Lecture 06
CNNs
Dealing with small datasets, Callbacks

cscie-89 Deep Learning, Fall 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, 1st Edition, Manning Publishing, 2017

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 of 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.
- Subsequently, we will introduce
  1. ***Data Augmentation,*** a powerful technique for mitigating overfitting in computer vision deep learning tasks. Data augmentation, as we will demonstrate, will improve the accuracy of our deep learning network.
- We will review two more essential 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 a pre-trained network*** (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 will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering.
- While learning those techniques, we will also 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 last lecture.






- 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 will 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 work with the same small dataset we used during the last lecture.
- We will also use a moderately simple CNN we defined last time.
- We will apply the same generator technique we illustrated with `train_generator` and `test_generator` to feed our `fit_generator() or fit()` method.
- As you recall, our network gave an unimpressive result.
- The network definition, summary and the training loss and accuracy are given on the next three slides.

Network Model

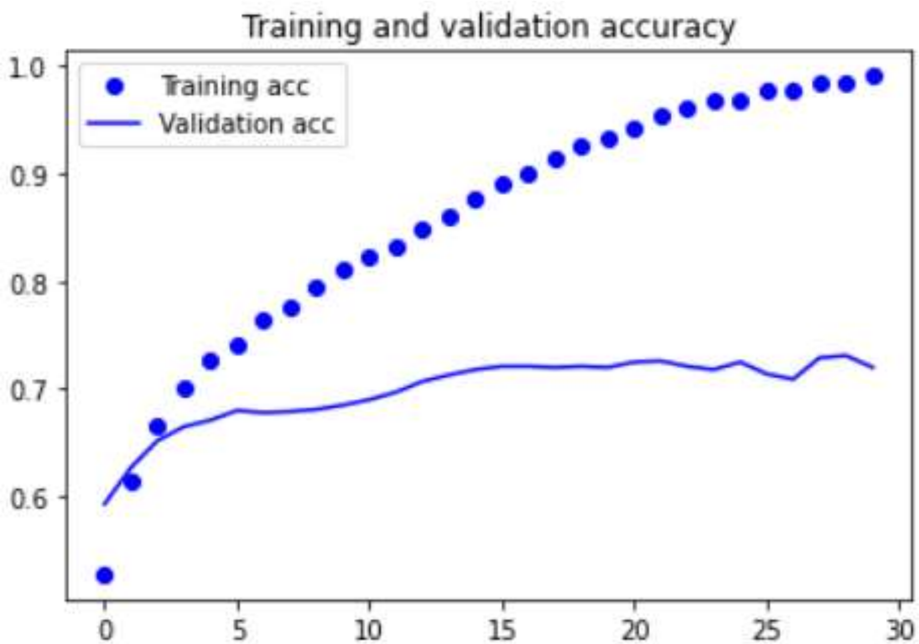
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import models
model = keras.models.Sequential()
model.add(keras.layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(keras.layers.MaxPooling2D((2, 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(lr=1e-4),
metrics=['acc'])
```

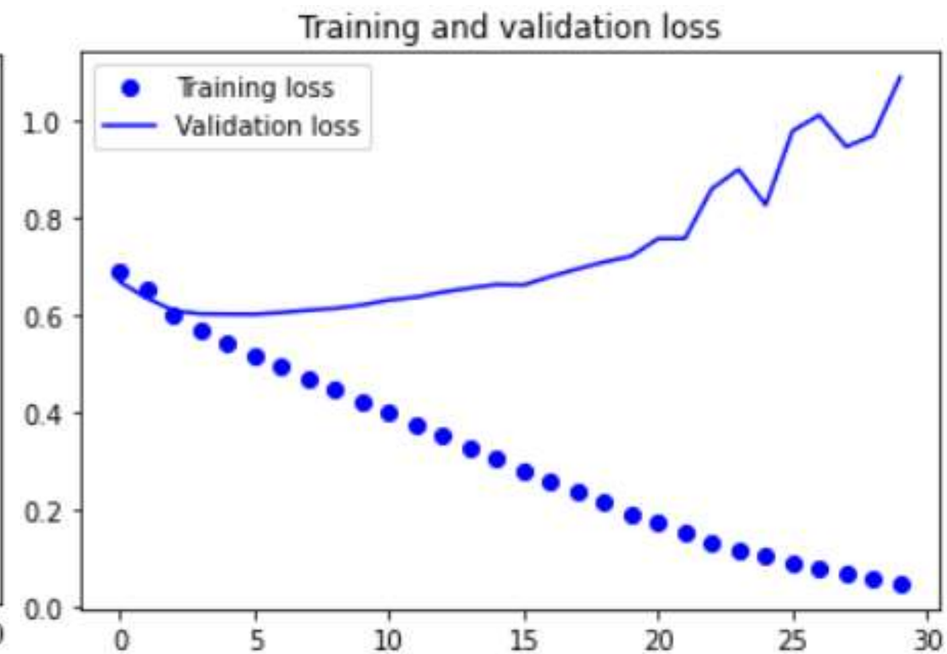
Dimensions of features maps

```
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
```

Modest Result

- Validation accuracy of our CNN reach some 70+%. That is not satisfactory, since 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

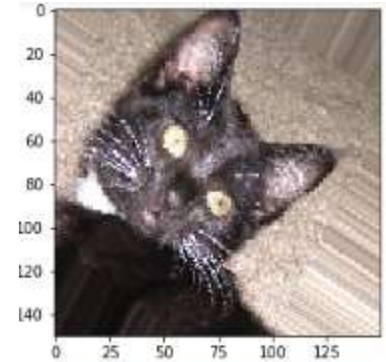
- Very often the issue is the lack of training data. In our case, the paucity of data is artificial, but we quite frequently have no mechanisms for making large number of real samples.
- One possible resolution is to generating more training data from existing training samples, by *augmenting* the samples via a number of random transformations that yield believable-looking images. The technique is called Data Augmentation.
- The goal is that at training time, the model never sees the exact same picture twice. This helps expose the model to more aspects of the data and generalizes better.
- In Keras, this can be readily 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_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(   # fit_generator() is not real needed. fit() does it
  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

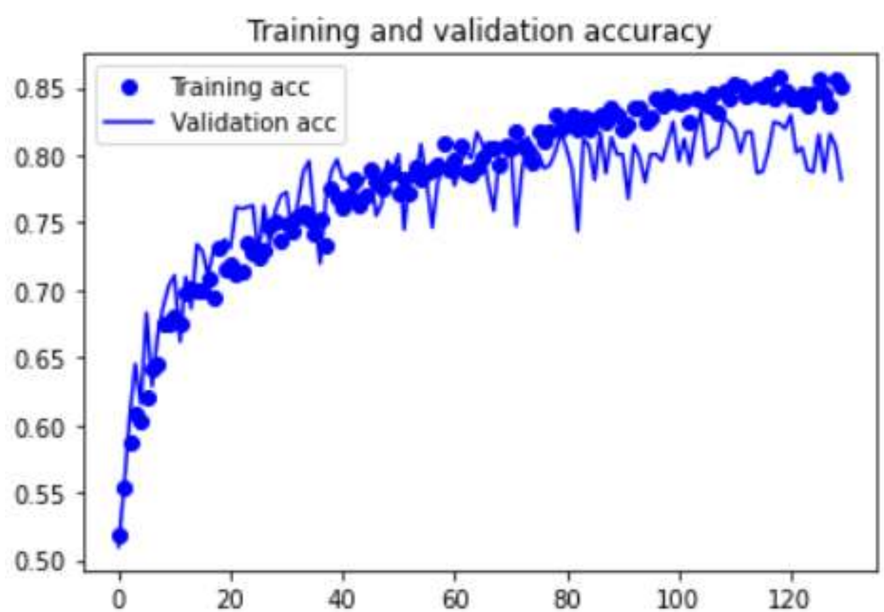
- On my Windows desktop each epoch took ~12 seconds.

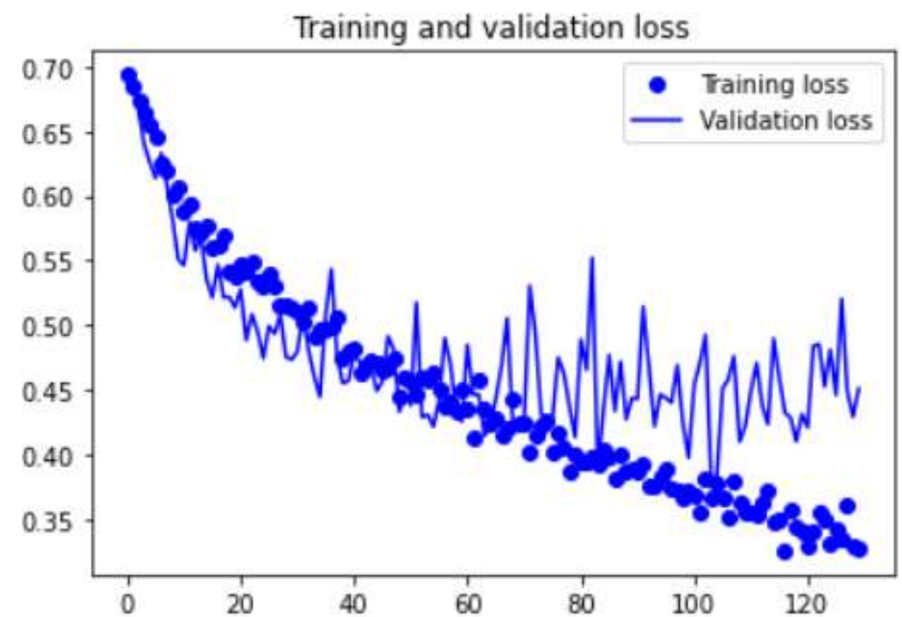
```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Epoch 1/15 100/100 [==] - 373s 4s/step - loss: 0.6903 - acc: 0.5322 -
val_loss: 0.6729 - val_acc: 0.6003
Epoch 2/15 100/100 [==] - 361s 4s/step - loss: 0.6763 - acc: 0.5597 -
val_loss: 0.6647 - val_acc: 0.5973
. . . .
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.
- 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 improvement in accuracy is limited 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

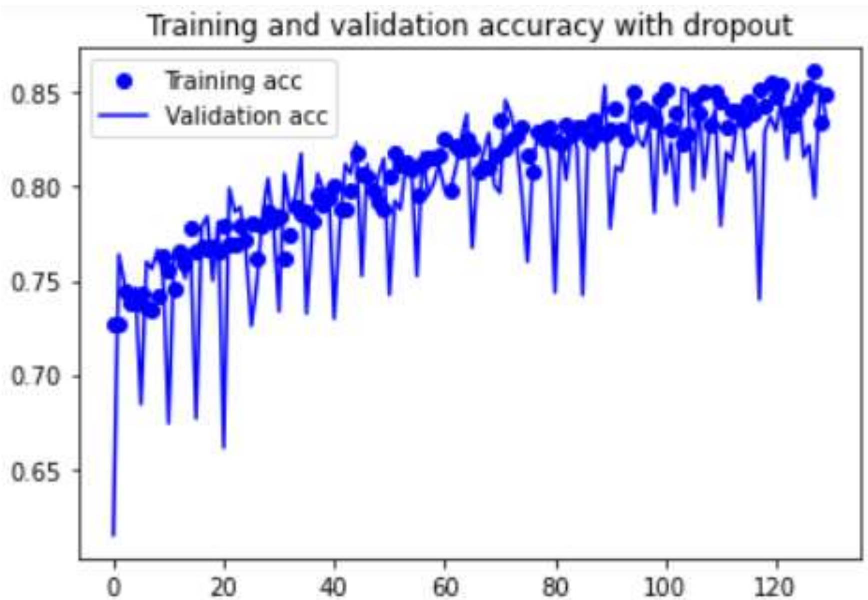


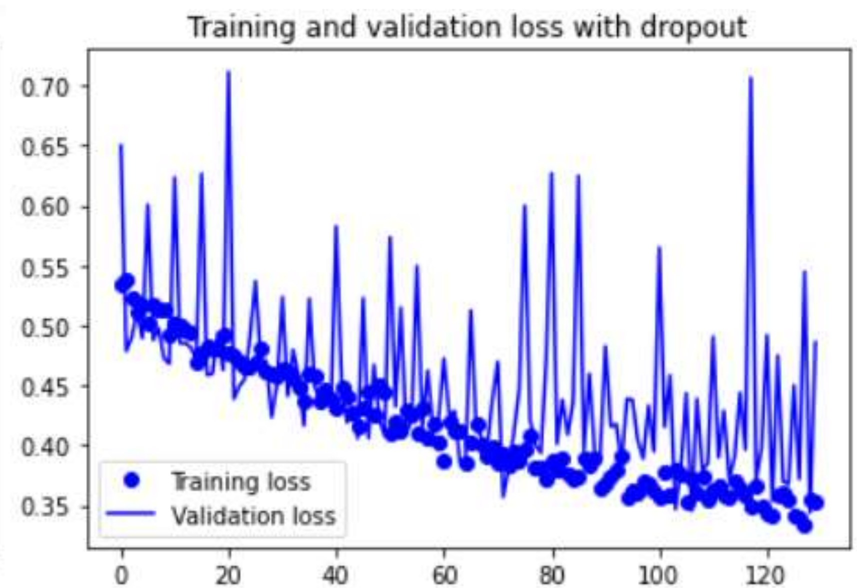


- Let us now add a Dropout Layer.

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

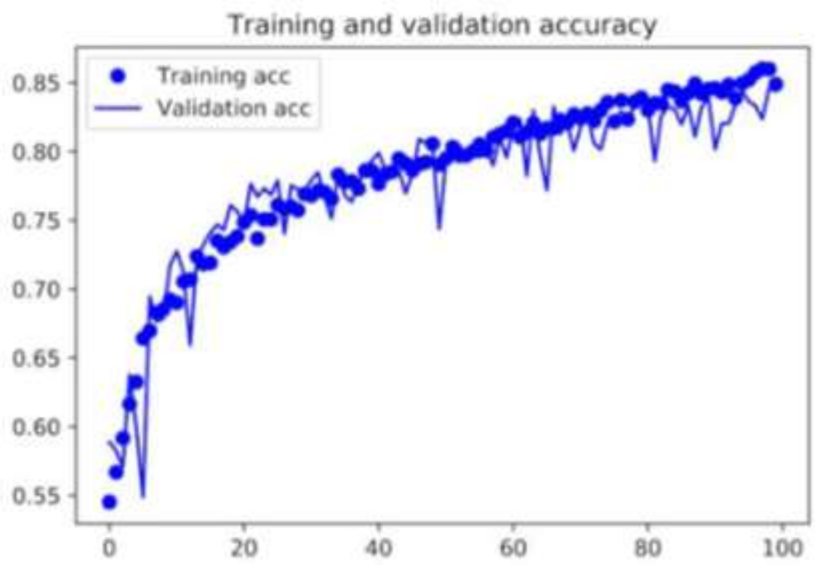


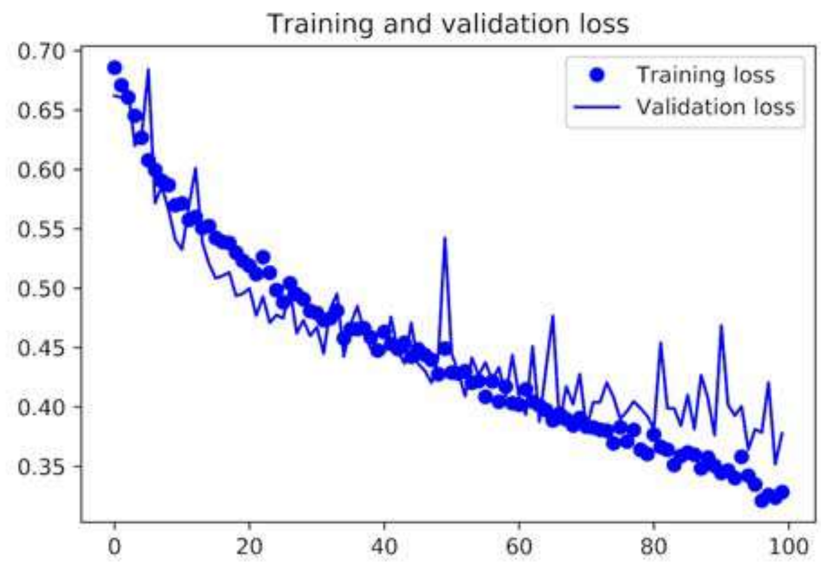


- Some further increase in accuracy could be obtained if we would add regularization procedure. We will examine another technique, called pre-trained networks.

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.





Transfer Learning or Feature Extraction

Pre-trained CNNs

- A common and highly effective approach to deep learning on small image datasets relies on the use of 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.

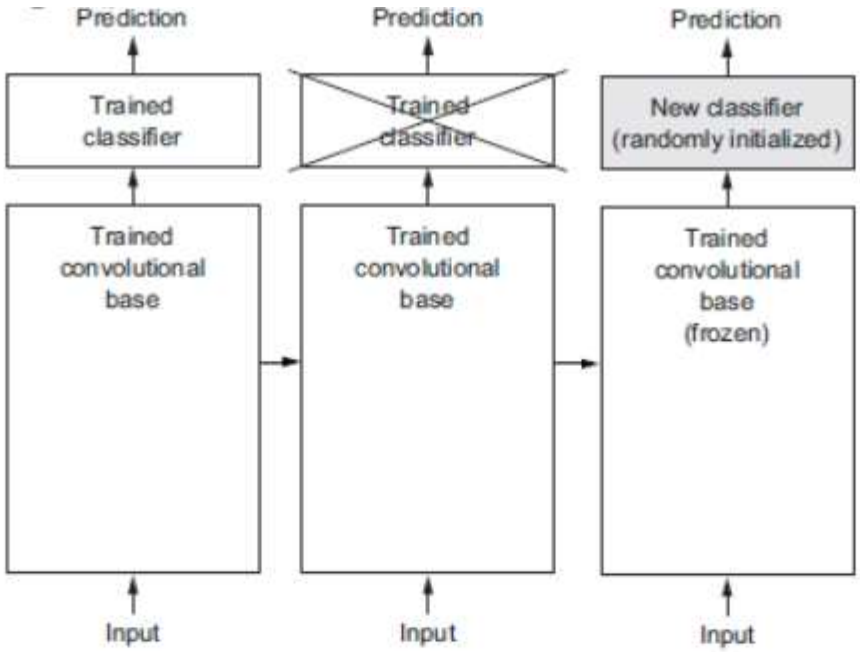
VGG16

- We will consider VGG16 architecture developed by Karen Simonyan and Andrew Zisserman in 2014; it's a simple and widely used CNN architecture for ImageNet.1
- VGG16i is a large CNN trained on the ImageNet dataset with 1.4 million labeled images and 1,000 different classes.
- ImageNet contains many animal classes, including different species of cats and dogs.
- We can expect VGG16 to perform well on the dogs-versus-cats classification problem.
- VGG16 is an older architecture but is similar to what we were using so far and is easy to understand without introducing any new concepts.
- There are many other architectures available: ResNet, Inception, Inception-ResNet, Xception, VGG19, MobileNet and so on. Trained networks with those and other architectures are available from `tf.keras.applications` package.
- There are two methods to use with pre-trained networks:
  - *feature extraction* and
  - *fine-tuning*.
- We will cover both.

Feature Extraction or Transfer Learning

- Feature extraction or Transfer Learning addresses a common use case when we need to perform classification of a small number of classes not present in an already trained model and our training set consists of a small number of new images.
- Transfer Learning uses the representations learned by the existing trained network to extract interesting features from new samples.
- Extracted features are then used in a new classifier, which is trained from scratch.

- Trained 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.
- Transfer Learning is performed by taking the convolutional base of a previously trained network, forward passing new data through it, and training a new classifier on top of the outputs.



Separation of Insights, Classification Head

- Should we reuse the densely connected classifier as well?
- In general, doing so should be avoided.
- The representations learned by the classifier will necessarily be specific to the set of classes on which the model was trained—they only contain information about the presence probability of this or that class in the entire picture.
- Representations found in densely connected layers contain no information about *where* objects are located in the input image. Densely connected layers eliminate the notion of space.
- Where object location matters, densely connected features are largely useless.

Separation of Insights, Convolutional Base

- 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.
- Object location is described by the convolutional feature maps.
- 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.

Immediate Objectives

- 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.

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 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 we stick into a densely connected classifier.

How to use imported convolutional base

At this point, there are two ways we could adopt:

1. Run the convolutional base over our 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 saw earlier.

   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. 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. Extend the model we have (`conv_base`) by adding `Dense` layers on top, and running the whole thing end to end on the input data. This allows us to use data augmentation, because every input image goes through the convolutional base every time it is seen by the model. For the same reason, this technique is far more expensive than the first.

Fast Feature Extraction without Data Augmentation

We define function `extract_features` which runs instances of `ImageDataGenerator` to read images from OS directory and feeds them to the `predict()` method of the `conv_base`. Result are 2 Numpy array containing features and their labels.

```
import os ; import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator
base_dir = 'cata-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, label
```

Fast feature extraction and Reshaping of arrays

- We extract features and labels from images contained in train, validate and test subdirectories:

```
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)

print(train_features.shape, train_labels.shape)
print(validation_features.shape, validation_labels.shape)
print(test_features.shape, test_labels.shape)

(2000, 4, 4, 512) (2000,)
(1000, 4, 4, 512) (1000,)
(1000, 4, 4, 512) (1000,)
```

- The extracted features are currently of shape `(samples, 4, 4, 512)`. This is coming from the dimensions of the last layer in the `conv_base`:

```
block5_pool (MaxPooling2D) (None, 4, 4, 512) 0
```

- To feed those features 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))
```

Densely Connected Classifier

- At this point, we define your densely connected classifier. The classifier will be trained on the data and labels that we just recorded.
- We will use a `Dropout` layer for regularization.

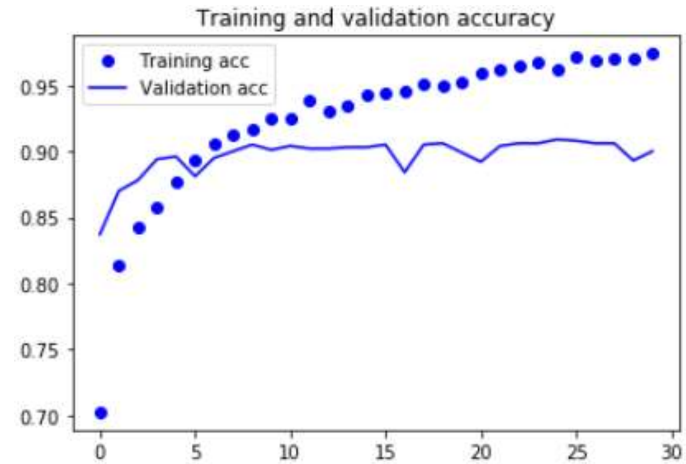
```
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import optimizers

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(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))
```

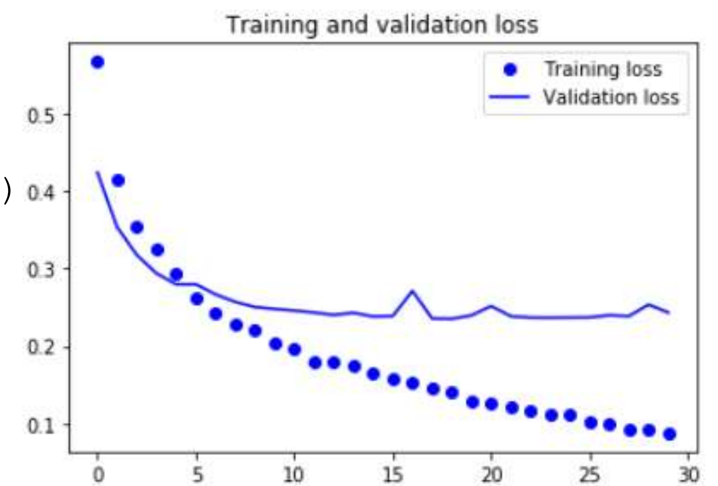
- Training is very fast, because you only have to deal with two Dense layers—an epoch takes less than one second even on CPU.

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%. This is better than we achieved 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 allows us 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 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 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

- We 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
```

- In Keras, we freeze a layer or a whole model by setting its `trainable` attribute to `False`.

```
>>> 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
```

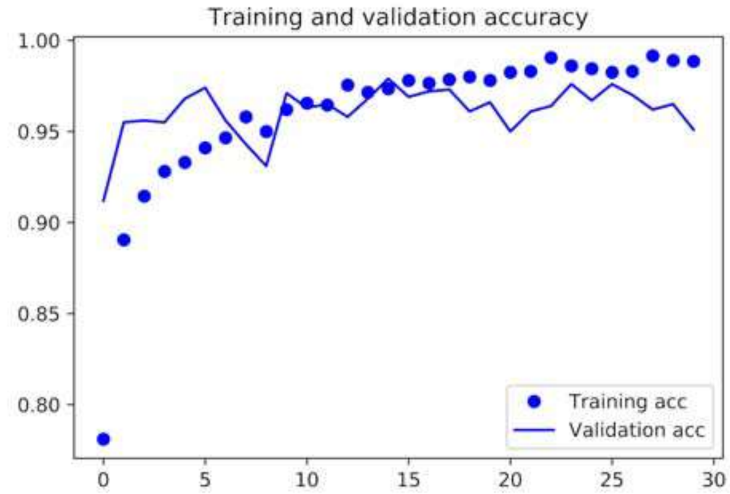
- With this setup, only the weights from the two Dense layers that you added will be trained. That's a total of four weight tensors: two per layer (the main weight matrix and the bias vector).
- Note that in order for these changes to take effect, you must first compile the model. If you ever modify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

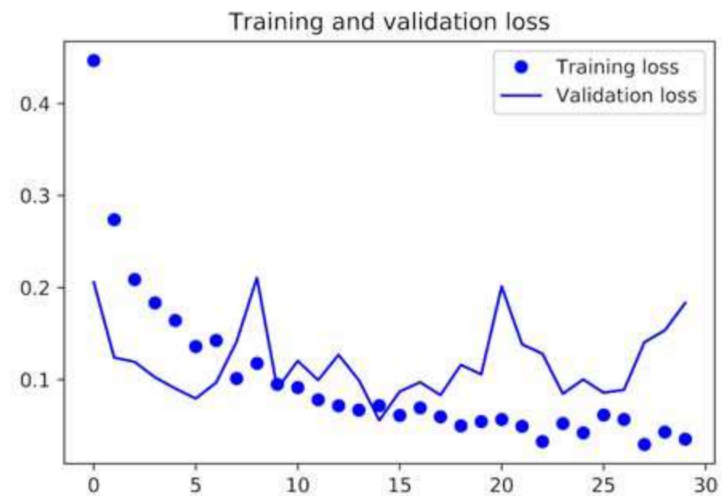
Training the model with a frozen convolutional base

```
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(
      train_generator,  steps_per_epoch=100,  epochs=5,
      validation_data=validation_generator,
      validation_steps=50,      verbose=2)
```

Training and Validation Accuracy and Loss

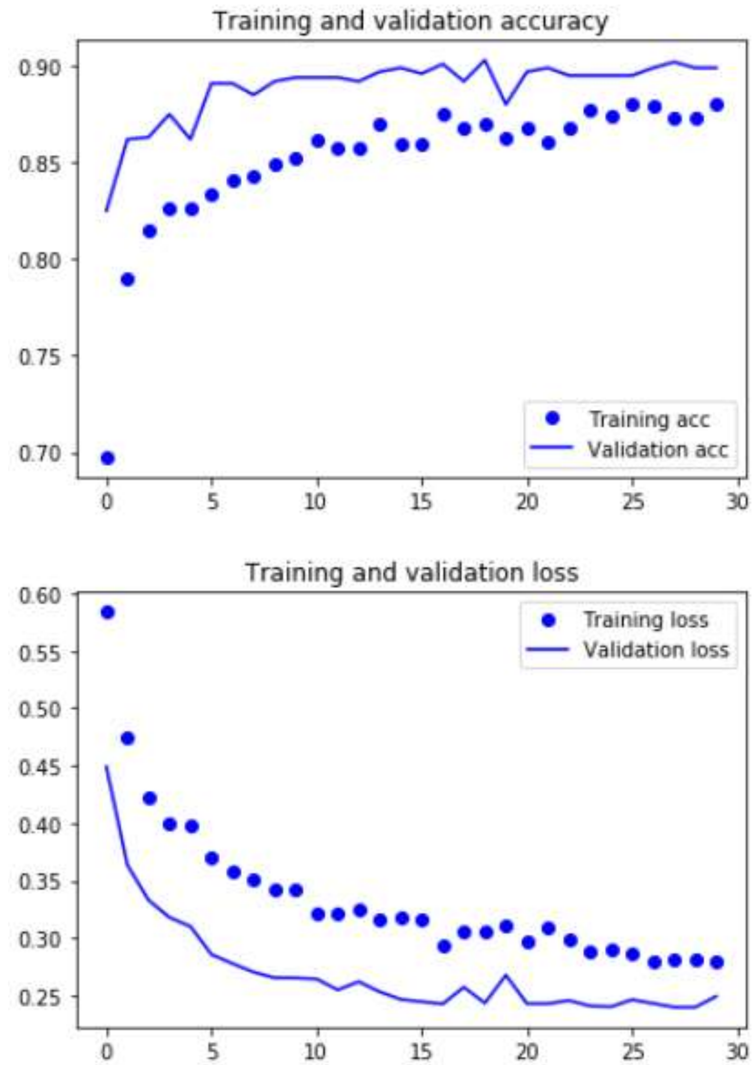
- Training and validation accuracy for feature extraction with data augmentation





Another run

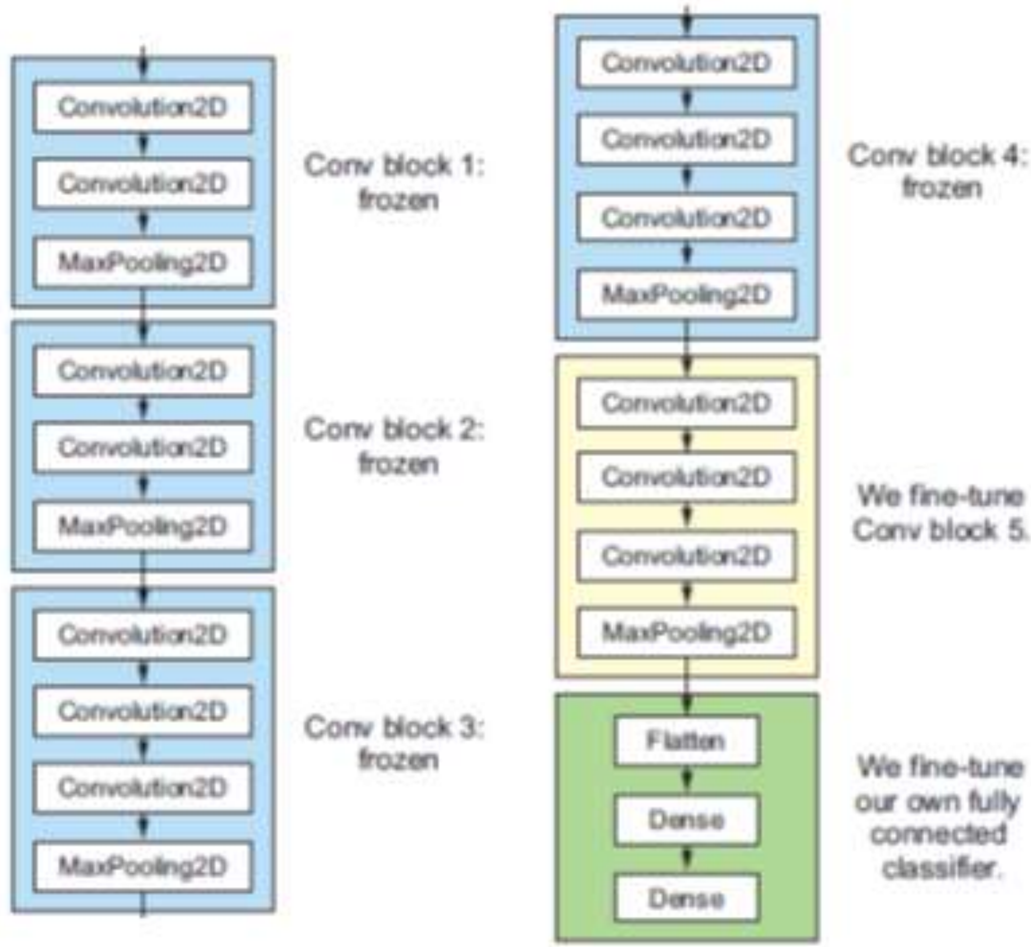
- Another run gave slightly different results.
- Curiously, the validation accuracy is higher than the training accuracy.



Fine Tuning

Fine Tuning

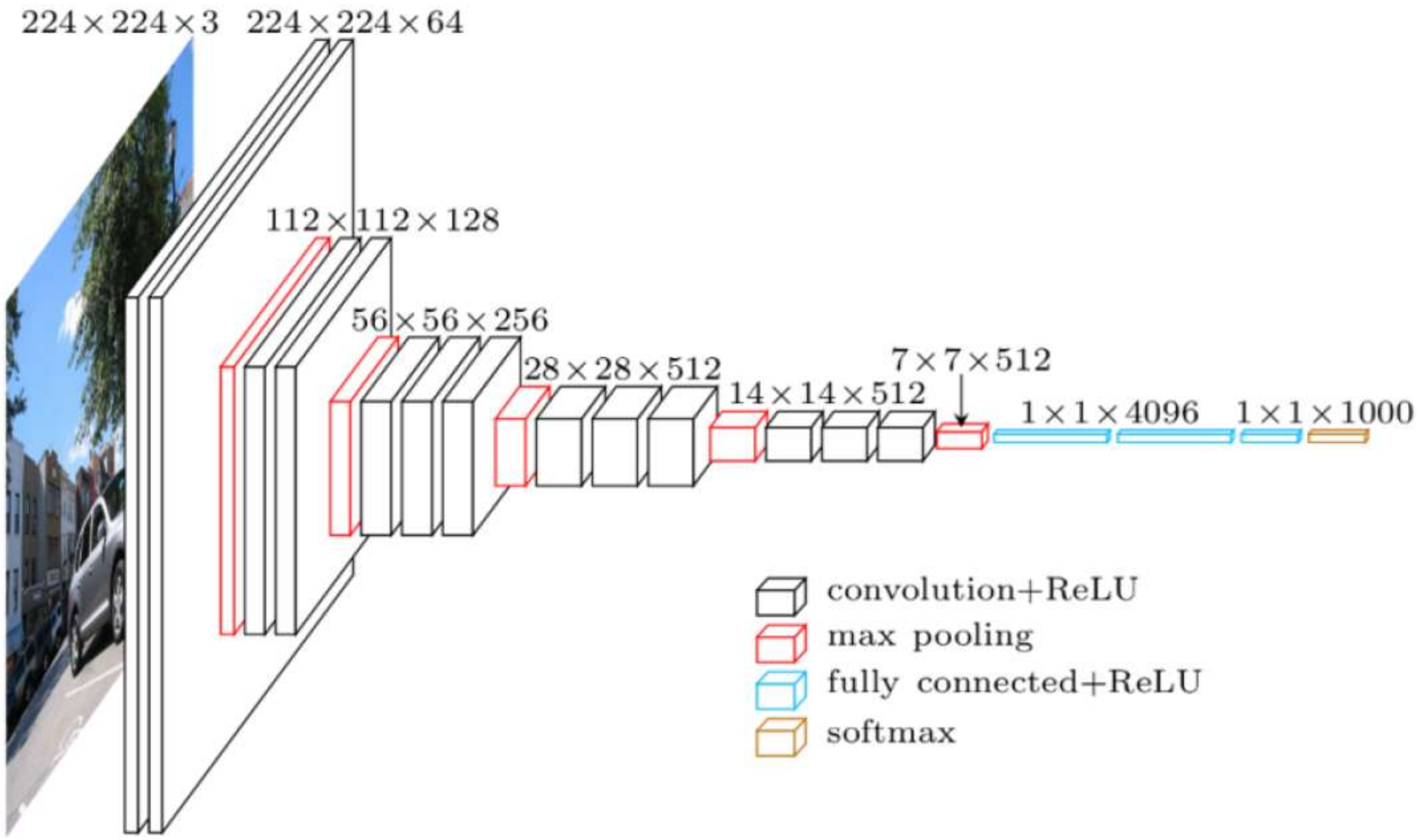
- Another widely used technique for model reuse is *fine-tuning*.
- Fine-tuning consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model (in this case, the fully connected classifier) and these top layers.



- 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.
- Above layers belong to VGG16 network. `Conv block 4` receives input from `Conv block 3`

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

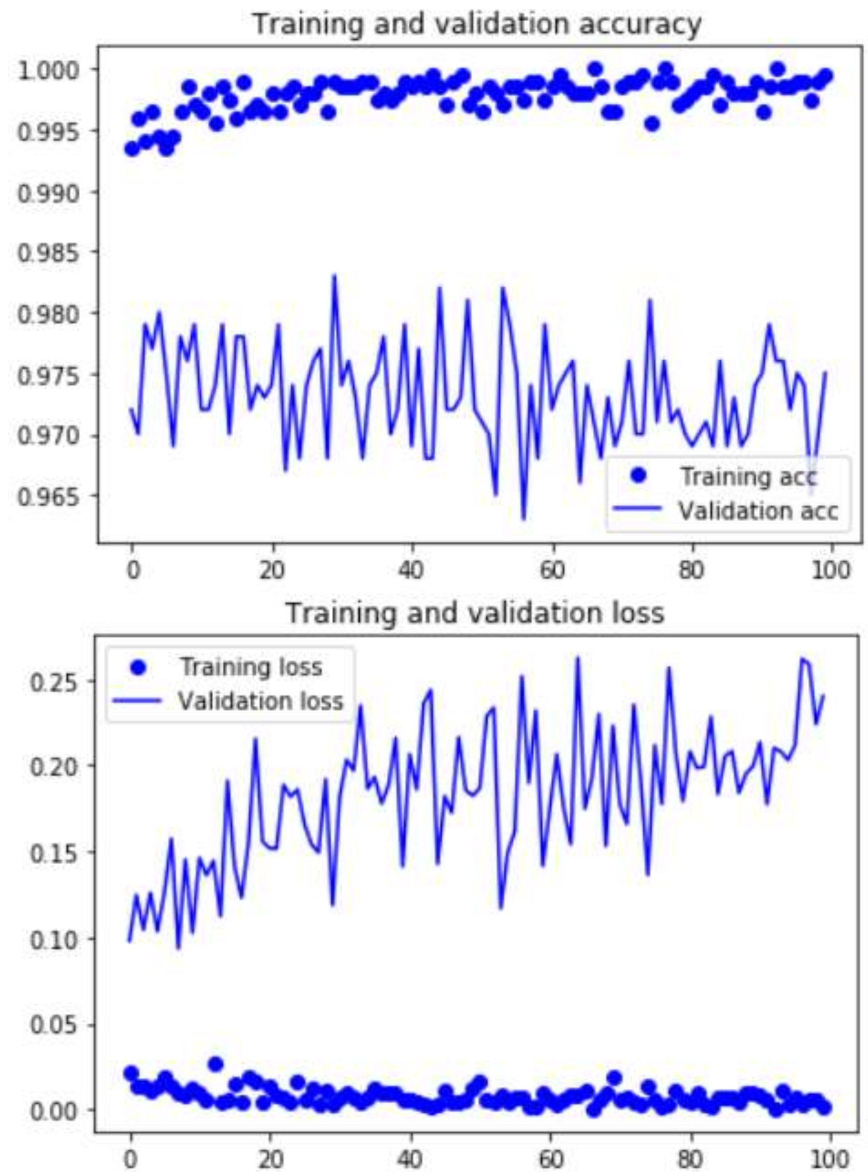
```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

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`
```

Keras Callbacks

Mechanisms of Control, Callbacks

- So Far we would launch `model.fit()` method and hope for the best. If everything went well, we would assess how much epochs we wasted on needless overfitting.
- The `tf.Keras callbacks API` will helps transform our `model.fit()` call into a smart, autonomous code that can self-introspect and dynamically take action.
- A `callback` is an object of a subclass of the class `keras.callbacks.Callback`. That object is passed in the call to `fit()`. Specific methods are called at various points during training.
- Object `callback` has access to the state of the `model` and its performance. It can take action: interrupt training, save a model, load a different weight set, or alter the state of the model.
- Some examples of the ways we can use callbacks:
  - *`Model checkpointing`*— Saving the current state of the model at different points during training.
  - *`Early stopping`*— Interrupting training when the validation loss is no longer improving (and of course, saving the best model obtained during training).
  - *`Dynamically adjusting the value of certain parameters during training`*— Such as the learning rate of the optimizer.
  - *`Logging training and validation metrics during training`*, or visualizing the representations learned by the model as they're updated.
- The Callback API includes a number of built-in callbacks. Some of them are:

```
keras.callbacks.ModelCheckpoint
keras.callbacks.EarlyStopping
keras.callbacks.LearningRateScheduler
keras.callbacks.ReduceLROnPlateau
keras.callbacks.CSVLogger
```

`EarlyStopping & ModelCheckpoint` Callbacks

- When you're training a model, there are many things you can't predict from the start. In particular, you can't tell how many epochs will be needed to get to an optimal validation loss.
- Our examples so far have adopted the strategy of training or enough epochs that you begin overfitting, using the first run to figure out the proper number of epochs to train for, and then finally launching a new training run from scratch using this optimal number. Of course, this approach is wasteful.
- A much better way to handle this is to stop training when you measure that the validation loss is no longer improving. This can be achieved using the `EarlyStopping` callback.
- The `EarlyStopping` callback interrupts training once a target metric being monitored has stopped improving for a fixed number of epochs. For instance, this callback allows you to interrupt training as soon as you start overfitting, thus avoiding having to retrain your model for a smaller number of epochs.
- The `EarlyStopping` callback is typically used in combination with `ModelCheckpoint`, which lets you continually save the model during training (and, optionally, save only the current best model so far: the version of the model that achieved the best performance at the end of an epoch):

Define callbacks

- We define a callback list with one or more callbacks

```
callbacks_list = [                          # 1
    keras.callbacks.EarlyStopping(          # 2
        monitor='acc',                      # 3
        patience=2,                         # 4
    ),
    keras.callbacks.ModelCheckpoint(        # 5
        filepath='my_checkpoint_path',      # 6
        monitor='val_loss',                 # 7
        save_best_only=True,                # 7
    )
]
```

1. Callbacks are passed to the model via the callbacks argument in `fit()`, which takes a list of callbacks. You can pass any number of callbacks.
2. Interrupts training when improvement stops
3. Monitors the model's validation `accuracy`
4. Interrupts training when `accuracy` has stopped improving for more than 2 epoch (that is, three epochs)
5. Saves the current weights after every epoch
6. Path to the destination model file
7. These two arguments mean you won't overwrite the model file unless `val_loss` has improved, which allows you to keep the best model seen during training.

Injecting `callback_list` into `model.fit()`

- Our model for a simple CNN was compiled as:

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

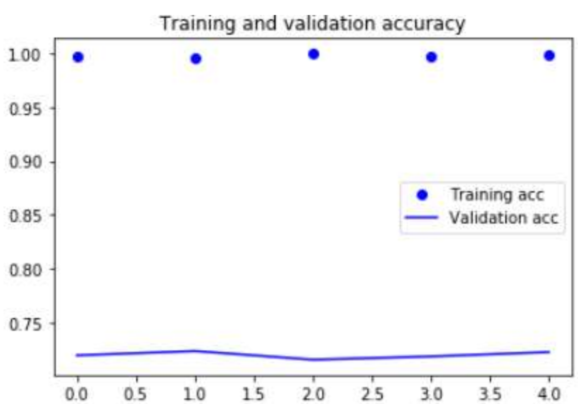
- Note that the `matrics` in the definition of callback has to match the `metrics` indicated in the `compile()` step
- Next, we modify the call to `model.fit()`, by adding the argument `callbacks=callback_list :`

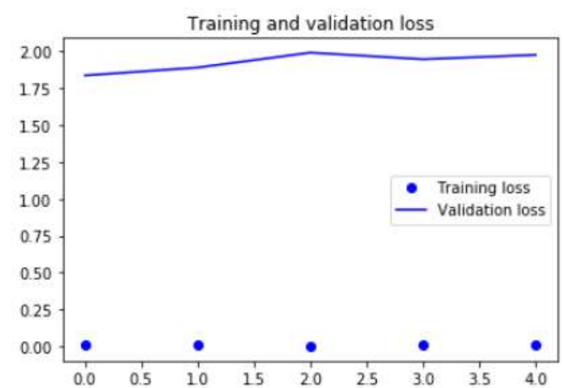
```
history = model.fit(
train_generator,
      steps_per_epoch=100,
      epochs=30,
      callbacks=callbacks_list,
      validation_data=validation_generator,
      validation_steps=50)
```

Run

```
Epoch 1/30
100/100 [==============================] - ETA: 0s - loss: 0.0146 - acc:
0.9965INFO:tensorflow:Assets written to: my_checkpoint_path2\assets
100/100 [==============================] - 15s 147ms/step - loss: 0.0146 - acc: 0.9965
- val_loss: 1.8375 - val_acc: 0.7200
Epoch 2/30
100/100 [==============================] - 13s 132ms/step - loss: 0.0144 - acc: 0.9955
- val_loss: 1.8911 - val_acc: 0.7240
Epoch 3/30
100/100 [==============================] - 13s 129ms/step - loss: 0.0016 - acc: 0.9995
- val_loss: 1.9921 - val_acc: 0.7160
Epoch 4/30
100/100 [==============================] - 13s 130ms/step - loss: 0.0119 - acc: 0.9965
- val_loss: 1.9476 - val_acc: 0.7190
Epoch 5/30
100/100 [==============================] - 13s 130ms/step - loss: 0.0071 - acc: 0.9985
- val_loss: 1.9770 - val_acc: 0.7230
```

- Our model stopped after 5 epochs. Plot the loss and accuracy during training:





Monitoring and visualization with `TensorBoard`

- To do good research or develop good models, we need rich, frequent, feedback about what is going on inside our models during our experiments.
- Making progress is an iterative process: we start with an idea and express it as an experiment, attempting to validate or invalidate our idea. We run this experiment and process the information it generates. Keras and fast GPUs help us go from idea to experiment in the least possible time.
- `TensorBoard` helps with processing and understanding the experimental results?
- TensorBoard is a browser-based application that we can run locally.
- With TensorBoard, we can:
  - Visually monitor metrics during training
  - Visualize your model architecture
  - Visualize histograms of activations and gradients
  - Explore embeddings in 3D
- If we monitoring more information than just the model's final loss, we can develop a clearer vision of what the model does and doesn't do, and can make progress more quickly.
- TensorBoard installs with Python pip as: `$ pip install tensorboard`
- The easiest way to use TensorBoard with a Keras `model` and the `fit()` method is the `keras.callbacks.TensorBoard` callback.

Instantiate `TensorBoard` Callback object

- TensorBoaRD callback object must have one argument, the directory where the information about the state of model training will be written. For example:

```
tensorboard = keras.callbacks.TensorBoard(
    log_dir='logs_directory',   # can use the full path to our directory
)
```

- This callback is passed to `model.fit()` like the previous one, or could be added to the list of existing callbacks:

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

- This starts the training process:

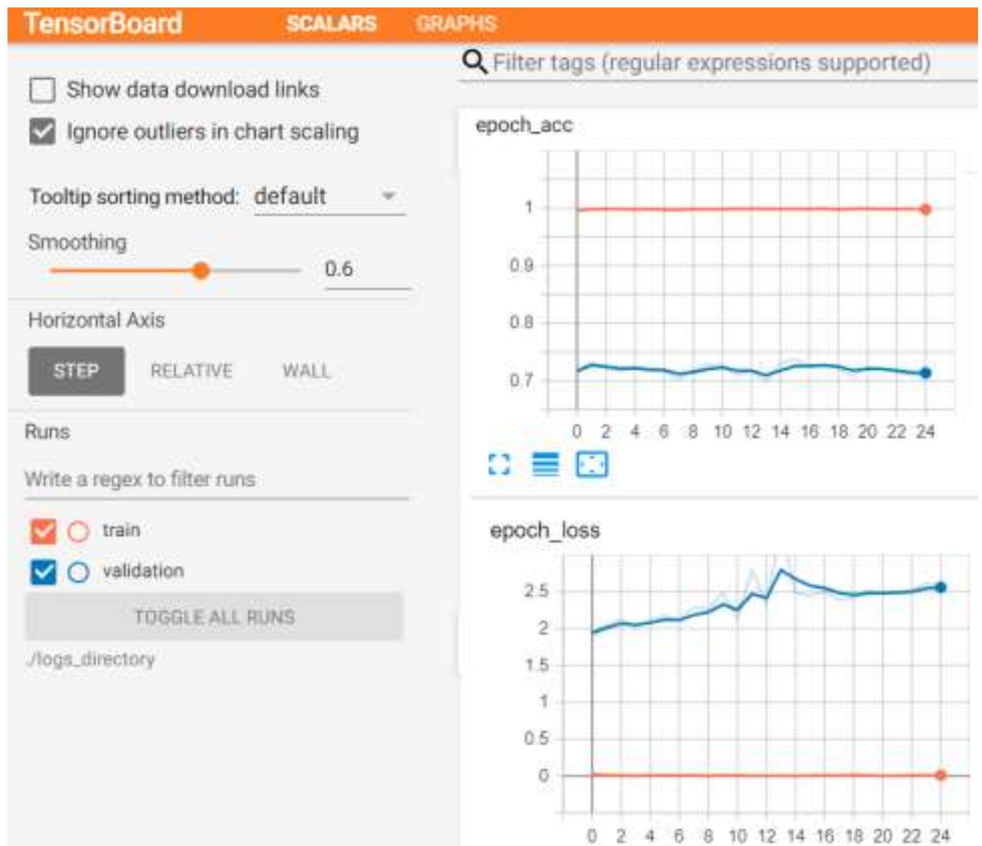
```
Epoch 1/30
  2/100 [..............................] - ETA: 28s - loss: 0.2869 - acc: 0.9500
WARNING:tensorflow:Method (on_train_batch_end) is slow compared to the batch
update (0.258329). Check your callbacks.
100/100 [==============================] - 15s 146ms/step - loss: 0.0225 - acc:
0.9955 - val_loss: 1.9400 - val_acc: 0.7170
Epoch 2/30
100/100 [==============================] - 14s 136ms/step - loss: 0.0050 - acc:
0.9990 - val_loss: 2.0542 - val_acc: 0.7330
Epoch 3/30
100/100 [==============================] - 13s 134ms/step - loss: 0.0025 - acc:
0.9990 - val_loss: 2.1292 - val_acc: 0.722
```

On the OS command prompt

- To start TensorBoard server, on the operating system prompt, type:

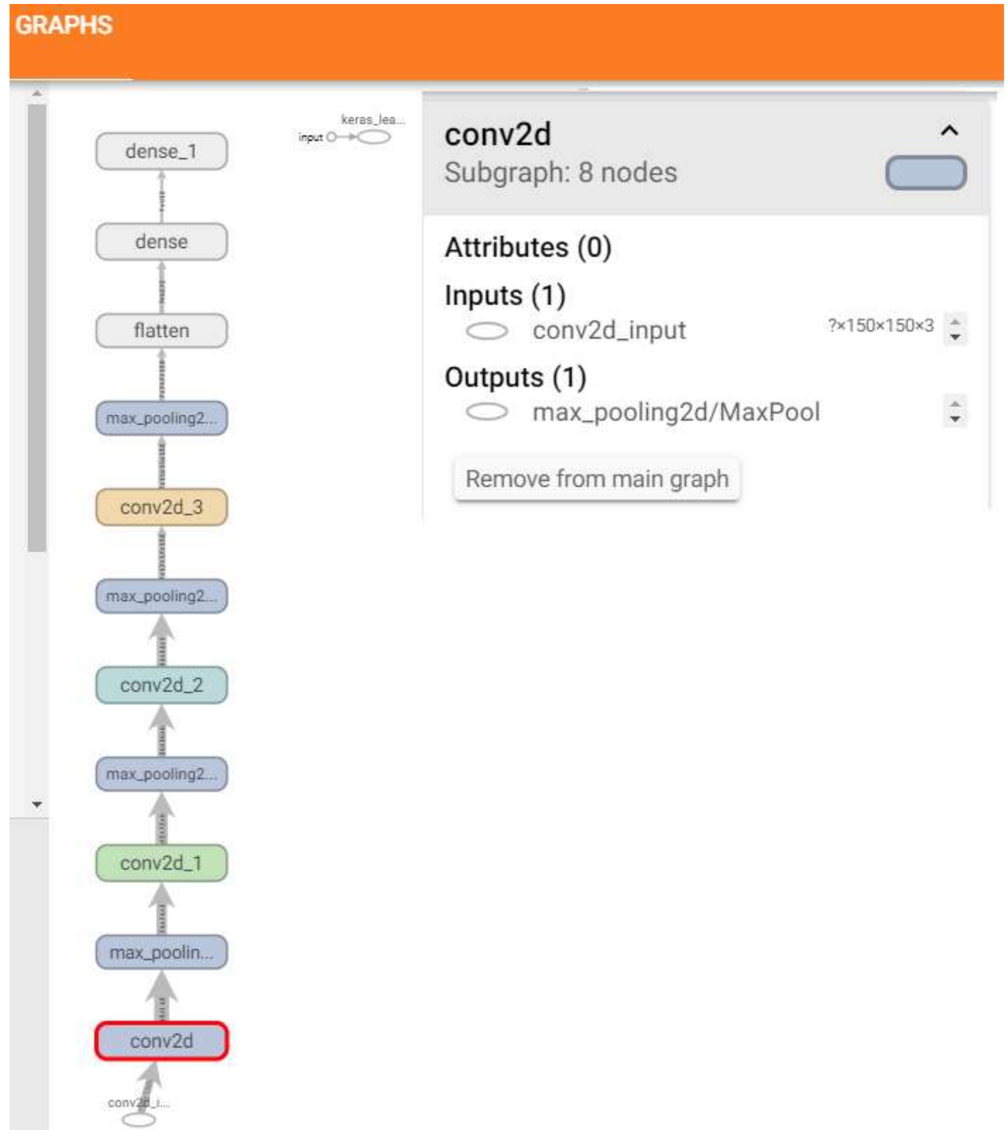
```
$ tensorboard --logdir ./logs_directory
2020-10-09 13:39:29.351923: I
tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened
dynamic library cudart64_101.dll
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --
bind_all
TensorBoard 2.2.2 at http://localhost:6006/ (Press CTRL+C to quit)
```

- TensorBoard server has opened the application on `http://localhost:6006.`
- Open the browser at that port. You will see a continuous display of accuracies and losses as epochs progress.



Network Diagram

- Previous display shows evolution of Scalar variables. If we click on Graph, we will see the graph of our network.
- Selecting any one layer will pop-up a display with basic characteristics of that 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.