy(preds, db["labels"][i:])
30
31 # display the rank-1 and rank-5 accuracies
32 print("[INFO] rank-1: {:.2f}%".format(rank1 * 100))
```

```
33 print("[INFO] rank-5: {:.2f}%".format(rank5 * 100))
34
35 # close the database
36 db.close()
```

Line 28 computes the probabilities for *each class label* for every data point in the testing set. Based on the predicted probabilities and the ground-truth labels of the testing data, we can compute the ranked accuracies on **Line 29**. **Lines 32 and 33** then display the rank-1 and rank-5 to our terminal, respectively.

Please take note that we have coded this example such that it will work with *any* example from Chapter 3 where we extracted features from a CNN and then trained a scikit-learn model on top of the features. Later in the *Practitioner Bundle* and *ImageNet Bundle*, we'll compute the rank-1 and rank-5 accuracies for Convolutional Neural Networks trained from scratch as well.

4.1.3 Ranked Accuracy on Flowers-17

To start, let's compute the rank-1 and rank-5 accuracy for the Flowers-17 dataset:

```
$ python rank_accuracy.py --db ../datasets/flowers17/hdf5/features.hdf5 \
    --model ../chapter03-feature_extraction/flowers17.cpickle
[INFO] loading pre-trained model...
[INFO] predicting...
[INFO] rank-1: 92.06%
[INFO] rank-5: 99.41%
```

On the Flowers-17 dataset, we obtain $\mathbf{92.06}\%$ rank-1 accuracy using a Logistic Regression classifier trained on features extracted from the VGG16 architecture. Examining the rank-5 accuracy we see that our classifier is nearly perfect, obtaining $\mathbf{99.41}\%$ rank-5 accuracy.

4.1.4 Ranked Accuracy on CALTECH-101

Let's try another example, this one on the larger CALTECH-101 dataset:

```
$ python rank_accuracy.py --db ../datasets/caltech-101/hdf5/features.hdf5 \
    --model ../chapter03-feature_extraction/caltech101.cpickle
[INFO] loading pre-trained model...
[INFO] predicting...
[INFO] rank-1: 95.58%
[INFO] rank-5: 99.45%
```

Here we obtain $\mathbf{95.58}\%$ rank-1 accuracy and $\mathbf{99.45}\%$ rank-5 accuracy, a substantial improvement from previous computer vision and machine learning techniques that struggled to break 60% classification accuracy.

4.2 Summary

In this chapter, we reviewed the concept of rank-1 and rank-5 accuracy. Rank-1 accuracy is the number of times our ground-truth label equals our class label with the largest probability. Rank-5 accuracy extends on rank-1 accuracy, allowing it to be a bit more "lenient" – here we compute rank-5 accuracy as the number of times our ground-truth label appears in the *top-5 predicted class labels* with the largest probability.

We typically report rank-5 accuracy on large, challenging datasets such as ImageNet where it is often hard for even humans to correctly label the image. In this case, we'll consider a prediction for our model to be "correct" if the ground-truth label simply exists in its top-5 predictions. As we discussed in Chapter 9 of the *Starter Bundle*, a network that is truly generalizing well will produce contextually similar predictions in its top-5 probabilities.

Finally, keep in mind that rank-1 and rank-5 accuracy are *not* specific to deep learning and image classification – you will often see these metrics in other classification tasks as well.

5. Fine-tuning Networks

In Chapter 3 we learned how to treat a pre-trained Convolutional Neural Network as feature extractor. Using this feature extractor, we forward propagated our dataset of images through the network, extracted the activations at a given layer, and saved the values to disk. A standard machine learning classifier (in this case, Logistic Regression) was then trained on top of the CNN features, exactly as we would do if we were using hand-engineered features such as SIFT [15], HOG [14], LBPs [16], etc. This CNN feature extractor approach, called *transfer learning*, obtained remarkable accuracy, far higher than any of our previous experiments on the Animals, CALTECH-101, or Flowers-17 dataset.

But there is *another* type of transfer learning, one that can actually *outperform* the feature extraction method if you have sufficient data. This method is called *fine-tuning* and requires us to perform "network surgery". First, we take a scalpel and cut off the final set of fully-connected layers (i.e., the "head" of the network) from a pre-trained Convolutional Neural Network, such as VGG, ResNet, or Inception. We then replace the head with a *new* set of fully-connected layers with random initializations. From there *all layers below the head* are frozen so their weights cannot be updated (i.e., the backward pass in backpropagation does not reach them).

We then train the network using a very small learning rate so the new set of `FC` layers can start to learn patterns from the *previously learned* `CONV` layers earlier in the network. Optionally, we may unfreeze the rest of the network and continue training. Applying fine-tuning allows us to apply pre-trained networks to recognize classes that they *were not originally trained on*; furthermore, this method can lead to higher accuracy than feature extraction.

In the remainder of this chapter we'll discuss the fine-tuning in more detail, including network surgery. We'll end by providing an example of applying fine-tuning to the Flowers-17 dataset and outperforming all other approaches we've tried in this book thus far.

5.1 Transfer Learning and Fine-tuning

Fine-tuning is a type of transfer learning. We apply fine-tuning to deep learning models that have *already* been trained on a given dataset. Typically, these networks are state-of-the-art architectures such as VGG, ResNet, and Inception that have been trained on the ImageNet dataset.

As we found out in Chapter 3 on feature extraction, these networks contain rich, discriminative filters that can be used on datasets and class labels *outside* the ones they have already been trained on. However, instead of simply applying feature extraction, we are going to perform network surgery and *modify the actual architecture* so we can re-train parts of the network.

If this sounds like something out of a bad horror movie; don't worry, there won't be any blood and gore – but we will have some fun and learn a lot with our experiments. To understand how fine-tuning works, consider Figure 5.1 (*left*) where we have the layers of the VGG16 network. As we know, the final set of layers (i.e., the "head") are our fully-connected layers along with our softmax classifier. When performing fine-tuning, we actually remove the head from the network, just as in feature extraction (*middle*). However, unlike feature extraction, when we perform *fine-tuning* we actually **build a new fully-connected head** and **place it on top of the original architecture** (*right*).

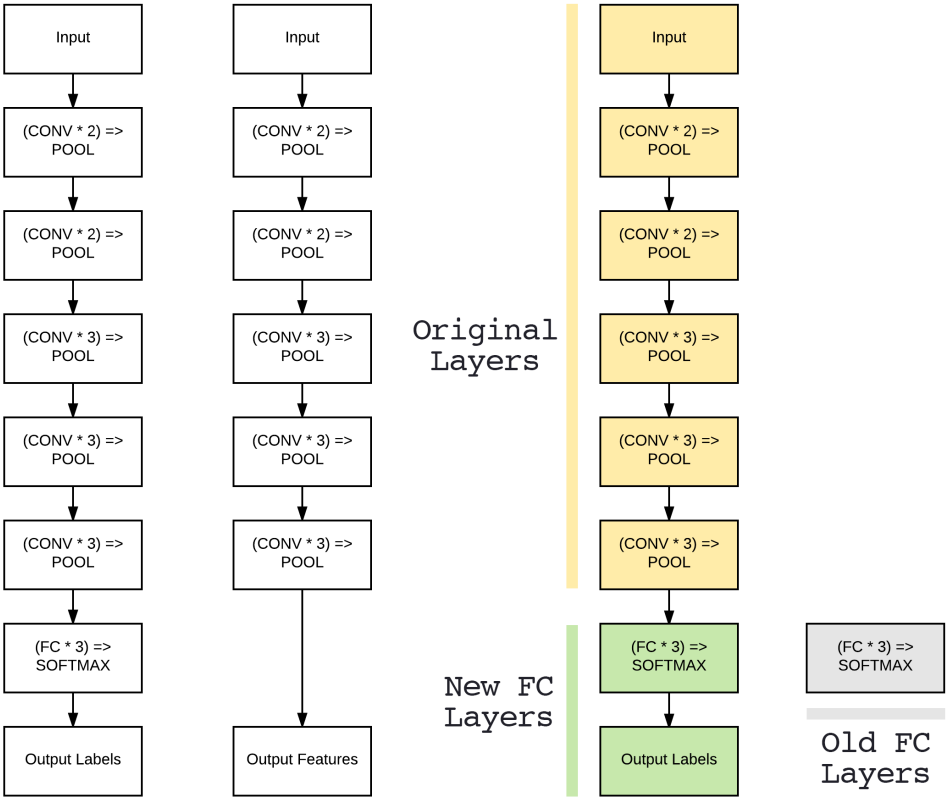


Figure 5.1: **Left:** The original VGG16 network architecture. **Middle:** Removing the `FC` layers from VGG16 and treating the final `POOL` layer as a feature extractor. **Right:** *Removing* the original `FC` layers and *replacing* them with a brand new `FC` head. These new `FC` layers can then be fine-tuned to the specific dataset (the old `FC` layers are no longer used).

In most cases your new `FC` head will have fewer parameters than the original one; however, that really depends on your particular dataset. The new `FC` head is randomly initialized (just like any other layer in a new network) and connected to the body of the original network, and we are ready to train.

However, there is a problem – our `CONV` layers have already learned rich, discriminating filters while our `FC` layers are brand new and totally random. If we allow the gradient to backpropagate from these random values all the way through the body of our network, we risk destroying these

powerful features. To circumvent this, we instead let our FC head "warm up" by (ironically) "freezing" all layers in the body of the network (I told you the cadaver analogy works well here) as in Figure 5.2 (*left*).

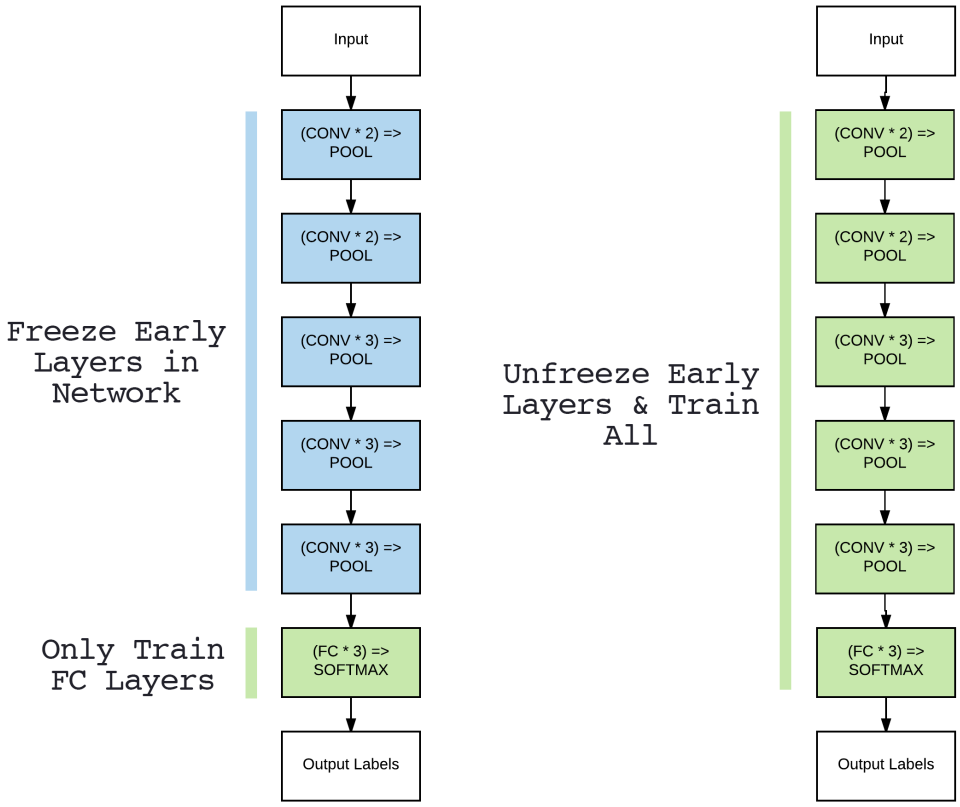


Figure 5.2: **Left:** When we start the fine-tuning process we freeze all CONV layers in the network and only allow the gradient to backpropagate through the FC layers. Doing this allows our network to "warm up". **Right:** After the FC layers have had a chance to warm up we may choose to unfreeze *all* layers in the network and allow each of them to be fine-tuned as well.

Training data is forward propagated through the network as we normally would; however, the backpropagation is stopped after the FC layers, which allows these layers to start to learn patterns from the highly discriminative CONV layers. In some cases, we may never unfreeze the body of the network as our new FC head may obtain sufficient accuracy. However, for some datasets it is often advantageous to allow the original CONV layers to be modified during the fine-tuning process as well (Figure 5.2, *right*).

After the FC head has started to learn patterns in our dataset, pause training, unfreeze the body, and then continue the training, *but with a very small learning rate* – we do not want to deviate our CONV filters dramatically. Training is then allowed to continue until sufficient accuracy is obtained.

Fine-tuning is a *super powerful method* to obtain image classifiers from pre-trained CNNs on custom datasets, even more powerful than feature extraction in most cases. The downside is that fine-tuning can require a bit more work and your choice in FC head parameters does play a big part in network accuracy – you can't rely strictly on regularization techniques here as your network has already been pre-trained and you can't deviate from the regularization already being performed by the network.

Secondly, for small datasets, it can be challenging to get your network to start "learning" from

a "cold" FC start, which is why we freeze the body of the network first. Even still, getting past the warm-up stage can be a bit of a challenge and might require you to use optimizers other than SGD (covered in Chapter 7). While fine-tuning *does* require a bit more effort, if it is done correctly, you'll nearly always enjoy higher accuracy.

5.1.1 Indexes and Layers

Prior to performing network surgery, we need to know the *layer name* and *index* of every layer in a given deep learning model. We need this information as we'll be required to "freeze" and "unfreeze" certain layers in a pre-trained CNN. Without knowing the layer names and indexes ahead of time, we would be "cutting blindly", an out-of-control surgeon with no game plan. If we instead take a few minutes to examine the network architecture and implementation, we can better prepare for our surgery.

Let's go ahead and take a look at the layer names and indexes in VGG16. Open up a new file, name it `inspect_model.py`, and insert the following code:

```
1  # import the necessary packages
2  from keras.applications import VGG16
3  import argparse
4
5  # construct the argument parse and parse the arguments
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--include-top", type=int, default=1,
8       help="whether or not to include top of CNN")
9  args = vars(ap.parse_args())
10
11 # load the VGG16 network
12 print("[INFO] loading network...")
13 model = VGG16(weights="imagenet",
14       include_top=args["include_top"] > 0)
15 print("[INFO] showing layers...")
16
17 # loop over the layers in the network and display them to the
18 # console
19 for (i, layer) in enumerate(model.layers):
20       print("[INFO] {}\t{}".format(i, layer.__class__.__name__))
```

Lines 2 imports our `VGG16` Keras implementation, the network we'll be examining and prepping for surgery. **Lines 6-9** parse our command line arguments. A single switch is needed here `--include-top`, which is used to indicate if the head of the network should be included in the model summary.

Lines 13 and 14 load `VGG16` with pre-trained ImageNet weights from disk – the head of the network is optionally included. Finally, **Lines 19 and 20** allow us to investigate our model.

For each `layer` in the network, we print its corresponding index, `i`. Given this information, we'll know the index of *where* the FC head starts (and where to replace it with our new FC head).

To investigate the VGG16 architecture, just execute the following command:

```
$ python inspect_model.py
[INFO] showing layers...
[INFO] 0     InputLayer
[INFO] 1     Conv2D
[INFO] 2     Conv2D
```

```
[INFO] 3     MaxPooling2D
[INFO] 4     Conv2D
[INFO] 5     Conv2D
[INFO] 6     MaxPooling2D
[INFO] 7     Conv2D
[INFO] 8     Conv2D
[INFO] 9     Conv2D
[INFO] 10     MaxPooling2D
[INFO] 11     Conv2D
[INFO] 12     Conv2D
[INFO] 13     Conv2D
[INFO] 14     MaxPooling2D
[INFO] 15     Conv2D
[INFO] 16     Conv2D
[INFO] 17     Conv2D
[INFO] 18     MaxPooling2D
[INFO] 19     Flatten
[INFO] 20     Dense
[INFO] 21     Dense
[INFO] 22     Dense
```

Here we can see that Layers 20-22 are our fully-connected layers. To verify, let's re-execute the `inspect_model.py` script, this time supplying the switch `--include-top -1` which will leave off the FC head:

```
$ python inspect_model.py --include-top -1
[INFO] showing layers...
[INFO] 0     InputLayer
[INFO] 1     Conv2D
[INFO] 2     Conv2D
[INFO] 3     MaxPooling2D
[INFO] 4     Conv2D
[INFO] 5     Conv2D
[INFO] 6     MaxPooling2D
[INFO] 7     Conv2D
[INFO] 8     Conv2D
[INFO] 9     Conv2D
[INFO] 10     MaxPooling2D
[INFO] 11     Conv2D
[INFO] 12     Conv2D
[INFO] 13     Conv2D
[INFO] 14     MaxPooling2D
[INFO] 15     Conv2D
[INFO] 16     Conv2D
[INFO] 17     Conv2D
[INFO] 18     MaxPooling2D
```

Notice how the final layer in the network is now the POOL layer (just like in Chapter 3 on feature extraction). This body of the network will serve as a starting point for fine-tuning.

5.1.2 Network Surgery

Before we can replace the head of a pre-trained CNN, *we need something to replace it with* – therefore, we need to define our own fully-connected head of the network. To start, create a new file named `fcheadnet.py` in the `nn.conv` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- lenet.py
|    |    |    |--- minivggnet.py
|    |    |    |--- fcheadnet.py
|    |    |    |--- shallownet.py
|    |--- preprocessing
|    |--- utils
```

Then, open up `fcheadnet.py` and insert the following code:

```
1  # import the necessary packages
2  from keras.layers.core import Dropout
3  from keras.layers.core import Flatten
4  from keras.layers.core import Dense
```

Lines 2-4 import our required Python packages. As you can see, these three packages are typically only used for fully-connected networks (the exception being the dropout layer).

Next, let's define the `FCHeadNet` class:

```
6  class FCHeadNet:
7      @staticmethod
8      def build(baseModel, classes, D):
9          # initialize the head model that will be placed on top of
10         # the base, then add a FC layer
11         headModel = baseModel.output
12         headModel = Flatten(name="flatten")(headModel)
13         headModel = Dense(D, activation="relu")(headModel)
14         headModel = Dropout(0.5)(headModel)
15
16         # add a softmax layer
17         headModel = Dense(classes, activation="softmax")(headModel)
18
19         # return the model
20         return headModel
```

Just as in our previous network implementations, we define the `build` method responsible for constructing the actual network architecture. This method requires three parameters: the `baseModel` (the body of the network), the total number of `classes` in our dataset, and finally `D`, the number of nodes in the fully-connected layer.

Line 11 initializes the `headModel` which is responsible for connecting our network with the rest of the body, `baseModel.output`. From there **Lines 12-17** build a very simple fully-connected architecture of:

```
INPUT => FC => RELU => DO => FC => SOFTMAX
```

Again, this fully-connected head is very simplistic compared to the *original* head from VGG16 which consists of two sets of 4,096 FC layers. However, for most fine-tuning problems you are not seeking to *replicate* the original head of the network, but rather *simplify it* so it is easier to fine-tune – the fewer parameters in the head, the more likely we'll be to correctly tune the network to a new classification task. Finally, **Line 20** returns the newly constructed FC head to the calling function.

As we'll see in the next section, we'll be replacing the head of VGG16 with our newly defined FCHeadNet via network surgery.

5.1.3 Fine-tuning, from Start to Finish

It is now time to apply fine-tuning from start to finish. Open up a new file, name it finetune_flowers17.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 AspectAwarePreprocessor
7  from pyimagesearch.datasets import SimpleDatasetLoader
8  from pyimagesearch.nn.conv import FCHeadNet
9  from keras.preprocessing.image import ImageDataGenerator
10 from keras.optimizers import RMSprop
11 from keras.optimizers import SGD
12 from keras.applications import VGG16
13 from keras.layers import Input
14 from keras.models import Model
15 from imutils import paths
16 import numpy as np
17 import argparse
18 import os
```

Lines 2-18 require importing our Python packages, more packages that we have seen before in our previous examples (although many of them we are already familiar with). **Lines 5-7** import our image preprocessors along with a our dataset load. **Line 8** imports our newly defined FCHeadNet to replace the head of VGG16 (**Line 12**). Importing the ImageDataGenerator class on **Line 9** implies that we'll be applying data augmentation to our dataset.

Lines 10 and 11 import our optimizers required for our network to actually learn patterns from the input data. We're already quite familiar with SGD, but we haven't yet covered RMSprop – we'll save a discussion on advanced optimization techniques until Chapter 7, but for the time being simply understand that RMSprop is frequently used in situations where we need to *quickly* obtain reasonable performance (as is the case when we are trying to "warm up" a set of FC layers).

Lines 13 and 14 import two classes required when applying fine-tuning with Keras – Input and Model. We'll need both of these when performing network surgery.

Let's go ahead and parse our command line arguments:

```
20 # construct the argument parse and parse the arguments
21 ap = argparse.ArgumentParser()
22 ap.add_argument("-d", "--dataset", required=True,
23     help="path to input dataset")
24 ap.add_argument("-m", "--model", required=True,
```

```
25      help="path to output model")
26  args = vars(ap.parse_args())
```

We'll require two command line arguments for our script, `--dataset`, the path to the input directory containing the Flowers-17 dataset, and `--model`, the path to our output serialized weights after training.

We can also initialize `ImageDataGenerator`, responsible for performing data augmentation when training our network:

```
28  # construct the image generator for data augmentation
29  aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
30      height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
31      horizontal_flip=True, fill_mode="nearest")
```

As I mentioned in Chapter 2, in nearly all cases you should be applying data augmentation as it rarely hurts accuracy and often helps increase it and avoid overfitting. The same is *especially true* for fine-tuning when we might not have enough data to train a deep CNN from scratch.

The next code block handles grabbing the `imagePaths` from disk along with parsing the `classNames` from the file paths:

```
33  # grab the list of images that we'll be describing, then extract
34  # the class label names from the image paths
35  print("[INFO] loading images...")
36  imagePaths = list(paths.list_images(args["dataset"]))
37  classNames = [pt.split(os.path.sep)[-2] for pt in imagePaths]
38  classNames = [str(x) for x in np.unique(classNames)]
```

Again, we make the assumption that our input dataset has the following directory structure:

```
dataset_name/{class_name}/example.jpg
```

Therefore, we can use the path separator to easily (and conveniently) extract the class label from the file path.

We are now ready to load our image dataset from disk:

```
40  # initialize the image preprocessors
41  aap = AspectAwarePreprocessor(224, 224)
42  iap = ImageToArrayPreprocessor()
43
44  # load the dataset from disk then scale the raw pixel intensities to
45  # the range [0, 1]
46  sdl = SimpleDatasetLoader(preprocessors=[aap, iap])
47  (data, labels) = sdl.load(imagePaths, verbose=500)
48  data = data.astype("float") / 255.0
```

Lines 41 and 42 initialize our image preprocessors. We'll be resizing all input images to 224×224 pixels (maintaining the original aspect ratio of the image), the required input size for the VGG16 network. **Lines 46 and 47** then apply our image preprocessors to load the `data` and `labels` from disk.

Next, let's create our training and testing splits (75% of the data for training, 25% for testing) and one-hot encode the labels:

```
50 # partition the data into training and testing splits using 75% of
51 # the data for training and the remaining 25% for testing
52 (trainX, testX, trainY, testY) = train_test_split(data, labels,
53      test_size=0.25, random_state=42)
54
55 # convert the labels from integers to vectors
56 trainY = LabelBinarizer().fit_transform(trainY)
57 testY = LabelBinarizer().transform(testY)
```

Here comes the fun part – performing network surgery:

```
59 # load the VGG16 network, ensuring the head FC layer sets are left
60 # off
61 baseModel = VGG16(weights="imagenet", include_top=False,
62      input_tensor=Input(shape=(224, 224, 3)))
63
64 # initialize the new head of the network, a set of FC layers
65 # followed by a softmax classifier
66 headModel = FCHeadNet.build(baseModel, len(classNames), 256)
67
68 # place the head FC model on top of the base model -- this will
69 # become the actual model we will train
70 model = Model(inputs=baseModel.input, outputs=headModel)
```

Lines 61 and 62 load the VGG16 architecture from disk using the supplied, pre-trained ImageNet weights. We purposely leave off the head of VGG16 as we'll be replacing it with our own `FCHeadNet`. We also want to *explicitly define* the `input_tensor` to be $224 \times 224 \times 3$ pixels (again, assuming channeling ordering) otherwise we'll run into errors when trying to train our network as the shapes of the volumes will not match up.

Line 66 instantiates the `FCHeadNet` using the `baseModel` body as input, `len(classNames)` as the total number of class labels (17 in the case of Flowers-17), along with 256 nodes in the `FC` layer.

The actual "surgery" is performed on **Line 70** where we construct a new model using the body of VGG16 (`baseModel.input`) as the `input` and the `headModel` as the `output`. However, we're not ready to train our network yet – keep in mind that earlier in this chapter, I mentioned we need to freeze the weights in the body so they are not updated during the backpropagation phase.

We can accomplish this freezing by setting the `.trainable` parameter to `False` for every layer in `baseModel`:

```
72 # loop over all layers in the base model and freeze them so they
73 # will *not* be updated during the training process
74 for layer in baseModel.layers:
75      layer.trainable = False
```

Now that we've connected the head to the body and frozen the layers in the body, we can warm up the new head of the network:

```
77 # compile our model (this needs to be done after our setting our
78 # layers to being non-trainable
79 print("[INFO] compiling model...")
```

```
80 opt = RMSprop(lr=0.001)
81 model.compile(loss="categorical_crossentropy", optimizer=opt,
82     metrics=["accuracy"])
83
84 # train the head of the network for a few epochs (all other
85 # layers are frozen) -- this will allow the new FC layers to
86 # start to become initialized with actual "learned" values
87 # versus pure random
88 print("[INFO] training head...")
89 model.fit_generator(aug.flow(trainX, trainY, batch_size=32),
90     validation_data=(testX, testY), epochs=25,
91     steps_per_epoch=len(trainX) // 32, verbose=1)
```

Line 80 initializes the RMSprop optimizer, an algorithm we'll discuss more in Chapter 7. Notice how we are using a small learning rate of $1e-3$ to warm up the `FC` head. When applying fine-tuning you'll *nearly always* use a learning rate that is one, *if not multiple*, orders of magnitude smaller than the original learning rate used to train the network.

Lines 88-91 then train our new `FC` head using our data augmentation method. Again, keep in mind that while each image is being *fully forward propagated*, the gradients are only being *partially backpropagated* – the backpropagation ends after the `FC` layers, as our goal here is to only "warm-up" the head and *not* change the weights in the body of the network. Here we allow the warm-up phase to train for 25 epochs. Typically you'll allow your own `FC` head to warmup for 10-30 epochs, depending on your dataset.

After the warm-up phase, we'll pause to evaluate network performance on the testing set:

```
93 # evaluate the network after initialization
94 print("[INFO] evaluating after initialization...")
95 predictions = model.predict(testX, batch_size=32)
96 print(classification_report(testY.argmax(axis=1),
97     predictions.argmax(axis=1), target_names=classNames))
```

The above code will allow us to compare the effects of fine-tuning *before* and *after* allowing the head to warm up.

Now that our `FC` layers have been partly trained and initialized, let's unfreeze *some* of the `CONV` layers in the body and make them trainable:

```
99 # now that the head FC layers have been trained/initialized, lets
100 # unfreeze the final set of CONV layers and make them trainable
101 for layer in baseModel.layers[15:]:
102     layer.trainable = True
```

Making a given layer in the body trainable again is an example of setting the parameter `.trainable` to `True` for the given layer. In some cases you'll want to allow the *entire body* to be trainable; however, for deeper architectures with many parameters such as VGG, I suggest only unfreezing the top `CONV` layers and then continuing training. If classification accuracy continues to improve (without overfitting), you may want to consider unfreezing *more* layers in the body.

At this point we should have a warm start to training, so we'll switch over to SGD (again with a small learning rate) and continue training:

```
104 # for the changes to the model to take affect we need to recompile
105 # the model, this time using SGD with a *very* small learning rate
```

```
106 print("[INFO] re-compiling model...")
107 opt = SGD(lr=0.001)
108 model.compile(loss="categorical_crossentropy", optimizer=opt,
109     metrics=["accuracy"])
110
111 # train the model again, this time fine-tuning *both* the final set
112 # of CONV layers along with our set of FC layers
113 print("[INFO] fine-tuning model...")
114 model.fit_generator(aug.flow(trainX, trainY, batch_size=32),
115     validation_data=(testX, testY), epochs=100,
116     steps_per_epoch=len(trainX) // 32, verbose=1)
```

This time we permit our network to train over 100 epochs, allowing the CONV filters to adapt to the underlying patterns in the training data.

Finally, we can evaluate our fine-tuned network as well as serialize the weights to disk:

```
118 # evaluate the network on the fine-tuned model
119 print("[INFO] evaluating after fine-tuning...")
120 predictions = model.predict(testX, batch_size=32)
121 print(classification_report(testY.argmax(axis=1),
122     predictions.argmax(axis=1), target_names=classNames))
123
124 # save the model to disk
125 print("[INFO] serializing model...")
126 model.save(args["model"])
```

To perform network surgery and fine-tune VGG16 on the Flowers-17 dataset, just execute the following command:

```
$ python finetune_flowers17.py --dataset ../datasets/flowers17/images \
      --model flowers17.model
[INFO] loading images...
[INFO] processed 500/1360
[INFO] processed 1000/1360
[INFO] compiling model...
[INFO] training head...
Epoch 1/25
10s - loss: 4.8957 - acc: 0.1510 - val_loss: 2.1650 - val_acc: 0.3618
...
Epoch 10/25
10s - loss: 1.1318 - acc: 0.6245 - val_loss: 0.5132 - val_acc: 0.8441
...
Epoch 23/25
10s - loss: 0.7203 - acc: 0.7598 - val_loss: 0.4679 - val_acc: 0.8529
Epoch 24/25
10s - loss: 0.7355 - acc: 0.7520 - val_loss: 0.4268 - val_acc: 0.8853
Epoch 25/25
10s - loss: 0.7504 - acc: 0.7598 - val_loss: 0.3981 - val_acc: 0.8971
[INFO] evaluating after initialization...
             precision    recall  f1-score   support

   bluebell       0.75      1.00      0.86        18
  buttercup       0.94      0.85      0.89        20
```

```
  coltsfoot       0.94      0.85      0.89        20
    cowslip       0.70      0.78      0.74        18
     crocus       1.00      0.80      0.89        20
   daffodil       0.87      0.96      0.91        27
      daisy       0.90      0.95      0.93        20
  dandelion       0.96      0.96      0.96        23
 fritillary       1.00      0.86      0.93        22
       iris       1.00      0.95      0.98        21
 lilyvalley       0.93      0.93      0.93        15
      pansy       0.83      1.00      0.90        19
  snowdrop        0.88      0.96      0.92        23
 sunflower        1.00      0.96      0.98        23
 tigerlily        0.90      1.00      0.95        19
      tulip       0.86      0.38      0.52        16
 windflower       0.83      0.94      0.88        16

avg / total       0.90      0.90      0.89       340
```

Notice how our initial accuracy is *extremely low* for the first epoch ($\approx 36\%$) during the warm up phase. This result is due to the fact that the `FC` layers in our new head are randomly initialized and still trying to learn the patterns from the previously trained `CONV` filters. However, accuracy quickly rises – by epoch 10 we are above 80% classification accuracy, and by the end of epoch 25 we have reached almost 90% accuracy.

Now that our `FCHeadNet` has obtained a warm start, we switch over to SGD and unfreeze the first set of `CONV` layers in the body, allowing the network to train for another 100 epochs. Accuracy continues to improve, all the way to *95%* classification accuracy, higher than the 93% we obtained using feature extraction:

```
1  ...
2  [INFO] re-compiling model...
3  [INFO] fine-tuning model...
4  Epoch 1/100
5  12s - loss: 0.5127 - acc: 0.8147 - val_loss: 0.3640 - val_acc: 0.8912
6  ...
7  Epoch 99/100
8  12s - loss: 0.1746 - acc: 0.9373 - val_loss: 0.2286 - val_acc: 0.9265
9  Epoch 100/100
10 12s - loss: 0.1845 - acc: 0.9402 - val_loss: 0.2019 - val_acc: 0.9412
11 [INFO] evaluating after fine-tuning...
12             precision    recall  f1-score   support
13
14    bluebell       0.94      0.94      0.94        18
15   buttercup       0.95      1.00      0.98        20
16   coltsfoot       1.00      0.90      0.95        20
17     cowslip       0.85      0.94      0.89        18
18      crocus       0.90      0.90      0.90        20
19    daffodil       1.00      0.78      0.88        27
20       daisy       1.00      0.95      0.97        20
21   dandelion       0.96      1.00      0.98        23
22  fritillary       1.00      0.95      0.98        22
23        iris       1.00      0.95      0.98        21
24  lilyvalley       1.00      0.93      0.97        15
25       pansy       1.00      1.00      1.00        19
26    snowdrop       0.92      0.96      0.94        23
```

```
27     sunflower       0.96      1.00      0.98        23
28     tigerlily       0.90      1.00      0.95        19
29         tulip       0.70      0.88      0.78        16
30    windflower       0.94      0.94      0.94        16
31
32   avg / total       0.95      0.94      0.94       340
```

Additional accuracy can be obtained by performing more aggressive data augmentation and continually unfreezing more and more CONV blocks in VGG16. While fine-tuning is certainly more work than feature extraction, it also enables us to tune and modify the weights in our CNN to a particular dataset – something that feature extraction does not allow. Thus, when given enough training data, consider applying fine-tuning as you'll likely obtain higher classification accuracy than simple feature extraction alone.

5.2 Summary

In this chapter, we discussed the second type of transfer learning, *fine-tuning*. Fine-tuning works by replacing the fully-connected head of a network with a new, randomly initialized head. The layers in the body of the original network are frozen while we train the new FC layers.

Once our network starts to obtain reasonable accuracy, implying that the FC layers have started to learn patterns from both (1) the underlying training data and (2) the previously trained CONV filters earlier in the network, we unfreeze part (if not all) of the body – training is then allowed to continue.

Applying fine-tuning is an *extremely powerful technique* as we do not have to train an *entire* network from scratch. Instead, we can leverage pre-existing network architectures, such as state-of-the-art models trained on the ImageNet dataset which consist of a rich, discriminative set of filters. Using these filters, we can "jump start" our learning, allowing us to perform network surgery, which ultimately leads to a higher accuracy transfer learning model with less effort (and headache) than training from scratch.

For more practical examples of transfer learning and fine-tuning, be sure to refer to the *ImageNet Bundle* where I demonstrate how to:

1. Recognize the make and model of a vehicle.
2. Automatically identify and correct image orientation.

6. Improving Accuracy with Ensembles

In this chapter, we'll explore the concept of ***ensemble methods***, the process of *taking multiple classifiers* and aggregating them into *one big meta-classifier*. By averaging multiple machine learning models together, we can outperform (i.e., achieve higher accuracy) using just a *single* model chosen at random. In fact, nearly all state-of-the-art publications you read that compete in the ImageNet challenge report their *best findings* over ***ensembles of Convolutional Neural Networks***.

We'll start with this chapter with a discussion on Jensen's Inequality – the theory ensemble methods hinge on. From there I'll demonstrate how to train multiple CNNs from a single script and evaluate their performance. We'll then combine these CNNs into a single meta-classifier and notice an increase in accuracy.

6.1 Ensemble Methods

The term "ensemble methods" generally refers to training a "large" number of models (where the exact value of "large" depends on the classification task) and then combining their output predictions via voting or averaging to yield an increase in classification accuracy. In fact, ensemble methods are hardly specific to deep learning and Convolutional Neural Networks. We've been using ensemble methods for *years*. Techniques such as AdaBoost [18] and Random Forests [19] are the quintessential examples of ensemble methods.

In Random Forests, we train multiple Decision Trees [20, 21] and use our forest to make predictions. As you can see from Figure 6.1, our Random Forest consists of *multiple decision trees* aggregated together. Each decision tree "votes" on what it thinks the final classification should be. These votes are tabulated by the meta-classifier, and the category with the most votes is chosen as the final classification.

The same concept can be applied to deep learning and Convolutional Neural Networks. Here we train *multiple networks* and then ask each network to return the *probabilities* for each class label given an input data point (Figure 6.2, *left*). These probabilities are averaged together, and the final classification is obtained. To understand why averaging predictions over multiple models works, we first need to discuss Jensen's Inequality. We'll then provide Python and Keras code to implement an ensemble of CNNs and see for ourselves that classification accuracy does indeed increase.

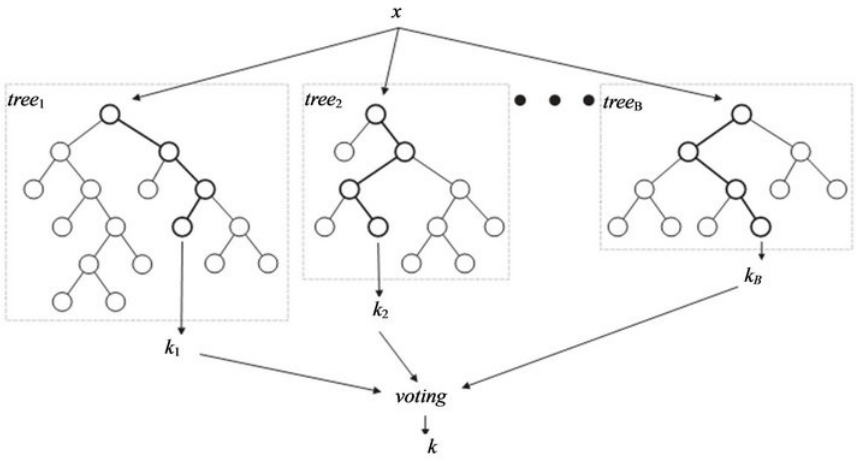


Figure 6.1: A Random Forest consists of multiple decision trees. The outputs of each decision tree are averaged together to obtain the final classification. Image reproduced from Nguyen et al. [22]

6.1.1 Jensen's Inequality

In the most general terms, an ensemble is a finite collection of models that can be used to obtain better average predictive accuracy than using a single model in the ensemble collection. The seminal work of Dietterich [23] details the theory of why ensemble methods can typically obtain higher accuracy than a single model alone.

Dietterich's work hinges on *Jensen's Inequality*, which is known as the "diversity" or the "ambiguity decomposition" in machine learning literature. The formal definition of Jensen's Inequality states that the convex combined (average) ensemble will have error less than or equal to the average error of the individual models. It may be that one individual model has a lower error than the average of all models, but since there is no criterion that we can use to "select" this model, we can be assured that the average of all models will perform *no worse* than selecting any single model at random. In short, we can only get better by averaging our predictions together; we don't have to fear making our classifier worse.

For those of us who enjoy visual examples, perhaps Jensen's Inequality and the concept of model averaging is best explained by asking you to look at this jar of candies and *guess* how many candies are inside (Figure 6.2, *right*).

How many candies would you guess? 100? 200? 500? Your guess might be extremely above or below the actual number of candies in the jar. It could be very close. Or if you're *very lucky*, you might guess the *exact* number of candies.

However, there is a little trick to this game – and it's based on Jensen's Inequality. If you were to ask me how many candies are in the jar, I would go around to you and everyone else who purchased a copy of *Deep Learning for Computer Vision with Python* and ask each of them what they thought the candy count is. **I would then take all of these guesses and average them together – *and I would use this average as my final prediction.***

Now, it may be that a handful of you are *really good guessers* and can beat the average; however, I don't have any criterion to determine which of you are really good guessers. Since I cannot tell who are the best guessers, I'll instead take the average of everyone I ask – and thereby I'm guaranteed to do no worse (on average) than selecting any one of your guesses at random. I may not win the candy guessing game each time we play, but I'll always be in the running; and that, in essence, is Jensen's Inequality.

The difference between randomly guessing candy counts and deep learning models is that

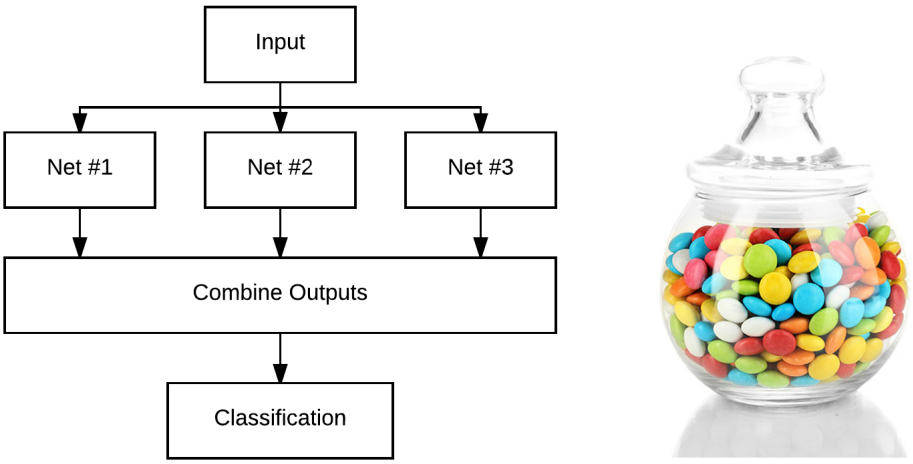


Figure 6.2: **Left:** An ensemble of neural networks consists of *multiple* networks. When classifying an input image the data point is passed to each network where it classifies the image *independently* of all other networks. The classifications across networks are then averaged to obtain the final prediction. **Right:** Ensemble methods are possible due to Jensen's Inequality. By averaging guesses as to the number of candies in the jar, we can better approximate the true number of candies.

we assume our CNNs are performing well and *are good guessers* (i.e., not randomly guessing). Therefore, if we average the results of these predictors together, we'll often see a rise in our classification accuracy. This improvement is *exactly why* you see state-of-the-art publications on deep learning train *multiple models* and then report their best accuracies over these ensembles.

6.1.2 Constructing an Ensemble of CNNs

The first step in building an ensemble of CNNs is to train each *individual* CNN. At this point in *Deep Learning for Computer Vision with Python* we've seen many examples of training a *single* CNN – *but how do we train **multiple** networks?* In general, we have two options:

1. Run the script we use to train a *single* network *multiple times*, changing the path to the output serialized model weights to be unique for each run.
2. Create a separate Python script that uses `for` loop to train N networks and outputs the serialized model at the end of each iteration.

Both methods are perfectly acceptable to train a simple ensemble of CNNs. Since we're quite comfortable running a *single* command to generate a *single* output CNN, let's try the second option where a *single* script is responsible for training *multiple networks*. Open up a new file, name it `train_models.py`, and insert the following code:

```
# set the matplotlib backend so figures can be saved in the background
import matplotlib
matplotlib.use("Agg")

# import the necessary packages
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report
from pyimagesearch.nn.conv import MiniVGGNet
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import SGD
```

```
11 from keras.datasets import cifar10
12 import matplotlib.pyplot as plt
13 import numpy as np
14 import argparse
15 import os
```

Lines 2 and 3 import the `matplotlib` package and then set the backend such that we can save plots to disk. **Lines 6-15** then import our remaining Python packages. All of these packages we have used before, but I'll call out the important ones below:

- **Line 8:** We'll be training multiple `MiniVGGNet` models to form our ensemble.
- **Line 9**: We'll be using the `ImageDataGenerator` class to apply data augmentation when training our network.
- **Lines 10 and 11:** Our MiniVGGNet models will be trained on the CIFAR-10 dataset using the SGD optimizer.

The `train_models.py` script will require two command line arguments followed by an additional optional one:

```
17 # construct the argument parse and parse the arguments
18 ap = argparse.ArgumentParser()
19 ap.add_argument("-o", "--output", required=True,
20     help="path to output directory")
21 ap.add_argument("-m", "--models", required=True,
22     help="path to output models directory")
23 ap.add_argument("-n", "--num-models", type=int, default=5,
24     help="# of models to train")
25 args = vars(ap.parse_args())
```

The `--output` argument will serve as the base output directory where we'll save classification reports along with loss/accuracy plots for each of the networks we will train. We then have the `--models` switch which controls the path to the output directory where we will be storing our serialized network weights.

Finally, the `--num-models` argument indicates the *number of networks* in our ensemble. We default this value to `5` networks. While traditional ensemble methods such as Random Forests typically consist of > 30 Decision Trees (and in many cases > 100), we normally only see 5-10 Convolutional Neural Networks in an ensemble – the reason is due to the fact that CNNs are much more time-consuming and computationally expensive to train.

Our next code block handles loading the CIFAR-10 dataset from disk, scaling the pixel intensities to the range $[0, 1]$, and one-hot encoding our class labels so we can apply categorical cross-entropy as our loss function:

```
27 # load the training and testing data, then scale it into the
28 # range [0, 1]
29 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
30 trainX = trainX.astype("float") / 255.0
31 testX = testX.astype("float") / 255.0
32
33 # convert the labels from integers to vectors
34 lb = LabelBinarizer()
35 trainY = lb.fit_transform(trainY)
36 testY = lb.transform(testY)
37
```

```
38 # initialize the label names for the CIFAR-10 dataset
39 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
40     "dog", "frog", "horse", "ship", "truck"]
```

We also need to initialize our `ImageDataGenerator` so we can apply data augmentation to the CIFAR-10 training data:

```
42 # construct the image generator for data augmentation
43 aug = ImageDataGenerator(rotation_range=10, width_shift_range=0.1,
44     height_shift_range=0.1, horizontal_flip=True,
45     fill_mode="nearest")
```

Here we'll allow images to be randomly rotated 10 degrees, shifted by a factor of 0.1, and randomly horizontally flipped.

We are now ready to train each individual MiniVGGNet model in the ensemble:

```
47 # loop over the number of models to train
48 for i in np.arange(0, args["num_models"]):
49     # initialize the optimizer and model
50     print("[INFO] training model {}/{}".format(i + 1,
51         args["num_models"]))
52     opt = SGD(lr=0.01, decay=0.01 / 40, momentum=0.9,
53         nesterov=True)
54     model = MiniVGGNet.build(width=32, height=32, depth=3,
55         classes=10)
56     model.compile(loss="categorical_crossentropy", optimizer=opt,
57         metrics=["accuracy"])
```

On **Line 48** we start looping over the number of `--num-models` to train. **Line 52** initializes the SGD optimizer using a learning rate of $\alpha = 0.01$, a momentum of $\gamma = 0.9$, and a standard Keras learning rate decay of the learning rate divided by total number of epochs (Chapter 16, *Starter Bundle*). We'll also indicate that Nesterov accelerations should be used. **Lines 54-57** then instantiate the individual `MiniVGGNet` model and compile it.

Next, let's train the network and serialize it to disk:

```
59     # train the network
60     H = model.fit_generator(aug.flow(trainX, trainY, batch_size=64),
61         validation_data=(testX, testY), epochs=40,
62         steps_per_epoch=len(trainX) // 64, verbose=1)
63
64     # save the model to disk
65     p = [args["models"], "model_{}.model".format(i)]
66     model.save(os.path.sep.join(p))
```

Lines 60-62 train our `MiniVGGNet` model using the `fit_generator` method. We use `fit_generator` because we need the `.flow` method of the `ImageDataGenerator` to apply data augmentation. The network will be trained for a total of 40 epochs using batch sizes of 64. The `steps_per_epoch` parameter controls the number of batches per epoch, which is simply the number of training samples divided by our batch size.

After the network finishes training, we construct a unique output path for it and save the weights to disk (**Lines 65 and 66**). Let's also save a `classification_report` to disk for each network as well so we can review performance once the script finishes executing:

```
68      # evaluate the network
69      predictions = model.predict(testX, batch_size=64)
70      report = classification_report(testY.argmax(axis=1),
71          predictions.argmax(axis=1), target_names=labelNames)
72
73      # save the classification report to file
74      p = [args["output"], "model_{}.txt".format(i)]
75      f = open(os.path.sep.join(p), "w")
76      f.write(report)
77      f.close()
```

The same goes for plotting our loss and accuracy over time:

```
79      # plot the training loss and accuracy
80      p = [args["output"], "model_{}.png".format(i)]
81      plt.style.use("ggplot")
82      plt.figure()
83      plt.plot(np.arange(0, 40), H.history["loss"],
84          label="train_loss")
85      plt.plot(np.arange(0, 40), H.history["val_loss"],
86          label="val_loss")
87      plt.plot(np.arange(0, 40), H.history["acc"],
88          label="train_acc")
89      plt.plot(np.arange(0, 40), H.history["val_acc"],
90          label="val_acc")
91      plt.title("Training Loss and Accuracy for model {}".format(i))
92      plt.xlabel("Epoch #")
93      plt.ylabel("Loss/Accuracy")
94      plt.legend()
95      plt.savefig(os.path.sep.join(p))
96      plt.close()
```

It's important to note that we would *never* jump straight to training an ensemble – we would first run a series of experiments to determine which combination of architecture, optimizer, and hyperparameters yields the highest accuracy on a given dataset.

Once you've reached this optimal set of combination, we would *then* switch over to training multiple models to form an ensemble. Training an ensemble as your very first experiment is considered **premature optimization** as you don't know what combination of architecture, optimizer, and hyperparameters will work best for your given dataset.

With that said, we know from Chapter 15 of the *Starter Bundle* that MiniVGGNet trained with SGD gives a reasonable classification accuracy of **83**% – by applying ensemble methods we hope to increase this accuracy.

To train our set of MiniVGGNet models, just execute the following command:

```
$ python train_models.py --output output --models models
[INFO] training model 1/5
[INFO] training model 2/5
```

```
[INFO] training model 3/5
[INFO] training model 4/5
[INFO] training model 5/5
```

Since we are now training *five* networks rather than *one*, it will take 5x as long for this script to run. Once it executes, take a look at your `output` directory:

```
$ ls output/
model_0.png  model_1.png  model_2.png  model_3.png  model_4.png
model_0.txt  model_1.txt  model_2.txt  model_3.txt  model_4.txt
```

Here you will see the output classification reports and training curves for each of the networks. Using `grep` we can easily extract the classification accuracy of each network:

```
$ grep 'weighted avg' output/*.txt
output/model_0.txt:weighted avg       0.83      0.83      0.83     10000
output/model_1.txt:weighted avg       0.83      0.83      0.83     10000
output/model_2.txt:weighted avg       0.83      0.83      0.83     10000
output/model_3.txt:weighted avg       0.82      0.82      0.82     10000
output/model_4.txt:weighted avg       0.83      0.83      0.83     10000
```

Four of the five networks obtain 83% classification accuracy while the remaining network reaches only 82% accuracy. Furthermore, looking at all five training plots (Figure 6.3) we can see that each set of learning curves looks somewhat similar, although each also looks unique, demonstrating that each MiniVGGNet model "learned" in a different manner.

Now that we've trained our five individual models, it's time to combine their predictions and see if our classification accuracy increases.

6.1.3 Evaluating an Ensemble

To construct and evaluate our ensemble of CNNs, create a separate file named `test_ensemble.py` and insert the following code:

```
1  # import the necessary packages
2  from sklearn.preprocessing import LabelBinarizer
3  from sklearn.metrics import classification_report
4  from keras.models import load_model
5  from keras.datasets import cifar10
6  import numpy as np
7  import argparse
8  import glob
9  import os
10
11 # construct the argument parse and parse the arguments
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-m", "--models", required=True,
14      help="path to models directory")
15 args = vars(ap.parse_args())
```

Lines 2-9 import our required Python packages while **Lines 12-15** parse our command line arguments. We only need a single switch with here, `--models`, the path to where our serialized network weights are stored on disk.

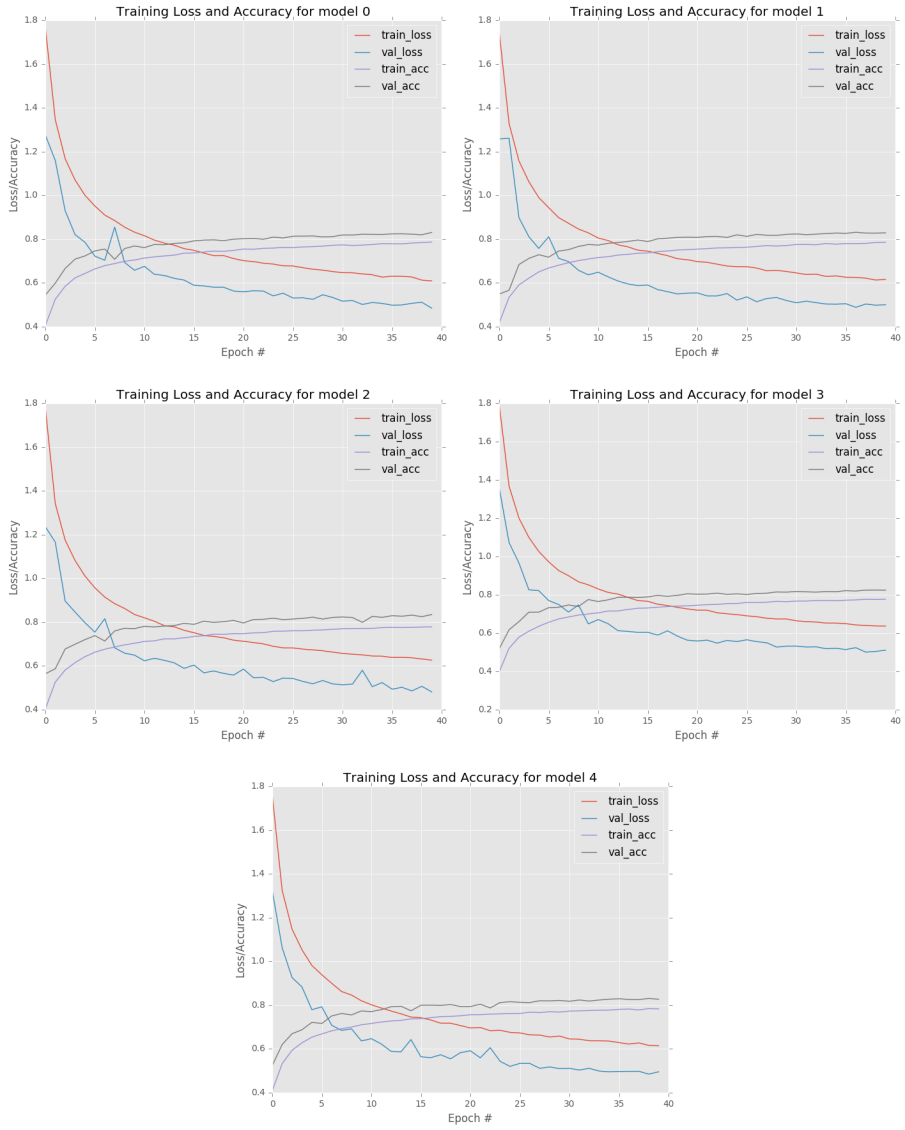


Figure 6.3: Training and validation plots for each of the five networks in our ensemble.

From there we can load the CIFAR-10 dataset, keeping only the testing set since we are only *evaluating* (and not training) our networks:

```
17 # load the testing data, then scale it into the range [0, 1]
18 (testX, testY) = cifar10.load_data()[1]
19 testX = testX.astype("float") / 255.0
20
21 # initialize the label names for the CIFAR-10 dataset
22 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
23     "dog", "frog", "horse", "ship", "truck"]
24
25 # convert the labels from integers to vectors
26 lb = LabelBinarizer()
27 testY = lb.fit_transform(testY)
```

We now need to gather the paths to our pre-trained MiniVGGNet networks, which is easy enough using the `glob` module built into Python:

```
29 # construct the path used to collect the models then initialize the
30 # models list
31 modelPaths = os.path.sep.join([args["models"], "*.model"])
32 modelPaths = list(glob.glob(modelPaths))
33 models = []
```

Line 31 constructs a wildcard path (notice the asterisk "*" in the file path) to all `.model` files in the `--models` directory. Using `glob.glob` on **Line 32** we can automatically find all file paths inside `--models` that end with the `.model` file extension. After executing **Line 32** our `modelPaths` list now contains the following entries:

```
['models/model_0.model', 'models/model_1.model', 'models/model_2.model',
    'models/model_3.model', 'models/model_4.model']
```

Line 33 then initializes a list of `models` which will store the deserialized MiniVGGNet networks loaded from disk.

Let's go ahead and load each model from disk now:

```
35 # loop over the model paths, loading the model, and adding it to
36 # the list of models
37 for (i, modelPath) in enumerate(modelPaths):
38     print("[INFO] loading model {}/{}".format(i + 1,
39         len(modelPaths)))
40     models.append(load_model(modelPath))
```

On **Line 37** we loop over each of the individual `modelPath` file paths. We then load the serialized network via `load_model` and append it to the `models` list.

Finally, we are ready to evaluate our ensemble:

```
42 # initialize the list of predictions
43 print("[INFO] evaluating ensemble...")
44 predictions = []
45
46 # loop over the models
47 for model in models:
48     # use the current model to make predictions on the testing data,
49     # then store these predictions in the aggregate predictions list
50     predictions.append(model.predict(testX, batch_size=64))
51
52 # average the probabilities across all model predictions, then show
53 # a classification report
54 predictions = np.average(predictions, axis=0)
55 print(classification_report(testY.argmax(axis=1),
56     predictions.argmax(axis=1), target_names=labelNames))
```

On **Line 44** we initialize our list of `predictions`. Each model in the `models` list will produce ten probabilities (one for each class label in the CIFAR-10 dataset) for *every* data point in the testing

set. Given that there are 10,000 data points in the CIFAR-10 dataset, each `model` will produce an array of size $10,000 \times 10$ – each row corresponds to a given data point and each column the corresponding probability.

To accumulate these predictions, we loop over each individual `model` on **Line 47**. We then call `.predict` on the testing data and update the `predictions` list with the probabilities produced by the respective model. After we have looped over the five models in our ensemble and updated the `predictions` list, our `predictions` array now has the shape `(5, 10000, 10)`, implying that there are five models, each of which produced 10 class label probabilities for each of the 10,000 testing data points. **Line 54** then *averages the probabilities* for each testing data point across all five models.

To see this for ourselves, we can investigate the shape of our `predictions` array which is now `(10000, 10)` implying that probabilities for each of the five models have been averaged together. The averaging is why we call this method an ensemble – we are taking the output of *multiple, independent models* and averaging them together to obtain our final output. According to Jensen's Inequality, applying ensemble methods should perform no worse (on average) than selecting one of the *individual* models at random.

Finally, **Lines 55 and 56** display a classification report for our ensemble predictions. To determine if our ensemble of MiniVGGNet models increased classification accuracy, execute the following command:

```
$ python test_ensemble.py --models models
[INFO] loading model 1/5
[INFO] loading model 2/5
[INFO] loading model 3/5
[INFO] loading model 4/5
[INFO] loading model 5/5
[INFO] evaluating ensemble...
              precision    recall  f1-score   support

    airplane       0.89      0.82      0.85      1000
  automobile       0.93      0.93      0.93      1000
        bird       0.80      0.74      0.77      1000
         cat       0.72      0.67      0.69      1000
        deer       0.80      0.86      0.83      1000
         dog       0.77      0.77      0.77      1000
        frog       0.86      0.91      0.88      1000
       horse       0.91      0.87      0.89      1000
        ship       0.89      0.94      0.92      1000
       truck       0.87      0.93      0.90      1000

avg / total       0.84      0.84      0.84     10000
```

Looking at the output classification report we can see that we increased our accuracy from 83% to **84%**, simply by combining the output of multiple networks, *even though* these networks were trained on the same dataset using the exact same hyperparameters. In general, you can expect an increase of 1-5% accuracy when applying ensembles of Convolutional Neural Networks depending on your dataset.

6.2 Summary

In this chapter, we reviewed the *ensembles* machine learning technique and how training *multiple, independent* models followed by *averaging the results together* can increase classification accuracy.

The theoretical justification for ensemble methods can be found by reviewing Jensen's Inequality which states that on average, we are better off averaging the results of multiple models together rather than picking one at random.

In fact, the top results you see reported by state-of-the-art papers (including Inception [17], ResNet [24], etc.) are the *average* over multiple models (typically 3-5, depending on how long the authors had to train their networks before their publication was due). Depending on your dataset, you can normally expect a 1-5% increase in accuracy.

While ensembles may be a *simple* method to improve classification accuracy they are also a *computationally expensive one* – rather than training a single network, we are now responsible for training N of them. Training a CNN is already a time-consuming operation, so a method that scales linearly may not be practical in some situations.

To alleviate the computational burden of training multiple models, Huang et al. [25] propose the idea of using cyclic learning rate schedules to train *multiple models* during a single training process in their 2017 paper, *Snapshot Ensembles: Train 1, get M for free*.

This method works by starting training with a high learning rate, quickly lowering it, saving the model weights, and then resetting the learning rate to its original value without re-initializing network weights. This action enables the network to theoretically extend coverage to areas of local minima (or at least areas of low loss) *multiple times* during the training process. Snapshot Ensembles are outside the scope of this book but are worth investigating if you need to boost your classification accuracy but cannot afford to train multiple models.

7. Advanced Optimization Methods

So far in this book, we have *only* studied and used Stochastic Gradient Descent (SGD) to optimize our networks – but there are other optimization methods that are used in deep learning. Specifically, these more advanced optimization techniques seek to either:

1. Reduce the amount of time (i.e., number of epochs) to obtain reasonable classification accuracy.
2. Make the network more "well-behaved" for a larger range of hyperparameters other than the learning rate.
3. Ideally, obtain higher classification accuracy than what is possible with SGD.

With the latest incarnation of deep learning, there has been an explosion of new optimization techniques, each seeking to improve on SGD and provide the concept of *adaptive learning rates*. As we know, SGD modifies *all* parameters in a network *equally* in proportion to a given learning rate. However, given that the learning rate of a network is (1) the most important hyperparameter to tune and (2) a hard, tedious hyperparameter to set correctly, deep learning researchers have postulated that it's possible to *adaptively* tune the learning rate (and in some cases, *per parameter*) as the network trains.

In this chapter, we'll review adaptive learning rate methods. I'll also provide suggestions on which optimization algorithms you should be using in your own projects.

7.1 Adaptive Learning Rate Methods

In order to understand each of the optimization algorithms in this section, we are going to examine them in terms of pseudocode – specifically the *update step*. Much of this chapter has been inspired by the excellent overview of optimization methods by Karpathy [26] and Ruder [27]. We'll extend (and in some cases, simplify) their explanations of these methods to make the content more digestible.

To get started, let's take a look at an algorithm we are already familiar with – the update phase of vanilla SGD:

```
W += -lr * dW
```

Here we have three values:

1. `W`: Our weight matrix.
2. `lr`: The learning rate.
3. `dW`: The gradient of `W`.

Our learning rate here is fixed and, provided it is small enough, we know our loss will decrease during training. We've also seen extensions to SGD which incorporate momentum and Nesterov acceleration in Chapter 7. Given this notation, let's explore common adaptive learning rate optimizers you will encounter in your deep learning career.

7.1.1 Adagrad

The first adaptive learning rate method we are going to explore is *Adagrad*, first introduced by Duchi et al [28]. Adagrad adapts the learning rate to the network parameters. Larger updates are performed on parameters that change infrequently while smaller updates are done on parameters that change frequently.

Below we can see a pseudocode representation of the Adagrad update:

```
cache += (dW ** 2)
W += -lr * dW / (np.sqrt(cache) + eps)
```

The first parameter you'll notice here is the `cache` – this variable maintains the per-parameter sum of squared gradients and is updated at *every mini-batch* in the training process. By examining the `cache`, we can see which parameters are updated frequently and which ones are updated infrequently.

We can then divide the `lr * dW` by the square-root of the cache (adding in an epsilon value for smoothing and preventing division by zero errors). Scaling the update by all previous sum of square gradients allows us to adaptively update the parameters in our network.

Weights that have *frequently updated/large gradients* in the cache will scale the size of the update down, effectively *lowering* the learning rate for the parameter. On the other hand, weights that have *infrequent updates/smaller gradients* in the cache will scale up the size of the update, effectively *raising* the learning rate for the specific parameter.

The primary benefit of Adagrad is that we no longer have to manually tune the learning rate – most implementations of the Adagrad algorithm leave the initial learning rate at 0.01 and allow the adaptive nature of the algorithm to tune the learning rate on a per-parameter basis.

However, the weakness of Adagrad can be seen by examining the cache. At each mini-batch, the squared gradients are accumulated in the denominator. Since the gradients are squared (and are therefore always positive), this accumulation keeps growing and growing during the training process. As we know, dividing a small number (the gradient) by a very large number (the cache) will result in an update that is infinitesimally small, too small for the network to actually learn anything in later epochs.

This phenomena occurs for even small, infrequently updated parameters as positive values in the cache grows monotonically, which is why we rarely see Adagrad used to train (modern) deep learning neural networks. However, it is important to review so we can understand the extensions to the Adagrad algorithm.

7.1.2 Adadelta

The Adadelta algorithm was proposed by Zeiler in their 2012 paper, *ADADELTA: An Adaptive Learning Rate Method* [29]. Adadelta can be seen as an extension to Adagrad that seeks to reduce the monotonically decreasing learning rate caused by the cache.

In the Adagrad algorithm, we update our cache with *all* of the previously squared gradients. However, Adadelta restricts this cache update by *only* accumulating a small number of past gradients – when actually implemented, this operation amounts to computing a *decaying average* of all past squared gradients.

Adadelta can thus be seen as an improvement to Adagrad; however, the very closely related RMSprop algorithm (which also performs cache decay) is often preferred to Adadelta.

7.1.3 RMSprop

Developed independently of Adadelta, the RMSprop algorithm is an (unpublished) optimization algorithm shown in the slides of Geoffrey Hinton's Coursera class [2]. Similar to Adadelta, RMSprop attempts to rectify the negative effects of a globally accumulated cache by converting the cache into an exponentially weighted moving average.

Let's take a look at the RMSprop pseudocode update:

```
cache = decay_rate * cache + (1 - decay_rate) * (dW ** 2)
W += -lr * dW / (np.sqrt(cache) + eps)
```

The first aspect of RMSprop you'll notice is that the actual update to the weight matrix `W` is identical to that of Adagrad – what matters here is how the `cache` is updated. The `decay_rate`, often defined as ρ, is a hyperparameter typically set to 0.9. Here we can see that previous entries in the cache will be weighted substantially smaller than new updates. This "moving average" aspect of RMSprop allows the cache to "leak out" old squared gradients and replace them with newer, "fresher" ones.

Again, the actual update to `W` is identical to that of Adagrad – the crux of the algorithm hinges on exponentially decaying the cache, enabling us to avoid monotonically decreasing learning rates during the training process. In practice, RMSprop tends to be more effective than both Adagrad and Adadelta when applied to training a variety of deep learning networks [5]. Furthermore, RMSprop tends to converge *significantly faster* than SGD.

Outside of SGD, RMSprop has arguably been the second most used optimization algorithm in recent deep learning literature; however, the next optimization method we are about to discuss, Adam, is now being used more than RMSprop.

7.1.4 Adam

The Adam (Adaptive Moment Estimation) optimization algorithm, proposed by Kingma and Ba in their 2014 paper, *Adam: A Method for Stochastic Optimization* [1] is essentially RMSprop only with momentum added to it:

```
m = beta1 * m + (1 - beta1) * dW
v = beta2 * v + (1 - beta2) * (dW ** 2)
W += -lr * m / (np.sqrt(v) + eps)
```

Again, I want to draw special attention to these pseudocode updates as they were derived and popularized by Karpathy's excellent optimization method notes [26].

The values of both `m` and `v` are similar to SGD momentum, relying on their respective previous values from time $t-1$. The value `m` represents the first moment (mean) of the gradients while `v` is the second moment (variance).

The actual update to `W` is near identical to RMSprop, only now we are using the "smoothed" version (due to computing the mean) of `m` rather than the raw gradient dW – using the mean tends to lead to more desirable updates as we can smooth over noisy updates in the raw `dW` values. Typically `beta1` is set to 0.9 while `beta2` is set to 0.999 – these values are rarely (if ever) changed when using the Adam optimizer.

In practice, Adam tends to work better than RMSprop in many situations. For more details on the Adam optimization algorithm, please see Kingma and Ba [1].

7.1.5 Nadam

Just like Adam is RMSprop with momentum, Nadam is RMSprop with Nesterov acceleration. Nadam was proposed by Timothy Dozat, a Ph.D. student at Stanford University [30]. We typically don't see Nadam used "in the wild", but is important to understand that this variation of Adam does exist.

7.2 Choosing an Optimization Method

Given the choices between all of these optimization algorithms, which one should you choose? Unfortunately, the answer is highly inconclusive – the work of Schaul et al. in 2014, *Unit tests for Stochastic Optimization* [31], attempted to benchmark many of these optimization methods and found that while adaptive learning rate algorithms performed favorably, there was no clear winner.

Deep learning optimization algorithms (and how to choose them) is still an open area of research, and will likely continue to be for many years. Therefore, instead of exhaustively trying *every* optimization algorithm you can find, throwing each at your dataset and noting what sticks, it's better to ***master*** two or three optimization algorithms. Often, the success of a deep learning project is a combination of the *optimization algorithm* (and associated parameters) along with *how adept the researcher is at "driving" the algorithm.*

7.2.1 Three Methods You Should Learn how to Drive: SGD, Adam, and RMSprop

> *"The choice of which algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning)."* – Goodfellow et al. [5]

Given the success of adaptive learning rate algorithms such as RMSprop and Adam, you might be tempted to simply ignore SGD and treat it like an archaic tool. After all, "better" methods exist, right?

However, implying ignoring SGD would be a *big* mistake. Take a look at any recent state-of-the-art deep learning publication on challenging image classification datasets such as ImageNet: AlexNet [6], VGGNet [11], SqueezeNet [32], Inception [17], ResNet [33] – ***all of these state-of-the-art architectures were trained using SGD***.

But why is this? We can clearly see the benefits in algorithms that apply adaptive learning rates such as RMSprop and Adam – networks can converge *faster*. However, the speed of convergence, while important, is not the most important factor – *hyperparameters still win out*. If you cannot tune the hyperparameters to a given optimizer (and associated model), your network will never obtain reasonable accuracy.

While SGD certainly converges *slower* than adaptive learning rate algorithms, it's also a more studied algorithm. Researchers are more familiar with SGD and have spent *years* using it to train networks.

For example, consider a professional race car driver who has been driving the same make and model of a race car for five years. Then, one day, the driver's sponsor changes and they are forced

to drive a new vehicle. The driver has no time to try out the new race car, and they are forced to start racing with no experience in the car. Will the driver perform as well in their first few races? Most likely not – the driver is not familiar with the vehicle and its intricacies (but still might perform reasonably as the driver is a professional after all).

The same goes for deep learning architectures and optimization algorithms. The more experiments we perform with a given architecture and optimization algorithm, *the more we learn about the intricacies of the training process*. Given that SGD has been the cornerstone of training neural networks for nearly 60 years, it's no wonder that this algorithm is still consistently used today – the rate of convergence simply doesn't matter (as much) when compared to the performance (accuracy) of the model.

Simply put: If we can obtain higher accuracy on a given dataset using SGD, we'll likely use SGD even if it takes 1.5x longer to train than when using Adam or RMSprop simply because we understand the hyperparameters better. Currently, the most used deep learning optimization algorithms are:

1. SGD
2. RMSprop
3. Adam

I would recommend that you master SGD *first* and apply it to every architecture and dataset you encounter. In some cases. it will perform great, and, in others, it will perform poorly. The goal here is for you to *expose* yourself to as many deep learning problems as possible using a *specific* optimization algorithm and learn how to tune the associated hyperparameters. Remember, deep learning is part science and part art – mastering an optimization algorithm is *absolutely* an art that requires much practice. From there, move on to either RMSprop or Adam.

I personally recommend studying Adam prior to RMSprop as, in my experience, Adam tends to outperform RMSprop in most situations.

7.3 Summary

In this chapter, we discussed adaptive learning rate optimization algorithms that can be used in place of SGD. Choosing an optimization algorithm to train a deep neural network is highly dependent on your familiarity with:

1. The dataset
2. The model architecture
3. The optimization algorithm (and associated hyperparameters)

Instead of exhaustively running experiments to try every optimization algorithm you can find, it's instead better to *master* two or three techniques and how to tune their hyperparameters. Becoming an expert at these techniques will enable you to apply *new model architectures* to *datasets you haven't worked with before* with much more ease.

My personal recommendation is to spend *a lot* of time early in your deep learning career mastering how to use SGD; specifically, SGD with momentum. Once you feel comfortable applying SGD to a variety of architectures and datasets, move on to Adam and RMSprop.

Finally, keep in mind that the speed of model rate convergence is secondary to loss and accuracy – choose an optimization algorithm that you can (confidently) tune the hyperparameters to, resulting in a reasonably performing network.

8. Optimal Pathway to Apply Deep Learning

In Chapter 10 of the *Starter Bundle*, we examined a recipe to train a neural network. The four ingredients to the recipe included:

1. Your dataset
2. A loss function
3. A neural network architecture
4. An optimization method

Using this recipe, we can train any type of deep learning model. However, what this recipe *does not* cover is the optimal way to combine these ingredients together, as well as which parts of the recipe you need to fiddle with if you aren't obtaining your desired results.

As you'll find out in your deep learning career, arguably the hardest aspect of deep learning is examining your accuracy/loss curve and making the decision on what to do next. If your training error is too high, what do you do? What happens if your validation error is also high? How do you adjust your recipe when your validation error matches your training error... but then your testing set error is high?

Inside this chapter, I'll discuss the optimal way to apply deep learning techniques, starting with rules of thumb you can use to adjust your recipe for training. I'll then provide a decision process that you can use when deciding if you should train your deep learning model from scratch or apply transfer learning. By the end of this chapter, you'll have a strong understanding of rules of thumb that expert deep learning practitioners use when training their own networks.

8.1 A Recipe for Training

The following section is heavily inspired by Andrew Ng's excellent tutorial at NIPS 2016 titled, *Nuts and Bolts of Building Deep Learning Applications* [34]. In this talk, Ng discussed how we can get deep learning methods to work in our own products, businesses, and academic research. Arguably the most important takeaway from Ng's talk follows (summarized by Malisiewicz [35]):

> *"Most issues in applied deep learning come from training data/testing data mismatch. In some scenarios this issue just doesn't come up, but you'd be surprised how*

> *often applied machine learning projects use training data (which is easy to collect and annotate) that is different from the target application.*" – Andrew Ng (summarized by Malisiewicz)

What both Ng and Malisiewicz are saying here is that you should take excruciating care to make sure your training data is representative of your validation and testing sets. Yes, obtaining, annotating, and labeling a dataset is extremely time consuming and even in some cases, very expensive. And yes, deep learning methods do tend to generalize well in certain situations. However, you cannot expect any machine learning model trained on data that is *not* representative to succeed.

For example, suppose we are tasked with the responsibility of building a deep learning system responsible for recognizing the make and model of a vehicle from a camera mounted to our car as we drive down the road (Figure 8.1, *left*).

Figure 8.1: **Left:** Cars on a highway that we wish to identify using deep learning. **Right:** Example "product shot" images of what our network was actually trained on.

The first step is to gather our training data. To speed up the data gathering process, we decide to scrape websites that have *both* photos of cars *and* their make and model listed on the webpage – great examples of such websites in include Autotrader.com, eBay, CarMax, etc. For each of these websites we can build a simple spider that crawls the website, finds individual product listings, (i.e., the "car pages" that list the specifications of the vehicle), and then download the images and make + model information.

This method is quite simplistic, and outside the time it takes us to develop the spider, it won't take us long to accumulate a reasonably large labeled dataset. We then split this dataset into two: training and validation, and proceed to train a given deep learning architecture to a high accuracy ($> 90\%$).

However, when we apply our newly trained model to example images, such as in Figure 8.1 (*left*), we find that results are *terrible* – we are lucky to obtain 5 percent accuracy when deployed in the real-world. Why is this?

The reason is that we took the easy way out. We didn't stop to consider that the product shots of cars listed on Autotrader, CarMax, and eBay (Figure 8.1, *right*) are *not representative* of the vehicles our deep learning vision system will be seeing mounted to the dash of our car. While our deep learning system may be great at identifying the make and model of a vehicle in a product shot, it will fail to recognize the make an model of a car from either a frontal or rear view, as is common when driving.

There is no shortcut to building your own image dataset. If you expect a deep learning system to obtain high accuracy in a given real-world situation, then make sure this deep learning system

was trained on images *representative* of where it will be deployed – otherwise you will be very disappointed in its performance.

Assuming we have gathered sufficient training data that is representative of the classification task we are trying to solve, Andrew Ng has provided with a four step process to aid us in our training [34].

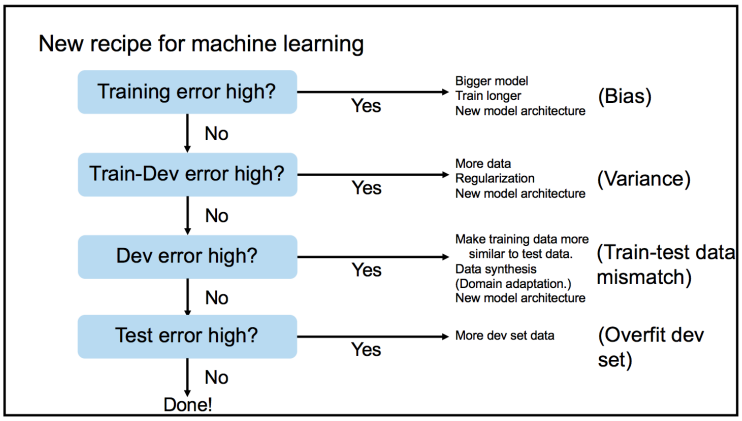


Figure 8.2: Slide 13 of Andrew Ng's talk [34]. Here Ng proposes four separate data splits when training a deep learning model.

Based on Figure 8.2 we can see that Ng is proposing four sets of data splits when training a deep learning model:

1. Training
2. Training-validation (which Ng refers to as "development")
3. Validation
4. Testing

We've already seen training, validation, and testing splits before – but what is this new "training-validation" set? Ng recommends that we take all of our data and split it into 60% for training and the remaining 40% for testing. We then split the testing data into two parts: one for validation and the other for true testing (i.e., the data we never touch until we are ready to evaluate the performance of our network). From our training set, we then take a small chunk of it and add it to our "training-validation set". The training set will help us determine the bias of our model while the training-validation set will help determine variance.

If our training error is too high, as in Figure 8.3 (*top-left*) below, then we should consider *deepening our current architecture* by adding in more layers and neurons. We should also consider training for longer (i.e., more epochs) while simultaneously tweaking our learning rate – using a smaller learning rate may enable you to train for longer while helping prevent overfitting. Finally, if after many experiments using our *current architecture* and *varying learning rates* does not prove useful, then we likely need to try an *entirely different model architecture*.

Moving on to the second item in the flow chart, ***if our training-validation error is high*** (Figure 8.3, *top-right*), then we should examine the regularization parameters in our network. Are we applying dropout layers inside the network architecture? Is data augmentation being used to help generate new training samples? What about the actual loss/update function itself – is a regularization penalty being included? Examine these questions in the context of your own deep learning experiments and start adding in regularization.

You should also consider gathering more training data (again, taking care that this training data is representative of where the model will be deployed) at this point – in nearly all cases having *more*

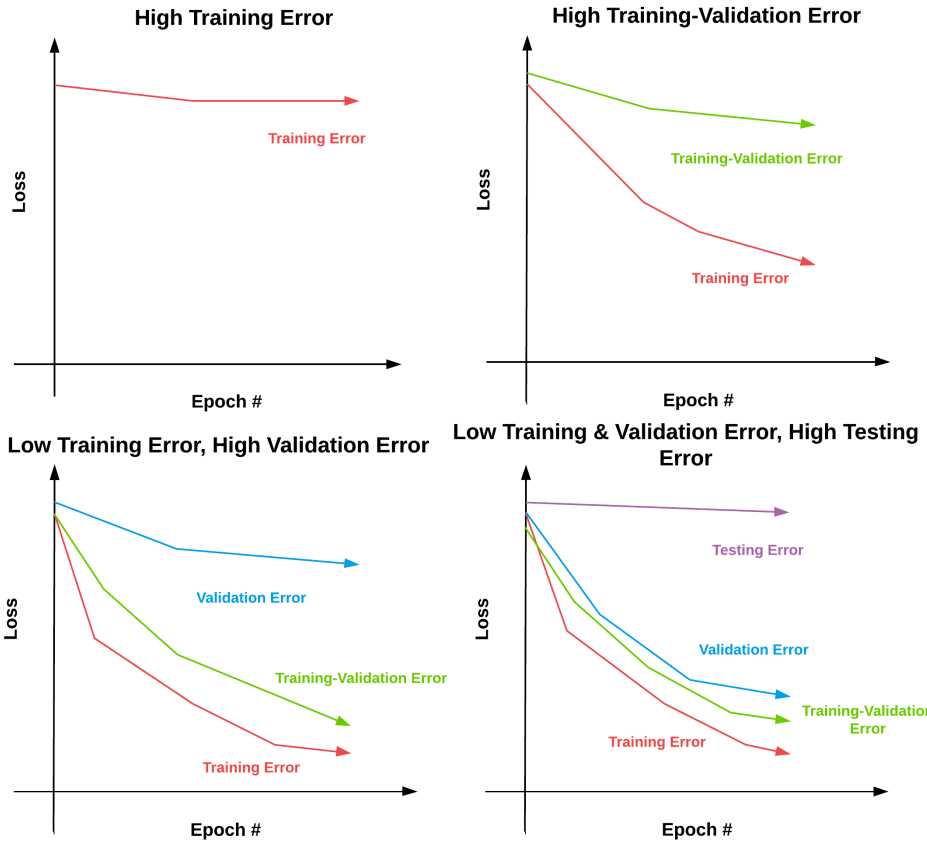


Figure 8.3: The four stages of Andrew Ng's machine learning recipe. **Top-left:** Our training error is high, implying that we need a more powerful model to represent the underlying patterns in the data. **Top-right:** Our training error has decreased, but our training-validation error is high. This implies we should obtain more data or apply strong regularization. **Bottom-left:** If both training and training-validation error are low, but validation error is high we should examine our training data and ensure it mimics our validation and testing sets properly. **Bottom-right:** If training, training-validation, and validation error are all low but testing error is high then we need to gather more training + validation data.

training data is never a bad thing. It is likely that your model does not have enough training data to learn the underlying patterns in your example images. Finally, after exhausting these options, you'll once again want to consider using a different network architecture.

Continuing through the flowchart in Figure 8.3 (*bottom-left*), ***if our training-validation error is low, but our validation set error is high***, we need to examine our training data with a closer eye. Are we absolutely, positively sure that our training images are similar to our validation images?

Be honest with yourself – you *cannot* expect a deep learning model trained on images not representative of the images they'll see in a validation or testing setting to perform well. If you make the hard realization that this is indeed the case, go back to the dataset collection phase and spend the time gathering more data. Without data representative of where your deep learning model will be deployed, you will not obtain high accuracy results. You should also again inspect your regularization parameters – are you regularizing strong enough? Finally, you may once again need to consider a new model architecture.

Finally, we move on to the last step in the flow chart – ***is our testing error high?*** At this point,

we've overfit our model to the training and validation data (Figure 8.3 *bottom-right*). We need to go back and gather more data for the validation set to help us identify when this overfitting is starting to occur. Using this methodology proposed by Andrew Ng, we can more easily make (correct) decisions regarding updating our model/dataset when our experiments don't turn out as we expected.

8.2 Transfer Learning or Train from Scratch

The following section is inspired by the excellent *"Transfer Learning"* lesson of Stanford's cs231n class [36]. I've also included my own anecdotal experiences to aid in your own experiments. Given the success of transfer learning in Chapter 3 on *feature extraction* and Chapter 5 on *fine-tuning*, you may wonder when you should be applying transfer learning and when you should be training a model from scratch.

To make this decision, you need to consider two important factors:

1. The *size* of your dataset.
2. The *similarity* of your dataset to the dataset the pre-trained CNN was trained on (which is typically ImageNet).

Based on these factors we can construct a chart to help us make a decision on whether or not we need to apply transfer learning or train from scratch (Figure 8.4). Let's review each of the four possibilities below.

	Similar Dataset	**Different Dataset**
Small Dataset	Feature extraction using FC layers + classifier	Feature extraction using lower level CONV layers + classifier
Large Dataset	Fine-tuning likely to work, but might have to train from scratch	Fine-tuning worth trying, but will likely not work; likely have to train from scratch

Figure 8.4: A table you can use to determine if you should train our network from scratch or transfer learning. Figure inspired by Greg Chu from Deep Learning Sandbox [37].

Your Dataset is Small and Similar to the Original Dataset

Since your dataset is small, you likely don't have enough training examples to train a CNN from scratch (again, keep in mind that you should *ideally* have 1,000-5,000 examples *per class* you want to recognize). Furthermore, given the lack of training data, it's likely not a good idea to attempt fine-tuning as we'll likely end up overfitting.

Instead, since your image dataset is similar to what the pre-trained network was trained on, you should treat the network as a *feature extractor* and train a simple machine learning classifier on top of these features. You should extract features from layers *deeper in the architecture* as these features are more rich and representative of the patterns learned from the original dataset.

Your Dataset is Large and Similar to the Original Dataset

With a large dataset, we should have enough examples to apply fine-tuning without overfitting. You may be tempted to train your own model from scratch here as well – this is an experiment worth running. However, since your dataset is similar to the *original dataset* the network was *already* trained on, the filters inside the network are likely already discriminative enough to obtain a reasonable classifier. Therefore, apply fine-tuning in this case.

Your Dataset is Small and Different than the Original Dataset

Again, given a small dataset, we likely won't obtain a high accuracy deep learning model by training from scratch. Instead, we should again apply feature extraction and train a standard machine learning model on top of them – but since our data is *different* from the original dataset, we should use *lower level layers* in the network as our feature extractors.

Keep in mind that the deeper we go into the network architecture, the more rich and discriminative the features are *specific* to the dataset it was trained on. By extracting features from lower layers in the network, we can still leverage these filters, but without the abstraction caused by the deeper layers.

Your New Dataset is Large and Different than Original Dataset

In this case, we have two options. Given that we have sufficient training data, we can likely train our own custom network from scratch. *However*, the pre-trained weights from models trained on dataset such as ImageNet make for excellent initializations, *even if the datasets are unrelated*. We should therefore perform two sets of experiments:

1. In the first set of experiments, attempt to fine-tune a pre-trained network to your dataset and evaluate the performance.
2. Then in the second set of experiments, train a brand new model from scratch and evaluate.

Exactly *which* method performs best is *entirely dependent* on your dataset and classification problem. However, I would recommend ***trying to fine-tune first*** as this method will allow you to ***establish a baseline to beat*** when you move on to your second set of experiments and train your network from scratch.

8.3 Summary

In this chapter, we explored the optimal pathway to apply deep learning techniques when training your own custom networks. When gathering your training data, keep in mind there are no shortcuts – take the time to ensure that data you use to *train your model* is *representative of the images your network will see when deployed in a real-world application*.

There is an old computer science anecdote that states *"Garbage in, garbage out"*. If your input data does not represent examples of data points your model will see after being trained, you're essentially falling into this *garbage in, garbage out* trap. That isn't to say your data is "garbage". Instead, remind yourself of this anecdote when performing your own experiments and realize that it's not possible for your deep learning model to perform well on data points it was *never trained to recognize* in the first place.

We also reviewed when you should consider *transfer learning* versus *training your own network from scratch*. With small datasets, you should consider feature extraction. For larger datasets, consider fine-tuning first (to establish a baseline) and then move on to training a model from scratch.

9. Working with HDF5 and Large Datasets

So far in this book, we've only worked with datasets that can fit into the main memory of our machines. For small datasets this is a reasonable assumption – we simply load each individual image, preprocess it, and allow it to be fed through our network. However, for large scale deep learning datasets (e.x., ImageNet), we need to create *data generators* that access only a *portion* of the dataset at a time (i.e., a mini-batch), then allow the batch to be passed through the network.

Luckily, Keras ships with methods that allow you to use the raw file paths on disk as inputs to a training process. You *do not* have to store the entire dataset in memory – simply supply the image paths to the Keras data generator and your images will be loaded in batches and fed through the network.

However, this method is terribly inefficient. Each and every image residing on your disk requires an I/O operation which introduces latency into your training pipeline. Training deep learning networks is already slow enough – we would do well to avoid the I/O bottleneck as much as possible.

A more elegant solution would be to generate an HDF5 dataset for your *raw images* just as we did in Chapter 3 on transfer learning and feature extraction, only this time we are storing the *images themselves* rather than extracted features. Not only is HDF5 capable of storing massive datasets, but it's optimized for I/O operations, especially for extracting batches (called "slices") from the file. As we'll see throughout the remainder of this book, taking the extra step to pack the raw images residing on disk into an HDF5 file allows us to construct a deep learning framework that can be used to rapidly build datasets and train deep learning networks on top of them.

In the remainder of this chapter, I'll demonstrate how to construct an HDF5 dataset for the Kaggle Dogs vs. Cats competition [3]. Then, in the next chapter, we'll use this HDF5 dataset to train the seminal AlexNet architecture [6], eventually resulting in a top-25 position on the leaderboard in the subsequent chapter.

9.1 Downloading Kaggle: Dogs vs. Cats

To download the Kaggle: Dogs vs. Cats dataset you'll first need to create an account on kaggle.com. From there, head to the Dogs vs. Cats homepage (http://pyimg.co/xb5lb).

You'll need to download `train.zip`. ***Do not*** download `test1.zip`. The images inside `test1.zip` are only used for computing predictions and submitting to the Kaggle evaluation server. Since we need the class labels to construct our own training and testing splits we only need `train.zip`. Submitting your own predicted results is outside the scope of this book but can easily be accomplished by writing your predictions on `test1.zip` following the file format outlined in `sampleSubmission.csv`.

After `train.zip` has been downloaded, unarchive it and you'll find a directory named `train` – this directory contains our actual images. The labels themselves can be derived from examining the file names. I have included a sample of the file names below:

```
kaggle_dogs_vs_cats/train/cat.11866.jpg
...
kaggle_dogs_vs_cats/train/dog.11046.jpg
```

As I recommended in the *Starter Bundle*, I'll be using the following data structure for this project:

```
|--- datasets
|    |--- kaggle_dogs_vs_cats
|    |    |--- hdf5
|    |    |--- train
|--- dogs_vs_cats
|    |--- config
|    |--- build_dogs_vs_cats.py
|    |--- ...
```

Notice how I'll be storing the `train` directory containing our example images in a folder dedicated exclusively to the Kaggle: Dogs vs. Cats competition. From there, I have the `dogs_vs_cats` directory which is where we'll be storing the code for this project.

Now that we have downloaded the Dogs vs. Cats dataset and examined our directory structure, let's create our configuration file.

9.2 Creating a Configuration File

Now that we are starting to build more advanced projects and deep learning methods, I like to create a special `config` Python module for each of my projects. For example, here is the directory structure for the Kaggle Dogs vs. Cats project:

```
--- dogs_vs_cats
|    |--- config
|    |    |--- __init__.py
|    |    |--- dogs_vs_cats_config.py
|    |--- build_dogs_vs_cats.py
|    |--- crop_accuracy.py
|    |--- extract_features.py
|    |--- train_alexnet.py
|    |--- train_model.py
|    |--- output/
|    |    |--- __init__.py
|    |    |--- alexnet_dogs_vs_cats.model
```

```
|    |    |--- dogs_vs_cats_features.hdf5
|    |    |--- dogs_vs_cats_mean.json
|    |    |--- dogs_vs_cats.pickle
```

You can ignore the actual Python scripts for now as we'll be reviewing them in the next chapter, but take a look at the directory named `config`. Inside of `config` you'll find a single Python file named `dogs_vs_cats_config.py` – I use this file to store all relevant configurations for the project, including:

1. The paths to the input images.
2. The total number of class labels.
3. Information on the training, validation, and testing splits.
4. The paths to the HDF5 datasets.
5. Paths to output models, plots, logs, etc.

Using a Python file rather than a JSON file allows me to include snippets of Python code and makes the configuration file more efficient to work with (a great example being manipulating file paths using the `os.path` module). I would suggest you get into the habit of using Python-based configuration files for your own deep learning projects as it will greatly improve your productivity and allow you to control most of the parameters in your project through a single file.

9.2.1 Your First Configuration File

Let's go ahead and take a look at my configuration file (`dogs_vs_cats_config.py`) for the Kaggle Dogs vs. Cats dataset:

```
1  # define the paths to the images directory
2  IMAGES_PATH = "../datasets/kaggle_dogs_vs_cats/train"
3
4  # since we do not have validation data or access to the testing
5  # labels we need to take a number of images from the training
6  # data and use them instead
7  NUM_CLASSES = 2
8  NUM_VAL_IMAGES = 1250 * NUM_CLASSES
9  NUM_TEST_IMAGES = 1250 * NUM_CLASSES
10
11 # define the path to the output training, validation, and testing
12 # HDF5 files
13 TRAIN_HDF5 = "../datasets/kaggle_dogs_vs_cats/hdf5/train.hdf5"
14 VAL_HDF5 = "../datasets/kaggle_dogs_vs_cats/hdf5/val.hdf5"
15 TEST_HDF5 = "../datasets/kaggle_dogs_vs_cats/hdf5/test.hdf5"
```

On **Line 2** I define the path to the directory containing the dog and cat images – these are the images that we'll be packing into a HDF5 dataset later in this chapter. **Lines 7-9** define the total number of class labels (two: one for *dog*, another for *cat*) along with the number of validation and testing images (2,500 for each). We can then specify the path to our output HDF5 files for the training, validation, and testing splits, respectively on **Lines 13-15**.

The second half of the configuration file defines the path to the output serialized weights, the dataset mean, and a general "output" path to store plots, classification reports, logs, etc.:

```
17 # path to the output model file
18 MODEL_PATH = "output/alexnet_dogs_vs_cats.model"
19
```

```
# define the path to the dataset mean
DATASET_MEAN = "output/dogs_vs_cats_mean.json"

# define the path to the output directory used for storing plots,
# classification reports, etc.
OUTPUT_PATH = "output"
```

The `DATASET_MEAN` file will be used to store the average red, green, and blue pixel intensity values *across the entire (training) dataset*. When we train our network, we'll subtract the mean RGB values from every pixel in the image (the same goes for testing and evaluation as well). This method, called ***mean subtraction***, is a type of data normalization technique and is more often used than scaling pixel intensities to the range $[0, 1]$ as it's shown to be more effective on large datasets and deeper neural networks.

9.3 Building the Dataset

Now that our configuration file has been defined, let's move on to actually building our HDF5 datasets. Open up a new file, name it `build_dogs_vs_cats.py`, and insert the following code:

```
1  # import the necessary packages
2  from config import dogs_vs_cats_config as config
3  from sklearn.preprocessing import LabelEncoder
4  from sklearn.model_selection import train_test_split
5  from pyimagesearch.preprocessing import AspectAwarePreprocessor
6  from pyimagesearch.io import HDF5DatasetWriter
7  from imutils import paths
8  import numpy as np
9  import progressbar
10 import json
11 import cv2
12 import os
```

Lines 2-12 import our required Python packages. I like to import our project configuration file as the *first* import in the project (**Line 2**). This method is a matter of taste, so feel free to place the import wherever you like in the file. I also rename `dogs_vs_cats_config` as simply `config` to make it less verbose when writing code.

From there, the rest of the imports you have encountered before in previous chapters; however, I would like to draw your attention to the `HDF5DatasetWriter` on **Line 6**, the very same `HDF5DatasetWriter` we defined in Chapter 3 – this class will be used to pack our raw images on disk into a single, serialized file.

We'll also again be using the `progressbar` module, a simple utility library I like to use when measuring the approximate time a given task is taking. This module is totally irrelevant to deep learning, but again, I find it convenient to use as for large datasets it may take several hours to pack a dataset of images into HDF5 format.

Next, let's grab the paths to the images in the Kaggle Dogs vs. Cats dataset:

```
14 # grab the paths to the images
15 trainPaths = list(paths.list_images(config.IMAGES_PATH))
16 trainLabels = [p.split(os.path.sep)[-1].split(".")[0]
17     for p in trainPaths]
```

```
18 le = LabelEncoder()
19 trainLabels = le.fit_transform(trainLabels)
```

The Dogs vs. Cats dataset has the following example directory structure:

```
kaggle_dogs_vs_cats/train/cat.11866.jpg
...
kaggle_dogs_vs_cats/train/dog.11046.jpg
```

Notice how the name of the class is built-into the actual filename. Therefore, we need to extract the file component of the file path, split on the `.` separator, and extract the class name – in fact, that is *exactly* what **Lines 16 and 17** do.

Given the paths to all images in the dataset, they loop over them individually and extract the labels from the file paths. If you find these lines of code confusing, I would suggest taking a second now to manually play with the code, specifically the `os.path.sep` variable and the `.split` function used on the file path string to further see how these utilities are used to manipulate file paths.

Lines 18 and 19 then encode the class labels. For the Kaggle Dogs vs. Cats project we'll need three splits: a training split, a validation split, and a testing split.

Our next code block handles generating each of these splits:

```
21 # perform stratified sampling from the training set to build the
22 # testing split from the training data
23 split = train_test_split(trainPaths, trainLabels,
24      test_size=config.NUM_TEST_IMAGES, stratify=trainLabels,
25      random_state=42)
26 (trainPaths, testPaths, trainLabels, testLabels) = split
27
28 # perform another stratified sampling, this time to build the
29 # validation data
30 split = train_test_split(trainPaths, trainLabels,
31      test_size=config.NUM_VAL_IMAGES, stratify=trainLabels,
32      random_state=42)
33 (trainPaths, valPaths, trainLabels, valLabels) = split
```

On **Lines 23-26** we take our input images and labels and use them to construct the training and testing split. However, we need to perform *another* split on **Lines 30-33** to create the validation set. The validation set is (almost always) taken from the training data. The size of the testing and validation splits are controlled via the `NUM_TEST_IMAGES` and `NUM_VAL_IMAGES`, each of which are defined in our `config` file.

Now that we have our training, testing, and validation splits, let's create a simple list that will allow us to loop over them and efficiently write the images in each dataset to our HDF5 file:

```
35 # construct a list pairing the training, validation, and testing
36 # image paths along with their corresponding labels and output HDF5
37 # files
38 datasets = [
39      ("train", trainPaths, trainLabels, config.TRAIN_HDF5),
40      ("val", valPaths, valLabels, config.VAL_HDF5),
41      ("test", testPaths, testLabels, config.TEST_HDF5)]
```

```
42
43 # initialize the image preprocessor and the lists of RGB channel
44 # averages
45 aap = AspectAwarePreprocessor(256, 256)
46 (R, G, B) = ([], [], [])
```

On **Line 38** we define a `datasets` list that includes our training, validation, and testing variables. Each entry in the list is a 4-tuple consisting of:

1. The *name* of the split (i.e., training, testing, or validation).
2. The respective *image paths* for the split.
3. The *labels* for the split.
4. The path to the output HDF5 file for the split.

We then initialize our `AspectAwarePreprocessor` on **Line 45** used to resize images to 256×256 pixels (keeping the aspect ratio of the image in mind) prior to being written to HDF5. We'll also initialize three lists on **Line 46** – `R`, `G`, and `B`, used to store the average pixel intensities for each channel.

Finally, we are ready to build our HDF5 datasets:

```
48 # loop over the dataset tuples
49 for (dType, paths, labels, outputPath) in datasets:
50     # create HDF5 writer
51     print("[INFO] building {}...".format(outputPath))
52     writer = HDF5DatasetWriter((len(paths), 256, 256, 3), outputPath)
53
54     # initialize the progress bar
55     widgets = ["Building Dataset: ", progressbar.Percentage(), " ",
56         progressbar.Bar(), " ", progressbar.ETA()]
57     pbar = progressbar.ProgressBar(maxval=len(paths),
58         widgets=widgets).start()
```

On **Line 49** we start looping over each of the 4-tuple values in the `datasets` list. For each data split, we instantiate the `HDF5DatasetWriter` on **Line 52**. Here the dimensions of the output dataset will be the `(len(paths), 256, 256, 3)`, implying there are `len(paths)` total images, each of them with a height and width of 256 pixels, a height of 256 pixels, and 3 channels.

Lines 54-58 then initialize our progress bar so we can easily monitor the process of the dataset generation. Again, this code block (along with the rest of the `progressbar` function calls) is entirely optional, so feel free to comment them out if you so wish.

Next, let's write each image in a given data split to the `writer`:

```
60     # loop over the image paths
61     for (i, (path, label)) in enumerate(zip(paths, labels)):
62         # load the image and process it
63         image = cv2.imread(path)
64         image = aap.preprocess(image)
65
66         # if we are building the training dataset, then compute the
67         # mean of each channel in the image, then update the
68         # respective lists
69         if dType == "train":
70             (b, g, r) = cv2.mean(image)[:3]
71             R.append(r)
```

```
72                 G.append(g)
73                 B.append(b)
74
75             # add the image and label # to the HDF5 dataset
76             writer.add([image], [label])
77             pbar.update(i)
78
79         # close the HDF5 writer
80         pbar.finish()
81         writer.close()
```

On **Line 61** we start looping over each *individual* image and corresponding class label in the data split. **Lines 63 and 64** load the image from disk and then apply our aspect-aware preprocessor to resize the image to 256×256 pixels.

We make a check on **Line 69** to see if we are examining the `train` data split and, if so, we compute the average of the Red, Green, and Blue channels (**Line 70**) and update their respective lists on **Lines 71-73**. Computing the average of the RGB channels is *only* done for the training set and is a requirement if we wish to apply mean subtraction normalization.

Line 76 adds the corresponding `image` and `label` to our `HDF5DatasetWriter`. Once all images in the data split have been serialized to the HDF5 dataset, we close the `writer` on **Line 81**.

The final step is to serialize our RGB averages to disk:

```
83 # construct a dictionary of averages, then serialize the means to a
84 # JSON file
85 print("[INFO] serializing means...")
86 D = {"R": np.mean(R), "G": np.mean(G), "B": np.mean(B)}
87 f = open(config.DATASET_MEAN, "w")
88 f.write(json.dumps(D))
89 f.close()
```

Line 86 constructs a Python dictionary of the *average* RGB values over *all* images in the training set. Keep in mind that each individual `R`, `G`, and `B` contains the average of channel for each image in the dataset. Computing the mean of this list gives us the average pixel intensity value for all images in the list. This dictionary is then serialized to disk in JSON format on **Lines 87-88**.

Let's go ahead and serialize the Kaggle Dogs vs. Cats dataset to HDF5 format. Open up a terminal and then issue the following command:

```
$ python build_dogs_vs_cats.py
[INFO] building kaggle_dogs_vs_cats/hdf5/train.hdf5...
Building Dataset: 100% |####################################| Time: 0:02:39
[INFO] building kaggle_dogs_vs_cats/hdf5/val.hdf5...
Building Dataset: 100% |####################################| Time: 0:00:20
[INFO] building kaggle_dogs_vs_cats/hdf5/test.hdf5...
Building Dataset: 100% |####################################| Time: 0:00:19
```

As you can see from my output, an HDF5 file was created for *each* of the training, testing, and validation splits. The training split generation took the longest to generate as this split contained the most data (2m39s). The testing and validation splits took substantially less time ($\approx 20s$) due the fact that there is less data in these splits.

We can see each of these output files on our disk by listing the contents of the `hdf5` directory:

```
$ ls -l ../datasets/kaggle_dogs_vs_cats/hdf5/
total 38400220
-rw-rw-r-- 1 adrian adrian  3932182144 Apr  7 18:00 test.hdf5
-rw-rw-r-- 1 adrian adrian 31457442144 Apr  7 17:59 train.hdf5
-rw-rw-r-- 1 adrian adrian  3932182144 Apr  7 18:00 val.hdf5
```

Looking at these file sizes you might be a bit surprised. The raw Kaggle Dogs vs. Cats images residing on disk are only 595MB – why are the `.hdf5` files so large? The `train.hdf5` file alone is 31.45GB while the `test.hdf5` and `val.hdf5` files are almost 4GB. Why?

Well, keep in mind that raw image file formats such as JPEG and PNG apply *data compression algorithms* to keep image file sizes small. However, we have effectively *removed* any type of compression and are storing the images as *raw NumPy arrays* (i.e., bitmaps). This lack of compression dramatically inflates our storage costs, but will also help speed up our training time as we won't have to waste processor time decoding the image – we can instead access the image directly from the HDF5 dataset, preprocess it, and pass it through our network.

Let's also take a look at our RGB mean file:

```
$ cat output/dogs_vs_cats_mean.json
{"B": 106.13178224639893, "R": 124.96761639328003, "G": 115.97504255599975}
```

Here we can see that the red channel has an average pixel intensity of 124.96 across all images in the dataset. The blue channel has an average of 106.13 and the green channel an average of 115.97. We'll be constructing a new image preprocessor to *normalize* our images by *subtracting* these RGB averages from the input images prior to passing them through our network. This mean normalization helps "center" the data around the zero mean. Typically, this normalization enables our network to learn faster and is also why we use this type of normalization (rather than $[0, 1]$ scaling) on larger, more challenging datasets.

9.4 Summary

In this chapter, we learned how to serialize raw images into an HDF5 dataset suitable for training a deep neural network. The reason we serialized the raw images into an HDF5 file rather than simply accessing mini-batches of image paths on disk when training is due to *I/O latency* – for each image on disk we would have to perform an I/O operation to read the image. This subtle optimization doesn't seem like a big deal, but I/O latency is a *huge* problem in a deep learning pipeline – the training process is already slow enough, and if we make it hard for our networks to access our data, we are only further shooting ourselves in the foot.

Conversely, if we serialize all images into an efficiently packed HDF5 file, we can leverage very fast array slices to extract our mini-batches, thereby *dramatically* reducing I/O latency *and* helping speed up the training process. Whenever you are using the Keras library and working with a dataset too large to fit into memory, be sure you consider serializing your dataset into HDF5 format first – as we'll find out in the next chapter, it makes training your network an easier (and more efficient) task.

10. Competing in Kaggle: Dogs vs. Cats

In our previous chapter, we learned how to work with HDF5 and datasets too large to fit into memory. To do so, we defined a Python utility script that can be used to take an input dataset of images and serialize them into a highly efficient HDF5 dataset. Representing a dataset of images in an HDF5 dataset allows us to avoid issues of I/O latency, thereby speeding up the training process.

For example, if we defined a dataset generator that loaded images sequentially from disk, we would need N read operations, one for each image. However, by placing our dataset of images into an HDF5 dataset, we can instead load *batches* of images using a single read. This action *dramatically* reduces the number of I/O calls and allows us to work with very large image datasets.

In this chapter, we are going to extend our work and learn how to define an *image generator* for HDF5 datasets suitable for training Convolutional Neural Networks with Keras. This generator will open the HDF5 dataset, yield batches of images and associated training labels for the network to be trained on, and proceed to do so until our model reaches sufficiently low loss/high accuracy.

To accomplish this process, we'll first explore three new image pre-processors designed to increase classification accuracy – *mean subtraction*, *patch extraction*, and *cropping* (also called *10-cropping* or *over-sampling*). Once we've defined our new set of pre-processors, we'll move on defining the actual HDF5 dataset generator.

From there, we'll implement the seminal AlexNet architecture from Krizhevsky et al.'s 2012 paper, *ImageNet Classification with Deep Convolutional Neural Networks* [6]. This implementation of AlexNet will then be trained on the Kaggle Dogs vs. Cats challenge. Given the trained model, we'll evaluate its performance on the testing set, followed by using over-sampling methods to boost classification accuracy further. As our results will demonstrate, our network architecture + cropping methods will enable us to obtain a position in the top-25 leaderboard of the Kaggle Dogs vs. Cats challenge.

10.1 Additional Image Preprocessors

In this section we'll implement three new image pre-preprocessors:

1. A mean subtraction pre-processor designed to subtract the mean Red, Green, and Blue pixel intensities across a dataset from an input image (which is a form of data normalization).

2. A patch preprocessor used to randomly extract $M \times N$ pixel regions from an image during training.
3. An over-sampling pre-processor used at testing time to sample five regions of an input image (the four corners + center area) along with their corresponding horizontal flips (for a total of 10 crops).

Using over-sampling, we can boost our classification accuracy by passing the 10 crops through our CNN and then averaging across the 10 predictions.

10.1.1 Mean Preprocessing

Let's get started with the mean pre-processor. In Chapter 9 we learned how to convert an image dataset to HDF5 format – part of this conversion involved computing the average Red, Green, and Blue pixel intensities across *all images* in the *training* dataset. Now that we have these averages, we are going to perform a pixel-wise subtraction of these values from our input images as a form of data normalization. Given an input image I and its R, G, B channels, we can perform mean subtraction via:

- $R = R - \mu_R$
- $G = G - \mu_G$
- $B = B - \mu_B$

Where μ_R, μ_G, and μ_B are computed when the image dataset is converted to HDF5 format. Figure 10.1 includes a visualization of subtracting the mean RGB values from an input image – notice how the subtraction is done pixel-wise.

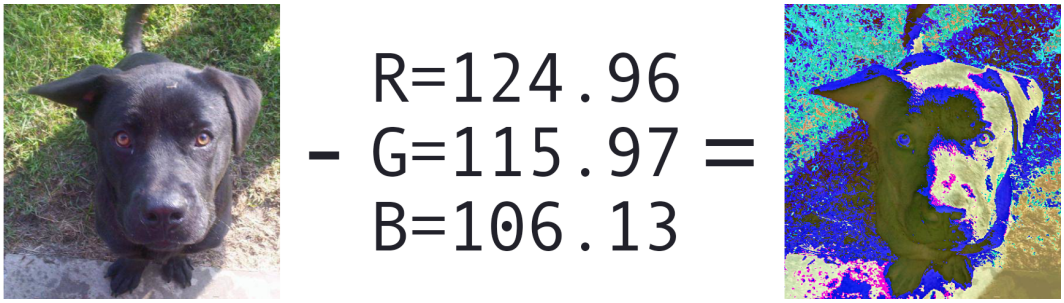


Figure 10.1: An example of applying mean subtraction to an input image (*left*) by subtracting $R = 124.96$, $G = 115.97$, $B = 106.13$ pixel-wise, resulting in the output image (*right*). Mean subtraction is used to reduce the affects of lighting variations during classification.

To make this concept more concrete, let's go ahead and implement our `MeanPreprocessor` class:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |--- preprocessing
|    |    |--- __init__.py
|    |    |--- aspectawarepreprocessor.py
|    |    |--- imagetoarraypreprocessor.py
|    |    |--- meanpreprocessor.py
|    |    |--- simplepreprocessor.py
|    |--- utils
```

Notice how I have placed a new file named `meanpreprocessor.py` in the `preprocessing` sub-module of `pyimagesearch` – this location is where our `MeanPreprocessor` class will live. Let's go ahead and implement this class now:

```
1  # import the necessary packages
2  import cv2
3
4  class MeanPreprocessor:
5      def __init__(self, rMean, gMean, bMean):
6          # store the Red, Green, and Blue channel averages across a
7          # training set
8          self.rMean = rMean
9          self.gMean = gMean
10         self.bMean = bMean
```

Line 5 defines the constructor to the `MeanPreprocessor`, which requires three arguments – the respective Red, Green, and Blue averages computed across the entire dataset. These values are then stored on **Lines 8-10**.

Next, let's define the `preprocess` method, a required function for *every* pre-processor we intend to apply to our image processing pipeline:

```
12     def preprocess(self, image):
13         # split the image into its respective Red, Green, and Blue
14         # channels
15         (B, G, R) = cv2.split(image.astype("float32"))
16
17         # subtract the means for each channel
18         R -= self.rMean
19         G -= self.gMean
20         B -= self.bMean
21
22         # merge the channels back together and return the image
23         return cv2.merge([B, G, R])
```

Line 15 uses the `cv2.split` function to split our input `image` into its respective RGB components. Keep in mind that OpenCV represents images in BGR order rather than RGB ([38], http://pyimg.co/ppao), hence why our return tuple has the signature `(B, G, R)` rather than `(R, G, B)`. We'll also ensure that these channels are of a floating point data type as OpenCV images are typically represented as unsigned 8-bit integers (in which case we can't have negative values, and modulo arithmetic would be performed instead).

Lines 17-20 perform the mean subtraction itself, subtracting the respective mean RGB values from the RGB channels of the input image. **Line 23** then merges the normalized channels back together and returns the resulting image to the calling function.

10.1.2 Patch Preprocessing

The `PatchPreprocessor` is responsible for randomly sampling $M \times N$ regions of an image during the *training process*. We apply patch preprocessing when the spatial dimensions of our input images are *larger* than what the CNN expects – this is a common technique to help reduce overfitting, and is, therefore, a form of regularization. Instead of using the *entire* image during training, we instead crop a random portion of it and pass it to the network (see Figure 10.2 for an example of crop preprocessing).

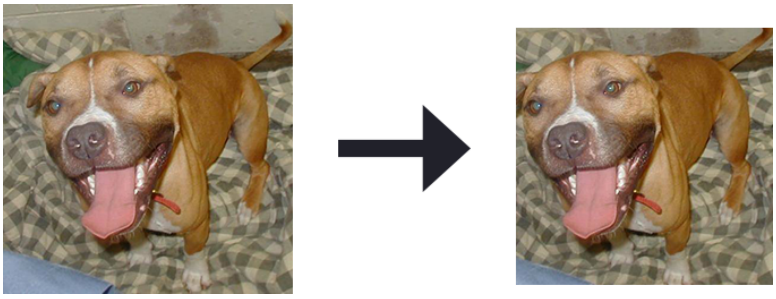


Figure 10.2: **Left:** Our original 256×256 input image. **Right:** *Randomly* cropping a 227×227 region from the image.

Applying this cropping implies that a network never sees the *exact* same image (unless by random happenstance), similar to data augmentation. As you know from our previous chapter, we constructed an HDF5 dataset of Kaggle Dogs vs. Cats images where each image is 256×256 pixels. However, the AlexNet architecture that we'll be implementing later in this chapter can only accept images of size 227×227 pixels.

So, what are we to do? Apply a `SimplePreprocessor` to resize our each of the 256×256 pixels down to 227×227? No, that would be wasteful, especially since this is an excellent opportunity to perform data augmentation by *randomly cropping* a 227×227 region from the 256×256 image during training – in fact, this process is exactly how Krizhevsky et al. trains AlexNet on the ImageNet dataset.

The `PatchPreprocessor`, just like all other image pre-processors, will be sorted in the `preprocessing` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |--- preprocessing
|    |    |--- __init__.py
|    |    |--- aspectawarepreprocessor.py
|    |    |--- imagetoarraypreprocessor.py
|    |    |--- meanpreprocessor.py
|    |    |--- patchpreprocessor.py
|    |    |--- simplepreprocessor.py
|    |--- utils
```

Open up the `patchpreprocessor.py` file and let's define the `PatchPreprocessor` class:

```
1  # import the necessary packages
2  from sklearn.feature_extraction.image import extract_patches_2d
3
4  class PatchPreprocessor:
5      def __init__(self, width, height):
6          # store the target width and height of the image
7          self.width = width
8          self.height = height
```

Line 5 defines the construct to `PatchPreprocessor` – we simply need to supply the target `width` and `height` of the cropped image.

We can then define the `preprocess` function:

```
10      def preprocess(self, image):
11          # extract a random crop from the image with the target width
12          # and height
13          return extract_patches_2d(image, (self.height, self.width),
14              max_patches=1)[0]
```

Extracting a random patches of size `self.width x self.height` is easy using the `extract_patches_2d` function from the scikit-learn library. Given an input `image`, this function randomly extracts a patch from image. Here we supply `max_patches=1`, indicating that we only need a *single* random patch from the input image.

The `PatchPreprocessor` class doesn't seem like much, but it's actually a very effective method to avoid overfitting by applying yet another layer of data augmentation. We'll be using the `PatchPreprocessor` when training AlexNet. The next pre-processor, `CropPreprocessor`, will be used when evaluating our trained network.

10.1.3 Crop Preprocessing

Next, we need to define a `CropPreprocessor` responsible for computing the 10-crops for over-sampling. During the evaluating phase of our CNN, we'll crop the four corners of the input image + the center region and then take their corresponding horizontal flips, for a total of ten samples per input image (Figure 10.3).

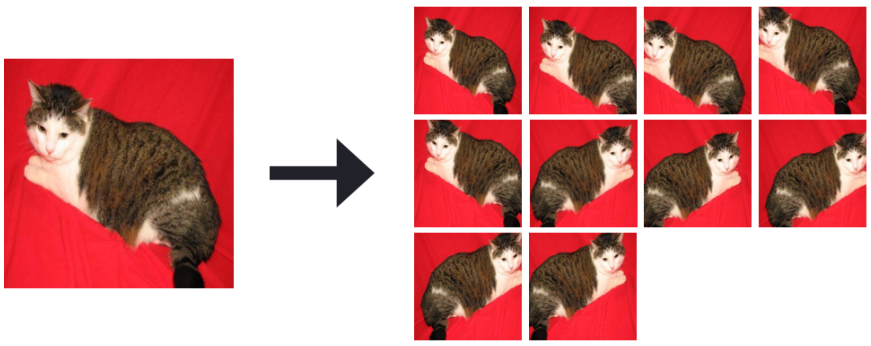


Figure 10.3: **Left:** The original 256×256 input image. **Right:** Applying the 10-crop preprocessor to extract ten 227×227 crops of the image including the center, four corners, and their corresponding horizontal mirrors.

These ten samples will be passed through the CNN, and then the probabilities averaged. Applying this over-sampling method tends to include 1-2 percent increases in classification accuracy (and in some cases, even higher).

The `CropPreprocessor` class will also live in the `preprocessing` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
```

```
|    |--- preprocessing
|    |    |--- __init__.py
|    |    |--- aspectawarepreprocessor.py
|    |    |--- croppreprocessor.py
|    |    |--- imagetoarraypreprocessor.py
|    |    |--- meanpreprocessor.py
|    |    |--- patchpreprocessor.py
|    |    |--- simplepreprocessor.py
|    |--- utils
```

Open up the `croppreprocessor.py` file and let's define it:

```
1  # import the necessary packages
2  import numpy as np
3  import cv2
4
5  class CropPreprocessor:
6      def __init__(self, width, height, horiz=True, inter=cv2.INTER_AREA):
7          # store the target image width, height, whether or not
8          # horizontal flips should be included, along with the
9          # interpolation method used when resizing
10         self.width = width
11         self.height = height
12         self.horiz = horiz
13         self.inter = inter
```

Line 6 defines the constructor to to `CropPreprocessor`. The only *required* arguments are the target `width` and `height` of each cropped region. We can also optionally specify whether horizontal flipping should be applied (defaults to `True`) along with the interpolation algorithm OpenCV will use for resizing. These arguments are all stored inside the class for use within the `preprocess` method.

Speaking of which, let's define the `preprocess` method now:

```
15     def preprocess(self, image):
16         # initialize the list of crops
17         crops = []
18
19         # grab the width and height of the image then use these
20         # dimensions to define the corners of the image based
21         (h, w) = image.shape[:2]
22         coords = [
23             [0, 0, self.width, self.height],
24             [w - self.width, 0, w, self.height],
25             [w - self.width, h - self.height, w, h],
26             [0, h - self.height, self.width, h]]
27
28         # compute the center crop of the image as well
29         dW = int(0.5 * (w - self.width))
30         dH = int(0.5 * (h - self.height))
31         coords.append([dW, dH, w - dW, h - dH])
```

The `preprocess` method requires only a single argument – the `image` which we are going to apply over-sampling. We grab the width and height of the input image on **Line 21**, which then

allows us to compute the (x, y)-coordinates of the four corners (top-left, top-right, bottom-right, bottom-left, respectively) on **Lines 22-26**. The center crop of the image is then computed on **Lines 29 and 30**, then added to the list of `coords` on **Line 31**.

We are now ready to extract each of the crops:

```
33          # loop over the coordinates, extract each of the crops,
34          # and resize each of them to a fixed size
35          for (startX, startY, endX, endY) in coords:
36               crop = image[startY:endY, startX:endX]
37               crop = cv2.resize(crop, (self.width, self.height),
38                    interpolation=self.inter)
39               crops.append(crop)
```

On **Line 35** we loop over each of the starting and ending (x, y)-coordinates of the rectangular crops. **Line 36** extracts the `crop` via NumPy array slicing which we then resize on **Line 37** to ensure the target `width` and `height` dimensions are met. The `crop` is the added to the `crops` list.

In the case that horizontal mirrors are to be computed, we can flip each of the five original crops, leaving us with ten crops overall:

```
41          # check to see if the horizontal flips should be taken
42          if self.horiz:
43               # compute the horizontal mirror flips for each crop
44               mirrors = [cv2.flip(c, 1) for c in crops]
45               crops.extend(mirrors)
46
47          # return the set of crops
48          return np.array(crops)
```

The array of `crops` is then returned to the calling function on **Line 48**. Using both the `MeanPreprocessor` for normalization and the `CropPreprocessor` for oversampling, we'll be able to obtain higher classification accuracy than is otherwise possible.

10.2 HDF5 Dataset Generators

Before we can implement the AlexNet architecture and train it on the Kaggle Dogs vs. Cats dataset, we first need to define a class responsible for yielding *batches* of images and labels from our HDF5 dataset. Chapter 9 discussed how to convert a set of images residing on disk into an HDF5 dataset – but how do we get them back out again?

The answer is to define an `HDF5DatasetGenerator` class in the `io` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |    |--- __init__.py
|    |    |--- hdf5datasetgenerator.py
|    |    |--- hdf5datasetwriter.py
|    |--- nn
|    |--- preprocessing
|    |--- utils
```

Previously, all of our image datasets could be loaded into memory so we could rely on Keras generator utilities to yield our batches of images and corresponding labels. However, now that our datasets are too large to fit into memory, we need to handle implementing this generator ourselves.

Go ahead and open the `hdf5datasetgenerator.py` file and we'll get to work:

```
1  # import the necessary packages
2  from keras.utils import np_utils
3  import numpy as np
4  import h5py
5
6  class HDF5DatasetGenerator:
7      def __init__(self, dbPath, batchSize, preprocessors=None,
8          aug=None, binarize=True, classes=2):
9          # store the batch size, preprocessors, and data augmentor,
10         # whether or not the labels should be binarized, along with
11         # the total number of classes
12         self.batchSize = batchSize
13         self.preprocessors = preprocessors
14         self.aug = aug
15         self.binarize = binarize
16         self.classes = classes
17
18         # open the HDF5 database for reading and determine the total
19         # number of entries in the database
20         self.db = h5py.File(dbPath)
21         self.numImages = self.db["labels"].shape[0]
```

On **Line 7** we define the constructor to our `HDF5DatasetGenerator`. This class accepts a number of arguments, two of which are required and the rest optional. I have detailed each of the arguments below:

- `dbPath`: The path to our HDF5 dataset that stores our images and corresponding class labels.
- `batchSize`: The size of mini-batches to yield when training our network.
- `preprocessors`: The list of image preprocessors we are going to apply (i.e., `MeanPreprocessor`, `ImageToArrayPreprocessor`, etc.).
- `aug`: Defaulting to `None`, we could also supply a Keras `ImageDataGenerator` to apply data augmentation *directly inside* our `HDF5DatasetGenerator`.
- `binarize`: Typically we will store class labels as *single integers* inside our HDF5 dataset; however, as we know, if we are applying categorical cross-entropy or binary cross-entropy as our loss function, we first need to *binarize* the labels as one-hot encoded vectors – this switch indicates whether or not this binarization needs to take place (which defaults to `True`).
- `classes`: The number of unique class labels in our dataset. This value is required to accurately construct our one-hot encoded vectors during the binarization phase.

These variables are stored on **Lines 12-16** so we can access them from the rest of the class. **Line 20** opens a file pointer to our HDF5 dataset file **Line 21** creates a convenience variable used to access the total number of data points in the dataset.

Next, we need to define a `generator` function, which as the name suggests, is responsible for yielding batches of images and class labels to the Keras `.fit_generator` function when training a network:

```
23     def generator(self, passes=np.inf):
24         # initialize the epoch count
```

```
25          epochs = 0
26
27          # keep looping infinitely -- the model will stop once we have
28          # reach the desired number of epochs
29          while epochs < passes:
30               # loop over the HDF5 dataset
31               for i in np.arange(0, self.numImages, self.batchSize):
32                    # extract the images and labels from the HDF dataset
33                    images = self.db["images"][i: i + self.batchSize]
34                    labels = self.db["labels"][i: i + self.batchSize]
```

Line 23 defines the `generator` function which can accept an optional argument, `passes`. Think of the `passes` value as the total number of *epochs* – in *most cases*, we don't want our generator to be concerned with the total number of epochs; our training methodology (fixed number of epochs, early stopping, etc.) should be responsible for that. However, in certain situations, it's often helpful to provide this information to the generator.

On **Line 29** we start looping over the number of desired epochs – by default, this loop will run indefinitely until either:

1. Keras reaches training termination criteria.
2. We explicitly stop the training process (i.e., ctrl + c).

Line 31 starts looping over each batch of data points in the dataset. We extract the `images` and `labels` of size `batchSize` from our HDF5 dataset on **Lines 33 and 34**.

Next, let's check to see if the `labels` should be one-hot encoded:

```
36                    # check to see if the labels should be binarized
37                    if self.binarize:
38                         labels = np_utils.to_categorical(labels,
39                              self.classes)
```

We can then also see if any image `preprocessors` should be applied:

```
41                    # check to see if our preprocessors are not None
42                    if self.preprocessors is not None:
43                         # initialize the list of processed images
44                         procImages = []
45
46                         # loop over the images
47                         for image in images:
48                              # loop over the preprocessors and apply each
49                              # to the image
50                              for p in self.preprocessors:
51                                   image = p.preprocess(image)
52
53                              # update the list of processed images
54                              procImages.append(image)
55
56                         # update the images array to be the processed
57                         # images
58                         images = np.array(procImages)
```

Provided the `preprocessors` is not `None` (**Line 42**), we loop over each of the images in the batch and apply each of the `preprocessors` by calling the `preprocess` method on the individual `image`. Doing this enables us to *chain together* multiple image pre-processors.

For example, our first pre-processor may resize the image to a fixed size via our `SimplePreprocessor` class. From there we may perform mean subtraction via the `MeanPreprocessor`. And after that, we'll need to convert the image to a Keras-compatible array using the `ImageToArrayPreprocessor`. At this point it should be clear why we defined all of our pre-processing classes with a `preprocess` method – it allows us to *chain* our pre-processors together inside the data generator. The preprocessed images are then converted back to a NumPy array on **Line 58**.

Provided we supplied an instance of `aug`, an `ImageDataGenerator` class used for data augmentation, we'll also want to apply data augmentation to the images as well:

```
60                      # if the data augmenator exists, apply it
61                      if self.aug is not None:
62                          (images, labels) = next(self.aug.flow(images,
63                              labels, batch_size=self.batchSize))
```

Finally, we can yield a 2-tuple of the batch of `images` and `labels` to the calling Keras generator:

```
65                      # yield a tuple of images and labels
66                      yield (images, labels)
67
68                  # increment the total number of epochs
69                  epochs += 1
70
71      def close(self):
72          # close the database
73          self.db.close()
```

Line 69 increments our total number of epochs after all mini-batches in the dataset have been processed. The `close` method on **Lines 71-73** is simply responsible for closing the pointer to the HDF5 dataset.

Admittedly, implementing the `HDF5DatasetGenerator` may not "feel" like we're doing any deep learning. After all, isn't this just a class responsible for yielding batches of data from a file? Technically, yes, that is correct. However, keep in mind that *practical* deep learning is more than just defining a model architecture, initializing an optimizer, and applying it to a dataset.

In reality, we need *extra tools* to help facilitate our ability to work with datasets, *especially* datasets that are too large to fit into memory. As we'll see throughout the rest of this book, our `HDF5DatasetGenerator` will come in handy a number of times – and when you start creating your own deep learning applications/experiments, you'll feel quite lucky to have it in your repertoire.

10.3 Implementing AlexNet

Let's now move on to implement the seminal AlexNet architecture by Krizhevsky et al. A table summarizing the AlexNet architecture can be seen in Table 10.1.

Notice how our input images are assumed to be $227 \times 227 \times 3$ pixels – this is actually the *correct* input size for AlexNet. As mentioned in Chapter 9, in the original publication, Krizhevsky et al. reported the input spatial dimensions to be $224 \times 224 \times 3$; however, since we know 224×224 cannot possible by tiled with an 11×11 kernel, we assume there was likely a typo in the publication, and 224×224 should actually be 227×227.

The first block of AlexNet applies 96, 11×11 kernels with a stride of 4×4, followed by a RELU activation and max pooling with a pool size of 3×3 and strides of 2×2, resulting in an output volume of size 55×55.

Layer Type	**Output Size**	**Filter Size / Stride**
INPUT IMAGE	$227 \times 227 \times 3$	
CONV	$55 \times 55 \times 96$	$11 \times 11/4 \times 4, K = 96$
ACT	$55 \times 55 \times 96$	
BN	$55 \times 55 \times 96$	
POOL	$27 \times 27 \times 96$	$3 \times 3/2 \times 2$
DROPOUT	$27 \times 27 \times 96$	
CONV	$27 \times 27 \times 256$	$5 \times 5, K = 256$
ACT	$27 \times 27 \times 256$	
BN	$27 \times 27 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$13 \times 13 \times 256$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 256$	$3 \times 3, K = 256$
ACT	$13 \times 13 \times 256$	
BN	$13 \times 13 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$6 \times 6 \times 256$	
FC	4096	
ACT	4096	
BN	4096	
DROPOUT	4096	
FC	4096	
ACT	4096	
BN	4096	
DROPOUT	4096	
FC	1000	
SOFTMAX	1000	

Table 10.1: A table summary of the AlexNet architecture. Output volume sizes are included for each layer, along with convolutional filter size/pool size when relevant.

We then apply a second `CONV => RELU => POOL` layer this, this time using 256, 5×5 filters with 1×1 strides. After applying max pooling again with a pool size of 3×3 and strides of 2×2 we are left with a 13×13 volume.

Next, we apply `(CONV => RELU) * 3 => POOL`. The first two `CONV` layers learn 384, 3×3 filters while the final `CONV` learns 256, 3×3 filters.

After another max pooling operation, we reach our two `FC` layers, each with 4096 nodes and RELU activations in between. The final layer in the network is our softmax classifier.

When AlexNet was first introduced we did not have techniques such as batch normalization – in our implementation, we are going to include batch normalization after the activation, as is standard for the majority of image classification tasks using Convolutional Neural Networks. We'll also include a very small amount of dropout after each `POOL` operation to further help reduce overfitting.

To implement AlexNet, let's create a new file named `alexnet.py` in the `conv` sub-module of `nn` in `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- alexnet.py
...
|    |--- preprocessing
|    |--- utils
```

From there, open up `alexnet.py`, and we'll implement this seminal architecture:

```
1  # import the necessary packages
2  from keras.models import Sequential
3  from keras.layers.normalization import BatchNormalization
4  from keras.layers.convolutional import Conv2D
5  from keras.layers.convolutional import MaxPooling2D
6  from keras.layers.core import Activation
7  from keras.layers.core import Flatten
8  from keras.layers.core import Dropout
9  from keras.layers.core import Dense
10 from keras.regularizers import l2
11 from keras import backend as K
```

Lines 2-11 import our required Keras classes – we have used all of these layers before in previous chapters of this book so I'm going to skip explicitly describing each of them. The only import I *do* want to draw your attention to is **Line 10** where we import the `l2` function – this method will be responsible for applying L2 weight decay to the weight layers in the network.

Now that our imports are taken care of, let's start the definition of AlexNet:

```
13 class AlexNet:
14     @staticmethod
15     def build(width, height, depth, classes, reg=0.0002):
16         # initialize the model along with the input shape to be
```

```
17          # "channels last" and the channels dimension itself
18          model = Sequential()
19          inputShape = (height, width, depth)
20          chanDim = -1
21
22          # if we are using "channels first", update the input shape
23          # and channels dimension
24          if K.image_data_format() == "channels_first":
25               inputShape = (depth, height, width)
26               chanDim = 1
```

Line 15 defines the `build` method of `AlexNet`. Just like in all previous examples in this book, the `build` method is required for constructing the actual network architecture and returning it to the calling function. This method accepts four arguments: the `width`, `height`, and `depth` of the input images, followed by the total number of `class` labels in the dataset. An optional parameter, `reg`, controls the amount of L2 regularization we'll be applying to the network. For larger, deeper networks, applying regularization is *critical* to reducing overfitting while increasing accuracy on the validation and testing sets.

Line 18 initializes the `model` itself along with the `inputShape` and channel dimension assuming we are using "channels last" ordering. If we are instead using "channels first" ordering, we update `inputShape` and `chanDim` (**Lines 24-26**).

Let's now define the first `CONV => RELU => POOL` layer set in the network:

```
28          # Block #1: first CONV => RELU => POOL layer set
29          model.add(Conv2D(96, (11, 11), strides=(4, 4),
30               input_shape=inputShape, padding="same",
31               kernel_regularizer=l2(reg)))
32          model.add(Activation("relu"))
33          model.add(BatchNormalization(axis=chanDim))
34          model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
35          model.add(Dropout(0.25))
```

Our first `CONV` layer will learn 96 filters, each of size 11×11 (**Lines 28 and 29**), using a stride of 4×4. By applying the `kernel_regularizer` parameter to the `Conv2D` class, we can apply our L2 weight regularization parameter – this regularization will be applied to *all* `CONV` and `FC` layers in the network.

A ReLU activation is applied after our `CONV`, followed by a `BatchNormalization` (**Lines 32 and 33**). The `MaxPooling2D` is then applied to reduce our spatial dimensions (**Line 34**). We'll also apply dropout with a small probability (25 percent) to help reduce overfitting (**Lines 35**).

The following code block defines another `CONV => RELU => POOL` layer set, this time learning 256 filters, each of size 5×5:

```
37          # Block #2: second CONV => RELU => POOL layer set
38          model.add(Conv2D(256, (5, 5), padding="same",
39               kernel_regularizer=l2(reg)))
40          model.add(Activation("relu"))
41          model.add(BatchNormalization(axis=chanDim))
42          model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
43          model.add(Dropout(0.25))
```

Deeper, richer features are learned in the third block of AlexNet where we stack *multiple* `CONV => RELU` together prior to applying a `POOL` operation:

```
45            # Block #3: CONV => RELU => CONV => RELU => CONV => RELU
46            model.add(Conv2D(384, (3, 3), padding="same",
47                 kernel_regularizer=l2(reg)))
48            model.add(Activation("relu"))
49            model.add(BatchNormalization(axis=chanDim))
50            model.add(Conv2D(384, (3, 3), padding="same",
51                 kernel_regularizer=l2(reg)))
52            model.add(Activation("relu"))
53            model.add(BatchNormalization(axis=chanDim))
54            model.add(Conv2D(256, (3, 3), padding="same",
55                 kernel_regularizer=l2(reg)))
56            model.add(Activation("relu"))
57            model.add(BatchNormalization(axis=chanDim))
58            model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
59            model.add(Dropout(0.25))
```

The first two `CONV` filters learn 384, 3×3 filters while the third `CONV` learns 256, 3×3 filters. Again, stacking multiple `CONV => RELU` layers on top of each other *prior* to applying a destructive `POOL` layer enables our network to learn richer, and potentially more discriminating features.

From there we collapse our multi-dimensional representation down into a standard feedforward network using two fully-connected layers (4096 nodes each):

```
61            # Block #4: first set of FC => RELU layers
62            model.add(Flatten())
63            model.add(Dense(4096, kernel_regularizer=l2(reg)))
64            model.add(Activation("relu"))
65            model.add(BatchNormalization())
66            model.add(Dropout(0.5))
67
68            # Block #5: second set of FC => RELU layers
69            model.add(Dense(4096, kernel_regularizer=l2(reg)))
70            model.add(Activation("relu"))
71            model.add(BatchNormalization())
72            model.add(Dropout(0.5))
```

Batch normalization is applied after each activation in the `FC` layer sets, just as in the `CONV` layers above. Dropout, with a larger probability of 50 percent, is applied after every `FC` layer set, as is standard with the vast majority of CNNs.

Finally, we define the softmax classifier using the desired number of `classes` and return the resulting `model` to the calling function:

```
74             # softmax classifier
75            model.add(Dense(classes, kernel_regularizer=l2(reg)))
76            model.add(Activation("softmax"))
77
78            # return the constructed network architecture
79            return model
```

As you can see, implementing AlexNet is a fairly straightforward process, *especially* when you have the "blueprint" of the architecture presented in Table 10.1 above. Whenever implementing architectures from publications, try to see if they provide such a table as it makes implementation

much easier. For your own network architectures, use Chapter 19 of the *Starter Bundle* on visualizing network architectures to aid you in ensuring your input volume and output volume sizes are what you expect.

10.4 Training AlexNet on Kaggle: Dogs vs. Cats

Now that the AlexNet architecture has been defined, let's apply it to the Kaggle Dogs vs. Cats challenge. Open up a new file, name it `train_alexnet.py`, and insert the following code:

```
1  # import the necessary packages
2  # set the matplotlib backend so figures can be saved in the background
3  import matplotlib
4  matplotlib.use("Agg")
5
6  # import the necessary packages
7  from config import dogs_vs_cats_config as config
8  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
9  from pyimagesearch.preprocessing import SimplePreprocessor
10 from pyimagesearch.preprocessing import PatchPreprocessor
11 from pyimagesearch.preprocessing import MeanPreprocessor
12 from pyimagesearch.callbacks import TrainingMonitor
13 from pyimagesearch.io import HDF5DatasetGenerator
14 from pyimagesearch.nn.conv import AlexNet
15 from keras.preprocessing.image import ImageDataGenerator
16 from keras.optimizers import Adam
17 import json
18 import os
```

Lines 3 and 4 import matplotlib, while ensuring the backend is set such that we can save figures and plots to disk as our network trains. We then implement our pre-processors on **Lines 8-11**. The `HDF5DatasetGenerator` is then imported on **Line 13** so we can access batches of training data from our serialized HDF5 dataset. AlexNet is also implemented on **Line 14**.

Our next code block handles initializing our data augmentation generator via the `ImageDataGenerator` class:

```
20 # construct the training image generator for data augmentation
21 aug = ImageDataGenerator(rotation_range=20, zoom_range=0.15,
22     width_shift_range=0.2, height_shift_range=0.2, shear_range=0.15,
23     horizontal_flip=True, fill_mode="nearest")
```

Let's take the time now to initialize each of our image pre-processors:

```
25 # load the RGB means for the training set
26 means = json.loads(open(config.DATASET_MEAN).read())
27
28 # initialize the image preprocessors
29 sp = SimplePreprocessor(227, 227)
30 pp = PatchPreprocessor(227, 227)
31 mp = MeanPreprocessor(means["R"], means["G"], means["B"])
32 iap = ImageToArrayPreprocessor()
```

On **Line 26** we load the serialized RGB means from disk – these are the means for each of the respective Red, Green, and Blue channels across our training dataset. These values will later be passed into a `MeanPreprocessor` for mean subtraction normalization.

Make sure you copy your `dogs_vs_cats_mean.json` file over from the previous chapter!

Line 29 instantiates a `SimplePreprocessor` used to resize an input image down to 227×227 pixels. This pre-processor will be used in the *validation* data generator as our input images are 256×256 pixels; however, AlexNet is intended to handle only 227×227 images (hence why we need to resize the image during validation).

Line 30 instantiates a `PatchPreprocessor` – this pre-processor will randomly sample 227×227 regions from the 256×256 input images during training time, serving as a second form of data augmentation.

We then initialize the `MeanPreprocessor` on **Line 31** using our respective, Red, Green, and Blue averages. Finally, the `ImageToArrayPreprocessor` (**Line 32**) is used to convert images to Keras-compatible arrays.

Given our pre-processors, let's define the `HDF5DatasteGenerator` for both the training and validation data:

```
34 # initialize the training and validation dataset generators
35 trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, 128, aug=aug,
36     preprocessors=[pp, mp, iap], classes=2)
37 valGen = HDF5DatasetGenerator(config.VAL_HDF5, 128,
38     preprocessors=[sp, mp, iap], classes=2)
```

Lines 35 and 36 create our *training* dataset generator. Here we supply the path to our training HDF5 file, indicating that we should use batch sizes of 128 images, data augmentation, and three pre-processors: patch, mean, and image to array, respectively.

Lines 37 and 38 are responsible for instantiating the *testing* generator. This time we'll supply the path to the validation HDF5 file, use a batch size of 128, *no data augmentation*, and a simple pre-processor rather than a patch pre-processor (since data augmentation is not applied to validation data).

Finally, we are ready to initialize the Adam optimizer and AlexNet architecture:

```
40 # initialize the optimizer
41 print("[INFO] compiling model...")
42 opt = Adam(lr=1e-3)
43 model = AlexNet.build(width=227, height=227, depth=3,
44     classes=2, reg=0.0002)
45 model.compile(loss="binary_crossentropy", optimizer=opt,
46     metrics=["accuracy"])
47
48 # construct the set of callbacks
49 path = os.path.sep.join([config.OUTPUT_PATH, "{}.png".format(
50     os.getpid())])
51 callbacks = [TrainingMonitor(path)]
```

On **Line 42** we instantiate the Adam optimizer using the default learning rate of 0.001. The reason I choose Adam for this experiment (rather than SGD) is two-fold:

1. I wanted to give you exposure to using the more advanced optimizers we covered in Chapter 7.
2. Adam performs better on this classification task than SGD (which I know from the multiple previous experiments I ran before publishing this book).

We then initialize AlexNet on **Lines 43 and 44**, indicating that each input image will have a width of 227 pixels, a height of 227 pixels, 3 channels, and the dataset itself will have two classes (one for dogs, and another for cats). We'll also apply a small regularization penalty of `0.0002` to help combat overfitting and increase the ability of our model to generalize to the testing set.

We'll use *binary* cross-entropy rather than *categorical* cross-entropy (**Lines 45 and 46**) as this is only a two-class classification problem. We'll also define a `TrainingMonitor` callback on **Line 51** so we can monitor the performance of our network as it trains.

Speaking of training the network, let's do that now:

```
53 # train the network
54 model.fit_generator(
55 	trainGen.generator(),
56 	steps_per_epoch=trainGen.numImages // 128,
57 	validation_data=valGen.generator(),
58 	validation_steps=valGen.numImages // 128,
59 	epochs=75,
60 	max_queue_size=10,
61 	callbacks=callbacks, verbose=1)
```

To train AlexNet on the Kaggle Dogs vs. Cats dataset using our `HDF5DatasetGenerator`, we need to use the `fit_generator` method of the `model`. First, we pass in `trainGen.generator()`, the HDF5 generator used to construct mini-batches of training data (**Line 55**). To determine the number of batches per epoch, we divide the total number of images in the training set by our batch size (**Line 56**). We do the same on **Lines 57 and 58** for the validation data. Finally, we'll indicate that AlexNet is to be trained for 75 epochs.

The last step is to simply serialize our `model` to file after training, along with closing each of the training and testing HDF5 datasets, respectively:

```
63 # save the model to file
64 print("[INFO] serializing model...")
65 model.save(config.MODEL_PATH, overwrite=True)
66
67 # close the HDF5 datasets
68 trainGen.close()
69 valGen.close()
```

To train AlexNet on the Kaggle Dogs vs. Cats dataset, execute the following command:

```
$ python train_alexnet.py
...
Epoch 73/75
415s - loss: 0.4862 - acc: 0.9126 - val_loss: 0.6826 - val_acc: 0.8602
Epoch 74/75
408s - loss: 0.4865 - acc: 0.9166 - val_loss: 0.6894 - val_acc: 0.8721
Epoch 75/75
401s - loss: 0.4813 - acc: 0.9166 - val_loss: 0.4195 - val_acc: 0.9297
[INFO] serializing model...
```

A plot of the training and validation loss/accuracy over the 75 epochs can be seen in Figure 10.4. Overall we can see that the training and accuracy plots correlate well with each other, although we could help stabilize variations in validation loss towards the end of the 75 epoch cycle by applying a bit of learning rate decay. Examining the classification report of AlexNet on the Dogs vs. Cats dataset, we see our obtained obtained **92.97**% on the validation set.

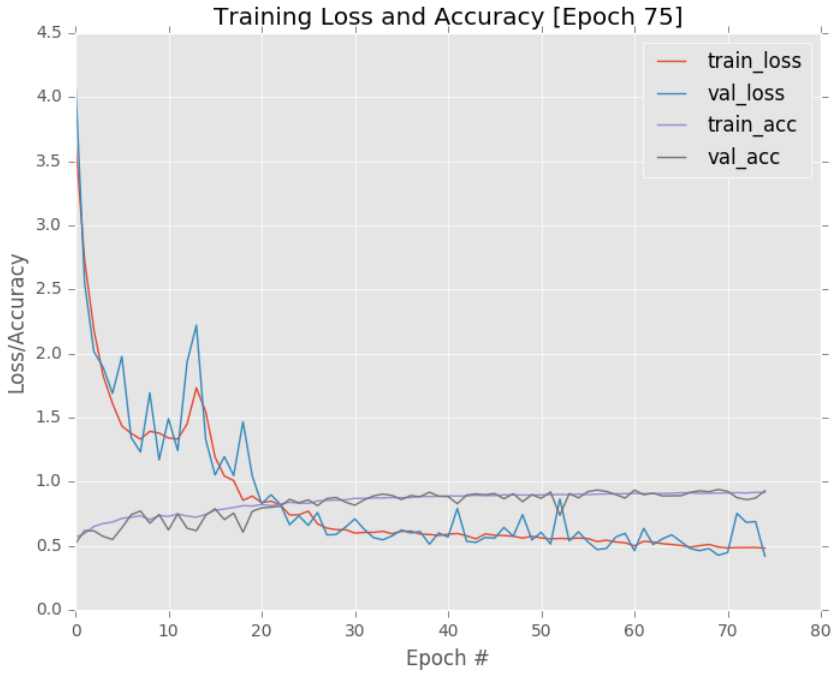


Figure 10.4: Training AlexNet on the Kaggle Dogs vs. Cats competition where we obtain 92.97% classification accuracy on our validation set. Our learning curve is stable with changes in training accuracy/loss being reflected in the respective validation split.

In the next section, we'll evaluate AlexNet on the testing set using both the standard method and over-sampling method. As our results will demonstrate, using over-sampling can increase your classification from 1-3% depending on your dataset and network architecture.

10.5 Evaluating AlexNet

To evaluate AlexNet on the testing set using both our standard method and over-sampling technique, let's create a new file named `crop_accuracy.py`:

```
--- dogs_vs_cats
|     |--- config
|     |--- build_dogs_vs_cats.py
|     |--- crop_accuracy.py
|     |--- extract_features.py
|     |--- train_alexnet.py
|     |--- train_model.py
|     |--- output
```

From there, open `crop_accuracy.py` and insert the following code:

```
# import the necessary packages
from config import dogs_vs_cats_config as config
from pyimagesearch.preprocessing import ImageToArrayPreprocessor
from pyimagesearch.preprocessing import SimplePreprocessor
from pyimagesearch.preprocessing import MeanPreprocessor
from pyimagesearch.preprocessing import CropPreprocessor
from pyimagesearch.io import HDF5DatasetGenerator
from pyimagesearch.utils.ranked import rank5_accuracy
from keras.models import load_model
import numpy as np
import progressbar
import json
```

Lines 2-12 import our required Python packages. **Line 2** imports our Python configuration file for the Dogs vs. Cats challenge. We'll also import our image preprocessors on **Lines 3-6**, including the `ImageToArrayPreprocessor`, `SimplePreprocessor`, `MeanPreprocessor`, and `CropPreprocessor`. The `HDF5DatasetGenerator` is required so we can access the *testing set* of our dataset and obtain predictions on this data using our pre-trained model.

Now that our imports are complete, let's load the RGB means from disk, initialize our image pre-preprocessors, and load the pre-trained AlexNet network:

```
# load the RGB means for the training set
means = json.loads(open(config.DATASET_MEAN).read())

# initialize the image preprocessors
sp = SimplePreprocessor(227, 227)
mp = MeanPreprocessor(means["R"], means["G"], means["B"])
cp = CropPreprocessor(227, 227)
iap = ImageToArrayPreprocessor()

# load the pretrained network
print("[INFO] loading model...")
model = load_model(config.MODEL_PATH)
```

Before we apply over-sampling and 10-cropping, let's first obtain a baseline on the testing set using only the original testing image as input to our network:

```
# initialize the testing dataset generator, then make predictions on
# the testing data
print("[INFO] predicting on test data (no crops)...")
testGen = HDF5DatasetGenerator(config.TEST_HDF5, 64,
     preprocessors=[sp, mp, iap], classes=2)
predictions = model.predict_generator(testGen.generator(),
     steps=testGen.numImages // 64, max_queue_size=10)

# compute the rank-1 and rank-5 accuracies
(rank1, _) = rank5_accuracy(predictions, testGen.db["labels"])
print("[INFO] rank-1: {:.2f}%".format(rank1 * 100))
testGen.close()
```

Lines 30 and 31 initialize the `HDF5DatasetGenerator` to access the testing dataset in batches of 64 images. Since we are obtaining a *baseline*, we'll use only the `SimplePreprocessor` to

resize the 256×256 input images down to 227×227 pixels, followed by mean normalization and converting the batch to a Keras-compatible array of images. **Lines 32 and 33** then use the generator to evaluate AlexNet on the dataset.

Given our `predictions`, we can compute our accuracy on the test set (**Lines 36-38**). Notice here how we only care about the `rank1` accuracy, which is because the Dogs vs. Cats is a 2-class dataset – computing the rank-5 accuracy for a 2-class dataset would trivially report 100 percent classification accuracy.

Now that we have a baseline for the standard evaluation technique, let's move on to over-sampling:

```
40 # re-initialize the testing set generator, this time excluding the
41 # `SimplePreprocessor`
42 testGen = HDF5DatasetGenerator(config.TEST_HDF5, 64,
43      preprocessors=[mp], classes=2)
44 predictions = []
45
46 # initialize the progress bar
47 widgets = ["Evaluating: ", progressbar.Percentage(), " ",
48      progressbar.Bar(), " ", progressbar.ETA()]
49 pbar = progressbar.ProgressBar(maxval=testGen.numImages // 64,
50      widgets=widgets).start()
```

On **Lines 42 and 43** we *re-initialize* the `HDF5DatasetGenerator`, this time instructing it to use *just* the `MeanPreprocessor` – we'll apply both over-sampling and Keras-array conversion later in the pipeline. **Lines 47-50** also initialize `progressbar` widgets to our screen if we are interested in having the evaluating progress displayed to our screen.

Given the re-instantiated `testGen`, we are now ready to apply the 10-cropping technique:

```
52 # loop over a single pass of the test data
53 for (i, (images, labels)) in enumerate(testGen.generator(passes=1)):
54      # loop over each of the individual images
55      for image in images:
56           # apply the crop preprocessor to the image to generate 10
57           # separate crops, then convert them from images to arrays
58           crops = cp.preprocess(image)
59           crops = np.array([iap.preprocess(c) for c in crops],
60                dtype="float32")
61
62           # make predictions on the crops and then average them
63           # together to obtain the final prediction
64           pred = model.predict(crops)
65           predictions.append(pred.mean(axis=0))
66
67      # update the progress bar
68      pbar.update(i)
```

On **Line 53** we start looping over every batch of images in the testing generator. Typically an `HDF5DatasetGenerator` is set to loop forever until we explicitly tell it to stop (normally by setting a maximum number of iterations via Keras when training); however, since we are now *evaluating*, we can supply `passes=1` to indicate the testing data only needs to be looped over once.

Then, for each image in the `images` batch (**Line 55**), we apply the 10-crop pre-processor on **Line 58**, which converts the `image` into an array of ten 227×227 images. These 227×227 crops were extracted from the original 256×256 batch based on the:

- Center of the image
- Top-left corner
- Top-right corner
- Bottom-right corner
- Bottom-left corner
- Corresponding horizontal flips

Once we have the `crops`, we pass them through the `model` on **Line 64** for prediction. The final prediction (**Line 65**) is the *average of the probabilities* across all ten crops.

Our final code block handles displaying the accuracy of the over-sampling method:

```
70 # compute the rank-1 accuracy
71 pbar.finish()
72 print("[INFO] predicting on test data (with crops)...")
73 (rank1, _) = rank5_accuracy(predictions, testGen.db["labels"])
74 print("[INFO] rank-1: {:.2f}%".format(rank1 * 100))
75 testGen.close()
```

To evaluate AlexNet on the Kaggle Dog vs. Cats dataset, just execute the following command:

```
$ python crop_accuracy.py
[INFO] loading model...
[INFO] predicting on test data (no crops)...
[INFO] rank-1: 92.60%
Evaluating: 100% |####################################| Time: 0:01:12
[INFO] predicting on test data (with crops)...
[INFO] rank-1: 94.00%
```

As our results demonstrate, we reach **92.60%** accuracy on the testing set. However, by applying the 10-crop over-sampling method, we are able to boost classification accuracy to **94.00%**, an increase of 1.4%, which this was all accomplished simply by taking multiple crops of the input image and averaging the results. This straightforward, uncomplicated trick is an easy way to eke out an extra few percentage points when evaluating your network.

10.6 Obtaining a Top-5 Spot on the Kaggle Leaderboard

Of course, if you were to look at the Kaggle Dogs vs. Cats leaderboard, you would notice that to even break into the top-25 position we would need 96.69% accuracy, which our current method is not capable of reaching. So, what's the solution?

The answer is *transfer learning*, specifically transfer learning via feature extraction. While the ImageNet dataset consists of 1,000 object categories, a good portion of those include both *dog species* and *cat species*. Therefore, a network trained on ImageNet could not only tell you if an image was of a *dog* or a *cat*, but what particular *breed* the animal is as well. Given that a network trained on ImageNet must be able to discriminate between such fine-grained animals, it's natural to hypothesize that the features extracted from a pre-trained network would likely lend itself well to claiming a top spot on the Kaggle Dogs vs. Cats leaderboard.

To test this hypothesis, let's first extract features from the pre-trained ResNet architecture and then train a Logistic Regression classifier on top of these features.

10.6.1 Extracting Features Using ResNet

The transfer learning via feature extraction technique we'll be using in this section is heavily based on Chapter 3. I'll review the entire contents of `extract_features.py` as a matter of completeness; however, please refer to Chapter 3 if you require further knowledge on feature extraction using CNNs.

To get started, open up a new file, name it `extract_features.py`, and insert the following code:

```
1  # import the necessary packages
2  from keras.applications import ResNet50
3  from keras.applications import imagenet_utils
4  from keras.preprocessing.image import img_to_array
5  from keras.preprocessing.image import load_img
6  from sklearn.preprocessing import LabelEncoder
7  from pyimagesearch.io import HDF5DatasetWriter
8  from imutils import paths
9  import numpy as np
10 import progressbar
11 import argparse
12 import random
13 import os
```

Lines 2-13 import our required Python packages. We import the `ResNet50` class on **Line 2** so we can access the pre-trained ResNet architecture. We'll also use the `HDF5DatasetWriter` on **Line 7** so we can write the extracted features to an efficiently HDF5 file format.

From there, 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("-o", "--output", required=True,
20      help="path to output HDF5 file")
21 ap.add_argument("-b", "--batch-size", type=int, default=16,
22      help="batch size of images to be passed through network")
23 ap.add_argument("-s", "--buffer-size", type=int, default=1000,
24      help="size of feature extraction buffer")
25 args = vars(ap.parse_args())
26
27 # store the batch size in a convenience variable
28 bs = args["batch_size"]
```

We only need two required command line arguments here, `--dataset`, which is the path to the input dataset of Dogs vs. Cats images, along with `--output`, the path to the output HDF5 file containing the features extracted via ResNet.

Next, let's grab the paths to the Dogs vs. Cats images residing on disk and then use the file paths to extract the label names:

```
30 # grab the list of images that we'll be describing then randomly
31 # shuffle them to allow for easy training and testing splits via
32 # array slicing during training time
```

```
print("[INFO] loading images...")
imagePaths = list(paths.list_images(args["dataset"]))
random.shuffle(imagePaths)

# extract the class labels from the image paths then encode the
# labels
labels = [p.split(os.path.sep)[-1].split(".")[0] for p in imagePaths]
le = LabelEncoder()
labels = le.fit_transform(labels)
```

Now we can load our pre-trained ResNet50 weights from disk (excluding the `FC` layers):

```
# load the ResNet50 network
print("[INFO] loading network...")
model = ResNet50(weights="imagenet", include_top=False)
```

In order to store the features extracted from ResNet50 to disk, we need to instantiate a `HDF5DatasetWriter` object:

```
# initialize the HDF5 dataset writer, then store the class label
# names in the dataset
dataset = HDF5DatasetWriter((len(imagePaths), 100352),
    args["output"], dataKey="features", bufSize=args["buffer_size"])
dataset.storeClassLabels(le.classes_)
```

The final average pooling layer of ResNet50 is 2048-d, hence why we supply a value of 2048 as the dimensionality to our `HDF5datasetWriter`.

We'll also initialize a `progressbar` so we can keep track of the feature extraction process:

```
# initialize the progress bar
widgets = ["Extracting Features: ", progressbar.Percentage(), " ",
    progressbar.Bar(), " ", progressbar.ETA()]
pbar = progressbar.ProgressBar(maxval=len(imagePaths),
    widgets=widgets).start()
```

Extracting features from a dataset using a CNN is the same as it was in Chapter 3. First, we loop over the `imagePaths` in batches:

```
# loop over the images in batches
for i in np.arange(0, len(imagePaths), bs):
    # extract the batch of images and labels, then initialize the
    # list of actual images that will be passed through the network
    # for feature extraction
    batchPaths = imagePaths[i:i + bs]
    batchLabels = labels[i:i + bs]
    batchImages = []
```

Followed by pre-processing each image:

```
68      # loop over the images and labels in the current batch
69      for (j, imagePath) in enumerate(batchPaths):
70          # load the input image using the Keras helper utility
71          # while ensuring the image is resized to 224x224 pixels
72          image = load_img(imagePath, target_size=(224, 224))
73          image = img_to_array(image)
74
75          # preprocess the image by (1) expanding the dimensions and
76          # (2) subtracting the mean RGB pixel intensity from the
77          # ImageNet dataset
78          image = np.expand_dims(image, axis=0)
79          image = imagenet_utils.preprocess_input(image)
80
81          # add the image to the batch
82          batchImages.append(image)
```

And then passing the `batchImages` through the network architecture, enabling us to extract features from the final `POOL` layer of ResNet50:

```
84      # pass the images through the network and use the outputs as
85      # our actual features
86      batchImages = np.vstack(batchImages)
87      features = model.predict(batchImages, batch_size=bs)
88
89      # reshape the features so that each image is represented by
90      # a flattened feature vector of the `MaxPooling2D` outputs
91      features = features.reshape((features.shape[0], 100352))
```

These extracted `features` are then added to our `dataset`:

```
93      # add the features and labels to our HDF5 dataset
94      dataset.add(features, batchLabels)
95      pbar.update(i)
96
97  # close the dataset
98  dataset.close()
99  pbar.finish()
```

In Keras v2.2.0 and greater the output dimensions of ResNet is $2,048 \times 7 \times 7 = 100,352$ rather than previous versions which were $2,048$. If you are using a version of Keras prior to v2.2.0+ make sure you update the `100352` in both the `.reshape` and `HDF5DatasetWriter` calls above to be `2048`.

To utilize ResNet to extract features from the Dogs vs. Cats dataset, simply execute the following command:

```
$ python extract_features.py --dataset ../datasets/kaggle_dogs_vs_cats/train \
      --output ../datasets/kaggle_dogs_vs_cats/hdf5/features.hdf5
[INFO] loading images...
[INFO] loading network...
Extracting Features: 100% |#####################################| Time: 0:06:18
```

After the command finishes executing, you should now have a file named `features.hdf5` in your `hdf5` directory:

```
$ ls -l ../datasets/kaggle_dogs_vs_cats/hdf5/features.hdf5
-rw-rw-r-- adrian 409806272 Jun  3 07:17 features.hdf5
```

Given these features, we can train a Logistic Regression classifier on top of them to (ideally) obtain a top-5 spot on the Kaggle Dogs vs. Cats leaderboard.

10.6.2 Training a Logistic Regression Classifier

To train our Logistic Regression classifier, open up a new file and name it `train_model.py`. From there, we can get started:

```
1  # import the necessary packages
2  from sklearn.linear_model import LogisticRegression
3  from sklearn.model_selection import GridSearchCV
4  from sklearn.metrics import classification_report
5  from sklearn.metrics import accuracy_score
6  import argparse
7  import pickle
8  import h5py
```

Lines 2-8 import our required Python packages. We'll then parse our command line arguments:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-d", "--db", required=True,
13      help="path HDF5 database")
14 ap.add_argument("-m", "--model", required=True,
15      help="path to output model")
16 ap.add_argument("-j", "--jobs", type=int, default=-1,
17      help="# of jobs to run when tuning hyperparameters")
18 args = vars(ap.parse_args())
```

We only need two switches here, the path to the input HDF5 `--db`, along with the path to the output Logistic Regression `--model` after training is complete.

Next, let's open the HDF5 dataset for reading and determine the training and testing split – 75% of the data for training and 25% for testing:

```
20 # open the HDF5 database for reading then determine the index of
21 # the training and testing split, provided that this data was
22 # already shuffled *prior* to writing it to disk
23 db = h5py.File(args["db"], "r")
24 i = int(db["labels"].shape[0] * 0.75)
```

Given this feature split, we'll perform a grid search over the `C` hyperparameter of the `LogisticRegression` classifier:

```
# define the set of parameters that we want to tune then start a
# grid search where we evaluate our model for each value of C
print("[INFO] tuning hyperparameters...")
params = {"C": [0.0001, 0.001, 0.01, 0.1, 1.0]}
model = GridSearchCV(LogisticRegression(solver="lbfgs",
     multi_class="auto"), params, cv=3, n_jobs=args["jobs"])
model.fit(db["features"][:i], db["labels"][:i])
print("[INFO] best hyperparameters: {}".format(model.best_params_))
```

Once we've found the best choice of `C`, we can generate a classification report for the testing set:

```
# generate a classification report for the model
print("[INFO] evaluating...")
preds = model.predict(db["features"][i:])
print(classification_report(db["labels"][i:], preds,
     target_names=db["label_names"]))

# compute the raw accuracy with extra precision
acc = accuracy_score(db["labels"][i:], preds)
print("[INFO] score: {}".format(acc))
```

And finally, the trained model can be serialized to disk for later use, if we so wish:

```
 # serialize the model to disk
print("[INFO] saving model...")
f = open(args["model"], "wb")
f.write(pickle.dumps(model.best_estimator_))
f.close()

# close the database
db.close()
```

To train our model on the ResNet50 features, simply execute the following command:

```
$ python train_model.py \
     --db ../datasets/kaggle_dogs_vs_cats/hdf5/features.hdf5 \
     --model dogs_vs_cats.pickle
[INFO] tuning hyperparameters...
[INFO] best hyperparameters: {'C': 0.001}
[INFO] evaluating...
              precision    recall  f1-score   support

         cat       0.99      0.98      0.99      3160
         dog       0.98      0.99      0.99      3090

avg / total       0.99      0.99      0.99      6250

[INFO] score: 0.98688
[INFO] saving model...
```

As you can see from the output, our approach of using transfer learning via feature extraction yields an impressive accuracy of $\mathbf{98.69}\%$, enough for us to claim the #2 spot on the Kaggle Dogs vs. Cats leaderboard.

10.7 Summary

In this chapter we took a deep dive into the Kaggle Dogs vs. Cats dataset and studied two methods to obtain $> 90\%$ classification accuracy on it:

1. Training AlexNet from scratch.
2. Applying transfer learning via ResNet.

The AlexNet architecture is a seminal work first introduced by Krizhevsky et al. in 2012 [6]. Using our implementation of AlexNet, we reached 94 percent classification accuracy. This is a very respectable accuracy, especially for a network trained from scratch. Further accuracy can likely be obtained by:

1. Obtaining *more* training data.
2. Applying more aggressive data augmentation.
3. Deepening the network.

However, the 94 percent we obtained is not even enough for us to break our way into the top-25 leaderboard, let alone the top-5. Thus, to obtain our top-5 placement, we relied on transfer learning via feature extraction, specifically, the ResNet50 architecture trained on the ImageNet dataset. Since ImageNet contains *many* examples of both dog and cat breeds, applying a pre-trained network to this task is a natural, easy method to ensure we obtain higher accuracy with less effort. As our results demonstrated, we were able to obtain $\mathbf{98.69}\%$ classification accuracy, high enough to claim the *second position* on the Kaggle Dogs vs. Cats leaderboard.

11. GoogLeNet

In this chapter, we will study the GoogLeNet architecture, introduced by Szegedy et al. in their 2014 paper, *Going Deeper With Convolutions* [17]. This paper is important for two reasons. First, the model architecture is tiny compared to AlexNet and VGGNet ($\approx 28MB$ for the weights themselves). The authors are able to obtain such a dramatic drop in network architecture size (while still increasing the depth of the overall network) by removing fully-connected layers and instead using global average pooling. Most of the weights in a CNN can be found in the dense `FC` layers – if these layers can be removed, the memory savings are *massive*.

Secondly, the Szegedy et al. paper makes usage of a *network in network* or *micro-architecture* when constructing the overall *macro-architecture*. Up to this point, we have seen only *sequential* neural networks where the output of one network feeds directly into the next. We are now going to see *micro-architectures*, small building blocks that are used inside the rest of the architecture, where the output from one layer can split into a number of various paths and be rejoined later.

Specifically, Szegedy et al. contributed the *Inception module* to the deep learning community, a building block that fits into a Convolutional Neural Network enabling it to learn `CONV` layers with *multiple filter sizes*, turning the module into a multi-level feature extractor.

Micro-architectures such as Inception have inspired other important variants including the Residual module in ResNet [24] and the Fire module in SqueezeNet [32]. We'll be discussing the Inception module (and its variants) later in this chapter. Once we've examined the Inception module and ensure we know how it works, we'll then implement a smaller version of GoogLeNet called "MiniGoogLeNet" – we'll train this architecture on the CIFAR-10 dataset and obtain higher accuracy than in any of our previous chapters.

From there, we'll move on to the more difficult cs231n Tiny ImageNet Challenge [4]. This challenge is offered to students enrolled in Stanford's cs231n *Convolutional Neural Networks for Visual Recognition* class [39] as part of their final project. It means to give them a taste of the challenges associated with large scale deep learning on modern architectures, without being as time-consuming or taxing to work with as the *entire* ImageNet dataset.

By training GoogLeNet from scratch on Tiny ImageNet, we'll demonstrate how to obtain a top ranking position on the Tiny ImageNet leaderboard. And in our next chapter, we'll utilize ResNet

to claim the top position from models trained from scratch.

Let's go ahead and get this chapter started by discussing the Inception module.

11.1 The Inception Module (and its Variants)

Modern state-of-the-art Convolutional Neural Networks utilize *micro-architectures*, also called *network-in-network* modules, originally proposed by Lin et al. [40]. I personally prefer the term *micro-architecture* as it better describes these modules as *building blocks* in context of the overall *macro-architecture* (i.e., what you actually build and train).

Micro-architectures are small building blocks designed by deep learning practitioners to enable networks to learn (1) faster and (2) more efficiently, all while increasing network depth. These micro-architecture building blocks are stacked, along with conventional layer types such as `CONV`, `POOL`, etc., to form the overall macro-architecture.

In 2014, Szegedy et al. introduced the *Inception module*. The general idea behind the Inception module is two-fold:

1. It can be hard to decide the size of the filter you need to learn at a given `CONV` layers. Should they be 5×5 filters? What about 3×3 filters? Should we learn local features using 1×1 filters? Instead, why not learn them *all* and let the model decide? Inside the Inception module, we learn all three 5×5, 3×3, and 1×1 filters (computing them in parallel) concatenating the resulting feature maps along the channel dimension. The next layer in the GoogLeNet architecture (which could be another Inception module) receives these concatenated, mixed filters and performs the same process. Taken as a whole, this process enables GoogLeNet to learn both local features via smaller convolutions and abstracted features with larger convolutions – we don't have to sacrifice our level of abstraction at the expense of smaller features.
2. By learning multiple filter sizes, we can turn the module into a multi-level feature extractor. The 5×5 filters have a larger receptive size and can learn more abstract features. The 1×1 filters are by definition local. The 3×3 filters sit as a balance in between.

11.1.1 Inception

Now that we've discussed the motivation behind the Inception module, let's look at the actual module itself in Figure 11.1.

An activation function (ReLU) is *implicitly applied* after every `CONV` layer. To save space, this activation function was not included in the network diagram (Figure 11.1). When we implement GoogLeNet, you will see how this activation is used in the Inception module.

Specifically take note of how the Inception module *branches* into four distinct paths from the input layer. The *first* branch in the Inception module simply learns a series of 1×1 local features from the input.

The *second* batch first applies 1×1 convolution, not only as a form of learning local features, but instead as *dimensionality reduction*. Larger convolutions (i.e., 3×3 and 5×5) by definition take more computation to perform. Therefore, if we can reduce the dimensionality of the inputs to these larger filters by applying 1×1 convolutions, we can reduce the amount of computation required by our network. Therefore, the number of filters learned in the 1×1 `CONV` in the second branch will always be smaller than the number of 3×3 filters learned directly afterward.

The *third* branch applies the same logic as the second branch, only this time with the goal of learning 5×5 filters. We once again reduce dimensionality via 1×1 convolutions, then feed the output into the 5×5 filters.

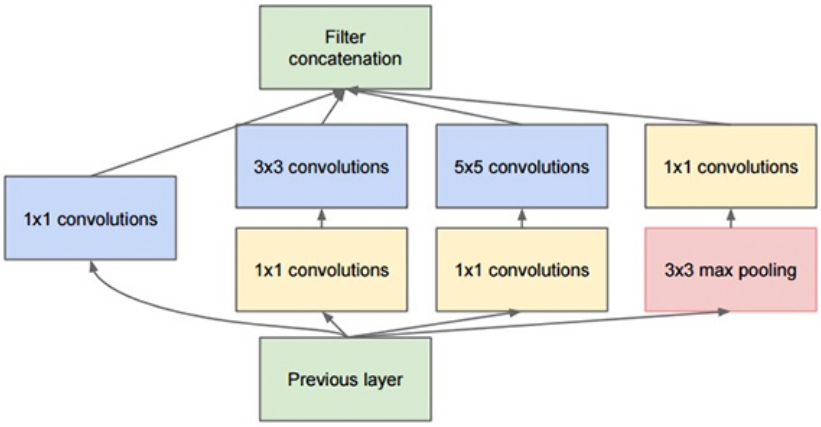


Figure 11.1: The original Inception module used in GoogLeNet. The Inception module acts as a "multi-level feature extractor" by computing 1×1, 3×3, and 5×5 convolutions within the *same* module of the network. Figure from Szegedy et al., 2014 [17].

The *fourth* and final branch of the Inception module performs 3×3 max pooling with a stride of 1×1 – this branch is commonly referred to as the *pool projection* branch. Historically, models that perform pooling have demonstrated an ability to obtain higher accuracy, although we now know through the work of Springenberg et al. in their 2014 paper, *Striving for Simplicity: The All Convolutional Net* [41] that this isn't necessarily true, and that `POOL` layers can be replaced with `CONV` layers for reducing volume size.

In the case of Szegedy et al., this `POOL` layer was added simply due to the fact that it was thought that they were needed for CNNs to perform reasonably. The output of the `POOL` is then fed into another series of 1×1 convolutions to learn local features.

Finally, all four branches of the Inception module converge where they are concatenated together along the channel dimension. Special care is taken during the implementation (via zero padding) to ensure the output of each branch has the same volume size, thereby allowing the outputs to be concatenated. The output of the Inception module is then fed into the next layer in the network. In practice, we often stack multiple Inception modules on top of each other before performing a pooling operation to reduce volume size.

11.1.2 Miniception

Of course, the original Inception module was designed for GoogLeNet such that it could be trained on the ImageNet dataset (where each input image is assumed to be $224 \times 224 \times 3$) and obtain state-of-the-art accuracy. For smaller datasets (with smaller image spatial dimensions) where fewer network parameters are required, we can simplify the Inception module.

I first became aware of the "Miniception" module from a tweet by @ericjang11 and @pluskid (https://twitter.com/ericjang11 and https://twitter.com/pluskid, respectively) where they beautifully visualize a smaller variant of Inception used when training the CIFAR-10 dataset (Figure 11.2; credit to @ericjang11 and @pluskid).

After doing a bit of research, it turns out that this graphic was from Zhang et al.'s 2017 publication, *Understanding Deep Learning Requires Re-Thinking Generalization* [42]. The top row of the figure describes three modules used in their MiniGoogLeNet implementation:

- **Left:** A convolution module responsible for performing convolution, batch normalization, and activation.

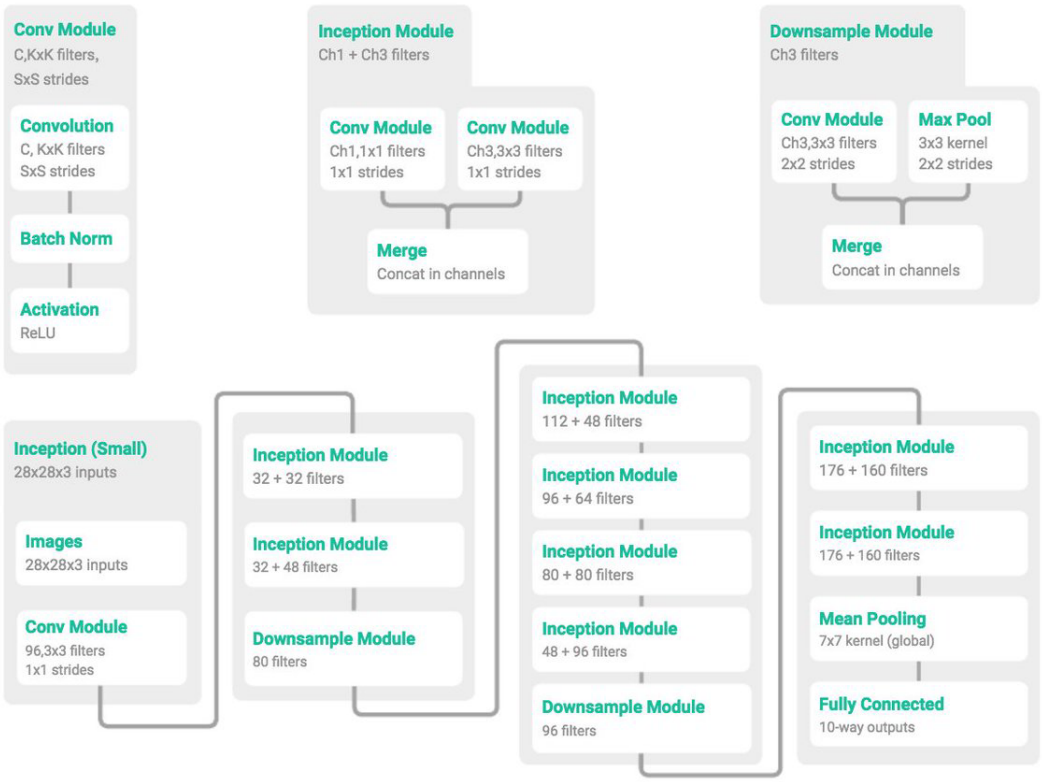


Figure 11.2: The MiniGoogLenet architecture consists of building blocks including a convolution module, Inception module, and Downsample module. These modules are put together to form the overall architecture.

- **Middle:** The Miniception module which performs two sets of convolutions, one for 1×1 filters and the other for 3×3 filters, then concatenates the results. No dimensionality reduction is performed before the 3×3 filter as (1) the input volumes will be smaller already (since we'll be using the CIFAR-10 dataset) and (2) to reduce the number of parameters in the network.
- **Right:** A downsample module which applies both a convolution and max pooling to reduce dimensionality, then concatenates across the filter dimension.

These building blocks are then used to build the MiniGoogLeNet architecture on the bottom row. You'll notice here that the authors placed the batch normalization *before* the activation (presumably because this is what Szegedy et al. did as well), in contrast to what is now recommended when implementing CNNs.

In this book I have stuck with the implementation of the *original author's work*, placing the batch normalization *before* activation in order to replicate results. In your own experiments, consider swapping this order.

In our next section, we'll implement the MiniGoogLeNet architecture and apply it to the CIFAR-10 dataset. From there, we'll be ready to implement the full Inception module and tackle the cs231n Tiny ImageNet challenge.

11.2 MiniGoogLeNet on CIFAR-10

In this section, we are going to implement the MiniGoogLeNet architecture using the Miniception module. We'll then train MiniGoogLeNet on the CIFAR-10 dataset. As our results will demonstrate,

this architecture will obtain $> 90\%$ accuracy on CIFAR-10, far better than all of our previous attempts.

11.2.1 Implementing MiniGoogLeNet

To get started, let's first create a file named `minigooglenet.py` inside the `conv` module of `pyimagesearch.nn` – this is where our implementation of the `MiniGoogLeNet` class will live:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- alexnet.py
|    |    |    |--- lenet.py
|    |    |    |--- minigooglenet.py
|    |    |    |--- minivggnet.py
|    |    |    |--- fcheadnet.py
|    |    |    |--- shallownet.py
|    |--- preprocessing
|    |--- utils
```

From there, open up `minigooglenet.py` and insert the following code:

```
1  # import the necessary packages
2  from keras.layers.normalization import BatchNormalization
3  from keras.layers.convolutional import Conv2D
4  from keras.layers.convolutional import AveragePooling2D
5  from keras.layers.convolutional import MaxPooling2D
6  from keras.layers.core import Activation
7  from keras.layers.core import Dropout
8  from keras.layers.core import Dense
9  from keras.layers import Flatten
10 from keras.layers import Input
11 from keras.models import Model
12 from keras.layers import concatenate
13 from keras import backend as K
```

Lines 2-13 import our required Python packages. Rather than importing the `Sequential` class where the output of one layer feeds directly into the next, we'll instead need to use the `Model` class (**Line 11**). Using `Model` rather than `Sequential` will allow us to create a network graph with splits and forks like in the Inception module. Another import you have not yet seen is the `concatenate` function on **Line 12**. As the name suggests, this function takes a set of inputs and concatenates them along a given axis, which in this case will be the channel dimension.

We'll be implementing the exact version of MiniGoogLeNet as detailed in Figure 11.2 above, so let's start off with the `conv_module`:

```
15 class MiniGoogLeNet:
16     @staticmethod
```

```
17      def conv_module(x, K, kX, kY, stride, chanDim, padding="same"):
18          # define a CONV => BN => RELU pattern
19          x = Conv2D(K, (kX, kY), strides=stride, padding=padding)(x)
20          x = BatchNormalization(axis=chanDim)(x)
21          x = Activation("relu")(x)
22
23          # return the block
24          return x
```

The `conv_module` function is responsible for applying a convolution, followed by a batch normalization, and then finally an activation. The parameters to the method are detailed below:

- `x`: The input layer to the function.
- `K`: The number of filters our `CONV` layer is going to learn.
- `kX` and `kY`: The size of each of the `K` filters that will be learned.
- `stride`: The stride of the `CONV` layer.
- `chanDim`: The channel dimension, which is derived from either "channels last" or "channels first" ordering.
- `padding`: The type of padding to be applied to the `CONV` layer.

On **Line 19** we create the convolutional layer. The actual parameters to `Conv2D` are identical to examples in previous architectures such as AlexNet and VGGNet, but what changes here is how we supply the *input* to a given layer.

Since we are using a `Model` rather than a `Sequential` to define the network architecture, we cannot call `model.add` as this would imply that the output from one layer follows *sequentially* into the next layer. Instead, we supply the *input layer* in parenthesis at the end of the function call, which is called a *Functional API*. Each layer instance in a `Model` is callable on a tensor and also returns a tensor. Therefore, we can supply the inputs to a given layer by calling it as a function once the object is instantiated.

A template for constructing layers in this manner can be seen below:

```
output = Layer(parameters)(input)
```

Take a second to familiarize yourself with this new style of adding layers to a network as we'll be using it whenever we define networks that are non-sequential.

The output of the `Conv2D` layer is then passed into the `BatchNormalization` layer on **Line 20**. The output of `BatchNormalization` then goes through a ReLU activation (**Line 21**). If we were to construct a figure to help us visualize the `conv_module` it would look like Figure 11.3.

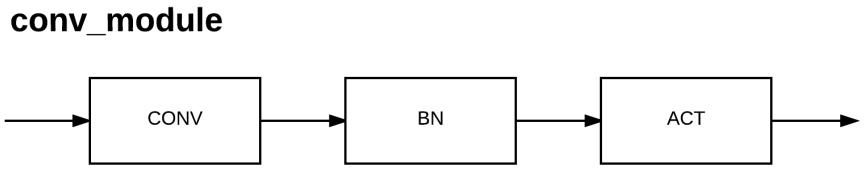


Figure 11.3: The `conv_module` of the MiniGoogLeNet architecture. This module includes no branching and is a simple `CONV => BN => ACT`.

First the convolution is applied, then a batch normalization, followed by an activation. Note that this module did not perform any branching. That is going to change with the definition of the `inception_module` below:

```
26      @staticmethod
27      def inception_module(x, numK1x1, numK3x3, chanDim):
28          # define two CONV modules, then concatenate across the
29          # channel dimension
30          conv_1x1 = MiniGoogLeNet.conv_module(x, numK1x1, 1, 1,
31              (1, 1), chanDim)
32          conv_3x3 = MiniGoogLeNet.conv_module(x, numK3x3, 3, 3,
33              (1, 1), chanDim)
34          x = concatenate([conv_1x1, conv_3x3], axis=chanDim)
35
36          # return the block
37          return x
```

Our Mininception module will perform two sets of convolutions – a 1×1 `CONV` and a 3×3 `CONV`. These two convolutions will be performed *in parallel* and the resulting features concatenated across the channel dimension.

Lines 30 and 31 use the handy `conv_module` we just defined to learn `numK1x1` filters (1×1). **Lines 32 and 33** then apply `conv_module` again to learn `numK3x3` filters (3×3). By using the `conv_module` function we are able to *reuse* code and not have to bloat our `MiniGoogLeNet` class by inserting many blocks of `CONV => BN => RELU` blocks – this stacking is taken care of concisely via `conv_module`.

Notice how both the input to the 1×1 and 3×3 `Conv2D` class is `x`, the input to the layer. When using the `Sequential` class, this type of layer structure was not possible. But by using the `Model` class, we can now have multiple layers accept the same input. Once we have both `conv_1x1` and `conv_3x3`, we concatenate them across the channel dimension.

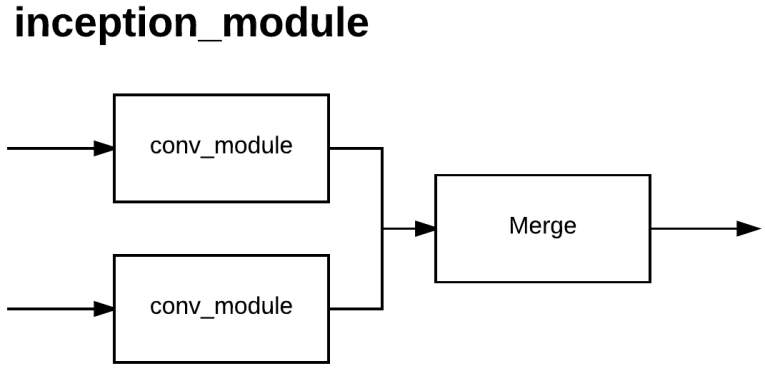


Figure 11.4: The (mini)-`inception_module` consists of two branches. The first branch is a `CONV` layer responsible for learning 1×1 filters. The second branch is another `CONV` layer that learns 3×3 filters. The filter responses are then concatenated along the channel dimension.

To visualize the "Mini"-Inception module, take a look at Figure 11.4. Our 1×1 and 3×3 `CONV` layers take a given input and apply their respective convolutions. The output of *both* convolutions is then concatenated (**Line 34**). We are allowed to concatenate the layer outputs because the output volume size for both convolutions is *identical* due to `padding="same"`.

Next comes the `downsample_module`, which as the name suggests, is responsible for reducing the spatial dimensions of an input volume:

```
39      @staticmethod
40      def downsample_module(x, K, chanDim):
```

```
41          # define the CONV module and POOL, then concatenate
42          # across the channel dimensions
43          conv_3x3 = MiniGoogLeNet.conv_module(x, K, 3, 3, (2, 2),
44               chanDim, padding="valid")
45          pool = MaxPooling2D((3, 3), strides=(2, 2))(x)
46          x = concatenate([conv_3x3, pool], axis=chanDim)
47
48          # return the block
49          return x
```

This method requires us to pass in an input `x`, the number of filters `K` our convolutional layer will learn, along with the `chanDim` for batch normalization and channel concatenation.

The first branch of the `downsample_module` learns a set of `K`, 3×3 filters using a stride of 2×2, thereby decreasing the output volume size (**Lines 43 and 44**). We apply max pooling on **Line 45** (the second branch), again with window size of 3×3 and stride of 2×2 to reduce volume size. The `conv_3x3` and `pool` outputs are then concatenated (**Line 46**) and returned to the calling function.

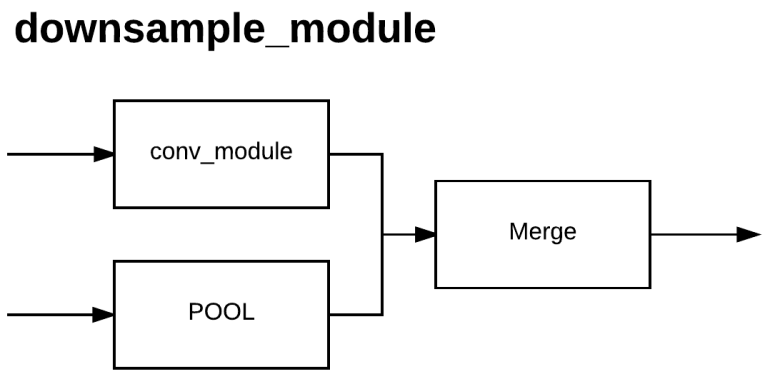


Figure 11.5: The `downsample_module` is responsible for reducing the spatial dimensions of our input volume. The first branch learns a set of filters with 2×2 stride to reduce the output volume. The second branch also reduces the spatial dimensions, this time by applying max pooling. The output of the `downsample_module` is concatenated along the channel dimension.

We can visualize the `downsample_module` in Figure 11.5. As the figure demonstrates, a convolution and max pooling operation are applied to the *same* input and then concatenated.

We are now ready to put all the pieces together:

```
51      @staticmethod
52      def build(width, height, depth, classes):
53          # initialize the input shape to be "channels last" and the
54          # channels dimension itself
55          inputShape = (height, width, depth)
56          chanDim = -1
57
58          # if we are using "channels first", update the input shape
59          # and channels dimension
60          if K.image_data_format() == "channels_first":
61              inputShape = (depth, height, width)
62              chanDim = 1
```

Line 52 defines the `build` method to our network, as is standard for all other examples in this book. Our `build` method accepts an input `width`, `height`, `depth`, and total number of `classes` that will be learned. **Lines 55 and 56** initialize our `inputShape` and `chanDim` assuming we are using "channels last" ordering. If we are instead using "channels first" ordering, **Lines 60-62** update these variables, respectively.

Let's define the model `Input` along with the first `conv_module`:

```
64          # define the model input and first CONV module
65          inputs = Input(shape=inputShape)
66          x = MiniGoogLeNet.conv_module(inputs, 96, 3, 3, (1, 1),
67              chanDim)
```

The call to `Input` on **Line 65** initializes the architecture – all inputs to the network will start at this layer which simply "holds" the input data (all networks need to have an input, after all). The first `CONV => BN => RELU` is applied on **Lines 66 and 67** where we learn 96, 3×3 filters.

From there, we stack two Inception modules followed by a downsample module:

```
69          # two Inception modules followed by a downsample module
70          x = MiniGoogLeNet.inception_module(x, 32, 32, chanDim)
71          x = MiniGoogLeNet.inception_module(x, 32, 48, chanDim)
72          x = MiniGoogLeNet.downsample_module(x, 80, chanDim)
```

The first Inception module (**Line 70**) learns 32 filters for both the 1×1 and 3×3 `CONV` layers. When concatenated, this module outputs a volume with $K = 32 + 32 = 64$ filters.

The second Inception module (**Line 71**) learns 32, 1×1 filters and 48, 3×3 filters. Again, when concatenated, we see that the output volume size is $K = 32 + 48 = 80$. The downsample module reduces our input volume sizes but keeps the same number of filters learned at 80.

Next, let's stack four Inception modules on top of each other before applying a downsample, allowing MiniGoogLeNet to learn deeper, richer features:

```
74          # four Inception modules followed by a downsample module
75          x = MiniGoogLeNet.inception_module(x, 112, 48, chanDim)
76          x = MiniGoogLeNet.inception_module(x, 96, 64, chanDim)
77          x = MiniGoogLeNet.inception_module(x, 80, 80, chanDim)
78          x = MiniGoogLeNet.inception_module(x, 48, 96, chanDim)
79          x = MiniGoogLeNet.downsample_module(x, 96, chanDim)
```

Notice how in some layers we learn more 1×1 filters than 3×3 filters, while other Inception modules learn more 3×3 filters than 1×1. This type of alternating pattern is done on purpose and was justified by Szegedy et al. after running many experiments. When we implement the deeper variant of GoogLeNet later in this chapter, we'll also see this pattern as well.

Continuing our implementation of Figure 11.2 by Zhang et al., we'll now apply two more inception modules followed by a global pool and dropout:

```
81          # two Inception modules followed by global POOL and dropout
82          x = MiniGoogLeNet.inception_module(x, 176, 160, chanDim)
83          x = MiniGoogLeNet.inception_module(x, 176, 160, chanDim)
84          x = AveragePooling2D((7, 7))(x)
85          x = Dropout(0.5)(x)
```

The output volume size after **Line 83** is $7 \times 7 \times 336$. Applying an average pooling of 7×7 reduces the volume size to $1 \times 1 \times 336$ and thereby alleviates the need to apply many dense fully-connected layers – instead, we simply average over the spatial outputs of the convolution. Dropout is applied with a probability of 50 percent on **Line 85** to help reduce overfitting.

Finally, we add in our softmax classifier based on the number of `classes` we wish to learn:

```
87          # softmax classifier
88          x = Flatten()(x)
89          x = Dense(classes)(x)
90          x = Activation("softmax")(x)
91
92          # create the model
93          model = Model(inputs, x, name="googlenet")
94
95          # return the constructed network architecture
96          return model
```

The actual `Model` is then instantiated on **Line 93** where we pass in the `inputs`, the layers (`x`, which includes the built-in branching), and optionally a name for the network. The constructed architecture is returned to the calling function on **Line 96**.

11.2.2 Training and Evaluating MiniGoogLeNet on CIFAR-10

Now that MiniGoogLeNet is implemented, let's train it on the CIFAR-10 dataset and see if we can beat our previous best of 84 percent. Open up a new file, name it `googlenet_cifar10.py`, and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from sklearn.preprocessing import LabelBinarizer
7  from pyimagesearch.nn.conv import MiniGoogLeNet
8  from pyimagesearch.callbacks import TrainingMonitor
9  from keras.preprocessing.image import ImageDataGenerator
10 from keras.callbacks import LearningRateScheduler
11 from keras.optimizers import SGD
12 from keras.datasets import cifar10
13 import numpy as np
14 import argparse
15 import os
```

Lines 2 and 3 configure `matplotlib` so we can save figures and plots to disk in the background. We then import the rest of our required packages on **Lines 6-15**. **Line 7** imports our implementation of `MiniGoogLeNet`.

Also notice how we are importing the `LearningRateScheduler` class on **Line 10**, which implies that we'll be defining a *specific* learning rate for our optimizer to follow when training the network. Specifically, we'll be defining a *polynomial decay* learning rate schedule. A polynomial learning rate scheduler will follow the equation:

$$\alpha = \alpha_0 * (1 - e/e_{max})^p \tag{11.1}$$

Where α_0 is the initial learning rate, e is the current epoch number, e_{max} is the *maximum* number of epochs we are going to perform, and p is the power of the polynomial. Applying this equation yields the learning rate α for the current epoch.

Given the maximum number of epochs, the learning rate will decay to zero. This learning rate scheduler can also be made *linear* by setting the power to 1.0 – which is often done – and, in fact, what we are going to do in this example. I have included a number of example polynomial learning rate schedules using a maximum of 70 epochs, an initial learning rate of $5e-3$, and varying powers in Figure 11.6. Notice how as the power increases, the faster the learning rate drops. Using a power of 1.0 turns the curve into a linear decay.

Figure 11.6: Plots of polynomial learning rate decays for varying values of the power, p. Notice how as the power increases the sharper the decay. Setting $p = 1.0$ turns a polynomial decay into a linear decay.

Let's go ahead and implement this learning rate schedule function below:

```
17 # define the total number of epochs to train for along with the
18 # initial learning rate
19 NUM_EPOCHS = 70
20 INIT_LR = 5e-3
21
22 def poly_decay(epoch):
23     # initialize the maximum number of epochs, base learning rate,
24     # and power of the polynomial
25     maxEpochs = NUM_EPOCHS
26     baseLR = INIT_LR
27     power = 1.0
28
29     # compute the new learning rate based on polynomial decay
```

```
30      alpha = baseLR * (1 - (epoch / float(maxEpochs))) ** power
31
32      # return the new learning rate
33      return alpha
```

Per our discussion of learning rate schedulers in Chapter 16 of the *Starter Bundle*, you know that a learning rate scheduling function can only accept a *single* argument, the current `epoch`. We then initialize the `maxEpochs` the network is allowed to train for (so we can decay the learning rate to zero), the base learning rate, as well as the power of the polynomial.

Computing the new learning rate based on the polynomial decay is handled on **Line 30** – the output of this equation will match our graphs exactly based on the parameters supplied. The new learning rate is returned to the calling function on **Line 33** so that the optimizer can update its internal learning rate. Again, for more information on learning rate schedulers, please see Chapter 16 of the *Starter Bundle*.

Now that we have defined our learning rate, we can parse our command line arguments:

```
35  # construct the argument parse and parse the arguments
36  ap = argparse.ArgumentParser()
37  ap.add_argument("-m", "--model", required=True,
38      help="path to output model")
39  ap.add_argument("-o", "--output", required=True,
40      help="path to output directory (logs, plots, etc.)")
41  args = vars(ap.parse_args())
```

Our script requires two arguments, `--model`, the path to the output file where `MiniGoogLeNet` will be serialized after training, along with `--output`, where we will store any plots, logs, etc.

The next step is to load the CIFAR-10 data from disk, perform pixel-wise mean subtraction, and then one-hot encode the labels:

```
43  # load the training and testing data, converting the images from
44  # integers to floats
45  print("[INFO] loading CIFAR-10 data...")
46  ((trainX, trainY), (testX, testY)) = cifar10.load_data()
47  trainX = trainX.astype("float")
48  testX = testX.astype("float")
49
50  # apply mean subtraction to the data
51  mean = np.mean(trainX, axis=0)
52  trainX -= mean
53  testX -= mean
54
55  # convert the labels from integers to vectors
56  lb = LabelBinarizer()
57  trainY = lb.fit_transform(trainY)
58  testY = lb.transform(testY)
```

To help combat overfitting and enable our model to obtain higher classification accuracy, we'll apply data augmentation:

```
60  # construct the image generator for data augmentation
61  aug = ImageDataGenerator(width_shift_range=0.1,
```

```
62      height_shift_range=0.1, horizontal_flip=True,
63      fill_mode="nearest")
```

We'll also construct a set of `callbacks` to monitor training progress as well as call our `LearningRateScheduler`:

```
65  # construct the set of callbacks
66  figPath = os.path.sep.join([args["output"], "{}.png".format(
67      os.getpid())])
68  jsonPath = os.path.sep.join([args["output"], "{}.json".format(
69      os.getpid())])
70  callbacks = [TrainingMonitor(figPath, jsonPath=jsonPath),
71      LearningRateScheduler(poly_decay)]
```

Finally, we are ready to train our network:

```
73  # initialize the optimizer and model
74  print("[INFO] compiling model...")
75  opt = SGD(lr=INIT_LR, momentum=0.9)
76  model = MiniGoogLeNet.build(width=32, height=32, depth=3, classes=10)
77  model.compile(loss="categorical_crossentropy", optimizer=opt,
78      metrics=["accuracy"])
79
80  # train the network
81  print("[INFO] training network...")
82  model.fit_generator(aug.flow(trainX, trainY, batch_size=64),
83      validation_data=(testX, testY), steps_per_epoch=len(trainX) // 64,
84      epochs=NUM_EPOCHS, callbacks=callbacks, verbose=1)
85
86  # save the network to disk
87  print("[INFO] serializing network...")
88  model.save(args["model"])
```

Line 75 initializes the SGD optimizer with an `INIT_LR` initial learning rate. This learning rate will be updated via the `LearningRateScheduler` once training starts. The `MiniGoogLeNet` architecture itself will accept input images with a width of 32 pixels, height of 32 pixels, depth of 3 channels, and a total of 10 class labels. **Lines 82-84** kick off the training process using mini-batch sizes of 64, training for a total of `NUM_EPOCHS`. Once training is complete, **Line 88** serializes our model to disk.

11.2.3 MiniGoogLeNet: Experiment #1

At this point in the *Practitioner Bundle*, it becomes important for you to understand the actual mindset, process, and set of experiments you'll need to perform to obtain a high accuracy model on a given dataset.

In previous chapters in both this bundle and the *Starter Bundle*, you were just getting your feet wet – and it was enough to simply see a piece of code, understand what it does, execute it, and look at the output. That was a great starting point to developing an understanding of deep learning.

However, now that we are working with more advanced architectures and challenging problems, you need to understand the *process* behind performing experiments, examining the results, and then updating the parameters. I provide a gentle introduction to this scientific method in this chapter

and the following chapter on ResNet. If you are interested in *mastering* this ability to perform experiments, examine the results, and make intelligent postulations as to what the next best course of action is, then please refer to the more advanced lessons in the *ImageNet Bundle*.

In my first experiment with GoogLeNet, I started with an initial learning rate of $1e-3$ with the SGD optimizer. The learning rate was then decayed linearly over the course of 70 epochs. Why 70 epochs? Two reasons:

1. ***A priori* knowledge.** After reading hundreds of deep learning papers, blog posts, tutorials and not to mention, *performing your own experiments*, you'll start to notice a pattern for some datasets. In my case, I knew from previous experiments in my career with the CIFAR-10 dataset that anywhere between 50-100 epochs is *typically* all that is required to train CIFAR-10. The deeper the network architecture is (with sufficient regularization), along with a decreasing learning rate, will normally allow us to train our network for longer. Therefore, I choose 70 epochs for my first experiment. After the experiment was finished running, I could examine the learning plot and decide if more/less epochs should be used (as it turns out, 70 epochs was spot on).
2. **Inevitable overfitting.** Secondly, we *know* from previous experiments in this book that we will eventually overfit when working with CIFAR-10. It's inevitable; even with strong regularization and data augmentation, it's still going to happen. Therefore, I decided on 70 epochs rather than risking 80-100 epochs where the effects of overfitting would become more pronounced.

I then started training using the following command:

```
$ python googlenet_cifar10.py --output output \
    --model output/minigooglenet_cifar10.hdf5
```

Using our initial learning rate of $1e-3$ with a linear decay over 70 epochs, we obtained $\mathbf{87.95}\%$ classification accuracy (Figure 11.7, *top-left*). Furthermore, looking at our plot, while there is a gap between the training and validation loss, the gap stays (relatively) proportional past epoch 40. The same can be said of the training and validation accuracy curves as well. Looking at this graph, I became concerned that we did not train hard enough and that we might be able to obtain higher classification accuracy with a $1e-2$ learning rate.

11.2.4 MiniGoogLeNet: Experiment #2

In our second MiniGoogLeNet experiment, I swapped out the initial SGD learning rate of $1e-3$ for a larger $1e-2$. This learning rate was linearly decayed over the course of 70 epochs. I once again trained the network and gathered the results:

```
$ python googlenet_cifar10.py --output output \
    --model output/minigooglenet_cifar10.hdf5
```

At the end of the 70th epoch, we are obtaining $\mathbf{91.79}\%$ accuracy on the validation set (Figure 11.7, *top-right*), certainly better than our previous experiment – *but we shouldn't get too excited yet*. Looking at the loss for the training curve, we can see that it falls *entirely* to zero past epoch 60. Furthermore, the classification accuracy for the training curve is fully saturated at 100%.

While we have increased our validation accuracy, we have done so at the expense of overfitting – the gap between validation loss and training loss is *huge* past epoch 20. Instead, we would do better to re-train our network with a $5e-3$ learning rate, falling square in the middle of $1e-2$ and $1e-3$. We might obtain slightly less validation accuracy, but we'll ideally be able to reduce the effects of overfitting.

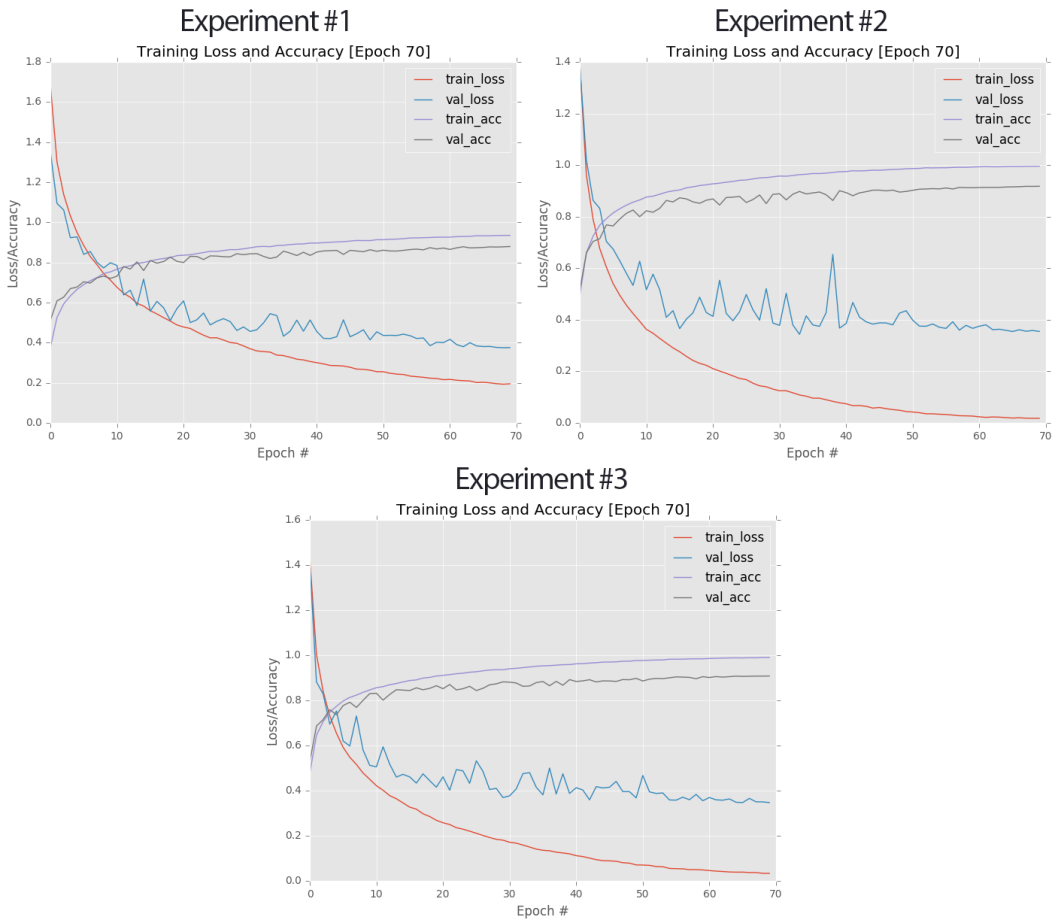


Figure 11.7: **Top-left:** Learning curves for Experiment #1. **Top-right:** Plots for Experiment #2. **Bottom:** Learning curves for Experiment #3. Our final experiment is a good balance between the first two, obtaining 90.81% accuracy, higher than all of our previous CIFAR-10 experiments.

For what it's worth, if you find that your network has totally saturated loss (0.0) and accuracy (100%) for the training set, be sure to pay close attention to your validation curves. If you see larges of gaps between the validation and training plot, you have surely overfit. Go back to your experiment and play with the parameters, introduce more regularization, and adjust the learning rate. Saturations such as those displayed in this experiment are indicative of a model that will *not* generalize well.

11.2.5 MiniGoogLeNet: Experiment #3

In this experiment, I adjusted my learning rate to $5e-3$. MiniGoogLeNet was trained for 70 epochs using the SGD optimizer and a linear learning rate decay:

```
$ python googlenet_cifar10.py --output output \
    --model output/minigooglenet_cifar10.hdf5
```

As the output shows, we obtained **90.81%** classification accuracy on the validation set (Figure 11.7, *bottom*) – lower than the previous experiment, but higher than the first experiment. We are definitely overfitting in this experiment (by approximately an order of magnitude), but we can accept this as an inevitability when working with CIFAR-10.

What's more important here is that we kept our training loss and accuracy saturation levels as low as possible – the training loss did not fall completely to zero, and the training accuracy did not reach 100%. We can also see a reasonable gap maintained between training and validation accuracy, even in the later epochs.

At this point I would consider this experiment to be an initial success (with the caveat that more experiments should be done to reduce overfitting) – we have successfully trained MiniGoogLeNet on CIFAR-10, reaching our goal of $> 90\%$ classification, beating out all previous experiments on CIFAR-10.

Future revisions of this experiment should consider being aggressive with regularization. Specifically, we did not use any type of weight regularization here. Applying L2 weight decay would help combat our overfitting (as our experiments in the next section will demonstrate).

Moving forward, use this experiment as a baseline. We know there is overfitting and we'd like to reduce it while increasing classification accuracy. In our next chapter on ResNet, we'll see how we can accomplish both.

Now that we've explored MiniGoogLeNet applied to CIFAR-10, let's move on to the more difficult classification task of the cs231n Tiny ImageNet challenge where we'll be implementing a deeper variant of GoogLeNet, similar to the architecture used by Szgedy et al. in their original paper.

11.3 The Tiny ImageNet Challenge

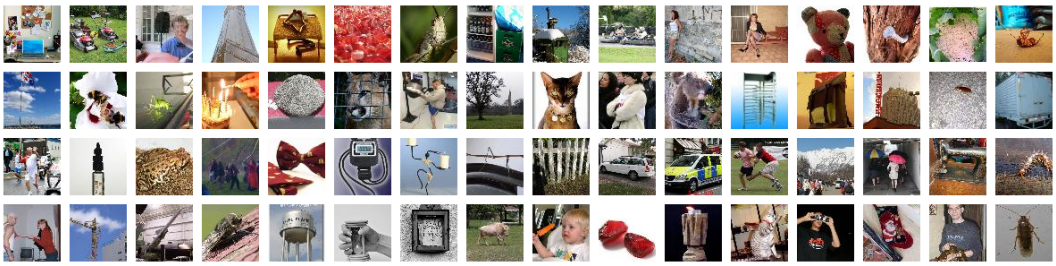


Figure 11.8: A sample of images from Stanford's Tiny ImageNet classification challenge.

The Tiny ImageNet Visual Recognition Challenge (a sample of which can be seen in Figure 11.8) is part of the cs231n Stanford course on *Convolutional Neural Networks for Visual Recognition* [39]. As part of their final project, students can compete in the classification by either training a CNN from scratch or performing transfer learning via fine-tuning (transfer learning via feature extraction is not allowed).

The Tiny ImageNet dataset is actually a *subset* of the full ImageNet dataset (hence why feature extraction cannot be used, as it would give the network an unfair advantage), consisting of 200 diverse classes, including everything from *Egyptian cats* to *volleyballs* to *lemons*. Given that there are 200 classes, guessing at random we would expect to be correct $1/200 = 0.5\%$ of the time; therefore, our CNN needs to obtain *at least* 0.5% to demonstrate it has learned discriminative underlying patterns in the respective classes.

Each class includes 500 training images, 50 validation images, and 50 testing images. Ground-truth labels are *only* provided for the training and validation images. Since we do not have access to the Tiny ImageNet evaluation server, we will use part of the training set to form our own testing set so we can evaluate the performance of our classification algorithms.

As readers of the *ImageNet Bundle* will discover (where we discuss how to train deep Convolutional Neural Networks on the full ImageNet dataset from scratch), the images in the ImageNet

Large Scale Visual Recognition Challenge (ILSVRC) have *varying* widths and heights. Therefore, whenever we work with ILSVRC, we first need to resize all images in the dataset to a *fixed* width and height before we can train our network. To help students focus strictly on the deep learning and image classification component (and not get caught up in image processing details), *all images* in the Tiny ImageNet dataset have been resized to 64×64 pixels and center cropped.

In some ways, having the images resized makes Tiny ImageNet a bit more challenging than it's bigger brother, ILSVRC. In ILSVRC we are free to apply any type of resizing, cropping, etc. operations that we see fit. However, with Tiny ImageNet, much of the image has already been discarded for us. As we'll find out, obtaining a reasonable rank-1 and rank-5 accuracy on Tiny ImageNet isn't as easy as one might think, making it a great, insightful dataset for budding deep learning practitioners to learn and practice on.

In the next few sections, you will learn how to obtain the Tiny ImageNet dataset, understand its structure, and create HDF5 files for the training, validation, and testing images.

11.3.1 Downloading Tiny ImageNet

You can download the Tiny ImageNet dataset from official cs231n leaderboard page here:

https://tiny-imagenet.herokuapp.com/

Alternatively, I have created a mirror for the file here as well:

http://pyimg.co/h28e4

The .zip file is $\approx 237MB$, so make sure you have a reasonable internet connection before attempting the download.

11.3.2 The Tiny ImageNet Directory Structure

After downloading and unpacking your `tiny-imagenet-200.zip` file, unarchive it, and you'll find the following directory structure:

```
--- tiny-imagenet-200
|     |--- test
|     |--- train
|     |--- val
|     |--- wnids.txt
|     |--- words.txt
```

Inside the `test` directory are the testing images – we'll be ignoring these images since we do not have access to the cs231n evaluation server (the labels are purposely left out from the download to ensure no one can "cheat" in the challenge).

We then have the `train` directory which contains subdirectories with strange names starting with the letter `n` followed by a series of numbers. These subdirectories are the WordNet [43] IDs called "synonym set" or "synsets" for short. Each WordNet ID maps to a specific word/object. Every image inside a given WordNet subdirectory contains examples of that object.

We can lookup the human readable label for a WordNet ID by parsing the `words.txt` file, which is simply a tab separated file with the WordNet ID in the first column and the human readable word/object in the second column. The `wnids.txt` file lists out the 200 WordNet IDs (one per line) in the ImageNet dataset.

Finally, the `val` directory stores our validation set. Inside the `val` directory, you'll find an `images` subdirectory and a file named `val_annotations.txt`. The `val_annotations.txt` provides the WordNet IDs for every image in the `val` directory.

Therefore, before we can even get started training GoogLeNet on Tiny ImageNet, we first need to write a script to parse these files and put them into HDF5 format. Keep in mind that being a deep

learning practitioner isn't about implementing Convolutional Neural Networks and training them from scratch. Part of being a deep learning practitioner involves using your programming skills to build simple scripts that can parse data.

The more general purpose programming skills you have, the better deep learning practitioner you can become – while other deep learning researchers are struggling to organize files on disk or understand how a dataset is structured, you'll have already converted your entire dataset to a format suitable for training a CNN.

In the next section, I'll teach you how to define your project configuration file and create a single, simple Python script that will convert the Tiny ImageNet dataset into an HDF5 representation.

11.3.3 Building the Tiny ImageNet Dataset

Let's go ahead and define the project structure for Tiny ImageNet + GoogLeNet:

```
--- deepergooglenet
|    |--- config
|    |    |--- __init__.py
|    |    |--- tiny_imagenet_config.py
|    |--- build_tiny_imagenet.py
|    |--- rank_accuracy.py
|    |--- train.py
|    |--- output/
|    |    |--- checkpoints/
|    |    |--- tiny-image-net-200-mean.json
```

We'll create a `config` module where we'll store any `tiny_imagenet_config.py` configurations. We then have the `build_tiny_imagenet.py` script which is responsible for taking Tiny ImageNet and converting it to HDF5. The `train.py` script will train GoogLeNet on the HDF5 version of Tiny ImageNet. Finally, we'll use `rank.py` to compute the rank-1 and rank-5 accuracies for the testing set.

Let's go ahead and take a look at `tiny_imagenet_config.py`:

```
1  # import the necessary packages
2  from os import path
3
4  # define the paths to the training and validation directories
5  TRAIN_IMAGES = "../datasets/tiny-imagenet-200/train"
6  VAL_IMAGES = "../datasets/tiny-imagenet-200/val/images"
7
8  # define the path to the file that maps validation filenames to
9  # their corresponding class labels
10 VAL_MAPPINGS = "../datasets/tiny-imagenet-200/val/val_annotations.txt"
```

Lines 5 and 6 define the paths to the Tiny ImageNet training and validation images, respectively. We then define the path to the validation file mappings which enables us to map the validation filenames to actual class labels (i.e., the WordNet IDs).

Speaking of WordNet IDs and human readable labels, let's define the paths to those as well:

```
12 # define the paths to the WordNet hierarchy files which are used
13 # to generate our class labels
14 WORDNET_IDS = "../datasets/tiny-imagenet-200/wnids.txt"
15 WORD_LABELS = "../datasets/tiny-imagenet-200/words.txt"
```

Given that we do not have access to the testing labels, we'll need to take a portion of the training data and use it for validation (since our training data *does* have labels associated with each image):

```
# since we do not have access to the testing data we need to
# take a number of images from the training data and use it instead
NUM_CLASSES = 200
NUM_TEST_IMAGES = 50 * NUM_CLASSES
```

The purpose of this section is to convert Tiny ImageNet to HDF5, therefore we need to supply paths to the training, validation, and testing HDF5 files:

```
# define the path to the output training, validation, and testing
# HDF5 files
TRAIN_HDF5 = "../datasets/tiny-imagenet-200/hdf5/train.hdf5"
VAL_HDF5 = "../datasets/tiny-imagenet-200/hdf5/val.hdf5"
TEST_HDF5 = "../datasets/tiny-imagenet-200/hdf5/test.hdf5"
```

When writing the images to disk, we'll want to compute the RGB means for the training set, enabling us to perform mean normalization – after we have the means, they will need to be serialized to disk as a JSON file:

```
# define the path to the dataset mean
DATASET_MEAN = "output/tiny-image-net-200-mean.json"
```

Finally, we'll define the paths to our output model and training logs/plots:

```
# define the path to the output directory used for storing plots,
# classification reports, etc.
OUTPUT_PATH = "output"
MODEL_PATH = path.sep.join([OUTPUT_PATH,
     "checkpoints/epoch_70.hdf5"])
FIG_PATH = path.sep.join([OUTPUT_PATH,
     "deepergooglenet_tinyimagenet.png"])
JSON_PATH = path.sep.join([OUTPUT_PATH,
     "deepergooglenet_tinyimagenet.json"])
```

As you can see, this configuration file is fairly straightforward. We are mainly just defining paths to input directories of images/label mappings along with output files. However, taking the time to create this configuration file makes our life much easier when actually *building* Tiny ImageNet and converting it to HDF5.

To see how this is true, let's go ahead and examine `build_tiny_imagenet.py`:

```
# import the necessary packages
from config import tiny_imagenet_config as config
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from pyimagesearch.io import HDF5DatasetWriter
from imutils import paths
import numpy as np
```

```
8  import progressbar
9  import json
10 import cv2
11 import os
```

Lines 2-11 import our required Python packages. On **Line 2** we import our newly coded configuration file so we can have access to the variables inside it. We'll be using the `LabelEncoder` to encode the WordNet IDs as integers. The `train_test_split` function will be applied to construct our training and testing split. We'll the use the `HDF5DatasetWriter` class to actually write the raw images to their respective HDF5 datasets.

Let's go ahead and grab the paths to the training images, extract the class labels, and encode them:

```
13 # grab the paths to the training images, then extract the training
14 # class labels and encode them
15 trainPaths = list(paths.list_images(config.TRAIN_IMAGES))
16 trainLabels = [p.split(os.path.sep)[-3] for p in trainPaths]
17 le = LabelEncoder()
18 trainLabels = le.fit_transform(trainLabels)
```

On **Line 15** we grab a list of all image paths inside the `TRAIN_IMAGES` directory. Every path in this list has the pattern:

```
tiny-imagenet-200/train/{wordnet_id}/{unique_filename}.JPEG
```

Therefore, to extract the WordNet ID (i.e., class label), we simply need to split on the path separator and grab the third to last index/entry (Python is zero-indexed, so we supply a value of `-3` here). Once we have all the `trainLabels`, we can initiate the `LabelEncoder` and convert all labels to unique integers (**Lines 17 and 18**).

Since we do not have a testing split, we need to sample a set of images from the training set to form one:

```
20 # perform stratified sampling from the training set to construct a
21 # a testing set
22 split = train_test_split(trainPaths, trainLabels,
23      test_size=config.NUM_TEST_IMAGES, stratify=trainLabels,
24      random_state=42)
25 (trainPaths, testPaths, trainLabels, testLabels) = split
```

Here we supply the `trainPaths` and `trainLabels`, along with the `test_size` of `NUM_TEST_IMAGES`, which is 50 images per class (for a total of 10,000 images). Our testing set is sampled from our training set, where we already have the class labels for the images, enabling us to evaluate the performance of our neural network when we are ready; however, we have not parsed the validation labels yet.

Parsing and encoding the validation labels is handled in the following code block:

```
27 # load the validation filename => class from file and then use these
28 # mappings to build the validation paths and label lists
29 M = open(config.VAL_MAPPINGS).read().strip().split("\n")
```

```
30 M = [r.split("\t")[:2] for r in M]
31 valPaths = [os.path.sep.join([config.VAL_IMAGES, m[0]]) for m in M]
32 valLabels = le.transform([m[1] for m in M])
```

On **Line 29** we load the entire contents of the `VAL_MAPPINGS` file (i.e., the tab separated file that maps validation image file names to their respective WordNet ID). For every line inside the `M`, we split it into two columns – the image filename and the WordNet ID (**Line 30**). Based on the path to the validation images (`VAL_IMAGES`) along with the filenames in `M`, we can then construct the paths to the validation files (**Line 31**). Similarly, we can transform the WordNet ID string to a unique class label integer on **Line 32** by looping over the WordNet IDs in each row and applying the label encoder.

For readers who struggle to understand this section of code, I would suggest stopping here to spend a while executing each line and investigating the contents of every variable. We are making heavy use of Python *list comprehensions* here, which are natural, succinct methods to build lists with very little code. Again, this code block has *nothing* to do with deep learning – it's simply parsing a file which is a general purpose programming problem. Take a few minutes and ensure you understand how we are able to parse the `val_annotations.txt` file using this code block.

Now that we have the paths to our training, validation, and testing images, we can define a `datasets` tuple that we'll loop over and write the images and associated class labels for each set to HDF5, respectively:

```
34 # construct a list pairing the training, validation, and testing
35 # image paths along with their corresponding labels and output HDF5
36 # files
37 datasets = [
38      ("train", trainPaths, trainLabels, config.TRAIN_HDF5),
39      ("val", valPaths, valLabels, config.VAL_HDF5),
40      ("test", testPaths, testLabels, config.TEST_HDF5)]
```

We'll also initialize our RGB averages as well:

```
42 # initialize the lists of RGB channel averages
43 (R, G, B) = ([], [], [])
```

Finally, we are ready to build our HDF5 datasets for Tiny ImageNet:

```
45 # loop over the dataset tuples
46 for (dType, paths, labels, outputPath) in datasets:
47      # create HDF5 writer
48      print("[INFO] building {}...".format(outputPath))
49      writer = HDF5DatasetWriter((len(paths), 64, 64, 3), outputPath)
50
51      # initialize the progress bar
52      widgets = ["Building Dataset: ", progressbar.Percentage(), " ",
53           progressbar.Bar(), " ", progressbar.ETA()]
54      pbar = progressbar.ProgressBar(maxval=len(paths),
55           widgets=widgets).start()
```

On **Line 46** we loop over the dataset type (`dType`), `paths`, `labels`, and `outputPath` in the `datasets` list. For each of these output HDF5 files we'll create an `HDF5DatasetWriter` which will

store a total of `len(paths)` images, each of which is a $64 \times 64 \times 3$ RGB image (**Line 49**). **Lines 52-55** simply initialize a `progressbar` so we can easily visualize the dataset creation process.

We now need to loop over each `path` and `label` pair in the respective set:

```
57      # loop over the image paths
58      for (i, (path, label)) in enumerate(zip(paths, labels)):
59          # load the image from disk
60          image = cv2.imread(path)
61
62          # if we are building the training dataset, then compute the
63          # mean of each channel in the image, then update the
64          # respective lists
65          if dType == "train":
66              (b, g, r) = cv2.mean(image)[:3]
67              R.append(r)
68              G.append(g)
69              B.append(b)
70
71          # add the image and label to the HDF5 dataset
72          writer.add([image], [label])
73          pbar.update(i)
74
75      # close the HDF5 writer
76      pbar.finish()
77      writer.close()
```

For each image, we load it from disk on **Line 60**. If the image is a training image, we need to compute the RGB average of the image and update the respective lists (**Lines 66-69**). **Line 72** adds the `image` (which is already $64 \times 64 \times 3$) and `label` to the HDF5 dataset while **Line 77** closes the dataset.

The final step is to compute the RGB averages across the *entire* dataset and write them to disk:

```
79  # construct a dictionary of averages, then serialize the means to a
80  # JSON file
81  print("[INFO] serializing means...")
82  D = {"R": np.mean(R), "G": np.mean(G), "B": np.mean(B)}
83  f = open(config.DATASET_MEAN, "w")
84  f.write(json.dumps(D))
85  f.close()
```

To build the Tiny ImageNet dataset in HDF5 format, just execute the following command:

```
$ python build_tiny_imagenet.py
[INFO] building ../datasets/tiny-imagenet-200/hdf5/train.hdf5...
Building Dataset: 100% |#####################################| Time: 0:00:36
[INFO] building ../datasets/tiny-imagenet-200/hdf5/val.hdf5...
Building Dataset: 100% |#####################################| Time: 0:00:04
[INFO] building ../datasets/tiny-imagenet-200/hdf5/test.hdf5...
Building Dataset: 100% |#####################################| Time: 0:00:05
[INFO] serializing means...
```

After the script finishes executing, you'll have three files in your `hdf5` directory, `train.hdf5`, `val.hdf5`, and `test.hdf5`. You can investigate each of these files with the `h5py` library if you wish to validate that the datasets do indeed contain the images:

```
>>> import h5py
>>> filenames = ["train.hdf5", "val.hdf5", "test.hdf5"]
>>> for filename in filenames:
...     db = h5py.File(filename, "r")
...     print(db["images"].shape)
...     db.close()
...
(90000, 64, 64, 3)
(10000, 64, 64, 3)
(10000, 64, 64, 3)
```

We'll be using these HDF5 dataset representations of ImageNet to train both GoogLeNet in this chapter as well as ResNet in the following chapter.

11.4 DeeperGoogLeNet on Tiny ImageNet

Now that we have our HDF5 representation of the Tiny ImageNet dataset, we are ready to train GoogLeNet on it – but instead of using MiniGoogLeNet as in the previous section, we are going to use a *deeper variant* which more closely models the Szegedy et al. implementation. This deeper variation will use the *original* Inception module as detailed in Figure 11.1 earlier in this chapter, which will help you understand the original architecture and implement it on your own in the future.

To get started, we'll first learn how to implement this deeper network architecture. We'll then train DeeperGoogLeNet on the Tiny ImageNet dataset and evaluate the results in terms of rank-1 and rank-5 accuracy.

11.4.1 Implementing DeeperGoogLeNet

I have provided a figure (replicated and modified from Szegedy et al.) detailing our DeeperGoogLeNet architecture in Figure 11.9. There are only two primary differences between our implementation and the *full* GoogLeNet architecture used by Szegedy et al. when training the network on the complete ImageNet dataset:

1. Instead of using 7×7 filters with a stride of 2×2 in the first `CONV` layer, we use 5×5 filters with a 1×1 stride. We use these due to the fact that our implementation of GoogLeNet is only able to accept $64 \times 64 \times 3$ input images while the original implementation was constructed to accept $224 \times 224 \times 3$ images. If we applied 7×7 filters with a 2×2 stride, we would reduce our input dimensions too quickly.
2. Our implementation is *slightly* shallower with two fewer Inception modules – in the original Szegedy et al. paper, two more Inception modules were added prior to the average pooling operation. This implementation of GoogLeNet will be more than enough for us to perform well on Tiny ImageNet and claim a spot on the cs231n Tiny ImageNet leaderboard. For readers who are interested in training the full GoogLeNet architecture from scratch on the *entire* ImageNet dataset (thereby replicating the performance of the Szegedy et al. experiments), please refer to Chapter 7 in the *ImageNet Bundle*.

To implement our `DeeperGoogLeNet` class, let's create a file name `deeergooglenet.py` inside the `nn.conv` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
```

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj
convolution	5×5/2	112×112×64	1						
max pool	3×3/2	56×56×64	0						
convolution	3×3/1	56×56×192	2		64	192			
max pool	3×3/2	28×28×192	0						
inception (3a)		28×28×256	2	64	96	128	16	32	32
inception (3b)		28×28×480	2	128	128	192	32	96	64
max pool	3×3/2	14×14×480	0						
inception (4a)		14×14×512	2	192	96	208	16	48	64
inception (4b)		14×14×512	2	160	112	224	24	64	64
inception (4c)		14×14×512	2	128	128	256	24	64	64
inception (4d)		14×14×528	2	112	144	288	32	64	64
inception (4e)		14×14×832	2	256	160	320	32	128	128
max pool	3×3/2	7×7×832	0						
avg pool	7×7/1	1×1×1024	0						
dropout (40%)		1×1×1024	0						
linear		1×1×1000	1						
softmax		1×1×1000	0						

Figure 11.9: Our modified GoogLeNet architecture which we will call "DeeperGoogLeNet". The DeeperGoogLeNet architecture is identical to the original GoogLeNet architecture with two modifications: (1) 5×5 filters with a stride of 1×1 are used in the first `CONV` layer and (2) the final two inception modules (5a and 5b) are left out.

```
|    |--- callbacks
|    |--- io
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- alexnet.py
|    |    |    |--- deepergooglenet.py
|    |    |    |--- lenet.py
|    |    |    |--- minigooglenet.py
|    |    |    |--- minivggnet.py
|    |    |    |--- fcheadnet.py
|    |    |    |--- shallownet.py
|    |--- preprocessing
|    |--- utils
```

From there, we can start working on the implementation:

```
# import the necessary packages
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import AveragePooling2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Dropout
from keras.layers.core import Dense
from keras.layers import Flatten
```

```
10 from keras.layers import Input
11 from keras.models import Model
12 from keras.layers import concatenate
13 from keras.regularizers import l2
14 from keras import backend as K
```

Lines 2-14 start by importing our required Python packages. Notice how we'll be using the `Input` and `Model` classes as in our MiniGoogLeNet implementation so we can construct a *graph* rather than a *sequential* network – this graph construct is a requirement due to how the Inception module branches. Also take note of **Line 13** where we import the `l2` class, implying that we will allow L2 weight regularization in the network to help reduce overfitting.

As a matter of convenience (and to ensure our code doesn't become bloated), let's define a `conv_module` function that will be responsible for accepting an input layer, performing a `CONV => BN => RELU`, and then returning the output. Typically I would prefer to place the `BN` after the `RELU`, but since we are replicating the original work of Szegedy et al., let's stick with the batch normalization prior to the activation. The implementation of `conv_module` can be seen below:

```
16 class DeeperGoogLeNet:
17     @staticmethod
18     def conv_module(x, K, kX, kY, stride, chanDim,
19         padding="same", reg=0.0005, name=None):
20         # initialize the CONV, BN, and RELU layer names
21         (convName, bnName, actName) = (None, None, None)
22
23         # if a layer name was supplied, prepend it
24         if name is not None:
25             convName = name + "_conv"
26             bnName = name + "_bn"
27             actName = name + "_act"
28
29         # define a CONV => BN => RELU pattern
30         x = Conv2D(K, (kX, kY), strides=stride, padding=padding,
31             kernel_regularizer=l2(reg), name=convName)(x)
32         x = BatchNormalization(axis=chanDim, name=bnName)(x)
33         x = Activation("relu", name=actName)(x)
34
35         # return the block
36         return x
```

The `conv_module` method accepts a number of parameters, including:

- `x`: The input to the network.
- `K`: The number of filters the convolutional layer will learn.
- `kX` and `kY`: The filter size for the convolutional layer.
- `stride`: The stride (in pixels) for the convolution. Typically we'll use a 1×1 stride, but we could use a larger stride if we wished to reduce the output volume size.
- `chanDim`: This value controls the dimension (i.e., axis) of the image channel. It is automatically set later in this class based on whether we are using "channels_last" or "channels_first" ordering.
- `padding`: Here we can control the padding of the convolution layer.
- `reg`: The L2 weight decay strength.
- `name`: Since this network is deeper than all others we have worked with in this book, we may wish to name the blocks of layers to help us (1) debug the network and (2) share/explain the

network to others.

Line 21 initializes the name of each of the convolution, batch normalization, and activation layers, respectively. Provided that the `name` parameter to `conv_module` is not `None`, we then update the names of our layers (**Lines 24-27**).

Defining the `CONV => BN => RELU` layer pattern is accomplished on **Lines 30-33** – notice how the name of each layer is also included. Again, the primary benefit of naming each layer is that we can visualize the name in the output, just as we did in Chapter 19 in the *Starter Bundle*. For example, using the `plot_model` on the `conv_module` would result in a chart similar to Figure 11.10.

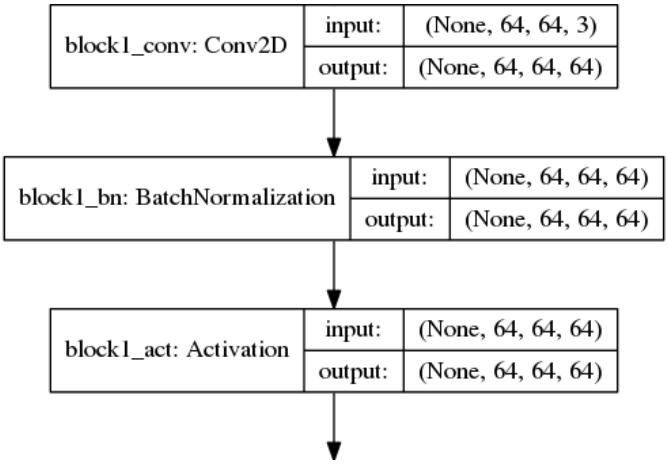


Figure 11.10: A sample of the DeeperGoogLeNet architecture visualization which includes the actual *names* for each layer. Naming layers makes it easier to keep track of them in larger, deeper networks.

Notice how the *names* of each layer are included in the chart, which is especially helpful when you are working with deep CNNs and can easily get "lost" examining the massive charts. The output of the `conv_module` is then returned to the calling function on **Line 36**.

Next, let's define the `inception_module`, as detailed by Szgedy et al. in their original publication and displayed in Figure 11.1:

```
38      @staticmethod
39      def inception_module(x, num1x1, num3x3Reduce, num3x3,
40           num5x5Reduce, num5x5, num1x1Proj, chanDim, stage,
41           reg=0.0005):
42           # define the first branch of the Inception module which
43           # consists of 1x1 convolutions
44           first = DeeperGoogLeNet.conv_module(x, num1x1, 1, 1,
45                (1, 1), chanDim, reg=reg, name=stage + "_first")
```

The Inception module includes four branches, the outputs of which are concatenated along the channel dimension. The `first` branch of in the Inception module simply performs a series of 1×1 convolutions – these enable the Inception module to learn local features.

The `second` branch of the Inception module first performs dimensionality reduction via 1×1 convolution followed by expanding with a 3×3 convolution – we call these our `num3x3Reduce` and `num3x3` variables, respectively:

```
47          # define the second branch of the Inception module which
48          # consists of 1x1 and 3x3 convolutions
49          second = DeeperGoogLeNet.conv_module(x, num3x3Reduce, 1, 1,
50              (1, 1), chanDim, reg=reg, name=stage + "_second1")
51          second = DeeperGoogLeNet.conv_module(second, num3x3, 3, 3,
52              (1, 1), chanDim, reg=reg, name=stage + "_second2")
```

Here we can see that the first `conv_module` applies the 1×1 convolutions to the input. The output of these 1×1 convolutions are then passed into the second `conv_module` which performs a series of 3×3 convolutions. The number of 1×1 convolutions is *always* smaller than the number of 3×3 convolutions, thereby serving as a form of dimensionality reduction.

The `third` branch in Inception is identical to the `second` branch, only instead of performing a 1×1 reduce followed by a 3×3 expand, we are now going to use a 1×1 reduce and a 5×5 expand:

```
54          # define the third branch of the Inception module which
55          # are our 1x1 and 5x5 convolutions
56          third = DeeperGoogLeNet.conv_module(x, num5x5Reduce, 1, 1,
57              (1, 1), chanDim, reg=reg, name=stage + "_third1")
58          third = DeeperGoogLeNet.conv_module(third, num5x5, 5, 5,
59              (1, 1), chanDim, reg=reg, name=stage + "_third2")
```

On **Lines 56 and 57** we learn `num5x5Reduce` kernels, each of size 1×1 based on the input to the `inception_module`. The output of the 1×1 convolutions are then passed into a second `conv_module` which then learns a `num5x5` filters, each of size 5×5. Again, the number of 1×1 convolutions in this branch is *always* smaller than the number of 5×5 filters.

The `fourth` and final branch of the Inception module is commonly referred to as the *pool projection*. Here we apply max pooling followed by a series of 1×1 convolutions:

```
61          # define the fourth branch of the Inception module which
62          # is the POOL projection
63          fourth = MaxPooling2D((3, 3), strides=(1, 1),
64              padding="same", name=stage + "_pool")(x)
65          fourth = DeeperGoogLeNet.conv_module(fourth, num1x1Proj,
66              1, 1, (1, 1), chanDim, reg=reg, name=stage + "_fourth")
```

The rationale for this branch is partially scientific and partially anecdotal. In 2014, most (if not all) Convolutional Neural Networks that were obtaining state-of-the-art performance on the ImageNet dataset were applying max pooling. Therefore, it was believed that a CNN *should* apply max pooling. While GoogLeNet *does* apply max pooling outside of the Inception module, Szegedy et al. decided to include the pool projection branch as another form of max pooling.

Now that we have all four branches computed, we can concatenate their output along the channel dimension and return the output to the calling function:

```
68          # concatenate across the channel dimension
69          x = concatenate([first, second, third, fourth], axis=chanDim,
70              name=stage + "_mixed")
71
72          # return the block
73          return x
```

If we were to call the `plot_model` function on the Inception module using the following parameters:

- `num1x1=64`
- `num3x3Reduce=96`
- `num3x3=128`
- `num5x5Reduce=16`
- `num5x5=32`
- `num1x1Proj=32`

The resulting graph would look like Figure 11.11. We can see in this visualization how the Inception module constructs four branches. The first branch is responsible for learning local 1×1 features. The second branch performs dimensionality reduction via a 1×1 convolution, followed by learning a larger filter size of 3×3. The third branch behaves similarly to the second branch, only learning 5×5 filters rather than 3×3 filters. Finally, the fourth branch applies max pooling.

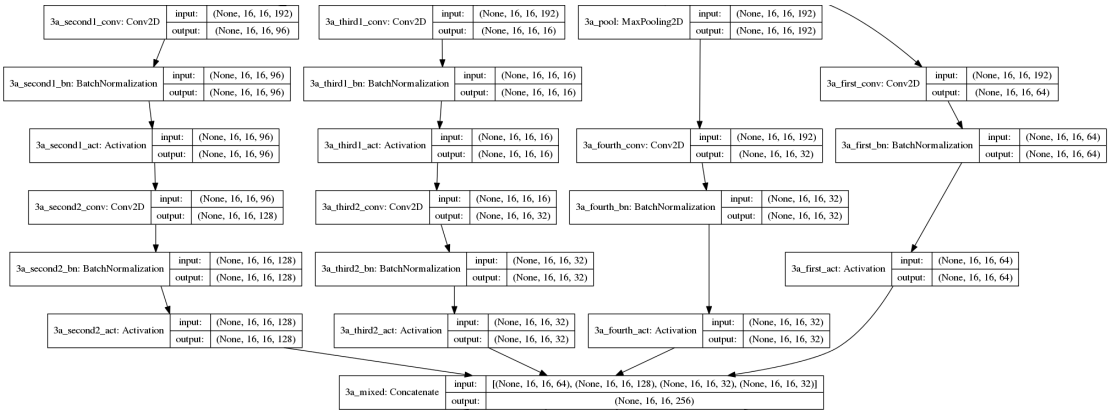


Figure 11.11: The full Inception module proposed by Szegedy et al. (zoomed out to save space). The key takeaway here is that there are four distinct branches in the Inception module.

By learning all three 1×1, 3×3, and 5×5 filters, the Inception module can learn both general (5×5 and 3×3) along with local (1×1) features *at the same time*. The actual optimization process will automatically determine how to value these branches and layers, essentially giving us a "general purpose" module that will learn the best set of features (local, small convolutions) or higher-level abstracted features (larger convolutions) at a given time. The output of the Inception module is thus 256, which is the concatenation of *all* the $64 + 128 + 32 + 32 = 256$ filters from each branch.

Now that the `inception_module` has been defined, we can create the `build` method responsible for constructing the complete `DeeperGoogLeNet` architecture:

```
75      @staticmethod
76      def build(width, height, depth, classes, reg=0.0005):
77          # initialize the input shape to be "channels last" and the
78          # channels dimension itself
79          inputShape = (height, width, depth)
80          chanDim = -1
81
82          # if we are using "channels first", update the input shape
83          # and channels dimension
84          if K.image_data_format() == "channels_first":
```

```
85                inputShape = (depth, height, width)
86                chanDim = 1
```

Our `build` method will accept the spatial input dimensions of our images, including the width, height, and depth. We'll also be able to supply the number of class labels the network is to learn, along with an optional regularization term for L2 weight decay. **Lines 77-86** then handle properly setting the `inputShape` and `chanDim` based on the "channels last" or "channels first" configuration in Keras.

Following Figure 11.9 above, our first block of layers will perform a sequence of `CONV => POOL => (CONV * 2) => POOL`:

```
88            # define the model input, followed by a sequence of CONV =>
89            # POOL => (CONV * 2) => POOL layers
90            inputs = Input(shape=inputShape)
91            x = DeeperGoogLeNet.conv_module(inputs, 64, 5, 5, (1, 1),
92                 chanDim, reg=reg, name="block1")
93            x = MaxPooling2D((3, 3), strides=(2, 2), padding="same",
94                 name="pool1")(x)
95            x = DeeperGoogLeNet.conv_module(x, 64, 1, 1, (1, 1),
96                 chanDim, reg=reg, name="block2")
97            x = DeeperGoogLeNet.conv_module(x, 192, 3, 3, (1, 1),
98                 chanDim, reg=reg, name="block3")
99            x = MaxPooling2D((3, 3), strides=(2, 2), padding="same",
100                name="pool2")(x)
```

The first `CONV` layer learns 64 5×5 filters with a stride of 1×1. We then apply max pooling with a window size of 3×3 and stride of 2×2 to reduce the volume size of the input. **Lines 95-98** are responsible for performing a reduce and expand. First, 64 1×1 filters are learned (**Lines 95 and 96**). Then, 192 3×3 filters are learned on **Lines 97 and 98**. This process is very similar to the Inception module (which will be applied in later layers of the network), only without the branching factor. Finally, another max pooling is performed on **Lines 99 and 100**.

Next, let's apply two Inception modules (3a and 3b) followed by a max pooling:

```
102           # apply two Inception modules followed by a POOL
103           x = DeeperGoogLeNet.inception_module(x, 64, 96, 128, 16,
104                32, 32, chanDim, "3a", reg=reg)
105           x = DeeperGoogLeNet.inception_module(x, 128, 128, 192, 32,
106                96, 64, chanDim, "3b", reg=reg)
107           x = MaxPooling2D((3, 3), strides=(2, 2), padding="same",
108                name="pool3")(x)
```

Looking at this code, you might wonder how we decided on the number of filters for each `CONV` layer. The answer is that all parameter values in this network were taken *directly* from the original Szegedy et al. paper on GoogLeNet where the authors performed a number of experiments to tune the parameters. In every case, you'll notice a common pattern with all Inception modules:

1. The number of 1×1 filters we learn in the first branch of the Inception module will be less than or equal to the 1×1 filters in the 3×3 (second) branch The 1×1 filters will also be greater than or equal to the number of 5×5 filters in the third branch..
2. The number of 1×1 filters will *always* be smaller than the 3×3 and 5×5 convolutions they feed into.

3. The number of filters we learn in the 3×3 branch will be more than the 5×5 branch which helps reduce the size of the network as well as improves the speed of training/evaluation.
4. The number of pool projection filters will always be smaller than the first branch of 1×1 local features.
5. Regardless of branch type, the number of filters will *increase* (or at least remain the same) as we go deeper in the network.

While there are certainly more parameters to keep track of when implementing this network, we're still following the same general rules of thumb as in previous CNNs – the deeper the network gets, the smaller the volume size is; therefore, the more filters we learn to compensate.

The network continues to become deeper, learning richer features as we now stack five Inception modules (4a-4e) on top of each other before applying a `POOL`:

```
110         # apply five Inception modules followed by POOL
111         x = DeeperGoogLeNet.inception_module(x, 192, 96, 208, 16,
112             48, 64, chanDim, "4a", reg=reg)
113         x = DeeperGoogLeNet.inception_module(x, 160, 112, 224, 24,
114             64, 64, chanDim, "4b", reg=reg)
115         x = DeeperGoogLeNet.inception_module(x, 128, 128, 256, 24,
116             64, 64, chanDim, "4c", reg=reg)
117         x = DeeperGoogLeNet.inception_module(x, 112, 144, 288, 32,
118             64, 64, chanDim, "4d", reg=reg)
119         x = DeeperGoogLeNet.inception_module(x, 256, 160, 320, 32,
120             128, 128, chanDim, "4e", reg=reg)
121         x = MaxPooling2D((3, 3), strides=(2, 2), padding="same",
122             name="pool4")(x)
```

After the final `POOL` on **Lines 121 and 122**, our volume size is `4 x 4 x classes`. To avoid the usage of computationally expensive fully-connected layers (not to mention, dramatically increased network size), we apply average pooling with a 4×4 kernel to reduce the volume size to `1 x 1 x classes`:

```
124         # apply a POOL layer (average) followed by dropout
125         x = AveragePooling2D((4, 4), name="pool5")(x)
126         x = Dropout(0.4, name="do")(x)
127
128         # softmax classifier
129         x = Flatten(name="flatten")(x)
130         x = Dense(classes, kernel_regularizer=l2(reg),
131             name="labels")(x)
132         x = Activation("softmax", name="softmax")(x)
133
134         # create the model
135         model = Model(inputs, x, name="googlenet")
136
137         # return the constructed network architecture
138         return model
```

Dropout is then applied with a probability of 40%. Typically we would use a 50% dropout rate, but again, we are simply following the original implementation.

Lines 130 and 131 create the `Dense` layer for the total number of `classes` we wish to learn. A softmax classifier is then applied after the fully-connected layer on **Line 132**. Finally, the actual `Model` is constructed based on the `inputs` and `x`, the actual *computational network graph*. This `model` is returned to the calling function on **Line 138**.

11.4.2 Training DeeperGoogLeNet on Tiny ImageNet

Now that our DeeperGoogLeNet architecture is implemented, we need to create a Python script that will train the network on Tiny ImageNet. We'll also need to create a second Python script that will be responsible for evaluating our model on the testing set by computing rank-1 and rank-5 accuracies.

Once we have completed both of these tasks, I'll share three experiments I ran when gathering the results for this chapter. These experiments will form a "case study" and enable you to learn how to run an experiment, investigate the results, and make an educated guess on how to tune your hyperparameters to obtain a better performing network in your next experiment.

11.4.3 Creating the Training Script

Let's go ahead and implement the training script – open up a new file, name it `train.py`, and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from config import tiny_imagenet_config as config
7  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
8  from pyimagesearch.preprocessing import SimplePreprocessor
9  from pyimagesearch.preprocessing import MeanPreprocessor
10 from pyimagesearch.callbacks import EpochCheckpoint
11 from pyimagesearch.callbacks import TrainingMonitor
12 from pyimagesearch.io import HDF5DatasetGenerator
13 from pyimagesearch.nn.conv import DeeperGoogLeNet
14 from keras.preprocessing.image import ImageDataGenerator
15 from keras.optimizers import Adam
16 from keras.models import load_model
17 import keras.backend as K
18 import argparse
19 import json
```

Lines 2 and 3 instruct the matplotlib library to use a backend such that we can save our loss and accuracy plots to disk. We then import the remainder of our required Python packages on **Lines 6-19**. Take a look at **Line 6** where we import the configuration file for the Tiny ImageNet experiment. We also import our implementation of `DeeperGoogLeNet` on **Line 13**. All the imports should feel relatively familiar to you now.

From there, we can parse our command line arguments:

```
21 # construct the argument parse and parse the arguments
22 ap = argparse.ArgumentParser()
23 ap.add_argument("-c", "--checkpoints", required=True,
24      help="path to output checkpoint directory")
25 ap.add_argument("-m", "--model", type=str,
26      help="path to *specific* model checkpoint to load")
27 ap.add_argument("-s", "--start-epoch", type=int, default=0,
28      help="epoch to restart training at")
29 args = vars(ap.parse_args())
```

We'll be using the `ctrl + c` method to train our network, meaning that we'll start the training process, monitor how the training is going, then stop script if overfitting/stagnation occurs, adjust any hyperparameters, and restart training. To start, we'll first need the `--checkpoints` switch, which is the path to the output directory that will store individual checkpoints for the `DeeperGoogLeNet` model. If we are restarting training, then we'll need to supply the path to a *specific* `--model` that we are restarting training from. Similarly, we'll also need to supply `--start-epoch` to obtain the integer value of the epoch we are restarting training from.

In order to obtain reasonable accuracy on the Tiny ImageNet dataset, we'll need to apply data augmentation to the training data:

```
31 # construct the training image generator for data augmentation
32 aug = ImageDataGenerator(rotation_range=18, zoom_range=0.15,
33     width_shift_range=0.2, height_shift_range=0.2, shear_range=0.15,
34     horizontal_flip=True, fill_mode="nearest")
35
36 # load the RGB means for the training set
37 means = json.loads(open(config.DATASET_MEAN).read())
```

We'll also load our RGB means (**Line 37**) for mean subtraction and normalization.

Let's move on to instantiating both our image pre-processors as well as the training and validation HDF5 dataset generators:

```
39 # initialize the image preprocessors
40 sp = SimplePreprocessor(64, 64)
41 mp = MeanPreprocessor(means["R"], means["G"], means["B"])
42 iap = ImageToArrayPreprocessor()
43
44 # initialize the training and validation dataset generators
45 trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, 64, aug=aug,
46     preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
47 valGen = HDF5DatasetGenerator(config.VAL_HDF5, 64,
48     preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
```

Both the training and validation generator will apply:

1. A simple preprocessor to ensure the image is resized to 64×64 pixels (which it already should be, but we'll include it here as a matter of completeness).
2. Mean subtraction to normalize the data.
3. An image to Keras-compatible array converter.

We'll be training our network in mini-batch sizes of 64. In the case that we are training DeeperGoogLeNet from the first epoch, we must instantiate the network and optimizer:

```
50 # if there is no specific model checkpoint supplied, then initialize
51 # the network and compile the model
52 if args["model"] is None:
53     print("[INFO] compiling model...")
54     model = DeeperGoogLeNet.build(width=64, height=64, depth=3,
55         classes=config.NUM_CLASSES, reg=0.0002)
56     opt = Adam(1e-3)
57     model.compile(loss="categorical_crossentropy", optimizer=opt,
58         metrics=["accuracy"])
```

Notice here how we are applying a L2 regularization strength of 0.0002 as well as the Adam optimizer – we'll find out why in Section 11.4.5 below.

Otherwise, we must be restarting training from a specific epoch, so we'll need to load the model and adjust the learning rate:

```
60 # otherwise, load the checkpoint from disk
61 else:
62     print("[INFO] loading {}...".format(args["model"]))
63     model = load_model(args["model"])
64
65     # update the learning rate
66     print("[INFO] old learning rate: {}".format(
67         K.get_value(model.optimizer.lr)))
68     K.set_value(model.optimizer.lr, 1e-5)
69     print("[INFO] new learning rate: {}".format(
70         K.get_value(model.optimizer.lr)))
```

We'll create two `callbacks`, one to serialize the model weights to disk every five epochs and another to create our loss/accuracy plot over time:

```
72 # construct the set of callbacks
73 callbacks = [
74     EpochCheckpoint(args["checkpoints"], every=5,
75         startAt=args["start_epoch"]),
76     TrainingMonitor(config.FIG_PATH, jsonPath=config.JSON_PATH,
77         startAt=args["start_epoch"])]
```

Finally, we can train our network:

```
79 # train the network
80 model.fit_generator(
81     trainGen.generator(),
82     steps_per_epoch=trainGen.numImages // 64,
83     validation_data=valGen.generator(),
84     validation_steps=valGen.numImages // 64,
85     epochs=10,
86     max_queue_size=10,
87     callbacks=callbacks, verbose=1)
88
89 # close the databases
90 trainGen.close()
91 valGen.close()
```

The exact number of epochs we choose to train our network for will depend on how our loss/accuracy plots look. We will make an informed decision regarding updating learning rates or applying early stopping based on model performance.

11.4.4 Creating the Evaluation Script

Once we are satisfied with our model performance on the training and validation set, we can move on to evaluating the network on the testing set. To do so, let's create a new file named `rank_accuracy.py`:

```
# import the necessary packages
from config import tiny_imagenet_config as config
from pyimagesearch.preprocessing import ImageToArrayPreprocessor
from pyimagesearch.preprocessing import SimplePreprocessor
from pyimagesearch.preprocessing import MeanPreprocessor
from pyimagesearch.utils.ranked import rank5_accuracy
from pyimagesearch.io import HDF5DatasetGenerator
from keras.models import load_model
import json
```

To start, we'll import our required Python packages. The `rank5_accuracy` function is imported on **Line 6** so we can compute both the rank-1 and rank-5 accuracy on the dataset.

From there, we load our RGB means, initialize our image pre-processors (in the same manner as we did for testing), and then initialize the testing dataset generator:

```
# load the RGB means for the training set
means = json.loads(open(config.DATASET_MEAN).read())

# initialize the image preprocessors
sp = SimplePreprocessor(64, 64)
mp = MeanPreprocessor(means["R"], means["G"], means["B"])
iap = ImageToArrayPreprocessor()

# initialize the testing dataset generator
testGen = HDF5DatasetGenerator(config.TEST_HDF5, 64,
	preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
```

The following code block handles loading the pre-trained model from disk via the `MODEL_PATH` we supply in our configuration file:

```
# load the pre-trained network
print("[INFO] loading model...")
model = load_model(config.MODEL_PATH)
```

You should set `MODEL_PATH` to be the final epoch checkpoint after training is complete. Alternatively, you can set this variable to *earlier* epochs to obtain an understanding on how testing accuracy increases in later epochs.

Once the model is loaded we can make predictions on the testing data and display both the rank-1 and rank-5 accuracies:

```
# make predictions on the testing data
print("[INFO] predicting on test data...")
predictions = model.predict_generator(testGen.generator(),
	steps=testGen.numImages // 64, max_queue_size=10)

# compute the rank-1 and rank-5 accuracies
(rank1, rank5) = rank5_accuracy(predictions, testGen.db["labels"])
print("[INFO] rank-1: {:.2f}%".format(rank1 * 100))
print("[INFO] rank-5: {:.2f}%".format(rank5 * 100))

# close the database
testGen.close()
```

Epoch	Learning Rate
$1-25$	$1e-2$
$26-35$	$1e-3$
$36-65$	$1e-4$

Table 11.1: Learning rate schedule used when training DeeperGoogLeNet on Tiny ImageNet for Experiment #1.

11.4.5 DeeperGoogLeNet Experiments

In the following sections I have included the results of three separate experiments I ran when training DeeperGoogLeNet on Tiny ImageNet. After each experiment I evaluated the results and then made an educated decision on how the hyperparameters and network architecture should be updated to increase accuracy.

Case studies like these are especially helpful to you as a budding deep learning practitioner. Not only do they demonstrate that deep learning is an *iterative* process requiring many experiments, but they also show *which* parameters you should be paying attention to and *how* to update them.

Finally, it's worth noting that some of these experiments required changes to the code. Both the implementations of `deepergooglenet.py` and `train.py` are my *final* implementations that obtained the best accuracy. I'll note the changes I made in earlier experiments in case you want to replicate my (less accurate) results.

DeeperGoogLeNet: Experiment #1

Given that this was my first time training a network on the Tiny ImageNet challenge, I wasn't sure what the optimal depth should be for a given architecture on this dataset. While I knew Tiny ImageNet would be a challenging classification task, I didn't think Inception modules 4a-4e were required, so I removed them from our DeeperGoogLeNet implementation above, leading to a substantially more shallow network architecture.

I decided to train DeeperGoogLenet using SGD with an initial learning rate of $1e-2$ and momentum term of 0.9 (no Nesterov acceleration was applied). I *always* use SGD in my first experiment. Per my guidelines and rules of thumb in Chapter 7, you should first try SGD to obtain a baseline, and then if need be, use more advanced optimization methods.

I started training using the following command:

```
$ python train.py --checkpoints output/checkpoints
```

The learning rate schedule detailed in Table 11.1 was then used. This table implies that after epoch 25 I stopped training, lowered the learning rate to $1e-3$, then resumed training for another 10 epochs:

```
$ python train.py --checkpoints output/checkpoints \
    --model output/checkpoints/epoch_25.hdf5 --start-epoch 25
```

After epoch 35 I again stopped training, lowered the learning rate to $1e-4$, and then resumed training for thirty more epochs:

```
$ python train.py --checkpoints output/checkpoints \
    --model output/checkpoints/epoch_35.hdf5 --start-epoch 35
```

Training for an extra thirty epochs was excessive, to say the least; however, I wanted to get a feel for the level of overfitting to expect for a large number of epochs after the original learning rate had been dropped (as this was the first time I had worked with GoogLeNet + Tiny ImageNet). You can see a plot of the loss/accuracy over time for both the training and validation in Figure 11.12 (*top-left*).

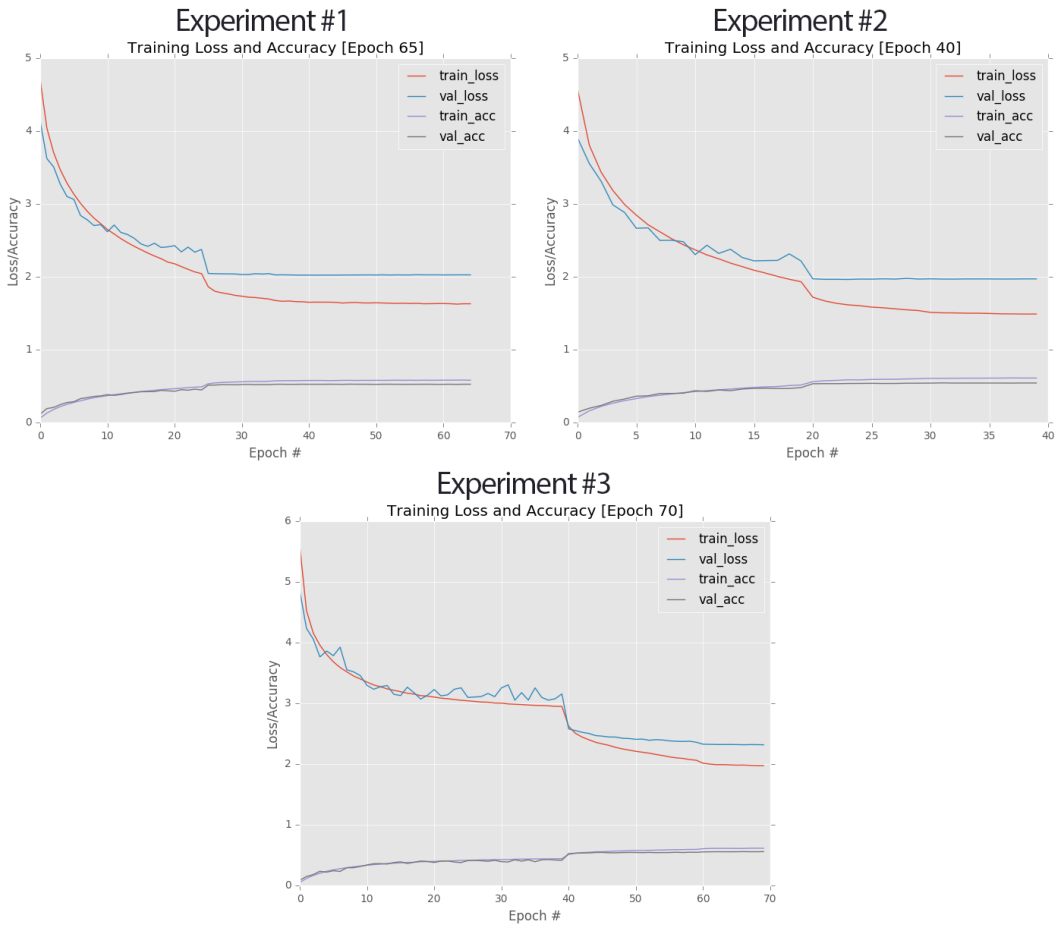


Figure 11.12: **Top-left:** Plots for Experiment #1. **Top-right:** Learning curves for Experiment #2. **Bottom:** Training/validation plots for Experiment #3. The final experiment obtains the best validation accuracy at 55.77%.

Starting at approximately epoch 15, there is a divergence in training and validation loss. By the time we get to epoch 25, the divergence is getting more significant, so I lowered the learning rate by an order of magnitude; the result is a nice jump in accuracy and decrease in loss. The problem is that after this point, both training and validation learning essentially stagnate. Even lowering the learning rate to $1e-4$ at epoch 35 does not introduce an extra boost in accuracy.

At the end of epoch 40 learning has stagnated completely. Had I not wanted to see the effects of a low learning rate for a long period of time, I would have stopped training after epoch 45. In this case, I let the network train until epoch 65 (with no change in loss/accuracy) where I stopped training and examined the results, noting that the network was obtaining 52.25% rank-1 accuracy on the validation set. However, given how our network performance plateaued quickly after dropping the learning rate, I determined that there is clearly more work to be done.

Epoch	Learning Rate
$1-20$	$1e-3$
$21-30$	$1e-4$
$31-40$	$1e-5$

Epoch	Learning Rate
$1-40$	$1e-3$
$41-60$	$1e-4$
$61-70$	$1e-5$

Table 11.2: **Left:** The learning rate schedule used when training DeeperGoogLeNet on Tiny ImageNet in Experiment #2. **Right:** The learning rate schedule for Experiment #3.

DeeperGoogLeNet: Experiment #2

In my second experiment with DeeperGoogLeNet + Tiny ImageNet, I decided to switch out the SGD optimizer for Adam. This decision was made strictly because I wasn't convinced that the network architecture needed to be deeper (yet). The Adam optimizer was used with the default initial learning rate of $1e-3$. I then used the learning rate schedule in Table 11.2 (*left*) to lower the learning rate.

A plot of the learning curves can be seen below in Figure 11.12 (*top-right*). This plot looks very similar to the *top-left* above. We initially start off strong, but validation loss diverges quickly past epoch 10, forcing me to lower the learning rate at epoch 20 (or otherwise risk overfitting). As soon as the learning rate is reduced, learning plateaus and I am unable to increase accuracy, even reducing learning rate a second time. However, at the end of the 40th epoch, I noticed that my validation loss was lower than the previous experiment *and* my accuracy was higher.

By swapping out SGD for Adam, I was able to boost validation accuracy to 54.20% rank-1, an increase of nearly 2%. However, I still had the issue of learning stagnation as soon as the initial learning rate was lowered.

DeeperGoogLeNet: Experiment #3

Given the learning stagnation, I postulated that the network was not deep enough to model the underlying patterns in the Tiny ImageNet dataset. Therefore, I decided to enable the Inception modules 4a-4e, creating a *much* deeper network architecture capable of learning deeper, more discriminative features. The Adam optimizer with an initial learning rate of $1e-3$ was used to train the network. I left the L2 weight decay term at 0.0002. DeeperGoogLeNet was then trained according to Table 11.2 (*right*).

You can see the plot in Figure 11.12 (*bottom*). Immediately you'll notice that using the deeper network architecture enabled me to train for *longer* without risking stagnation or severe overfitting. At the end of the 70th epoch, I was obtaining 55.77% rank-1 accuracy on the validation set.

At this point, I decided it was time to evaluate DeeperGoogLeNet on the testing set:

```
$ python rank_accuracy.py
[INFO] loading model...
[INFO] predicting on test data...
[INFO] rank-1: 54.38%
[INFO] rank-5: 78.96%
```

The evaluation script reported a rank-1 accuracy of 54.38%, or an error rate of $1-0.5438=0.4562$. It's also interesting to note that our rank-5 accuracy is 78.96%, which is quite impressive for this challenge. Looking at the Tiny ImageNet leaderboard (http://pyimg.co/h5q0o) below, we can see this error rate is enough to claim the #7 position, a great start to journey to climb to the top of the leaderboard (Figure 11.13).

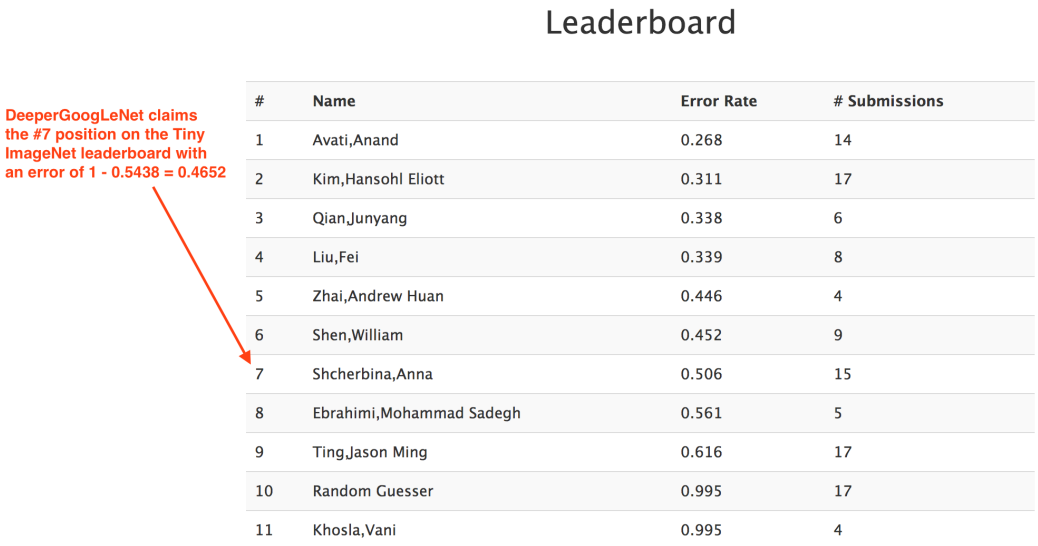


Figure 11.13: Our first successful attempt at training DeeperGoogLeNet on the Tiny ImageNet dataset allows us to claim the #7 position on the leaderboard. We'll be able to reach higher positions in our next chapter on ResNet.

Positions 1-4 on the Tiny ImageNet leaderboard were achieved by transfer learning via fine-tuning on networks *already* trained on the full ImageNet dataset. Since we are training our networks from *scratch*, we are more concerned with claiming the #5 position, the highest position achieved without transfer learning. As we'll find out in the next chapter on ResNet, we'll easily be able to claim this position. For more information on the techniques the cs231n students used to achieve their error rates, please see the Stanford cs231n project page [4].

For readers interested in trying to boost the accuracy of DeeperGoogLeNet further, I would suggest the following experiments:

1. Change the `conv_module` to use `CONV => RELU => BN` instead of the original `CONV => BN => RELU` ordering.
2. Attempt using ELUs instead of ReLUs, which will likely lead to a small $0.5 - 1\%$ gain in accuracy.

11.5 Summary

In this chapter, we reviewed the work of Szgedy et al. [17] which introduced the now famous Inception module. The Inception module is an example of a *micro-architecture*, a building block that fits into the overall *macro-architecture* of the network. Current state-of-the-art Convolutional Neural Networks tend to use some form of micro-architecture.

We then applied the Inception module to create two variants of GoogLeNet:

1. One for CIFAR-10.
2. And another for the more challenging Tiny ImageNet.

When training on CIFAR-10, we obtained our best accuracy thus far of $\mathbf{90.81\%}$ (and improvement from the previous best of 84%).

On the challenging Tiny ImageNet dataset we reached $\mathbf{54.38\%}$ rank-1 and $\mathbf{78.96\%}$ rank-5 accuracy on the testing set, enabling us to claim position #7 on the Tiny ImageNet leaderboard. Our goal is to climb the leaderboard to position #5 (the highest position obtained when training a network from scratch, all higher positions applied fine-tuning on networks *pre-trained* on the

ImageNet dataset, giving them an unfair advantage). To reach our goal of position #5, we'll need to use the ResNet architecture detailed in the following chapter.

12. ResNet

In our previous chapter, we discussed the GoogLeNet architecture and the Inception module, a *micro-architecture* that acts as a building block in the overall *macro-architecture*. We are now going to discuss another network architecture that relies on micro-architectures – *ResNet*.

ResNet uses what's called a *residual module* to train Convolutional Neural Networks to depths previously thought impossible. For example, in 2014, the VGG16 and VGG19 architectures were considered *very* deep [11]. However, with ResNet, we have successfully trained networks with > 100 *layers* on the challenging ImageNet dataset and *over 1,000 layers* on CIFAR-10 [24].

These depths are only made possible by using "smarter" weight initialization algorithms (such as Xavier/Glorot [44] and MSRA/He et al. [45]) along with *identity mapping*, a concept we'll discuss later in this chapter. Given the depths of ResNet networks, perhaps it comes as no surprise that ResNet took first place in *all three* ILSVRC 2015 challenges (classification, detection, and localization).

In this chapter, we are going to discuss the ResNet architecture, the residual module, along with updates to the residual module that have made it capable of obtaining higher classification accuracy. From there we'll implement and train variants of ResNet on the CIFAR-10 dataset and the Tiny ImageNet challenge – in each case, our ResNet implementations will outperform every experiment we have executed in this book.

12.1 ResNet and the Residual Module

First introduced by He et al. in their 2015 paper, *Deep Residual Learning for Image Recognition* [24], the ResNet architecture has become a seminal work, demonstrating that *extremely deep* networks can be trained using standard SGD and a reasonable initialization function. In order to train networks at depths greater than 50-100 (and in some cases, 1,000) layers, ResNet relies on a micro-architecture called the *residual module*.

Another interesting component of ResNet is that pooling layers are used *extremely sparingly*. Building on the work of Springenberg et al. [41], ResNet *does not* strictly rely on max pooling operations to reduce volume size. Instead, convolutions with strides > 1 are used to not only learn

weights, but reduce the output volume spatial dimensions. In fact, there are only two occurrences of pooling being applied in the full implementation of the architecture:

1. The first (and only) occurrence of max pooling happens early in the network to help reduce spatial dimensions.
2. The second pooling operation is actually an *average pooling layer* used in place of fully-connected layers, like in GoogLeNet.

Strictly speaking, there is only *one* max pooling layer – all other reductions in spatial dimensions are handled by convolutional layers.

In this section, we'll review the *original* residual module, along with the bottleneck residual module used to train deeper networks. From there, we'll discuss extensions and updates to the original residual module by He et al. in their 2016 publication, *Identity Mappings in Deep Residual Networks* [33], that allow us to further increase classification accuracy. Later in this chapter, we'll implement ResNet from scratch using Keras.

12.1.1 Going Deeper: Residual Modules and Bottlenecks

The original residual module introduced by He et al. in 2015 relies on *identity mappings*, the process of taking the *original input* to the module and adding it to the *output* of a series of operations. A graphical depiction of this module can be seen in Figure 12.1 (*left*). Notice how this module only has two branches, unlike the four branches in the Inception module of GoogLeNet. Furthermore, this module is *highly simplistic*.

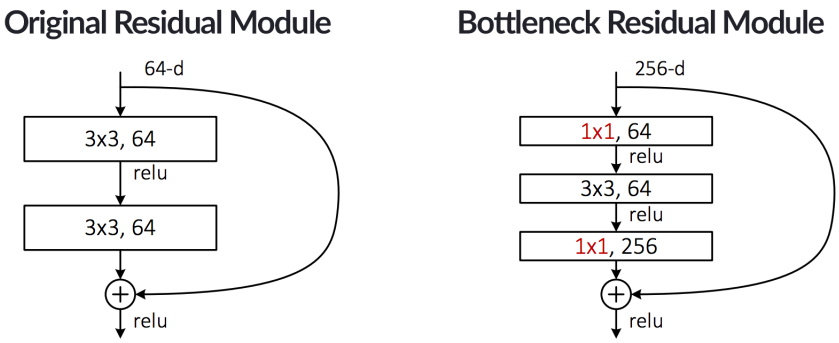


Figure 12.1: **Left:** The original Residual module proposed by He et al. **Right:** The more commonly used bottleneck variant of the Residual module.

At the top of the module, we accept an *input* to the module (i.e., the previous layer in the network). The right branch is a "linear shortcut" – it connects the input to an addition operation at the bottom of the module. Then, on the left branch of the residual module, we apply a series of convolutions (all of which are 3×3), activations, and batch normalizations. This is a fairly standard pattern to follow when constructing Convolutional Neural Networks.

But what makes ResNet interesting is that He et al. suggested *adding* the *original input* to the *output* of the `CONV`, `RELU` and `BN` layers. We call this addition an *identity mapping* since the input (the identity) is added to the output of series of operations. It is also why the term "residual" is used. The "residual" input is added to the output of a series of layer operations. The connection between the input and the addition node is called the ***shortcut***. Note that we are *not* referring to concatenation along the channel dimension as we have done in previous chapters. Instead, we are performing simple $1 + 1 = 2$ addition at the bottom of the module between the two branches.

While traditional neural network layers can be seen as learning a function $y = f(x)$, a residual layer attempts to approximate y via $f(x) + id(x) = f(x) + x$ where $id(x)$ is the identity function.

These residual layers *start* at the identity function and *evolve* to become more complex as the network learns. This type of residual learning framework allows us to train networks that are *substantially deeper* than previously proposed network architectures.

Furthermore, since the input is included in every residual module, it turns out the network can learn *faster* and with *larger learning rates*. It is very common to see the base learning rates for ResNet implementations start at $1e-1$. For most architectures such as AlexNet or VGGNet, this high of a learning rate would almost guarantee the network would not converge. But since ResNet relies on residual modules via identity mappings, this higher learning rate is completely possible.

In the same 2015 work, He et al. also included an extension to the original residual module called ***bottlenecks*** (Figure 12.1, *right*). Here we can see that the same identity mapping is taking place, only now the `CONV` layers in the left branch of the residual module have been updated:

1. We are utilizing three `CONV` layers rather than just two.
2. The first and last `CONV` layers are 1×1 convolutions.
3. The number of filters learned in the first two `CONV` layers are $1/4$ the number of filters learned in the final `CONV`.

To understand why we call this a "bottleneck", consider the following figure where two residual modules are stacked on top of each other, with one residual feeding into the next (Figure 12.2).

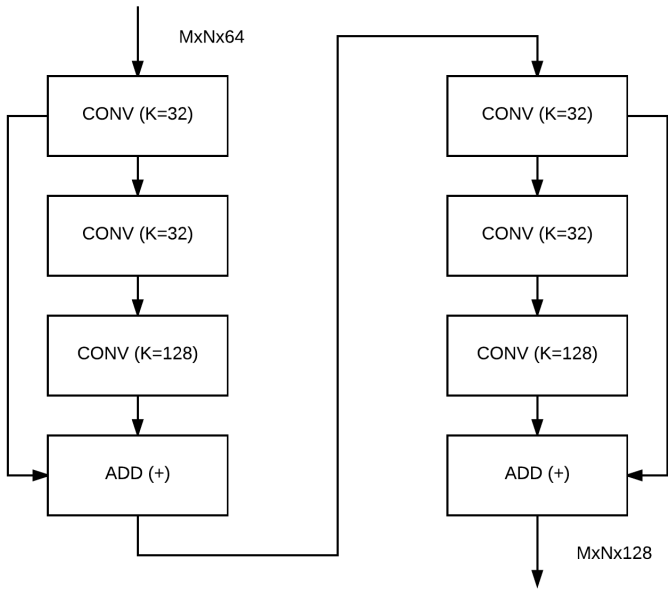


Figure 12.2: An example of two stacked residual modules where one feeds into next. Both modules learn $K = 32$, 32, and 128 filters, respectively. Notice how the dimensionality is reduced during the first two `CONV` layers then increased during the final `CONV` layer.

The first residual module accepts an input volume of size $M \times N \times 64$ (the actual width and height are arbitrary for this example). The three `CONV` layers in the first residual module learn $K = 32$, 32, and 128 filters, respectively. After applying the first residual module our output volume size is $M \times N \times 128$ which is then fed into the second residual module.

In the second residual module, our number of filters learned by each of the three `CONV` layers stays the same at $K = 32$, 32, and 128, respectively. However, notice that $32 < 128$, implying that we are actually reducing the volume size during the 1×1 and 3×3 `CONV` layers. This result has the benefit of leaving the 3×3 bottleneck layer with smaller input and output dimensions.

The final 1×1 `CONV` then applies $4x$ the number of filters than the first two `CONV` layers, thereby

increasing dimensionality once again, which is why we call this update to the residual module the "bottleneck" technique. When building our own residual modules, it's common to supply pseudocode such as `residual_module(K=128)` which implies that the *final* `CONV` layer will learn 128 filters, while the first two will learn $128/4 = 32$ filters. This notation is often easier to work with as it's understood that the bottleneck `CONV` layers will learn $1/4th$ the number of filters as the final `CONV` layer.

When it comes to training ResNet, we typically use the bottleneck variant of the residual module rather than the original version, *especially* for ResNet implementations with > 50 layers.

12.1.2 Rethinking the Residual Module

In 2016, He et al. published a second paper on the residual module entitled *Identity Mappings in Deep Residual Networks* [33]. This publication described a comprehensive study, both theoretically and empirically, on the ordering of convolutional, activation, and batch normalization layers *within* the residual module itself. Originally, the residual module (with bottleneck) looked like Figure 12.3 (*left*).

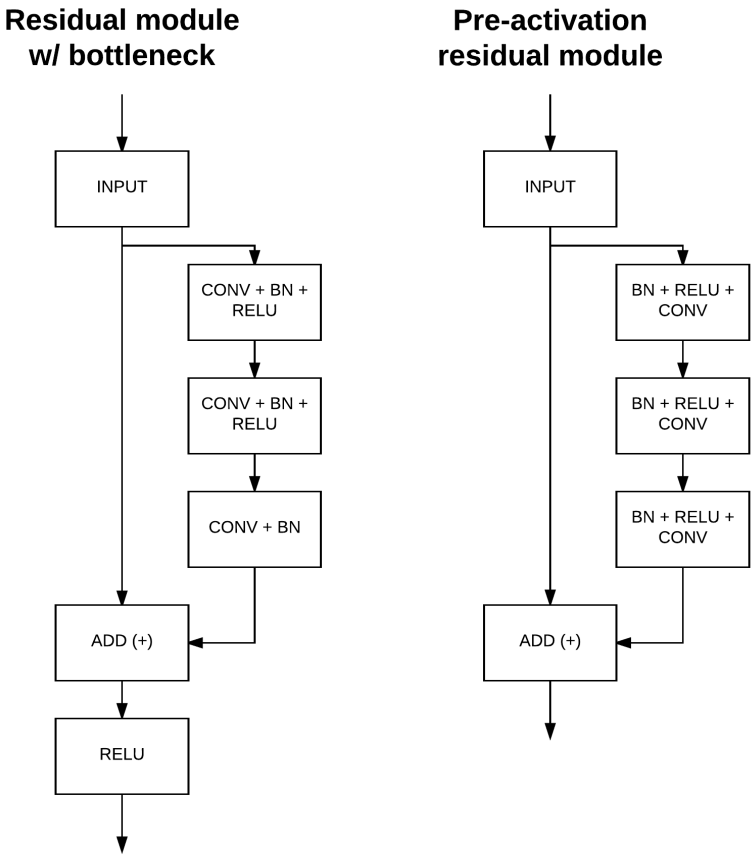


Figure 12.3: **Left:** The original residual module with bottleneck. **Right:** Adapting the bottleneck module to use pre-activations.

The original residual module with bottleneck accepts an input (a ReLU activation map) and then applies a series of `(CONV => BN => RELU) * 2 => CONV => BN` before adding this output to the original input and applying a final ReLU activation (which is then fed into the next residual module in the network). However, the He et al. 2016 study, it was found there was a more optimal layer ordering capable of obtaining higher accuracy – this method is called ***pre-activation***.

In the pre-activation version of the residual module, we remove the ReLU at the bottom of the module and re-order the batch normalization and activation such that they come *before* the convolution (Figure 12.3, *right*).

Now, instead of starting with a convolution, we apply a series of `(BN => RELU => CONV) * 3` (assuming the bottleneck is being used, of course). The output of the residual module is now the *addition operation* which is subsequently fed into the next residual module in the network (since residual modules are stacked on top of each other).

We call this layer ordering *pre-activation* as our ReLUs and batch normalization are placed *before* the convolutions, which is in contrast to the typical approach of applying ReLUs and batch normalizations *after* the convolutions. In our next section, we'll implement ResNet from scratch using both bottlenecks and pre-activations.

12.2 Implementing ResNet

Now that we have reviewed the ResNet architecture, let's go ahead and implement in Keras. For this specific implementation, we'll be using the most recent incarnation of the residual module, including bottlenecks and pre-activations. To update your project structure, create a new file named `resnet.py` inside the `nn.conv` sub-module of `pyimagesearch` – that is where our `ResNet` implementation will live:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- alexnet.py
|    |    |    |--- deepergooglenet.py
|    |    |    |--- lenet.py
|    |    |    |--- minigooglenet.py
|    |    |    |--- minivggnet.py
|    |    |    |--- fcheadnet.py
|    |    |    |--- resnet.py
|    |    |    |--- shallownet.py
|    |--- preprocessing
|    |--- utils
```

From there, open up `resnet.py` and insert the following code:

```
1  # import the necessary packages
2  from keras.layers.normalization import BatchNormalization
3  from keras.layers.convolutional import Conv2D
4  from keras.layers.convolutional import AveragePooling2D
5  from keras.layers.convolutional import MaxPooling2D
6  from keras.layers.convolutional import ZeroPadding2D
7  from keras.layers.core import Activation
8  from keras.layers.core import Dense
9  from keras.layers import Flatten
10 from keras.layers import Input
11 from keras.models import Model
```

```
12 from keras.layers import add
13 from keras.regularizers import l2
14 from keras import backend as K
```

We start off by importing our fairly standard set of classes and functions when building Convolutional Neural Networks. However, I would like to draw your attention to **Line 12** where we import the `add` function. Inside the residual module, we'll need to add together the outputs of two branches, which will be accomplished via this `add` method. We'll also import the `l2` function on **Line 13** so that we can perform L2 weight decay. Regularization is *extremely* important when training ResNet since, due to the network's depth, it is prone to overfitting.

Next, let's move on to our `residual_module`:

```
16 class ResNet:
17     @staticmethod
18     def residual_module(data, K, stride, chanDim, red=False,
19         reg=0.0001, bnEps=2e-5, bnMom=0.9):
```

This specific implementation of ResNet was inspired by both He et al. in their Caffe distribution [46] as well as the mxnet implementation from Wei Wu [47], therefore we will follow their parameter choices as closely as possible. Looking at the `residual_module` we can see that the function accepts more parameters than any of our previous functions – let's review each of them in detail.

The `data` parameter is simply the *input* to the residual module. The value `K` defines the number of filters that will be learned by the *final* `CONV` in the bottleneck. The first two `CONV` layers will learn `K / 4` filters, as per the He et al. paper. The `stride` controls the stride of the convolution. We'll use this parameter to help us reduce the spatial dimensions of our volume *without* resorting to max pooling.

We then have the `chanDim` parameter which defines the axis which will perform batch normalization – this value is specified later in the `build` function based on whether we are using "channels last" or "channels first" ordering.

Not all residual modules will be responsible for reducing the dimensions of our spatial volume – the `red` (i.e., "reduce") boolean will control whether we are reducing spatial dimensions (`True`) or not (`False`).

We can then supply a regularization strength to all `CONV` layers in the residual module via `reg`. The `bnEps` parameter controls the ε responsible for avoiding "division by zero" errors when normalizing inputs. In Keras, ε defaults to 0.001; however, for our particular implementation, we'll allow this value to be reduced significantly. The `bnMom` controls the momentum for the moving average. This value normally defaults to 0.99 inside Keras, but He et al. as well as Wei Wu recommend decreasing the value to 0.9.

Now that the parameters of `residual_module` are defined, let's move on to the body of the function:

```
20         # the shortcut branch of the ResNet module should be
21         # initialize as the input (identity) data
22         shortcut = data
23
24         # the first block of the ResNet module are the 1x1 CONVs
25         bn1 = BatchNormalization(axis=chanDim, epsilon=bnEps,
26             momentum=bnMom)(data)
27         act1 = Activation("relu")(bn1)
```

```
28            conv1 = Conv2D(int(K * 0.25), (1, 1), use_bias=False,
29                 kernel_regularizer=l2(reg))(act1)
```

On **Line 22** we initialize the `shortcut` in the residual module, which is simply a reference to the input `data`. We will later add the `shortcut` to the output of our bottleneck + pre-activation branch.

The first pre-activation of the bottleneck branch can be seen in **Lines 25-29**. Here we apply a batch normalization layer, followed by ReLU activation, and then a 1×1 convolution, using $K/4$ total filters. You'll also notice that we are *excluding* the bias term from our `CONV` layers via `use_bias=False`. Why might we wish to purposely leave out the bias term? According to He et al., the biases are in the `BN` layers that immediately follow the convolutions [48], so there is no need to introduce a second bias term.

Next, we have our second `CONV` layer in the bottleneck, this one responsible for learning a total of $K/4$, 3×3 filters:

```
31            # the second block of the ResNet module are the 3x3 CONVs
32            bn2 = BatchNormalization(axis=chanDim, epsilon=bnEps,
33                 momentum=bnMom)(conv1)
34            act2 = Activation("relu")(bn2)
35            conv2 = Conv2D(int(K * 0.25), (3, 3), strides=stride,
36                 padding="same", use_bias=False,
37                 kernel_regularizer=l2(reg))(act2)
```

The final block in the bottleneck learns K filters, each of which are 1×1:

```
39            # the third block of the ResNet module is another set of 1x1
40            # CONVs
41            bn3 = BatchNormalization(axis=chanDim, epsilon=bnEps,
42                 momentum=bnMom)(conv2)
43            act3 = Activation("relu")(bn3)
44            conv3 = Conv2D(K, (1, 1), use_bias=False,
45                 kernel_regularizer=l2(reg))(act3)
```

For more details on why we call this a "bottleneck" with "pre-activation", please see Section 12.1 above.

The next step is to see if we need to reduce spatial dimensions, thereby alleviating the need to apply max pooling:

```
47            # if we are to reduce the spatial size, apply a CONV layer to
48            # the shortcut
49            if red:
50                 shortcut = Conv2D(K, (1, 1), strides=stride,
51                      use_bias=False, kernel_regularizer=l2(reg))(act1)
```

If we *are* instructed to reduce spatial dimensions, we'll do so with a convolutional layer (applied to the shortcut) with a stride > 1.

The output of the final `conv3` in the bottleneck is the added together with the `shortcut`, thus serving as the output of the `residual_module`:

```
53            # add together the shortcut and the final CONV
54            x = add([conv3, shortcut])
55
56            # return the addition as the output of the ResNet module
57            return x
```

The `residual_module` will serve as our building block when creating deep residual networks. Let's move on to using this building block inside the `build` method:

```
59        @staticmethod
60        def build(width, height, depth, classes, stages, filters,
61              reg=0.0001, bnEps=2e-5, bnMom=0.9, dataset="cifar"):
```

Just as our `residual_module` requires more parameters than previous micro-architecture implementations, the same is true for our `build` function. The `width`, `height`, and `depth` classes all control the input spatial dimensions of the images in our dataset. The `classes` variable dictates how many overall classes our network should learn – these variables you have already seen.

What is interesting are the `stages` and `filters` parameters, both of which are *lists*. When constructing the ResNet architecture, we'll be stacking a number of residual modules on top of each other (using the same number of filters for each stack), followed by reducing the spatial dimensions of the volume – this process is then continued until we are ready to apply our average pooling and softmax classifier.

To make this point clear, let's suppose that `stages=(3, 4, 6)` and `filters=(64, 128, 256, 512)`. The first filter value, 64, will be applied to the only `CONV` layer *not* part of the residual module (i.e., first convolutional layer in the network). We'll then stack *three* residual modules on top of each other – each of these residual modules will learn $K = 128$ filters. The spatial dimensions of the volume will be reduced, and then we'll move on to the second entry in `stages` where we'll stack *four* residual modules on top of each other, each responsible for learning $K = 256$ filters. After these four residual modules, we'll again reduce dimensionality and move on to the final entry in the `stages` list, instructing us to stack *six* residual modules on top of each other, where each residual module will learn $K = 512$.

The benefit of specifying both `stages` and `filters` in a list (rather than hardcoding them) is that we can easily leverage `for` loops to build the very deep network architectures without introducing code bloat – this point will become more clear later in our implementation.

Finally, we have the `dataset` parameter which is assumed to be a string. Depending on the dataset we are building ResNet for, we may want to apply more/less convolutions and batch normalizations *before* we start stacking our residual modules. We'll see why we might want to vary the number of convolutional layers in Section 12.5 below, but for the time being, you can safely ignore this parameter.

Next, let's initialize our `inputShape` and `chanDim` based on whether we are using "channels last" (**Lines 64 and 65**) or "channels first" (**Lines 69-71**) ordering.

```
62            # initialize the input shape to be "channels last" and the
63            # channels dimension itself
64            inputShape = (height, width, depth)
65            chanDim = -1
66
67            # if we are using "channels first", update the input shape
68            # and channels dimension
```

```
69          if K.image_data_format() == "channels_first":
70              inputShape = (depth, height, width)
71              chanDim = 1
```

We are now ready to define the `Input` to our `ResNet` implementation:

```
73          # set the input and apply BN
74          inputs = Input(shape=inputShape)
75          x = BatchNormalization(axis=chanDim, epsilon=bnEps,
76              momentum=bnMom)(inputs)
77
78          # check if we are utilizing the CIFAR dataset
79          if dataset == "cifar":
80              # apply a single CONV layer
81              x = Conv2D(filters[0], (3, 3), use_bias=False,
82                  padding="same", kernel_regularizer=l2(reg))(x)
```

Unlike previous network architectures we have seen in this book (where the first layer is typically a `CONV`), we see that ResNet uses a `BN` as the first layer. The reasoning behind applying batch normalization to your input is an added level of normalization. In fact, performing batch normalization on the input itself can sometimes remove the need to apply mean normalization to the inputs. In either case, the `BN` on **Lines 75 and 76** acts as an added level of normalization.

From there, we apply a single `CONV` layer on **Lines 81 and 82**. This `CONV` layer will learn a total of `filters[0]`, 3×3 filters (keep in mind that `filters` is a *list*, so this value is specified via the `build` method when constructing the architecture).

You'll also notice that I've made a check to see if we are using the CIFAR-10 dataset (**Line 79**). Later in this chapter, we'll be updating this `if` block to include an `elif` statement for Tiny ImageNet. Since the input dimensions to Tiny ImageNet are larger, we'll apply a series of convolutions, batch normalizations, and max pooling (the only max pooling in the ResNet architecture) before we start stacking residual modules. However, for the time being, we are only using the CIFAR-10 dataset.

Let's go ahead and start stacking residual layers on top of each other, the cornerstone of the ResNet architecture:

```
84          # loop over the number of stages
85          for i in range(0, len(stages)):
86              # initialize the stride, then apply a residual module
87              # used to reduce the spatial size of the input volume
88              stride = (1, 1) if i == 0 else (2, 2)
89              x = ResNet.residual_module(x, filters[i + 1], stride,
90                  chanDim, red=True, bnEps=bnEps, bnMom=bnMom)
91
92              # loop over the number of layers in the stage
93              for j in range(0, stages[i] - 1):
94                  # apply a ResNet module
95                  x = ResNet.residual_module(x, filters[i + 1],
96                      (1, 1), chanDim, bnEps=bnEps, bnMom=bnMom)
```

On **Line 85** we start looping over the list of `stages`. Keep in mind that every entry in the `stages` list is an *integer,* indicating how many residual modules will be stacked on top of each other. Following the work of Springenberg et al., ResNet tries to reduce the usage of pooling as much as possible, relying on `CONV` layers to reduce the spatial dimensions of a volume.

To reduce volume size without pooling layers, we must set the `stride` of the convolution on **Line 88**. If this is the *first* entry in the `stage`, we'll set the `stride` to `(1, 1)`, indicating that *no* downsampling should be performed. However, for every *subsequent* stage we'll apply a residual module with a `stride` of `(2, 2)`, which will allow us to decrease the volume size.

From there, we'll loop over the number of layers in the current stage on **Line 93** (i.e., the number of residual modules that will be stacked on top of each other). The number of filters each residual module will learn is controlled by the corresponding entry in the `filters` list. The reason we use `i + 1` as the index into `filters` is because the *first* filter value was used on **Lines 81 and 82**. The rest of the filter values correspond to the number of filters in each stage. Once we have stacked `stages[i]` residual modules on top of each other, our `for` loop brings us back up to **Lines 88-90** where we decrease the spatial dimensions of the volume and repeat the process.

At this point, our volume size has been reduced to `8 x 8 x num_filters` (you can verify this for yourself by computing the input/output volume sizes for each layer, or better yet, simply using the `plot_model` function from Chapter 19 of the *Starter Bundle*).

In order to avoid using dense fully-connected layers, we'll instead apply average pooling to reduce the volume size to `1 x 1 x classes`:

```
98          # apply BN => ACT => POOL
99          x = BatchNormalization(axis=chanDim, epsilon=bnEps,
100             momentum=bnMom)(x)
101         x = Activation("relu")(x)
102         x = AveragePooling2D((8, 8))(x)
```

From there, we create a dense layer for the total number of `classes` we are going to learn, followed by applying a softmax activation to obtain our final output probabilities:

```
104         # softmax classifier
105         x = Flatten()(x)
106         x = Dense(classes, kernel_regularizer=l2(reg))(x)
107         x = Activation("softmax")(x)
108
109         # create the model
110         model = Model(inputs, x, name="resnet")
111
112         # return the constructed network architecture
113         return model
```

The fully constructed ResNet `model` is then returned to the calling function on **Line 113**.

12.3 ResNet on CIFAR-10

Outside of training smaller variants of ResNet on the *full* ImageNet dataset, I had never attempted to train ResNet on CIFAR-10 (or Stanford's Tiny ImageNet challenge, as we'll see in this section). Because of this fact, I have decided to treat this section and the next as candid case studies where I reveal my personal rules of thumb and best practices I have developed over *years* of training neural networks.

These best practices allow me to approach a new problem with an initial plan, iterate on it, and eventually arrive at a solution that obtains good accuracy. In the case of CIFAR-10, we'll be able to replicate the performance of He et al. and claim a spot amongst other state-of-the-art approaches [49].

12.3.1 Training ResNet on CIFAR-10 With the ctrl + c Method

Whenever I start a new set of experiments with either a network architecture I am unfamiliar with, a dataset I have never worked with, or both, I always begin with the `ctrl + c` method of training. Using this method, I can start training with an initial learning rate (and associated set of hyperparameters), monitor training, and quickly adjust the learning rate based on the results as they come in. This method is *especially helpful* when I am totally unsure on the approximate number of epochs it will take for a given architecture to obtain reasonable accuracy or a specific dataset.

In the case of CIFAR-10, I have previous experience (as do you, after reading all the other chapters in this book), so I'm quite confident that it will take 60-100 epochs, but I'm not exactly sure since I've never trained ResNet on the CIFAR-10 before.

Therefore, our first few experiments will rely on the `ctrl + c` method of training to narrow in on what hyperparameters we should be using. Once we are comfortable with our set of hyperparameters, we'll switch over to a *specific* learning rate decay schedule in hopes of milking every last bit of accuracy out of the training process.

To get started, open up a new file, name it `resnet_cifar10.py`, and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from sklearn.preprocessing import LabelBinarizer
7  from pyimagesearch.nn.conv import ResNet
8  from pyimagesearch.callbacks import EpochCheckpoint
9  from pyimagesearch.callbacks import TrainingMonitor
10 from keras.preprocessing.image import ImageDataGenerator
11 from keras.optimizers import SGD
12 from keras.datasets import cifar10
13 from keras.models import load_model
14 import keras.backend as K
15 import numpy as np
16 import argparse
17 import sys
18
19 # set a high recursion limit so Theano doesn't complain
20 sys.setrecursionlimit(5000)
```

We start by importing our required Python packages on **Lines 6-17**. Since we'll be using the `ctrl + c` method to training, we'll make sure to import the `EpochCheckpoint` class (**Line 8**) to serialize ResNet weights to disk during the training process, allowing us to stop and restart training from a specific checkpoint. Since this is our first experiment with ResNet, we'll be using the SGD optimizer (**Line 11**) – time will tell if we decide to switch and use a different optimizer (we'll let our results dictate that).

On **Line 20** I update the recursion limit for the Python programming language. I wrote this book with *both* TensorFlow and Theano in mind, so if you are using TensorFlow, you don't have to worry about this line. However, if you are using Theano, you may encounter an error when instantiating the `ResNet` architecture that a maximum recursion level has been reached. This is a known "bug" with Theano and can be resolved simply by increasing the recursion limit of the Python programming language [50].

Next, let's parse our command line arguments:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-c", "--checkpoints", required=True,
	help="path to output checkpoint directory")
ap.add_argument("-m", "--model", type=str,
	help="path to *specific* model checkpoint to load")
ap.add_argument("-s", "--start-epoch", type=int, default=0,
	help="epoch to restart training at")
args = vars(ap.parse_args())
```

Our script will require only the `--checkpoints` switch, the path to the directory where we will store the ResNet weights every N epochs. In the case that we need to restart training from a particular epoch, we can supply the `--model` path along with an integer indicating the *specific* epoch number.

The next step is to load the CIFAR-10 dataset from disk (pre-split into training and testing), perform mean subtraction, and one-hot encode the integer labels as vectors:

```
# load the training and testing data, converting the images from
# integers to floats
print("[INFO] loading CIFAR-10 data...")
((trainX, trainY), (testX, testY)) = cifar10.load_data()
trainX = trainX.astype("float")
testX = testX.astype("float")

# apply mean subtraction to the data
mean = np.mean(trainX, axis=0)
trainX -= mean
testX -= mean

# convert the labels from integers to vectors
lb = LabelBinarizer()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)
```

While we're at it, let's also initialize an `ImageDataGenerator` so we can apply data augmentation to CIFAR-10:

```
# construct the image generator for data augmentation
aug = ImageDataGenerator(width_shift_range=0.1,
	height_shift_range=0.1, horizontal_flip=True,
	fill_mode="nearest")
```

In the case we are training ResNet from the very first epoch, we need to instantiate the network architecture:

```
# if there is no specific model checkpoint supplied, then initialize
# the network (ResNet-56) and compile the model
if args["model"] is None:
	print("[INFO] compiling model...")
	opt = SGD(lr=1e-1)
```

```
59      model = ResNet.build(32, 32, 3, 10, (9, 9, 9),
60          (64, 64, 128, 256), reg=0.0005)
61      model.compile(loss="categorical_crossentropy", optimizer=opt,
62          metrics=["accuracy"])
```

To start, take a look at the learning rate for our SGD optimizer on **Line 58** – at $1e-1$ this learning rate is by far the largest we have used in this book (by an order of magnitude). The reason we are able to get away with such a high learning rate is due to the identity mappings built into the residual module. Learning rates this high would not (typically) work for networks such as AlexNet, VGG, etc.

We then instantiate our `ResNet` model on **Lines 59 and 60**. Here we can see that the network will accept input images with a width of 32 pixels, height of 32 pixels, and depth of 3 (one for each of the RGB channels in the CIFAR-10 dataset). Since the CIFAR-10 dataset has ten classes, we'll learn ten output labels.

The next parameter we need to supply is `(9, 9, 9)`, or the number of stages in our architecture. This tuple indicates that we will be learning three stages with each stage containing nine residual modules stacked on top of each other. In between each stage, we will apply an additional residual module to decrease the volume size.

The next parameter, `(64, 64, 128, 256)` is the number of filters that the `CONV` layers will learn. The first `CONV` layer (before any residual model is applied) will learn $K = 64$ filters. The remaining entries, 64, 128, and 256 correspond to the number of filters each of the residual module stages will learn. For example, the first nine residual modules will learn $K = 64$ filters. The second set of nine residual modules will learn $K = 128$ filters. And finally, the last set of nine residual modules will learn $K = 256$ filters. The last argument we'll supply to `ResNet` is `reg`, or our L2 regularization strength for weight decay – this value is crucial as it will enable us to prevent overfitting.

In the case we start restarting training from a specific epoch, we need to load the network weights from disk and update the learning rate:

```
64  # otherwise, load the checkpoint from disk
65  else:
66      print("[INFO] loading {}...".format(args["model"]))
67      model = load_model(args["model"])
68
69      # update the learning rate
70      print("[INFO] old learning rate: {}".format(
71          K.get_value(model.optimizer.lr)))
72      K.set_value(model.optimizer.lr, 1e-5)
73      print("[INFO] new learning rate: {}".format(
74          K.get_value(model.optimizer.lr)))
```

Let's also construct a set of `callbacks` so we can both (1) checkpoint ResNet weights every five epochs and (2) monitor training:

```
76  # construct the set of callbacks
77  callbacks = [
78      EpochCheckpoint(args["checkpoints"], every=5,
79          startAt=args["start_epoch"]),
80      TrainingMonitor("output/resnet56_cifar10.png",
81          jsonPath="output/resnet56_cifar10.json",
82          startAt=args["start_epoch"])]
```

Finally, we'll train our network in batch sizes of 128:

```
# train the network
print("[INFO] training network...")
model.fit_generator(
	aug.flow(trainX, trainY, batch_size=128),
	validation_data=(testX, testY),
	steps_per_epoch=len(trainX) // 128, epochs=100,
	callbacks=callbacks, verbose=1)
```

Now that our `resnet_cifar10.py` script is coded up, let's move on to running experiments with it.

ResNet on CIFAR-10: Experiment #1

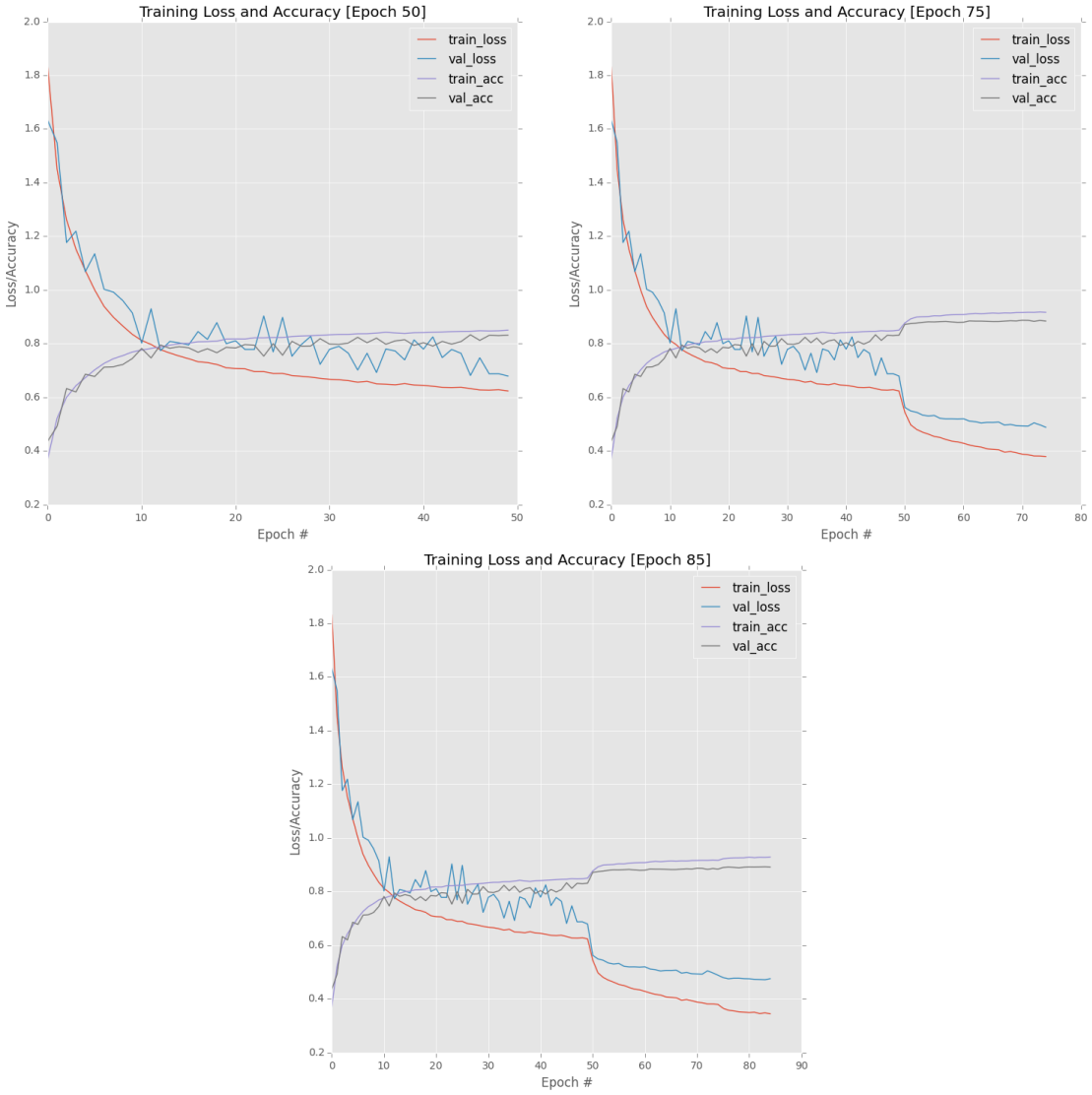


Figure 12.4: **Top-left:** First 50 epochs when training ResNet on CIFAR-10 in Experiment #1. **Top-right:** Next 25 epochs. **Bottom:** Final 10 epochs.

In my very first experiment with CIFAR-10, I was worried about the number of filters in the network, especially regarding overfitting. Because of this concern, my initial filter list consisted of `(16, 16, 32, 64)` along with `(9, 9, 9)` stages of residual modules. I also applied a very small amount of L2 regularization with `reg=0.0001` – I knew regularization would be needed, but I wasn't sure on the correct amount (yet). ResNet was trained using SGD with a base learning rate of $1e-1$ and a momentum term of 0.9.

I started training using the following command:

```
$ python resnet_cifar10.py --checkpoints output/checkpoints
```

Past epoch 50 I noticed training loss starting to slow as well as some volatility in the validation loss (and a growing gap between the two) (Figure 12.4, *top-left*). I stopped training, lowered the learning rate to $1e-2$, and then continued training:

```
$ python resnet_cifar10.py --checkpoints output/checkpoints \
    --model output/checkpoints/epoch_50.hdf5 --start-epoch 50
```

The drop in learning rate proved very effective, stabilizing validation loss, but also overfitting on the training set start to creep in (in inevitability when working with CIFAR-10) around epoch 75 (Figure 12.4, *top-right*). After epoch 75 I once again stopped training, lowered the learning rate to $1e-3$, and allowed ResNet to continue training for another 10 epochs:

```
$ python resnet_cifar10.py --checkpoints output/checkpoints \
    --model output/checkpoints/epoch_75.hdf5 --start-epoch 75
```

The final plot is shown in Figure 12.4 (*bottom*), where we reach 89.06% accuracy on the validation set. For our very first experiment 89.06% is a good start; however, it's not as high as the 90.81% achieved by GoogLeNet in Chapter 11. Furthermore, He et al. reported an accuracy of 93% with ResNet on CIFAR-10, so we clearly have some work to do.

12.3.2 ResNet on CIFAR-10: Experiment #2

Our previous experiment achieved a reasonable accuracy of 89.06% accuracy – but we need higher accuracy. Instead of increasing the depth of the network (by adding more stages), I decided to add more *filters* to each of the `CONV` layers. Thus, my `filters` list was updated to be `(16, 64, 128, 256)`.

Notice how the number of filters in all residual modules have *doubled* from the previous experiment (the number of filters in the first `CONV` layer was left the same). SGD was once again used to train the network with a momentum term of 0.9. I also kept the regularization term at 0.0001.

In Figure 12.5 (*top-left*) you can find a plot of my first 40 epochs: We can clearly see a gap between training loss and validation loss, but overall, validation accuracy is still keeping up with training accuracy. In an effort to improve the accuracy, I decided to lower the learning rate from $1e-1$ to $1e-2$ and train for another five epochs – the result was that learning stagnated entirely (*top-right*). Lowering learning rate again from $1e-2$ to $1e-3$ even caused overfitting through a slight rise in validation loss (*bottom*)

Interestingly, the loss stagnated for *both* the training set *and* the validation set, not unlike previous experiments we've run with GoogLeNet. After the initial drop in learning rate, it appears that our network could not learn any more underlying patterns in the dataset. All that said, after the 50th epoch validation accuracy had increased to **90.10%**, an improvement from our first experiment.

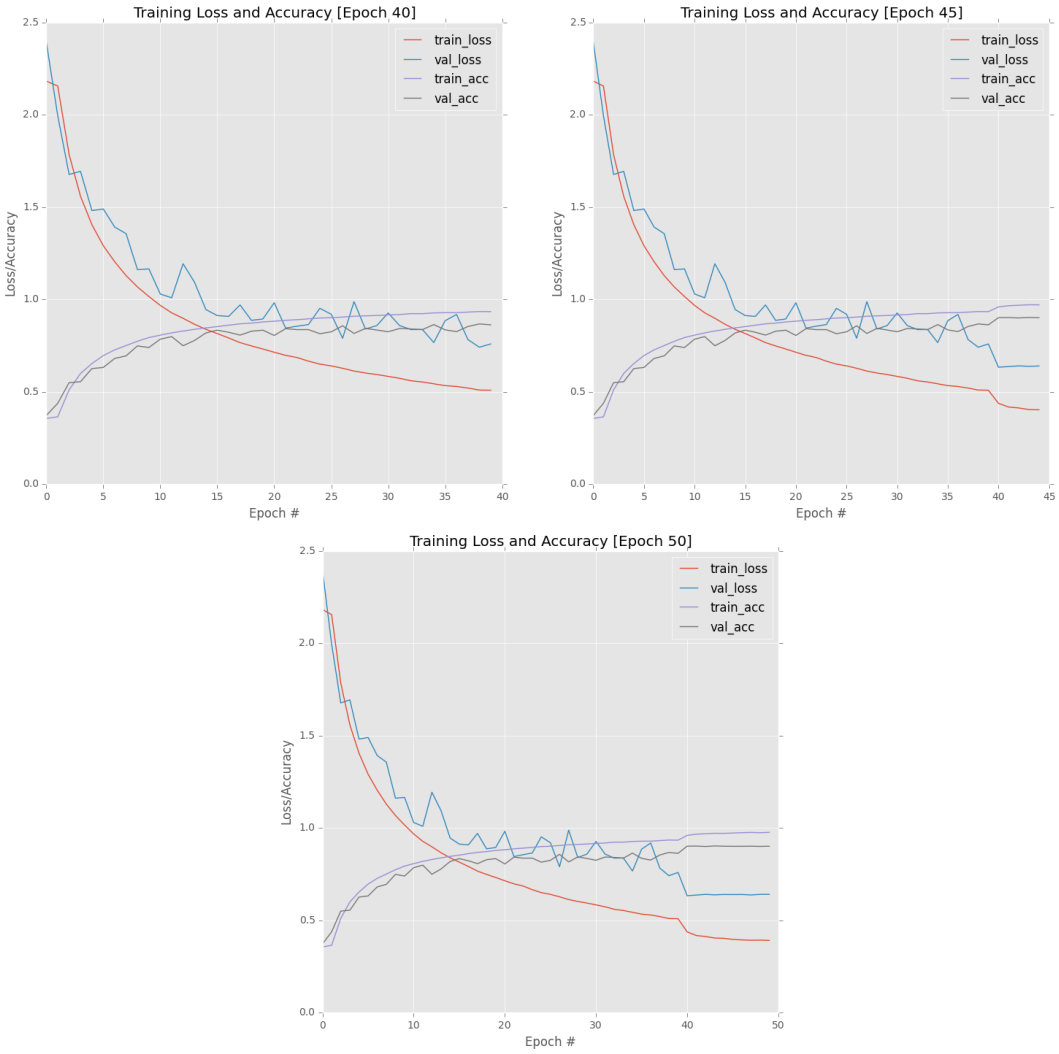


Figure 12.5: **Top-left:** First 40 epochs when training ResNet on CIFAR-10 in Experiment #2. **Top-right:** Next 5 epochs. **Bottom:** Final 5 epochs.

ResNet on CIFAR-10: Experiment #3

At this point, I was starting to become more comfortable training ResNet on CIFAR-10. Clearly the increase of filters helped, but the stagnation in learning after the first learning rate drop was still troubling. I was confident that a slow, linear decrease in learning rate would help combat this problem, but I wasn't convinced that I had obtained a good set of hyperparameters to warrant switching over to learning rate decay.

Instead, I decided to increase the number of filters learned in the *first* `CONV` layer to 64 (up from 16), turning the `filters` list into `(64, 64, 128, 256)`. The increase in filters helped in the second experiment, and there is no reason the first `CONV` layer should miss out on these benefits as well. The SGD optimizer was left alone with an initial learning rate of $1e-1$ and momentum of 0.9.

Furthermore, I also decided to dramatically increase regularization from 0.0001 to 0.0005. I had a suspicion that allowing the network to train for *longer* would result in higher validation accuracy – using a larger regularization term would likely enable me to train for longer.

I was also considering lowering my learning rate, but given that the network was making

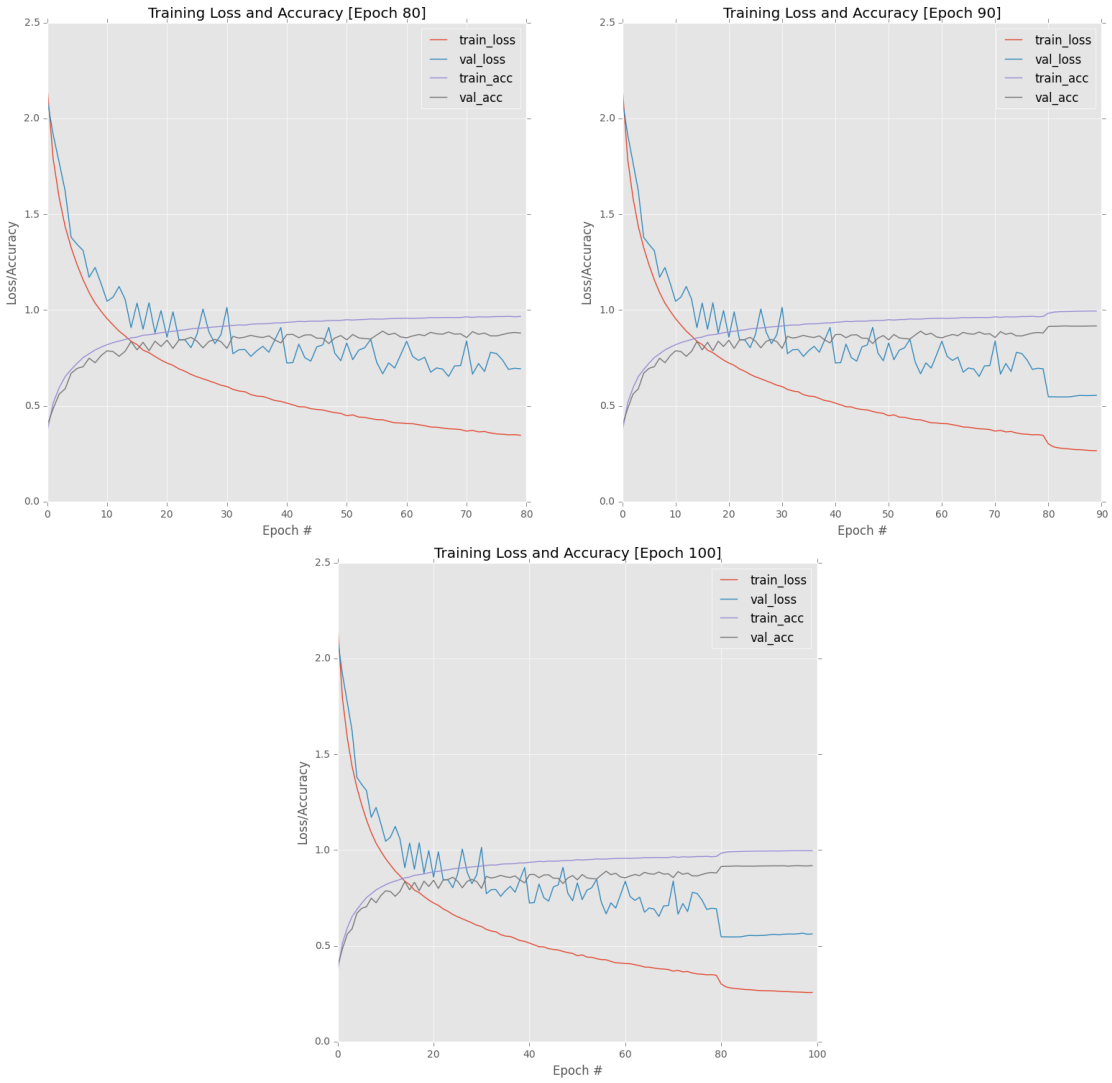


Figure 12.6: **Top-left:** First 80 epochs when training ResNet on CIFAR-10 in Experiment #3. **Top-right:** Next 10 epochs. **Bottom:** Final 10 epochs.

traction without a problem at $1e-1$, it hardly seemed worth it to lower to $1e-2$. Doing so might have stabilized training (i.e., less fluctuation in validation loss/accuracy), but would have ultimately led to lower accuracy after training completed. If my larger learning rate + larger regularization term suspicion turned out to be correct, then it would make sense to switch over to a learning rate decay to avoid stagnation after order of magnitude drops. But before I could make this switch, I first needed to prove my hunch.

As my plot of the first 80 epochs demonstrates (Figure 12.6, *top-left*), there certainly is overfitting as validation quickly diverges from training. However, what's interesting here is that while overfitting is undoubtedly occurring (as training loss drops much faster than validation loss), we are able to train the network for *longer* without validation loss starting to *increase*.

After the 80th epoch, I stopped training, lowered the learning rate to $1e-2$, then trained for another 10 epochs (Figure 12.6, *top-right*). We see an initial drop and loss and increase in accuracy, but from there validation loss/accuracy plateaus. Furthermore, we can start to see the validation loss *increase*, a sure sign of overfitting.

To validate that overfitting was indeed happening, I stopped training at epoch 90, lowered the learning rate to 1e-3, then trained for another 10 epochs (Figure 12.6, *bottom*). Sure enough, this is the telltale sign of overfitting: validation loss *increasing* while training loss decreases/remains constant.

However, what's very interesting is that after the 100th epoch we obtained $\mathbf{91.83\%}$ validation accuracy, higher than our second experiment. The downside is that we are overfit – we need a way to maintain this level of accuracy (and increase it) without overfitting. To do so, I decided to switch from `ctrl + c` training to learning rate decay.

12.4 Training ResNet on CIFAR-10 with Learning Rate Decay

At this point, it seems that we have gotten as far as we can using standard `ctrl + c` training. We've also been able to see that our most successful experiments occur when we can train for longer, in the range of 80-100 epochs. However, there are two major problems we need to overcome:

1. Whenever we drop the learning rate by an order of magnitude and restart training, we obtain a nice bump in accuracy, but then we quickly plateau.
2. We are overfitting.

To solve these problems, and boost accuracy further, a good experiment to try is linearly decreasing the learning rate over a large number of epochs, typically about the same as your *longest* `ctrl + c` experiments (if not slightly longer). To start this, let's open up a new file, name it `resnet_cifar10_decay.py`, and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from sklearn.preprocessing import LabelBinarizer
7  from pyimagesearch.nn.conv import ResNet
8  from pyimagesearch.callbacks import TrainingMonitor
9  from keras.preprocessing.image import ImageDataGenerator
10 from keras.callbacks import LearningRateScheduler
11 from keras.optimizers import SGD
12 from keras.datasets import cifar10
13 import numpy as np
14 import argparse
15 import sys
16 import os
17
18 # set a high recursion limit so Theano doesn't complain
19 sys.setrecursionlimit(5000)
```

Line 2 configures matplotlib so we can save plots in the background. We then import the remainder of our Python packages on **Lines 6-16**. Take a look at **Line 10** where we import our `LearningRateScheduler` so we can define a custom learning rate decay for the training process. We then set a high system recursion limit in order to avoid any issues with the Theano backend (just in case you are using it).

The next step is to define the learning rate decay schedule:

```
21 # define the total number of epochs to train for along with the
22 # initial learning rate
```

```
NUM_EPOCHS = 100
INIT_LR = 1e-1

def poly_decay(epoch):
	# initialize the maximum number of epochs, base learning rate,
	# and power of the polynomial
	maxEpochs = NUM_EPOCHS
	baseLR = INIT_LR
	power = 1.0

	# compute the new learning rate based on polynomial decay
	alpha = baseLR * (1 - (epoch / float(maxEpochs))) ** power

	# return the new learning rate
	return alpha
```

We'll train our network for a total of 100 epochs with a base learning rate of $1e-1$. The `poly_decay` function will decay our $1e-1$ learning rate *linearly* over the course of 100 epochs. This is a *linear* decay due to the fact that we set `power=1`. For more information on learning rate schedules, see Chapter 16 of the *Starter Bundle* along with Chapter 11 of the *Practitioner Bundle*.

We then need to supply two command line arguments:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
	help="path to output model")
ap.add_argument("-o", "--output", required=True,
	help="path to output directory (logs, plots, etc.)")
args = vars(ap.parse_args())
```

The `--model` switch controls the path to our final serialized model after training, while `--output` is the base directory to where we will store any logs, plots, etc.

We can now load the CIFAR-10 dataset and mean normalize it:

```
# load the training and testing data, converting the images from
# integers to floats
print("[INFO] loading CIFAR-10 data...")
((trainX, trainY), (testX, testY)) = cifar10.load_data()
trainX = trainX.astype("float")
testX = testX.astype("float")

# apply mean subtraction to the data
mean = np.mean(trainX, axis=0)
trainX -= mean
testX -= mean
```

Encode the integer labels as vectors:

```
# convert the labels from integers to vectors
lb = LabelBinarizer()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)
```

As well as initialize our `ImageDataGenerator` for data argumentation:

```
# construct the image generator for data augmentation
aug = ImageDataGenerator(width_shift_range=0.1,
	height_shift_range=0.1, horizontal_flip=True,
	fill_mode="nearest")
```

Our `callbacks` list will consist of both a `TrainingMonitor` along with a `LearningRateScheduler` with the `poly_decay` function supplied as the only argument – this class will allow us to decay our learning rate as we train.

```
# construct the set of callbacks
figPath = os.path.sep.join([args["output"], "{}.png".format(
	os.getpid())])
jsonPath = os.path.sep.join([args["output"], "{}.json".format(
	os.getpid())])
callbacks = [TrainingMonitor(figPath, jsonPath=jsonPath),
	LearningRateScheduler(poly_decay)]
```

We'll then instantiate `ResNet` with the best parameters we found from Section 12.6 (three stacks of nine residual modules: 64 filters in the first `CONV` layer before the residual modules, and 64, 128, and 256 filters for each stack of respective residual modules):

```
# initialize the optimizer and model (ResNet-56)
print("[INFO] compiling model...")
opt = SGD(lr=INIT_LR, momentum=0.9)
model = ResNet.build(32, 32, 3, 10, (9, 9, 9),
	(64, 64, 128, 256), reg=0.0005)
model.compile(loss="categorical_crossentropy", optimizer=opt,
	metrics=["accuracy"])
```

We'll then train our network using learning rate decay:

```
# train the network
print("[INFO] training network...")
model.fit_generator(
	aug.flow(trainX, trainY, batch_size=128),
	validation_data=(testX, testY),
	steps_per_epoch=len(trainX) // 128, epochs=NUM_EPOCHS,
	callbacks=callbacks, verbose=1)

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

The big question is – will our learning rate decay pay off? To find out, proceed to the next section.

ResNet on CIFAR-10: Experiment #4

As the code in the previous section indicates, we are going to use the SGD optimizer with a base learning rate of $1e-1$ and a momentum term of 0.9. We'll train ResNet for a total of 100 epochs, linearly decreasing the linear rate from $1e-1$ down to zero. To train ResNet on CIFAR-10 with learning rate decay, I executed the following command:

```
$ python resnet_cifar10_decay.py --output output \
    --model output/resnet_cifar10.hdf5
```

Figure 12.7: Training ResNet on CIFAR-10 using learning rate decay beats out all previous experiments.

After training was complete, I took a look at the plot (Figure 12.7). As in previous experiments, training and validation loss start to diverge early on, but more importantly, *the gap remains approximately constant* after the initial divergence. This result is important as it indicates that our overfitting is *controlled*. We have to accept that we *will* overfit when training on CIFAR-10, but we need to *control* this overfitting. By applying learning rate decay, we were able to successfully do so.

The question is, *did we obtain higher classification accuracy?* To answer that, take a look at the output of the last few epochs:

```
...
Epoch 98/100
247s - loss: 0.1563 - acc: 0.9985 - val_loss: 0.3987 - val_acc: 0.9351
Epoch 99/100
245s - loss: 0.1548 - acc: 0.9987 - val_loss: 0.3973 - val_acc: 0.9358
Epoch 100/100
```

```
244s - loss: 0.1538 - acc: 0.9990 - val_loss: 0.3978 - val_acc: 0.9358
[INFO] serializing network...
```

After the 100th epoch, ResNet is reaching **93.58%** accuracy on our testing set. This result is *substantially* higher than our previous two experiments, and more importantly, it has allowed us to replicate the results from He et al. when training ResNet on CIFAR-10.

Taking a look at the CIFAR-10 leaderboard [49], we see that He et al. reached 93.57% accuracy [24], near identical to our result (Figure 12.8). The red arrow indicates our accuracy, safely landing us in the top-10 leaderboard.

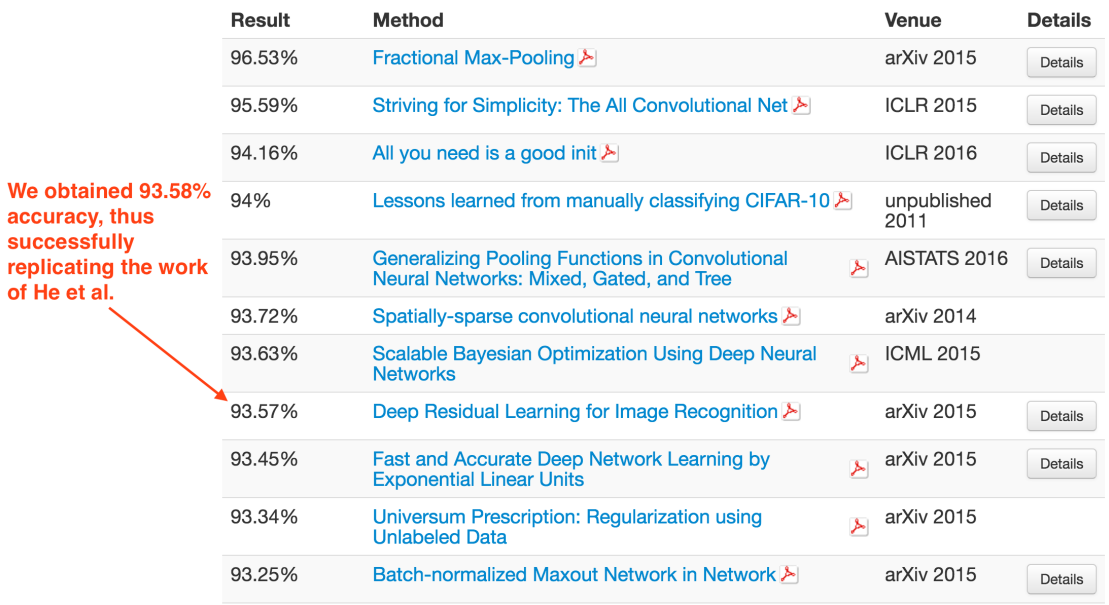


Figure 12.8: We have successfully replicated the work of He et al. when applying ResNet to CIFAR-10 down to 0.01%.

12.5 ResNet on Tiny ImageNet

In this section, we will train the ResNet architecture (with bottleneck and pre-activation) on Stanford's cs231n Tiny ImageNet challenge. Similar to the ResNet + CIFAR-10 experiments earlier in this chapter, I have never trained ResNet on Tiny ImageNet before, so I'm going to apply my same exact experiment process:

1. Start with `ctrl + c`-based training to obtain a baseline.
2. If stagnation/plateauing occurs after order of magnitude learning rate drops, then switch over to learning rate decay.

Given that we've already applied a similar technique to CIFAR-10, we should be able to save ourselves some time noticing signs of overfitting and plateauing earlier. That said, let's get started by reviewing directory structure for this project, which is near identical to the GoogLeNet and Tiny ImageNet challenge from the previous chapter:

```
|--- resnet_tinyimagenet.py
|    |--- config
|    |    |--- __init__.py
```

```
|    |    |--- tiny_imagenet_config.py
|    |--- rank_accuracy.py
|    |--- train.py
|    |--- train_decay.py
|    |--- output/
|    |    |--- checkpoints/
|    |    |--- tiny-image-net-200-mean.json
```

Here you can see we have created `config` Python module where we have stored a file named `tiny_imagenet_config.py`.

This file was copied directly from the GoogLeNet chapter. I then updated **Lines 31-39** to point to the output ResNet `MODEL_PATH`, `FIG_PATH` (learning plot), and `JSON_PATH` (serialized log of training history):

```
31 # define the path to the output directory used for storing plots,
32 # classification reports, etc.
33 OUTPUT_PATH = "output"
34 MODEL_PATH = path.sep.join([OUTPUT_PATH,
35      "resnet_tinyimagenet.hdf5"])
36 FIG_PATH = path.sep.join([OUTPUT_PATH,
37      "resnet56_tinyimagenet.png"])
38 JSON_PATH = path.sep.join([OUTPUT_PATH,
39      "resnet56_tinyimagenet.json"])
```

From there we have `train.py` which will be responsible for training ResNet using the standard `ctrl + c` method. In the case that we wish to apply learning rate decay, we'll be able to use `train_decay.py`. Finally, `rank_accuracy.py` will be used to compute the rank-1 and rank-5 accuracy of ResNet on Tiny ImageNet.

Additionally, do not forget to copy your `tiny-image-net-200-mean.json` file from Chapter 11 to the `output` directory for the ResNet project. You will need the mean file when performing mean subtraction during training and evaluation.

12.5.1 Updating the ResNet Architecture

Earlier in this chapter we reviewed our implementation of the `ResNet` architecture in detail. Specifically, we noted the `dataset` parameter supplied to the `build` method, like so:

```
59      @staticmethod
60      def build(width, height, depth, classes, stages, filters,
61          reg=0.0001, bnEps=2e-5, bnMom=0.9, dataset="cifar"):
```

This value defaulted to `cifar`; however, since we are now working with Tiny ImageNet, we need to update our `ResNet` implementation to include an `if/elif` block. Go ahead and open up your `resnet.py` file in the `nn.conv` sub-module of `PyImageSearch` and insert the following code:

```
78          # check if we are utilizing the CIFAR dataset
79          if dataset == "cifar":
80              # apply a single CONV layer
81              x = Conv2D(filters[0], (3, 3), use_bias=False,
82                  padding="same", kernel_regularizer=l2(reg))(x)
```

```
83
84          # check to see if we are using the Tiny ImageNet dataset
85          elif dataset == "tiny_imagenet":
86               # apply CONV => BN => ACT => POOL to reduce spatial size
87               x = Conv2D(filters[0], (5, 5), use_bias=False,
88                    padding="same", kernel_regularizer=l2(reg))(x)
89               x = BatchNormalization(axis=chanDim, epsilon=bnEps,
90                    momentum=bnMom)(x)
91               x = Activation("relu")(x)
92               x = ZeroPadding2D((1, 1))(x)
93               x = MaxPooling2D((3, 3), strides=(2, 2))(x)
```

Lines 79-82 we have already reviewed before – these lines are where we apply a single 3×3 `CONV` layer for CIFAR-10. However, we are now updating the architecture to check for Tiny ImageNet (**Line 85**).

Provided we are instantiating ResNet for Tiny ImageNet, we need to add in some additional layers. To start, we apply 5×5 `CONV` layer to learn larger feature maps (in the *full* ImageNet dataset implementation, we'll actually be learning 7×7 filters).

Next, we apply a batch normalization followed a ReLU activation. Max pooling, the *only* max pooling layer in the ResNet architecture, is applied on **Line 93** using a size of 3×3 and stride of 2×2. Combined with the previous zero padding layer (**Line 92**), pooling ensures that our output spatial volume size is 32×32, the exact same spatial dimensions as the input images from CIFAR-10. Validating that the output volume size is 32×32 ensures we can easily reuse the rest of the ResNet implementation without having to make any additional changes.

12.5.2 Training ResNet on Tiny ImageNet With the ctrl + c Method

Now that our ResNet implementation has been updated, let's code up a Python script responsible for the actual training process. Open up a new file, name it `train.py`, and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from config import tiny_imagenet_config as config
7  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
8  from pyimagesearch.preprocessing import SimplePreprocessor
9  from pyimagesearch.preprocessing import MeanPreprocessor
10 from pyimagesearch.callbacks import EpochCheckpoint
11 from pyimagesearch.callbacks import TrainingMonitor
12 from pyimagesearch.io import HDF5DatasetGenerator
13 from pyimagesearch.nn.conv import ResNet
14 from keras.preprocessing.image import ImageDataGenerator
15 from keras.optimizers import SGD
16 from keras.models import load_model
17 import keras.backend as K
18 import argparse
19 import json
20 import sys
21
22 # set a high recursion limit so Theano doesn't complain
23 sys.setrecursionlimit(5000)
```

Lines 2 and 3 configure matplotlib so we can save our figures and plots to disk during the training process. We then import the remainder of our Python packages on **Lines 6-20**. We have seen all of these imports before from Chapter 11 on GoogLeNet + Tiny ImageNet, only now we are importing ResNet (**Line 13**) rather than GoogLeNet. We'll also update the maximum recursion limit just in case you are using the Theano backend.

Next comes our command line arguments:

```
25 # construct the argument parse and parse the arguments
26 ap = argparse.ArgumentParser()
27 ap.add_argument("-c", "--checkpoints", required=True,
28     help="path to output checkpoint directory")
29 ap.add_argument("-m", "--model", type=str,
30     help="path to *specific* model checkpoint to load")
31 ap.add_argument("-s", "--start-epoch", type=int, default=0,
32     help="epoch to restart training at")
33 args = vars(ap.parse_args())
```

These command line switches can be used to start training from scratch *or* restart training from a specific epoch. For a more detailed review of each command line argument, please see Section 12.3.1.

Let's also initialize our `ImageDataGenerator` that will be applied to the training set for data augmentation:

```
35 # construct the training image generator for data augmentation
36 aug = ImageDataGenerator(rotation_range=18, zoom_range=0.15,
37     width_shift_range=0.2, height_shift_range=0.2, shear_range=0.15,
38     horizontal_flip=True, fill_mode="nearest")
39
40 # load the RGB means for the training set
41 means = json.loads(open(config.DATASET_MEAN).read())
```

Line 41 then loads our RGB means computed over the training set of mean subtraction.

Our image pre-processors are fairly standard here, consisting of ensuring the image is resized to 64×64 pixels, performing mean normalization, and then converting the image to a Keras-compatible array:

```
43 # initialize the image preprocessors
44 sp = SimplePreprocessor(64, 64)
45 mp = MeanPreprocessor(means["R"], means["G"], means["B"])
46 iap = ImageToArrayPreprocessor()
47
48 # initialize the training and validation dataset generators
49 trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, 64, aug=aug,
50     preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
51 valGen = HDF5DatasetGenerator(config.VAL_HDF5, 64,
52     preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
```

Lines 49-52 construct an `HDF5DatasetGenerator`, supplying the image pre-processors and data augmentation classes. Batches of 64 images will be polled at a time for these generators and passed through the network.

If we are training ResNet from the very first epoch, we need to instantiate the model and compile it:

```
54 # if there is no specific model checkpoint supplied, then initialize
55 # the network and compile the model
56 if args["model"] is None:
57     print("[INFO] compiling model...")
58     model = ResNet.build(64, 64, 3, config.NUM_CLASSES, (3, 4, 6),
59         (64, 128, 256, 512), reg=0.0005, dataset="tiny_imagenet")
60     opt = SGD(lr=1e-1, momentum=0.9)
61     model.compile(loss="categorical_crossentropy", optimizer=opt,
62         metrics=["accuracy"])
```

Lines 58 and 59 initialize the `ResNet` model itself. The `model` will accept input images with $64 \times 64 \times 3$ spatial dimensions and will learn a total of `NUM_CLASSES`, which in the case of Tiny ImageNet is 200. My choice for `stages=(3, 4, 6)` was inspired by the *Deep Residual Learning for Image Recognition* [24] paper where a similar version of residual module layer stacking was used for the full ImageNet dataset.

Given that Tiny ImageNet will require more discriminative filters than CIFAR-10, I also updated the `filters` list to learn more filters for each of the `CONV` layers: `(64, 128, 256, 512)`. This list of filters implies that the *first* `CONV` layer (before any of the residual modules) will learn a total of 64 5×5 filters. From there, three residual modules will be stacked on top of each other, each responsible for learning $K = 128$ filters. Dimensionality is then reduced, then four residual modules are stacked, this time learning $K = 256$ filters. Once again the spatial dimensions of the volume are reduced, then six residual modules are stacked, each module learning $K = 512$ filters.

We'll also apply a regularization strength of 0.0005 as regularization seemed to aide us in training ResNet on CIFAR-10. To train the network, SGD will be used with a base learning rate of $1e-1$ and a momentum term of 0.9.

If we have stopped training, updated any hyperparameters (such as the learning rate), and wish to restart training, the following code block will handle that process for us:

```
64 # otherwise, load the checkpoint from disk
65 else:
66     print("[INFO] loading {}...".format(args["model"]))
67     model = load_model(args["model"])
68
69     # update the learning rate
70     print("[INFO] old learning rate: {}".format(
71         K.get_value(model.optimizer.lr)))
72     K.set_value(model.optimizer.lr, 1e-5)
73     print("[INFO] new learning rate: {}".format(
74         K.get_value(model.optimizer.lr)))
```

Our `callbacks` list will consist of checkpointing ResNet weights to disk every five epochs, followed by plotting the training history:

```
76 # construct the set of callbacks
77 callbacks = [
78     EpochCheckpoint(args["checkpoints"], every=5,
79         startAt=args["start_epoch"]),
```

```
80      TrainingMonitor(config.FIG_PATH, jsonPath=config.JSON_PATH,
81          startAt=args["start_epoch"])]
```

Finally, we'll kick off the training process and train our network using mini-batches of size 64:

```
83  # train the network
84  model.fit_generator(
85      trainGen.generator(),
86      steps_per_epoch=trainGen.numImages // 64,
87      validation_data=valGen.generator(),
88      validation_steps=valGen.numImages // 64,
89      epochs=50,
90      max_queue_size=10,
91      callbacks=callbacks, verbose=1)
92
93  # close the databases
94  trainGen.close()
95  valGen.close()
```

The exact number of epochs we'll need to train ResNet is unknown at this point, so we'll set `epochs` to a large number and adjust as needed.

ResNet on Tiny ImageNet: Experiment #1

To start training, I executed the following command:

```
$ python train.py --checkpoints output/checkpoints
```

After monitoring training for the first 25 epochs, it became clear that training loss was starting to stagnate a bit (Figure 12.9, *top-left*). To combat this stagnation, I stopped training, reduced my learning rate from $1e-1$ to $1e-2$, and resumed training:

```
$ python train.py --checkpoints output/checkpoints \
    --model output/checkpoints/epoch_25.hdf5 --start-epoch 25
```

Training continued from epochs 25-35 at this lower learning rate. We can immediately see the benefit of lowering the learning rate by an order of magnitude – loss drops dramatically, and accuracy enjoys a nice bump. (Figure 12.9, *top-right*).

However, after this initial bump, the training loss continued to drop at a much faster rate than the validation loss. I once again stopped training at epoch 35, lowered the learning rate from $1e-2$ to $1e-3$, and resumed training:

```
$ python train.py --checkpoints output/checkpoints \
    --model output/checkpoints/epoch_35.hdf5 --start-epoch 35
```

I only allowed ResNet to train for another 5 epochs as I started to notice clear signs of overfitting (Figure 12.9, *bottom*). Loss dips slightly upon the $1e-3$ change, then starts to *increase*, all the while training loss decreases at a faster rate – this is a telltale sign of overfitting. I stopped training altogether at this point, and noted the validation accuracy to be 53.14%.

To determine the accuracy on the testing set, I executed the following command (keeping in mind that the `rank_accuracy.py` script is *identical* to the one from Chapter 11 on GoogLeNet:

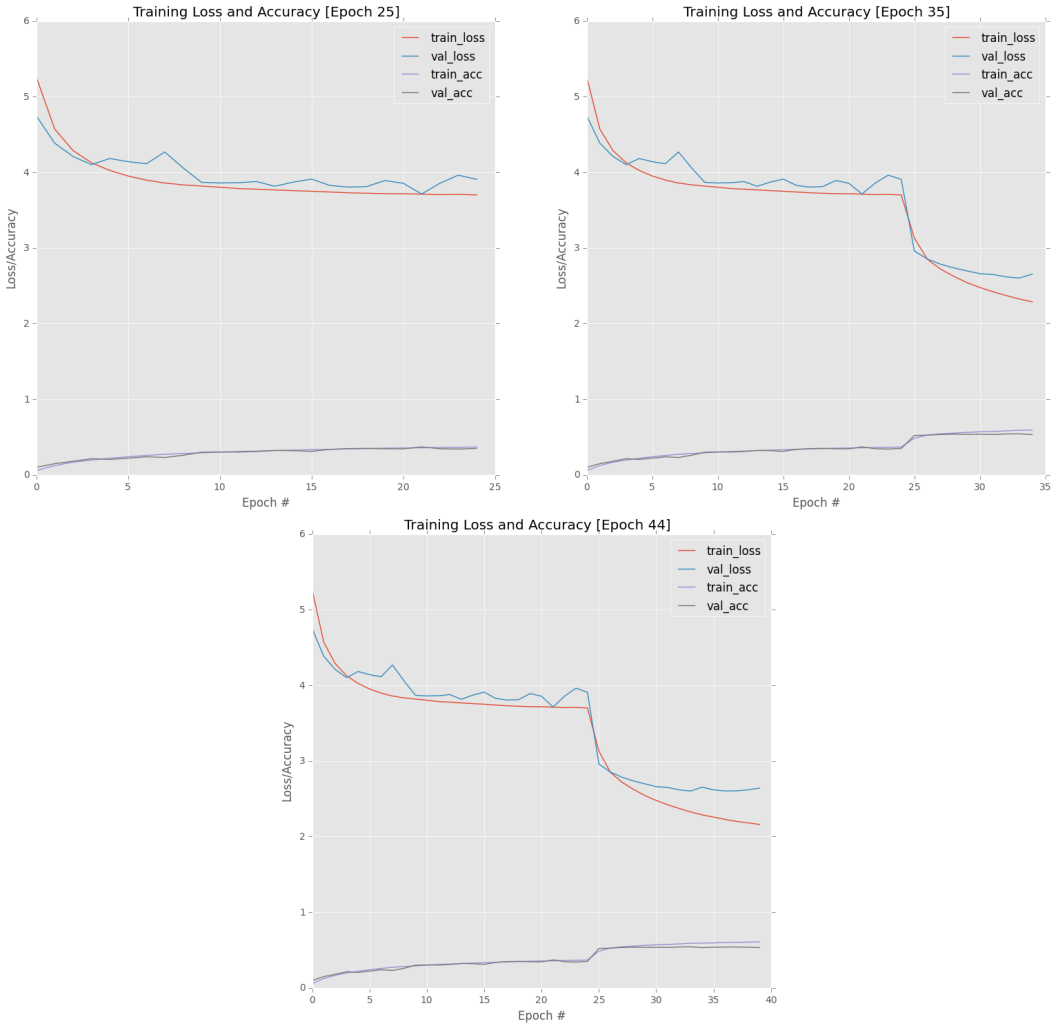


Figure 12.9: **Top-left:** First 25 epochs when training ResNet on Tiny ImageNet in Experiment #1. **Top-right:** Next 10 epochs. **Bottom:** Final 5 epochs. Notice the telltale sign of overfitting as validation loss starts to increase during the final epochs.

```
$ python rank_accuracy.py
[INFO] loading model...
[INFO] predicting on test data...
[INFO] rank-1: 53.10%
[INFO] rank-5: 75.43%
```

As the output demonstrates, we are obtaining 53.10% rank-1 accuracy on the testing set. This first experiment was not a bad one as we are already closing in on the GoogLeNet + Tiny ImageNet accuracy. Given the success of applying learning rate decay to Tiny ImageNet from the previous chapter, I immediately decided to apply the same process to ResNet.

12.5.3 Training ResNet on Tiny ImageNet with Learning Rate Decay

To train ResNet using learning rate decay, open up a new file, name it `train_decay.py`, and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from config import tiny_imagenet_config as config
7  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
8  from pyimagesearch.preprocessing import SimplePreprocessor
9  from pyimagesearch.preprocessing import MeanPreprocessor
10 from pyimagesearch.callbacks import TrainingMonitor
11 from pyimagesearch.io import HDF5DatasetGenerator
12 from pyimagesearch.nn.conv import ResNet
13 from keras.preprocessing.image import ImageDataGenerator
14 from keras.callbacks import LearningRateScheduler
15 from keras.optimizers import SGD
16 import argparse
17 import json
18 import sys
19 import os
20
21 # set a high recursion limit so Theano doesn't complain
22 sys.setrecursionlimit(5000)
```

Lines 6-19 import our required Python packages. We'll need to import the `LearningRateScheduler` class on **Line 14** so we can apply a learning rate schedule to the training process. The system recursion limit is then set on **Line 22** just in case you are using the Theano backend.

Next, we define the actual function responsible for applying the learning rate decay:

```
24 # define the total number of epochs to train for along with the
25 # initial learning rate
26 NUM_EPOCHS = 75
27 INIT_LR = 1e-1
28
29 def poly_decay(epoch):
30     # initialize the maximum number of epochs, base learning rate,
31     # and power of the polynomial
32     maxEpochs = NUM_EPOCHS
33     baseLR = INIT_LR
34     power = 1.0
35
36     # compute the new learning rate based on polynomial decay
37     alpha = baseLR * (1 - (epoch / float(maxEpochs))) ** power
38
39     # return the new learning rate
40     return alpha
```

Here we indicate that we'll train for a maximum of 75 epochs (**Line 26**) with a base learning rate of $1e-1$ (**Line 27**). The `poly_decay` function is defined on **Line 29** which accepts a single parameter, the current `epoch` number. We set `power=1.0` on **Line 34** to turn the polynomial decay into a *linear* one (see Chapter 11 for more details). The new learning rate (based on the current epoch) is computed on **Line 37** and then returned to the calling function on **Line 40**.

Let's move on to the command line arguments:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
	help="path to output model")
ap.add_argument("-o", "--output", required=True,
	help="path to output directory (logs, plots, etc.)")
args = vars(ap.parse_args())
```

Here we simply need to provide a path to our output serialized `--model` after training is complete along with an `--output` path to store any plots/logs.

We can now initialize our data augmentation class and load the RGB means from disk:

```
# construct the training image generator for data augmentation
aug = ImageDataGenerator(rotation_range=18, zoom_range=0.15,
	width_shift_range=0.2, height_shift_range=0.2, shear_range=0.15,
	horizontal_flip=True, fill_mode="nearest")

# load the RGB means for the training set
means = json.loads(open(config.DATASET_MEAN).read())
```

As well as create our training and validation `HDF5DatasetGenerators`:

```
# initialize the image preprocessors
sp = SimplePreprocessor(64, 64)
mp = MeanPreprocessor(means["R"], means["G"], means["B"])
iap = ImageToArrayPreprocessor()

# initialize the training and validation dataset generators
trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, 64, aug=aug,
	preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
valGen = HDF5DatasetGenerator(config.VAL_HDF5, 64,
	preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
```

Our `callbacks` list will consist of a `TrainingMonitor` and a `LearningRateScheduler`:

```
# construct the set of callbacks
figPath = os.path.sep.join([args["output"], "{}.png".format(
	os.getpid())])
jsonPath = os.path.sep.join([args["output"], "{}.json".format(
	os.getpid())])
callbacks = [TrainingMonitor(figPath, jsonPath=jsonPath),
	LearningRateScheduler(poly_decay)]
```

Below we instantiate our `ResNet` architecture and SGD optimizer using the same parameters as our first experiment:

```
# initialize the optimizer and model (ResNet-56)
print("[INFO] compiling model...")
model = ResNet.build(64, 64, 3, config.NUM_CLASSES, (3, 4, 6),
	(64, 128, 256, 512), reg=0.0005, dataset="tiny_imagenet")
```

```
opt = SGD(lr=INIT_LR, momentum=0.9)
model.compile(loss="categorical_crossentropy", optimizer=opt,
    metrics=["accuracy"])
```

Finally, we can train our network in mini-batches of 64:

```
# train the network
print("[INFO] training network...")
model.fit_generator(
    trainGen.generator(),
    steps_per_epoch=trainGen.numImages // 64,
    validation_data=valGen.generator(),
    validation_steps=valGen.numImages // 64,
    epochs=NUM_EPOCHS,
    max_queue_size=10,
    callbacks=callbacks, verbose=1)
```

The number of epochs we are going to train for is controlled by `NUM_EPOCHS` defined earlier in this script. We'll be linearly decreasing our learning rate from 1e-1 down to zero over the course of `NUM_EPOCHS`. And finally serialize the model to disk once training is complete:

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

# close the databases
trainGen.close()
valGen.close()
```

ResNet on Tiny ImageNet: Experiment #2

In this experiment, I trained the ResNet architecture detailed above on the Tiny ImageNet dataset using the SGD optimizer, a base learning rate of $1e-1$, and a momentum term of 0.9. The learning rate was decayed linearly over the course of 75 epochs. To perform this experiment, I executed the following command:

```
$ python train_decay.py --model output/resnet_tinyimagenet_decay.hdf5 \
    --output output
```

The resulting learning plot can be found in Figure 12.11. Here we can see the dramatic effect that learning rate decay can have on a training process. While both training loss and validation loss diverge from each other, it's not until epoch 60 where the gap widens past previous epochs. Furthermore, our validation loss continues to *decrease* as well. At the end of the 75th epoch, I obtained 58.32% rank-1 accuracy.

I then evaluated the network on the testing set using the following command:

```
$ python rank_accuracy.py
[INFO] loading model...
[INFO] predicting on test data...
[INFO] rank-1: 58.03%
[INFO] rank-5: 80.46%
```

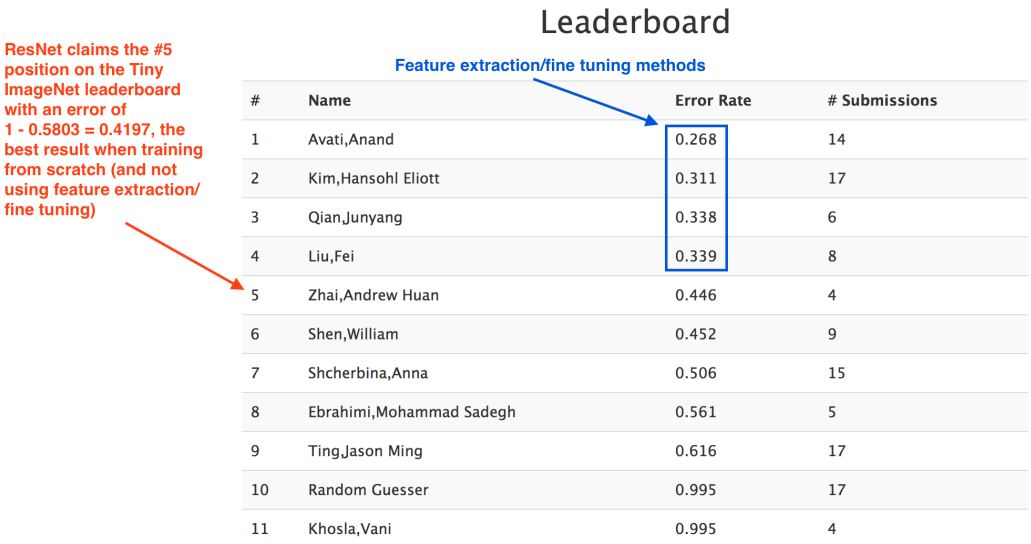


Figure 12.10: Using ResNet we are able to reach the #5 position on the Tiny ImageNet leaderboard, beating out all other approaches that attempted to train a network from scratch. All results with lower error than our approach applied fine-tuning/feature extraction.

As the results demonstrated, we have reached $\mathbf{58.03}\%$ rank-1 and $\mathbf{80.46}\%$ rank-5 accuracy on the testing set, a *substantial* improvement over our previous chapter on GoogLeNet. This result leads to a test error of $1 - 0.5803 = 0.4197$, which *easily* takes position #5 on the Tiny ImageNet leaderboard (Figure 12.10).

This is the *best accuracy* obtained on the Tiny ImageNet dataset that *does not* perform some form of transfer learning or fine-tuning. Given that the accuracies for positions 1-4 were obtained from fine-tuning a network that was *already* trained on the fully ImageNet dataset, it's hard to compare a fine-tuned network to one that was trained entirely from scratch. If we (fairly) compare our network that was trained from scratch to the other networks trained from scratch on the leaderboard, we can see that our ResNet has obtained the highest accuracy amongst the group.

12.6 Summary

In this chapter, we discussed the ResNet architecture in detail, including the residual module micro-architecture. The original residual module proposed by He et al. in their 2015 paper, *Deep Residual Learning for Image Recognition* [24] has gone through many revisions. Originally the module consisted of two `CONV` layers and an identity mapping "shortcut". In the same paper, it was found that adding the "bottleneck" sequence of 1×1, 3×3, and 1×1 `CONV` layers improved accuracy.

Then, in their 2016 study, *Identity Mappings in Deep Residual Networks* [33], the *pre-activation* residual module was introduced. We call this update "pre-activation" because we apply the activation and batch normalization *before* the convolution, going against the "conventional wisdom" when building Convolutional Neural Networks.

From there, we implemented the ResNet architecture using both bottleneck and pre-activation using the Keras framework. This implementation was then used to train ResNet on both the CIFAR-10 and Tiny ImageNet datasets. In CIFAR-10, we were able to replicate the results of He et al., obtaining $\mathbf{93.58}\%$ accuracy. Then, on Tiny ImageNet, reached $\mathbf{58.03}\%$ accuracy, the highest accuracy thus far from a network *trained from scratch* on Stanford's cs231n Tiny ImageNet

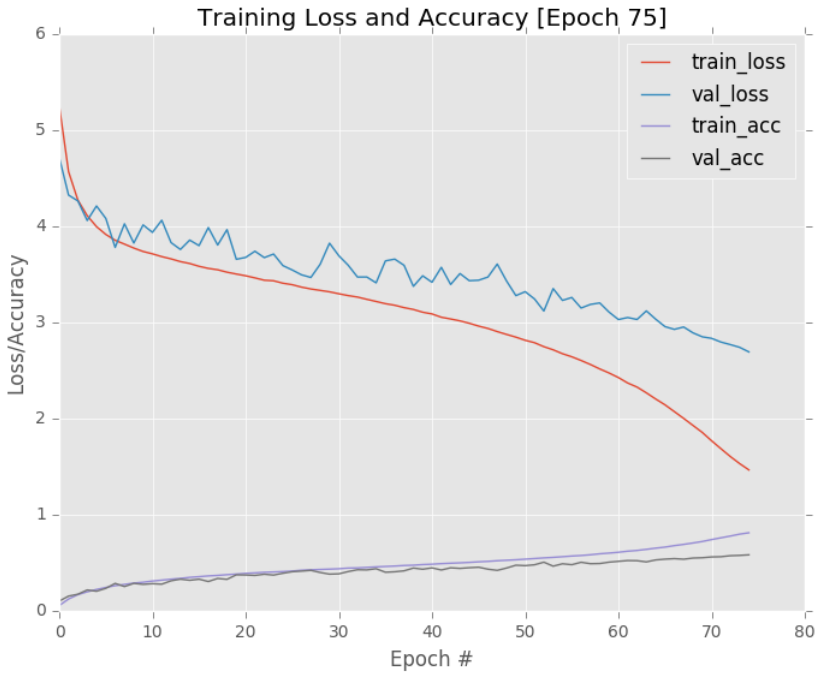


Figure 12.11: Applying learning rate to ResNet when trained on Tiny ImageNet leads to 58.32% validation and 58.03% testing accuracy – significantly higher than our previous experiment using `ctrl + c` training.

challenge.

Later in the *ImageNet Bundle*, we'll investigate ResNet and train it on the *complete* ImageNet dataset, once again replicating the work of He et al.

13. Fundamentals of Object Detection

Before we can train Faster R-CNNs [51] and Single Shot Detectors (SSDs) [52] in the *ImageNet Bundle*, we need to understand the fundamentals of object detection.

Object detection is hardly a new area of study in computer vision and machine learning — in fact, one can argue that object detection is a *core component* of image understanding and dates back to the inception of the computer vision field itself. Prior to the latest incarnation of neural networks and deep learning, popular object detection methods utilized three crucial components:

1. Sliding windows
2. Image pyramids
3. Non-maxima suppression

In the rest of this chapter we'll be discussing each of these components in detail, implementing them by hand, and then using this framework to utilize a CNN to not only *recognize* objects in an image but *localize* where the object is in an image.

This review of object detection fundamentals is critical for you to understand the history of object detection and how methods such as Faster R-CNN and SSDs build from previous object detection work (for a detailed discussion of Faster R-CNN and SSDs, including how to train them on your own custom datasets, please see their corresponding chapters in the *ImageNet Bundle*).

13.1 A Brief Object Detection History Lesson

While there are many object detection methods in the computer vision literature, two stand out amongst the others:

1. Haar cascades
2. HOG + Linear SVM

In this section we'll briefly discuss both of them. This is *not* meant to be a thorough treatment of the history of object detection nor is it meant to be an in-depth dive into Haar cascades and Linear SVM. Instead, this section is meant to give you historical context on seminal object detection methods in computer vision and machine learning history.

Later in this chapter, we'll apply some of the components from these classic object detection algorithms to deep learning and devise our own deep learning object detector.

13.1.1 Haar Cascades and HOG + Linear SVM

One of the most famous object detectors, *Rapid Object Detection using a Boosted Cascade of Simple Features*, by Viola and Jones in their 2001 CVPR paper [53], go by another name you may be more familiar with — Haar cascades.

Pre-trained Haar cascades are distributed with the OpenCV library and are arguably the most used models for face detection (although their accuracy leaves much to be desired).

While Haar cascades are fast, they:

1. Tend to have a high false-positive detection rate
2. Can miss objects entirely based on the parameters supplied to the cascade — some parameter choices will work better than others but, unfortunately, the parameters may need to be tuned on an image-to-image basis

The Viola-Jones paper was published in *International Conference on Computer Vision* as well. You may see references to both papers.

In 2005, Dalal and Triggs published their seminal *Histogram of Oriented Gradients for Human Detection* [14] — you may know this method as **HOG + Linear SVM**.

The HOG + Linear SVM method tends to be easier to train, including:

- Higher detection accuracy
- Fewer parameters to tune during detection/inference
- A lower false-positive rate

The downside of HOG + Linear SVM is that they tend to be slower than Haar cascades.

Both Haar cascades and HOG + Linear SVM share two important components: **sliding windows** and **image pyramids**.

A sliding window allows an object detector to localize and detect *where* in an image an object exists. When we combine a sliding window with an image pyramid we can detect objects at different *scales* in the image. If your image contains objects at varying scales (closer or further away from the camera), you'll need an image pyramid to detect it.

We'll review both sliding windows and image pyramids, along with the other components needed to build a simple object detector in the next section.

13.2 The Object Detection Pipeline

When building a deep learning object detector using classic object detection paradigms we need four key components (Figure 13.1, *left*):

1. A **sliding window** that sits on top of our image and slides from left-to-right and top-to-bottom, classifying each Region of Interest (ROI) along the way. The sliding window enables us to detect exactly *where* in an image an object is.
2. We also need an **image pyramid** that sequentially reduces (or in some cases, increases) the size of our input image. We call this an image pyramid because if we stack our images from largest to smallest it looks like a pyramid (Figure 13.1, *right*). Our sliding window runs on each scale of the image pyramid, enabling us to detect objects that are both closer and farther away from the camera.
3. Using a sliding window and image pyramid implies that our model will report *multiple detections* for the same object (a phenomenon we'll discuss later in this section). When multiple detections surrounding the same object happen, we need to apply **non-maxima suppression** to keep only the most confident prediction.
4. Finally, to ensure our object detector runs as fast as possible, especially if a GPU is available, we need a mechanism to apply **batch processing** to the ROIs.

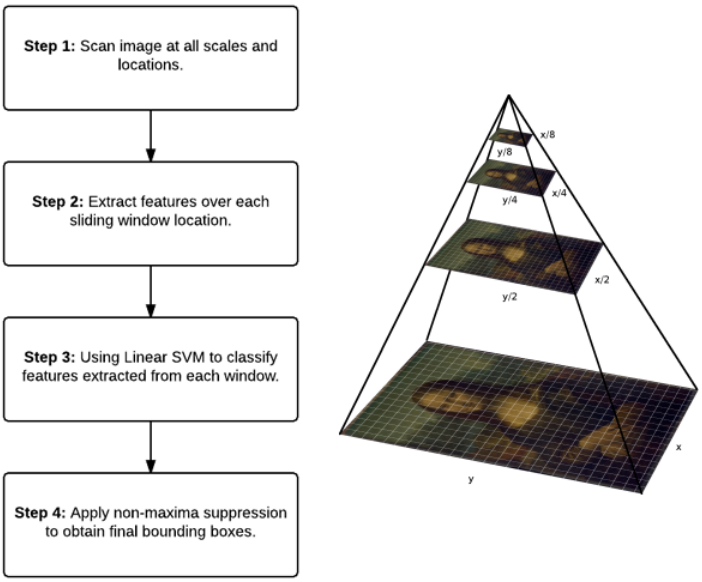


Figure 13.1: **Left:** The four fundamental steps of traditional object detection. **Right:** A visualization of an image pyramid. The original image sits at the bottom of the pyramid. Subsequent layers of the pyramid reduce image size (Image credit: IIPImage [54]).

Figure 13.1, *left* provides a visualization of the standard object detection pipeline. First, we utilize an image pyramid + sliding window to scan all scales and locations of our input image. We then classify each of the ROIs using our CNN. Finally, we apply non-maxima suppression to obtain our final bounding boxes.

Let's examine each of these components individually and implement them by hand to form our final object detection pipeline.

13.2.1 Sliding Windows

Sliding windows play an integral role in object detection as they enable us to localize exactly *where*, in terms of (x, y)-coordinates, in an image an object resides. Utilizing both a sliding window and an image pyramid (to be discussed in the next section), we are able to detect objects and various cases and locations. So, what exactly *is* a sliding window?

In the context of computer vision and machine learning (and as the name suggests), a sliding window is rectangular region of *fixed width and height* that "slides" across an image, from left-to-right and top-to-bottom.

Figure 13.2 provides visualizations of a sliding window. The background is the image we are trying to detect an object in, while the foreground, the green box, is our visualization of the sliding window. Here we can see that the sliding window is sliding from left-to-right and top-to-bottom. For each of these windows we take the window region (called our ROI) and pass it through our CNN to determine if the window contains an object of interest to us.

A book, by definition, cannot contain animations. I have put together a separate discussion of sliding windows, including animations and visualizations, on the PyImageSearch blog: http://pyimg.co/c8c0j

Let's go ahead and implement our sliding window. Create a new file named `simple_obj_det.py` in the `utils` sub-module of `pyimagesearch` — this file is where we will store our helper functions

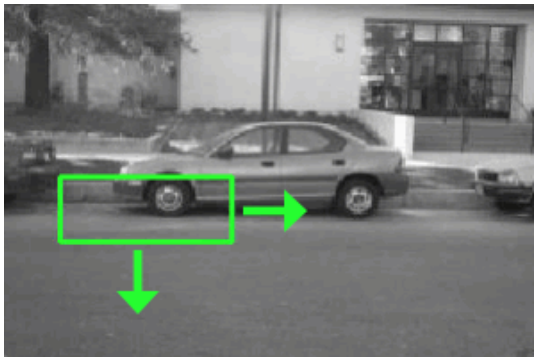


Figure 13.2: A sliding window slides from left-to-right and top-to-bottom across an input image taking N pixel steps at a time. The ROI at each step of the sliding window is extracted and passed into the feature extraction/object detection pipeline.

for object detection:

```
--- pyimagesearch
|    |--- __init__.py
...
|    |--- utils
|    |    |--- __init__.py
|    |    |--- ...
|    |    |--- simple_obj_det.py
```

From there, open up `simple_obj_det.py` and insert the following code:

```
1  # import the necessary packages
2  from keras.applications import imagenet_utils
3  import imutils
4
5  def sliding_window(image, step, ws):
6       # slide a window across the image
7       for y in range(0, image.shape[0] - ws[1], step):
8            for x in range(0, image.shape[1] - ws[0], step):
9                 # yield the current window
10                yield (x, y, image[y:y + ws[1], x:x + ws[0]])
```

Our `sliding_window` function requires three arguments. The first is the `image` that we are going to loop over. The second is `step`, the step size indicating the number of pixels we are going to "skip" in both the x and y direction.

You may be tempted to set `step=1`, implying that no pixels are skipped; however, we typically do not want to do this. Setting `step=1` would make it computationally prohibitive to apply our CNN at every ROI in the image. Instead, the `step` is determined on a *per-dataset* basis and is tuned to give optimal performance on your dataset of images. In practice, it's common to see a `step` of 4, 8, or 16 pixels.

Remember, the smaller your step size is, the more windows you'll need to examine. The larger your step size is, the fewer windows you'll need to examine — note, however, that while a larger step size may be computationally more efficient, you may miss detecting objects in images if your step size becomes too large.

The last argument, `ws`, defines the width and height (in terms of pixels) of the window we are going to extract from our `image`.

Lines 7-10 are fairly straightforward and handle the actual "sliding" of the window. This action is accomplished by defining two `for` loops (**Lines 7 and 8**) that loop over the (x, y)-coordinates of the image, incrementing their respective `x` and `y` counters by the provided step size.

Then, **Line 10** returns a tuple containing the (x, y)-coordinates of the sliding window along with the ROI itself. This ROI will later be passed into a CNN for classification.

13.2.2 Image Pyramids

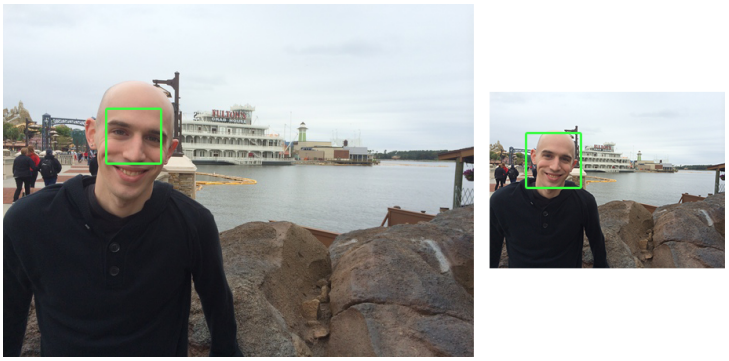


Figure 13.3: **Left:** A sliding window requires *fixed* spatial dimensions. If the object in the window is too large or small for the sliding window size, we can miss the detection. **Right:** To resolve this problem, we can utilize image pyramids and apply a sliding window to *each* layer in the pyramid. The image pyramid layer that is 50% smaller than the input image allows our sliding window to neatly fit overtop the face, giving our machine learning model a better chance at detecting the face.

An image pyramid is simply a *multi-scale representation* of an image (Figure 13.1, *right*). Using an image pyramid allows us to find objects in images at different scales of an image.

At the bottom of the pyramid, we have the original image at its original size (in terms of width and height). At each subsequent layer, the image is resized (subsampled). The image is progressively subsampled until some stopping criterion is met which is typically a minimum size being reached, indicating that no further subsampling needs to take place.

When combining an image pyramid with a sliding window, we can find objects in images at both *varying locations* and *varying scales* — to see how this behavior is true, take a look at Figure 13.3.

On the *left*, we are applying a sliding window to our original input image. Notice how the object is too large to fit inside our sliding window, implying that we will very likely miss this detection. However, by applying an image pyramid and subsampling to a smaller size (*right*) our sliding window fits neatly overtop the object, implying that we will likely detect it.

Let's go ahead and implement our `image_pyramid` function — open up `simple_object_det.py` and append the following code:

```
12 def image_pyramid(image, scale=1.5, minSize=(224, 224)):
13     # yield the original image
14     yield image
15
16     # keep looping over the image pyramid
17     while True:
```

```
18            # compute the dimensions of the next image in the pyramid
19            w = int(image.shape[1] / scale)
20            image = imutils.resize(image, width=w)
21
22            # if the resized image does not meet the supplied minimum
23            # size, then stop constructing the pyramid
24            if image.shape[0] < minSize[1] or image.shape[1] < minSize[0]:
25                 break
26
27            # yield the next image in the pyramid
28            yield image
```

Our `image_pyramid` function requires a single argument: the `image` we wish to apply the pyramid to. We can then optionally supply the `scale` which controls how the image is resized at each layer. A small `scale` yields more layers in the pyramid, while larger `scale` yields less layers. Typically the scale is set between `1.1-1.5` in 0.1 or 0.05 increments.

We then have the `minSize` attribute, which is the minimum required width and height of the layer. If an image in the pyramid falls below `minSize`, we stop constructing the image pyramid.

Line 14 yields the original image in the pyramid (the bottom layer).

From there, we start looping over the image pyramid on **Line 17**. **Lines 19 and 20** handle computing the size of the image in the next layer of the pyramid (while preserving the aspect ratio). The scale is controlled by the `scale` factor.

On **Lines 24 and 25** we make a check to ensure the newly scaled image meets our `minSize` requirements. If it does not, we break from the loop. Finally, **Line 28** yields our resized image to the calling function.

Combining both `sliding_window` and `image_pyramid` we can now detect objects at varying scales and locations of an image.

13.2.3 Batch Processing

So far we have learned how to apply image pyramids and sliding windows to extract individual ROIs — but how do we actually *predict* the class label for each ROI?

Simple! We'll just use the `model.predict` function from Keras, just as if we were performing image classification.

As we know from our previous chapters in both the *Starter Bundle* and *Practitioner Bundle*, CNNs are most efficient when processing *batches*; therefore, we need to devise a function that can accept a set of batch ROIs and batch (x, y)-locations, make predictions on them, and return the labels and associated probabilities for the most confident classes.

To accomplish this batching, let's define our `classify_batch` function in `simple_object_det.py`:

```
30 def classify_batch(model, batchROIs, batchLocs, labels, minProb=0.5,
31       top=10, dims=(224, 224)):
32       # pass our batch ROIs through our network and decode the
33       # predictions
34       preds = model.predict(batchROIs)
35       P = imagenet_utils.decode_predictions(preds, top=top)
```

Our `classify_batch` function accepts a number of parameters, each detailed below:

- `model`: The Keras model that we will be using for classification.
- `batchROIs`: A NumPy array containing the batch of ROIs we will be classifying. This array is constructed in the exact same manner as the previous chapter where we build batches of images.

- `batchLocs`: The (x, y)-coordinates (in terms of the original image) of each ROI in `batchROIs`.
- `labels`: A class labels dictionary that will be maintained during the entire classification process. The key to this dictionary is the label name, and the value is a list of tuples, including bounding box coordinates and associated class label probability.
- `minProb`: The minimum required probability for a classification to be considered a valid detection. We use this parameter to filter out weak detections.
- `top`: The number of top-K predictions to be returned by the network.
- `dims`: Spatial dimensions of the bounding box (should match spatial dimensions of the network).

Line 34 makes a call to `model.predict`, passing in our `batchROIs`. Since we will be using a network pre-trained on ImageNet for this example (to be covered in Section 13.3), we'll need to decode the predictions (**Line 35**).

We can now loop over our decoded predictions:

```
37      # loop over the decoded predictions
38      for i in range(0, len(P)):
39          for (_, label, prob) in P[i]:
40              # filter out weak detections by ensuring the
41              # predicted probability is greater than the minimum
42              # probability
43              if prob > minProb:
44                  # grab the coordinates of the sliding window for
45                  # the prediction and construct the bounding box
46                  (pX, pY) = batchLocs[i]
47                  box = (pX, pY, pX + dims[0], pY + dims[1])
48
49                  # grab the list of predictions for the label and
50                  # add the bounding box + probability to the list
51                  L = labels.get(label, [])
52                  L.append((box, prob))
53                  labels[label] = L
54
55      # return the labels dictionary
56      return labels
```

On **Lines 38 and 39** we loop over each of the predictions made by our CNN.

Line 43 filters out weak predictions by ensuring the predicted probability is greater than the minimum required probability. Provided `prob > minProb` we extract the corresponding bound box (x, y)-coordinates from `batchLocs` and construct our bounding box (**Lines 46 and 47**).

We then need to maintain our `labels` dictionary on **Lines 51-53** by grabbing the list of predictions for the label and adding the bounding box + associated probability to the list. Finally, **Line 56** returns the `labels` dictionary to the calling function.

In Section 13.3 of this chapter we'll see how `sliding_window`, `image_pyramid`, and `classify_batch` all fit together to create a fully-functioning object detection pipeline.

13.2.4 Non-maxima Suppression

There's an inescapable "issue" you must handle when building object detection systems — *overlapping bounding boxes*, as demonstrated by Figure 13.4 (*left*).

Overlapping bounding boxes is going to happen, there's no way around it. In fact, it's actually a good sign that your object detector is working properly so it's not even fair to call it a true "issue" at all.

It would be far worse if your detector (1) reported a false positive (i.e., detected an object where there isn't one) or (2) failed to detect an object altogether.

The reason we see multiple bounding boxes reported per object lies in the very nature of our object detection system: *sliding windows*. Consider how a sliding window works — it slides from left-to-right and top-to-bottom of an image, classifying an input ROI via our CNN at each step of the way.

Therefore, when a sliding window starts to get closer to an object it starts to report a high(er) probability for the corresponding label. The sliding window is thus approaching the "hot" zone. The closer the sliding window gets to the true object, the hotter it gets, and thus more bounding boxes are reported. The farther the sliding window gets from the true object, the fewer bounding boxes will be reported as the window is now in the "cool" zone.

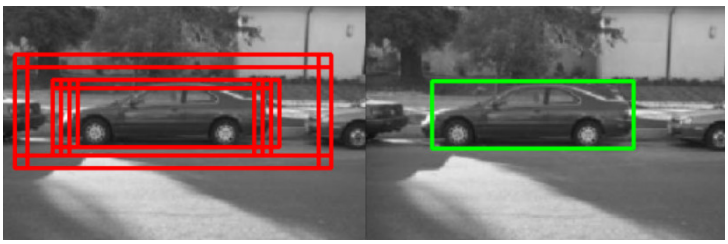


Figure 13.4: **Left:** A set of bounding box returned by our object detection pipeline. **Right:** Applying non-maxima suppression (NMS) to suppress overlapping bounding boxes, keeping only the one with the largest probability/confidence.

When combined with image pyramids, this behavior implies that we will have *multiple bounding boxes* surrounding the object at *multiple scales*, even though there may only be one "true" object in the image.

We thus need a way to suppress and ignore the bounding boxes *except* for the most confident prediction.

To handle the removal of overlapping bounding boxes (that refer to the same object) we can apply non-maxima suppression (NMS). NMS works by computing the ratio of overlap between bounding boxes, then suppressing (i.e., removing) bounding boxes that have significant overlap.

Implementing NMS by hand is outside the scope of this chapter, but I encourage you to review the following two blog posts on PyImageSearch that discuss two variants of NMS and their associated implementations in detail:

- *Non-Maximum Suppression for Object Detection in Python:* http://pyimg.co/fz1ak
- *(Faster) Non-Maximum Suppression in Python:* http://pyimg.co/gwunq

I would also recommend looking at my non-maxima suppression implementation in the `imutils` library which we will be using later in this chapter: http://pyimg.co/9dne5

13.3 Implementing Object Detection With a CNN

Now that we have implemented our helper methods, we are ready to put them all together and form the complete object detection pipeline. Open up a new file, name it `simple_detection.py`, and insert the following code:

```
# import the necessary packages
from pyimagesearch.utils.simple_obj_det import image_pyramid
from pyimagesearch.utils.simple_obj_det import sliding_window
from pyimagesearch.utils.simple_obj_det import classify_batch
```

```
5  from keras.applications import ResNet50
6  from keras.preprocessing.image import img_to_array
7  from keras.applications import imagenet_utils
8  from imutils.object_detection import non_max_suppression
9  import numpy as np
10 import argparse
11 import time
12 import cv2
13
14 # construct the argument parse and parse the arguments
15 ap = argparse.ArgumentParser()
16 ap.add_argument("-i", "--image", required=True,
17      help="path to the input image")
18 ap.add_argument("-c", "--confidence", type=float, default=0.5,
19      help="minimum probability to filter weak detections")
20 args = vars(ap.parse_args())
```

Lines 2-12 handle importing our required Python packages. In particular notice how **Lines 2-4** import our helper functions for the image pyramid, sliding window, and batch classification routines. **Line 8** then imports the non-maxima suppression function from the `imutils` library.

We'll only need one command line argument for the script, the path to the input `--image`. We can optionally supply a `--confidence` (i.e., minimum probability for an input ROI to be considered a valid detection) used to filter out weak detections.

Next, we have some important constants to initialize:

```
22 # initialize variables used for the object detection procedure
23 INPUT_SIZE = (350, 350)
24 PYR_SCALE = 1.5
25 WIN_STEP = 16
26 ROI_SIZE = (224, 224)
27 BATCH_SIZE = 64
```

These variables control how the object detection is performed:

- `INPUT_SIZE`: These are the width and height of our input `--image`. Our image is resized, ignoring aspect ratio, to `INPUT_SCALE` prior to being fed through our neural network.
- `PYR_SCALE`: The scale of our image pyramid. A smaller scale corresponds to *more layers* generated while a larger scale implies *fewer layers*.
- `WIN_STEP`: The step of our sliding window. The smaller the step, the more windows will be evaluated, and consequently the slower our detector will run. The larger the step, fewer windows will be evaluated and our detector will run faster. There is a tradeoff between window size, speed, and accuracy. If your step is too large you may miss detections. If your step is too small, your detector will take a long time to run.
- `ROI_SIZE`: The input ROI size to our CNN as if we were performing classification.
- `BATCH_SIZE`: The size of the batch to be built and passed through the CNN.

Let's continue with our initializations and preparing our image detector:

```
29 # load our the network weights from disk
30 print("[INFO] loading network...")
31 model = ResNet50(weights="imagenet", include_top=True)
32
33 # initialize the object detection dictionary which maps class labels
```

```
34 # to their predicted bounding boxes and associated probability
35 labels = {}
36
37 # load the input image from disk and grab its dimensions
38 orig = cv2.imread(args["image"])
39 (h, w) = orig.shape[:2]
40
41 # resize the input image to be a square
42 resized = cv2.resize(orig, INPUT_SIZE, interpolation=cv2.INTER_CUBIC)
43
44 # initialize the batch ROIs and (x, y)-coordinates
45 batchROIs = None
46 batchLocs = []
47
48 # start the timer
49 print("[INFO] detecting objects...")
50 start = time.time()
```

On **Line 31** we initialize `ResNet50`, pre-trained on the ImageNet dataset. You can swap in your own network architecture + weights if you wish.

On **Line 35** we initialize the `labels` dictionary. The key to `labels` will be the predicted label from our CNN. The value will be the (x,y)-coordinates of the prediction. We will later run non-maxima suppression on each of the bounding boxes for each label.

Lines 38-42 load our input image from disk and resize it to `INPUT_SIZE`. We then initialize our batch ROIs and batch (x,y)-locations on **Lines 46**.

We are now ready to start the object detection procedure:

```
52 # loop over the image pyramid
53 for image in image_pyramid(resized, scale=PYR_SCALE,
54      minSize=ROI_SIZE):
55      # loop over the sliding window locations
56      for (x, y, roi) in sliding_window(image, WIN_STEP, ROI_SIZE):
57           # take the ROI and pre-process it so we can later classify the
58           # region with Keras
59           roi = img_to_array(roi)
60           roi = np.expand_dims(roi, axis=0)
61           roi = imagenet_utils.preprocess_input(roi)
62
63           # if the batch is None, initialize it
64           if batchROIs is None:
65                batchROIs = roi
66
67           # otherwise, add the ROI to the bottom of the batch
68           else:
69                batchROIs = np.vstack([batchROIs, roi])
70
71           # add the (x, y)-coordinates of the sliding window to the batch
72           batchLocs.append((x, y))
```

On **Line 53** we start looping over every scale of the image generated by the `image_pyramid`. For each scale, we need to apply our `sliding_window` to extract each ROI every `WIN_STEP` (**Line 56**).

Lines 59-61 handle preprocessing our `roi` just as we would for classification. **Lines 63-72** build our set of `batchROIs` and `batchLocs` so we can classify the batch.

If our batch is full, we apply the `classify_batch` function to efficiently classify the ROIs:

```
74          # check to see if our batch is full
75          if len(batchROIs) == BATCH_SIZE:
76               # classify the batch, then reset the batch ROIs and
77               # (x, y)-coordinates
78               labels = classify_batch(model, batchROIs, batchLocs,
79                    labels, minProb=args["confidence"])
80
81               # reset the batch ROIs and (x, y)-coordinates
82               batchROIs = None
83               batchLocs = []
```

Since we are batch processing, we may encounter a situation where we have finished looping over our image pyramid + sliding windows, but still have images left in our batch — the following code block checks to see if there are still images in the batch, and if so, applies `classify_batch`:

```
85  # check to see if there are any remaining ROIs that still need to be
86  # classified
87  if batchROIs is not None:
88       labels = classify_batch(model, batchROIs, batchLocs, labels,
89            minProb=args["confidence"])
90
91  # show how long the detection process took
92  end = time.time()
93  print("[INFO] detections took {:.4f} seconds".format(end - start))
```

The `labels` dictionary contains the class labels and associated (x,y)-coordinates for each object in the image — our job is to loop over the `labels` dictionary to obtain our final detections:

```
95  # loop over the labels for each of detected objects in the image
96  for k in labels.keys():
97       # clone the input image so we can draw on it
98       clone = resized.copy()
99
100      # loop over all bounding boxes for the label and draw them on
101      # the image
102      for (box, prob) in labels[k]:
103           (xA, yA, xB, yB) = box
104           cv2.rectangle(clone, (xA, yA), (xB, yB), (0, 255, 0), 2)
105
106      # show the image *without* apply non-maxima suppression
107      cv2.imshow("Without NMS", clone)
108      clone = resized.copy()
```

On **Line 96** we start looping over each of the keys, `k`, in `labels`. Looking up the key `k` in `labels`, we have all bounding boxes and associated probabilities for the current label (**Line 102**). We draw each of the bounding boxes on our image *without* applying non-maxima suppression on **Line 104**.

Let's now apply non-maxima suppression so we can visualize the difference and importance that non-maxima suppression makes:

```
110     # grab the bounding boxes and associated probabilities for each
111     # detection, then apply non-maxima suppression to suppress
112     # weaker, overlapping detections
113     boxes = np.array([p[0] for p in labels[k]])
114     proba = np.array([p[1] for p in labels[k]])
115     boxes = non_max_suppression(boxes, proba)
116
117     # loop over the bounding boxes again, this time only drawing the
118     # ones that were *not* suppressed
119     for (xA, yA, xB, yB) in boxes:
120         cv2.rectangle(clone, (xA, yA), (xB, yB), (0, 0, 255), 2)
121
122     # show the output image
123     print("[INFO] {}: {}".format(k, len(boxes)))
124     cv2.imshow("With NMS", clone)
125     cv2.waitKey(0)
```

Lines 113 and 114 extract the bounding box (x,y)-coordinates and associated probabilities from our `labels` dictionary with label `k`. **Line 115** then applies the non-maxima suppression routine, suppressing weaker, overlapping detections.

Our final localizations are then drawn on our input image and displayed to our screen (**Lines 119-125**).

13.4 Simple Object Detection Results

To see our full deep learning object detection pipeline in action, open up a shell and execute the following command:

```
$ python simple_detection.py --image beagle.png --confidence 0.75
[INFO] loading network...
[INFO] detecting objects...
[INFO] detections took 1.9330 seconds
[INFO] beagle: 1
```

Taking a look at Figure 13.5 (*left*) we can see our input image and bounding boxes drawn on it — these bounding boxes *have not* had non-maxima suppression applied to them. Notice how there are *multiple* bounding boxes surrounding the beagle even though there is only *one* dog in the image.

Applying non-maxima suppression (*right*) suppresses the weaker, overlapping detections and leaves us with the most confident bounding box.

13.5 Summary

In this chapter we discussed the fundamentals of object detection in the context of deep learning. We then implemented a simple deep learning-based object detector, using:

1. Sliding windows
2. Image pyramids
3. Non-maxima suppression
4. Batch processing

The benefit of this approach is that we can treat *any* deep learning model trained for *classification* as an *object detector*. We utilized ResNet pre-trained on ImageNet; however, you can just as easily swap in your own models trained for classification and utilize them for object detector.

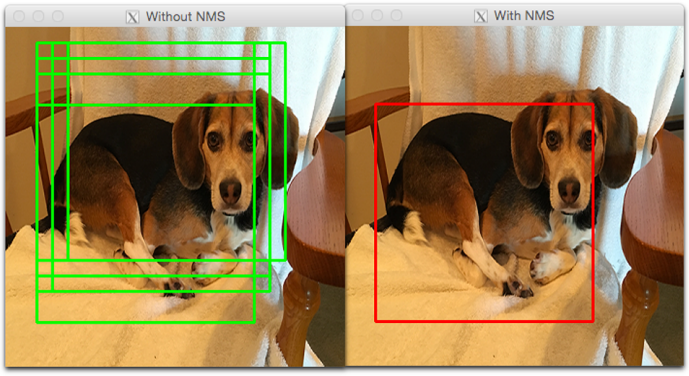


Figure 13.5: **Left:** Applying our object detector yields multiple bounding boxes even though there is only one dog in the image. **Right:** Applying non-maxima suppression to the weaker bounding boxes, yielding only a single bounding box.

However, there are many downsides to treating a neural network trained for classification as an object detector, namely:

- Sliding windows + image pyramids are incredibly slow, even when utilizing a GPU for inference
- It can be tedious to tune the scale for the image pyramid and step size for sliding window
- Due to the tediousness of the parameter selection, we can easily miss objects in our images

With these negatives in mind, it raises the question:

> *"Is there a way to build an end-to-end object detector with deep learning? And if so, why even bother studying the fundamentals of object detection?"*

The answer to the first part of the question is, *yes, we can train end-to-end deep learning object detectors*, but we need to leverage specific network architectures and frameworks to do so, namely Faster R-CNNs [51] and SSDs [52].

To answer the second question, we need to understand the concept of sliding window to understand how traditional methods localized objects. Deep learning-based object detectors utilize either:

1. Region proposal methods to zero in on the areas of an image that look "interesting" and therefore deserve closer attention and more computation.
2. Image division where an image is partitioned into regions, passed into a CNN, and then the regions are modified and grouped together based on the output predictions.

It would be extremely challenging, if not impossible, to understand and appreciate these methods to object detection without first understanding the classical approach of image pyramids and sliding windows.

For a detailed discussion on these architectures and frameworks, along with a thorough guide on how to train your own object detectors using Faster R-CNNs and SSDs, please refer to Chapters 14-17 in the *ImageNet Bundle*.

14. DeepDream

DeepDream, colloquially known as "deep dreaming", is a technique used to produce dream-like hallucinogenic images using Convolutional Neural Networks (CNNs). This method was first introduced by Mordvintsev et al. in a 2015 Google Research blog post, *DeepDream: A Code Example for Visualizing Neural Networks* [55]. Immediately after their release of the method, the technique went viral.

Deep learning researchers and practitioners were incredibly interested in visualizing what their CNNs "dreamed about", if for nothing more:

1. An exercise in running a CNN in reverse
2. A fun novelty experiment

The algorithm and associated results were *so popular* that within days of it being released I had authored *bat-country*, an extendible, lightweight Python package for deep dreaming with Caffe + Python [56], along with three separate blog posts on PyImageSearch [57, 58, 59].

Even non-programmers have found the results of the algorithm interesting and share-worthy — entire online communities have cropped up, sharing their latest generated dreams with others [60].

In this chapter we are going to:

- Discuss the DeepDream algorithm
- Implement it using Python + Keras
- Visualize the results of CNN generated "hallucinations"

14.1 What Is DeepDream?

The DeepDream algorithm can be summarized concisely as "running a (pre-trained) CNN in reverse":

1. We start with an input image
2. We process the input image at different scales, called *octaves*
3. For each octave we maximize the activation of entire layer sets, mixing the results together to obtain "trippy" effects

An example of an image generated by the DeepDream algorithm can be seen in Figure 14.1. On the *left* we have the original input image and on the *right* we have the output generated dream.

Figure 14.1: Vincent van Gogh's *The Starry Night* (*left*) after the DeepDream algorithm has been ran on it, generating "psychedelic hallucinations" (*right*). Image credit ArtNet [61].

Unlike other deep learning examples in this book, instead of training a network and modifying the weights of each layer along the way, we instead freeze the weights (implying that they cannot be modified) and instead modify the actual *input image*. This process creates a *feedback loop* where the input image is fed back into the CNN, the CNN runs in reverse, and the output effects are amplified on our image.

Utilizing lower-level layers of the network generate edge-like regions in images. Intermediate layers are able to represent basic shapes and components of objects (doorknobs, eyes, noses, etc.). The highest-level layers of the network are able to construct a complete interpretation of an object such as dogs, cats, birds, and other well-formed objects — *the highest-level layers of the network generate your most psychedelic images*.

We typically use CNNs trained on the ImageNet dataset to produce more interesting dreams; however, networks pre-trained on ImageNet have the byproduct of consistently generating hallucinations that include dogs, cats, and birds. The reason for this effect is simple — these classes are overrepresented in the ImageNet dataset.

14.2 Implementing DeepDream

Our implementation of the DeepDream algorithm is based on François Chollet's code from the official Keras repository [62] and his book, *Deep Learning with Python* [63]. We'll be covering this code in additional detail and providing supplementary results and experiments to help you generate your own DeepDream images.

To get started, open up a new file, name it `deep_dream.py`, and insert the following code:

```
# import the necessary packages
from keras.applications import InceptionV3
from keras.applications.inception_v3 import preprocess_input
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img
from keras import backend as K
from scipy import ndimage
import numpy as np
import argparse
import cv2
```

We start off with our set of imports. We'll be using the Inception architecture pre-trained on the ImageNet dataset to generate our dreams — we can technically use any architecture we wish for this process, but Inception will enable us to utilize the same architecture as Mordvintsev et al. (plus, the name is fitting for a program that will generate hallucination-like images).

In order to pre-process, generate, and de-process our images we'll need a handful of helper functions — let's review each of the helper functions *before* we reach the main pipeline of the program.

The first function is `preprocess`, a function that should look similar to you from our previous chapters:

```
12 def preprocess(p):
13     # load the input image, convert it to a Keras-compatible array,
14     # expand the dimensions so we can pass it through the model, and
15     # then finally preprocess it for input to the Inception network
16     image = load_img(p)
17     image = img_to_array(image)
18     image = np.expand_dims(image, axis=0)
19     image = preprocess_input(image)
20
21     # return the preprocessed image
22     return image
```

This function accepts the path to an input image, `p`, loads the image, and preprocesses it for input to our CNN.

Just as we need to pre-process an image, we also need to "de-process" it so we can visualize the results and save it to disk:

```
24 def deprocess(image):
25     # we are using "channels last" ordering so ensure the RGB
26     # channels are the last dimension in the matrix
27     image = image.reshape((image.shape[1], image.shape[2], 3))
28
29     # "undo" the preprocessing done for Inception to bring the image
30     # back into the range [0, 255]
31     image /= 2.0
32     image += 0.5
33     image *= 255.0
34     image = np.clip(image, 0, 255).astype("uint8")
35
36     # we have been processing images in RGB order; however, OpenCV
37     # assumes images are in BGR order
38     image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
39
40     # return the deprocessed image
41     return image
```

Assuming we are using channels-last ordering, we reshape our input array to place the RGB channels as the last dimension (**Line 27**).

Next, **Lines 31-34** handle undoing the scaling performed by Inceptions `preprocess_input` function. **Line 34** is important as it ensures any values that fall outside the range $[0, 255]$ are clipped. We assume images are in RGB order; however, since we'll be using OpenCV in this example (which assumes BGR ordering) we perform channel reordering on **Line 38**.

When building our output dream image we'll need to perform a process called *detail reinfection* to avoid losing detail in our image (a process we'll cover later in this section). To facilitate this process we need `resize_image`, a helper function used to resize an input image to specific spatial dimensions:

```
43 def resize_image(image, size):
44     # resize the image
45     resized = np.copy(image)
46     resized = ndimage.zoom(resized,
47         (1, float(size[0]) / resized.shape[1],
48         float(size[1]) / resized.shape[2], 1), order=1)
49
50     # return the resized image
51     return resized
```

The actual resizing on the image takes place on **Lines 46-48** where we call SciPy's `.zoom` function. The scale of both the width and height are determined by dividing the requested output `size` by the shape of the image *prior* to resizing.

From there we need a helper function to fetch the loss and gradients from a Keras function:

```
53 def eval_loss_and_gradients(X):
54     # fetch the loss and gradients given the input
55     output = fetchLossGrads([X])
56     (loss, G) = (output[0], output[1])
57
58     # return a tuple of the loss and gradients
59     return (loss, G)
```

The `fetchLossGrads` method called on **Line 55** is a Keras function we'll be defining in the main pipeline of our script — this function is responsible for retrieving the value of the loss and gradients given an input image. We unpack the loss and gradients tuple on **Line 56** and then return them to the calling function (**Line 59**).

The next function, `gradient_ascent`, is responsible for generating our actual dream:

```
61 def gradient_ascent(X, iters, alpha, maxLoss=-np.inf):
62     # loop over our number of iterations
63     for i in range(0, iters):
64         # compute the loss and gradient
65         (loss, G) = eval_loss_and_gradients(X)
66
67         # if the loss is greater than the max loss, break from the
68         # loop early to prevent strange effects
69         if loss > maxLoss:
70             break
71
72         # take a step
73         print("[INFO] Loss at {}: {}".format(i, loss))
74         X += alpha * G
75
76     # return the output of gradient ascent
77     return X
```

This function requires three parameters and an optional argument:

- `X`: This is our input tensor (i.e., the input image).
- `iters`: The total number of iterations to run for.
- `alpha`: The step size/learning rate when applying gradient descent.
- `maxLoss`: If our `loss` exceeds `maxLoss` we terminate the gradient ascent process early, preventing us from generating artifacts in our output image.

The actual process of applying gradient ascent is very similar to the examples we covered in Chapters 8, 9, and 10 of the *Starter Bundle*:

1. We first loop over a number of iterations (**Line 63**).
2. We compute the loss and gradients for our input (**Line 65**).
3. And then finally apply the actual gradient ascent step (**Line 74**).

We'll see how all these helper functions enable us to perform deep dreaming in the main body of our script:

```
79 # construct the argument parse and parse the arguments
80 ap = argparse.ArgumentParser()
81 ap.add_argument("-i", "--image", required=True,
82      help="path to input image")
83 ap.add_argument("-o", "--output", required=True,
84      help="path to output dreamed image")
85 args = vars(ap.parse_args())
```

Here we parse our command line arguments: `--image`, the path to the input image, and `--output`, the path to the output dreamed image.

From there we need to define a dictionary that specifies which layers we'll be using generate the dream:

```
87 # define the dictionary that includes (1) the layers we are going
88 # to use for the dream and (2) their respective weights (i.e., the
89 # larger the weight, the more the layer contributes to the dream)
90 LAYERS = {
91      "mixed2": 2.0,
92      "mixed3": 0.5,
93 }
```

Lower layers can be used to generate edges and geometric patterns while high layers result in the injection of trippy visual patterns, including hallucinations of dogs, cats, and birds.

The layers you choose here will have a big impact on your output image. Be sure to explore the output of `model.summary()` and experiment with combinations of layers and corresponding weights (the larger the weight, the *more* the layer contributes to the output dream).

We also need to define a number of constants:

```
95 # define the number of octaves, octave scale, alpha (step for
96 # gradient ascent) number of iterations, and max loss -- tweaking
97 # these values will produce different dreams
98 NUM_OCTAVE = 3
99 OCTAVE_SCALE = 1.4
100 ALPHA = 0.001
101 NUM_ITER = 50
102 MAX_LOSS = 10.0
```

The NUM_OCTAVE variable is the number of octaves (resolutions) we are going to generate. Each successive octave, controlled by OCTAVE_SCALE will be 1.4x larger than the previous on (i.e., 40% larger). We'll start with a small input image and then gradually increase the resolution via our resize_image function.

The ALPHA variable is our step size for gradient ascent. NUM_ITERS controls the total number of gradient ascent operations we'll apply.

Finally, MAX_LOSS is our early stopping criteria — if our computed loss exceeds MAX_LOSS we'll break from the gradient ascent loop early to prevent generating artifacts in our output image.

Next, let's load our Inception network, pre-trained on ImageNet, from disk:

```
104 # indicate that Keras *should not* be update the weights of any
105 # layer during the deep dream
106 K.set_learning_phase(0)
107
108 # load the (pre-trained) Inception model from disk, then grab a
109 # reference variable to the input tensor of the model (which we'll
110 # then be using to perform our CNN hallucination)
111 print("[INFO] loading inception network...")
112 model = InceptionV3(weights="imagenet", include_top=False)
113 dream = model.input
```

Line 106 we call .set_learning_phase and set the value to 0 — since we won't be training our network but instead constructing a feedback loop for our input image by running Inception in reverse, this function call disables all training related operations. **Line 113** initializes the dream variable — this tensor holds our actual generated hallucination image. We will be updating dream throughout this script.

We are now ready to compute the loss, the value we'll be maximizing when running gradient ascent:

```
115 # define our loss value, then build a dictionary that maps the
116 # *name* of each layer inside of Inception to the actual *layer*
117 # object itself -- we'll need this mapping when building the loss
118 # of the dream
119 loss = K.variable(0.0)
120 layerMap = {layer.name: layer for layer in model.layers}
121
122 # loop over the layers that will be utilized in the dream
123 for layerName in LAYERS:
124     # grab the output of the layer we will use for dreaming, then add
125     # the L2-norm of the features to the layer to the loss (we use
126     # array slicing here to avoid border artifacts caused by border
127     # pixels)
128     x = layerMap[layerName].output
129     coeff = LAYERS[layerName]
130     scaling = K.prod(K.cast(K.shape(x), "float32"))
131     loss += coeff * K.sum(K.square(x[:, 2: -2, 2: -2, :])) / scaling
```

Line 119 initializes the loss value itself as a Keras variable. **Line 120** builds a dictionary that, for all layers in the model, maps the layer name to the layer object itself — this mapping will enable us to look up each layer in LAYERS and build the loss value.

On **Line 123** we loop over all `LAYERS` that will be utilizing in building our dream. **Line 128** grabs the output, `x` (i.e., the activations) of the current `layerName` using our `layerMap`. The `coeff` value (**Line 129**) is the corresponding weight/contribution of the layer to the output dream.

In order to generate our deep dream we need to compute the L2-loss of the activations (`K.sum` and `K.square`) and add them to our `loss`. We perform array slicing on **Line 131** (`x[:, 2: -2, 2: -2, :]`) to remove border pixels, thereby removing border artifacts in our output dream.

We can now compute our gradients and normalize:

```
133 # compute the gradients of the dream with respect to loss and then
134 # normalize
135 grads = K.gradients(loss, dream)[0]
136 grads /= K.maximum(K.mean(K.abs(grads)), 1e-7)
```

And then define a Keras function that can be used to grab the value of the loss and gradients of our image:

```
138 # we now need to define a function that can retrieve the value of the
139 # loss and gradients given an input image
140 outputs = [loss, grads]
141 fetchLossGrads = K.function([dream], outputs)
```

Let's now load our input `--image` from disk:

```
143 # load and preprocess the input image, then grab the (original) input
144 # height and width
145 image = preprocess(args["image"])
146 dims = image.shape[1:3]
147
148 # in order to perform deep dreaming we need to build multiple scales
149 # of the input image (i.e., set of images at lower and lower
150 # resolutions) -- this list stores the spatial dimensions that we
151 # will be resizing our input image to
152 octaveDims = [dims]
153
154 # here we loop over the number of octaves (resolutions) we are going
155 # to generate
156 for i in range(1, NUM_OCTAVE):
157     # compute the spatial dimensions (i.e., width and height) for the
158     # current octave, then update the dimensions list
159     size = [int(d / (OCTAVE_SCALE ** i)) for d in dims]
160     octaveDims.append(size)
161
162 # reverse the octave dimensions list so that the *smallest*
163 # dimensions are at the *front* of the list
164 octaveDims = octaveDims[::-1]
```

Line 145 calls our `preprocess` helper method to load our input image and apply any preprocessing. **Line 146** extracts the spatial dimensions (height and width) from the `.shape` tuple (we'll need these values when constructing the octaves and resizing the image).

In order to perform deep dreaming, we need to build a list of dimensions we are going to resize our input image to. Each resized image is called an *octave*, therefore the dimensions of each octave

can be stored in a list named `octaveDims`. We initialize `octaveDims` with the current `dims` of the input image.

To generate all other octave dimension we loop over the `NUM_OCTAVE` to generate (**Line 156**). We can then compute the spatial dimensions (where each successive value is smaller than the rest) and append them to the `octaveDims` list (**Lines 159 and 160**). We then reverse the `octaveDims` list on **Line 164** so the *smallest* dimensions are at the *front* of the list.

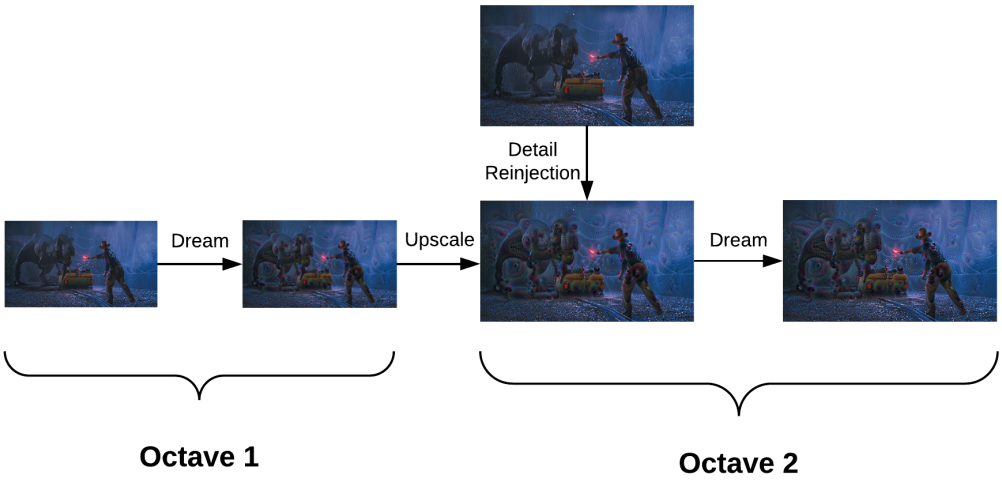


Figure 14.2: The actual DeepDream process involves (1) applying deep dreaming to in input image, (2) utilizing detail reinjnection, and (3) upscaling the image. We apply this process across a number of scales of an called octaves.

The actual DeepDream process can be visualized in Figure 14.2 (which is a reproduction and inspired by Figure 8.4 of Chollet [63]). On the *left* we have a small version of our input image. We then apply gradient ascent, resulting in a small hallucination of our input image.

To avoid blurry or pixelated images (called *loss of detail*) after each successive step, we apply *detail reinjection* — this process is the difference between (1) the shrunk image that has been *upscaled* to the current octave and (2) the original image that has been *downscaled* to the current octave. We then *add* the result of this subtraction to our dream. The process repeats until we have applied our desired number of octaves.

To see this method in code, we first need to create a clone of our input `image` and then resize the input `image` to the first set of spatial dimensions in `octaveDims` (i.e., the *smallest* width and height):

```
166 # clone the original image and then create a resized input image that
167 # matches the smallest dimensions
168 orig = np.copy(image)
169 shrunk = resize_image(image, octaveDims[0])
```

The process detailed in Figure 14.2 can be found in the code below:

```
171 # loop over the octave dimensions from smallest to largest
172 for (o, size) in enumerate(octaveDims):
173     # resize the image and then apply gradient ascent
174     print("[INFO] starting octave {}...".format(o))
175     image = resize_image(image, size)
```

```
176         image = gradient_ascent(image, iters=NUM_ITER, alpha=ALPHA,
177             maxLoss=MAX_LOSS)
178
179         # to compute the lost detail we need two images: (1) the shrunk
180         # image that has been upscaled to the current octave and (2) the
181         # original image that has been downscaled to the current octave
182         upscaled = resize_image(shrunk, size)
183         downscaled = resize_image(orig, size)
184
185         # the lost detail is computed via a simple subtraction which we
186         # immediately back in to the image we applied gradient ascent to
187         lost = downscaled - upscaled
188         image += lost
189
190         # make the original image be the new shrunk image so we can
191         # repeat the process
192         shrunk = resize_image(orig, size)
```

On **Line 172** we loop over the octave dimensions, from smallest to largest. **Line 175** scales up our dream image to the target dimensions.

Lines 176 and 177 apply gradient descent — combined with our `for` loop on **Line 172** you can see we have constructed a *feedback loop* where our dream is iteratively fed back through the network.

Lines 182-188 handle reinjection of the lost detail. In order to inject the lost detail we need to compute the shrunk image that has been upscaled to the current octave (**Line 182**) along with the original image that has been downscaled to the current octave (**Line 183**).

Computing the lost detail is a simple subtraction between the `downscaled` and `upscaled` image (**Line 187**). To inject this lost detail we add `lost` to our `image` (**Line 188**). Finally, we need to generate a new `shrunk` image so we can repeat the process (**Line 192**).

The final step in the DeepDream algorithm is to de-process our output dream and write it to disk:

```
194 # deprocess our dream and save it to disk
195 image = deprocess(image)
196 cv2.imwrite(args["output"], image)
```

14.3 DeepDream Results

Figure 14.3: **Left:** The original input image from a famous scene from the movie *Jurassic Park*. **Right:** The output deep dream image. Image and still credit to *Jurassic Park* Universal Pictures.

To apply our implementation of DeepDream, open up a terminal and execute the following command:

```
$ python deep_dream.py --image jp.jpg --output dream.png
[INFO] starting octave 0...
[INFO] Loss at 0: 0.8034266233444214
...
[INFO] Loss at 47: 3.2595596313476562
[INFO] Loss at 48: 3.285323143005371
[INFO] Loss at 49: 3.310485363006592
```

The output of running our script can be seen in Figure 14.3. On the *left*, we have our input image — a famous scene from the movie *Jurassic Park*. On the *right*, we have our output image. Notice how the input image has been transformed with geometric shapes and objects added to the image, but without significantly changing the actual content of the image.

You can create different results by mixing together different layers and corresponding weights. It is a fun and worthwhile exercise to write a Python script to perform a "random walk" amongst both the layers of a CNN and layer weights.

To perform this random walk, I would suggest randomly choosing (1) a set of layers for the CNN to deep dream on and (2) a set of weights for each layer. Perform this random sampling N times, logging your choices of layers and weights, and let the process run overnight. When you wake up in the morning, see which set of layers and weights produced the most interesting results.

14.4 Summary

In this chapter we learned how to implement the DeepDream algorithm, an viral internet sensation following the publication of Mordvintsev et al.'s work on the Google Research blog [55].

While DeepDream is fun to implement and play with, the novelty quickly wears off after you spend a few hours with it. From there you find yourself wishing for a way to transfer the style from one image to another... which is *exactly* the topic of our next chapter.

15. Neural Style Transfer

Our previous chapter discussed the concept of "deep dreaming" and how we can run a CNN in reverse to generate artistic, and in some cases, hallucinogenic-style images. This deep dreaming algorithm sparked brand new online communities, all generating, sharing, and even competing to create the best deep dreams.

However, the excitement over deep dreaming pales in comparison to *neural style transfer*. The neural style transfer algorithm was first introduced by Gatys et al. in their 2015 paper, *A Neural Algorithm of Artistic Style* [64].

While it's possible to apply *guided deep dreaming* [59] to "guide" your input images to visualize features of a content image, the results were still similar to normal deep dreaming — due to the abundance of dogs, cats, and other animals in the ImageNet dataset (which is what most CNNs were trained on prior to applying deep dreaming), the hype surrounding the algorithm reached an asymptote.

Instead, what users *really* wanted was a method to take:

1. An input image (i.e., a *content image*)
2. A *style image*
3. And apply the *style image* to their *content image*, thereby generating more interesting images and works of art

The work of Gatys et al. made this method possible. Since then, neural style transfer has gone under many iterations and refinements, been built into smartphone apps, and even resulted in style transfer works being sold as works of art.

In this chapter we'll be focusing on the original Gatys et al. implementation — this background will give you a strong foundation to explore other neural style transfer algorithms (and potentially even create your own).

15.1 The Neural Style Transfer Algorithm

The neural style transfer algorithm consists of taking the *style* of one image and then applying it to the *content* of another.

Figure 15.1: **Left:** Our input style image is an autumn mountain range, painted by Bob Ross, the creator of *The Joy of Painting*. **Middle:** An architectural drawing of Frank Lloyd Wright's *Fallingwater* (credit: Larry Hunter [65]). **Right:** The output after after applying the style of the middle image to the input content image.

An example of this process can be seen in Figure 15.1. On the *left* we have our content image — a serene autumn mountain range, painted by critically acclaimed painter and TV host Bob Ross. In the *middle* is our style image, a rendered blueprint poster of Frank Lloyd Wright's famous *Fallingwater* house.

And on the *right* is the output of applying the style of the blueprint image to the content of Bob Ross' mountain range (hence the term *neural style transfer*). Notice how we have *retained* the content of the mountains, river, and trees, but have applied the *style* of the blueprint — it's as if Bob Ross put down his paintbrush and instead roughly sketched the mountain scene instead.

The question is, *how do we define a neural network to perform style transfer?* Interestingly, we don't need to "define" an architecture. We don't need special layers in the network. We don't even need to update the weights of the network!

Instead, consider the core component of neural networks and deep learning: *the loss function.* We define a loss function that will enable us to achieve our end goal and then we optimize over the loss function. Therefore, the question isn't *"what neural network do we use?"* but rather *"what loss function do we use?"*

The answer is a three-component loss function, including:

- Content loss
- Style loss
- Total-variation loss

Each component will be computed individually then combined in a single loss function. By minimizing the "meta" loss function we will in turn be jointly minimizing the content, style, and total-variation losses as well.

15.1.1 Content Loss

As we discussed in the *Starter Bundle*, deep learning algorithms are hierarchical learners. Layers earlier in the network capture local information such as edge-like regions and color blobs. Mid-layers build on the earlier layers to recognize corners. Finally, the highest-level layers in the network are able to capture abstract features, including object parts.

Since the higher-level layers of the network capture the most abstract qualities of the image, a good starting point for our content loss would be to examine the *activations* of these higher-level layers.

According to Gatys et al., to build our loss function we need to:

1. Utilize a pre-trained network, typically on ImageNet or similar dataset
2. Select a higher-level layer of the network to serve as our content loss. This is a hyperparameter we can tune to achieve different output transfers, but for most situations you'll want to use a

higher-level layer of the network.
3. Compute the activation of this particular layer for *both* the content image and style image
4. Take the L2-norm between these activations, respectively

Since the higher-level layers of the network capture more abstract qualities of the input image this loss function ensures that our output generated image will at least look somewhat similar to the content image.

15.1.2 Style Loss

Our content loss used only a *single* layer of a CNN but our style loss will use *multiple* layers. The reason for this is we wish to construct a multi-scale representation of style and texture.

Obtaining this multi-scale representation enables us to capture the style at lower-level layers (edge-like regions), mid-level layers (where corners and other structures start to form), and higher-level layers (which include abstract concepts). This process enables us to capture the texture/style of our style image but *not* the global arrangement of objects in the content image.

Trying to build a loss function that captures the style and texture of multiple layers in a CNN may sound like a complex task, but it's actually fairly straightforward with a bit of statistics. In particular, we'll be computing the *correlations* between the activations of layers in our CNN — we can capture the correlations by computing the *Gram matrix* between the layers activations.

A Gram matrix is the inner product of a set of features maps (if you're familiar with Support Vector Machines you have likely already encountered the Gram matrix or even utilized it without knowing.) During training, our goal will be to minimize the loss between the style of our *output image* and the style of our *style image*, thereby forcing the style of the output image to *correlate* with the style of the style image.

15.1.3 Total-variation Loss

The final loss component is our *total-variation loss*. Unlike the previous two loss functions which considered:

1. The output image
2. And either the content or style image

The total-variation loss operates *solely* on the output image (it does not examine either the content image or style image). Total-variation loss was not included in the original Gatys et al. paper but it has been found to generate better, more aesthetically pleasing style transfers by encouraging spatial smoothness across the output image.

15.1.4 Combining the Loss Functions

Our final loss function is the *weighted combination* of all three content loss, style loss, and total-variation loss, respectively. Written in pseudocode (and inspired by Chollet's style transfer example [63]) our loss function would look like:

```
loss = (alpha * D(style(original_image) - style(gen_image))) +
        (beta * D(content(original_image) - content(gen_image))) +
        (gamma * tv(gen_image))
```

The first component is our style loss. We compute the distance `D` between the style feature representations for both the input reference image and output generated image and weight it by some value `alpha`. This component of the loss function ensures that the style of the reference image is mapped to the output image.

The second component handles our content loss. Here we compute the distance between the content feature representations of the original input image and the generated image. We weight this

distance by `beta`. This component ensures that our output image still has similar content to the original input image.

The final component is our total-variation loss which operates over the output image, ensuring spatial continuity. The total-variation loss is weighted by `gamma`.

The final `loss` is the sum of these three weighted loss components. The α, β, and γ values are all hyperparameters we can play with and adjust to generate different style transfers.

15.2 Implementing Neural Style Transfer

Our neural style transfer implementation will consist of two components:

1. A `NeuralStyle` class that encapsulates all logic required to apply neural style transfer to a content image and style image.
2. A Python driver script used to store any parameters, including input image paths, content and style layers, weight values, etc. This script will be responsible for kicking off the neural style transfer process.

15.2.1 Implementing the Style Transfer Driver Script

For the sake of simplicity, we are going to implement the driver script *before* the `NeuralStyle` class. Implementing the driver script script before the `NeuralStyle` class will enable you to see how we supply input image paths, weight values, and layer sets to facilitate the style transfer.

Open up a new file, name it `style_transfer.py`, and insert the following code:

```
# import the necessary packages
from pyimagesearch.nn.conv import NeuralStyle
from keras.applications import VGG19
```

Our first code block handles our imports — we only need two of them here:

1. Our `NeuralStyle` implementation
2. The network we are using to apply neural style transfer. As per the work of Gatys et al., we know that VGG19 tends to produce better style transfers than VGG16.

From there we define our `SETTINGS` dictionary:

```
# initialize the settings dictionary
SETTINGS = {
	# initialize the path to the input (i.e., content) image,
	# style image, and path to the output directory
	"input_path": "inputs/jp.jpg",
	"style_path": "inputs/starry_night.jpg",
	"output_path": "output",

	# define the CNN to be used for style transfer, along with
	# the set of content layer and style layers, respectively
	"net": VGG19,
	"content_layer": "block4_conv2",
	"style_layers": ["block1_conv1", "block2_conv1",
		"block3_conv1", "block4_conv1", "block5_conv1"],

	# store the content, style, and total variation weights,
	# respectively
	"content_weight": 1.0,
	"style_weight": 100.0,
```

```
24         "tv_weight": 10.0,
25
26         # number of iterations
27         "iterations": 50,
28     }
```

This dictionary encapsulates all settings required to run our `NeuralStyle` class. **Lines 9-12** start off by initializing the path to our input image (i.e., the content image), the path to our style image, and then finally the path to our output directory where we'll store the results of applying our neural style transfer algorithm.

Lines 15-18 initialize the CNN we are using for the style transfer (VGG19) along with the `content_layer` and list of `style_layers`.

We then have our content weight, style weight, and total-variation weight on **Lines 22-24**, respectively. **Line 27** controls the number of iterations to apply.

The final code block instantiates our `NeuralStyle` object and starts the transfer process:

```
30 # perform neural style transfer
31 ns = NeuralStyle(SETTINGS)
32 ns.transfer()
```

In the next section we'll execute the `style_transfer.py` script and examine the results of applying neural style transfer.

15.2.2 Implementing the Style Transfer Class

Our implementation of neural style transfer is loosely based on the official implementation François Chollet in the Keras repository [66] and more strongly based on Kevin Zakka's version [67] which wraps the entire algorithm in a single Python class. My main contribution is to provide detailed commentary on the code.

To implement neural style transfer, create a new file named `neuralstyle.py` in the `conv` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
...
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- alexnet.py
|    |    |    |--- deepergooglenet.py
...
|    |    |    |--- neuralstyle.py
...
|    |    |    |--- shallownet.py
...
```

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

```
1 # import the necessary packages
2 from keras.applications.vgg16 import preprocess_input
```

```
3  from keras.preprocessing.image import img_to_array
4  from keras.preprocessing.image import load_img
5  from keras import backend as K
6  from scipy.optimize import fmin_l_bfgs_b
7  import numpy as np
8  import cv2
9  import os
```

Our set of imports look fairly standard here, but you'll notice one import we haven't encountered yet: `fmin_l_bfgs_b`. he `fmin_l_bfgs_b` function is an implementation of the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm used to iteratively minimize unconstrained nonlinear optimization problems.

In our case we'll be using L-BFGS to minimize our neural style transfer loss function. L-BFGS is a common optimization function but is outside the scope of this book. If you're interested in learning more about L-BFGS, refer to Liu and Nocedal [68] along with this excellent introduction article from Aria Haghighi [69].

Let's move on to the actual `NeuralStyle` class:

```
11 class NeuralStyle:
12     def __init__(self, settings):
13         # store the settings dictionary
14         self.S = settings
15
16         # grab the dimensions of the input image
17         (w, h) = load_img(self.S["input_path"]).size
18         self.dims = (h, w)
19
20         # load content image and style images, forcing the dimensions
21         # of our input image
22         self.content = self.preprocess(settings["input_path"])
23         self.style = self.preprocess(settings["style_path"])
24         self.content = K.variable(self.content)
25         self.style = K.variable(self.style)
```

On **Line 14** we store our `settings` dictionary which is assumed to be the same `SETTINGS` from our driver script.

Lines 17 and 18 load our input image (i.e., the content image) from disk and grab the spatial dimensions — we'll need these dimensions when preprocessing an image, de-processing an image, and computing our loss.

From there we load our `content` and `style` images from disk and preprocess them (**Lines 22 and 23**), forcing the spatial dimensions of the `content` image on the `style` image to ensure both images are the same size. **Lines 24 and 25** instantiate Keras variables from the input images.

Next, let's prepare our `input` and `output` tensors:

```
27         # allocate memory of our output image, then combine the
28         # content, style, and output into a single tensor so they can
29         # be fed through the network
30         self.output = K.placeholder((1, self.dims[0],
31             self.dims[1], 3))
32         self.input = K.concatenate([self.content, self.style,
33             self.output], axis=0)
```

Lines 30 and 31 initialize our `output` image (i.e., where the final style transfer image will be stored). We assume channels-last ordering here so if you are using channels-first ordering you'll need to modify the code.

The `input` tensor is then defined on **Lines 32 and 33** consisting of all three `content`, `style`, and `output` images concatenated together. Our goal will be to minimize our style loss, content loss, and total-variation loss based on this `input` tensor.

From there we can load our model from disk:

```
35          # load our model from disk
36          print("[INFO] loading network...")
37          self.model = self.S["net"](weights="imagenet",
38               include_top=False, input_tensor=self.input)
39
40          # build a dictionary that maps the *name* of each layer
41          # inside the network to the actual layer *output*
42          layerMap = {l.name: l.output for l in self.model.layers}
43
44          # extract features from the content layer, then extract the
45          # activations from the style image (index 0) and the output
46          # image (index 2) -- these will serve as our style features
47          # and output features from the *content* layer
48          contentFeatures = layerMap[self.S["content_layer"]]
49          styleFeatures = contentFeatures[0, :, :, :]
50          outputFeatures = contentFeatures[2, :, :, :]
51
52          # compute the feature reconstruction loss, weighting it
53          # appropriately
54          contentLoss = self.featureReconLoss(styleFeatures,
55               outputFeatures)
56          contentLoss *= self.S["content_weight"]
```

Line 42 builds a dictionary which maps the name of a layer in our CNN to the corresponding output of the layer — we'll need this mapping to compute our loss and build the style transfer. **Lines 48-50** extracts the "features" (i.e., activations) from the *single* content layer.

Given our `styleFeatures` and `outputFeatures` for the content layer we can compute the *feature reconstruction loss*, or more simply, the *style loss* (**Lines 54-56**). We'll be implementing `featureReconloss` later in this section, but for the time being treat it as a blackbox function that computes style loss. Also take note of **Line 56** and how we weight the `contentLoss` accordingly.

Let's move on to computing the style loss:

```
58          # initialize our style loss along with the value used to
59          # weight each style layer (in proportion to the total number
60          # of style layers
61          styleLoss = K.variable(0.0)
62          weight = 1.0 / len(self.S["style_layers"])
63
64          # loop over the style layers
65          for layer in self.S["style_layers"]:
66               # grab the current style layer and use it to extract the
67               # style features and output features from the *style
68               # layer*
69               styleOutput = layerMap[layer]
```

```
70                styleFeatures = styleOutput[1, :, :, :]
71                outputFeatures = styleOutput[2, :, :, :]
72
73                # compute the style reconstruction loss as we go
74                T = self.styleReconLoss(styleFeatures, outputFeatures)
75                styleLoss += (weight * T)
76
77            # finish computing the style loss, compute the total
78            # variational loss, and then compute the total loss that
79            # combines all three
80            styleLoss *= self.S["style_weight"]
81            tvLoss = self.S["tv_weight"] * self.tvLoss(self.output)
82            totalLoss = contentLoss + styleLoss + tvLoss
```

Line 61 initializes our `styleLoss` while **Line 62** initializes the `weight` used to weight our `styleLoss`, proportional to the total number of style layers we are using for the transfer.

Line 65 starts looping over each of the style layers. For each `layer`, we first use the `layerMap` to grab the `styleOutput`. From the `styleOutput` we can extract the `styleFeatures` (index 1) and the `outputFeatures` (index 2).

Based on both the `styleFeatures` and `outputFeatures` we compute the *style reconstruction loss*, or more simply, *the style loss*. We will be implementing the `styleReconLoss` function later in this section, but again, treat it as a blackbox function until we get there.

Line 81 computes our total-variation loss (which operates solely on the `output`) while **Line 82** computes the final `totalLoss`, combining `contentLoss`, `styleLoss`, and `tvLoss` together.

The last step in our initialize is to compute our gradients and initialize the function we'll be apply L-BFGS to:

```
84            # compute the gradients out of the output image with respect
85            # to loss
86            grads = K.gradients(totalLoss, self.output)
87            outputs = [totalLoss]
88            outputs += grads
89
90            # the implementation of L-BFGS we will be using requires that
91            # our loss and gradients be *two separate functions* so here
92            # we create a Keras function that can compute both the loss
93            # and gradients together and then return each separately
94            # using two different class methods
95            self.lossAndGrads = K.function([self.output], outputs)
```

Lines 86-88 compute the gradients of our `output` image with respect to our `totalLoss`.

Line 95 then initializes the `lossAndGrads` function which accepts our `output` tensor and `outputs` gradients as arguments. Due to how the implementation of L-BFGS works, we need two separate functions for (1) the loss and (2) the gradients. This Keras function will return both and enable us to easily exact the loss and/or gradients deepening on which we are interested in.

Our next code block creates two functions used to `preprocess` and `deprocess` images, respectively:

```
97        def preprocess(self, p):
98            # load the input image (while resizing it to the desired
99            # dimensions) and preprocess it
```

```
100             image = load_img(p, target_size=self.dims)
101             image = img_to_array(image)
102             image = np.expand_dims(image, axis=0)
103             image = preprocess_input(image)
104
105             # return the preprocessed image
106             return image
107
108         def deprocess(self, image):
109             # reshape the image, then reverse the zero-centering by
110             # *adding* back in the mean values across the ImageNet
111             # training set
112             image = image.reshape((self.dims[0], self.dims[1], 3))
113             image[:, :, 0] += 103.939
114             image[:, :, 1] += 116.779
115             image[:, :, 2] += 123.680
116
117             # clip any values falling outside the range [0, 255] and
118             # convert the image to an unsigned 8-bit integer
119             image = np.clip(image, 0, 255).astype("uint8")
120
121             # return the deprocessed image
122             return image
```

The `preprocess` function (**Lines 97-106**) loads an input image from disk and preprocesses it so it can be passed through a Keras network pre-trained on the ImageNet dataset. The `deprocess` function on **Lines 108-122** takes the output of our neural stylet transfer algorithm and *de-processes it* so we can save the image to disk.

We're de-processing the image based on the reverse of the pre-processing used for VGG16 and VGG19. If you're *not* using either VGG16 or VGG19 you'll need to modify the `deprocess` function to perform the inverse of the pre-processing required by your network.

The next function `gramMat`, is responsible for computing the Gram matrix:

```
124         def gramMat(self, X):
125             # the gram matrix is the dot product between the input
126             # vectors and their respective transpose
127             features = K.permute_dimensions(X, (2, 0, 1))
128             features = K.batch_flatten(features)
129             features = K.dot(features, K.transpose(features))
130
131             # return the gram matrix
132             return features
```

The Gram matrix is the dot product between the input vectors and their transpose. We'll need the Gram matrix when computing the style loss.

From here we can define `featureReconLoss` which is responsible for computing the *content loss:*

```
134         def featureReconLoss(self, styleFeatures, outputFeatures):
135             # the feature reconstruction loss is the squared error
```

```
136             # between the style features and output output features
137             return K.sum(K.square(outputFeatures - styleFeatures))
```

The content-loss is the L2 norm (sum of squared differences) between the features of our input image and the features of the target, output image. By minimizing the L2 norm of these activations we can force our output image to have similar structural content (but not necessarily similar style) as our input image.

Next, we need to create the `styleReconLoss` function:

```
139     def styleReconLoss(self, styleFeatures, outputFeatures):
140             # compute the style reconstruction loss where A is the gram
141             # matrix for the style image and G is the gram matrix for the
142             # generated image
143             A = self.gramMat(styleFeatures)
144             G = self.gramMat(outputFeatures)
145 
146             # compute the scaling factor of the style loss, then finish
147             # computing the style reconstruction loss
148             scale = 1.0 / float((2 * 3 * self.dims[0] * self.dims[1]) ** 2)
149             loss = scale * K.sum(K.square(G - A))
150 
151             # return the style reconstruction loss
152             return loss
```

To compute the style loss we need to first compute the Gram matrix for both the `styleFeatures` of the input style image and the `outputFeatures` of our output image (**Lines 143 and 144**).

Line 148 computes the scaling factor of the style loss based on the input dimensions of the image. If you're curious how the scaling factors are derived, please refer to the *"Methods"* section of Gatys et al. [64] **Line 149** computes the L2 norm (sum of squared differences) between the Gram matrices and scales appropriately.

We need one final loss function — our total-variation loss:

```
154     def tvLoss(self, X):
155             # the total variational loss encourages spatial smoothness in
156             # the output page -- here we avoid border pixels to avoid
157             # artifacts
158             (h, w) = self.dims
159             A = K.square(X[:, :h - 1, :w - 1, :] - X[:, 1:, :w - 1, :])
160             B = K.square(X[:, :h - 1, :w - 1, :] - X[:, :h - 1, 1:, :])
161             loss = K.sum(K.pow(A + B, 1.25))
162 
163             # return the total variational loss
164             return loss
```

This loss function was not included in the original Gatys et al. paper, but nearly all neural style transfer implementations include this additional loss function as it encourages spatial smoothness throughout the output image). We utilize array slicing here along the borders of the image to avoid border artifacts.

We can now define the `transfer` function which is responsible for jointly minimizing our content loss, style loss, and total variation loss:

```
166     def transfer(self, maxEvals=20):
167         # generate a random noise image that will serve as a
168         # placeholder array, slowly modified as we run L-BFGS to
169         # apply style transfer
170         X = np.random.uniform(0, 255,
171             (1, self.dims[0], self.dims[1], 3)) - 128
172 
173         # start looping over the desired number of iterations
174         for i in range(0, self.S["iterations"]):
175             # run L-BFGS over the pixels in our generated image to
176             # minimize the neural style loss
177             print("[INFO] starting iteration {} of {}...".format(
178                 i + 1, self.S["iterations"]))
179             (X, loss, _) = fmin_l_bfgs_b(self.loss, X.flatten(),
180                 fprime=self.grads, maxfun=maxEvals)
181             print("[INFO] end of iteration {}, loss: {:.4e}".format(
182                 i + 1, loss))
183 
184             # deprocess the generated image and write it to disk
185             image = self.deprocess(X.copy())
186             p = os.path.sep.join([self.S["output_path"],
187                 "iter_{}.png".format(i)])
188             cv2.imwrite(p, image)
```

Lines 170 and 171 initializes `X`, a randomly generated placeholder NumPy array with the spatial dimensions of our input content image. The `X` variable will be modified during the training process to take on both the content and style of our respective input images.

On **Line 173** we start looping over the desired number of iterations. We then run L-BFGS over the pixels of generated image to minimize the neural style loss (**Lines 179 and 180**). The function we are minimizing is our `loss`. The gradients of `loss` are specified by `fprime`, our `grads` function. We'll be implementing both `loss` and `grads` in our next code block. Finally, we save the de-processed image at each iteration (**Lines 186-188**).

Our final code block handles defining two helper functions, `loss` and grads':

```
190     def loss(self, X):
191         # extract the loss value
192         X = X.reshape((1, self.dims[0], self.dims[1], 3))
193         lossValue = self.lossAndGrads([X])[0]
194 
195         # return the loss
196         return lossValue
197 
198     def grads(self, X):
199         # compute the loss and gradients
200         X = X.reshape((1, self.dims[0], self.dims[1], 3))
201         output = self.lossAndGrads([X])
202 
203         # extract and return the gradient values
204         return output[1].flatten().astype("float64")
```

As we saw, our `fmin_l_bfgs_b` function requires *two* functions when applying optimization:

1. A loss function

2. A *separate* function that returns the gradients of the loss

Lines 192-196 utilize `lossAndGrads` to grab the `lossValue` and return it (index `0`). **Lines 200-204** then extract the gradients (index `1`) from `lossAndGrads` and return it.

Specifying the indexes here is important as we defined the Keras function, `lossAndGradients` in our constructor to compute both the loss and gradients — specifying the index enables us to extract each, respectively.

15.3 Neural Style Transfer Results

We are now ready to apply neural style transfer to our own input images. Open up `style_transfer.py` and update the `SETTINGS` to include:

```
"input_path": "inputs/jp.jpg",
"style_path": "inputs/mcescher.jpg",
...
"content_weight": 1.0,
"style_weight": 100.0,
"tv_weight": 10.0,
```

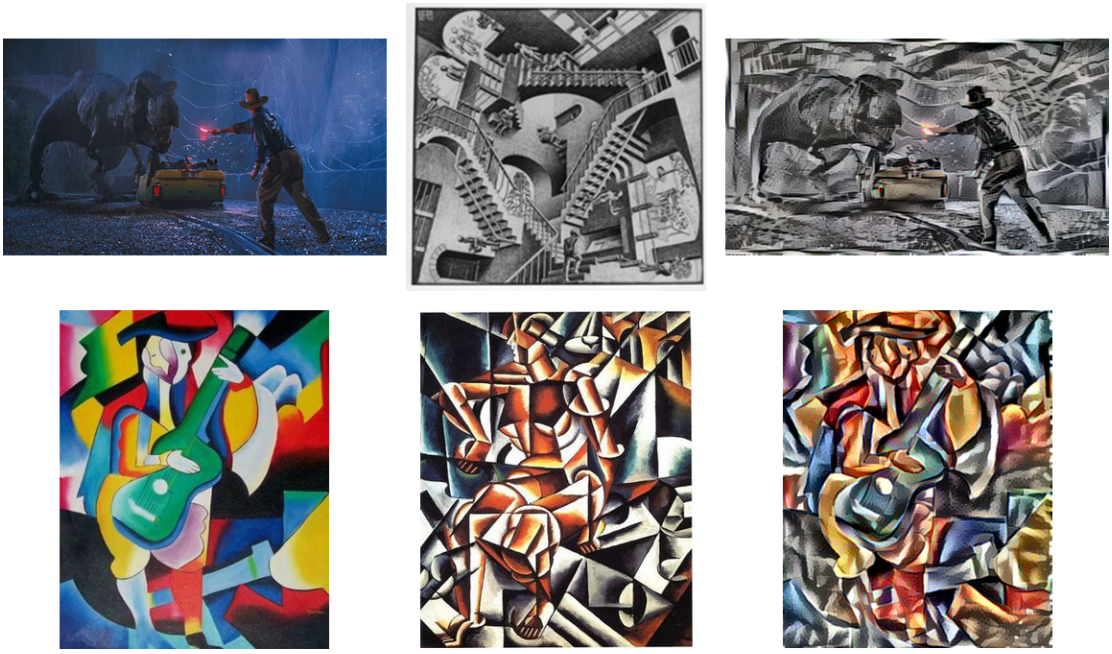


Figure 15.2: **Top:** Applying neural style transfer using an iconic scene from the movie *Jurassic Park* as the input and MC Escher as the style. (Image and still credit to *Jurassic Park* Universal Pictures.) **Bottom:** Pablo Piccaso is used as the content image and Popova as the style.

And then execute the `style_transfer.py` script:

```
$ python style_transfer.py
```

The results of applying our neural style transfer algorithm can be seen in Figure 15.2 (*top*). Here we have used an iconic scene from the movie *Jurassic Park* as the input content image. The

style image is MC Escher' `Relativity lattice`. The output image maintains the content of the *Jurassic Park* still but has geometrical shapes similar to MC Escher's work.

Let's try another style transfer, but before we do so, let's update the `SETTINGS` again:

```
"input_path": "inputs/pablo_picasso.jpg",
"style_path": "inputs/popova_air_man_space.jpg",
...
"content_weight": 1.0,
"style_weight": 100.0,
"tv_weight": 100.0,
```

Here we are setting the style weight to be two orders of magnitude larger than the content weight, implying that we are willing to sacrifice a bit of the content for a more artistic style transfer. We will also increase our total variation weight to help ensure spatial smoothness. The result of these updated `SETTINGS` can be seen in Figure 15.2 (*bottom*).

Some of the best style transfers can be generated by doing random walks through the parameter space. This process involves randomly generating values for each of the loss weights. When I perform neural style transfer I typically define a set of parameters to explore, start the experiment, and let the script run overnight. The next morning I go through the results I like the best and then double-down on the parameter ranges that produced the most aesthetically appealing results.

15.4 Summary

In this chapter we discussed the neural style transfer algorithm by Gatys et al. [64], implemented neural style transfer using Python and Keras, and explored varying parameters for content weight, style weight, and total-variation weight.

The neural style transfer algorithm will work best with:

1. Content images that do not require high levels of detail to be either visually appealing or recognizable.
2. Style images that contain a lot of texture. "Flat" style images will not produce aesthetically appealing results.

Finally, be sure to explore your parameter space with applying the neural style transfer algorithm — vary your content weight, style weight, and total-variation weight will give you different results. I suggest creating a Python script that iteratively runs 10-15 experiments with varying parameters so you can get a sense of what parameters (or maybe none at all) will generate aesthetically pleasing results for both your content image and style image.

To help you get started running your own experiments that *automatically* varies parameters, I have included a special script named `generate_examples.py` in the downloads associated with this book. Be sure to refer to this script as a starting point for your own experiments.

16. Generative Adversarial Networks (GANs)

Generative Adversarial Networks were first introduced by Goodfellow et al. in their 2014 paper, *Generative Adversarial Networks* [70]. These networks can be used to generate synthetic (i.e., fake) images that are perceptually near identical to their ground-truth, authentic originals.

In order to generate synthetic images, we make use of *two* neural networks during training:

1. A **generator** that accepts an input vector of randomly generated noise and produces an output "imitation" image that looks similar, if not identical to an authentic image
2. A ***discriminator*** or ***adversary*** which attempts to determine if a given image is an "authentic" or "fake"

By training both of these networks at the same time, one giving feedback to the other, we can learn to generate synthetic images.

Inside this chapter we'll be implementing a variation of Radford et al.'s 2015 paper, *Unsupervised Representation Learning with Deep Convolution Generative Adversarial Networks* — or more simply, **DCGANs** [71]. As we'll find out, training GANs can be a *notoriously* hard task, so we'll be implementing a number of best practices recommended by both Redford et al. [71] and Chollet [63].

16.1 What Are GANs?

The quintessential explanation of GANs typically involves some variant of two people working in collusion to forge a set of documents, replicate a piece of artwork, or print counterfeit money [63] — the counterfeit money printers is my personal favorite. In this example, we have two people:

- Jack, the counterfeit printer (the *generator*)
- Jason, an employee of the U.S. Treasury (which is responsible for printing money in the United States) who specializes in detecting counterfeit money (the *discriminator*)

Jack and Jason have been childhood friends, both growing up without much money in the rough parts of Boston. After much hard work in school, Jason was awarded a college scholarship — Jack was not and over time started to turn towards illegal ways to make money. He wasn't very good, but he knew he could get better with the proper training.

One day, after a few too many pints at a local pub during Thanksgiving holiday, Jason let it slip to Jack that he wasn't happy with his job. He was underpaid. His boss was nasty and spiteful, often yelling and embarrassing Jason in front of other employees. Jason was even thinking of quitting.

Jack saw an opportunity to use Jason's access at the U.S. Treasury to create an elaborate counterfeit printing scheme. Their conspiracy worked like this:

1. Jack, the counterfeit printer, would print fake bills and then mix *both* the fake bills and real money together, then show them to the expert, Jason.
2. Jason would sort through the bills, classifying each bill as "fake" or "authentic", giving feedback to Jack along the way on how he can improve his counterfeit printing.

At first, Jack is doing a pretty poor job at printing counterfeit money. But over time, with Jason's guidance, Jack eventually improves to the point where Jason is no longer able to spot the difference between the bills. By the end of this process, both Jack and Jason have stacks of counterfeit money that can fool most people.

16.1.1 The General Training Procedure

We've discussed what GANs are in terms of an analogy, but what is the actual *procedure* to train them? Most GANs are trained using a six step process.

To start (step 1), we randomly generate a vector (i.e., noise). We pass this noise through our generator which generates an actual image (step 2). We then sample authentic images from our training set and mix them with our synthetic images (step 3).

The next step (step 4) is to train our discriminator using this mixed set. The goal of the discriminator is to correctly label each image as "real" or "fake".

Next, we'll once again generate random noise, but this time we'll purposely label each noise vector as a "real image" (step 5). We'll then train the GAN using the noise vectors and "real image" labels even though they are not actual real images (step 6).

The reason this process works is due to:

1. We have frozen the weights of the discriminator at this stage, implying that the discriminator is not learning when we update the weights of the generator.
2. We're trying to "fool" the discriminator into being unable to determine which images are real vs. synthetic. The feedback from the discriminator will allow the generator to learn how to produce more authentic images.

If you're confused with this process I would continue reading through our implementation in Section 16.2 below — seeing a GAN implemented in Python and then explained makes it easier to understand the process.

16.1.2 Guidelines and Best Practices When Training

GANs are notoriously hard to train due to an *evolving loss landscape*. At each iteration of our algorithm we are:

1. Generating random images and then training the discriminator to correctly distinguish the two
2. Generating additional synthetic images, but this time purposely trying to fool the discriminator
3. Updating the weights of the generator based on the feedback of the discriminator, thereby allowing us to generate more authentic images

From this process you'll notice there are two losses we need to minimize: one loss for the discriminator and a second loss for the generator. And since the loss landscape of the generator can be changed based on the feedback from the discriminator we end up with a dynamic system.

Our goal is to not seek a minimum loss value but instead find some equilibrium between the two [63]. This concept of finding an equilibrium may make sense on paper, but once you try to

implement and train your own GANs you'll find that this is a non-trivial process.

Radford et al. recommends the following architecture guidelines for more stable GANS:

- Replace any pooling layers with strided convolutions (we have seen this concept used in ResNet in Chapter 11)
- Use batch normalization in both the generator and discriminator
- Remove fully-connected layers in deeper networks
- Use ReLU in the generator except for the final layer which will utilize *tanh*
- Use Leaky ReLU in the discriminator

François Chollet then provided additional recommendations in his book [63]:

1. Sample random vectors from a *normal distribution* (i.e., Gaussian distribution) rather than a *uniform distribution*
2. Add dropout to the discriminator
3. Add noise to the class labels when training the discriminator
4. To reduce checkerboard pixel artifacts in the output image use a kernel size that is divisible by the stride when utilizing convolution or transposed convolution in both the generator and discriminator
5. If your adversarial loss rises dramatically while your discriminator loss falls to zero, try reducing the learning rate of the discriminator and increasing the dropout of the discriminator.

Keep in mind that these are all just *heuristics* found to work in a number of situations — we'll be using *some* of techniques suggested by both Radford et al. and Chollet, but not *all* of them. It is possible, and even *probable*, that the techniques listed here will not work on your GANs. Take the time now to set your expectations that you'll likely be running *orders of magnitude more experiments* when tuning the hyperparameters of your GANs as compared to previous experiments in this book.

16.2 Implementing a DCGAN

Now that we've learned about GANs, let's go ahead and implement one. Specifically, we'll be implementing a variant of the Deep Convolutional Generative Adversarial Network (DCGAN) from Radford et al. [71].

Our DCGAN implementation will live in the `conv` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
...
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- alexnet.py
|    |    |    |--- dcgan.py
...
|    |    |    |--- deepergooglenet.py
...
|    |    |    |--- shallownet.py
...
```

Open up the `dcgan.py` file and insert the following code:

```
1 # import the necessary packages
2 from keras.models import Sequential
```

```
3  from keras.layers.normalization import BatchNormalization
4  from keras.layers.convolutional import Conv2DTranspose
5  from keras.layers.convolutional import Conv2D
6  from keras.layers.advanced_activations import LeakyReLU
7  from keras.layers.core import Activation
8  from keras.layers.core import Flatten
9  from keras.layers.core import Dense
10 from keras.layers.core import Reshape
```

Lines 2-10 handle importing our required packages. All of these classes should look familiar to you with the exception of `Conv2DTranspose`. Transposed convolution layers, sometimes referred to as *fractionally-strided convolution* or (incorrectly) *deconvolution*, are used when we need a transform going in the *opposite* direction of a normal convolution [72].

The generator of our GAN will accept an input N-dimensional vector (i.e., a *list* of numbers, but a *volume* like an image) and then transform this N-dimensional vector into an output image. This process implies that we need to *reshape* and then *upscale* this vector into a volume as it passes through our network — to accomplish this reshaping and upscaling we'll need transposed convolution.

We can thus look at transposed convolution as method to:

1. Accept an input volume from a previous layer in the network
2. Produce an output volume that is *larger* than the input volume
3. Maintain a connectivity pattern between the input and output

In essence, our transposed convolution layer will reconstruct our target spatial resolution and perform a normal convolution operation, utilizing fancy zero-padding schemes to ensure our output spatial dimensions are met.

To learn more about transposed convolution take a look at the *Convolution arithmetic tutorial* inside the Theano documentation [72] along with *An Introduction to different Types of Convolutions in Deep Learning* by Paul-Louis Pröve [73].

Let's move on to the `DCGAN` class:

```
12 class DCGAN:
13     @staticmethod
14     def build_generator(dim, depth, channels=1, inputDim=100,
15         outputDim=512):
16         # initialize the model along with the input shape to be
17         # "channels last" and the channels dimension itself
18         model = Sequential()
19         inputShape = (dim, dim, depth)
20         chanDim = -1
```

Here we define the `build_generator` function inside `DCGAN`. The `build_generator` accepts a number of arguments:

- `dim`: The target spatial dimensions (width and height) of the generator after reshaping.
- `depth`: The target depth of the volume after reshaping.
- `channels`: The number of channels in the output volume from the generator (i.e., `1` for grayscale images and `3` for RGB images).
- `inputDim`: Dimensionality of the randomly generated input vector to the generator.
- `outputDim`: Dimensionality of the output fully-connected layer from the randomly generated input vector.

The usage of these parameters will become more clear as we define the body of the network in the next code block. **Line 19** defines the `inputShape` of the volume after we reshape it from the

fully-connected layer (which we'll review in the next code block). For simplicity, we'll assume we're using channels-last ordering (**Line 20**). You can update this line if you are using channels-first ordering as we discuss in previous example in this book.

Below we can find the body of the generator network:

```
22            # first set of FC => RELU => BN layers
23            model.add(Dense(input_dim=inputDim, units=outputDim))
24            model.add(Activation("relu"))
25            model.add(BatchNormalization())
26
27            # second set of FC => RELU => BN layers, this time preparing
28            # the number of FC nodes to be reshaped into a volume
29            model.add(Dense(dim * dim * depth))
30            model.add(Activation("relu"))
31            model.add(BatchNormalization())
32
33            # reshape the output of the previous layer set, upsample +
34            # apply a transposed convolution, RELU, and BN
35            model.add(Reshape(inputShape))
36            model.add(Conv2DTranspose(32, (5, 5), strides=(2, 2),
37                 padding="same"))
38            model.add(Activation("relu"))
39            model.add(BatchNormalization(axis=chanDim))
40
41            # apply another upsample and transposed convolution, but
42            # this time output the TANH activation
43            model.add(Conv2DTranspose(channels, (5, 5), strides=(2, 2),
44                 padding="same"))
45            model.add(Activation("tanh"))
46
47            # return the generator model
48            return model
```

Lines 23-25 define our first set of `FC => RELU => BN` layers — applying batch normalization to stabilize GAN training is a guideline from Radford et al.

Notice how our `FC` layer will have an input dimension of `inputDim` (the randomly generated input vector) and then an output dimensionality of `outputDim`. Typically, `outputDim` will be larger than `inputDim`.

Lines 29-31 apply a second set of `FC => RELU => BN` layers, but this time we prepare the number of nodes in the `FC` layer to equal the number of units in `inputShape`. Even though we are still utilizing a flattened representation, we need to ensure the output of this `FC` layer can be reshaped to our target volume size (i.e., `inputShape`).

The actual reshaping takes place on **Line 35** — a call to `Reshape` while supplying `inputShape` allows us to create a 3D volume from the fully-connected layer on **Line 29**. Again, this reshaping is only possible due to the fact that the number of output nodes `FC` layer matches the target `inputShape`.

We now reach an important guideline when training your own GANs:

1. To *increase* spatial resolution, use a transposed convolution with a stride > 1.
2. To create a deeper GAN *without* increasing spatial resolution you can use either a standard convolution or transposed convolution.

Here our transposed convolution layer is learning 32 filters, each of which are 5×5, while applying 2×2 stride — since our stride is > 1 we can increase our spatial resolution.

Another transposed convolution takes place on **Lines 43 and 44**, again increasing the spatial resolution, but taking care to ensure the number of filters learned is equal to the target `depth`.

We then apply a *tanh* activation per the recommendation of Radford et al. The model is returned to the calling function on **Line 47**.

Assuming `dim=7`, `depth=1`, `inputDim=100`, and `outputDim=512` (as we will use when training our DCGAN on MNIST in Section 16.3), I have included the model summary below:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 512)               51712
_________________________________________________________________
activation_1 (Activation)    (None, 512)               0
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048
_________________________________________________________________
dense_2 (Dense)              (None, 3136)              1608768
_________________________________________________________________
activation_2 (Activation)    (None, 3136)              0
_________________________________________________________________
batch_normalization_2 (Batch (None, 3136)              12544
_________________________________________________________________
reshape_1 (Reshape)          (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 14, 14, 32)        51232
_________________________________________________________________
activation_3 (Activation)    (None, 14, 14, 32)        0
_________________________________________________________________
batch_normalization_3 (Batch (None, 14, 14, 32)        128
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 28, 28, 1)         801
_________________________________________________________________
activation_4 (Activation)    (None, 28, 28, 1)         0
=================================================================
```

Our model will accept an input vector that is 100-d, then transform it to 512-d via a `FC` layer.

We then add a second `FC` layer, this one with $7 \times 7 \times 64 = 3,136$ nodes. We reshape these $3,136$ nodes into a 3D volume with shape $7 \times 7 \times 64$ — this reshaping is only possible since our previous `FC` layer matches the number of nodes in reshaped volume. Applying a transposed convolution with a 2×2 stride increases our spatial dimensions from 14×14.

A second transposed convolution (again with a stride of 2×2) again increases our spatial resolution to 28×28 with 1 channel, which is the *exact* dimensions of our input images in the MNIST dataset.

When implementing your own GANs, make sure the spatial dimensions of the output volume match the spatial dimensions of your input images. Use transposed convolution to increase the spatial dimensions of the volumes in the generator. I would recommend using `model.summary()` often to help you debug the spatial dimensions.

The discriminator model is substantially more simplistic, similar to basic CNNs we have implemented earlier in this book. Keep in mind that while the generator is intended to create synthetic images, the discriminator is used to classify whether an input image is real or fake.

Let's take a look at the discriminator now:

```
50      @staticmethod
51      def build_discriminator(width, height, depth, alpha=0.2):
52          # initialize the model along with the input shape to be
53          # "channels last"
54          model = Sequential()
55          inputShape = (height, width, depth)
56
57          # first set of CONV => RELU layers
58          model.add(Conv2D(32, (5, 5), padding="same", strides=(2, 2),
59              input_shape=inputShape))
60          model.add(LeakyReLU(alpha=alpha))
61
62          # second set of CONV => RELU layers
63          model.add(Conv2D(64, (5, 5), padding="same", strides=(2, 2)))
64          model.add(LeakyReLU(alpha=alpha))
65
66          # first (and only) set of FC => RELU layers
67          model.add(Flatten())
68          model.add(Dense(512))
69          model.add(LeakyReLU(alpha=alpha))
70
71          # sigmoid layer outputting a single value
72          model.add(Dense(1))
73          model.add(Activation("sigmoid"))
74
75          # return the discriminator model
76          return model
```

As we can see, this network is simple and straightforward. We first learn 32, 5×5 filters, followed by a second `CONV` layer, this one learning 64, 5×5 filters. We only have a single `FC` layer here, this one with 512 nodes.

All layers utilize a Leaky ReLU activation to stabilizing training, *except* for the final activation which is a sigmoid. We use a sigmoid here to capture the probability of whether the input image is real or synthetic.

16.3 Training a DCGAN

Now that we have implemented our DCGAN architecture, let's train it on the MNIST dataset to generate fake digits. By the end of the training process we will be unable to identify *real* images from *synthetic* ones.

Open up a new file, name it `dcgan_mnist.py`, and insert the following code:

```
1  # import the necessary packages
2  from pyimagesearch.nn.conv import DCGAN
3  from keras.models import Model
4  from keras.layers import Input
5  from keras.optimizers import Adam
6  from keras.datasets import mnist
7  from sklearn.utils import shuffle
8  from imutils import build_montages
9  import numpy as np
10 import argparse
```

```
11 import cv2
12 import os
```

We start off by importing our required Python packages. We've seen most of these packages before, but take notice of `DCGAN` (**Line 2**), which is our implementation from the previous section, along with `build_montages` on **Line 8** — this is a convenience function that will enable us to easily build a montage of images and display them to our a screen as a single image.

Let's move on to parsing our command line arguments:

```
14 # construct the argument parse and parse the arguments
15 ap = argparse.ArgumentParser()
16 ap.add_argument("-o", "--output", required=True,
17     help="path to output directory")
18 ap.add_argument("-e", "--epochs", type=int, default=50,
19     help="# epochs to train for")
20 ap.add_argument("-b", "--batch-size", type=int, default=128,
21     help="batch size for training")
22 args = vars(ap.parse_args())
23
24 # store the epochs and batch size in convenience variables
25 NUM_EPOCHS = args["epochs"]
26 BATCH_SIZE = args["batch_size"]
```

We'll require a single command line argument for this script, `--output`, which is the path to where we'll store montages of generated images so we can visualize the training process. We can optionally supply `--epochs`, the total number of epochs to train for, and `--batch-size`, used to control the batch size when training. We store both the number of epochs and batch size in convenience variables on **Lines 25 and 26**.

We can now load the MNIST dataset:

```
28 # load the MNIST dataset and stack the training and testing data
29 # points so we have additional training data
30 print("[INFO] loading MNIST dataset...")
31 ((trainX, _), (testX, _)) = mnist.load_data()
32 trainImages = np.concatenate([trainX, testX])
33
34 # add in an extra dimension for the channel and scale the images
35 # into the range [-1, 1] (which is the range of the tanh
36 # function)
37 trainImages = np.expand_dims(trainImages, axis=-1)
38 trainImages = (trainImages.astype("float") - 127.5) / 127.5
```

Line 31 loads the MNIST dataset from disk. We ignore the class labels here since we do not need them — we're only interested in the actual pixel data.

Furthermore, there is not a concept of a "test set" for GANs. Our goal when training isn't minimal loss or high accuracy. Instead, we are seeking an equilibrium between the generator and the discriminator. To help us obtain this equilibrium we can *combine* both the training and testing images (**Line 32**) to give us additional training data.

Lines 37 and 38 prepare our data for training by scaling it to the range $[-1, 1]$, the output range of the *tanh* activation function.

Let's now initialize our generator and discriminator:

```
40 # build the generator
41 print("[INFO] building generator...")
42 gen = DCGAN.build_generator(7, 64, channels=1)
43
44 # build the discriminator
45 print("[INFO] building discriminator...")
46 disc = DCGAN.build_discriminator(28, 28, 1)
47 discOpt = Adam(lr=0.0002, beta_1=0.5, decay=0.0002 / NUM_EPOCHS)
48 disc.compile(loss="binary_crossentropy", optimizer=discOpt)
```

Line 42 initializes the generator that will transform the input random vector to a volume of shape $7 \times 7 \times 64$-channel map. **Lines 46 and 47** builds the discriminator and then compiles it using the Adam optimizer and binary cross-entropy loss.

Keep in mind that we are using *binary* cross-entropy loss here as (1) our discriminator has a sigmoid activation function that will (2) return a probability indicating whether the input image is real vs. fake. Since there are only two class labels (real and synthetic) we use binary cross-entropy.

The learning rate and beta value for the Adam optimizer were experimentally tuned. I've found that a lower learning rate and beta value for the Adam optimizer improves training on the MNIST dataset. Applying decay helps stabilize training as well.

Given both the generator and discriminator we can build the GAN:

```
50 # build the adversarial model by first setting the discriminator to
51 # *not* be trainable, then combine the generator and discriminator
52 # together
53 print("[INFO] building GAN...")
54 disc.trainable = False
55 ganInput = Input(shape=(100,))
56 ganOutput = disc(gen(ganInput))
57 gan = Model(ganInput, ganOutput)
58
59 # compile the GAN
60 ganOpt = Adam(lr=0.0002, beta_1=0.5, decay=0.0002 / NUM_EPOCHS)
61 gan.compile(loss="binary_crossentropy", optimizer=discOpt)
```

The actual GAN consists of *both* the generator and the discriminator; however, we first need to freeze the discriminator weights (**Line 54**) before we combine the models (**Lines 55-57**).

Here we can see that the input to the `gan` will take a random vector that is 100-d. This value will be passed through the generator first and then the output will go to the discriminator. The discriminator weights are frozen at this point so the feedback from the discriminator will enable the generator to learn how to generate better synthetic images.

Lines 60-61 then compile the `gan`. I again used the Adam optimizer with the same hyperparameters as the optimizer for the discriminator — this process worked for our architecture + dataset, but you may need to tune these values yourself for other datasets and models. I've often found that setting the learning rate of the actual GAN to be *half* of the discriminator is a good starting point.

Throughout the training process we'll want to see how our GAN evolves to construct synthetic images from random noise. To accomplish this process, we'll need to generate some benchmark random noise used to visualize the training process:

```
63 # randomly generate some benchmark noise so we can consistently
64 # visualize how the generative modeling is learning
```

```
65 print("[INFO] starting training...")
66 benchmarkNoise = np.random.uniform(-1, 1, size=(256, 100))
```

Notice how our `benchmarkNoise` is generated from a uniform distribution in the range $[-1, 1]$ (the range of the *tanh* activation function). We'll be generating 256 images where each input vector is 100-d.

Let's see how we can train our GAN:

```
68 # loop over the epochs
69 for epoch in range(NUM_EPOCHS):
70     # show epoch information and compute the number of batches per
71     # epoch
72     print("[INFO] starting epoch {} of {}...".format(epoch + 1,
73         NUM_EPOCHS))
74     batchesPerEpoch = int(trainImages.shape[0] / BATCH_SIZE)
75
76     # loop over the batches
77     for i in range(0, batchesPerEpoch):
78         # initialize an (empty) output path
79         p = None
80
81         # select the next batch of images, then randomly generate
82         # noise for the generator to predict on
83         imageBatch = trainImages[i * BATCH_SIZE:(i + 1) * BATCH_SIZE]
84         noise = np.random.uniform(-1, 1, size=(BATCH_SIZE, 100))
85
86         # generate images using the noise + generator model
87         genImages = gen.predict(noise, verbose=0)
```

On **Line 69** we loop over our desired number of epochs. **Line 74** computes the number of batches per epoch by dividing the number of training images by the supplied batch size.

We then loop over each batch on **Line 77**. **Line 83** sequentially extracts the next `imageBatch` while **Line 84** generates random noise that we'll be passing through the generator. Given the `noise` we can use the generator to generate synthetic images (**Line 87**).

Given our generated images we need to train the discriminator to recognize the difference between real and synthetic images:

```
89         # concatenate the *actual* images and the *generated* images,
90         # construct class labels for the discriminator, and shuffle
91         # the data
92         X = np.concatenate((imageBatch, genImages))
93         y = ([1] * BATCH_SIZE) + ([0] * BATCH_SIZE)
94         (X, y) = shuffle(X, y)
95
96         # train the discriminator on the data
97         discLoss = disc.train_on_batch(X, y)
```

Line 92 concatenates the current `imageBatch` and the synthetic `genImages` together. We then need to build our class labels on **Line 93** — each *real* image will have a class label of `1` while every fake image will be labeled `0`.

The training data is then jointly shuffled on **Line 94** so our real images and fake images do not sequentially follow each other one-by-one. I have found this shuffling process improves the stability of discriminator training. **Line 97** trains the discriminator on the current (shuffled) batch.

The final step in our training process is to train the gan itself:

```
99          # let's now train our generator via the adversarial model by
100         # (1) generating random noise and (2) training the generator
101         # with the discriminator weights frozen
102         noise = np.random.uniform(-1, 1, (BATCH_SIZE, 100))
103         ganLoss = gan.train_on_batch(noise, [1] * BATCH_SIZE)
```

We first generate a total of BATCH_SIZE random vectors. However, unlike in our previous code block where we were nice enough to tell our discriminator what is real vs. fake, we're now going to try to trick the discriminator by labeling the random noise as a real image.

The feedback from the discriminator enables use to actually train the generator (keeping in mind that the discriminator weights are frozen for this operation). Not only is looking at the loss values important when training a GAN, but you *also* need to visually examine the output of the gan on your benchmarkNoise:

```
105         # check to see if this is the end of an epoch, and if so,
106         # initialize the output path
107         if i == batchesPerEpoch - 1:
108             p = [args["output"], "epoch_{}_output.png".format(
109                 str(epoch + 1).zfill(4))]
110
111         # otherwise, check to see if we should visualize the current
112         # batch for the epoch
113         else:
114             # create more visualizations early in the training
115             # process
116             if epoch < 10 and i % 25 == 0:
117                 p = [args["output"], "epoch_{}_step_{}.png".format(
118                     str(epoch + 1).zfill(4), str(i).zfill(5))]
119
120             # visualizations later in the training process are less
121             # interesting
122             elif epoch >= 10 and i % 100 == 0:
123                 p = [args["output"], "epoch_{}_step_{}.png".format(
124                     str(epoch + 1).zfill(4), str(i).zfill(5))]
```

If we have reached the end of the epoch we'll build the path, p, to our output visualization (**Lines 107-109**). Otherwise, I find it helpful to visually inspect the output of our GAN in *earlier* steps rather than *later* ones (**Lines 111-124**).

The output visualization should be totally random salt and pepper noise at the beginning but should quickly start to develop characteristics of the input data. These characteristics may not look real, but the evolving attributes will demonstrate to you that the network is actually learning. If your output visualizations are still salt and pepper noise after 5-10 epochs it's a sign that you need to tune your hyperparameters, potentially including model architecture as well.

Our final code block handles writing the synthetic image visualization to disk:

```
126         # check to see if we should visualize the output of the
127         # generator model on our benchmark data
```

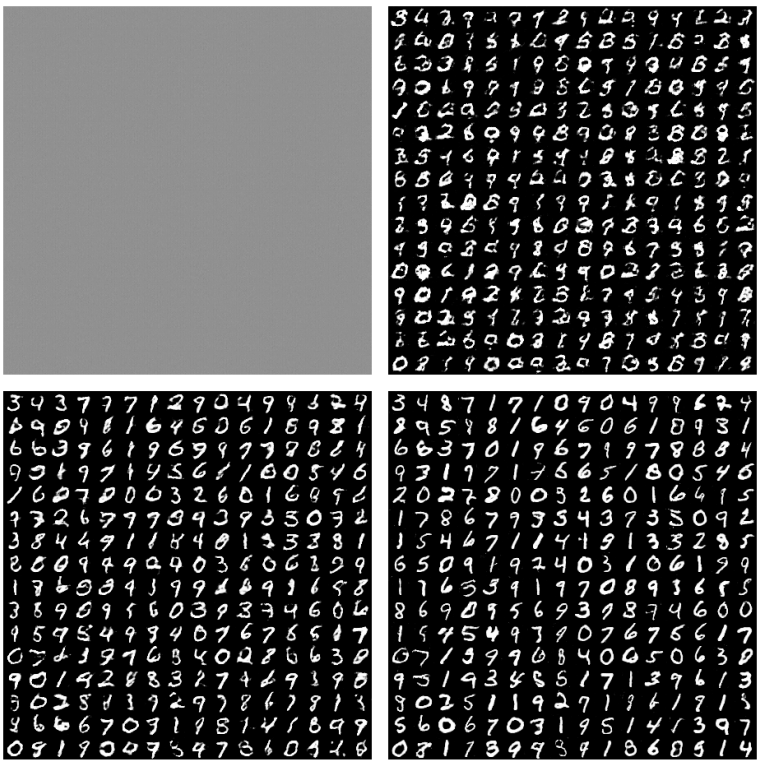


Figure 16.1: **Top-left:** The initial random noise of 256 input noise vectors. **Top-right:** The same random noise vectors after 2 epochs. We are starting to see the makings of digits. **Bottom-left:** The digits are now more clear after 5 epochs. **Bottom-right:** The final digits after 50 epochs look very authentic.

```
128             if p is not None:
129                 # show loss information
130                 print("[INFO] Step {}_{}: discriminator_loss={:.6f}, "
131                     "adversarial_loss={:.6f}".format(epoch + 1, i,
132                         discLoss, ganLoss))
133
134                 # make predictions on the benchmark noise, scale it back
135                 # to the range [0, 255], and generate the montage
136                 images = gen.predict(benchmarkNoise)
137                 images = ((images * 127.5) + 127.5).astype("uint8")
138                 images = np.repeat(images, 3, axis=-1)
139                 vis = build_montages(images, (28, 28), (16, 16))[0]
140
141                 # write the visualization to disk
142                 p = os.path.sep.join(p)
143                 cv2.imwrite(p, vis)
```

Since we are generating single channel images, we repeat the grayscale channel three times to construct a 3-channel RGB image (**Line 138**). The `build_montages` convenience function generates a 16×16 grid, each with a 28×28 image. This montage is written to disk on **Line 143**.

16.4 GAN Results

To train our GAN on the MNIST dataset, open up a terminal and execute the following command:

```
$ python dcgan_mnist.py --output output
[INFO] loading MNIST dataset...
[INFO] building generator...
[INFO] building discriminator...
[INFO] building GAN...
[INFO] starting training...
[INFO] starting epoch 1 of 50...
[INFO] Step 1_0: discriminator_loss=0.699669, adversarial_loss=0.645862
[INFO] Step 1_25: discriminator_loss=0.011363, adversarial_loss=3.094030
...
[INFO] Step 50_545: discriminator_loss=0.429998, adversarial_loss=1.854306
```

I used a single Titan X GPU to train our GAN on the MNIST time. From start to finish, the total training time was **17m15s**. Figure 16.1 shows our random noise vectors (i.e., `benchmarkNoise`) during different moments of training.

The *top-left* contains 256 (in an 8×8 grid) of our initial random noise vectors before even starting to train the GAN. We can clearly see there is no pattern in this noise. No digit-like structures have been learned by the GAN. However, by the end of the second epoch (*top-right*) digit-like structures are starting to appear.

By the end of the fifth epoch (*bottom-left*) the digits are *significantly* more clear. At the end of the fiftieth epoch (*bottom-right*) our digits look authentic.

Again, it's important to understand that these digits are generated from random noise input vectors — they are *totally synthetic* digits.

16.5 Summary

In this chapter we discussed Generative Adversarial Networks (GANs). We learned that GANs actually consist of *two* networks:

1. A *generator* which is responsible for generating fake images
2. A *discriminator* that tries to spot the synthetic images from the authentic ones

By training both of these networks at the same time we can learn to generate very realistic output images.

We then implemented Deep Convolutional Generative Adversarial Networks (DCGANs) [71], a variation of Goodfellow et al.'s original GAN implementation [70]. Using our DCGAN implementation, we trained both the generator and discriminator on the MNIST dataset, resulting in output images of digits that:

1. Are *not* part of the training set and are *completely* synthetic
2. Look nearly identical and indistinguishable to any hand drawn digit in the MNIST dataset

The problem is that training GANs can be *extremely* challenging, more so than *any* other architecture or method we have discussed in this book. The reason GANs are notoriously hard to train is due to the *evolving loss landscape* — with every step, our loss landscape changes slightly and is thus ever evolving. The evolving loss landscape is in stark contrast to other classification or regression tasks where the loss landscape is "fixed" and non-moving.

When training your own GANs you'll undoubtedly have to carefully tune both your model architecture and associated hyperparameters. To help you better train your own GANs, we discussed guidelines and best practices from Radford et al. [71] and Chollet [63].

17. Image Super Resolution

Just as deep learning and Convolutional Neural Networks have completely changed the landscape of art generated via deep learning methods, the same is true for super resolution algorithms; however, it's worth noting that the super resolution sub-field of computer vision has been studied with more rigor. Previous methods are primarily example-based and tend to either (1) use internal similarities of an input image to build the super resolution output [74, 75, 76], (2) learn low resolution to high resolution patches [77, 78, 79], or (3) use some variant of sparse-coding [80].

In this chapter we are going to implement the work of Dong et al. in their 2015 paper, *Image Super-Resolution Using Deep Learning Convolutional Neural Networks* [81]. This method demonstrates that previous sparse-coding methods are effectively equivalent to applying deep Convolutional Neural Networks — the primary difference being that the method we are implementing is faster, produces better results, and is entirely end-to-end.

While there have been many super resolution papers since the work of Dong et al. in 2015 (including a wonderful paper by Stanford's Justin Johnson on framing super resolution as style transfer [82]), the work of Dong et al. forms a foundation on which many others Super Resolution Convolutional Neural Networks (SRCNNs) are built.

17.1 Understanding SRCNNs

SRCNNs have a number of important characteristics. The most significant attributes are listed below:

1. **SRCNNs are fully convolutional** (which is not to be confused with *fully-connected*). We can input any image size (provided the width and height will tile) and run the SRCNN on it. This makes SRCNNs very fast.
2. **We train for filters, not for accuracy.** In previous chapters we have been primarily concerned with training our CNNs to achieve as high accuracy as possible on a given dataset. In this context we're concerned with the actual *filters* learned by the SRCNN which will enable us to upscale an image — the actual accuracy obtained on the training dataset to learn these filters is inconsequential.

3. **They do not require solving an optimization on usage.** After an SRCNN has learned a set of filters, it can apply a simple forward pass to obtain the output super resolution image. We do not have to optimize a loss function on a per-image basis to obtain the output.
4. **They are totally end-to-end.** Again, SRCNNs are totally end-to-end: input an image to the network and obtain the higher resolution output. There are no intermediary steps. Once training is complete we are ready to apply super resolution.

As mentioned above, the goal of our SRCNN is to learn a set of filters that allow us to **map low resolution inputs** to **a higher resolution output**. Therefore, instead of actual full-resolution images, we are going to construct two sets of image patches:

1. A low resolution patch that will be the input to the network
2. A high resolution patch that will be the *target* for the network to predict/reconstruct

In this way our SRCNN will learn how to reconstruct high resolution patches from low resolution input ones. In practice, this means that we:

1. First need to build a dataset of low and high resolution input patches
2. Train a network to learn to map the low resolution patches to their high resolution counterparts
3. Create a script that utilizes loops over the input patches of a low resolution images, passes them through the network, and then creates the output high resolution image from the predicted patches.

As we'll see later in this chapter, building an SRCNN is arguably easier than some of the other classification challenges we have tackled in this book.

17.2 Implementing SRCNNs

In the first part of this section we'll review the directory structure for this project, including any configuration files needed. From there we'll review a Python script used to build both our low resolution and high resolution patch datasets. We'll then implement our SRCNN architecture itself and train it. Finally, we'll utilize our trained model to apply SRCNNs to input images.

17.2.1 Starting the Project

Let's start by reviewing the directory structure for this project. The first step is to create a new file named `srcnn.py` inside the `conv` sub-module of `pyimagesearch` — this is where our Super Resolution Convolutional Neural Network will live:

```
--- pyimagesearch
|    |--- __init__.py
...
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- alexnet.py
|    |    |    |--- dcgan.py
...
|    |    |    |--- srcnn.py
...
```

From there we have the following directory structure for the project itself:

```
--- super_resolution
|    |--- build_dataset.py
```

```
|    |--- config
|    |    |--- sr_config.py
|    |--- output/
|    |--- resize.py
|    |--- train.py
|    |--- uk_bench/
```

The `sr_config.py` file stores any configurations we'll need. We'll then use `build_dataset.py` to create our low resolution and high resolution patches for training.

Dong et al. [81] refers to *patches* as *sub-images* instead. This attempt at clarification is meant to avoid any ambiguity regarding patches (which may implying overlapping ROIs in some computer vision literature). I will use both terms interchangeably as I believe the context of the work will define if a patch overlaps or not — both terms are totally valid.

From there we have the `train.py` script that will actually train our network. And finally, we'll implement `resize.py` to accept a low resolution input image and create the high resolution output.

The `output` directory will store (1) our HDF5 training set of images and (2) the output model itself. The `uk_bench` directory will contain the example images we are learning patterns from.

Let's go ahead and take a look at the `sr_config.py` file now:

```
1  # import the necessary packages
2  import os
3
4  # define the path to the input images we will be using to build the
5  # training crops
6  INPUT_IMAGES = "ukbench100"
7
8  # define the path to the temporary output directories
9  BASE_OUTPUT = "output"
10 IMAGES = os.path.sep.join([BASE_OUTPUT, "images"])
11 LABELS = os.path.sep.join([BASE_OUTPUT, "labels"])
12
13 # define the path to the HDF5 files
14 INPUTS_DB = os.path.sep.join([BASE_OUTPUT, "inputs.hdf5"])
15 OUTPUTS_DB = os.path.sep.join([BASE_OUTPUT, "outputs.hdf5"])
16
17 # define the path to the output model file and the plot file
18 MODEL_PATH = os.path.sep.join([BASE_OUTPUT, "srcnn.model"])
19 PLOT_PATH = os.path.sep.join([BASE_OUTPUT, "plot.png"])
```

Line 6 defines the path to the `ukbench100` dataset (included in the download of this book), a subset of the larger UKBench dataset [83]. Dong et al. experimented with both a 91-image dataset for training along with the full 1.2 million ImageNet dataset. Replicating the work of an SRCNN trained on ImageNet is outside the scope of this book so we'll stick with an image dataset closer to the size of Dong et al.'s first experiment.

Lines 9-11 build the path to a temporary output directory where we'll be storing our low resolution and high resolution sub-images. Given the low and high resolution sub-images, we'll be generating output HDF5 datasets from them (**Lines 14 and 15**). **Lines 18 and 19** then define the path to our output model file along with a training plot.

Let's continue defining our configurations:

```
# initialize the batch size and number of epochs for training
BATCH_SIZE = 128
NUM_EPOCHS = 10

# initialize the scale (the factor in which we want to learn how to
# enlarge images by) along with the input width and height dimensions
# to our SRCNN
SCALE = 2.0
INPUT_DIM = 33

# the label size should be the output spatial dimensions of the SRCNN
# while our padding ensures we properly crop the label ROI
LABEL_SIZE = 21
PAD = int((INPUT_DIM - LABEL_SIZE) / 2.0)

# the stride controls the step size of our sliding window
STRIDE = 14
```

We'll only be training for ten epochs as `NUM_EPOCHS` defines. Dong et al. found that training for longer can actually *hurt* performance (where "performance" here is defined as the quality of the output super resolution image). Ten epochs should be sufficient for our SRCNN to learn a set of filters to map our low resolution patches to their higher resolution counterparts.

The `SCALE` (**Line 28**) defines the factor by which we are upscaling our images — here we are upscaling by $2\times$ but you could upscale by $3\times$ or $4\times$ as well.

The `INPUT_DIM` is the spatial width and height of our sub-windows (33×33 pixels). Our `LABEL_SIZE` is the output spatial dimensions of the SRCNN while our `PAD` ensures we properly crop the label ROI when building our dataset and applying super resolution to input images.

Finally, the `STRIDE` controls the step size of our sliding window when creating sub-images. Dong et al. suggest a stride of 14 pixels for smaller image datasets and a stride of 33 pixels for larger datasets.

Before we get too far, you might be confused why the `INPUT_DIM` is larger than the `LABEL_SIZE` — isn't the entire idea here to build a higher resolution output image? How is that possible when the outputs of our neural network are *smaller* than the inputs? The answer is twofold.

First, as we'll see in Section 17.2.3, our SRCNN contains no zero-padding. Using zero-padding introduces border artifacts which would degrade the quality of our output image. Since we are not using zero-padding, our spatial dimensions will naturally reduce after each `CONV` layer.

Secondly, when applying super resolution to an input image (after training) we'll actually be *increasing* the input, low resolution image by a factor of `SCALE` — the network will then transform the low resolution image at the higher scale to a high resolution output image. If this processes seems confusing, don't worry, the remaining sections in this chapter will help make it clear.

17.2.2 Building the Dataset

Let's go ahead and build our training dataset for the SRCNN. Open up `build_dataset.py` and insert the following code:

```
# import the necessary packages
from pyimagesearch.io import HDF5DatasetWriter
from conf import sr_config as config
from imutils import paths
from scipy import misc
import shutil
```

```
7  import random
8  import cv2
9  import os
10
11 # if the output directories do not exist, create them
12 for p in [config.IMAGES, config.LABELS]:
13      if not os.path.exists(p):
14           os.makedirs(p)
15
16 # grab the image paths and initialize the total number of crops
17 # processed
18 print("[INFO] creating temporary images...")
19 imagePaths = list(paths.list_images(config.INPUT_IMAGES))
20 random.shuffle(imagePaths)
21 total = 0
```

Lines 2-9 handle our imports. Notice how we're once again using the `HDF5DatasetWriter` class to write our dataset to disk in HD5 format. Our `config` is imported on **Line 3** so we have access to our specified values.

Lines 12-14 create temporary output directories where we'll be storing our sub-windows. Once all sub-windows are generated we'll add them to the HDF5 dataset and then delete the temporary directories. **Lines 19-21** then grab the paths to our input images and initialize a counter to count the `total` number of sub-windows generated.

Let's loop over each of the image paths:

```
23 # loop over the image paths
24 for imagePath in imagePaths:
25      # load the input image
26      image = cv2.imread(imagePath)
27
28      # grab the dimensions of the input image and crop the image such
29      # that it tiles nicely when we generate the training data +
30      # labels
31      (h, w) = image.shape[:2]
32      w -= int(w % config.SCALE)
33      h -= int(h % config.SCALE)
34      image = image[0:h, 0:w]
35
36      # to generate our training images we first need to downscale the
37      # image by the scale factor...and then upscale it back to the
38      # original size -- this process allows us to generate low
39      # resolution inputs that we'll then learn to reconstruct the high
40      # resolution versions from
41      scaled = misc.imresize(image, 1.0 / config.SCALE,
42           interp="bicubic")
43      scaled = misc.imresize(scaled, config.SCALE / 1.0,
44           interp="bicubic")
```

For each input image we first load it from disk (**Line 26**) and then crop the image such that it tiles nicely when generating our sub-windows (**Lines 31-34**). If we did not take this step, our stride size would not fit and we would crop patches *outside* of the image's spatial dimensions.

In order to generate training data for our SRCNN we need to:

1. Downscale the original input image by a factor of `SCALE` (for `SCALE = 2.0` we are halving the size of the input image)
2. And then rescale it back to the original size

This process generates a low resolution image with the same original spatial dimensions. We will learn how to *reconstruct* a high resolution image from this low resolution input.

I found that trying to use OpenCV's bilinear interpolation (recommended by Dong et al.) produced inferior results (it also introduced more code to perform the scaling). I opted for SciPy's `.imresize` function as it was easier to use and generated better results.

We can now generate our sub-windows for both the inputs and targets:

```
46      # slide a window from left-to-right and top-to-bottom
47      for y in range(0, h - config.INPUT_DIM + 1, config.STRIDE):
48          for x in range(0, w - config.INPUT_DIM + 1, config.STRIDE):
49              # crop output the `INPUT_DIM x INPUT_DIM` ROI from our
50              # scaled image -- this ROI will serve as the input to our
51              # network
52              crop = scaled[y:y + config.INPUT_DIM,
53                  x:x + config.INPUT_DIM]
54
55              # crop out the `LABEL_SIZE x LABEL_SIZE` ROI from our
56              # original image -- this ROI will be the target output
57              # from our network
58              target = image[
59                  y + config.PAD:y + config.PAD + config.LABEL_SIZE,
60                  x + config.PAD:x + config.PAD + config.LABEL_SIZE]
61
62              # construct the crop and target output image paths
63              cropPath = os.path.sep.join([config.IMAGES,
64                  "{}.png".format(total)])
65              targetPath = os.path.sep.join([config.LABELS,
66                  "{}.png".format(total)])
67
68              # write the images to disk
69              cv2.imwrite(cropPath, crop)
70              cv2.imwrite(targetPath, target)
71
72              # increment the crop total
73              total += 1
```

Lines 47-48 slide a window from left-to-right and top-to-bottom across our images. We crop the `INPUT_DIM x INPUT_DIM` sub-window on **Lines 52 and 53** — this `crop` is the 33×33 from our `scaled` (i.e., low resolution image) input to our neural network.

We also need a `target` for the SRCNN to predict (**Lines 58 and 60**) — the `target` is the `LABEL_SIZE x LABEL_SIZE` (21×21) output that the SRCNN will be trying to reconstruct. We write both the `target` and `crop` to disk on **Lines 62-70**.

The last step is to build two HDF5 datasets, one for the inputs and another for the outputs (i.e., the targets):

```
75  # grab the paths to the images
76  print("[INFO] building HDF5 datasets...")
```

```
77  inputPaths = sorted(list(paths.list_images(config.IMAGES)))
78  outputPaths = sorted(list(paths.list_images(config.LABELS)))
79
80  # initialize the HDF5 datasets
81  inputWriter = HDF5DatasetWriter((len(inputPaths), config.INPUT_DIM,
82       config.INPUT_DIM, 3), config.INPUTS_DB)
83  outputWriter = HDF5DatasetWriter((len(outputPaths),
84       config.LABEL_SIZE, config.LABEL_SIZE, 3), config.OUTPUTS_DB)
85
86  # loop over the images
87  for (inputPath, outputPath) in zip(inputPaths, outputPaths):
88       # load the two images and add them to their respective datasets
89       inputImage = cv2.imread(inputPath)
90       outputImage = cv2.imread(outputPath)
91       inputWriter.add([inputImage], [-1])
92       outputWriter.add([outputImage], [-1])
93
94  # close the HDF5 datasets
95  inputWriter.close()
96  outputWriter.close()
97
98  # delete the temporary output directories
99  print("[INFO] cleaning up...")
100 shutil.rmtree(config.IMAGES)
101 shutil.rmtree(config.LABELS)
```

It's important to note that the class label is irrelevant (hence why I specify a value of `-1`). The "class label" is technically the output sub-window that we would try to train our SRCNN to reconstruct. We'll be writing a custom generator in `train.py` to yield a tuple of both the input sub-windows and target output sub-windows.

After our HDF5 datasets are generated, **Lines 100 and 101** delete the temporary output directories.

17.2.3 The SRCNN Architecture

The SRCNN architecture we are implementing follows Dong et al.'s exactly — which has the added benefit of making it easy to implement. Open up `srcnn.py` 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 import backend as K
6
7  class SRCNN:
8       @staticmethod
9       def build(width, height, depth):
10           # initialize the model
11           model = Sequential()
12           inputShape = (height, width, depth)
13
14           # if we are using "channels first", update the input shape
15           if K.image_data_format() == "channels_first":
16                inputShape = (depth, height, width)
```

```
17
18          # the entire SRCNN architecture consists of three CONV =>
19          # RELU layers with *no* zero-padding
20          model.add(Conv2D(64, (9, 9), kernel_initializer="he_normal",
21              input_shape=inputShape))
22          model.add(Activation("relu"))
23          model.add(Conv2D(32, (1, 1), kernel_initializer="he_normal"))
24          model.add(Activation("relu"))
25          model.add(Conv2D(depth, (5, 5),
26              kernel_initializer="he_normal"))
27          model.add(Activation("relu"))
28
29          # return the constructed network architecture
30          return model
```

Compared to other architectures we have reviewed in the *Practitioner Bundle*, such as GoogLeNet in Chapter 11 and ResNet in Chapter 12, our SRCNN could not be more straightforward — in fact, the entire architecture consists of only three `CONV => RELU` layers with *no* zero-padding (we avoid using zero-padding to ensure we don't introduce any border artifacts in the output image).

Our first `CONV` layer learns 64 filters, each of which are 9×9. This volume is fed into a second `CONV` layer where we learn 32 1×1 filters which are used to reduce dimensionality and learn local features. The final `CONV` layer learns a total of `depth` channels (which will be `3` for RGB images and `1` for grayscale), each of which are 5×5.

There are two important components of this network architecture:

1. It's small and compact, meaning that it will be fast to train (remember, our goal isn't to obtain higher accuracy in a classification sense — we're more interested in the *filters* learned from the network which will enable us to perform super resolution).
2. It's fully-convolutional, making it, (1) again, faster, and (2) possible for us to accept any input image size provided it tiles nicely.

17.2.4 Training the SRCNN

Training our SRCNN is a fairly straightforward process. Open up `train.py` and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from conf import sr_config as config
7  from pyimagesearch.io import HDF5DatasetGenerator
8  from pyimagesearch.nn.conv import SRCNN
9  from keras.optimizers import Adam
10 import matplotlib.pyplot as plt
11 import numpy as np
```

Lines 2-11 handle our imports. We'll need the `HDF5DatasetGenerator` to access our serialized HDF5 datasets along with our `SRCNN` implementation. However, a slight modification is required to work with our HDF5 dataset.

Keep in mind that we have two HDF5 datasets: the input sub-windows and the target output sub-windows. Our `HDF5DatasetGenerator` class is meant to work with only one HDF5 file, not two. Luckily, this is an easy fix with our `super_res_generator` function:

```
13 def super_res_generator(inputDataGen, targetDataGen):
14     # start an infinite loop for the training data
15     while True:
16         # grab the next input images and target outputs, discarding
17         # the class labels (which are irrelevant)
18         inputData = next(inputDataGen)[0]
19         targetData = next(targetDataGen)[0]
20
21         # yield a tuple of the input data and target data
22         yield (inputData, targetData)
```

This function requires two arguments, `inputDataGen` and `targetDataGen` which are both assumed to be `HDF5DatasetGenerator` objects.

We start an infinite loop that will continue to loop over our training data on **Line 15**. Calling `next` (a built-in Python function to return the next item in a generator) on each of these objects yields us the next batch set. We discard the class labels (since we do not need them) and return a tuple of the `inputData` and `targetData`.

We can now initialize our `HDF5DatasetGenerator` objects along with our model and optimizer:

```
24 # initialize the input images and target output images generators
25 inputs = HDF5DatasetGenerator(config.INPUTS_DB, config.BATCH_SIZE)
26 targets = HDF5DatasetGenerator(config.OUTPUTS_DB, config.BATCH_SIZE)
27
28 # initialize the model and optimizer
29 print("[INFO] compiling model...")
30 opt = Adam(lr=0.001, decay=0.001 / config.NUM_EPOCHS)
31 model = SRCNN.build(width=config.INPUT_DIM, height=config.INPUT_DIM,
32     depth=3)
33 model.compile(loss="mse", optimizer=opt)
```

While Dong et al. used RMSprop, I found that:

1. Using Adam obtained better results with less hyperparameter tuning
2. A little bit of learning rate decay yields better, more stable training

Finally, note that we'll be using mean-squared loss (MSE) rather than binary/categorical cross-entropy.

We are now ready to train our model:

```
35 # train the model using our generators
36 H = model.fit_generator(
37     super_res_generator(inputs.generator(), targets.generator()),
38     steps_per_epoch=inputs.numImages // config.BATCH_SIZE,
39     epochs=config.NUM_EPOCHS, verbose=1)
```

Our model will be trained for a total of `NUM_EPOCHS` (10 epochs, according to our configuration file). Notice how we use our `super_res_generator` to jointly yield training batches from both the `inputs` and `targets` generators, respectively.

Our final code block handles saving our trained model to disk, plotting the loss, and closing our HDF5 datasets:

```
# save the model to file
print("[INFO] serializing model...")
model.save(config.MODEL_PATH, overwrite=True)

# plot the training loss
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, config.NUM_EPOCHS), H.history["loss"],
     label="loss")
plt.title("Loss on super resolution training")
plt.xlabel("Epoch #")
plt.ylabel("Loss")
plt.legend()
plt.savefig(config.PLOT_PATH)

# close the HDF5 datasets
inputs.close()
targets.close()
```

17.2.5 Increasing Image Resolution With SRCNNs

We are now ready to implement `resize.py`, the script responsible for constructing a high resolution output image from a low resolution input image. Open up `resize.py` and insert the following code:

```
# import the necessary packages
from conf import sr_config as config
from keras.models import load_model
from scipy import misc
import numpy as np
import argparse
import cv2

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
     help="path to input image")
ap.add_argument("-b", "--baseline", required=True,
     help="path to baseline image")
ap.add_argument("-o", "--output", required=True,
     help="path to output image")
args = vars(ap.parse_args())

# load the pre-trained model
print("[INFO] loading model...")
model = load_model(config.MODEL_PATH)
```

Lines 10-17 parse our command line arguments — we'll need three switches for this script:

- `--image`: The path to our input, low resolution image that we wish to upscale.
- `--baseline`: The output baseline image after standard bilinear interpolation — this image will give us a baseline to which we can compare our SRCNN results.
- `--output`: The path to the output image after applying super resolution.

Line 21 then loads our serialized SRCNN after disk.

Next, let's prepare our image for upscaling:

```
23 # load the input image, then grab the dimensions of the input image
24 # and crop the image such that it tiles nicely
25 print("[INFO] generating image...")
26 image = cv2.imread(args["image"])
27 (h, w) = image.shape[:2]
28 w -= int(w % config.SCALE)
29 h -= int(h % config.SCALE)
30 image = image[0:h, 0:w]
31
32 # resize the input image using bicubic interpolation then write the
33 # baseline image to disk
34 scaled = misc.imresize(image, config.SCALE / 1.0,
35      interp="bicubic")
36 cv2.imwrite(args["baseline"], scaled)
37
38 # allocate memory for the output image
39 output = np.zeros(scaled.shape)
40 (h, w) = output.shape[:2]
```

We first load the image from disk on **Line 27**. **Lines 28-30** crop our image such that it tiles nicely when applying our sliding window and passing the sub-images through our SRCNN. **Lines 34 and 35** apply standard bilinear interpolation to our input `--image` using SciPy's `imresize` function.

Upscaling our image by a factor of `SCALE` serves two purposes:

1. It gives us a baseline of what standard upsizing will look like using traditional image processing.
2. Our SRCNN requires a high resolution input of the original low resolution image — this `scaled` image serves that purpose.

Finally, **Line 38** allocates memory for our `output` image.

We can now apply our sliding window:

```
42 # slide a window from left-to-right and top-to-bottom
43 for y in range(0, h - config.INPUT_DIM + 1, config.LABEL_SIZE):
44      for x in range(0, w - config.INPUT_DIM + 1, config.LABEL_SIZE):
45           # crop ROI from our scaled image
46           crop = scaled[y:y + config.INPUT_DIM,
47                x:x + config.INPUT_DIM]
48
49           # make a prediction on the crop and store it in our output
50           # image
51           P = model.predict(np.expand_dims(crop, axis=0))
52           P = P.reshape((config.LABEL_SIZE, config.LABEL_SIZE, 3))
53           output[y + config.PAD:y + config.PAD + config.LABEL_SIZE,
54                x + config.PAD:x + config.PAD + config.LABEL_SIZE] = P
```

For each stop along the way, in `LABEL_SIZE` steps, we `crop` out the sub-image from `scaled` (**Lines 46 and 47**). The spatial dimensions of `crop` match the input dimensions required by our SRCNN.

We then take the `crop` sub-image and pass it through our SRCNN for inference. The output of the `SRCNN`, `P` has spatial dimensions `LABEL_SIZE x LABEL_SIZE x CHANNELS`, which in this

Figure 17.1: **Left:** The original input image. **Middle:** Applying standard bilinear interpolation to the input image. Notice how the image is blurry and not crisp. **Right:** Increasing the input image resolution by $2\times$ using our SRCNN. The image is significantly more visually appealing.

case is $21 \times 21 \times 3$ — we then store the high resolution prediction from the network in the `output` image.

> **R** For the sake of simplicity, I am processing one sub-image at a time via the `.predict` method. To achieve a faster throughput rate, especially on a GPU, you'll want to batch process the `crop` sub-images. This action can be accomplished by maintaining a list of (x, y)-coordinates for the batch that map each sample in the batch to their corresponding `output` location.

Our last step is to remove any black borders on the output image caused by padding and then save the image to disk:

```
56 # remove any of the black borders in the output image caused by the
57 # padding, then clip any values that fall outside the range [0, 255]
58 output = output[config.PAD:h - ((h % config.INPUT_DIM) + config.PAD),
59      config.PAD:w - ((w % config.INPUT_DIM) + config.PAD)]
60 output = np.clip(output, 0, 255).astype("uint8")
61
62 # write the output image to disk
63 cv2.imwrite(args["output"], output)
```

At this point we are totally done implementing the SRCNN pipeline! In our next section we'll apply `resize.py` to a few example input, low resolution images and compare our SRCNN results to traditional image processing.

17.3 Super Resolution Results

Now that we have (1) trained our SRCNN and (2) implemented `resize.py` we are ready to apply super resolution to an input image. Open up a shell and execute the following command:

```
$ python resize.py --image jemma.png --baseline baseline.png \
     --output output.png
[INFO] loading model...
[INFO] generating image...
```

Figure 17.1 contains our output image. On the *left* is our input image we wish to increase the resolution of (250×333).

Then, in the *middle*, we have the input image resolution increased by $2\times$ to 483×648 via standard bilinear interpolation. This image serves as our baseline. Notice how the image is low resolution, blurry, and in general, visually unappealing.

Finally, on the *right*, we have the output image from the SRCNN. Here we can see that we have again increased the resolution by $2\times$, but this time the image is significantly less blurry and more aesthetically pleasing.

We can also increase our image resolution by higher multiples provided we have trained our SRCNN to do so.

I would either (1) use a PDF viewer to examine the input, baseline, and output images or (2) examine the output files included in the downloads associated with this book. The print and eBook editions may not contain enough detail for you to see the improvements made by the SRCNN over the baseline.

17.4 Summary

In this chapter we reviewed the concept of "Super Resolution" and then implemented Super Resolution Convolutional Neural Networks (SRCNN). After training our SRCNN we were then able to apply super resolution to our input images.

Our implementation followed the work of Dong et al. [81] — while there have been many super resolution papers since then (and there will continue to be), Dong et al.'s 2015 paper is still one of the easiest ones to understand and implement, making it an excellent starting point for anyone interested in studying super resolution.

18. Where to Now?

Congratulations! You have just finished the *Practitioner Bundle* of *Deep Learning for Computer Vision with Python*. Let's take a second and review the skills you've acquired. Inside this book you have learned how to:

- Apply *data augmentation* and use it as a form of regularization to increase the accuracy of networks by *generating* new training examples from existing data points.
- Grasp the concept of rank-1 and rank-5 accuracy (and why we report both, especially on large, challenging image datasets).
- Utilize *feature extraction* as a form of transfer learning to quickly develop highly accurate image classifiers by treating pre-trained CNNs as black box feature extractors.
- Apply *fine-tuning* (a second form of transfer learning) where the weights of an *existing* network architecture are tuned to predict classes that the original network was not trained on.
- Utilize advanced optimized algorithms such as RMPprop and Adam, and learned how to "drive" these optimization algorithms (along with SGD) to quickly train networks on unfamiliar datasets.
- Apply my optimal pathway to choosing *which* deep learning technique you should use when confronted with a new problem (i.e., train from scratch, extract features, fine-tune, etc.).
- Construct HDF5 datasets from raw images residing on disk, allowing you to efficiently train neural networks too large to fit into memory.
- Compete in the Kaggle Dogs vs. Cats competition (and take home the #1 position).
- Apply "exotic" architectures such as GoogLeNet and ResNet, and how to use them in Stanford's cs231n Tiny ImageNet challenge.
- Perform basic object detection with CNNs using image pyramids, sliding windows, and non-maxima suppression.
- Leverage deep learning to generate works of for art using deep learning and neural style transfer.
- Generate realistic looking images using Generative Adversarial Networks.
- Construct higher resolution output images for low-resolution inputs using super resolution

CNN algorithms.

Each of these techniques are important tools for any deep learning practitioner to have in their tool belt. You'll find transfer learning via feature extraction and fine-tuning *especially* useful when you need to quickly obtain high accuracy deep learning classifiers with minimal effort.

Furthermore, the ability to rapidly construct HDF5 datasets from image datasets too large to fit into memory will allow you to run more experiments faster. Finally, our overview of GANs and super resolution algorithms will help prepare you for upcoming trends in deep learning literature.

After going through this book, you are *undoubtedly* a deep learning practitioner. But if you're ready to take your knowledge to the next level and start working with *state-of-the-art* network architectures on challenging real-world datasets, you should keep reading...

18.1 What's Next?

The *Practitioner Bundle* has given you the necessary skills you need to be successful when applying deep learning and Convolutional Neural Networks to a number of real-world projects. Using this skill set, you'll be able to solve many of the projects and problems you encounter.

However, if you want to *reproduce the results* of state-of-the-art deep learning publications and learn how to train large scale neural networks on the massive ImageNet dataset, then you should consider moving on to the *ImageNet Bundle* of *Deep Learning for Computer Vision with Python*.

The ImageNet dataset is the *de facto* standard for benchmarking image classification algorithms, consisting of over 1.2 million images and 1,000 image categories. Learning how to successfully train a Convolutional Neural Network on ImageNet is a *rite of passage* – one that separates the true deep learning *practitioners* from the deep learning *experts*.

If your goal is to reach true deep learning mastery and learn how to:

- Train state-of-the-art CNNs such as ResNet, SqueezeNet, GoogLeNet, VGG, and AlexNet *from scratch* on the ImageNet dataset.
- Recognize emotion and facial expressions in real-time video streams.
- Fine-tune a CNN to recognize over 160 vehicle make and models.
- Correctly predict the age and gender of a person in an image.

... then I would encourage you to keep going on your deep learning journey.

If you've enjoyed the *Starter Bundle* and *Practitioner Bundle*, then I have absolutely no doubt that you'll get tremendous value from the *ImageNet Bundle* as well. However, I will say that the techniques you will learn inside the *ImageNet Bundle* will require time, effort, and a drive to master deep learning – and not to mention, *at least* one dedicated GPU. These techniques are not learned easily, but they *will* set you apart from other deep learning practitioners in the computer vision space.

I hope you'll allow me to help you continue your journey to becoming a deep learning expert and join me on your next steps inside the *ImageNet Bundle*.

I'll see you there.

–Adrian Rosebrock

Bibliography

[1] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: http://arxiv.org/abs/1412.6980 (cited on pages 13, 87, 88).

[2] Geoffrey Hinton. *Neural Networks for Machine Learning*. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (cited on pages 13, 87).

[3] Kaggle Team. *Kaggle: Dogs vs. Cats*. https://www.kaggle.com/c/dogs-vs-cats (cited on pages 14, 97).

[4] Andrej Karpathy. *Tiny ImageNet Challenge*. http://cs231n.stanford.edu/project.html (cited on pages 14, 133, 170).

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cited on pages 15, 87, 88).

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Edited by F. Pereira et al. Curran Associates, Inc., 2012, pages 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (cited on pages 16, 88, 97, 105, 131).

[7] Yann Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. 1998, pages 2278–2324 (cited on page 16).

[8] Adrian Rosebrock. *PyImageSearch Gurus*. https://www.pyimagesearch.com/pyimagesearch-gurus/. 2016 (cited on page 18).

[9] Richard Szeliski. *Computer Vision: Algorithms and Applications*. 1st. New York, NY, USA: Springer-Verlag New York, Inc., 2010. ISBN: 1848829345, 9781848829343 (cited on page 18).

[10] Maria-Elena Nilsback and Andrew Zisserman. “A Visual Vocabulary for Flower Classification.” In: *CVPR (2)*. IEEE Computer Society, 2006, pages 1447–1454. URL: http://dblp.uni-trier.de/db/conf/cvpr/cvpr2006-2.html#NilsbackZ06 (cited on page 19).

[11] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556 (cited on pages 34, 88, 173).

[12] The HDF Group. *Hierarchical data format version 5*. http://www.hdfgroup.org/HDF5 (cited on page 35).

[13] Andrew Ng. *Machine Learning*. https://www.coursera.org/learn/machine-learning (cited on page 46).

[14] Navneet Dalal and Bill Triggs. “Histograms of Oriented Gradients for Human Detection”. In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*. CVPR '05. Washington, DC, USA: IEEE Computer Society, 2005, pages 886–893. ISBN: 0-7695-2372-2. DOI: 10.1109/CVPR.2005.177. URL: http://dx.doi.org/10.1109/CVPR.2005.177 (cited on pages 49, 59, 208).

[15] David G. Lowe. “Object Recognition from Local Scale-Invariant Features”. In: *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*. ICCV '99. Washington, DC, USA: IEEE Computer Society, 1999, pages 1150–. ISBN: 0-7695-0164-8. URL: http://dl.acm.org/citation.cfm?id=850924.851523 (cited on pages 49, 59).

[16] T. Ojala, M. Pietikainen, and T. Maenpaa. “Multiresolution gray-scale and rotation invariant texture classification with local binary patterns”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 24.7 (2002), pages 971–987 (cited on pages 49, 59).

[17] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *Computer Vision and Pattern Recognition (CVPR)*. 2015. URL: http://arxiv.org/abs/1409.4842 (cited on pages 53, 83, 88, 133, 135, 170).

[18] Yoav Freund and Robert E Schapire. “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”. In: *J. Comput. Syst. Sci.* 55.1 (Aug. 1997), pages 119–139. ISSN: 0022-0000. DOI: 10.1006/jcss.1997.1504. URL: http://dx.doi.org/10.1006/jcss.1997.1504 (cited on page 73).

[19] Leo Breiman. “Random Forests”. In: *Mach. Learn.* 45.1 (Oct. 2001), pages 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: http://dx.doi.org/10.1023/A:1010933404324 (cited on page 73).

[20] L. Breiman et al. *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984 (cited on page 73).

[21] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001 (cited on page 73).

[22] Cuong Nguyen, Yong Wang, and Ha Nam Nguyen. “Random forest classifier combined with feature selection for breast cancer diagnosis and prognostic”. In: *Journal of Biomedical Science and Engineering* (2013). URL: http://file.scirp.org/Html/6-9101686_31887.htm (cited on page 74).

[23] Thomas G. Dietterich. “Ensemble Methods in Machine Learning”. In: *Proceedings of the First International Workshop on Multiple Classifier Systems*. MCS ’00. London, UK, UK: Springer-Verlag, 2000, pages 1–15. ISBN: 3-540-67704-6. URL: http://dl.acm.org/citation.cfm?id=648054.743935 (cited on page 74).

[24] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). URL: http://arxiv.org/abs/1512.03385 (cited on pages 83, 133, 173, 194, 198, 204).

[25] Gao Huang et al. “Snapshot Ensembles: Train 1, get M for free”. In: *CoRR* abs/1704.00109 (2017). URL: http://arxiv.org/abs/1704.00109 (cited on page 83).

[26] Andrej Karpathy. *Neural Networks (Part III)*. http://cs231n.github.io/neural-networks-3/ (cited on pages 85, 87).

[27] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *CoRR* abs/1609.04747 (2016). URL: http://arxiv.org/abs/1609.04747 (cited on page 85).

[28] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *J. Mach. Learn. Res.* 12 (July 2011), pages 2121–2159. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1953048.2021068 (cited on page 86).

[29] Matthew D. Zeiler. “ADADELTA: An Adaptive Learning Rate Method”. In: *CoRR* abs/1212.5701 (2012). URL: http://arxiv.org/abs/1212.5701 (cited on page 86).

[30] Timothy Dozat. *Incorporating Nesterov Momentum into Adam*. http://cs229.stanford.edu/proj2015/054_report.pdf (cited on page 88).

[31] Tom Schaul, Ioannis Antonoglou, and David Silver. “Unit Tests for Stochastic Optimization”. In: *CoRR* abs/1312.6055 (2013). URL: http://arxiv.org/abs/1312.6055 (cited on page 88).

[32] Forrest N. Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size”. In: *CoRR* abs/1602.07360 (2016). URL: http://arxiv.org/abs/1602.07360 (cited on pages 88, 133).

[33] Kaiming He et al. “Identity Mappings in Deep Residual Networks”. In: *CoRR* abs/1603.05027 (2016). URL: http://arxiv.org/abs/1603.05027 (cited on pages 88, 174, 176, 204).

[34] Andrew Ng. *Nuts and Bolts of Building Applications using Deep Learning*. https://nips.cc/Conferences/2016/Schedule?showEvent=6203. 2016 (cited on pages 91, 93).

[35] Tomasz Malisiewicz. *Nuts and Bolts of Building Deep Learning Applications: Ng at NIPS2016*. http://www.computervisionblog.com/2016/12/nuts-and-bolts-of-building-deep.html (cited on page 91).

[36] Andrej Karpathy. *Transfer Learning*. http://cs231n.github.io/transfer-learning/ (cited on page 95).

[37] Greg Chu. *How to use transfer learning and fine-tuning in Keras and Tensorflow to build an image recognition system and classify (almost) any object*. https://deeplearningsandbox.com/how-to-use-transfer-learning-and-fine-tuning-in-keras-and-tensorflow-to-build-an-image-recognition-94b0b02444f2 (cited on page 95).

[38] Adrian Rosebrock. *Practical Python and OpenCV + Case Studies*. PyImageSearch.com, 2016. URL: https://www.pyimagesearch.com/practical-python-opencv/ (cited on page 107).

[39] Andrej Karpathy. *CS231n: Convolutional Neural Networks for Visual Recognition*. http://cs231n.stanford.edu/. 2016 (cited on pages 133, 148).

[40] Min Lin, Qiang Chen, and Shuicheng Yan. "Network In Network". In: *CoRR* abs/1312.4400 (2013). URL: http://arxiv.org/abs/1312.4400 (cited on page 134).

[41] Jost Tobias Springenberg et al. "Striving for Simplicity: The All Convolutional Net". In: *CoRR* abs/1412.6806 (2014). URL: http://arxiv.org/abs/1412.6806 (cited on pages 135, 173).

[42] Chiyuan Zhang et al. "Understanding deep learning requires rethinking generalization". In: *CoRR* abs/1611.03530 (2016). URL: http://arxiv.org/abs/1611.03530 (cited on page 135).

[43] WordNet. *About WordNet*. http://wordnet.princeton.edu. 2010 (cited on page 149).

[44] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*. 2010 (cited on page 173).

[45] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *CoRR* abs/1502.01852 (2015). URL: http://arxiv.org/abs/1502.01852 (cited on page 173).

[46] Kaiming He. *Deep Residual Networks*. https://github.com/KaimingHe/deep-residual-networks (cited on page 178).

[47] Wei Wu. *ResNet*. https://github.com/tornadomeet/ResNet (cited on page 178).

[48] Kaiming He. *ResNet: Should the convolution layers have biases?* https://github.com/KaimingHe/deep-residual-networks/issues/10#issuecomment-194037195 (cited on page 179).

[49] Rodrigo Benenson. *CIFAR-10: Who is the best in CIFAR-10?* http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#43494641522d3130 (cited on pages 182, 194).

[50] Theano Community. *Theano: Max Recursion Limit*. https://github.com/Theano/Theano/issues/689 (cited on page 183).

[51] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497 (2015). URL: http://arxiv.org/abs/1506.01497 (cited on pages 207, 219).

[52] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *CoRR* abs/1512.02325 (2015). URL: http://arxiv.org/abs/1512.02325 (cited on pages 207, 219).

[53] Paul Viola and Michael Jones. "Rapid object detection using a boosted cascade of simple features". In: 2001, pages 511–518 (cited on page 208).

[54] IIPImage Open Source Community. *IIPImage: Images and Formats*. http://iipimage.sourceforge.net/documentation/images/. 2016 (cited on page 209).

[55] C. Olah A. Mordvintsev and M. Tyka. *DeepDream - a code example for visualizing Neural Networks*. https://research.googleblog.com/2015/07/deepdream-code-example-for-visualizing.html. 2015 (cited on pages 221, 230).

[56] Adrian Rosebrock. *bat-country: A lightweight, extendible, easy to use Python package for deep dreaming and image generation with Caffe and CNNs*. https://github.com/jrosebr1/bat-country. 2015 (cited on page 221).

[57] Adrian Rosebrock. *bat-country: an extendible, lightweight Python package for deep dreaming with Caffe and Convolutional Neural Networks*. https://www.pyimagesearch.com/2015/07/06/bat-country-an-extendible-lightweight-python-package-for-deep-dreaming-with-caffe-and-convolutional-neural-networks/. 2015 (cited on page 221).

[58] Adrian Rosebrock. *Deep dream: Visualizing every layer of GoogLeNet*. https://www.pyimagesearch.com/2015/08/03/deep-dream-visualizing-every-layer-of-googlenet/. 2015 (cited on page 221).

[59] Adrian Rosebrock. *Generating art with guided deep dreaming*. https://www.pyimagesearch.com/2015/07/13/generating-art-with-guided-deep-dreaming/. 2015 (cited on pages 221, 231).

[60] /r/deepdream Reddit Community. *Generating art with guided deep dreaming*. https://www.reddit.com/r/deepdream/. 2015 (cited on page 221).

[61] Sarah Cascone. *Google's 'Inceptionism' Art Sells Big at San Francisco Auction*. https://news.artnet.com/market/google-inceptionism-art-sells-big-439352. 2016 (cited on page 222).

[62] F. Chollet. *Jupyter notebooks for the code samples of the book "Deep Learning with Python"*. https://github.com/fchollet/deep-learning-with-python-notebooks. 2018 (cited on page 222).

[63] F. Chollet. *Deep Learning with Python*. Manning Publications Company, 2017. ISBN: 9781617294433. URL: https://books.google.com/books?id=Yo3CAQAACAAJ (cited on pages 222, 228, 233, 245–247, 257).

[64] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. "A Neural Algorithm of Artistic Style". In: *CoRR* abs/1508.06576 (2015). URL: http://arxiv.org/abs/1508.06576 (cited on pages 231, 240, 243).

[65] Larry Hunter. *Fallingwater Blueprint Poster*. https://fineartamerica.com/products/fallingwater-blueprint-larry-hunter-poster.html. 2018 (cited on page 232).

[66] François Chollet and Keras Team. *Keras - Neural Style Transfer Example*. https://github.com/keras-team/keras/blob/master/examples/neural_style_transfer.py. 2017 (cited on page 235).

[67] Kevin Zakka. *style-transfer: A Keras Implementation of "A Neural Algorithm of Artistic Style"*. https://github.com/kevinzakka/style-transfer. 2017 (cited on page 235).

[68] D. C. Liu and J. Nocedal. "On the Limited Memory BFGS Method for Large Scale Optimization". In: *Math. Program.* 45.3 (Dec. 1989), pages 503–528. ISSN: 0025-5610 (cited on page 236).

[69] Aria Haghighi. *Numerical Optimization: Understanding L-BFGS*. http://aria42.com/blog/2014/12/understanding-lbfgs. 2014 (cited on page 236).

[70] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2014, pages 2672–2680. URL: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf (cited on pages 245, 257).

[71] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *CoRR* abs/1511.06434 (2015). URL: http://arxiv.org/abs/1511.06434 (cited on pages 245, 247, 257).

[72] Theano Development Team. *Convolution arithmetic tutorial*. http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html. 2017 (cited on page 248).

[73] Paul-Louis Pröve. *An Introduction to different Types of Convolutions in Deep Learning*. https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d. 2017 (cited on page 248).

[74] Zhen Cui et al. "Deep Network Cascade for Image Super-resolution". In: *Computer Vision – ECCV 2014*. Edited by David Fleet et al. Cham: Springer International Publishing, 2014, pages 49–64 (cited on page 259).

[75] Gilad Freedman and Raanan Fattal. "Image and Video Upscaling from Local Self-examples". In: *ACM Trans. Graph.* 30.2 (Apr. 2011), 12:1–12:11. ISSN: 0730-0301. DOI: 10.1145/1944846.1944852. URL: http://doi.acm.org/10.1145/1944846.1944852 (cited on page 259).

[76] Daniel Glasner, Shai Bagon, and Michal Irani. "Super-Resolution from a Single Image". In: *ICCV*. 2009. URL: http://www.wisdom.weizmann.ac.il/~vision/SingleImageSR.html (cited on page 259).

[77] Hong Chang, Dit-Yan Yeung, and Yimin Xiong. "Super-resolution through neighbor embedding". In: *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.* 2004 (cited on page 259).

[78] William T. Freeman, Egon C. Pasztor, and Owen T. Carmichael. "Learning Low-Level Vision". In: *International Journal of Computer Vision* 40.1 (Oct. 2000), pages 25–47 (cited on page 259).

[79] Kui Jia, Xiaoou Tang, and Xiaogang Wang. "Image Transformation Based on Learning Dictionaries Across Image Spaces". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 35.2 (Feb. 2013), pages 367–380. ISSN: 0162-8828 (cited on page 259).

[80] Jianchao Yang et al. "Image Super-resolution via Sparse Representation". In: *Trans. Img. Proc.* 19.11 (Nov. 2010) (cited on page 259).

[81] Chao Dong et al. "Image Super-Resolution Using Deep Convolutional Networks". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 38.2 (Feb. 2016) (cited on pages 259, 261, 271).

[82] Justin Johnson, Alexandre Alahi, and Fei-Fei Li. "Perceptual Losses for Real-Time Style Transfer and Super-Resolution". In: *CoRR* abs/1603.08155 (2016). URL: http://arxiv.org/abs/1603.08155 (cited on page 259).

[83] Nister and Stewenius. *UKBench Dataset*. https://archive.org/details/ukbench. 2006 (cited on page 261).