Lecture 05

CNNs
Data Augmentation,
Feature Extraction
Fine Tuning

cscie-89 Deep Learning, Spring 2020

Zoran B. Djordjević

Objectives and Reference

- In what follows we will examine some realistic convolutional networks and learn how to use them for processing and classification of small sets of images.
- These notes follow Chapter 5 of "Deep Learning in Python" by Francois Chollet

Training CNNs on Small Datasets

- Having to train an image-classification model using very little data is a common situation.
- A small dataset means a few hundred to a few tens of thousands of images.
- As a practical example, we'll focus on classifying images as dogs or cats, in a dataset containing 4,000 pictures of cats and dogs (2,000 cats, 2,000 dogs). We'll use 2,000 pictures for training—1,000 for validation, and 1,000 for testing.
- We start by training a small CNN on the 2,000 training samples, without any regularization. At that point, the main issue will be overfitting.
- We will introduce ***data augmentation*****,** a powerful technique for mitigating overfitting in computer vision. By using data augmentation, you'll improve the accuracy of our deep learning network.
- We will review two more essential techniques for applying deep learning to small datasets:
  - *feature extraction with a pre-trained network* (which will get you to an accuracy of 90% to 96%) and
  - *fine-tuning a pre-trained network* (this will get you to a final accuracy of 97%).
- These three strategies—training a small model from scratch, doing feature extraction using a pre-trained model, and fine-tuning a pre-trained model—are the toolbox for tackling the problem of performing image classification with small datasets.
- Training a CNN from scratch on a very small image dataset will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering.

Dogs and Cats Dataset from Kaggle

- Download the original dataset from `www.kaggle.com/c/dogs-vs-cats/data` (you need to create a Kaggle account, which is simple to set up.)






- This dataset contains 25,000 images of dogs and cats (12,500 from each class), size of 543 MB (compressed). After downloading and uncompressing the data, we create a new dataset containing three subsets: a training set with 1,000 samples of each class, a validation set with 500 samples of each class, and a test set with 500 samples of each class.

Creating Directories for Small Datasets

- Before proceeding, we need to install two Python packages: `pillow` (Python Image Library) and `h5py` package for storing hierarchical datasets. "`pip install`" will do
- These are actions to create appropriate directories and small sets of cat images:

```
import os, shutil
# The path to the directory where the original
# dataset was uncompressed
original_dataset_dir = 'D:/CLASSES/Code/codedl_05cnn/cats_and_dogs/train'

# The directory where we will
# store our smaller dataset
base_dir = 'D:/CLASSES/Code/codedl_05cnn/small'
os.mkdir(base_dir)

# Directories for our training,
# validation and test splits
train_dir = os.path.join(base_dir, 'train')
os.mkdir(train_dir)
validation_dir = os.path.join(base_dir, 'validation')
os.mkdir(validation_dir)
test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)

# Directory with our training cat pictures
train_cats_dir = os.path.join(train_dir, 'cats')
os.mkdir(train_cats_dir)
```

Copy small subset of cats to cat directories

```
# Copy first 1000 cat images to train_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 cat images to validation_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 cat images to test_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_cats_dir, fname)
    shutil.copyfile(src, dst)
```

- You can check whether things work by printing the number of images in each directory

```
print('total training cat images:', len(os.listdir(train_cats_dir)))
```

Building the Network

- We are dealing with bigger images and a more complex problem than when working with MNIST dataset. We will make our network larger. It will have one more Conv2D + MaxPooling2D stage.
- This augments the capacity of the network, and further reduces the size of the feature maps, so that they aren't overly large when we reach the Flatten layer.
- Here, since we start from inputs of size 150x150 (a somewhat arbitrary choice), we end up with feature maps of size 7x7 right before the Flatten layer.
- Note that the depth of the feature maps is progressively increasing in the network (from 32 to 128), while the size of the feature maps is decreasing (from 148x148 to 7x7). This is a pattern that you will see in almost all CNNs.
- Since we are attacking a binary classification problem, we are ending the network with a single unit (a Dense layer of size 1) and a sigmoid activation. This unit will encode the probability that the network is looking at one class or the other.

Network Model

```
import tensorflow as tf
from tensorflow import keras
from keras import layers
from keras import models
model = keras.models.Sequential()
model.add(keras.layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(64, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(128, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(128, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(512, activation='relu'))
model.add(keras.layers.Dense(1, activation='sigmoid'))

# Compilation step
from keras import optimizers
model.compile(loss='binary_crossentropy',
optimizer=keras.optimizers.RMSprop(lr=1e-4),
metrics=['acc'])
```

Dimensions of features maps

```
model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_3 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

The Need for Data Preprocessing

- Data should be formatted into appropriate floating-point tensors before being fed into the network.
- Original data are JPEG files. The steps for transforming images are roughly as follows:
  1. Read the picture files.
  2. Decode the JPEG content to RGB grids of pixels.
  3. Convert these into floating-point tensors.
  4. Rescale the pixel values (between 0 and 255) to the [0, 1] interval (neural networks prefer to deal with small input values).
- Keras has utilities to take care of these steps automatically. Keras has a module with image-processing helper tools, contained in package: `keras.preprocessing.image`.
- In particular, this package contains the class `ImageDataGenerator`, which lets you quickly set up Python generators that automatically turn image files on disk into batches of preprocessed tensors.

tf.keras.preprocessing.image.ImageDataGenerator

- Generate batches of tensor image data including real-time data augmentation.

```
tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center=False, samplewise_center=False,
    featurewise_std_normalization=False,samplewise_std_normalization=False,
    zca_whitening=False, zca_epsilon=1e-06, rotation_range=0,
        width_shift_range=0.0, height_shift_range=0.0,
brightness_range=None,  shear_range=0.0, zoom_range=0.0,
    channel_shift_range=0.0, fill_mode='nearest', cval=0.0,
horizontal_flip=False,
    vertical_flip=False, rescale=None, preprocessing_function=None,
    data_format=None, validation_split=0.0, dtype=None
)
```

Data Preprocessing

```
from keras.preprocessing.image import ImageDataGenerator
# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')  # we need binary labels (dog vs. cat)
```

- This code yields batches of 150x150 RGB images (shape (20, 150, 150, 3)) and binary labels (shape (20,)).

Limit Batch Creation, `fit_generator()`

- 20 is the number of samples in each batch (the batch size).

```
for data_batch, labels_batch in train_generator:
    print('data batch shape:', data_batch.shape)
    print('labels batch shape:', labels_batch.shape)
    break

data batch shape: (20, 150, 150, 3)
labels batch shape: (20,).
```

- `fit_generator()` method expects as its first argument a Python generator that will yield batches of inputs and targets indefinitely, like this one does.
- Because the data is being generated endlessly, the Keras model needs to know how many samples to draw from the generator before declaring an epoch over. This is the role of the `steps_per_epoch` argument: after having drawn `steps_per_epoch` batches from the generator—that is, after having run for `steps_per_epoch` gradient descent steps—the fitting process will go to the next epoch. In this case, batches are 20 samples, so it will take 100 batches until you see your target of 2,000 samples.

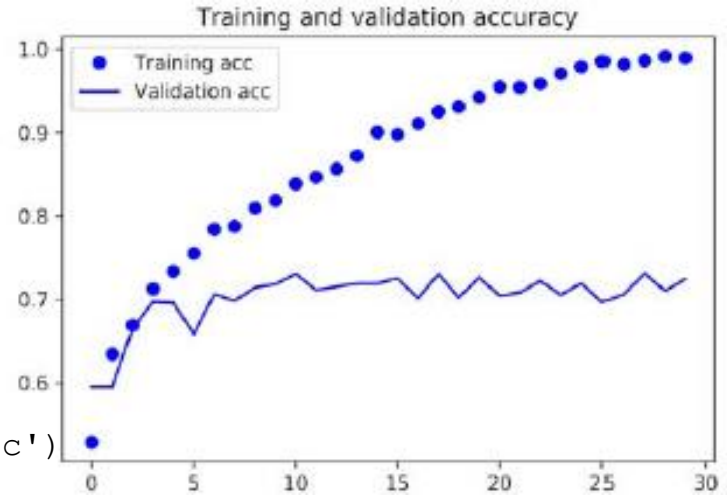
Limit Batch Creation, `fit_generator()`

- When using `fit_generator`, you can pass a `validation_data` argument, much as with the fit method. It's important to note that this argument is allowed to be a data generator, but it could also be a tuple of Numpy arrays. If you pass a generator as `validation_data`, then this generator is expected to yield batches of validation data endlessly; thus you should also specify the `validation_steps` argument, which tells the process how many batches to draw from the validation generator for evaluation.

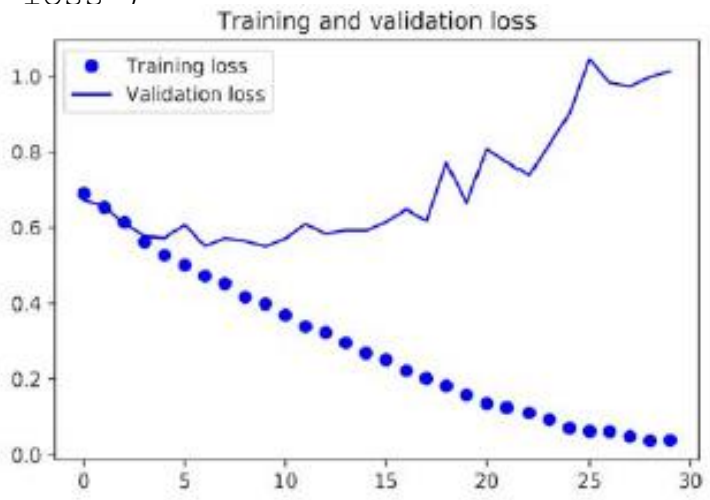
```
history = model.fit_generator(
        train_generator,
        steps_per_epoch=100,
        epochs=30,
        validation_data=validation_generator,
        validation_steps=50)
```

Accuracy over training and validation data

```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')
```




```
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```



- These plots are characteristic of overfitting.
- We have relatively few training samples (2,000)
- Overfitting is the number-one concern.
- You should always save your models

```
model.save('cats_and_dogs_small_1.h5')
```

Data Augmentation

Data Augmentation

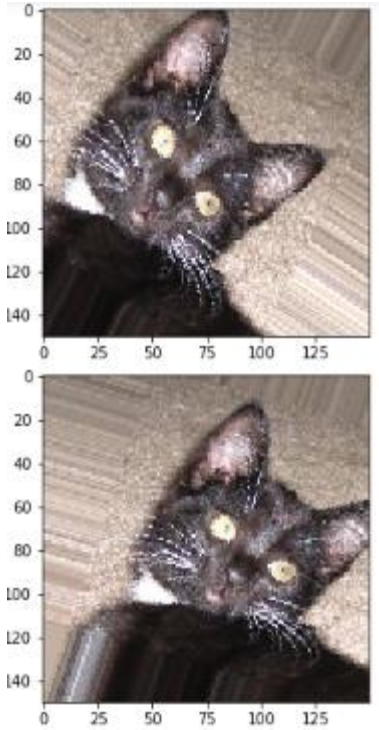
- Data augmentation takes the approach of generating more training data from existing training samples, by *augmenting* the samples via a number of random transformations that yield believable-looking images.
- The goal is that at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data and generalize better.
- In Keras, this can be done by configuring a number of random transformations to be performed on the images read by the ImageDataGenerator instance.

```
datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')
```

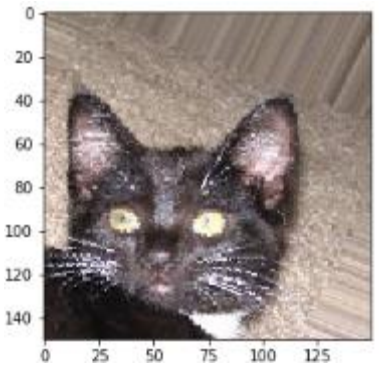
Options of `ImageDataGenerator`

- These are a few of the options available (see the Keras documentation).
  - `rotation_range` is a value in degrees (0–180), a range within which to randomly rotate pictures.
  - `witdth_shif` and `height_shift` are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
  - `shear_range` is for randomly applying shearing transformations.
  - `zoom_range` is for randomly zooming inside pictures.
  - `horizontal_flip` is for randomly flipping half the images horizontally—relevant when there are no assumptions of horizontal asymmetry (for example, real-world pictures).
  - `fill_mode` is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

Displaying Augmented Images



```
# This is module with image preprocessing utilities
from keras.preprocessing import image
fnames =[os.path.join(train_cats_dir,fname)
     for fname in os.listdir(train_cats_dir)]
# We pick one image to "augment"
img_path = fnames[5]
# Read the image and resize it
img = image.load_img(img_path, target_size=(150, 150))
# Convert it to a Numpy array with shape (150, 150, 3)
x = image.img_to_array(img)
# Reshape it to (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)
# The .flow() command below generates batches of randomly transformed images.
# It will loop indefinitely, so we need to `break` the loop at some point!
i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```



Network for Augmented Images

- If you train a new network using this data-augmentation configuration, the network will never see the same input twice. But the inputs it sees are still heavily inter-correlated, because they come from a small number of original images—you can't produce new information, you can only remix existing information.
- As such, image augmentation may not be enough to completely get rid of overfitting. To further fight overfitting, you'll also add a Dropout layer to your model, right before the densely connected classifier. New network model is now:

```
model = keras.models.Sequential()
model.add(keras.layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(150, 150, 3)))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(64, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(128, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(128, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(512, activation='relu'))
model.add(keras.layers.Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
        optimizer=keras.optimizers.RMSprop(lr=1e-4),
        metrics=['acc'])
```

Training the Network

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,)
# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
   train_dir, # This is the target directory
   target_size=(150, 150), # All images will be resized to 150x150
   batch_size=16,
   # Since we use binary_crossentropy loss, we need binary labels
   class_mode='binary')
validation_generator = test_datagen.flow_from_directory(
   validation_dir,
   target_size=(150, 150),
   batch_size=16,
   class_mode='binary')
history = model.fit( # _generator(
  train_generator,      steps_per_epoch=100,
  epochs=100,   validation_data=validation_generator,  validation_steps=50)
model.save('cats_and_dogs_small_2.h5')
```

Training Start to become long

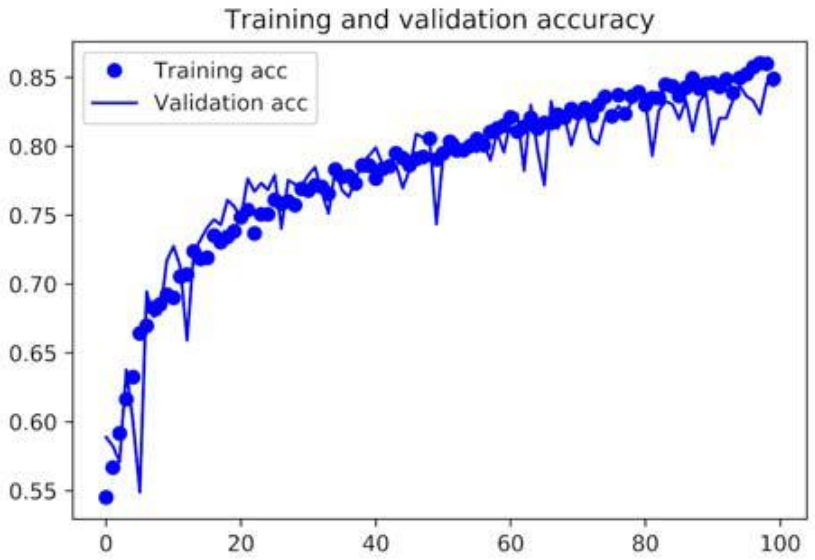
- On my Windows desktop each epoch took ~12 seconds.

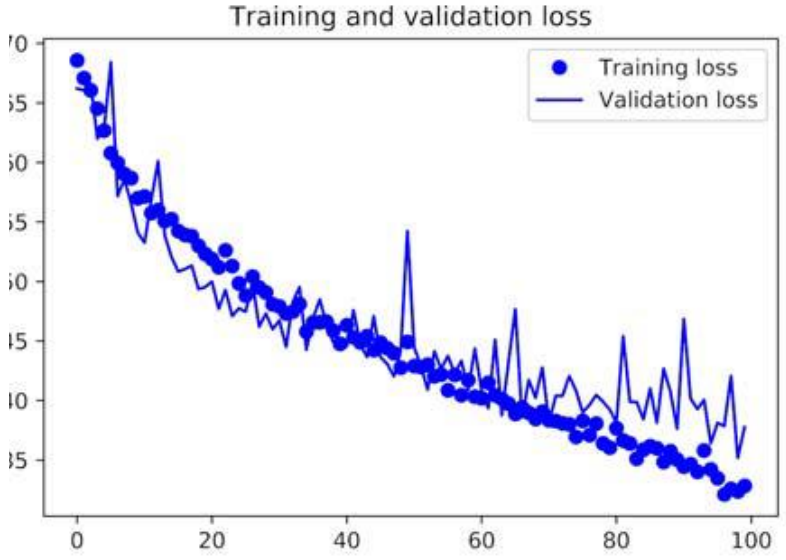
```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Epoch 1/15 100/100 [==] - 373s 4s/step - loss: 0.6903 - acc: 0.5322 -
val_loss: 0.6729 - val_acc: 0.6003
Epoch 2/15 100/100 [==] - 361s 4s/step - loss: 0.6763 - acc: 0.5597 -
val_loss: 0.6647 - val_acc: 0.5973
. . . .
Epoch 14/15 100/100 [==] - 347s 3s/step - loss: 0.5510 - acc: 0.7141 -
val_loss: 0.5202 - val_acc: 0.7468
Epoch 15/15 100/100 [==] - 348s 3s/step - loss: 0.5364 - acc: 0.7309 -
val_loss: 0.5494 - val_acc: 0.7191
```

You need to go to 100 epochs to see high accuracy.

Training and Validation Accuracy

- Thanks to data augmentation and dropout, you're no longer overfitting: the training curves are closely tracking the validation curves. You now reach an accuracy of 82%, a 15% relative improvement over the non-regularized model.





- Some further increase in accuracy could be obtained if we would add regularization procedure. We will examine another technique, called pre-trained networks.

Transfer Learning or Feature Extraction

Pre-trained CNNs

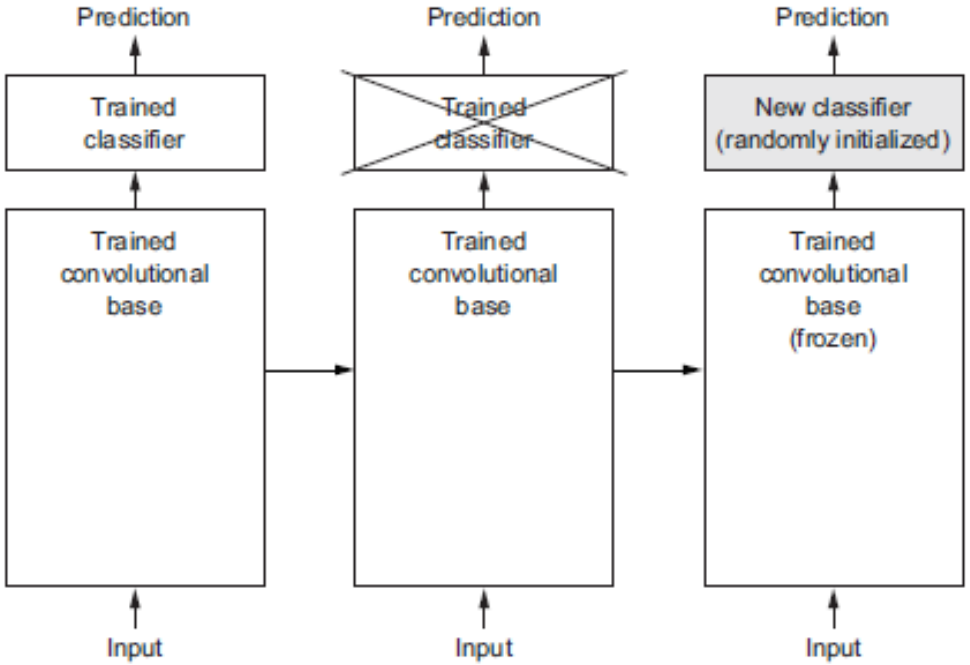
- A common and highly effective approach to deep learning on small image datasets is to use pre-trained networks.
- A *pre-trained network* is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task.
- If this original dataset is large enough and general enough, then the spatial hierarchy of features learned by the pre-trained network can effectively act as a generic model of the visual world, and hence its features can prove useful for many different computer vision problems, even though these new problems may involve completely different classes than those of the original task.
- For instance, you might train a network on ImageNet (where classes are mostly animals and everyday objects) and then repurpose this trained network for something as remote as identifying furniture items in images.
- Such portability of learned features across different problems is a key advantage of deep learning compared to many older, shallow-learning approaches, and it makes deep learning very effective for small-data problems.

VGG16

- We will consider a large CNN trained on the ImageNet dataset (1.4 million labeled images and 1,000 different classes). ImageNet contains many animal classes, including different species of cats and dogs, and you can thus expect to perform well on the dogs-versus-cats classification problem.
- We will use the VGG16 architecture, developed by Karen Simonyan and Andrew Zisserman in 2014; it's a simple and widely used CNN architecture for ImageNet.1
- This is an older model, but its architecture is like what we were using and is easy to understand without introducing any new concepts.
- There are many other model available: VGG, ResNet, Inception, Inception-ResNet, Xception, and so on. They will encounter them if you keep doing deep learning for computer vision.
- There are two methods to use a pre-trained network:
  - *feature extraction* and
  - *fine-tuning*.
- We will cover both methods.

Feature Extraction or Transfer Learning

- Feature extraction (Transfer Learning) consists of using the representations learned by a previous network to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.
- CNNs used for image classification comprise two parts: a series of pooling and convolution layers, and a densely connected classifier. The first part is called the *convolutional base* of the model.
- In the case of CNNs, feature extraction consists of taking the convolutional base of a previously trained network, running the new data through it, and training a new classifier on top of the output.



Separation of Insights

- Could you reuse the densely connected classifier as well? In general, doing so should be avoided. The representations learned by the convolutional base are likely to be more generic and therefore more reusable: the feature maps of a CNN are presence maps of generic concepts in analyzed pictures, which is likely to be useful regardless of the computer-vision problem at hand.
- The representations learned by the classifier will necessarily be specific to the set of classes on which the model was trained—they will only contain information about the presence probability of this or that class in the entire picture.
- Representations found in densely connected layers no longer contain any information about *where* objects are located in the input image: these layers get rid of the notion of space, whereas the object location is still described by convolutional feature maps.
- Where object location matters, densely connected features are largely useless.
- The level of generality (and therefore reusability) of the representations extracted by specific convolution layers depends on the depth of the layer in the model. Layers that come earlier in the model extract local, highly generic feature maps (such as visual edges, colors, and textures), whereas layers that are higher up extract more-abstract concepts (such as "cat ear" or "dog eye").
- If your new dataset differs a lot from the dataset on which the original model was trained, you may be better off using only the first few layers of the model to do feature extraction, rather than using the entire convolutional base.

Models Pre-packaged with Keras

- ImageNet class set contains multiple dog and cat classes, and it's likely to be beneficial to reuse the information contained in the densely connected layers of the original model.
- We will choose not to do that, in order to cover the more general case where the class set of the new problem doesn't overlap the class set of the original model.
- We will use the convolutional base of the VGG16 network, trained on ImageNet, to extract interesting features from cat and dog images, and then train a dogs-versus-cats classifier on top of these features.
- The VGG16 model, among others, comes prepackaged with Keras. You can import it from the `keras.applications` module.
- The many image-classification models, pre-trained on the ImageNet dataset, are available as part of `keras.applications` package:
  - Xception
  - Inception V3
  - ResNet50
  - VGG16
  - VGG19
  - MobileNet

- To see the full list go to: https://github.com/keras-team/keras-applications

Instantiate VGG16 Model

```
from keras.applications import VGG16
conv_base = VGG16(weights='imagenet',
    include_top=False,
    input_shape=(150, 150, 3))
```

- Arguments of the constructor:
  - `weights` specifies the weight checkpoint from which to initialize the model.
  - `include_top` refers to including (or not) the densely connected classifier on top of the network. By default, this densely connected classifier corresponds to the 1,000 classes from ImageNet. Because you intend to use your own densely connected classifier (with only two classes: cat and dog), you don't need to include it.
  - `input_shape` is the shape of the image tensors that you plan to feed to the network. This argument is optional: if you don't pass it, the network will be able to process inputs of any size.

- VGG16 convolutional base is similar to the simple CNNs we are familiar with:

```
conv_base.summary()
```

Architecture of VGG16 convolutional base

```
Layer (type) Output Shape Param #
=================================================================
input_1 (InputLayer) (None, 150, 150, 3)
block1_conv1 (Convolution2D) (None, 150, 150, 64) 1792
_________________________________________________________________
block1_conv2 (Convolution2D) (None, 150, 150, 64) 36928
_________________________________________________________________
block1_pool (MaxPooling2D) (None, 75, 75, 64) 0
_________________________________________________________________
block2_conv1 (Convolution2D) (None, 75, 75, 128) 73856
_________________________________________________________________
block2_conv2 (Convolution2D) (None, 75, 75, 128) 147584
_________________________________________________________________
block2_pool (MaxPooling2D) (None, 37, 37, 128) 0
_________________________________________________________________
block3_conv1 (Convolution2D) (None, 37, 37, 256) 295168
_________________________________________________________________
block3_conv2 (Convolution2D) (None, 37, 37, 256) 590080
_________________________________________________________________
block3_conv3 (Convolution2D) (None, 37, 37, 256) 590080
_________________________________________________________________
block3_pool (MaxPooling2D) (None, 18, 18, 256) 0
_________________________________________________________________
block4_conv1 (Convolution2D) (None, 18, 18, 512) 1180160
_________________________________________________________________
block4_conv2 (Convolution2D) (None, 18, 18, 512) 2359808
_________________________________________________________________
block4_conv3 (Convolution2D) (None, 18, 18, 512) 2359808
_________________________________________________________________
block4_pool (MaxPooling2D) (None, 9, 9, 512) 0
_________________________________________________________________
block5_conv1 (Convolution2D) (None, 9, 9, 512) 2359808
_________________________________________________________________
block5_conv2 (Convolution2D) (None, 9, 9, 512) 2359808
_________________________________________________________________
block5_conv3 (Convolution2D) (None, 9, 9, 512) 2359808
_________________________________________________________________
block5_pool (MaxPooling2D) (None, 4, 4, 512) 0
=================================================================
```

```
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

- The final feature map has shape (4, 4, 512). That's the feature on top of which you'll
- stick a densely connected classifier.

How to use imported convolutional base

At this point, there are two ways you could proceed:

1. Running the convolutional base over your dataset, recording its output to a Numpy array on disk, and then using this data as input to a standalone, densely connected classifier similar to those you saw in part 1 of this book.

   This solution is fast and cheap to run, because it only requires running the convolutional base once for every input image, and the convolutional base is by far the most expensive part of the pipeline. But for the same reason, this technique won't allow you to use data augmentation.

- We will refer to this approach as "Fast feature extraction without data augmentation".

2. Extending the model you have (conv_base) by adding Dense layers on top, and running the whole thing end to end on the input data. This will allow you to use data augmentation, because every input image goes through the convolutional base every time it's seen by the model. For the same reason, this technique is far more expensive than the first.

Fast feature extraction without data augmentation

We start by running instances of `ImageDataGenerator` to extract images as Numpy arrays as well as their labels.

We extract features from these images by calling the `predict` method of the conv_base model.

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
base_dir = 'E:/CLASSES/Code/dl/codedl06/small'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')
datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20
def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
        features_batch = conv_base.predict(inputs_batch)
        features[i * batch_size : (i + 1) * batch_size] = features_batch
        labels[i * batch_size : (i + 1) * batch_size] = labels_batch
        i += 1
        if i * batch_size >= sample_count:
            # we must `break` after every image has been seen once.
            break
    return features, labels
train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels =
extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)
```

Reshape Data

- The extracted features are of shape (samples, 4, 4, 512). To feed them to a densely connected classifier, we must flatten them to (samples, 8192):

```
train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
test_features = np.reshape(test_features, (1000, 4 * 4 * 512))
```

- At this point, you can define your densely connected classifier (note the use of dropout for regularization) and train it on the data and labels that you just recorded.
- Defining and training the densely connected classifier:

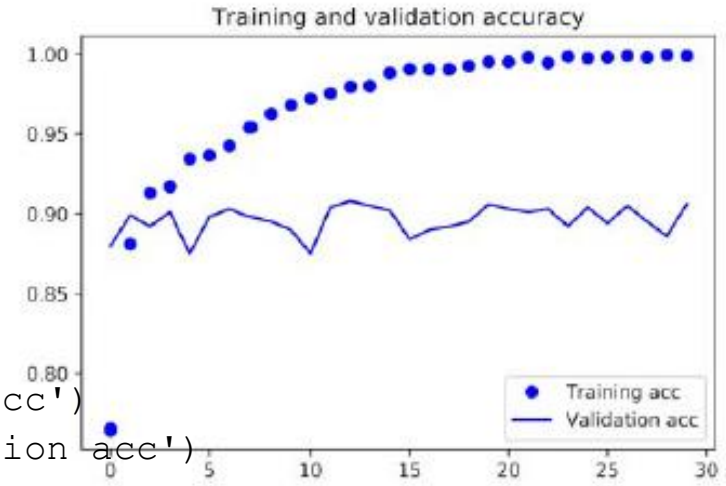
```
from keras import models
from keras import layers
from keras import optimizers
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer=optimizers.RMSprop(lr=2e-5),
loss='binary_crossentropy',
metrics=['acc'])
history = model.fit(train_features, train_labels,
epochs=30,
batch_size=20,
validation_data=(validation_features, validation_labels))
```

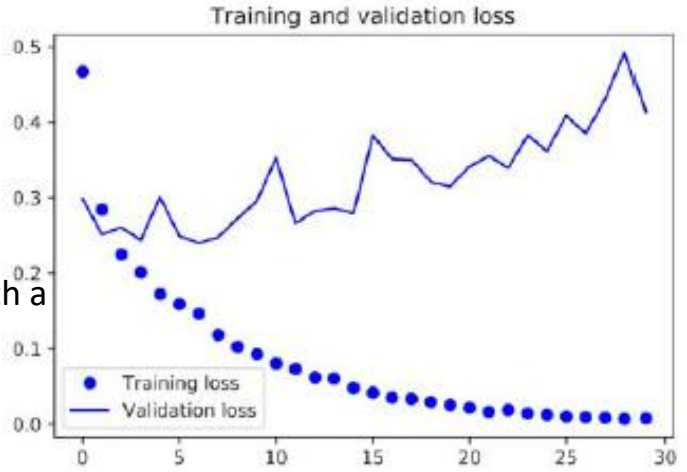
- Training is very fast, because you only have to deal with two Dense layers—an epoch takes less than one second even on CPU.

Loss and Accuracy, Simple Feature Extraction

- To plot the results we do:

```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation
loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```





- Validation accuracy is about 90%—better than you achieved in the previous section with the small model trained from scratch.
- We are overfitting almost from the start—despite using dropout with a fairly large rate. That's because this technique doesn't use data augmentation,

Feature Extraction with Data Augmentation

- The second technique for feature extraction is much slower and more expensive, but which allows you to use data augmentation during training.
- We will extend the conv_base model and run it end to end on the inputs.
- This technique is so expensive that you should only attempt it if you have access to a GPU—it's absolutely intractable on CPU.
- Models behave just like layers, so we can add a model (like conv_base) to a Sequential model just like we would add a layer.
- We are adding a densely connected classifier on top of the convolutional base

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Combined Model

```
>>> model.summary()
Layer (type) Output Shape Param #
=================================================================
vgg16 (Model) (None, 4, 4, 512) 14714688
_________________________________________________________________
flatten_1 (Flatten) (None, 8192) 0
_________________________________________________________________
dense_1 (Dense) (None, 256) 2097408
_________________________________________________________________
dense_2 (Dense) (None, 1) 257
=================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

- The convolutional base of VGG16 has 14,714,688 parameters, which is very large. The classifier you're adding on top has 2 million parameters.
- Before you compile and train the model, it's very important to freeze the convolutional base.
- *Freezing* a layer or set of layers means preventing their weights from being updated during training. If you don't do this, then the representations that were previously learned by the convolutional base will be modified during training. Because the Dense layers on top are randomly initialized, very large weight updates would be propagated through the network, effectively destroying the representations previously learned.

Freezing a layer

- In Keras, you freeze a network by setting its `trainable` attribute to `False`.
- You can examine the number of trainable weights by:

```
>>> print('This is the number of trainable weights '
        'before freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights before freezing the conv base: 30
>>> conv_base.trainable = False
>>> print('This is the number of trainable weights '
        'after freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights after freezing the conv base: 4
```

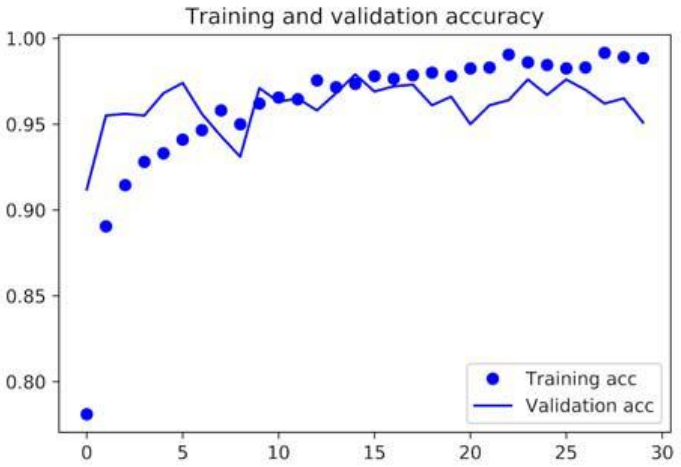
- With this setup, only the weights from the two Dense layers that you added will be trained. That's a total of four weight tensors: two per layer (the main weight matrix and the bias vector).
- Note that in order for these changes to take effect, you must first compile the model. If you ever modify weight trainability after compilation, you should then recompile the model, or these changes will be ignored.

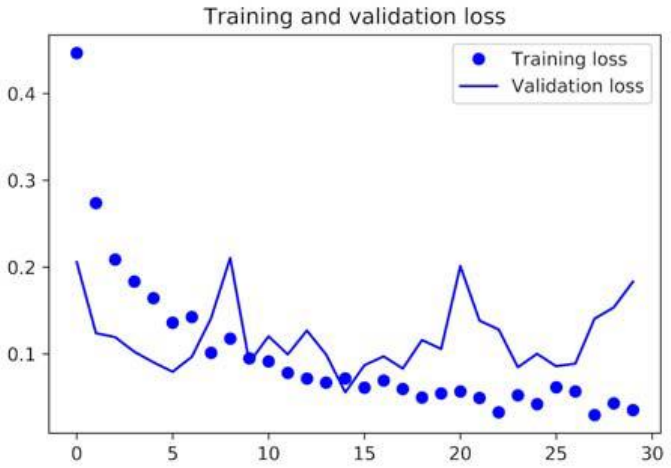
Training the model with a frozen convolutional base

```
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers
train_datagen = ImageDataGenerator(
      rescale=1./255, rotation_range=40,
      width_shift_range=0.2, height_shift_range=0.2,
      shear_range=0.2, zoom_range=0.2,
      horizontal_flip=True, fill_mode='nearest')
# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150), batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')
validation_generator = test_datagen.flow_from_directory(
        validation_dir, target_size=(150, 150),
        batch_size=20,  class_mode='binary')
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=2e-5), metrics=['acc'])
history = model.fit_generator(
      train_generator,  steps_per_epoch=100,  epochs=5,
      validation_data=validation_generator,
      validation_steps=50,      verbose=2)
```

Training and Validation Accuracy and Loss

- Training and validation accuracy for feature extraction with data augmentation

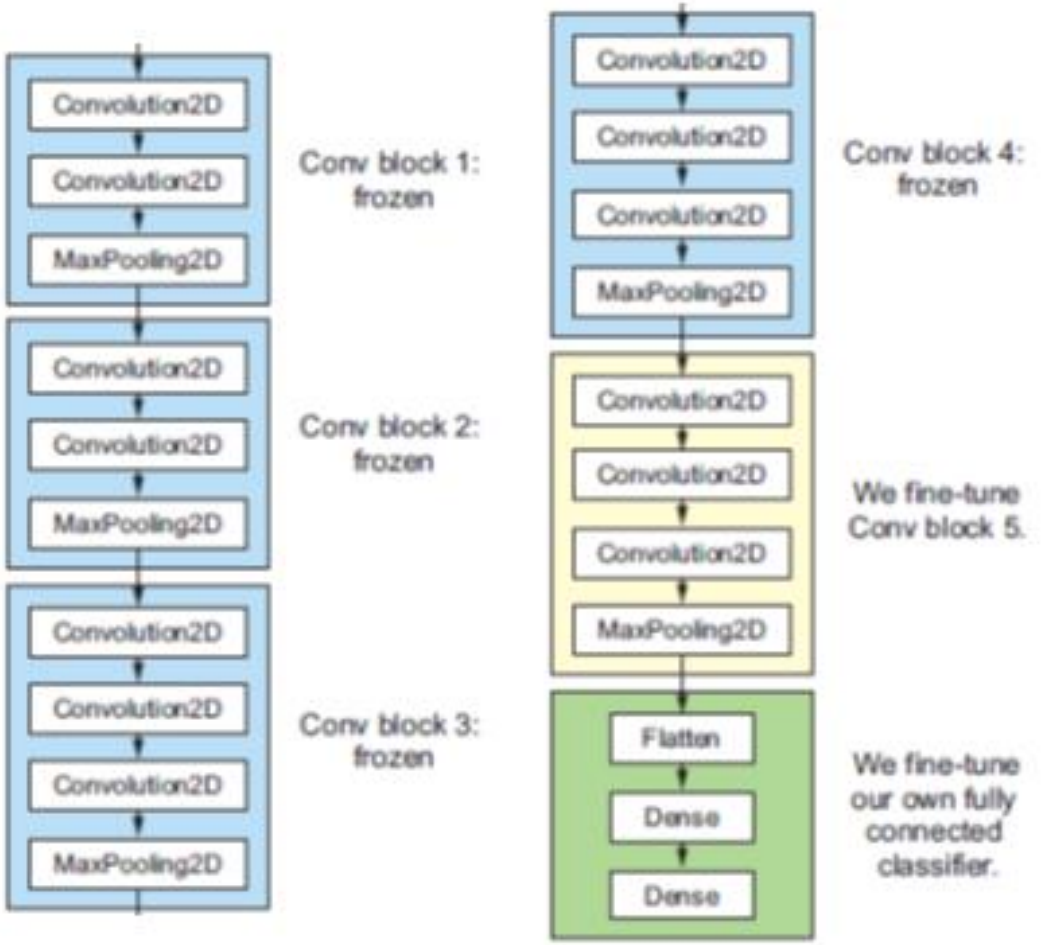




Fine Tuning

Fine Tuning

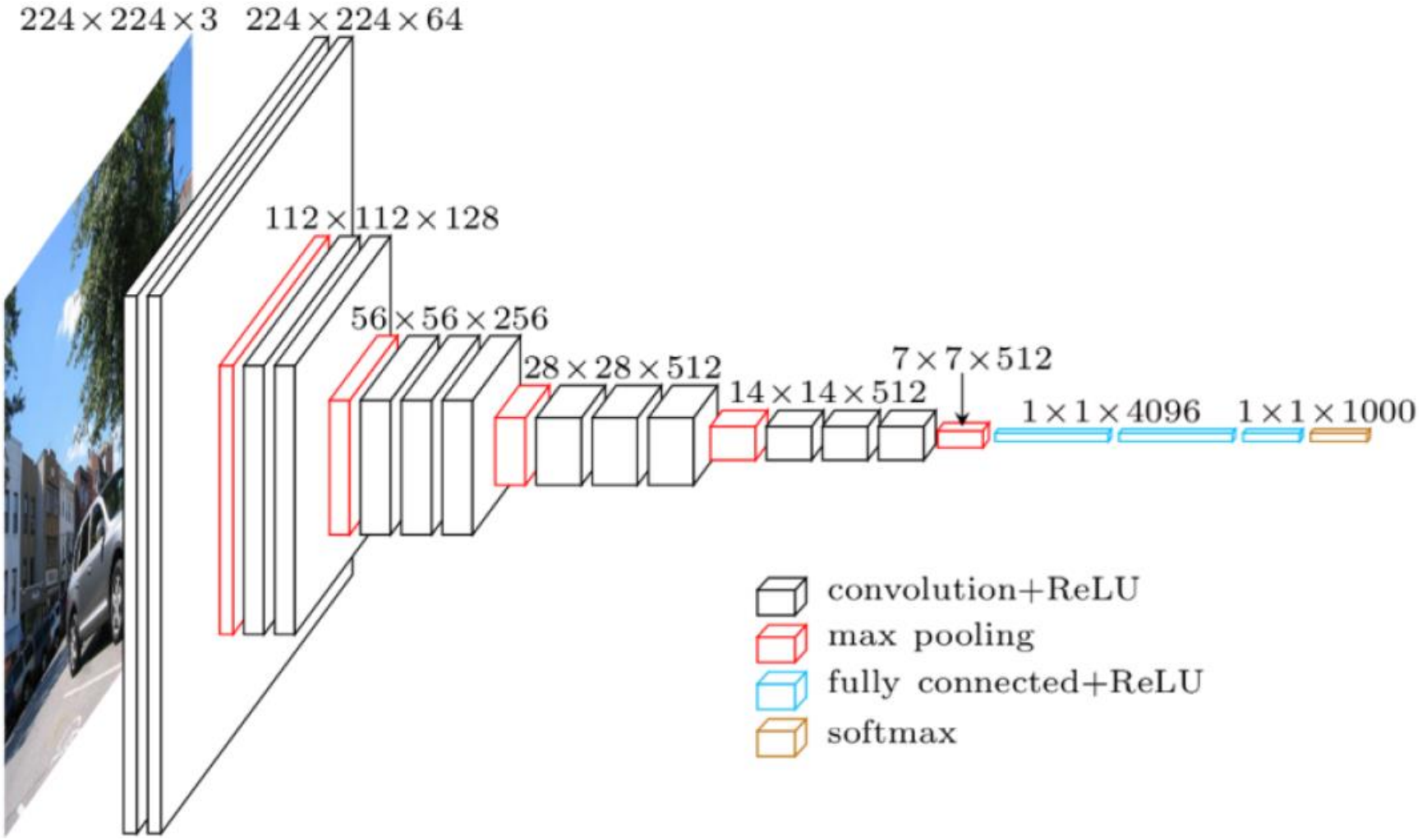
- Another widely used technique for model reuse, complementary to feature extraction, is *fine-tuning*.
- Fine-tuning consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model (in this case, the fully connected classifier) and these top layers.



- This is called *fine-tuning* because it slightly adjusts the more abstract representations of the model being reused, in order to make them more relevant for the problem at hand.
- Layers above belong to VGG16 network.

VGGNet Architecture

- The image below gives you more details about the architecture of VGG network.



Steps in Fine Tuning

- It is necessary to freeze the convolution base of VGG16 in order to be able to train a randomly initialized classifier on top. For the same reason, it's only possible to fine-tune the top layers of the convolutional base once the classifier on top has already been trained.
- If the classifier isn't already trained, then the error signal propagating through the network during training will be too large, and the representations previously learned by the layers being fine-tuned will be destroyed. Thus the steps for fine-tuning a network are as follow:
  1. Add your custom network on top of an already-trained base network.
  2. Freeze the base network.
  3. Train the part you added.
  4. Unfreeze some layers in the base network.
  5. Jointly train both these layers and the part you added.
- We already completed the first three steps when doing feature extraction. Let's proceed with step 4: we will unfreeze your conv_base and then freeze individual layers inside it.

Fine Tuning Last 3 Convolutional Layers

- You'll fine-tune the last three convolutional layers, which means all layers up to block4_pool should be frozen, and the layers block5_conv1, block5_conv2, and block5_conv3 should be trainable.
- Why not fine-tune more layers or the entire convolutional base?
- We could. But you need to consider the following:
  - Earlier layers in the convolutional base encode more-generic, reusable features, whereas layers higher up encode more-specialized features. It's more useful to fine-tune the more specialized features, because these are the ones that need to be repurposed on your new problem. There would be fast-decreasing returns in fine-tuning lower layers.
  - The more parameters you're training, the more you're at risk of overfitting. The convolutional base has 15 million parameters, so it would be risky to attempt to train it on your small dataset.
- In this situation, it's a good strategy to fine-tune only the top two or three layer in the convolutional base.

Freezing all layers up to a specific one

```
conv_base.trainable = True
conv_bas.set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

- Now we can start fine-tuning our network. We will do this with the RMSprop optimizer, using a very low learning rate. The reason for using a low learning rate is that we want to limit the magnitude of the modifications we make to the representations of the 3 layers that we are fine-tuning. Updates that are too large may harm these representations

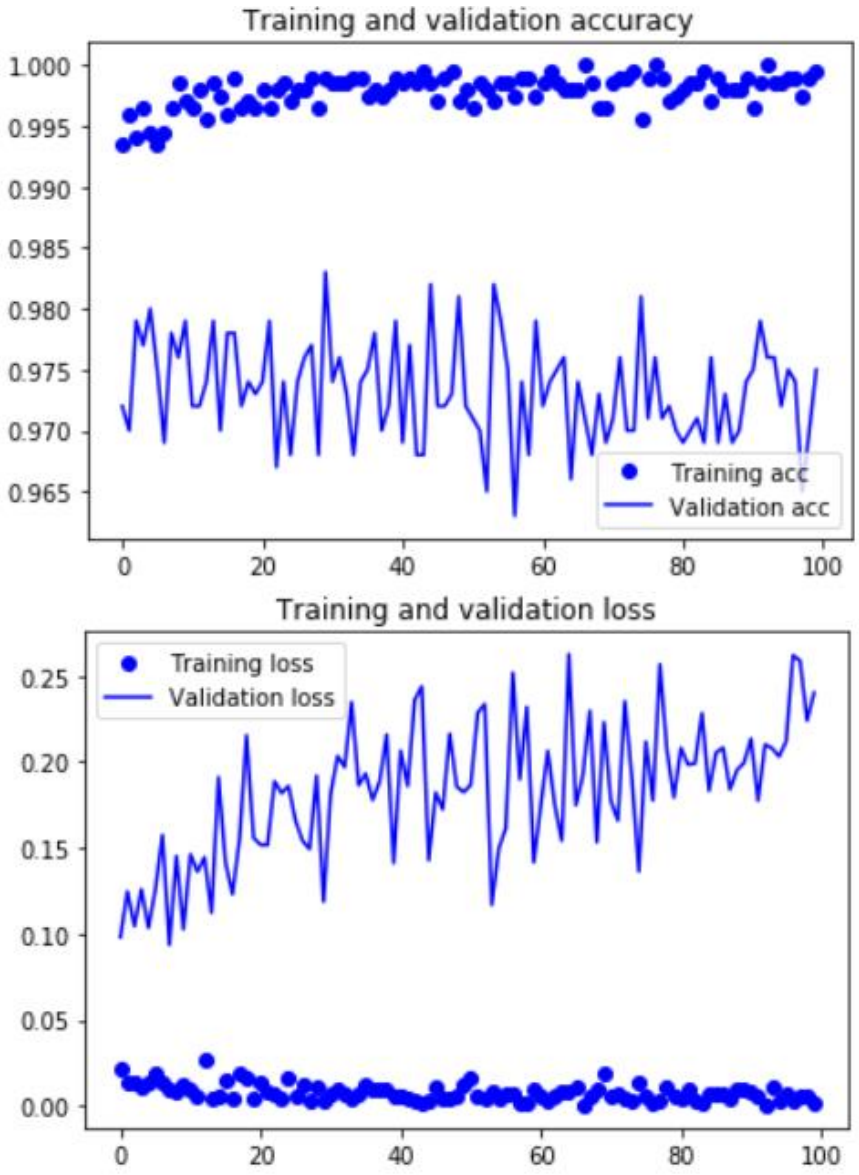
```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=50)

model.save('cats_and_dogs_small_4.h5')
```

Plot Training and Validation Accuracy

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

Training and Validation Accuracy

We are seeing a nice 1% absolute improvement.
Note that the loss curve does not show any real improvement (in fact, it is deteriorating).
You may wonder, how could accuracy improve if the loss isn't decreasing? The answer is simple: what we display is an average of pointwise loss values, but what actually matters for accuracy is the distribution of the loss values, not their average, since accuracy is the result of a binary thresholding of the class probability predicted by the model. The model may still be improving even if this isn't reflected in the average loss.



Freezing Keras Layers

- To "freeze" a layer means to exclude it from training, i.e. its weights will never be updated. This is useful in the context of fine-tuning a model, or using fixed embeddings for a text input.
- You can pass a trainable argument (boolean) to a layer constructor to set a layer to be non-trainable:

```
frozen_layer = Dense(32, trainable=False)
```

- Additionally, you can set the trainable property of a layer to True or False after instantiation. For this to take effect, you will need to call `compile()` on your model after modifying the trainable property. Here's an example:

```
x = Input(shape=(32,))
layer = Dense(32)
layer.trainable = False
y = layer(x)
•
frozen_model = Model(x, y)
# in the model below, the weights of `layer` will not be updated during training
frozen_model.compile(optimizer='rmsprop', loss='mse')

layer.trainable = True
trainable_model = Model(x, y)
# with this model the weights of the layer will be updated during training
# (which will also affect the above model since it uses the same layer instance)
trainable_model.compile(optimizer='rmsprop', loss='mse')

frozen_model.fit(data, labels)  # this does NOT update the weights of `layer`
trainable_model.fit(data, labels)  # this updates the weights of `layer`
```

Summary

- CNNs are the best type of machine learning models for computer vision tasks. It is possible to train one from scratch even on a very small dataset, with decent results.
- On a small dataset, overfitting will be the main issue. Data augmentation is a powerful way to fight overfitting when working with image data.
- It is easy to reuse an existing CNN on a new dataset, via feature extraction. This is a very valuable technique for working with small image datasets.
- As a complement to feature extraction, one may use fine-tuning, which adapts to a new problem some of the representations previously learned by an existing model. This pushes performance a bit further.