Lecture 06

Vision Models - Fine Tuning & Transfer Learning

cscie-89 Deep Learning, Fall 2024

Zoran B. Djordjević & Rahul Joglekar

Incase you missed this .....

Share this

English
English (pdf)
Swedish
Swedish (pdf)

KUNGL. VETENSKAPS-AKADEMIEN
THE ROYAL SWEDISH ACADEMY OF SCIENCES

8 October 2024

The Royal Swedish Academy of Sciences has decided to award the Nobel Prize in Physics 2024 to

John J. Hopfield
Princeton University, NJ, USA

Geoffrey E. Hinton
University of Toronto, Canada

"for foundational discoveries and inventions that enable machine learning with artificial neural networks"

They trained artificial neural networks using physics

Incase you missed this …..

See all chemistry laureates or learn about the nomination process.

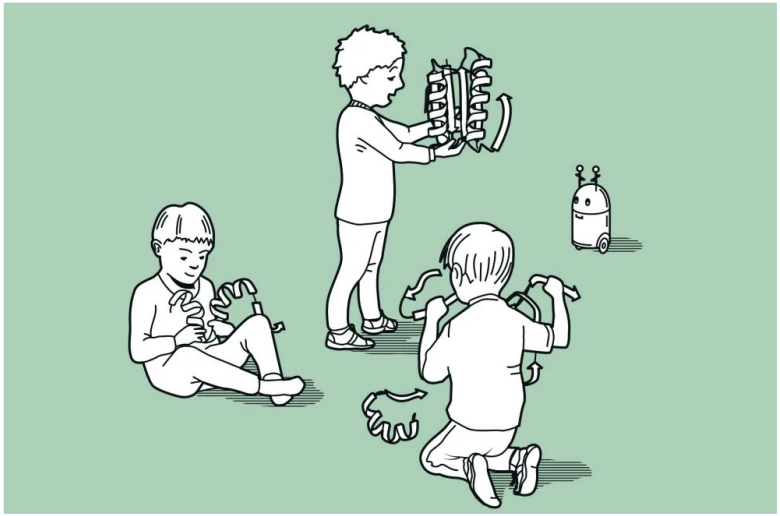
The Nobel Prize in Chemistry 2024

They cracked the code for proteins' amazing structures

The Nobel Prize in Chemistry 2024 is about proteins, life's ingenious chemical tools. David Baker has succeeded with the almost impossible feat of building entirely new kinds of proteins. Demis Hassabis and John Jumper have developed an AI model to solve a 50-year-old problem: predicting proteins' complex structures. These discoveries hold enormous potential.

Related articles

Press release
Popular information: They have revealed proteins' secrets through computing and artificial intelligence
Scientific background: Computational protein design and protein structure prediction



© Johan Jarnestad/The Royal Swedish Academy of Sciences

Objectives and Reference

- We will examine some realistic convolutional networks and learn how to use them for processing and classification of data
  - VGG16
  - RESNET
  - LeNet
- We will learn techniques known as:
  - Data Augmentation
  - Transfer Learning or Feature Extraction, and
  - Fine Tuning
- We will illustrate above techniques on CNNs and Vision problems. They can be generalized and in an appropriate form applied to all other types of networks.

These notes follow:

- Chapter 8 of "Deep Learning in Python" by Francois Chollet, 2nd Edition, Manning Publishing, 2021
- Chapter 5 of "Deep Learning in Python" by Francois Chollet, 1st Edition, Manning Publishing, 2017
- Chapter 11 of “Hands-on ML with Scikit-Learn, Keras … by Aurelien Geron, 2nd Edition, Manning Publishing, 2019

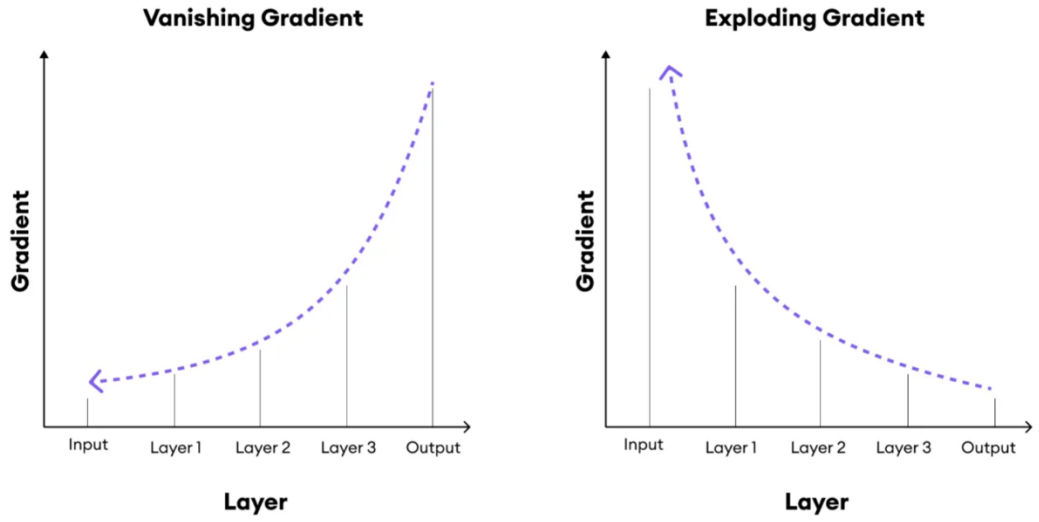
CNN Trivia

- How many Feature-maps in a layer
- Does a CNN have a Backprop ?
- Is the Backprop also a Convolution ?
- Why even bother adding a Pooling layer ?
- What's the difference between a Kernel and a Filter ?
- How do you define the shape of a filter ? Say a Whisker of a cat or tail of a horse or leg of a cow etc etc
- What is a Filter and how do you tell what filter to use and number of filters to a CNN ?

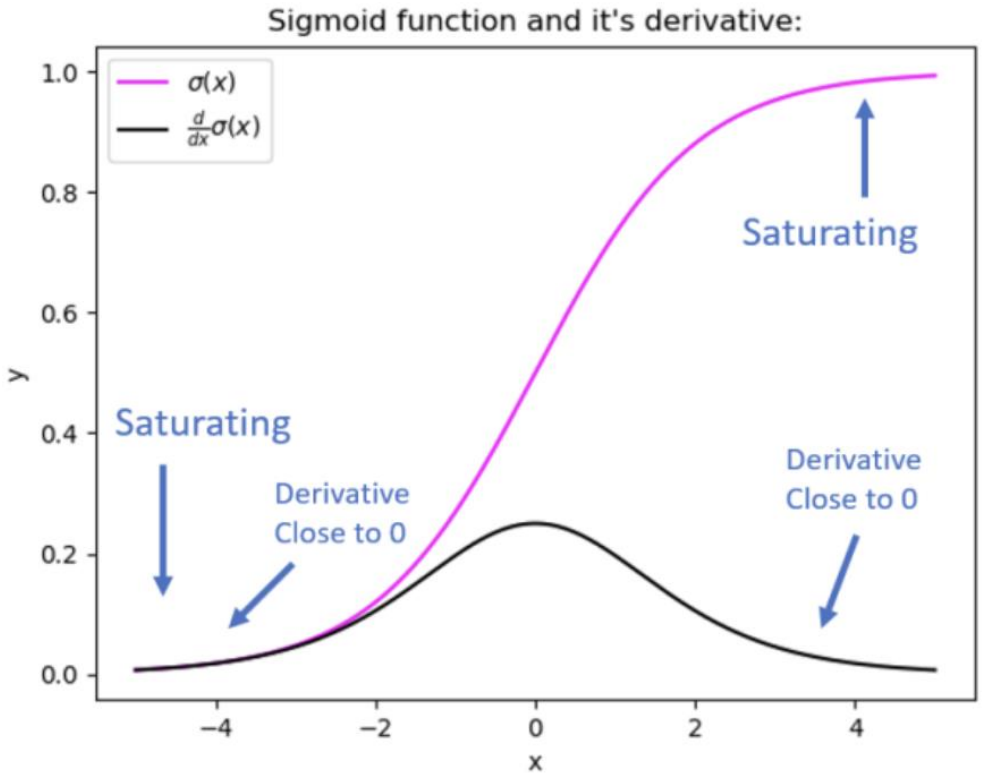
Batch Normalization

Challenges in training CNNs

- **Some of the challenges during training of CNN's**
  - **Internal covariate shift**: As the input data passes through multiple layers, the distribution of activations can change, making it difficult for the model to adapt and learn effectively.
  - **Vanishing and exploding gradients**: Deep networks can suffer from vanishing or exploding gradients, where the gradients become too small or too large during backpropagation, preventing effective weight updates.
  - **Sensitivity to initialization**: The initial weights of the network can significantly impact the training process, and poor initialization can lead to slow convergence or even training failure.



Batch Normalization



The vanishing or exploding
Gradient issue is commonly seen with
Tanh or Sigmoid activation functions
Since the O/P tends to be concentrated
towards the extreme ends of curve
0 or 1 for Sigmoid
-1 or 1 for tanh

- **Batch Norm to the rescue - It is a normalization technique done between the layers of a Neural Network instead of in the raw data**. It is done along mini-batches instead of the full data set.

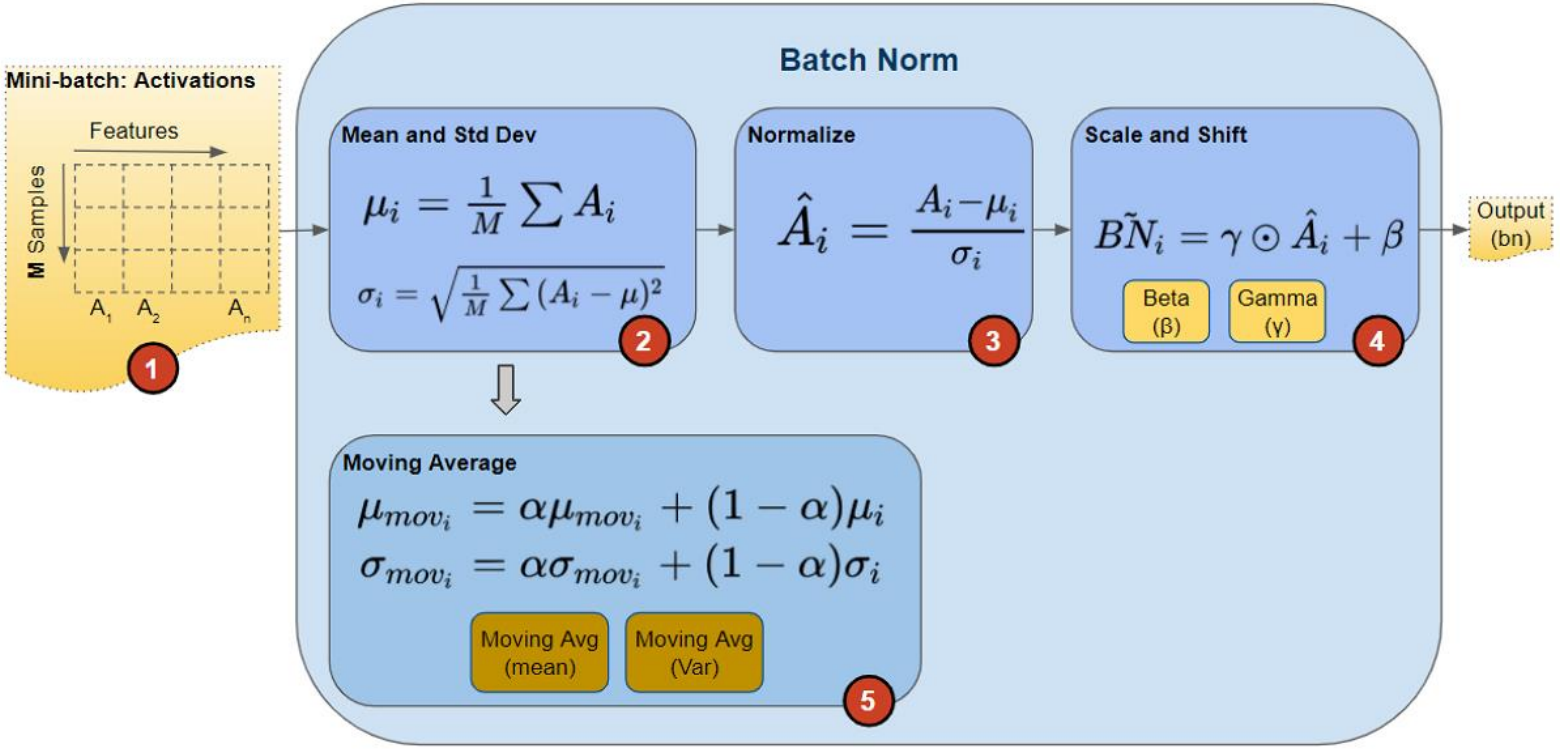
Batch Normalization

- The most common form of data normalization is centering the data on zero by subtracting the mean from the data, and giving the data a unit standard deviation by dividing the data by its standard deviation.
- In effect, this makes the assumption that the data follows a normal (or Gaussian) distribution and makes sure this distribution is centered and scaled to unit

$$normalized\_data = (data - np.mean(data, axis = \cdots)) \,/\, np.std(data, axis = \cdots)$$

- Our previous examples normalized data before feeding it into models. But data normalization may be of interest after every transformation operated by the network. Even if the data entering a Dense or Conv2D network has a 0 mean and unit variance, there's no reason to expect a priori that this will be the case for the data coming. **Batch normalization acts on intermediate activations.**
- The name comes from the fact that both them mean and the standard deviation are calculated for the **whole mini_batch of samples**.
- The technique is invented by: Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Proceedings of the 32nd International Conference on Machine Learning (2015), `https:// arxiv.org/abs/1502.03167.g`

Batch Normalization



Just like the parameters (eg. weights, bias) of any network layer, a Batch Norm layer also has parameters of its own:

1. Two learnable parameters called beta and gamma.
2. Two non-learnable parameters (Mean Moving Average and Variance Moving Average) are saved as part of the 'state' of the Batch Norm layer.

BatchNormalization layer

- The BatchNormalization layer can be used after any layer—Dense, Conv2D, etc.:

```
x = ...
x = layers.Conv2D(32, 3, use_bias=False)(x)
x = layers.BatchNormalization()(x)
```

- NOTE Both Dense and Conv2D involve a bias vector, a learned variable whose purpose is to make the layer affine rather than purely linear. Because the output of the Conv2D or Dense layer gets normalized, the layer doesn't need its own bias vector.
- It is generally recommended placing the previous layer's activation *after* the batch normalization layer. The following is how not to use Batch Normalization.

```
x = layers.Conv2D(32, 3, activation="relu")(x)
x = layers.BatchNormalization()(x)
```

- The following is the proper way of ordering layers

```
x = layers.Conv2D(32, 3, use_bias=False)(x)
x = layers.BatchNormalization()(x)
x = layers.Activation("relu")(x)
```

- `Note:` The intuitive reason for this approach is that batch normalization will center your inputs on zero, while your `relu` activation uses zero as a pivot for keeping or dropping activated channels: doing normalization before the activation maximizes the utilization of the `relu`.
- Batch normalization is usually applied after every convolutional block, sometimes even after every convolutional layer.

Extensions to Batch Normalization

- **Variants and Extensions of Batch Normalization**
  - **Layer normalization** operates on the activations across all channels within a layer, rather than across the batch dimension. Layer normalization is particularly useful for recurrent neural networks (RNNs) and scenarios where the batch size is small or variable.
  - **Group normalization** divides the channels into groups and computes the normalization statistics within each group. Group normalization is effective in scenarios where the batch size is limited, such as in memory-constrained environments or when processing high-resolution images.
  - **Instance normalization** applies normalization to each individual channel of each sample in the batch. Instance normalization has been shown to improve the quality and stability of generated images in such applications
  - **Batch renormalization** extends batch normalization by introducing additional correction terms to the normalization process. It aims to address the discrepancy between the batch statistics and the population statistics
  - **Weight Normalization** is a technique that reparameterizes a neural network's weights to decouple the weights' magnitude and direction.

Training CNNs on Small Datasets

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 will focus on classifying images of dogs or cats, in a dataset containing 4,000 pictures of cats and dogs (2,000 cats, 2,000 dogs). We will 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.
- Subsequently, we will introduce
  1. ***Data Augmentation*,** a powerful technique for mitigating overfitting in computer vision deep learning tasks. Data augmentation, as will be demonstrated, will improve the accuracy of our deep learning network.
- We will review two additional techniques for applying deep learning to small datasets:
  2. ***Transfer Learning or Feature Extraction with a pre-trained network*** (which will get us to an accuracy of 90% to 96%) and
  3. ***Fine-tuning of pre-trained networks*** (this will get us to a final accuracy of 97%).
- These three strategies are the main tools for tackling the problem of performing image classification with small datasets.
- Training a CNN from scratch on a very small image dataset yields reasonable results despite a relative lack of data, without the need for any custom feature engineering.

- While learning above techniques, we will practice the use of existing, large pretrained networks.

Dogs and Cats Dataset from Kaggle

- We will continue to use the dataset from www.kaggle.com/c/dogs-vs-cats/data, we saw in the last lecture.






- This dataset contains 25,000 images of dogs and cats (12,500 from each class), size of 800 MB (compressed). After downloading and uncompressing the data, we create a smaller 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.

Building the Network

- We will start with the small dataset we created during the last lecture.
- We will also use a moderately simple CNN we defined last time.
- We will apply the same technique with `train_generator` and `test_generator` to feed our `fit()` method.
- As you recall, our network gave unimpressive results.

Network Model

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.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 tensorflow.keras import optimizers
model.compile(loss='binary_crossentropy',
optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),
metrics=['acc'])
```

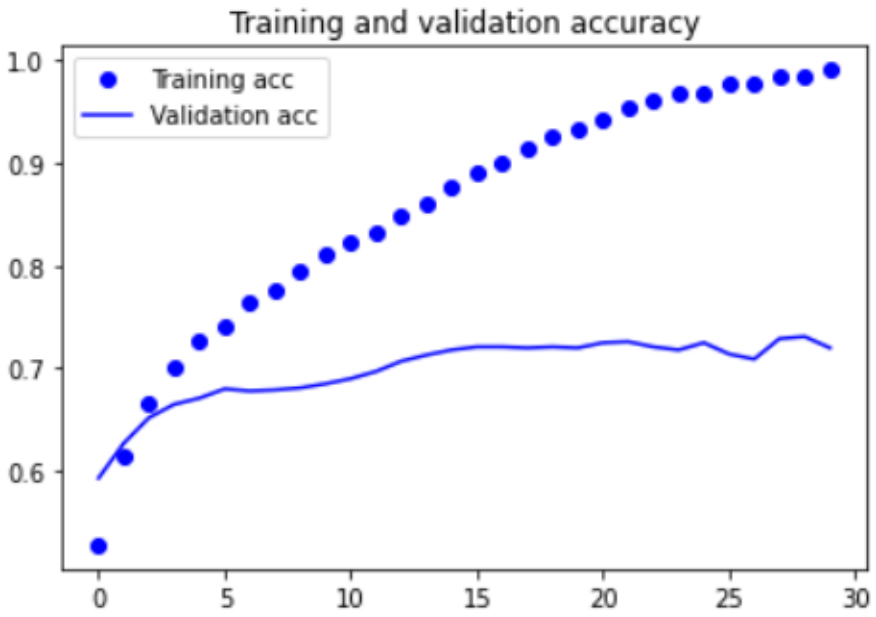
Dimensions of features maps

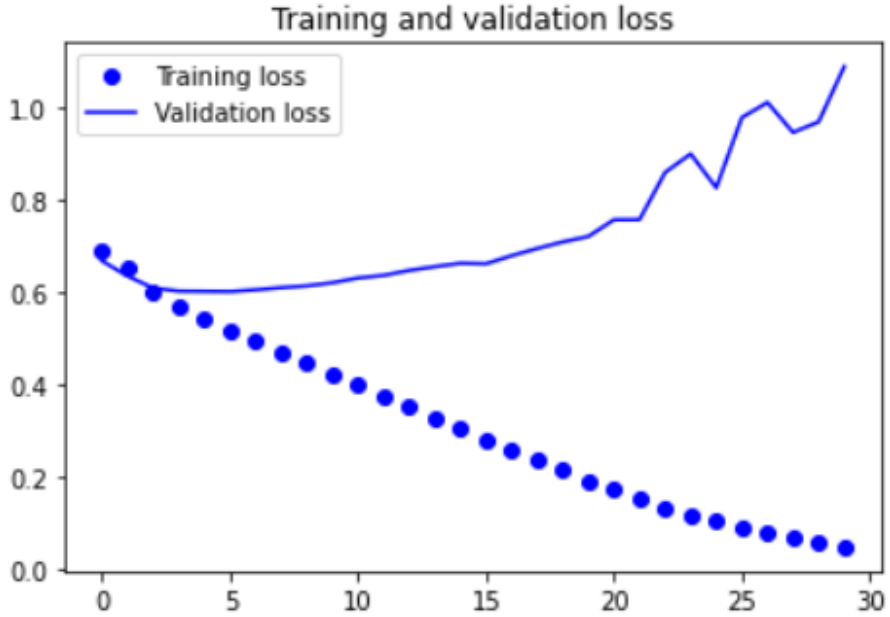
```
model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (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
```

Modest Result

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

- Validation accuracy of our CNN reach some 70+%. That is not satisfactory. We know that we can distinguish cats from dogs with a much higher accuracy.





- We also see that network overfits after 5-10 epochs.

Data Augmentation

Data Augmentation

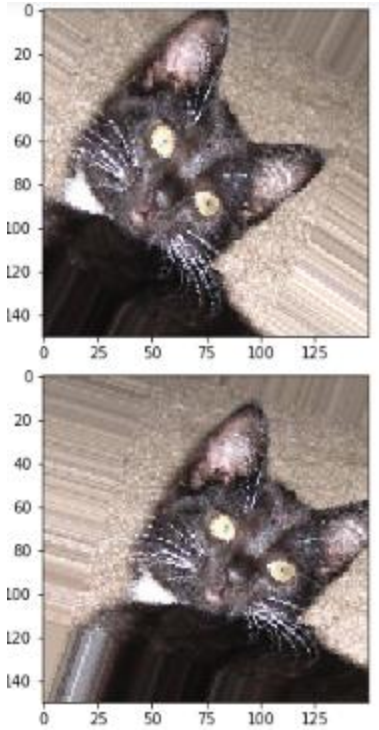
- A most frequent problem is the lack of training data. In our case, the paucity of data is artificial. Quite often we have no mechanisms for acquiring very many real samples. For example, it takes a lot of time to make and label 1 million photographs.
- One possible resolution is to generating more training data from the existing training samples, by artificially modifying (*augmenting)* existing samples via many random transformations that yield believable-looking images. The technique is called *Data Augmentation.*
- We generate augmented data at the training time, and the model never sees the exact same picture twice. This helps expose the model to more aspects of the data and model generalizes better.
- In Keras, this can readily be done by performing a number of random transformations 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')
```

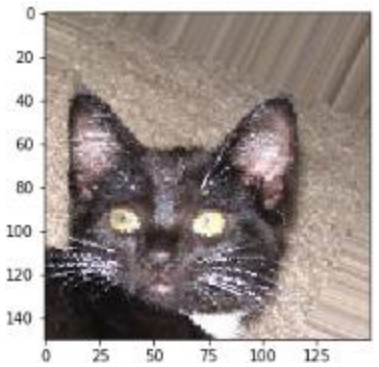
Options of `ImageDataGenerator`

- These are a few of the options available (see the Keras documentation for all).
  - `rotation_range` is a value in degrees (0–180), a range within which to randomly rotate pictures.
  - `witdth_shift` 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 tensorflow.keras import utils
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 = utils.load_img(img_path, target_size=(150, 150))
# Convert it to a Numpy array with shape (150, 150, 3)
x = utils.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 cannot 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, we also add a `Dropout` layer to the model, right before the `Dense`ly connected classifier. New model now reads:

```
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(learning_rate=1e-4),
        metrics=['acc'])
```

Training the Network

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
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=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,           # steps_per_epoch = int(train_samples_no/batch_size)
   class_mode='binary') # validation_steps = int(validation_samples_no/batch_size)
history = model.fit( train_generator, steps_per_epoch=int(2000/20),
  epochs=100,   validation_data=validation_generator, validation_steps=int(1000/20))
model.save('cats_and_dogs_small_2.h5')
```

Training starts to become long

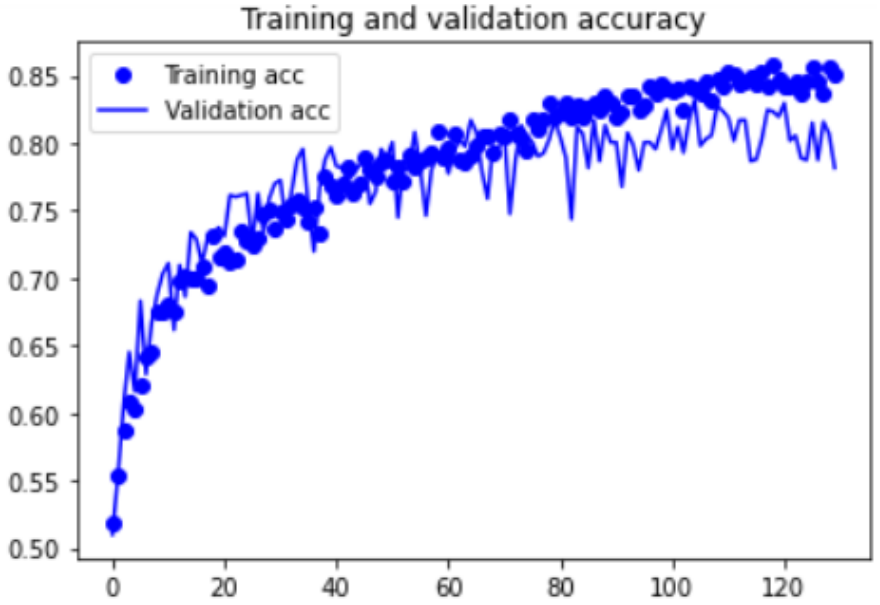
- On mac m1 each epoch took ~12 seconds . . . .

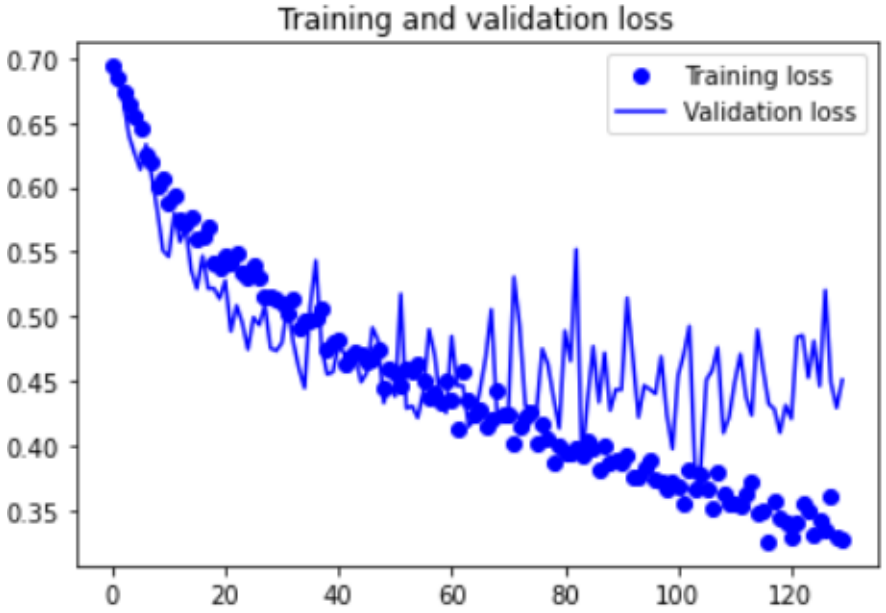
```
ound 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Epoch 1/30
100/100 [============] - 12s 115ms/step - loss: 0.6943 - acc: 0.5140 - val_loss: 0.6879 - val_acc: 0.5540
Epoch 2/30
100/100 [============] - 11s 114ms/step - loss: 0.6868 - acc: 0.5510 - val_loss: 0.6793 - val_acc: 0.5610
Epoch 3/30
100/100 [===  =======] - 11s 114ms/step - loss: 0.6731 - acc: 0.5785 - val_loss: 0.6550 - val_acc: 0.5920
Epoch 4/30
100/100 [============] - 11s 115ms/step - loss: 0.6619 - acc: 0.5990 - val_loss: 0.6480 - val_acc: 0.6100
.  . . . .
Epoch 28/30
100/100 [============] - 12s 123ms/step - loss: 0.5309 - acc: 0.7295 - val_loss: 0.5333 - val_acc: 0.7420
Epoch 29/30
100/100 [============] - 12s 120ms/step - loss: 0.5253 - acc: 0.7410 - val_loss: 0.5005 - val_acc: 0.7560
Epoch 30/30
100/100 [============] - 12s 119ms/step - loss: 0.5172 - acc: 0.7450 - val_loss: 0.5643 - val_acc: 0.7020
Let's save our model -- we will be using it in the section on convnet visualization.
```

- We need to go to run for 100 epochs to see higher accuracy.
- A few observations: Because we pass every image read from the `train_directory` through `ImageDataGenerator`, we end up with a different randomly transformed image whenever we visit a dog's or a cat's image. Without `ImageDataGenerator` set to generate random transformations, every epoch will only train on original images which would repeat from epoch to epoch. Now, every epoch sees different samples.
- Information content of those samples is not vastly different from the information content of the original images and the improved accuracy reaches to only 82-83%.

Accuracy & Loss, Augmentation

- Thanks to data augmentation overfitting takes place later: the training curves are closely tracking the validation curves. You now reach an accuracy of 80%, a 10% relative improvement over the non-augmented model. Note that the overfitting starts only around 80 epochs. The total duration of this experiment is 130 epochs

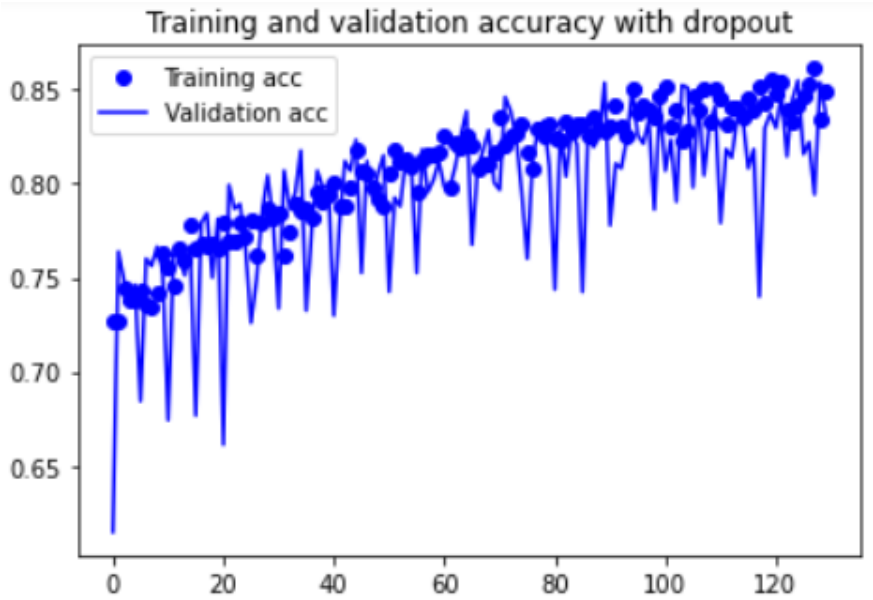


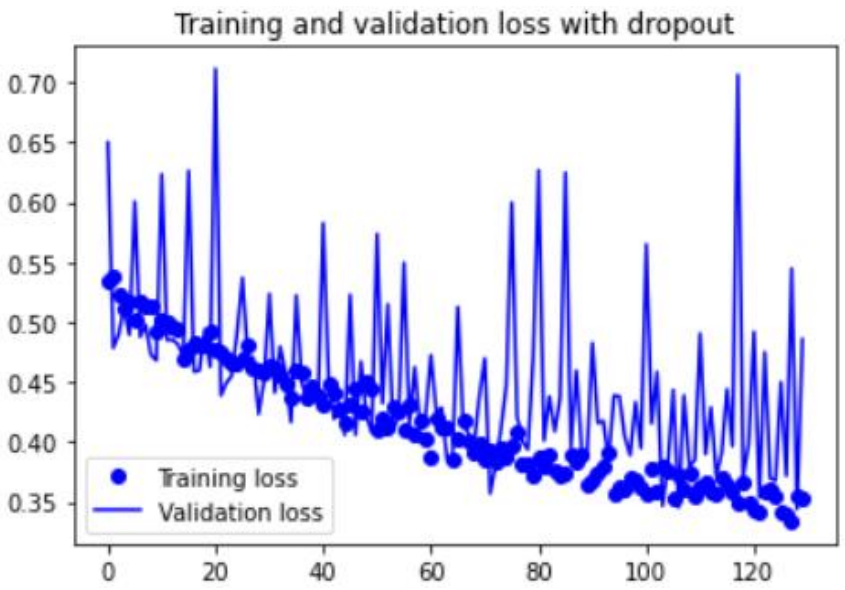


- Let us now add a Dropout Layer.

Accuracy & Loss, Augmentation & Dropout

- With both augmentation and dropout, the overfitting is pushed beyond 120-13 epochs. The training curves are closely tracking the validation curves. You now reach an accuracy of 84%, a 14-15% relative improvement over the non-regularized model. Experiments with 130 epochs

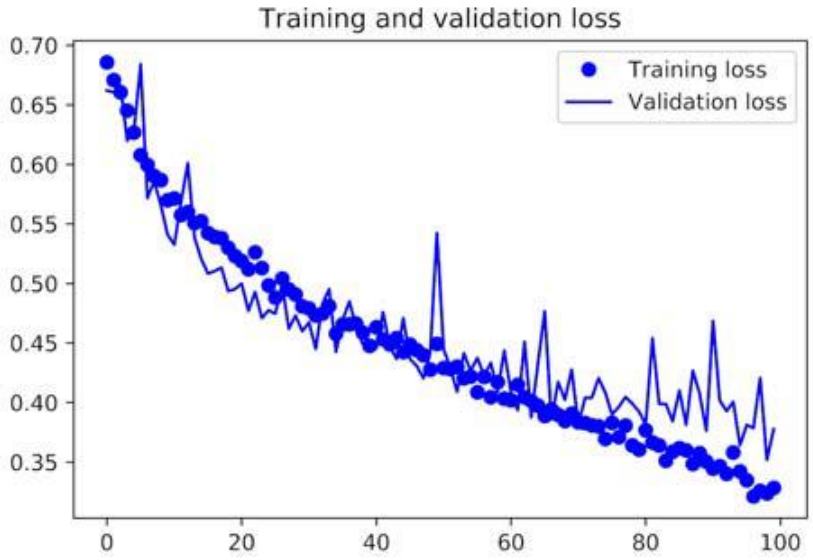




- Some further increase in accuracy could be obtained if we would add regularization procedure. We will examine another technique, called pre-trained networks.

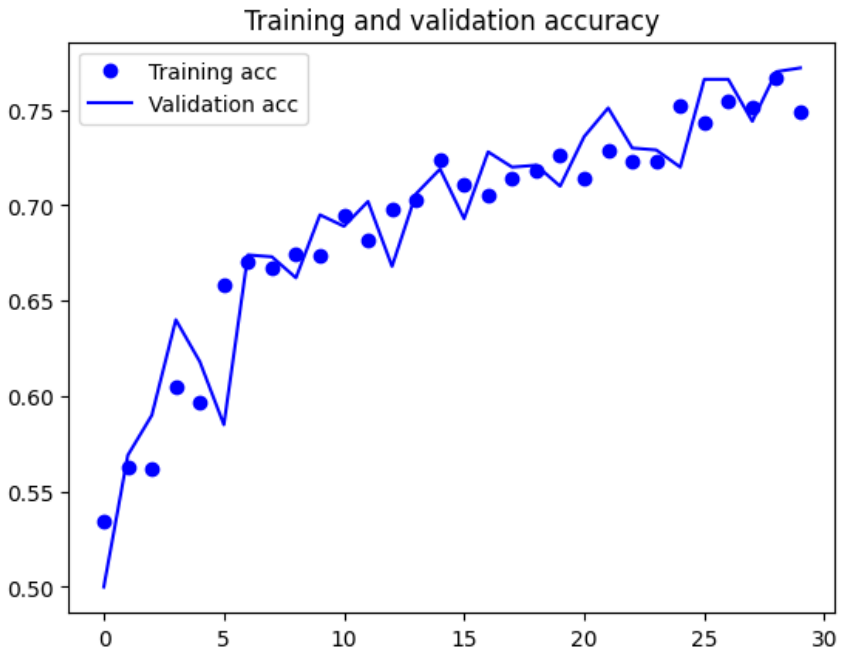
Training and Validation, Augmentation & Dropout

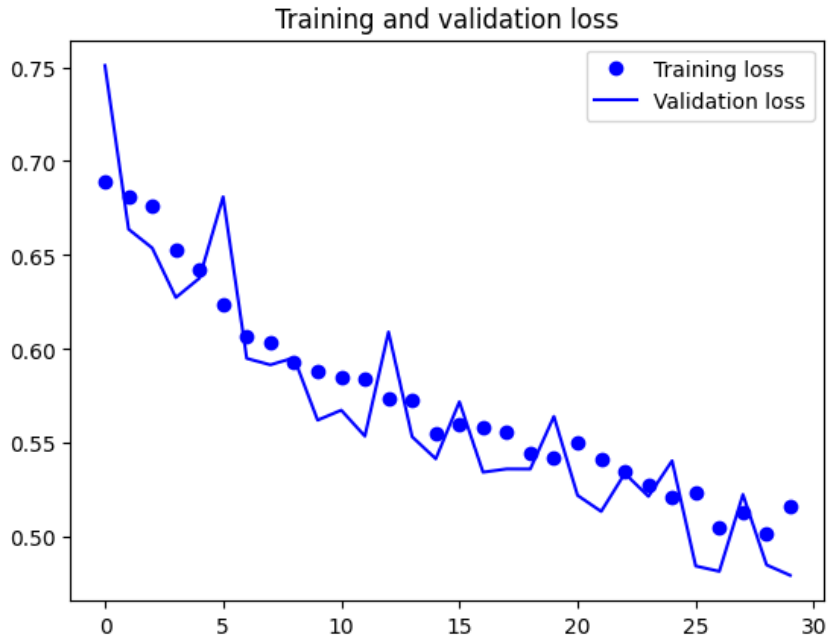
- These are curves from a shorter experiment with 100 epochs.
- We see an accuracy of 83-84%. No overfitting is visible.



Training and Validation, Augmentation & Dropout

- These are curves from a shorter experiment with 30 epochs.
- We see an accuracy of 76%. No overfitting is visible.





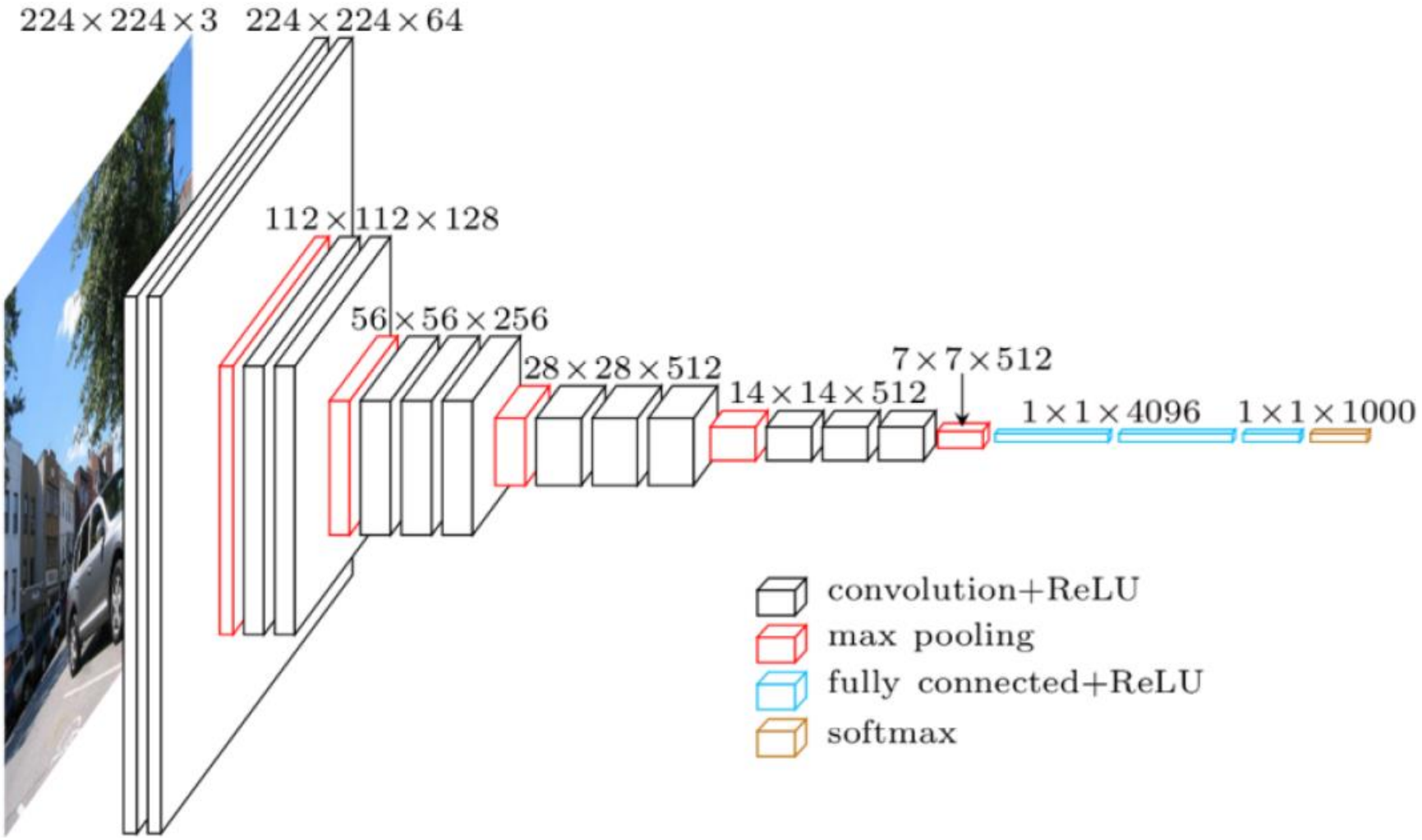
Transfer Learning

Foundational Pretrained Networks

Model	Specification
ResNet (Residual Networks)	ResNet was introduced by Microsoft Research by using residual connections to mitigate the vanishing gradient problem in deep networks.
GoogLeNet/Inception	Developed by Google, the Inception network uses inception modules to capture multi-scale features.
VGG	Developed by the Visual Geometry Group at the University of Oxford, VGG models are known for their simplicity and depth.
MobileNet	Developed by Google, MobileNet models are designed for mobile and embedded vision applications.
Xception (Extreme Inception)	Developed by Google, Xception is an extension of the Inception architecture with depthwise separable convolutions.
AlexNet	Developed by Alex Krizhevsky, AlexNet is one of the earliest deep learning models that popularized the use of CNNs in image classification.
Vision Transformers (ViT)	Developed by Google, Vision Transformers apply the transformer architecture, initially designed for NLP, to image classification.

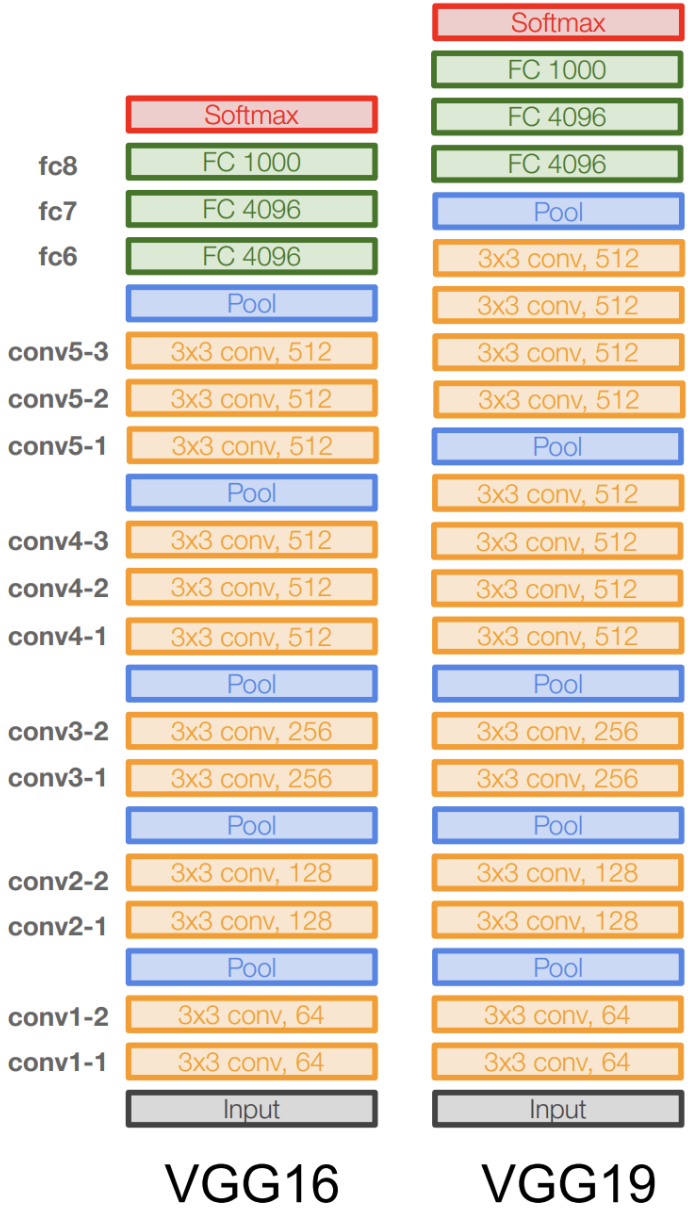
VGGNet Architecture

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

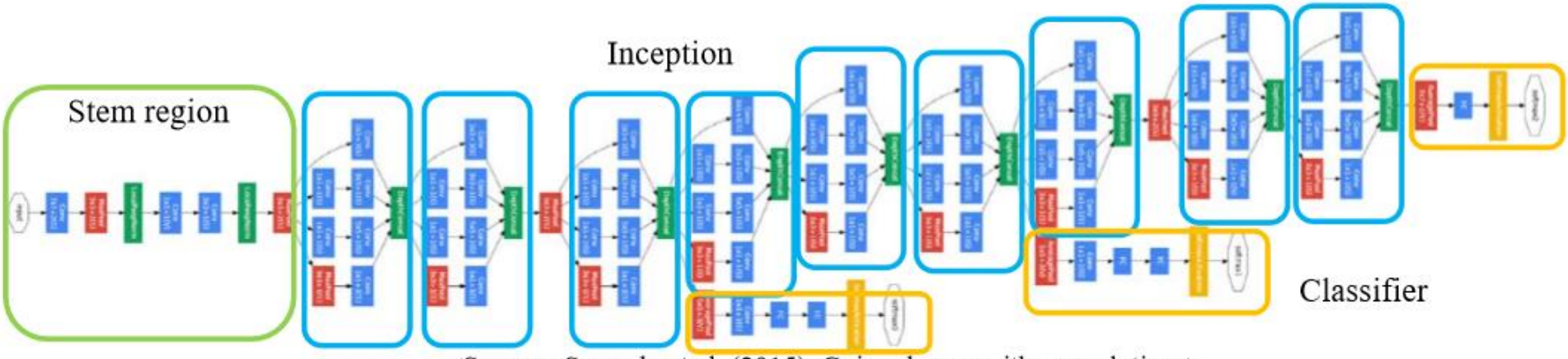


VGG16 & VGG19

Layer	VGG16	VGG19
Size of Layer	41	47
Image Input Size	224x224 pixel	224x224 pixel
Convolutional Layer	13	16
Filter Size	64 & 128	64,128,256, & 512
ReLU	5	18
Max Pooling	5	5
FCL	3	3
Drop Out	0.5	0.5
Softmax	1	1



GoogLeNet Architecture



<Source : Szegedy et al. (2015), Going deeper with convolutions>

GoogLeNet is a type of convolutional neural network based on the Inception architecture. It utilises Inception modules, which allow the network to choose between multiple convolutional filter sizes in each block. An Inception network stacks these modules on top of each other, with occasional max-pooling layers with stride 2 to halve the resolution of the grid.

Pre-trained CNNs

- A common and highly effective approach to deep learning on small image datasets is the use of pre-trained networks.
- A *pre-trained network* is a saved network that was previously trained on a large dataset, such as a large-scale image-classification task.
- If the original dataset is large enough and general enough, 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.

How to Use Pre-Trained Models

- For example, a model may be downloaded and used as-is, such as embedded into an application and used to classify new photographs.
- Alternately, models may be downloaded and used as feature extraction models. Typically, the output of a model from a layer prior to the output layer is used as input to a new classifier model.
- Recall that convolutional layers closer to the input layer of the model learn low-level features such as lines, that layers in the middle of the layer learn complex abstract features that combine the lower-level features extracted from the input, and layers closer to the output interpret the extracted features in the context of a classification task.
- This means that a level of detail for feature extraction from an existing pre-trained model can be chosen. For example,
  - If a new classification task is quite different from classifying objects in photographs (e.g., new images do not contain members of classes in ImageNet), then perhaps the output of the pre-trained model after the few layers would be appropriate.
  - If a new task is quite like the task of classifying objects in the original photographs, then perhaps the output from layers much further into the model can be used, or even the output of the fully connected layer prior to the output layer can be used.
- Pre-trained model can be used as a separate feature extraction program, in which case input can be pre-processed by the model or portion of the model to give an output (e.g., vector or tensor of numbers) for each input image. That vector (tensor) can then be used as an input when training a new model.
- Alternately, the pre-trained model or desired portion of the model can be integrated directly into a new neural network model. In this usage, the weights of the pre-trained can be frozen so that they are not updated as the new model is trained.
- Also, the weights may be updated during the training of the new model, perhaps with a lower learning rate, allowing the pre-trained model to act like a weight initialization scheme when training the new model.

Types of Transfer Learning

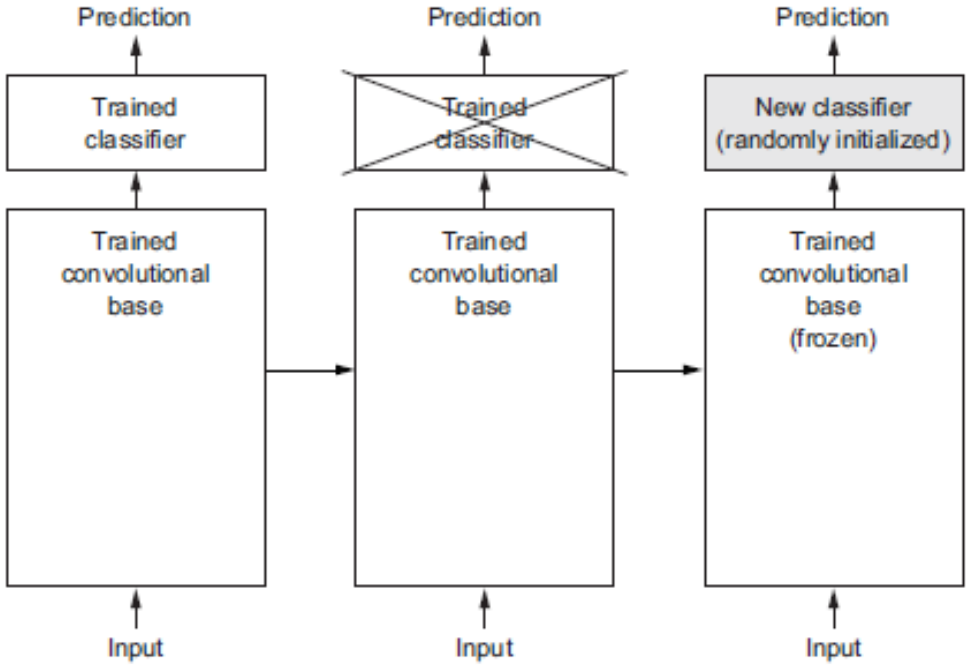
- Transfer learning could be used in several ways:
  - **Classifier:** The pre-trained model is used directly to classify new images.
  - **Standalone Feature Extractor:** The pre-trained model, or some portion of the model, is used to pre-process images and extract relevant features.
  - **Integrated Feature Extractor**: The pre-trained model, or some portion of the model, is integrated into a new model, but layers of the pre-trained model are frozen during training.
  - **Weight Initialization:** The pre-trained model, or some portion of the model, is integrated into a new model, and the layers of the pre-trained model are trained in concert with the new model.
- Each approach can be effective and save significant time in developing and training a deep convolutional neural network model.

VGG16

- For our illustrations we will use VGG16, 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.
- VGG16 architecture is 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 architecture but is quite similar to 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. You will encounter those models if you keep doing deep learning for computer vision.
- Two main methods of use of pre-trained networks are:
  - *Transfer learning (feature extraction)* and
  - *Fine-tuning*.
- We will cover both methods.

Transfer Learning (Feature Extraction)

- Transfer Learning (Feature extraction) uses 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 has 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

- The representations learned by the convolutional base are more generic and therefore more reusable: the feature maps of a CNN maps generic concepts in analyzed pictures, what are likely to be useful regardless of the computer-vision problem at hand.
- The representations learned by the (Dense) classifier are necessarily specific to the set of classes on which the model is trained—they only contain information about the probability of this or that class in the entire picture.
- Representations found in densely connected layers no longer contain any information about *the location of* objects in the input image. Dense classifier layers lose the notion of space. The object location is still described by convolutional feature maps.
- In the cases when 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.
- To see the full list of Keras applications, go to: https://github.com/keras-team/keras-applications

Instantiate VGG16 Model

```
from tensorflow.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 like 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 we used in previous examples.

   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. This technique is far more expensive than the first.

Fast feature extraction without data augmentation

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 = 'cats_and_dogs_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:

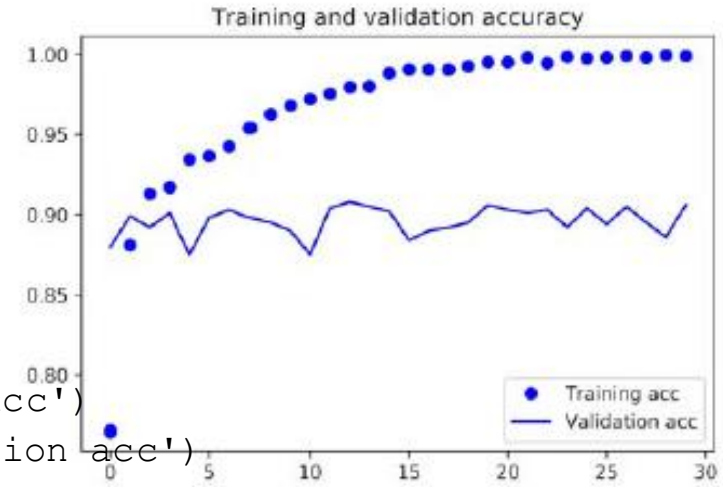
```
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import optimizers
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer=optimizers.RMSprop(learning_rate=2e-5),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

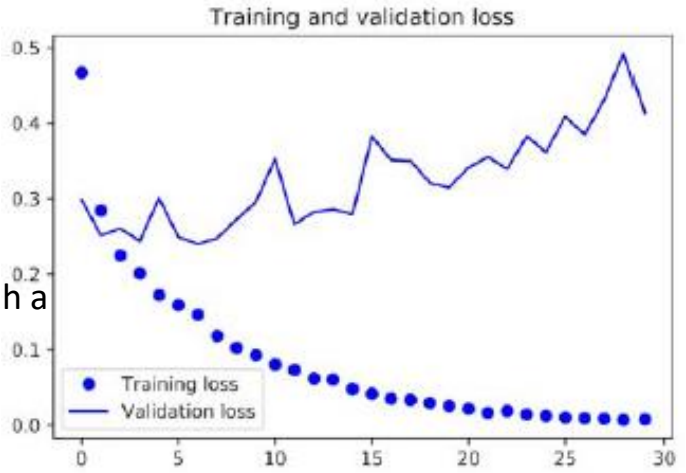
- Training is very fast, because you only deal with two `Dense` layers—an epoch takes less than one second even on CPU.

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. It uses 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 tensorflow.keras import models
from tensorflow.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 a very large number. 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

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
```

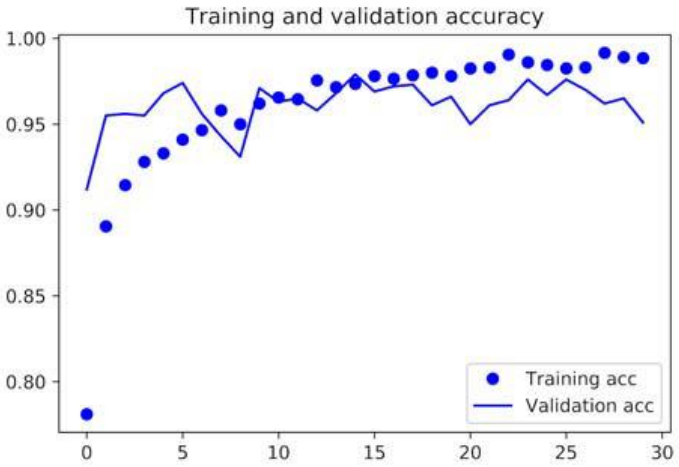
- With this setup, only the weights from the two Dense layers that you added will be trained. That's a total of four weight tensors: two per layer (the main weight matrix and the bias vector).
- Note that in order for these changes to take effect, you must first compile the model. If you ever modify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

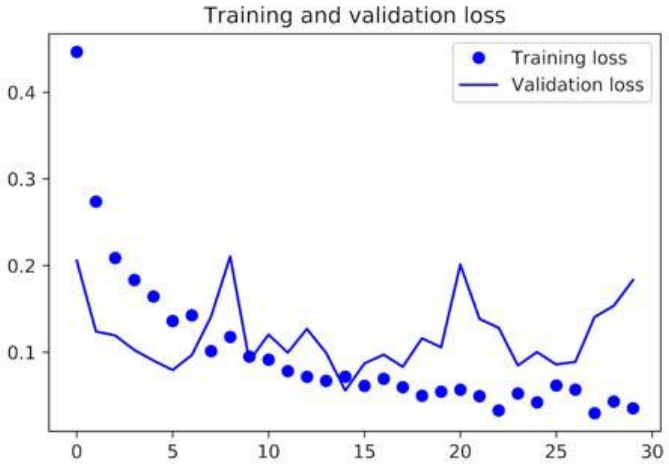
Training the model with a frozen convolutional base

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.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(learning_rate=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

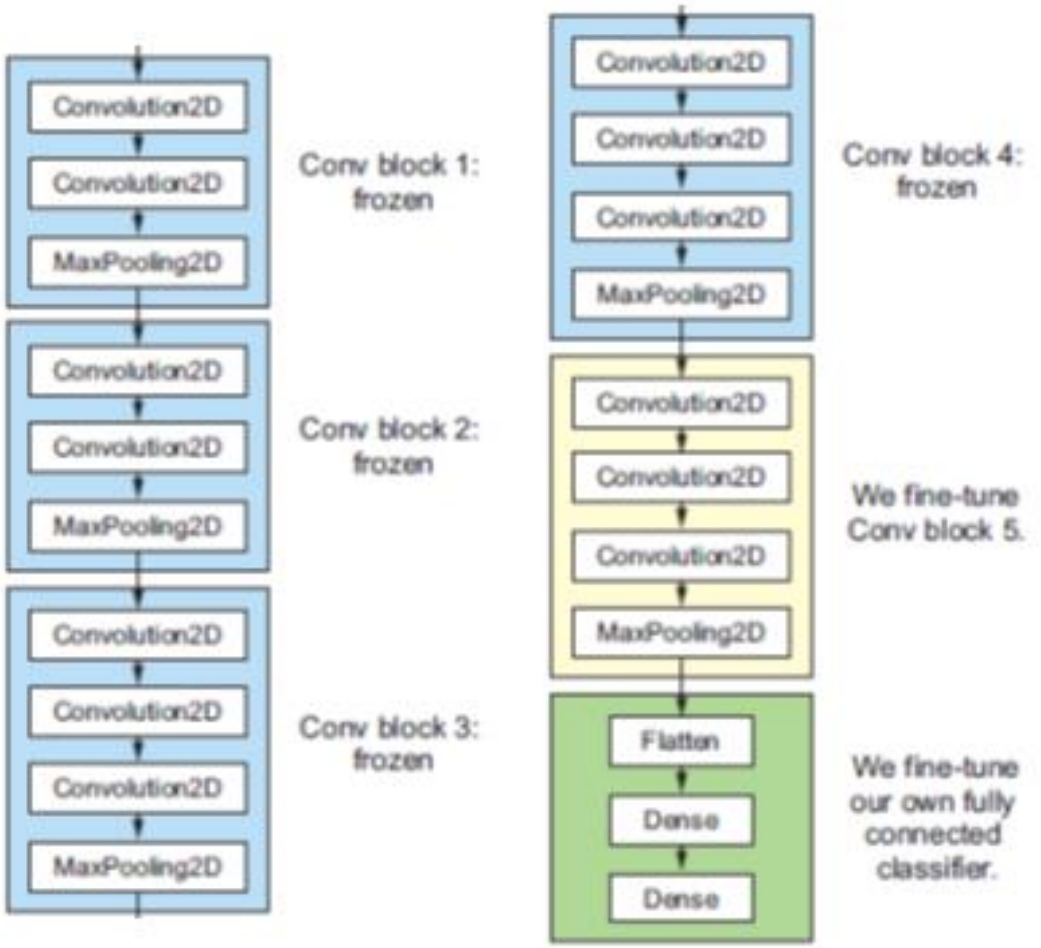




Fine Tuning

Fine Tuning

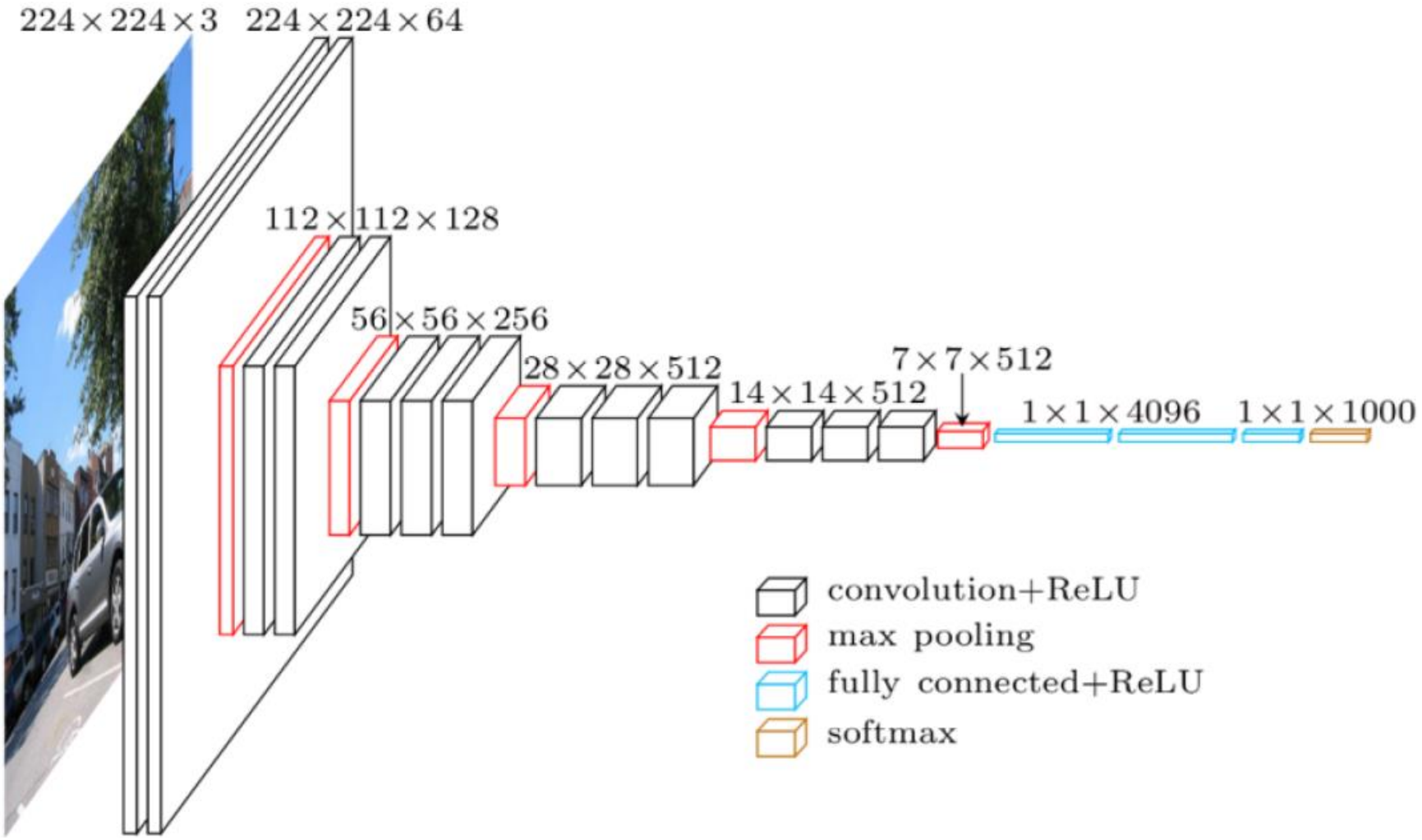
- Another frequently used technique for model reuse, complementary to feature extraction, is *fine-tuning*.
- Fine-tuning consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model (in this case, the fully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

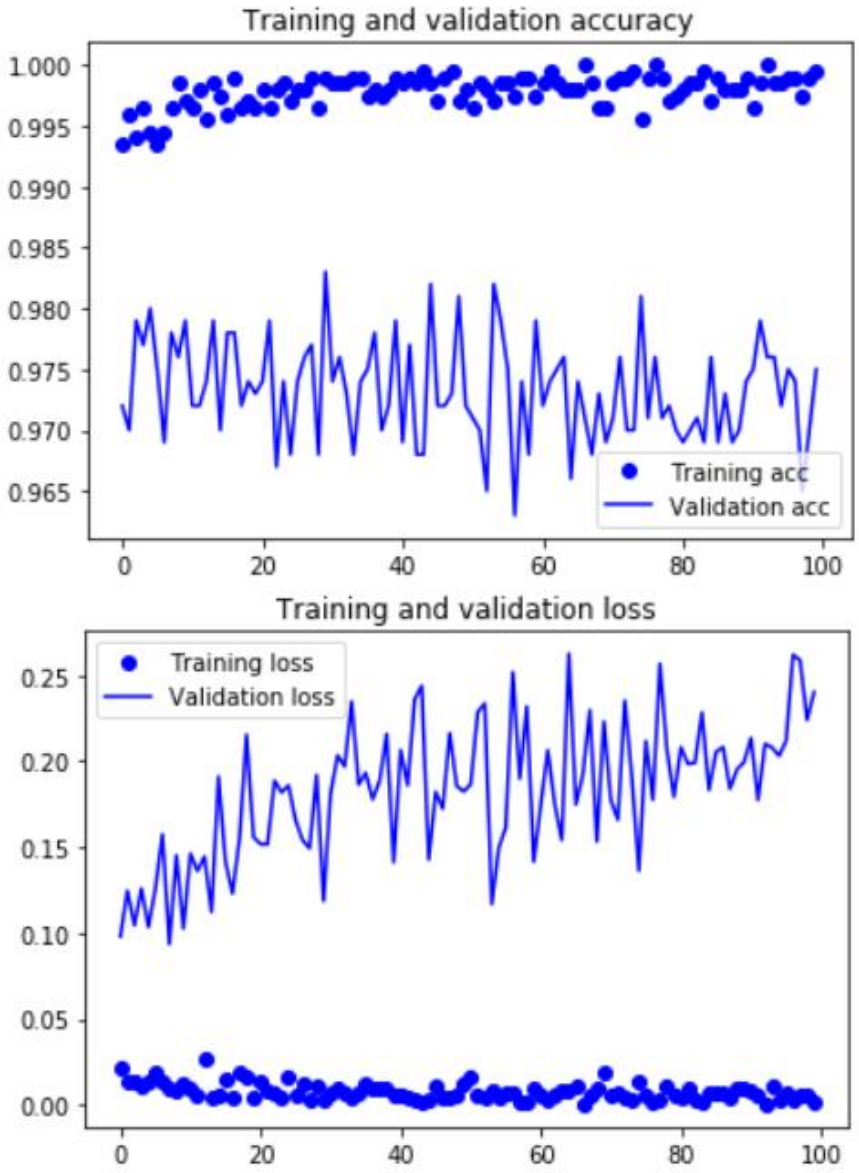
```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(learning_rate=1e-5),
              metrics=['acc'])

history = model.fit(
      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 not 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 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.

CNN Trivia

- How many Feature-maps in a layer
  - Equal to number of filters
- Does a CNN have a Backprop ?
  - Yes it does
- Is the Backprop also a Convolution ?
  - It is a convolution
- Why even bother adding a Pooling layer ?
  - Subsample the input image and reduce computational load
- What's the difference between a Kernel and a Filter ?
  - No Difference they are the same
- How do you define the shape of a filter ? Say a Whisker of a cat or tail of a horse or leg of a cow etc etc
  - You do not ! That's the whole point , filters are learned !
- What is a Filter and how do you tell what filter to use and number of filters to a CNN ?
  - It's a matrix representation of neurons weights (small image of size 3x3 or 7x 7 ..etc of receptive field
  - model.add(Conv2D(32, (3, 3), activation='relu', input_shape=input_shape))