O'REILLY®

Practical Deep Learning for Cloud & Mobile

Hands-On Computer Vision Projects Using Python, Keras & TensorFlow

Free Chapters

compliments of

dataiku

Anirudh Koul,
Siddha Ganju & Meher Kasam

TURN DATA SCIENCE

INTO ENTERPRISE AI

How does a data scientist go from creating and managing a few models to the hundreds (or thousands) required in an Enterprise AI world ?

ONE WORD: EFFICIENCY

That's why Dataiku brings:

Automation

Automate data-preprocessing, feature engineering, model training, & more

Flexibility

Use open-source tools and libraies (R,Python, MLlib, H2o, XGBoost,etc.) or build & run data services from scratch

Quick Modeling

Save time with parallel model building and autoML functionalities

data iku

Operationalization

Create a one-stop-shop in a visual interface to load models and serve requests in just a few clicks

Reproducibility

Any code you deploy is reproducible - projects won't fall through the cracks and fail

GET STARTED WITH DATAIKU

Practical Deep Learning for Cloud and Mobile

Hands-On Computer Vision Projects Using Python, Keras and TensorFlow

This Preview Edition of *Practical Deep Learning for Cloud and Mobile*, Chapters 2 and 3, is a work in progress. The final book is currently scheduled for release in October 2019 and will be available at oreilly.com once it is published.

Anirudh Koul, Siddha Ganju, and Meher Kasam

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Practical Deep Learning for Cloud and Mobile
by Anirudh Koul, Siddha Ganju, and Meher Kasam

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Nicole Tache and Jeff Bleiel
Production Editor: Christopher Faucher
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2019: First Edition

Revision History for the Early Release
2018-12-03: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492034865* for release details.

978-1-492-03479-7

[LSI]

Table of Contents

CHAPTER 1

Image Classification with Keras

A Note for Early Release Readers

This will be the second chapter of the final book.

Now that we have started our journey into deep learning, let's get our hands a little dirty.

If you have skimmed through deep learning literature, you may have come across a barrage of confusing academic explanations laced with scary mathematics. Don't worry. We will ease you into practical deep learning by showing how easily you can classify images with just a few lines of code.

In this chapter, we will introduce Keras, discuss its place in the deep learning landscape, and then use it to classify a few images using existing state-of-the-art classifiers. We will visually investigate how these classifiers operate by using heatmaps. With these heatmaps, we'll make a fun project where we classify objects in videos.

Where's the theory behind this, you might wonder? That will come later. Using this chapter as a foundation, we will delve deeper into the nuts and bolts of convolutional neural networks in the chapters that follow. After all, there's no better way to learn about and appreciate the components of a system than to dive right in and use them!

Introduction to Keras

Keras is a high-level neural network API designed to provide a simplified abstraction layer above several deep learning libraries such as TensorFlow, Theano, CNTK, PlaidML, MXNet, and more. This abstraction makes it easier and quicker to code deep neural networks with Keras than using the libraries themselves. While beginnerfriendly, Keras has enough functionality for quick prototyping and even professional-level, heavy-duty training. In this book, we will primarily use Keras with a TensorFlow backend.

Layers of Abstraction

One can draw parallels between the layers of abstraction in deep learning and those in computer programming. Much like how a computer programmer could write code in machine language (theoretically albeit painfully), assembly language, or higher-level languages, a deep learning practitioner can write training and inference programs using low-level frameworks such as CUDA, libraries like TensorFlow, or high-level frameworks such as Keras. In both cases, greater abstraction means greater ease of use, at the expense of flexibility.

Here are the building blocks for most deep learning libraries running on NVIDIA GPUs. The higher the level, the higher the abstraction.

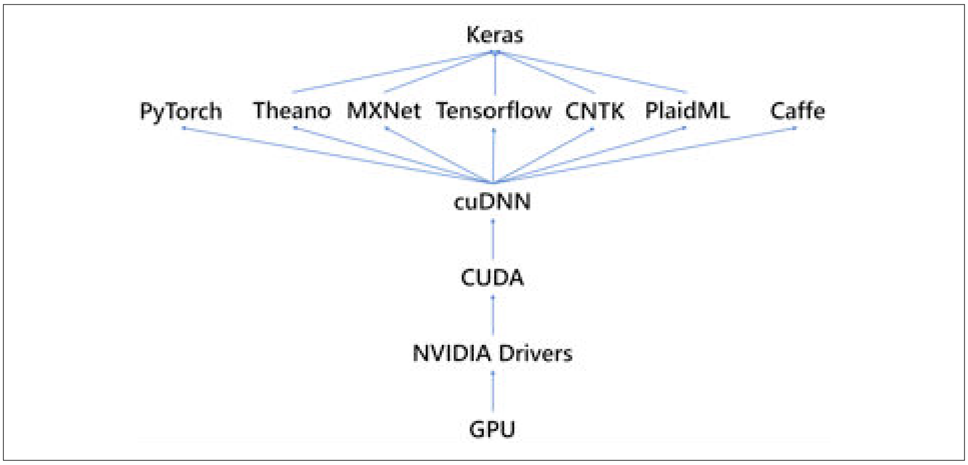


Figure 1-1. Levels of abstraction for different libraries. Abstraction increases in the direction of the arrows.

Since NVIDIA is the leading provider of GPUs used for deep learning, it provides drivers, CUDA, and cuDNN. Drivers help the operating system interface with the hardware GPU. CUDA, which stands for *Compute Unified Device Architecture*, provides direct access to the virtual instruction set of the GPU and the ability to execute parallel compute kernels. The CUDA Deep Neural Network library or cuDNN, which is built on top of CUDA, provides highly tuned implementations for standard routines and primitives for deep neural networks, such as convolution, pooling, normalization, and activation layers. Deep learning libraries like TensorFlow reuse these primitives and provide the inference engine (i.e. the system to compute predictions from the data). Finally, Keras provides another level of abstraction to further compact the necessary code to make this model.

Higher-level abstractions provide a form of leverage: you can do more with fewer lines of code. Let's test this theory out by comparing Keras with TensorFlow on one of the most famous tasks in deep learning: training a handwritten digit classifier (on the MNIST dataset) using a convolutional neural network. Using publicly available code in tutorials, we stripped off everything except for the core code and found that Keras requires roughly half the keystrokes when compared to TensorFlow code for the same task, as shown in Table 1-1.

Table 1-1. Example showing lines of code and character count at two abstraction levels. Higher levels of abstraction permit the same work to be accomplished with fewer lines and characters.

Library	Line count	Character count (no spaces)	Avg. character count per line
TensorFlow	31	2162	70
Keras	22	1018	46

In addition to being easier to use, Keras is quite popular within the open-source community. A good measure of an open-source project's popularity is the number of people who contribute to its codebase. As of March 2018, the following is a comparison of Keras to other libraries on GitHub:

Table 1-2. Stars and contributions to each framework's GitHub repo. It's worth remembering that many contributors to TensorFlow are Googlers, while Keras is a lot more "grassroots," with a diverse contributor base.

Library	Stars	Contributors
tensorflow/tensorflow	92150	1357
fchollet/keras	26744	638
BVLC/caffe	23159	264
Microsoft/CNTK	13995	173
dmlc/mxnet	13318	492
pytorch/pytorch	12835	414
deeplearning4j/deeplearning4j	8472	140
caffe2/caffe2	7540	176

Since its inception in 2015, Keras has consistently attracted more users, quickly becoming the framework of choice for deep learning after TensorFlow. Due a large user base and the open-source development community behind it, you can readily find many examples of tasks on Github and other documentation sources, making it easy for beginners to learn Keras. It is also versatile, allowing various deep learning backends to be used for training (like TensorFlow, CNTK, Theano, etc.) so it does not lock you into a specific ecosystem. Keras is therefore quite ideal for anyone who is making their foray into deep learning.

Keras in Practice

Predicting an Image's Category

As we covered in Chapter 1, image classification answers the question "does the image contain X?" where X can be virtually any category or class of objects. The process can be broken down into the following steps:

1. Load an image
2. Resize it to 224×224 size
3. Normalize the values of the pixel to the range [−1,1] a.k.a preprocessing
4. Select a pre-trained model

Here's some sample code for predicting categories of an image, which uses some of the helpful functions that Keras provides in its modules. As you do more coding, you'll often find that the layer or pre-processing step you need is already implemented in Keras, so remember to read the documentation.

In the GitHub repo, navigate to *code/chapter2*. All the steps we will be following are also detailed in the Jupyter notebook '*1_predict_class.ipynb*'.

We start by importing all the necessary modules from the Keras and Python packages.

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Model
from keras.layers import Input, Flatten, Dense, Dropout, GlobalAveragePooling2D
from keras.applications.resnet50 import ResNet50
import keras
from keras.applications.imagenet_utils import preprocess_input, decode_predictions
from keras.preprocessing import image
import numpy as np
import matplotlib.pyplot as plt
```

Next, we load and display the image that we want to classify.

```
img_path = '../../sample_images/cat.jpg'
img = image.load_img(img_path, target_size=(224, 224))
plt.imshow(img)
plt.show()
```

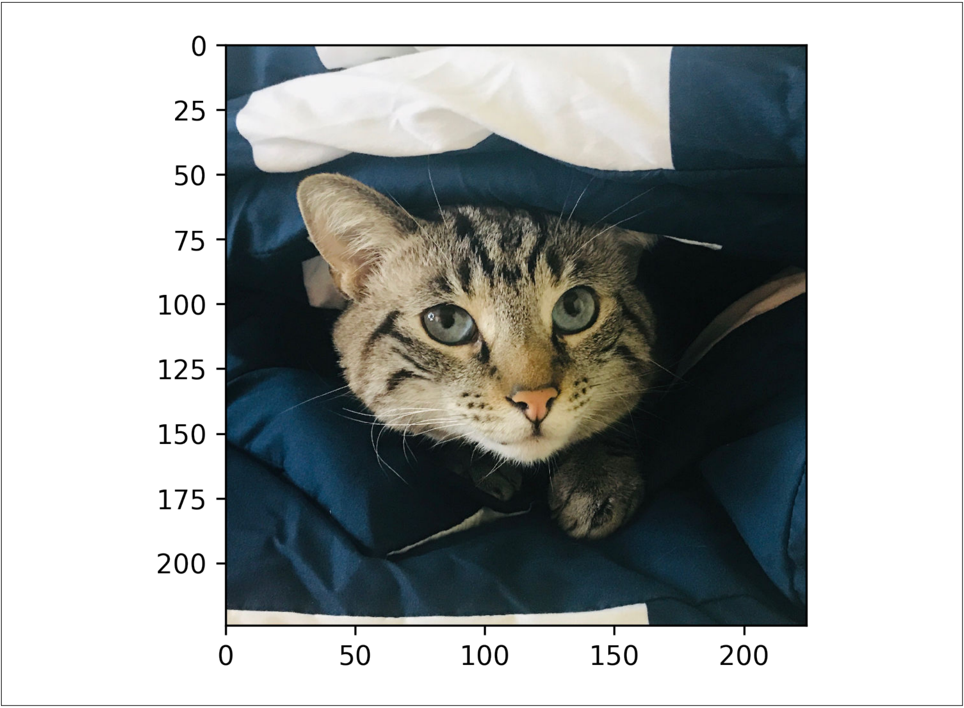


Figure 1-2. Plot showing the contents of the file cat.jpg.

It's a cat! And that's what our model should ideally be predicting.

A Brief Refresher on Images

Before we dive into how images are processed, it would be good to take a look at how images store information. At the most basic level, an image is a collection of pixels that are laid out in a rectangular grid. Depending on the type of image, each pixel can consist of 1 to 4 parts (also known as components or channels). With image we will be using, these components represent the intensities of Red, Green and Blue colors (RGB). They are typically 8 bits in length, so their values range between 0 and 255 (i.e. 2^8 -1).

In machine learning, it is empirically shown that taking an input within an arbitrary range and scaling it to the interval [0,1] or [−1,1] improves its chances of learning. This step is commonly referred to as normalization. Normalizing is one of the core steps in preprocessing images to make them suitable for deep learning.

We want to replicate the same preprocessing steps as the ones undergone during the original training of the pretrained models. Luckily, Keras provides a handy function, "`preprocess_input`" that does this for us. Before feeding any image to Keras, we want to convert it to a standard format. This standardization involves resizing the image to 224×224 pixels to ensure that the image size is uniform. Since the model has been trained to accept a batch of multiple images rather than one image at a time. Since we only have one image, we create a batch of one image to feed to our model. We can achieve that by adding an extra dimension (which represents the position of the image within that batch) at the start of our image matrix, as shown in the code below:

```
img_array = image.img_to_array(img)
img_batch = np.expand_dims(img_array, axis=0)
```

We then send the batch to the Keras preprocessing function.

The model we will use is 'ResNet-50', which belongs to the family of models that won the ImageNet 2015 competition in classification, detection and localization tasks. It also won the MS COCO 2015 competition in detection and segmentation tasks.

First we need to load the model. Instead of hunting for model architecture and pre-trained weights on the internet, Keras provides access to them in a single function call. When you run them for the first time, they will be downloaded from a remote server.

The resulting preprocessed image is input to the model variable, which gives us probability predictions for each class. Keras also provides the "`decode_predictions`" function, which tells us the probability of the image belonging to a variety of relevant category names.

```
model = ResNet50()
img_array = image.img_to_array(img)
img_batch = np.expand_dims(img_array, axis=0)
img_preprocessed = preprocess_input(img_batch)
prediction = model.predict(img_preprocessed)
decode_predictions(prediction, top=3)[0]
[('n02123045', 'tabby', 0.50009364),
 ('n02124075', 'Egyptian_cat', 0.21690978),
 ('n02123159', 'tiger_cat', 0.2061722)]
```

The predicted categories for this image are various felines. Why doesn't it simply predict the word 'cat' instead? The short answer is that the ResNet-50 model was trained on a granular dataset with many categories and does not include the more general 'cat'. We will soon investigate this dataset in more detail, but first let's load another sample image.

```
img_path = '../../sample_images/dog.jpg'
img = image.load_img(img_path, target_size=(224, 224))
plt.imshow(img)
plt.show()
```

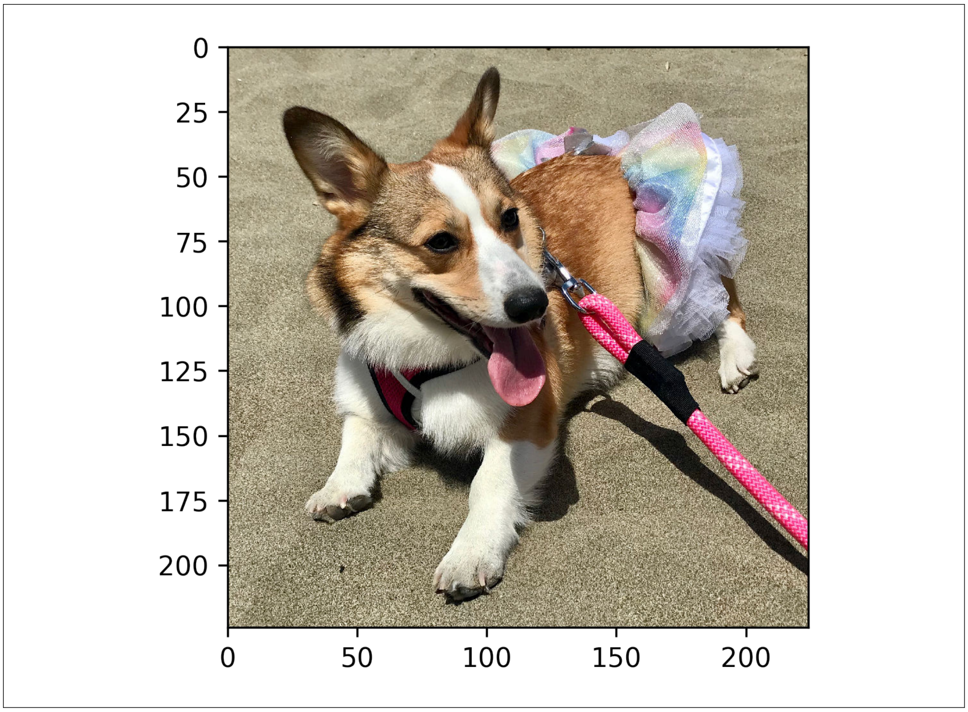


Figure 1-3. Plot showing the contents of the file dog.jpg.

And again we load our modules:

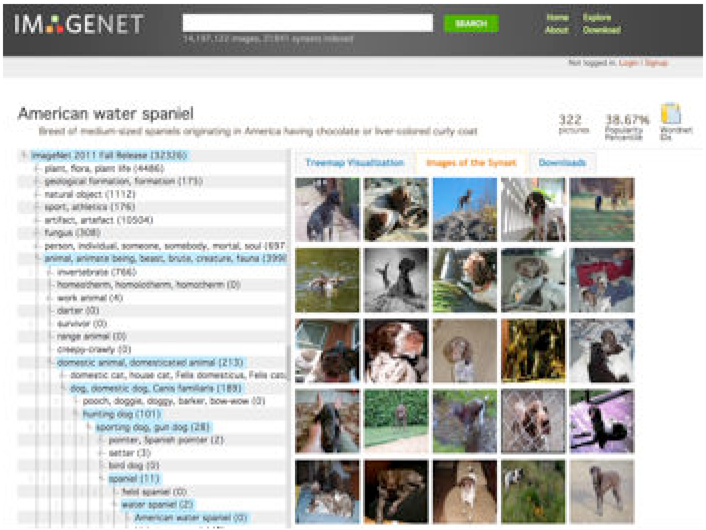
```
model = ResNet50()
img_array = image.img_to_array(img)
img_batch = np.expand_dims(img_array, axis=0)
img_preprocessed = preprocess_input(img_batch)
prediction = model.predict(img_preprocessed)
decode_predictions(prediction, top=3)[0]
[('n02113186', 'Cardigan', 0.71606547),
 ('n02113023', 'Pembroke', 0.26909366),
 ('n02110806', 'basenji', 0.0051731034)]
```

As expected, we get different breeds of the canine family (and not just the 'dog' category).

When using a pre-trained model, it is important to know the preprocessing steps involved in the training of the model. The same preprocessing steps need to be applied for images that are used for predictions. As an example, for a model previously trained in Caffe, preprocessing involves converting images from RGB to BGR and then zero-centering each color channel with respect to the ImageNet dataset without scaling (i.e., subtracting the mean value of each color channel in the ImageNet dataset).

ImageNet

So, let's investigate the ImageNet dataset on which ResNet was trained. ImageNet (*http://www.image-net.org/*), as the name suggests, is a network of images: an image database. It is organized in a hierarchical manner (like the WordNet hierarchy) such that the parent node encompasses a collection of images of all different varieties possible within that parent. For example, within the "animal" parent node, there are fish, birds, mammals, invertebrates, and so on. Each category has multiple subcategories, and these have sub-subcategories, and so on, with each only containing images that match the associated name.



IMAGENET
SEARCH
Home
Explore
About
Download
American water spaniel
Breed of medium-sized spaniels originating in America having chocolate or liver-colored curly coat
322 pictures
38.67% Popularity Percentile
Wordnet IDs
Treemap Visualization
Images of the Synset
Downloads
ImageNet 2011 Fall Release (32326)
plant, flora, plant life (4486)
geological formation, formation (175)
natural object (1112)
sport, athletics (176)
artifact, artefact (10504)
fungus (308)
person, individual, someone, somebody, mortal, soul (697
animal, animate being, beast, brute, creature, fauna (399
invertebrate (766)
homeotherm, homoiotherm, homotherm (0)
work animal (4)
darter (0)
survivor (0)
range animal (0)
creepy-crawly (0)
domestic animal, domesticated animal (213)
domestic cat, house cat, Felis domesticus, Felis cat
dog, domestic dog, Canis familiaris (189)
pooch, doggie, doggy, barker, bow-wow (0)
hunting dog (101)
sporting dog, gun dog (28)
pointer, Spanish pointer (2)
setter (3)
bird dog (0)
spaniel (11)
field spaniel (0)
water spaniel (2)
American water spaniel (0)

Figure 1-4. Screenshot of the categories and subcategories in the ImageNet Dataset. The category "American water spaniel" is 8 levels from the root. The dog category contains 189 total sub-categories in 5 hierarchical levels.

Visually, we developed the tree diagram in Figure 1-5 to understand the wide variety of high-level entities that the ImageNet dataset contains.

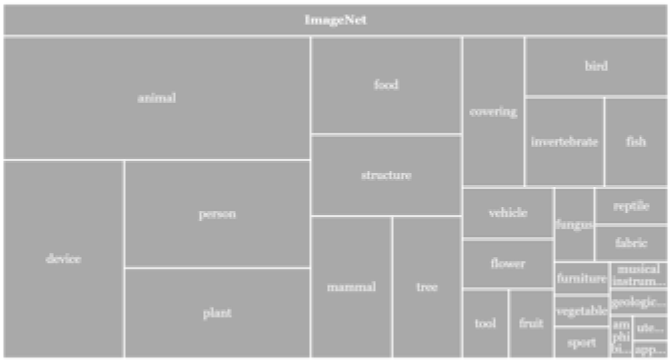


Figure 1-5. Tree map of ImageNet and its classes. This tree map shows the relative percentage of different categories that make up the ImageNet dataset.

The ImageNet dataset was the basis for the famous ImageNet Large Scale Visual Recognition Challenge (ILSVRC) which started in 2010 to benchmark progress in computer vision and challenge researchers to innovate on tasks including object classification. This dataset and challenge are considered to be the single biggest reason for drastic improvements in computer vision tasks, taking us from a nearly 30% error rate to ~2.5%, about as well as humans.

One of the key reasons for the fast pace of improvement was that researchers were openly sharing models trained on datasets like ImageNet and others. In the next section, we will learn about model reuse in more detail.

Analysis

A Model Zoo in Keras

A model zoo is a place where organizations or individuals place open-source models so others can use them. These models can be trained using a particular framework (e.g. Caffe, Tensorflow, etc), for a particular task (e.g. classification, detection, etc.), or trained on a particular dataset (e.g. ImageNet, Street View House Numbers dataset, etc). Any model zoo is a collection of different models trained on a set of similar constraints.

The tradition of model zoos started with Caffe, one of the first deep learning frameworks, developed at the University of California, Berkeley. Training a deep learning model from scratch on a multi-million-image database requires weeks of training time and lots of GPU computational energy, making it a difficult task. The research community recognized this as a bottleneck, and the organizations that participated in the ImageNet competition open-sourced their trained models on Caffe's website. Other frameworks soon followed suit.

If you are starting out on a new task, remember to first check if there is already an existing model that could be of assistance.

The Model Zoo in Keras is a collection of various architectures trained using the Keras framework on the ImageNet dataset. We tabulate their details in Table 1-3.

Table 1-3. Details of ImageNet trained models

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Inception-ResNet-V2	215 MB	0.804	0.953	55,873,736	572
Xception	88 MB	0.79	0.945	22,910,480	126
Inception-v3	92 MB	0.788	0.944	23,851,784	159
DenseNet-201	80 MB	0.77	0.933	20,242,984	201
ResNet-50	99 MB	0.759	0.929	25,636,712	168
DenseNet-169	57 MB	0.759	0.928	14,307,880	169
DenseNet-121	33 MB	0.745	0.918	8,062,504	121
VGG-19	549 MB	0.727	0.91	143,667,240	26
VGG-16	528 MB	0.715	0.901	138,357,544	23
MobileNet	17 MB	0.665	0.871	4,253,864	88

The column 'Top-1 Accuracy' indicates how many times the best guess was the correct answer, and the column 'Top-5 Accuracy' indicates how many times at least one out of five guesses was correct. The 'Depth' of the network indicates how many layers are present in the network. The 'Parameters' column indicates the size of the model: the more parameters, the "heavier" the model is, and the slower it is to make predictions. In this book, you will often see us use ResNet-50 (the most common architecture cited in research papers for high accuracy) and MobileNet (for good balance between speed, size, and accuracy).

What Does My Neural Network Think?

Now we will perform a fun experiment to try to understand why the neural network made a particular prediction. What part of an image made the neural network decide that it contained, for example, a cat or a dog? It would be helpful to be able to visualize the decision-making going on within the network, which we can do with a *heatmap*. This tool uses color to help visually identify the areas within an image that prompted a decision. "Hot" spots, represented by warmer colors (red, orange, and yellow) highlight the areas with the maximum signal, where a signal indicates the magnitude of contribution of an area in the image towards the category being predicted.

In the GitHub repo, navigate to *code/chapter2*. We will find a handy Jupyter notebook '*2_what_does_my_neural_network_think.ipynb*' which describes the following steps.

First, we will need to install the necessary libraries:

```
(practicaldl) $ pip3 install keras-vis --user
(practicaldl) $ pip3 install Pillow --user
(practicaldl) $ pip3 install matplotlib --user
```

We then run the visualization script on a single image to generate the heatmap for it:

```
(practicaldl) $ python3 visualization.py --process image --path ../sample_images/dog.jpg
```

You should see a newly created file called *dog_output.jpg* that shows a side-by-side view of the original image and its heatmap. As you can see from Figure 1-6, the right half of the image indicates the "areas of heat" along with the correct prediction of a 'Cardigan (Welsh Corgi)'.

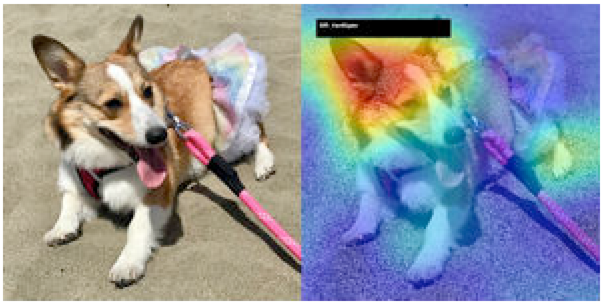


Figure 1-6. Original image of a dog and its generated heatmap.

Next, we want to visualize the heatmap for frames in a video. For that, we need `ffmpeg`, an open source multimedia framework. You can find the download binary as well as the installation instructions for your operating system at www.ffmpeg.org.

We will use `ffmpeg` to split up a video into individual frames and then run our visualization script on each of those frames. We must first create a directory to store these frames and use its name into the `ffmpeg` command.

```
(practicaldl) $ mkdir kitchen
(practicaldl) $ ffmpeg -i video/kitchen_input.mov -vf fps=25 kitchen/thumb%04d.jpg -hide_banner
```

We then run the visualization script with the path of the directory containing the frames from the previous step:

```
(practicaldl) $ python3 visualization.py --process video --path kitchen/
```

You should see a newly created *kitchen_output* directory that contains all the heatmaps for the frames from the input directory.

Finally, compile a video from those frames using `ffmpeg`:

```
(practicaldl) $ ffmpeg -framerate 25 -i kitchen_output/result_%04d.jpg kitchen_output.mp4
```

Perfect! Imagine generating heatmaps to analyze the strong points and shortfalls of your trained model or a pretrained model. Don't forget to post your videos on Twitter with the hashtag #PracticalDL!

Summary

In this chapter, we got a glimpse of the deep learning universe using Keras. It's an easy-to-use yet powerful framework that we'll use in the next several chapters. We observed that there is often no need to collect millions of images and use powerful GPUs to train a custom model, because we can use a pretrained model to predict the category of an image. By diving deeper into datasets like ImageNet, we learned the kinds of categories these pretrained models can predict. We also learned about finding these models in model zoos that exist for most frameworks.

In the next chapter, we will explore how we can tweak an existing pretrained model to make predictions on classes of input for which it was not originally intended. As with the current chapter, our approach is geared toward obtaining output without needing millions of images and lots of hardware resources to train a classifier.

CHAPTER 2

Cats vs Dogs - Transfer Learning in 30 lines with Keras

A Note for Early Release Readers

This will be the third chapter of the final book.

Imagine you want to learn how to play the ukulele. If you have no musical background, and you are starting fresh with the ukulele as your very first instrument, it'll take you a few months to get proficient at playing it. On the other hand, if you are accustomed to playing the guitar, it might just take a week, due to how similar the two instruments are. Taking the learnings from one task and fine-tuning them on a similar task is something we often do in real life. The more similar the two tasks are, the easier it is to adapt the learnings from one task to the other.

This phenomenon from real life can also be applied to the world of deep learning. It is relatively quick to start with a pretrained model, reuse the knowledge that it learned during its training, and adapt it to the task at hand. This process is known as **Transfer Learning**.

In this chapter, we will use transfer learning to modify existing models by training our own classifier in minutes with Keras. Otherwise, training from scratch would have taken anywhere from days to weeks. By the end, you will have several tools in your arsenal to create high-quality image classifiers on any topic.

Transfer Learning

Before we discuss the process of transfer learning, let's quickly take a step back and review the primary reasons for the boom in deep learning:

1. Availability of bigger and better quality datasets like ImageNet.
2. Better compute available i.e. faster and cheaper GPU.
3. Better algorithms (model architecture, optimizer, and training procedure).
4. Availability of pretrained models that have taken months to train but can be quickly reused.

The last point is probably one of the biggest reasons for widespread adoption of deep learning into the masses. If every training task still took a month, not more than a handful of researchers with deep pockets would be working in this area. Thanks to transfer learning, the underappreciated hero of training models, you can now modify an existing model to suit your task in as little as a few minutes.

For example, the pretrained ResNet-50 model which is trained on ImageNet can predict feline and canine breeds, among thousands of other categories. So, if we just want to classify between the high level 'cat' and 'dog' categories (and not the lower level breeds), we can start with the ResNet-50 model and quickly retrain this model to classify 'cats' and 'dogs'. All we have to do is to show it a dataset with these two categories during training, which should take anywhere between a few minutes to hours. In comparison, if you had to train a cat vs dog model without a pre-trained model, this might take several days.

Solving the world's most pressing computer vision problem

It's early 2014 and Microsoft Research (MSR) is figuring out how to solve the world's most pressing problem - "Differentiating cats and dogs"! Yes, you heard it right. To facilitate this research, MSR released the Asirra (Animal Species Image Recognition for Restricting Access) dataset. The big picture behind the Asirra dataset is to develop a hard enough CAPTCHA system. Over 3 million images, labeled by animal shelters throughout the United States were provided by Petfinder.com to Microsoft Research. When this problem was initially introduced, the highest possible accuracy attained was around 80%. And by using deep learning, in just a few weeks, it went to 98%! This (now relatively easy) task shows the power of deep learning and the many things it can achieve.

Understanding Different Layers in a CNN in the Context of Transfer Learning

Imagine you wanted to detect a human face. You might want to use a convolutional neural network (CNN) to classify an image and detect whether it contains a face. Such a CNN would be composed of several sequential layers. The image is fed to this network and the resulting output is a list of object classes and their probabilities (categories like a ball - 90%, dog - 56%, etc.). If the output for an image contains a 'face' class with 70% probability, we conclude that there is a 70% likelihood that the image contains a human face.

The layers in CNN represent mathematical operations. Each layer can take in an input and gives an output. The output of one layer becomes the input to the subsequent layer. The complexity and power of what a layer can recognize increases as we go closer to the final layers. Conversely, the reusability of a layer decreases as we go closer to the output.

Lower level layers (Figure 2-1) (layers that are closer to the input image) get "activated" for simpler shapes e.g. edges and curves. They are reusable in many classification tasks. Middle-level layers (Figure 2-1) get activated for more complex shapes e.g. eyes, nose, lips. These layers are not nearly as reusable as the lower level layers. And higher level layers (Figure 2-1) get activated for even more complex shapes e.g. most of the human face. These layers tend to be more task-specific and thus the least reusable across other image classification problems.

Convolutional Neural Networks are composed of convolutional filters. As the name implies - they act as a sieve of information, only letting something they can recognize "pass through". We say that they got "activated" for that information. For irrelevant information, their output is close to zero. CNNs are the bouncers of the deep learning world!

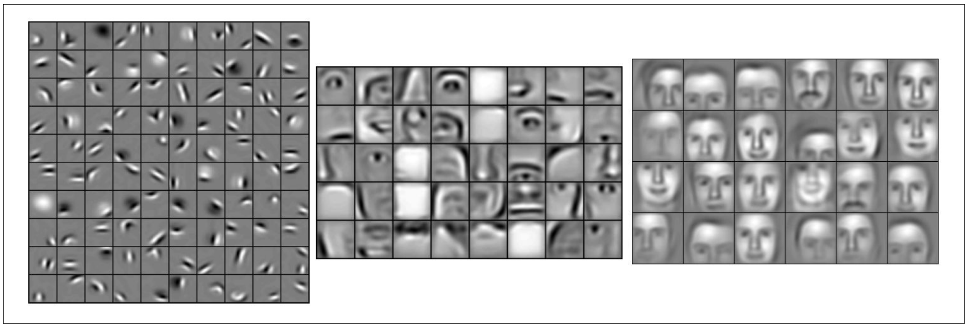


Figure 2-1. (a) Lower level activations, followed by (b) mid level activations and (c) upper layer activations. Source: Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations, Lee et al, ICML 2009 (http://ai.stanford.edu/~ang/papers/icml09-ConvolutionalDeepBeliefNetworks.pdf)

If you want to transfer knowledge from one model to another, you want to reuse more of the “general” layers, and fewer of the “specific” layers. In other words, you want to remove the last few layers so that you can utilize the more generic ones, and add layers that are geared towards your specific classification task. This is how transfer learning is achieved.

While transfer learning is the concept, fine-tuning is the implementation process. Fine-tuning, as the name suggests, typically involves tweaking the weights of the last few of layers in the model. You will often hear data scientists saying, “I fine-tuned the model.” which means they took a pretrained model, froze the lower layers, trained the upper part of the network on the new dataset they had (thereby modifying the weights of these upper layers).

How many layers of a CNN should we fine-tune? This can be guided by the following two factors:

1. How much data do we have?

   If you have a couple hundred labeled images, it would be hard to train and test a network from scratch. Hence, you should fine-tune the last few layers. But, if you had a million labeled images, it would be feasible to fine-tune all layers of the network, and if necessary, train from scratch (i.e build the model architecture with random weights). So, the amount of task-specific data dictates whether or not, and how much you can fine-tune.

2. How similar is the data?

   If the task-specific data is similar to the data used for the pretrained network, then you can fine-tune the last few layers. But if your task is identifying different bones in an X-ray image, and you want to start out from an ImageNet trained network, the high dissimilarity between regular ImageNet and X-ray images would essentially require all layers to be trained.

To summarize here is an easy to follow cheat sheet:

Table 2-1. When and how to fine-tune

	High similarity amongst datasets	Low similarity amongst datasets
Large amount of training data	Fine-tune all layers	Train from scratch, or fine-tune all layers
Small amount of training data	Fine-tune last few layers	Tough luck! Train on a smaller network with heavy data augmentation

Often when you attempt to train a neural network on a small amount of data, the result is a model that performs extremely well on the training data itself, but makes rather poor predictions on unseen data. Such a model would be described as an "overfitted" model and the problem itself is known as "overfitting".

Overfitting is common when you have little training data. To avoid this problem, fine-tune only the last couple of layers.

In chapter 4 we will discuss fine-tuning in more detail.

Building a Custom Classifier in Keras with Transfer Learning

As promised, it's time to build our state of the art classifier in 30 lines or fewer. At a high level, we will follow the steps shown below:

1. **Organize the data:** Download labeled images of cats and dogs. Then divide the images into training and validation folders.
2. **Set up the configuration:** Define a pipeline for reading data, including preprocessing the images (e.g. resizing) and batching multiple images together.
3. **Augment the data:** In the absence of a ton of training images, make small changes (augmentation) like rotation, zooming, etc to increase variation in training data.
4. **Define the model:** Take a pre-trained model, remove the last few layers, and append a new classifier layer. Freeze the weights of original layers (i.e. make them unmodifiable). Select an optimizer algorithm and a metric to track (like accuracy).

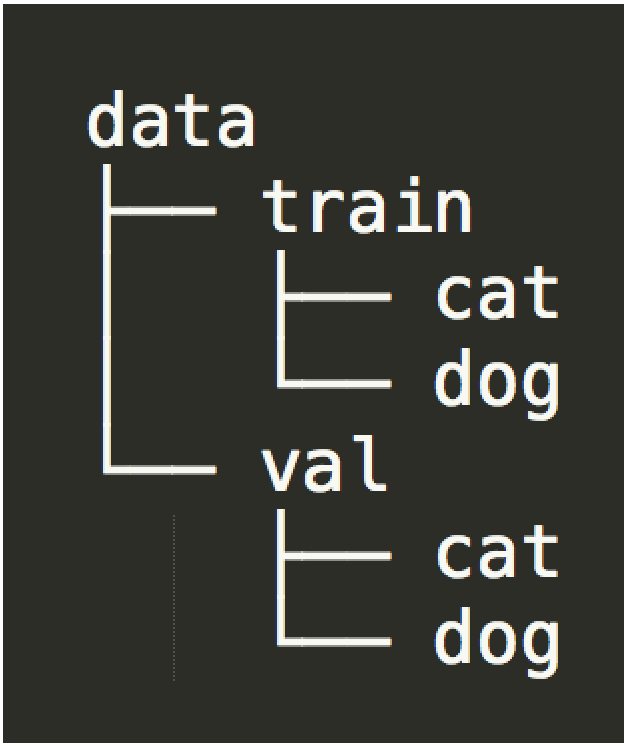
5. **Train and test:** Start training for a few iterations. Save the model to eventually load inside any application for predictions.

Let's explore this process in detail.

Organize the data

Before training, we need to store our dataset in the right folder structure. We'll divide the images into two sets – training and validation. Classification accuracy on previously unseen images (in the validation folder) is a good proxy for how the classifier would perform in the real world. Ideally, the more training images, the better the learning will be. And the more validation images, the better our classifier would perform in the real-world.

For an image file, Keras will automatically assign the name of the class (category) based on its parent folder name. Here's the ideal structure to recreate:



```
data
├── train
│   ├── cat
│   └── dog
└── val
    ├── cat
    └── dog
```

Figure 2-2. Example directory structure of the training and validation data for different classes.

In Linux/Mac, the following lines of command can help achieve this directory structure:

```
# Preparing the data
# Download from https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/download/train.zip
unzip train.zip
mv train data
cd data
mkdir train val
mkdir train/cat train/dog
mkdir val/cat val/dog
```

The 25,000 files inside the data folder are prefixed with 'cat' and 'dog'. Now move the files into their respective directories. To keep our initial experiment short, we'll pick 250 random files per class and place them in training and validation folders. You can increase/decrease this number anytime, to experiment with a trade-off between accuracy and speed.

```
ls | grep cat | sort -R | head –250 | xargs -I {} mv {} train/cat/
ls | grep dog | sort -R | head –250 | xargs -I {} mv {} train/dog/
ls | grep cat | sort -R | head –250 | xargs -I {} mv {} val/cat/
ls | grep dog | sort -R | head –250 | xargs -I {} mv {} val/dog/
```

Set up the Configuration

To start off with our Python program, we'll begin with importing the necessary packages.

```
import keras
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Model
from keras.layers import Input, Flatten, Dense, Dropout, GlobalAveragePooling2D
from keras.applications.mobilenet import MobileNet
import math
```

Place all configuration up-front, which you can modify based on your dataset.

```
TRAIN_DATA_DIR = 'data/train_data/'
VALIDATION_DATA_DIR = 'data/val_data/'
TRAIN_SAMPLES = 500
VALIDATION_SAMPLES = 500
NUM_CLASSES=2
IMG_WIDTH, IMG_HEIGHT = 224, 224
BATCH_SIZE=64
```

Number of classes

With two classes to distinguish between, we can treat this problem as:

1. A binary classification task, or

2. A multi-class classification task.

Binary Classification. As a binary classification task, it's important to note that 'cat vs dog' is really 'cat vs non-cat'. A dog would be classified as a 'non-cat' much like a desk or a ball would. For a given image, the model will give a single probability value corresponding to the cat class (and hence the probability of not-cat is 1-p(cat)). If the probability is higher than 0.5, then we predict it as a cat, otherwise not-cat. To keep things simple, we assume it's guaranteed that the test set would only contain images of either cats or dogs. Since 'cat vs non-cat' is a binary classification task, we set the number of classes to 1 i.e. 'cat'. Anything that cannot be classified as 'cat' will be classified as 'non-cat'.

Keras processes the input data in the alphabetical order of the folder names. Since 'cat' comes before 'dog' alphabetically, our 1 class for prediction is 'cat'. For a multi-class task, we can apply the same concept and infer each class id based on the folder sort order.

Multi-Classification. Unfortunately, binary classification would not work where there is no guarantee that the test data would contain only pictures of either cats or dogs. As explained before, even a ball or a sofa would be classified as a dog. For a real-world scenario, treating this as a multi-classification task is far more useful. As a multi-classification task, we predict separate probability values for each class, and the highest one is our winner. In the case of cat vs dog, we set the number of classes to 2. To keep our code reusable for future tasks, we will treat this as a multi-classification task.

Batch size

At a high-level, the CNN training process includes the following steps:

1. Make predictions on images (forward pass).
2. Determine which predictions were incorrect and propagate back the difference between the prediction and the true value (backpropagation).
3. Rinse and repeat till the predictions become sufficiently accurate.

It's quite likely that the initial iteration would have close to 0% accuracy. Repeating the process several times, however, can yield a highly accurate model (> 90%).

The batch size defines how many images are seen by the CNN at a time. It's important that each batch have a good variety of images from different classes in order to prevent large fluctuations in the accuracy metric between iterations. A sufficiently large batch size would be necessary for that. However, it's important not to set the batch size too large for a couple of reasons. First, if the batch size is too large, you could end up crashing the program due to lack of memory. Second, the training process would be slower. Usually, batch sizes are set as powers of 2. 64 is a good number to start with for most problems and you can play with the number by increasing/decreasing it.

Data Augmentation

Usually, when you hear deep learning, you associate it with millions of images. But 500 images like what we have might be a low number for real-world training. Now, these deep neural networks are powerful, a little too powerful for small quantities of data. The danger of a limited set of training images is that the neural network might memorize your training data, and show great prediction performance on the training set, but bad accuracy on the validation set. In other words, the model has overtrained and does not generalize on previously unseen images. And we don't want that, right?

There are often cases where there's not enough data available. Perhaps you're working on a niche problem and data is hard to come by. There are a few ways you can artificially augment your dataset:

1. Rotation: In our example, we might want to rotate the 500 images randomly by 20 degrees in either direction, yielding up to 20000 possible unique images.
2. Random Shift: Shift the images slightly to the left, or to the right.
3. Zoom: Zoom in and out slightly of the image

By combining rotation, shifting and zooming, the program can generate almost infinite unique images. This important step is called data augmentation. Keras provides `ImageDataGenerator` function that augments the data while it is being loaded from the directory. Example augmentations generated by the imgaug (*https://github.com/aleju/imgaug*) for a sample image are shown in Figure 2-3.

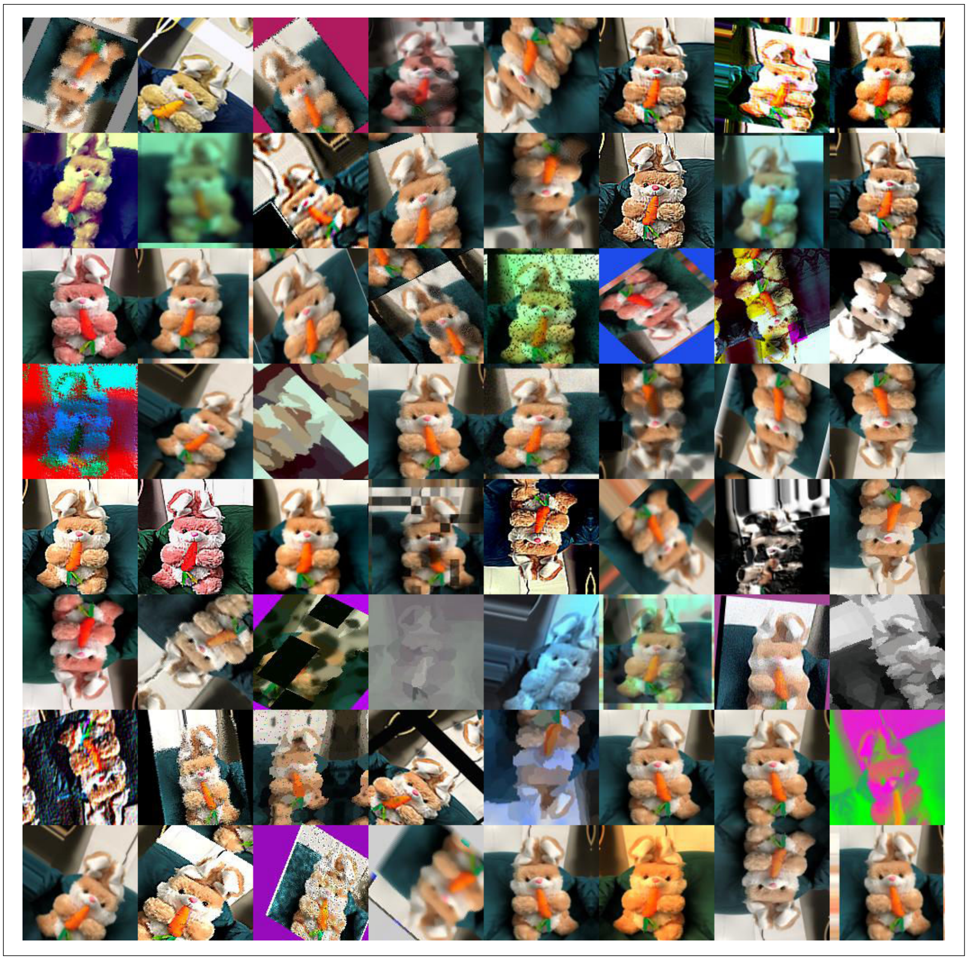


Figure 2-3. Possible image augmentations generated from a single image by imgaug library

Colored images usually have 3 channels - red, green, and blue. Each channel has an intensity value ranging from 0 to 255. To normalize it (i.e. scale down the value to between 0 and 1), we will divide each pixel by 255.

```
train_datagen = ImageDataGenerator(rescale=1./255,
                                   rotation_range=20,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   zoom_range=0.2)
val_datagen = ImageDataGenerator(rescale=1./255)
```

Sometimes knowing the label of a training image can be useful in determining appropriate ways of augmenting it. For example, when training a digit recognizer, you might be okay with augmentation by flipping vertically for an ‘8’ digit image, but not for ‘6’ and ‘9’.

Unlike in the training set, we don’t want to augment our validation dataset. The reason is that with dynamic augmentation, the validation set would keep changing in each iteration, and the resulting accuracy metric would be inconsistent and hard to compare across other iterations.

Time to load the data from its directories. Training one image at a time can be pretty inefficient, so we can batch them into groups. To introduce more randomness during the training process, we’ll keep shuffling the images in each batch. To bring reproducibility during multiple runs of the same program, we’ll give the random number generator a seed value.

```
train_generator = train_datagen.flow_from_directory(
                        TRAIN_DATA_DIR,
                        target_size=(IMG_WIDTH, IMG_HEIGHT),
                        batch_size=BATCH_SIZE,
                        shuffle=True,
                        seed=12345,
                        class_mode='categorical')
validation_generator = val_datagen.flow_from_directory(
                        VALIDATION_DATA_DIR,
                        target_size=(IMG_WIDTH, IMG_HEIGHT),
                        batch_size=BATCH_SIZE,
                        shuffle=False,
                        class_mode='categorical')
```

Model Definition

Now that the data is taken care of, we come to the most crucial component of our training process - the model. In the code below, we will reuse a CNN previously trained on the ImageNet dataset (MobileNet in our case), throw away the last layer (ImageNet specific classifier/Softmax), and replace it with our own classifier that is suited to the task at hand. For transfer learning, we’ll “freeze” the weights of the original model, i.e. set those layers as unmodifiable, so only the layers of the new classifier (that we add) can be modified. We use MobileNet here to keep things fast, but this method will work just as well for any neural network. Don’t worry about the specific layers, we’ll dig deeper into those details in chapter 4.

```
def model_maker():
    base_model = MobileNet(include_top=False, input_shape = (IMG_WIDTH,IMG_HEIGHT,3))
    for layer in base_model.layers[:]:
        layer.trainable = False # Freeze the layers
    input = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3))
    custom_model = base_model(input)
    custom_model = GlobalAveragePooling2D()(custom_model)
    custom_model = Dense(64, activation='relu')(custom_model)
    custom_model = Dropout(0.5)(custom_model)
    predictions = Dense(NUM_CLASSES, activation='softmax')(custom_model)
    return Model(inputs=input, outputs=predictions)
```

Train and Test

Training Parameters

With both the data and model ready, all we have left to do is train the model. This is also known as fitting the model to the data. For training any model, we need to pick a loss function, an optimizer, initial learning rate and a metric.

- **Loss function:** The loss function is the objective being minimized. For example, in a task to predict house prices, the loss function could be the mean squared error.
- **Optimizer:** This is an optimization algorithm that helps minimize the loss function. We'll choose Adam, one of the fastest optimizers out there.
- **Learning rate:** This defines how quickly or slowly you update the weights during training. Choosing an optimal learning rate is crucial - a big value can cause the training process to jump around, missing the target. On the other hand, a tiny value can cause the training process to take ages to reach the target. We'll keep it at 0.001 for now.
- **Metric:** Choose a metric to judge the performance of the trained model. Accuracy is a good explainable metric, especially when the classes are not imbalanced, i.e. roughly equal in size. Note that this metric is not used during training to maximize or minimize an objective.

In the following piece of code, we create the custom model using the `model_maker` function we wrote earlier. We use the parameters described here to customize this model further for our task of cats vs dogs.

```
model = model_maker()
model.compile(loss='categorical_crossentropy',
              optimizer= keras.optimizers.Adam(lr=0.001),
              metrics=['acc'])
model.fit_generator(train_generator,
                    steps_per_epoch = math.ceil(float(TRAIN_SAMPLES) / BATCH_SIZE),
                    epochs=10,
                    validation_data = validation_generator,
                    validation_steps = math.ceil(float(VALIDATION_SAMPLES) / BATCH_SIZE))
```

You might have noticed the term 'epoch' here. One epoch means a full training step where the network has gone over the entire dataset. One epoch may consist of several mini-batches.

Start the Engines

Run this program and let the magic begin. If you don't have a GPU, brew a cup of coffee while you wait. You'll notice 4 statistics - loss and accuracy on both the training and validation data. You are rooting for the val_acc.

```
> Epoch 1/100 7/7 [====] - 5s - loss: 0.6888 - acc: 0.6756 - val_loss: 0.2786 - val_acc: 0.9018
> Epoch 2/100 7/7 [====] - 5s - loss: 0.2915 - acc: 0.9019 - val_loss: 0.2022 - val_acc: 0.9220
> Epoch 3/100 7/7 [====] - 4s - loss: 0.1851 - acc: 0.9158 - val_loss: 0.1356 - val_acc: 0.9427
> Epoch 4/100 7/7 [====] - 4s - loss: 0.1509 - acc: 0.9341 - val_loss: 0.1451 - val_acc: 0.9404
> Epoch 5/100 7/7 [====] - 4s - loss: 0.1455 - acc: 0.9464 - val_loss: 0.1637 - val_acc: 0.9381
> Epoch 6/100 7/7 [====] - 4s - loss: 0.1366 - acc: 0.9431 - val_loss: 0.2319 - val_acc: 0.9151
> Epoch 7/100 7/7 [====] - 4s - loss: 0.0983 - acc: 0.9606 - val_loss: 0.1420 - val_acc: 0.9495
> Epoch 8/100 7/7 [====] - 4s - loss: 0.0841 - acc: 0.9731 - val_loss: 0.1423 - val_acc: 0.9518
> Epoch 9/100 7/7 [====] - 4s - loss: 0.0714 - acc: 0.9839 - val_loss: 0.1564 - val_acc: 0.9509
> Epoch 10/100 7/7 [====] - 5s - loss: 0.0848 - acc: 0.9677 - val_loss: 0.0882 - val_acc: 0.9702
```

All it took was 5 seconds in the very first epoch to reach 90% accuracy on the validation set, with just 500 training images. Whoa! And by the 10th step, we observe about 97% validation accuracy. That's the power of transfer learning. Without having the model previously trained on ImageNet, getting a decent accuracy on this task would have taken (1) training time anywhere between a couple of hours to a few days (2) tons of more data to get decent results.

That's all the code you need to train a state-of-the-art classifier on any problem. Place your data into folders with the name of the class, and change the corresponding values in the configuration variables. In case your task has more than 2 classes, you should use `categorical_crossentropy` as the loss function and replace the activation function in the last layer with `softmax`. Table 2-2 below illustrates this.

Table 2-2. Deciding the loss and activation type based on the task

Classification type	Class Mode	Loss	Activation on the last layer
1 or 2 class	binary	binary_crossentropy	sigmoid
Multi-class, single label	categorical	categorical_crossentropy	softmax
Multi-class, multi-label	categorical	binary_crossentropy	sigmoid

Before we forget, save the model you trained.

```
model.save('model.h5')
```

An 'activation layer' is a layer that essentially acts as a filter - based on a condition, it will allow some values to pass and some to be filtered out. For example,

Test the Model

Now that you have a trained model, you might eventually want to use it later for your application. We can now load this model anytime and classify an image. `load_model`, as the name suggests loads the model.

```
from keras.models import load_model
model = load_model('model.h5')
```

Obscure trivia warning - MobileNet happens to use ReLu6 which has an upper bound with value 6. Standard ReLu, on the other hand, does not have an upper bound.

Now let's try loading our original sample images and see what results we get.

```
img_path = '../../sample_images/dog.jpg'
img = image.load_img(img_path, target_size=(224,224))
img_array = image.img_to_array(img)
expanded_img_array = np.expand_dims(img_array, axis=0)
preprocessed_img = expanded_img_array / 255. # Preprocess the image
prediction = model.predict(preprocessed_img)
print(prediction)
print(validation_generator.class_indices)
[[0.9967706]]
{'dog': 1, 'cat': 0}
```

Printing the value of the probability, we see that it 0.996. This is the probability of the given image belonging to the class '1', which is a dog. Since the probability is greater than 0.5, the image is predicted as a dog.

In Chapter 1, we used `preprocess_input` while classifying a single image from an ImageNet pretrained model. In the code above, we instead just normalized the image data by dividing with 255. This is because `preprocess_input` function is built specifically for ImageNet data, and unless we use it for retraining on the new data too, it should not be used again. Remember to use the exact same steps for reprocessing both the training and test images.

That's all that you need to train your own classifiers. Throughout this book, you can expect to reuse this code for training with minimal modifications. You can also reuse this code in your own projects. Play with the number of epochs and images, and observe how it affects the accuracy. Also, play with any other data you can find online. You have the power!

Analyzing the results

With our trained model, we can analyze how it's performing over the validation dataset. Beyond the simpler accuracy metrics, looking at the actual images of mispredictions should give an intuition on whether the example was truly hard or if our model is not sophisticated enough.

There are 3 questions we want to answer for each category (cat, dog):

- Which images are we most confident about being a cat/dog?
- Which images are we least confident about being a cat/dog?
- Which images have incorrect predictions in spite of being highly confident?

Before we get to that, let's get predictions over the entire validation dataset. First, we set the pipeline configuration correctly:

```
#####################
##### VARIABLES #####
#####################

IMG_WIDTH, IMG_HEIGHT = 224, 224
VALIDATION_DATA_DIR = 'data/val_data/'
VALIDATION_BATCH_SIZE = 64

#####################
## DATA GENERATORS ##
#####################

validation_datagen = ImageDataGenerator(
        rescale=1./255)

validation_generator = validation_datagen.flow_from_directory(
        VALIDATION_DATA_DIR,
        target_size=(IMG_WIDTH, IMG_HEIGHT),
        batch_size=VALIDATION_BATCH_SIZE,
        shuffle=False,
        class_mode='categorical')

ground_truth = validation_generator.classes
```

Then we get the predictions.

```
predictions = model.predict_generator(validation_generator)
```

To make our analysis easier, we make a dictionary storing the image index to the prediction and ground truth (the expected prediction) for each image.

```
# prediction_table is a dict with index, prediction, ground truth
prediction_table = {}
for index, val in enumerate(predictions):
    #get argmax index
    index_of_highest_probability = np.argmax(val)
    value_of_highest_probability = val[index_of_highest_probability]
    prediction_table[index] = [value_of_highest_probability, index_of_highest_probability,
    ground_truth[index]]
assert len(predictions) == len(ground_truth) == len(prediction_table)
```

For the next two code blocks, we provide boilerplate code which we will be reusing regularly throughout the book.

We'll make a helper function to find the images with the highest/lowest probability value for a given category.

```
# Helper function that finds images that are closest
# Input parameters:
#   prediction_table: dictionary from the image index to the prediction
and ground truth for that image
#   get_highest_probability: boolean flag to indicate if the results
need to be highest (True) or lowest (False) probabilities
#   label: id of category
#   number_of_items: num of results to return
#   only_false_predictions: boolean flag to indicate if results
should only contain incorrect predictions
def get_images_with_sorted_probabilities(prediction_table, get_highest_probability,
 label, number_of_items, only_false_predictions=False):
    sorted_prediction_table = [ (k, prediction_table[k]) for k in
    sorted(prediction_table, key=prediction_table.get, reverse= get_highest_probability)]
    result = []
    for index, key in enumerate(sorted_prediction_table):
        image_index, [probability, predicted_index, gt] = key
        if predicted_index == label:
            if only_false_predictions == True:
                if predicted_index != gt:
                    result.append([image_index, [probability, predicted_index, gt] ])
            else:
                result.append([image_index, [probability, predicted_index, gt] ])
        if len(result) >= number_of_items:
            return result
```

We'll also make two functions to display images.

```
# Helper functions to plot the nearest images given a query image
def plot_images(filenames, distances, message):
    images = []
    for filename in filenames:
        images.append(mpimg.imread(filename))
    plt.figure(figsize=(20,15))
    columns = 5
    for i, image in enumerate(images):
        ax = plt.subplot(len(images) / columns + 1, columns, i + 1)
        ax.set_title( "\n\n"+  filenames[i].split("/")[-1]+"\n"+"\nProbability: " +
        str(float("{0:.2f}".format(distances[i]))))
        plt.suptitle( message, fontsize=20, fontweight='bold')
        plt.axis('off')
        plt.imshow(image)


def display(sorted_indicies, message):
    similar_image_paths = []
    distances = []
    for name, value in sorted_indicies:
        [probability, predicted_index, gt] = value
        similar_image_paths.append(VALIDATION_DATA_DIR + fnames[name])
        distances.append(probability)
    plot_images(similar_image_paths,distances, message)
```

Now the fun starts!

Which images are we most confident contain dogs? Let's find images with the highest prediction probability (i.e. closest to 1.0) with the predicted class dog (i.e. 1).

```
# Most confident predictions of 'dog'
indices = get_images_with_sorted_probabilities(prediction_table, True, 1, 10, False)
message = 'Images with the highest probability of containing dogs'
display(indices[:10], message)
```

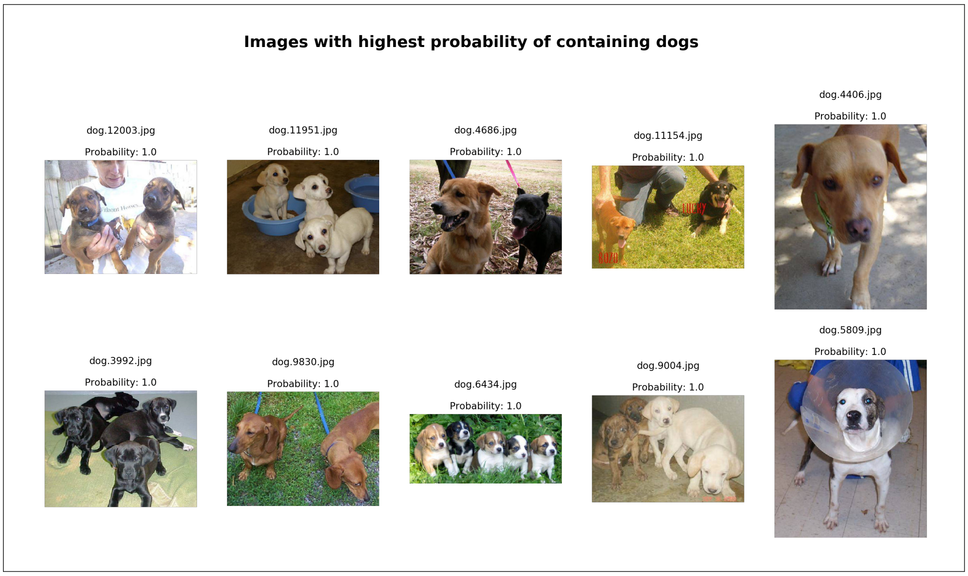


Figure 2-4. Images with the highest probability of containing dogs.

These images are indeed very dog-like. One of the reasons why the probability is so high can be attributed to containing multiple dogs, as well as clear, unambiguous views. Now let's try to find which images are we least confident of containing dogs?

```
# Least confident predictions of 'dog'
indices = get_images_with_sorted_probabilities(prediction_table, False, 1, 10, False)
message = 'Images with the lowest probability of containing dogs'
display(indices[:10], message)
```

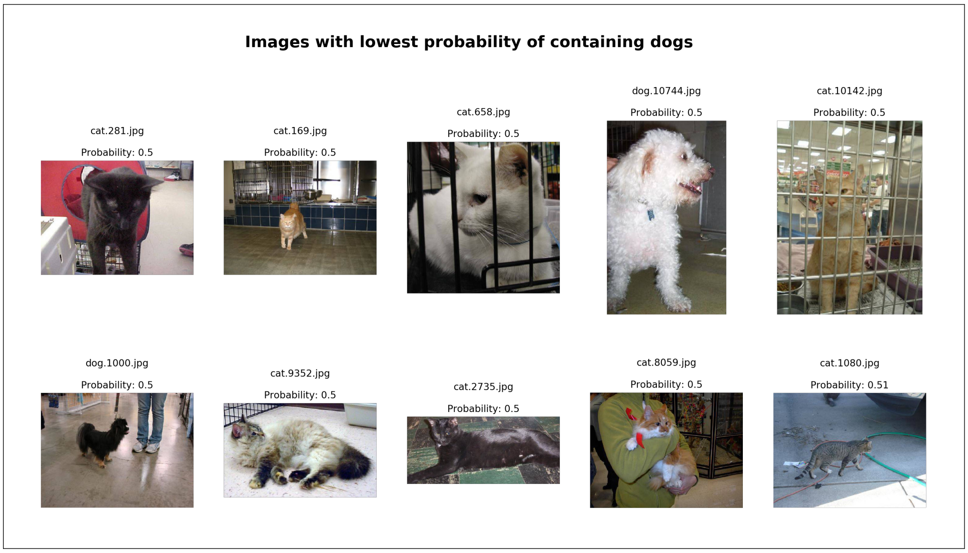


Figure 2-5. Images with the lowest probability of containing dogs.

To repeat, these are the images our classifier is most unsure of containing a dog. Most of these predictions are right at the tipping point (i.e. 0.5 probability) to be the majority prediction. Keep in mind the probability of being a cat is just slightly smaller, around 0.49. Compared to the previous set of images, these images are often smaller and less clear images. And often contain mispredictions - only 2 of the 10 images were the correctly predicted. One possible way to do better here is training with a larger set of images.

If you are concerned about these misclassifications, worry not. A simple trick to improve the classification accuracy is to have a higher threshold for accepting a classifier's results, say 0.75. If the classifier is unsure of an image category, its results are withheld. In chapter 4, we will look at how to find an optimal threshold.

Talking of mispredictions, they are obviously expected when the classifier has low confidence (i.e. near 0.5 probability for a 2 class problem). But what we don't want is to mispredict when our classifier is really sure of its predictions. Let's check which images the classifier is confident contain dogs in spite of them being cats.

```
# Incorrect predictions of 'dog'
indices = get_images_with_sorted_probabilities(prediction_table, True, 1, 10, True)
message = 'Images of cats with the highest probability of containing dogs'
display(indices[:10], message)
```

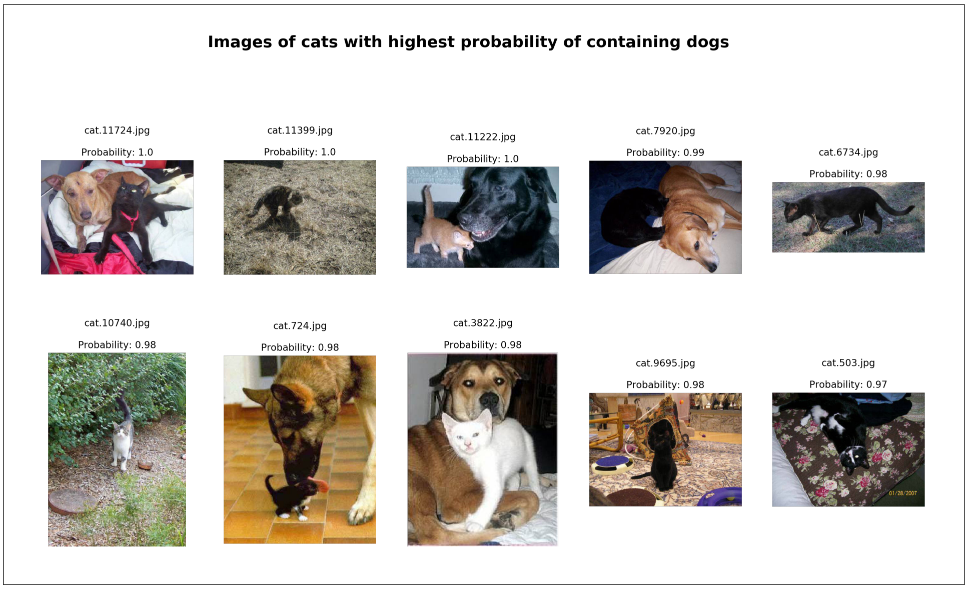


Figure 2-6. Images of cats with the highest probability of containing dogs.

Hmm… turns out half of these images contain both cats and dogs, and our classifier is correctly predicting the dog category as they are bigger in size in these images. Thus it's not the classifier but the data which is incorrect here. This often happens in large datasets. The remaining half often contain unclear and relatively smaller objects (but ideally we want to expect lower confidence for these hard to identify images).

Repeating the same set of question for the cat class, which images are more cat-like?

```
# Most confident predictions of 'cat'
indices = get_images_with_sorted_probabilities(prediction_table, True, 0, 10, False)
message = 'Images with the highest probability of containing cats'
display(indices[:10], message)
```

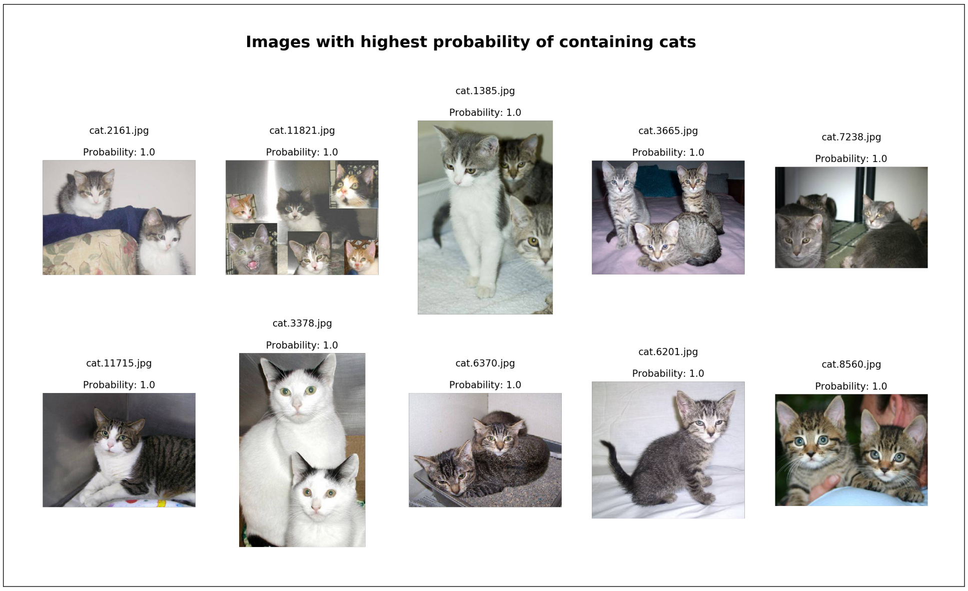


Figure 2-7. Images with the highest probability of containing cats.

Interestingly, many of these have multiple cats. This affirms our previous hypothesis that multiple clear, unambiguous views of cats can give higher probabilities. On the other hand, which images are we most unsure about containing cats?

```
# Least confident predictions of 'cat'
indices = get_images_with_sorted_probabilities(prediction_table, False, 0, 10, False)
message = 'Images with the lowest probability of containing cats'
display(indices[:10], message)
```

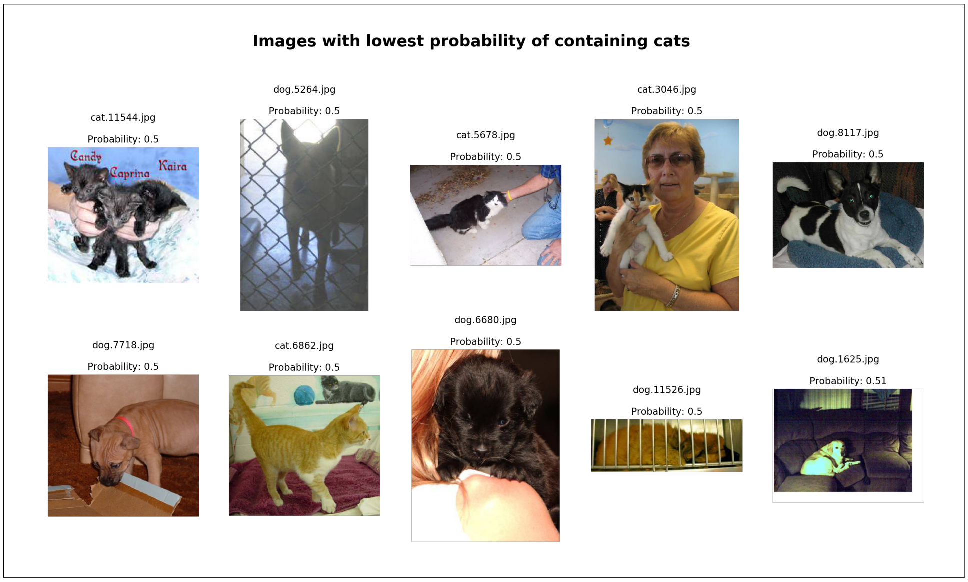


Figure 2-8. Images with the lowest probability of containing cats.

Like previously seen, the key object size is small, and some of the images are quite unclear meaning there is too much contrast in some cases or the object is too bright, something not in line with most of the training images. For example, the camera flash in the eighth (dog.6680) and tenth (dog.1625) images makes it hard to recognize. The sixth image contains a dog in front of a sofa of the same color. Two images contain cages.

Lastly, which images is our classifier mistakenly sure of containing cats?

```
# Incorrect predictions of 'cat'
indices = get_images_with_sorted_probabilities(prediction_table, True, 0, 10, True)
message = 'Images of dogs with the highest probability of containing cats'
display(indices[:10], message)
```

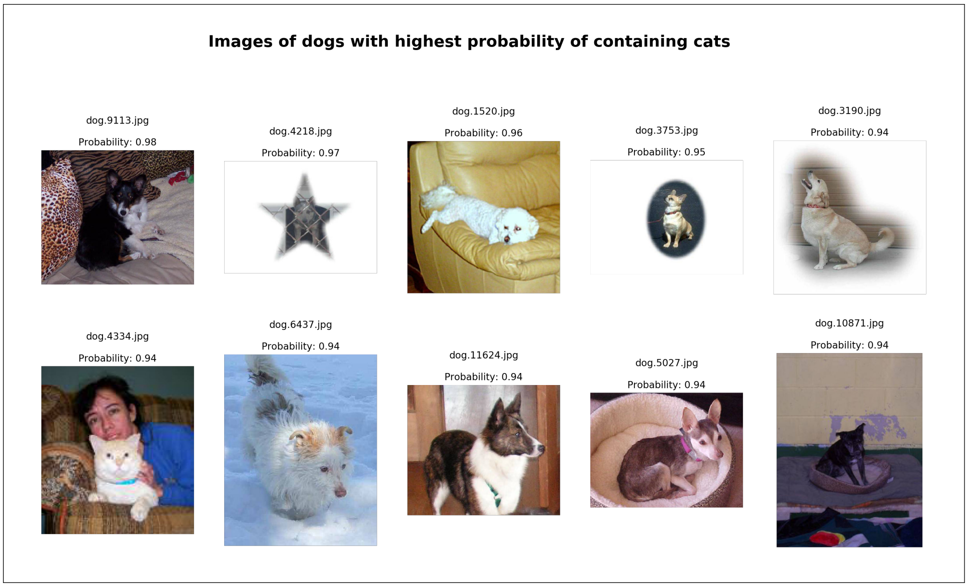


Figure 2-9. Images of dogs with the highest probability of containing cats.

These mispredictions are what we want to reduce. Some of them are clearly wrong, while others are understandably confusing images. The sixth image (dog.4334) seems to be incorrectly labeled as a dog. Seventh and tenth images are difficult to distinguish against the background. The first and tenth lack enough texture within them to give the classifier enough identification power. And some of the dogs are too small, like the second and fourth.

Going over the various analyses, we can summarize that mispredictions can be caused by low illumination, unclear, difficult to distinguish backgrounds, lack of texture and smaller occupied area with regard to the image.

Analyzing your predictions is a great way to understand what your model has learned, what it's bad at, and highlights opportunities to enhance its predictive power. Increasing the size of the training examples and more robust augmentation will help in improving the classification. It's also important to note that showing your model real world images (images that look similar to the scenario where your app will be used) will help improve its accuracy drastically. In Chapter 4, we will make the classifier more robust.

Summary

In this chapter, we introduced the concept of transfer learning. We reused a pre-trained model to build our own cats vs dogs classifier in under 30 lines of code and with barely 500 images, reaching state-of-art accuracy in a few minutes. By writing this code, we also debunk the myth that you need millions of images and powerful GPUs to train your classifier (though they help).

Hopefully with these skills, you might be able to finally answer the age-old question, “Who let the dogs out?!”

In the next chapter, we will use this learning to understand CNNs in more depth and take the model accuracy to the next level.

About the Authors

Anirudh Koul is the Head of AI & Research at Aira, and was previously at Microsoft AI & Research where he founded Seeing AI - the defacto app used by the blind community worldwide. With features shipped to about a billion people, he brings over a decade of production-oriented applied research experience on petabyte-scale datasets. He has been transforming ideas to reality using AI for Augmented Reality, Robotics, Speech, Productivity as well as building tools for people with disabilities. His work, which the IEEE has called "life changing," has been honored by CES, FCC, Cannes Lions, American Council of the Blind, showcased at events by UN, White House, House of Lords, World Economic Forum, TEDx, on Netflix, National Geographic, and applauded by world leaders including Justin Trudeau and Theresa May.

Siddha Ganju, who Forbes featured in their 30 under 30 list, is a Self-Driving Architect at Nvidia. Previously at Deep Vision, she developed deep learning models for resource constraint edge devices. A graduate from Carnegie Mellon University, her prior work ranges from Visual Question Answering to Generative Adversarial Networks to gathering insights from CERN's petabyte-scale data and has been published at top-tier conferences including CVPR and NeurIPS. Serving as an AI domain expert, she has also been guiding teams at NASA as well as featured as a jury member in several international tech competitions.

Meher Kasam is a seasoned software developer with apps used by tens of millions of users every day. Currently at Square, and previously at Microsoft, he shipped features for a range of apps, from Square's Point of Sale to the Bing app. He was the mobile development lead for Microsoft's Seeing AI app, which has received widespread recognition and awards from Mobile World Congress, CES, FCC, American Council of the Blind to name a few. A hacker at heart with a flair for fast prototyping, he has won close to two dozen hackathons and converted them to features shipped in widely-used products. He also serves as a judge of international competitions including Global Mobile Awards, Edison Awards.