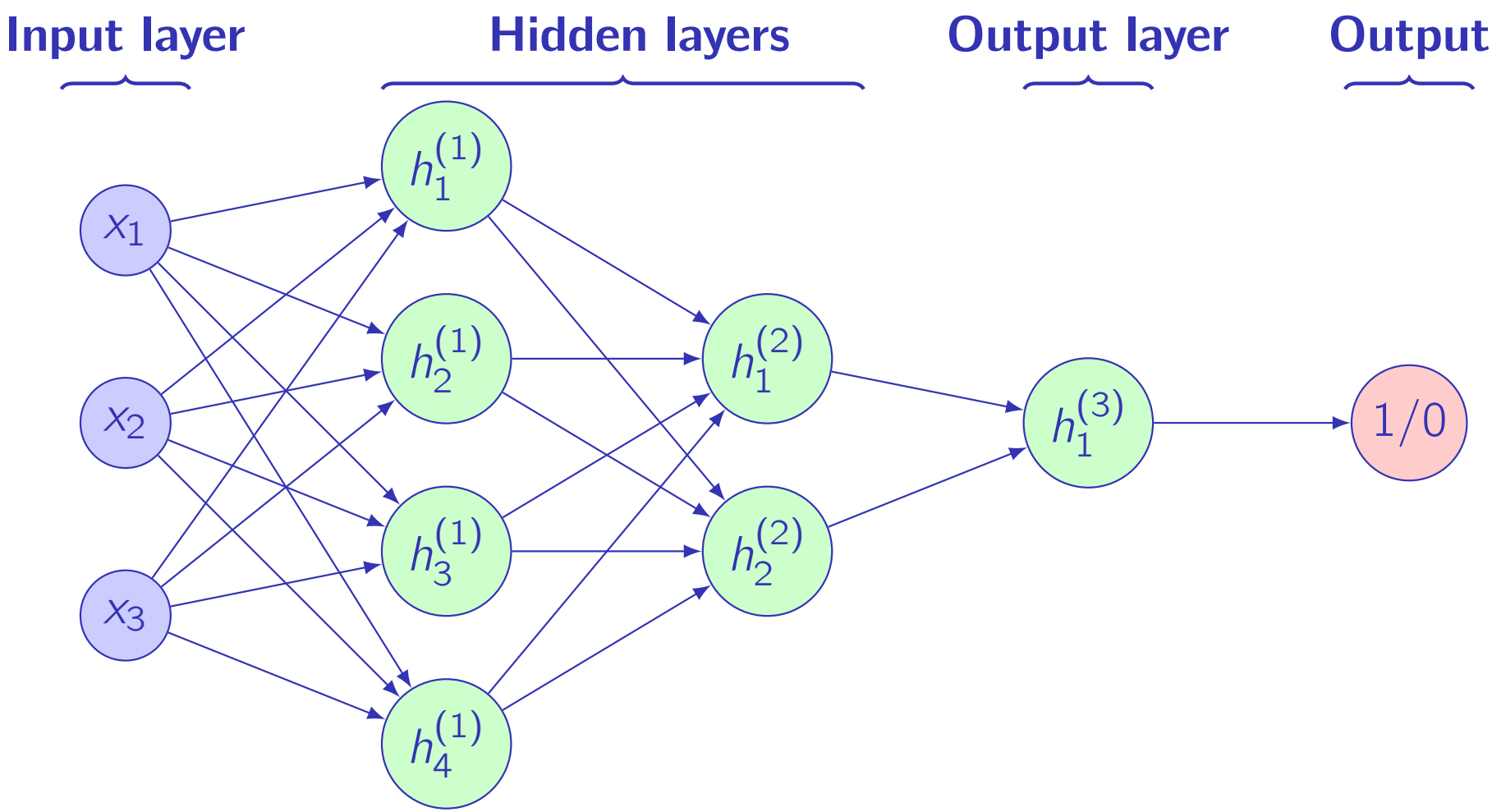
EDAN95
Applied Machine Learning
Lecture 8: Convolutional Networks

Pierre Nugues

Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/

November 24, 2021

Classifying Images with a Multilayer Perceptron



Each input node corresponds to a pixel of the input image.

Code Example

Experiment: Jupyter Notebook from Chollet's book: `chapter02_mathematical-building-blocks.ipynb`, Sect. A first look at a neural network

- The original images are 28×28 matrices with 256 gray levels
- The rows are simply scaled to 1.0 and appended into long vectors of 784 dimensions;
- Although efficient on digits, multilayer perceptron or logistic regression cannot capture optimally patterns in images;
- It is now replaced by networks that embed a pattern-extraction mechanism.

The Origins: The Convolution

The product of a function and a moving window, called the kernel.
Mathematical definition:

$$(f * g)(x) = \int_{-\infty}^{+\infty} f(x-t)g(t)\,\mathrm{d}t,$$

where f is the function and g, the convolution kernel
Notice that one of the function, here f, is reversed and shifted to guarantee commutativity
In the discrete case, we have:

$$(f * g)(i) = \sum_{j=-\infty}^{\infty} f(i-j)g(j)$$

It can be extended to two dimensions.

Convolution in Image Processing

Convolution is used extensively in pattern recognition to implement spatial filtering.
In image processing, f is an image and g a small window, most frequently its dimensions are: (3, 3) or (5, 5).
For a kernel of dimensions (M, N), normally odd numbers, we have:

$$(f * g)(x, y) = \sum_{i=-M/2}^{M/2} \sum_{j=-N/2}^{N/2} f(x-i, y-j) g(i, j),$$

where g is centered at 0.

Example of a Convolution

A blurring kernel, normally normalized by its sum (1/9):

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \mathbf{219} & 253 & 247 \\ 0 & 0 & 190 & 0 \\ 0 & 0 & \mathbf{0} & 93 \\ 0 & 0 & 221 & 253 \\ 136 & 212 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \mathbf{662} & & 0 \\ 0 & & & 0 \\ 0 & & \mathbf{757} & 0 \\ 0 & & & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$219 + 253 + 190 = 662$

Borders can either be padded or ignored. In the latter case, the kernel must always fit in the image and the output image has a reduced size. (Complete the matrix...)

Spatial Filters

The kernels enable us to create filters, for instance to smooth or sharpen the image.
The Sobel operator is a popular edge detector. It corresponds to the gradient norm of the input image and sharpens the edges.
We compute the x and y derivatives using two kernels:

$$\mathbf{G}_x = \mathbf{I} * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}; \mathbf{G}_y = \mathbf{I} * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$

We can also compute the gradient angle:

$$\tan\theta = \frac{\mathbf{G}_y}{\mathbf{G}_x}; \theta = \arctan\frac{\mathbf{G}_y}{\mathbf{G}_x}$$

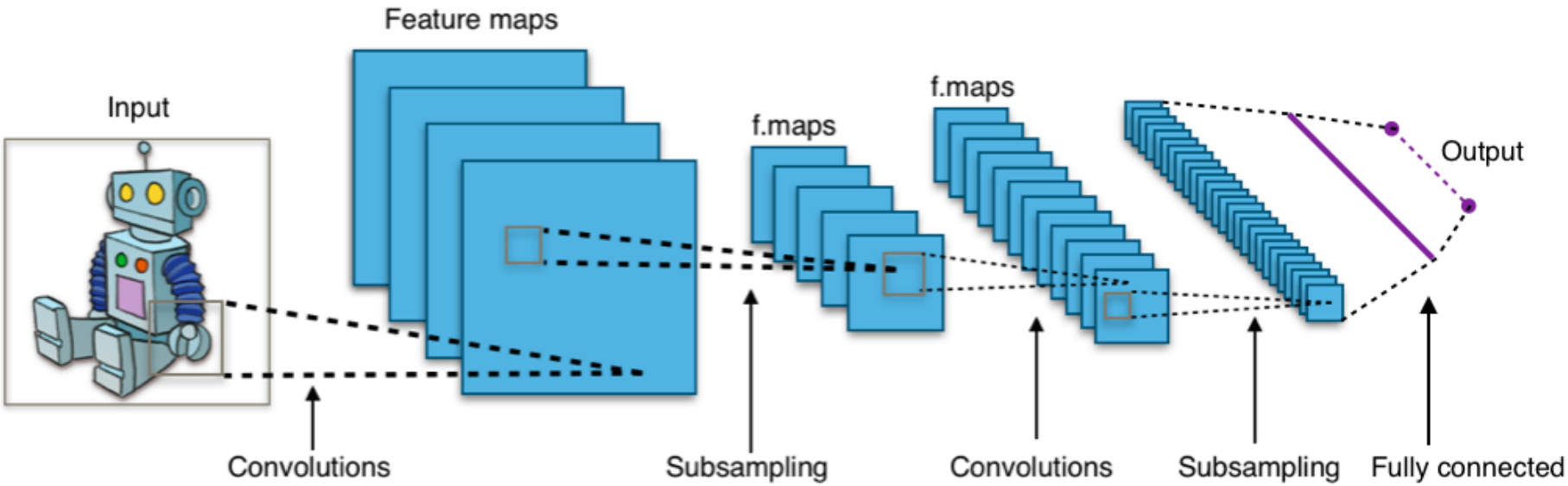
Code Example

Experiment: Jupyter Notebook: `2.1-convolution.ipynb`

Architecture of a Convolutional Neural Network (CNN)

A pipeline of convolution and subsampling operations:

1. In the figure, the network will learn four kernels in the first layer;
2. in the second layer, it will subsample the images, keep one pixel every four pixels, for instance.



Credits: Wikipedia

Subsampling

Subsampling is generally carried out using the max-pooling operation:

1. We partition the image into squared tiles of size (2, 2) or (3, 3);
2. We reduce each tile to one value: the max value in the tile.

For example, using a tile of size (2, 2), we have:

0	0	0	0
0	**219**	**253**	247
0	0	**190**	0
0	0	0	93
0	0	221	**253**
136	**212**	0	0

$\rightarrow$

219	253
0	190
212	253

We have reduced the image size from (6, 4) to (3, 2).
In addition, it makes the images invariant to small rotations and translations

A Small Convolutional Network

Encoding a CNN is straightforward in Keras:

1. We declare a sequential model
2. We add the layers

From Chollet, Listing 8.1, a small network with three convolutional layers and two subsampling layers:

```
from tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Understanding the Parameters

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling 2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 128)	73856

First layer: 32 kernels of size (3, 3) and an intercept:
$(3 \times 3 + 1) \times 32 = 320$.
The output consists of 32 images, where the size is reduced by two so that the kernel fits in the image
The downsampling reduces the images to (13, 13)
The third layer has $(3 \times 3 \times 32 + 1) \times 64 = 18496$ parameters

Adding a Classifier

The output the the convolutions consists of 128 (3, 3) images. We need to flatten them.
The rest is just a classifier with 10 outputs:

```
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Code Example

Experiment: Jupyter Notebook: Chollet 8.1 `https://github.com/fchollet/deep-learning-with-python-notebooks`

A Larger Dataset: The cats and dogs from Kaggle

We will follow Chollet again and review how to classify images of cats and dogs drawn from a Kaggle competition.
We will examine three strategies:

1. Raw data set, but smaller than the original one
2. Dataset augmented with image distortions
3. Using a pretrained network

In the next lab, you will apply the same procedure to another dataset from Kaggle and categorize five types of flowers

First Experiment: Raw Dataset

To process a raw dataset, we create a pipeline of convolutional layers and of max-pooling (Chollet, 2021, Listing 8.7).

1. As for other networks, the activation is the relu function except the last layer, which uses a logistic function.
2. As a general rule, the output images, feature maps, are smaller, but their number increases.
3. As architecture, Chollet proposed 5 convolutional layers and 4 subsampling layers, and a final classifier on the flattened images.
4. As with all datasets, you must rescale your data

The Pipeline (Chollet, 2021, Listing 8.7)

Rescaling:

```
inputs = keras.Input(shape=(180, 180, 3))
x = layers.Rescaling(1./255)(inputs)
```

The neural net:

```
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Image Loading

Keras has a builtin module to read images from a folder and supply them to the network (Chollet, 2021, Listing 8.9). It is based on `tf.data.Dataset`, which does not load all the data in memory (as a stream).

```
from tensorflow.keras.utils import image_dataset_from_directory

train_dataset = image_dataset_from_directory(
    new_base_dir / "train",
    image_size=(180, 180),
    batch_size=32)
validation_dataset = image_dataset_from_directory(
    new_base_dir / "validation",
    image_size=(180, 180),
    batch_size=32)
test_dataset = image_dataset_from_directory(
    new_base_dir / "test",
    image_size=(180, 180),
    batch_size=32)
```

The `train_dataset` and `validation_dataset` loop endlessly.

Fitting Function

The training procedure:

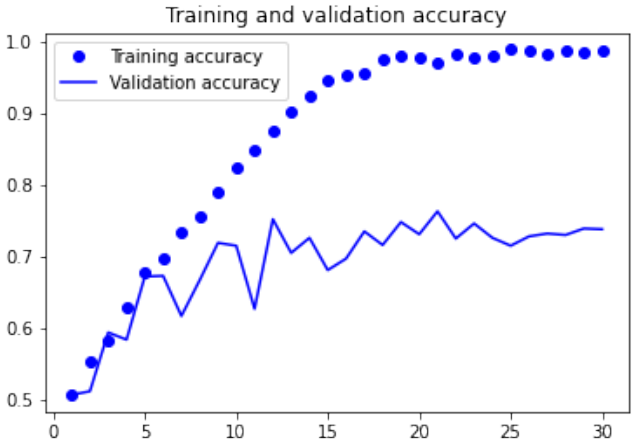
```
history = model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

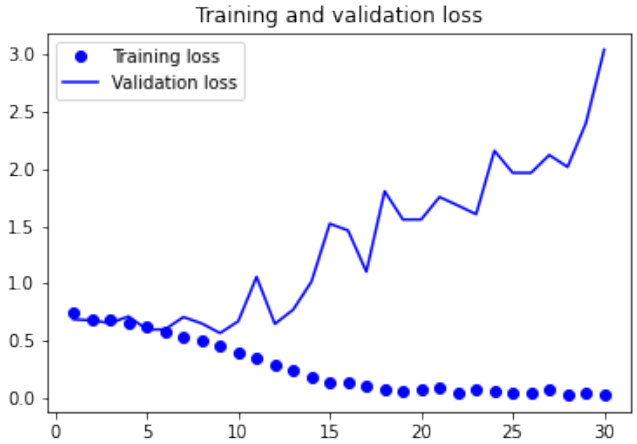
The callback saves the best model

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch.keras",
        save_best_only=True,
        monitor="val_loss")
]
```

Loss Curves and Evaluation

Jupyter Notebook: Chollet (2021), Listing 8.12






```
test_model = keras.models.load_model("convnet_from_scratch.ker
test_loss, test_acc = test_model.evaluate(test_dataset)
```

Code Example

Experiment: Jupyter Notebook: Chollet (2021), Listing 8.12, https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter08_intro-to-dl-for-computer-vision.ipynb
Accuracy: 71%

Second Experiment with Data Augmentation

Small datasets are prone to overfit: The model fits perfectly the training data, but cannot generalize to other kinds of data.
To avoid overfit, we can "augment" data in the training set with small transformations, for example a rotation.
To make it easier, Keras has a set of predefined random transformations:

```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
    ]
)
```

Architecture

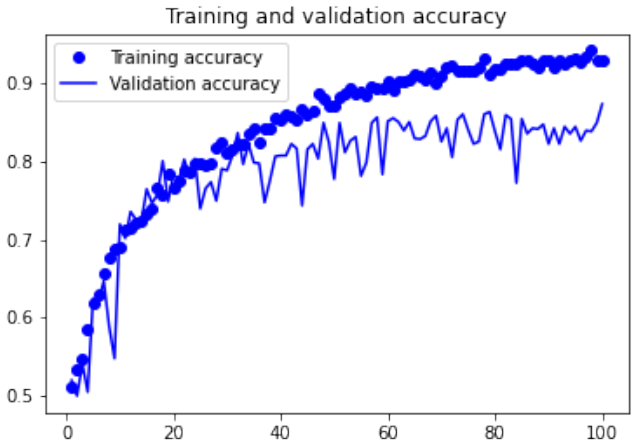
We add the augmentation as a layer before the network description:

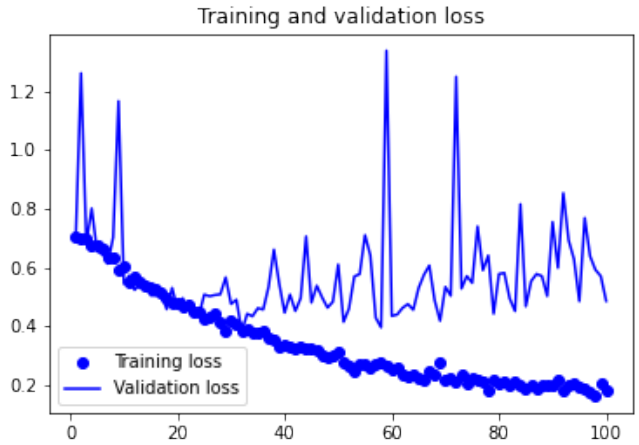
```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = layers.Rescaling(1./255)(x)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu"
...
```

The rest is the same

Code Example

Experiment: Jupyter Notebook: Chollet (2021), Listing 8.16, `https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter08_intro-to-dl-for-computer-vision.ipynb`





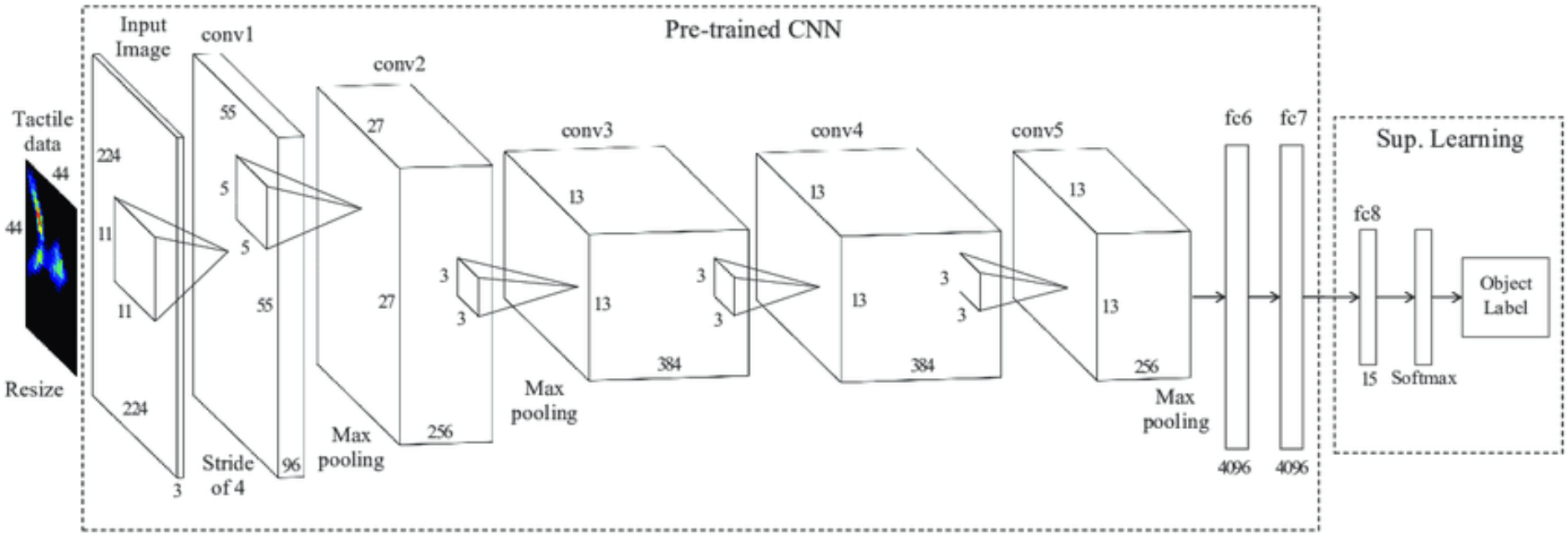
Accuracy: 81.9%

Third Experiment with Pretrained Convnet

In our datasets, we have a few thousands images and a handful of classes.
Some groups trained models on millions of images and thousands of classes, and they were kind enough to make their models available.
As for simpler CNNs, models learn more and more symbolic patterns as we proceed in the pipeline.
It can be difficult for an individual to train a large network on many images with many classes.
But we can reuse large existing models to extract the patterns.

Architecture of a Pretrained Convnet

This architecture consists of a pretrained convolutional base on which we plug a trainable classifier.



[1]

[1] Juan Manuel Gandarias, Jesus Gomez-de-Gabriel, Alfonso J. García-Cerezo, Enhancing Perception with Tactile Object Recognition in Adaptive Grippers for Human-Robot Interaction, Sensors 18(3):692, February 2018

Training Strategies with a Pretrained Convnet

We can either:

1. Use the pretrained convolutional base to create inputs. We use these inputs to train a new classifier;
2. Build a new network that consists of the frozen pretrained part as a base and extend it with a top that we train;
3. Train parts of the pipeline that consists of the pretrained convolutional base and a top carrying out the classification.

Extracting Features from the Convolutional Base

For the first option, we need to read the predictions from the base (Chollet, Listings 8.19 and 8.20):
We first retrieve a pretrained network from Keras:

```
conv_base = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False,
    input_shape=(180, 180, 3))
```

We then run a loop to extract the features for each image:

```
preprocessed_images = keras.applications.vgg16.preprocess_input(imag
features = conv_base.predict(preprocessed_images)
```

and we build a dataset from them.

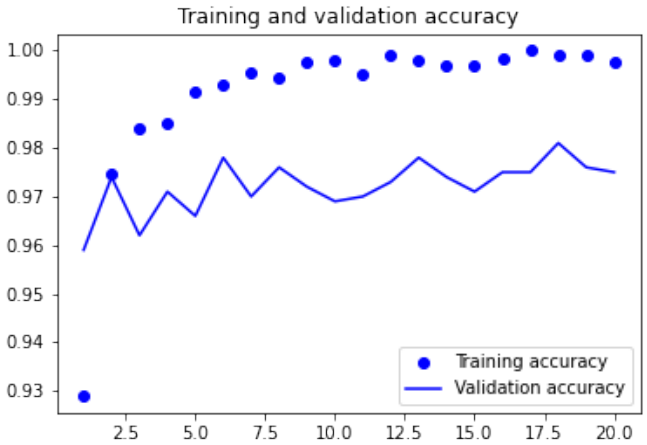
Classifying Features from the Convolutional Base

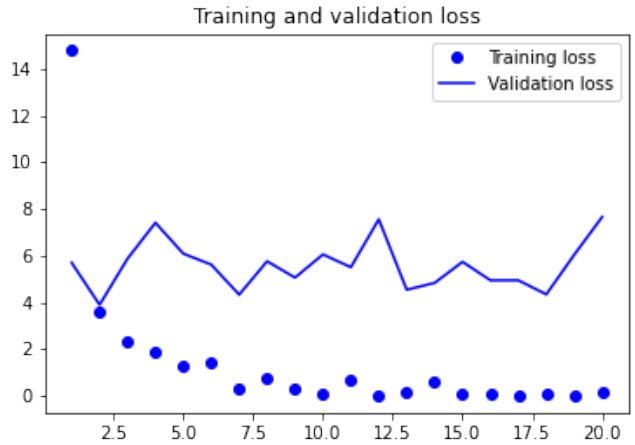
The feature dataset serves as input to a new classifier

```
inputs = keras.Input(shape=(5, 5, 512))
x = layers.Flatten()(inputs)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
```

Code Example

Experiment: Jupyter Notebook: Chollet (2021), Listings 8.17-8.22, https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter08_intro-to-dl-for-computer-vision.ipynb





Accuracy: 97.5%

Extending the Convolutional Base

The second option is very easy with Keras: We just extract the base (Chollet, 2021, Listings 8.23 and 8.25):

```
conv_base  = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False)
conv_base.trainable = False
```

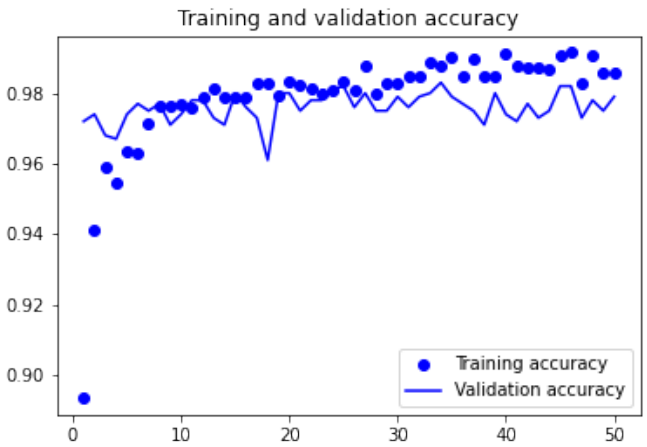
Extending the Convolutional Base (II)

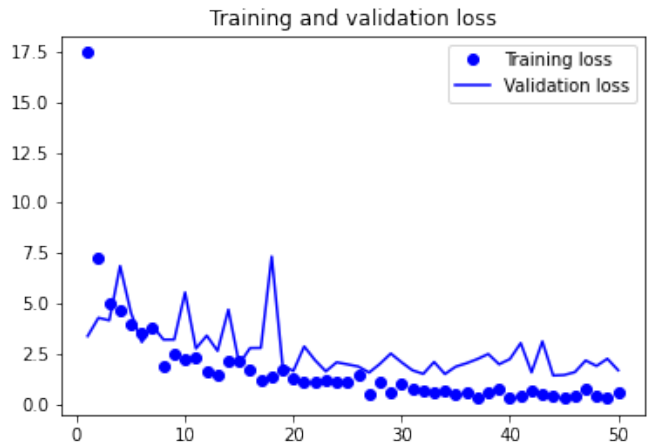
We just insert the base as first step of a dense network (Chollet, 2021, Listings 8.23 and 8.25). We also add an image augmentation:

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = keras.applications.vgg16.preprocess_input(x)
x = conv_base(x)
x = layers.Flatten()(x)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
```

Code Example

Experiment: Jupyter Notebook: Chollet (2021), Listings 8.13-8.26, https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter08_intro-to-dl-for-computer-vision.ipynb





Accuracy: 97.8%

Fine-tuning the Convolutional Base

We can finally tine-tune some layers of the convolutional base (Chollet, 2021, Listings 8.27):

```
conv_base.trainable = True
for layer in conv_base.layers[:-4]:
    layer.trainable = False
```

Here the last four layers.
The experiment is left as an exercise.
VGG16 is one of the available networks in Keras, for other pretrained convnets, see `https://keras.io/applications/#documentation-for-individual-models`.