Deep Computer Vision using Convolutional Neural Networks

Deep Learning

Stijn Lievens & Sabine Devreese

2023-2024

Deep Computer Vision Using CNNs

Easy Things Are/Were Hard

- Deep Blue beat Kasparov in 1996 at chess.
  - At that time, computers could *not* reliably recognize a puppy in a picture.
  - Nor could they recognize spoken words.
- However, we think of chess as more “difficult” than recognizing a puppy.

Emergence of CNNs

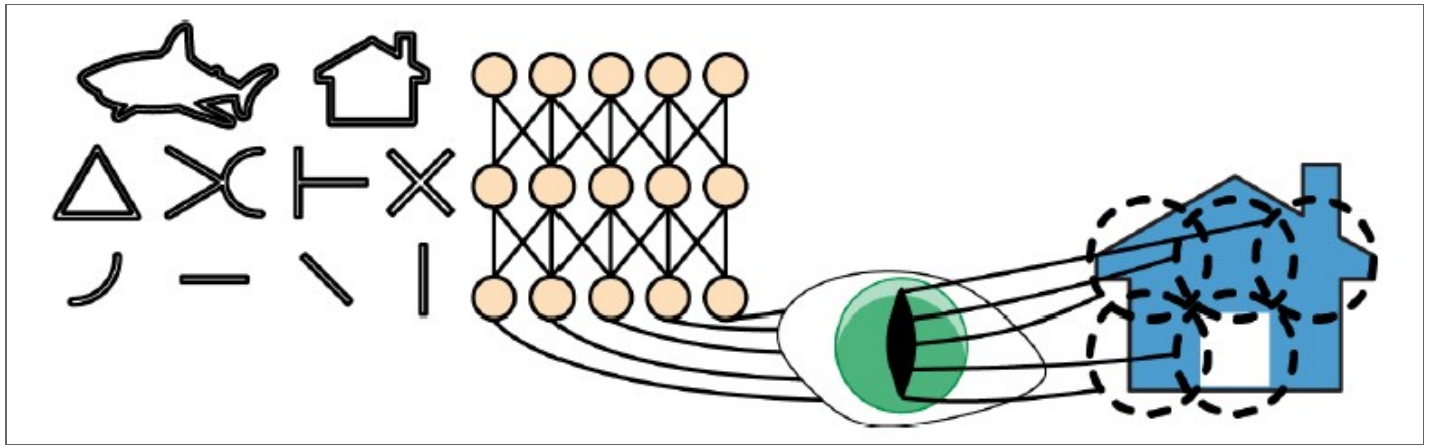
- **Convolutional Neural Networks** (**CNN**s) emerged from the study of the brain's visual cortex.
- CNNs can now outperform humans on some complex visual tasks. Thanks to:
  - the increase in computing power
  - the availability of large training datasets
  - better training techniques (as presented in Chapter 11)

14.1 The Architecture of the Visual Cortex

The Visual Cortex

- It has been shown that many neurons in the visual cortex have a small *local receptive field*.
  - They react only to visual stimuli located in a limited region of the visual field.
- The receptive fields of different neurons may overlap.
  - Together, they tile the whole visual field.

Structure of the Visual Cortex



Structure of the visual cortex

The Visual Cortex

- Neurons with the same receptive field may respond to different stimuli.
  - Some may respond to horizontal lines, others to vertical lines, etc.
- Other neurons seem to have larger receptive fields.
  - Idea: the higher-level neurons are based on the outputs of neighboring lower-level neurons.

14.2 Convolutional Layers

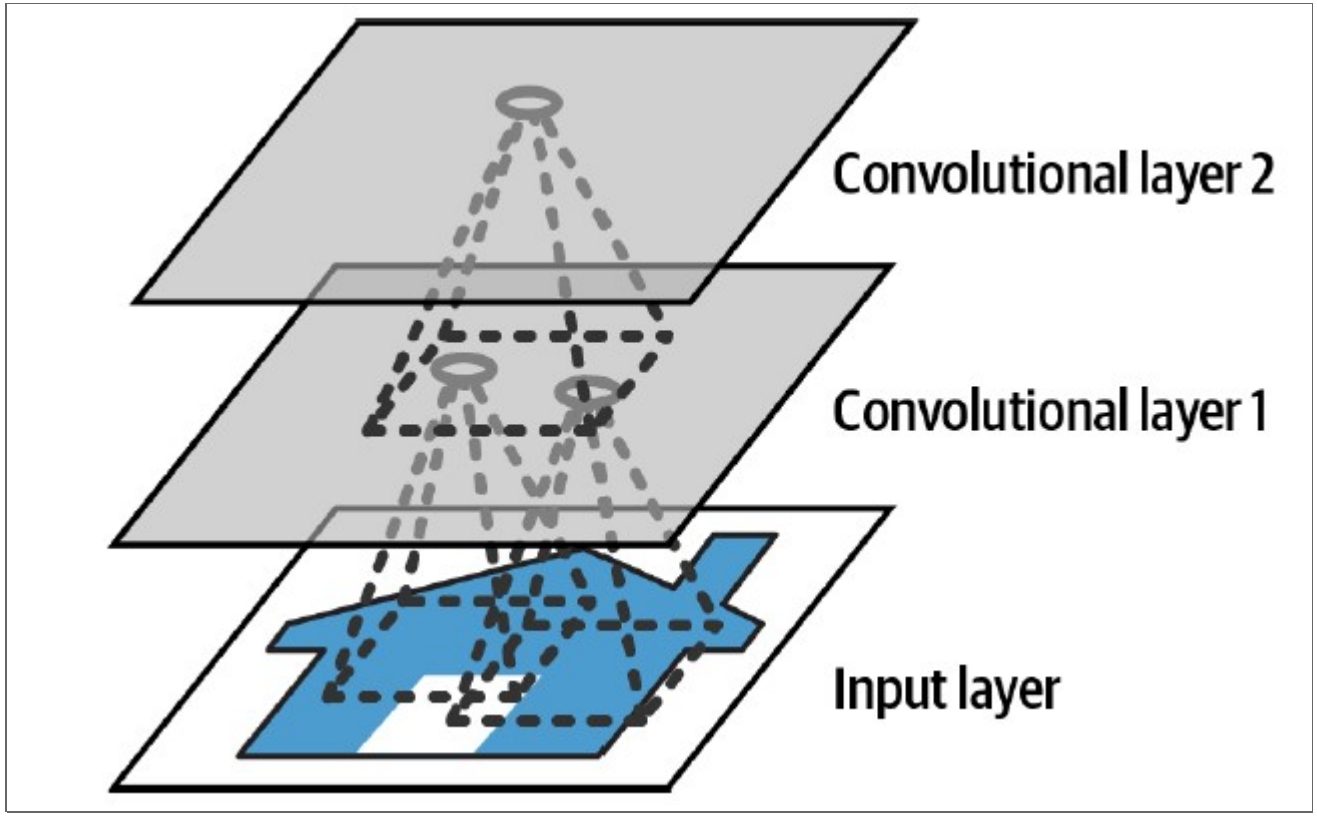
Convolutional Layers

- The most important building block of a CNN is the **convolutional layer**.
  - Neurons in the first convolutional layer are only connected to the neurons in their receptive fields (which is only a small rectangle of the complete image).
  - Neurons in the second convolutional layer are only connected to neurons located within a small rectangle in the first layer.
  - Etc.

Convolutional Layers

- The architecture of a CNN allows (the network) to concentrate on low-level features in the first layer, then assemble them into higher-level features in the next layer, etc.
- This hierarchical structure is common in real-world images.
  - E.g., a low-level feature may be a line, a higher-level feature may be a corner, etc.
  - This is why CNNs work so well for image recognition.

Convolutional Layers

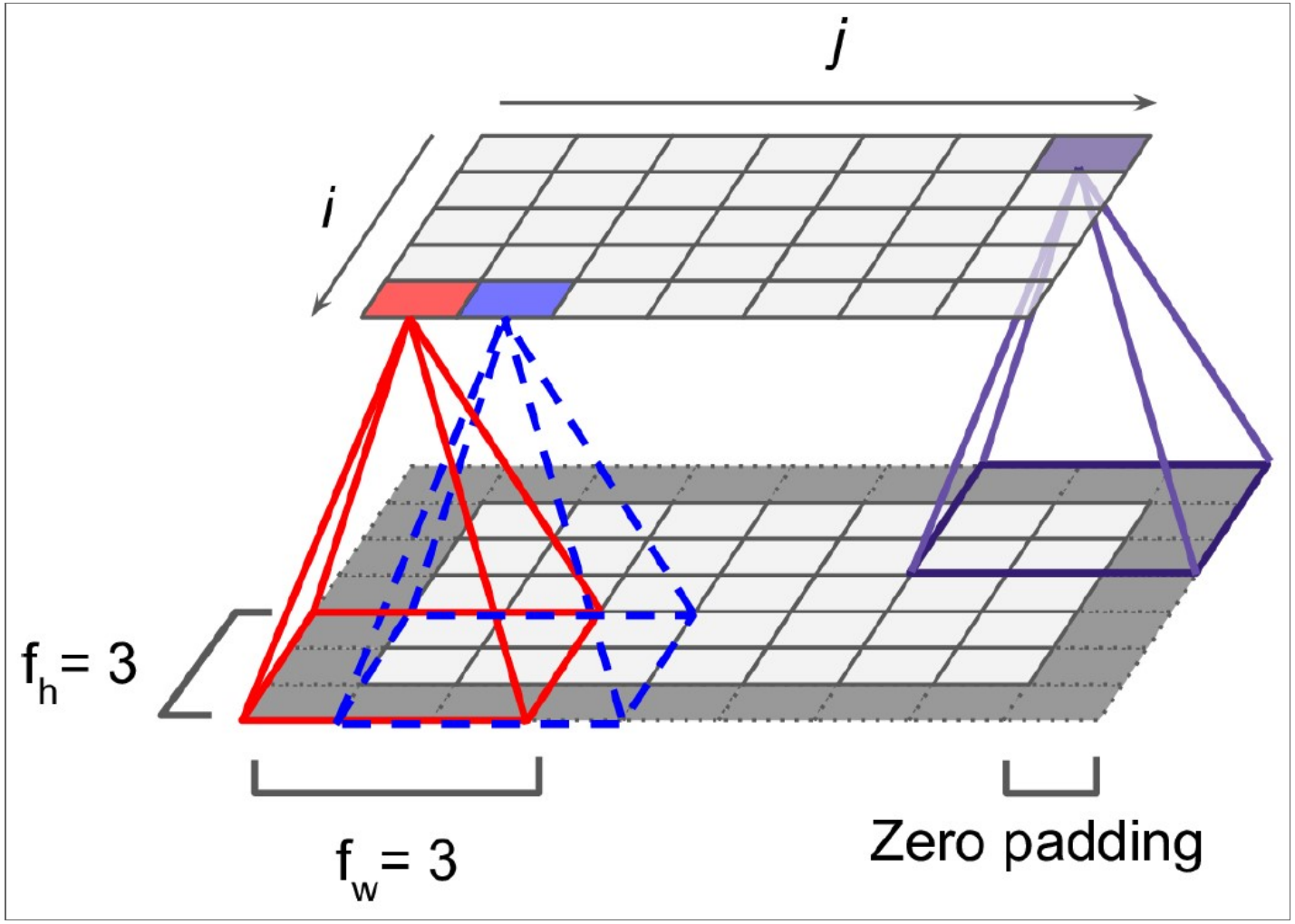


CNN layers with rectangular local receptive fields

Convolutional Layers

- In a CNN, each layer is represented in *2 (spatial) dimensions* (instead of one long row of neurons as in a regular MLP).
- A neuron at position (i, j) is connected to neurons in the previous layer within a small rectangular receptive field, located at positions (i', j') with $i \leq i' \leq i + f_h - 1$ and $j \leq j' \leq j + f_w - 1$. Here f_h and f_w are the height and width of the receptive field.
- We need **zero padding** to ensure that the output has the same size as the input.

Receptive Fields and Zero Padding

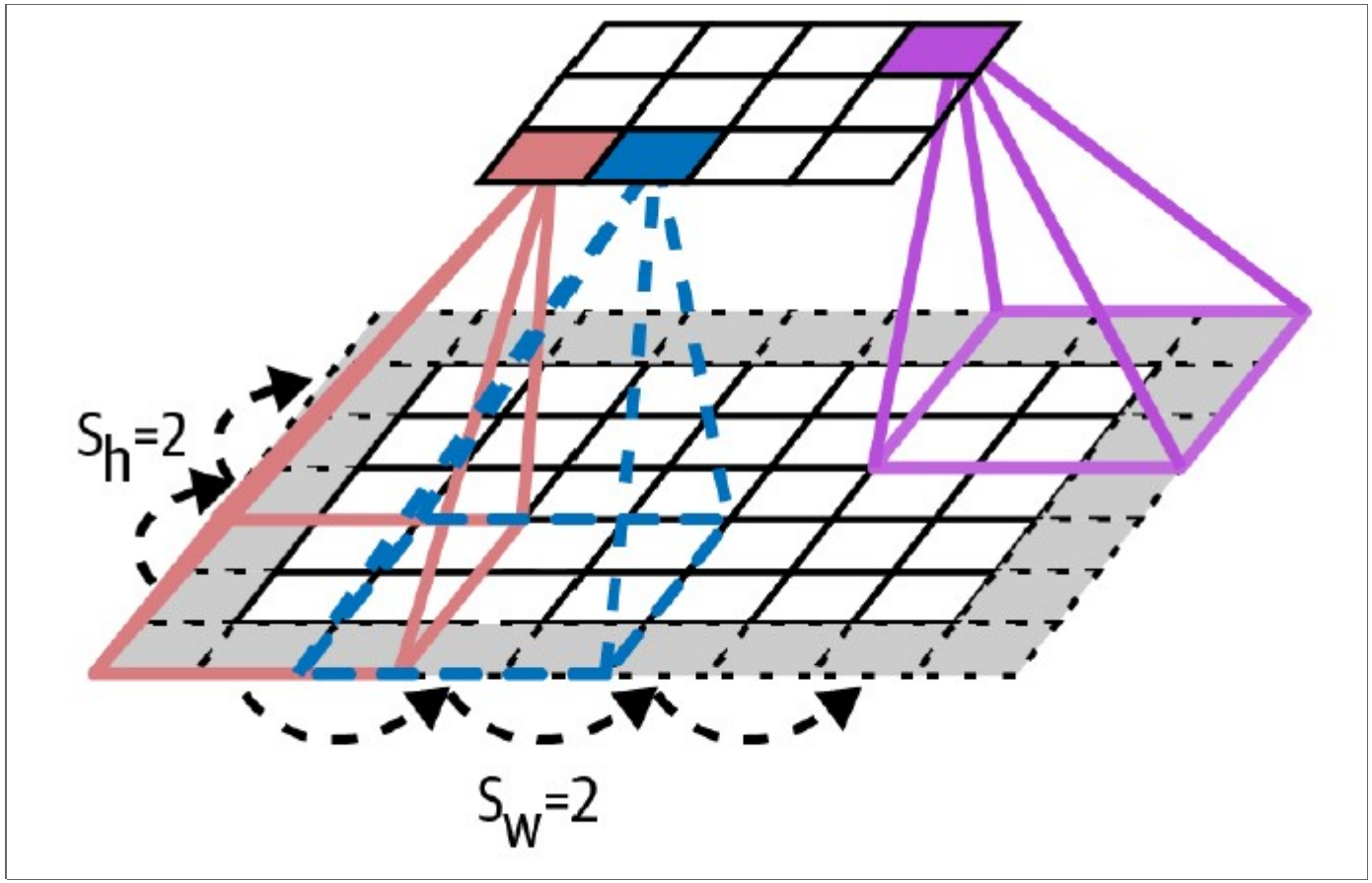


Connection between layers and zero padding

Strides

- If you *space out* the receptive fields, the output will be smaller than the input.
- This will also decrease the computational load.
- The **stride** is the number of pixels between two consecutive receptive fields.
  - Very often, the stride is the same in horizontal and vertical directions, but this is not required.

Strides



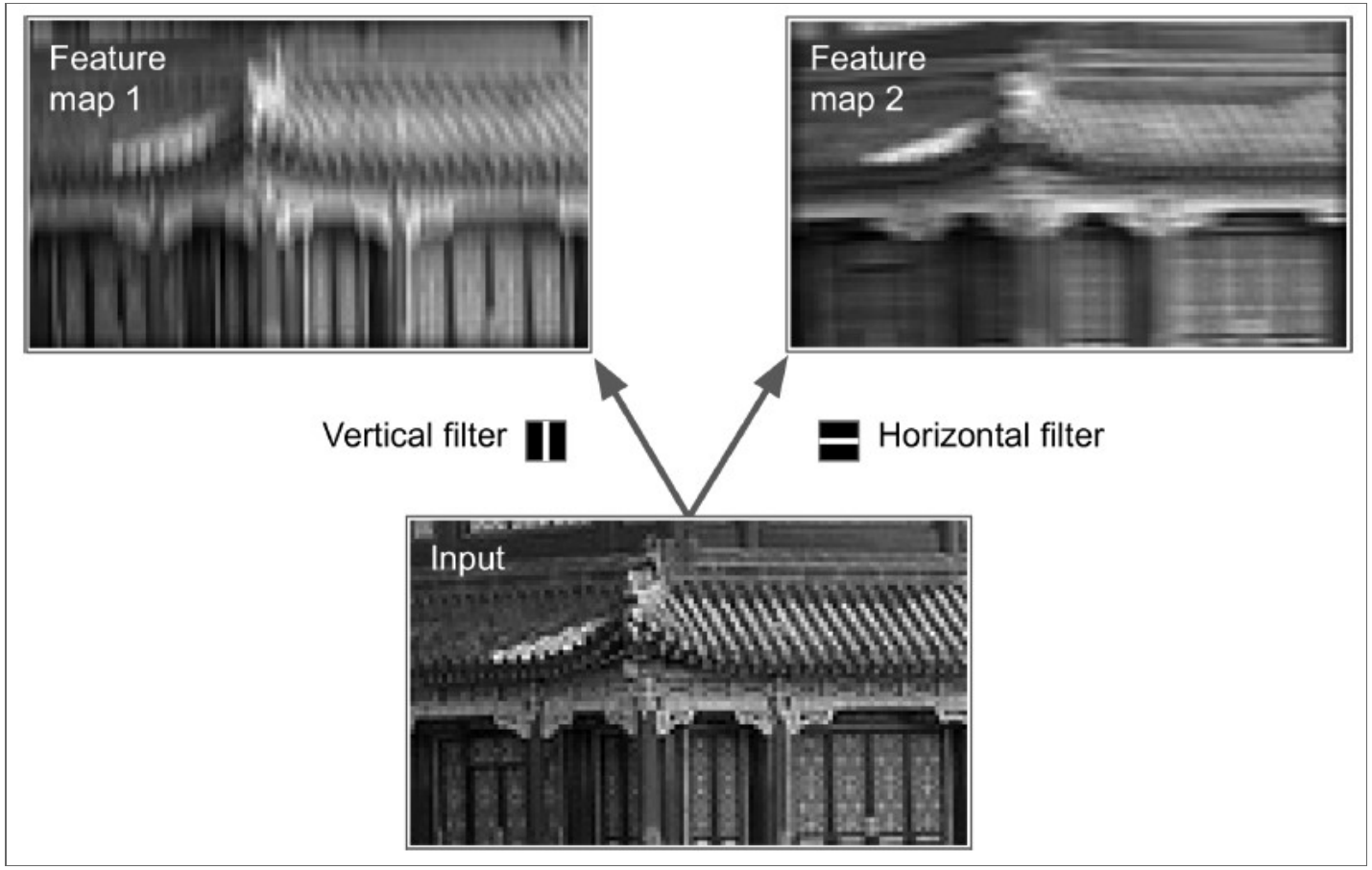
Reducing the image using a stride of 2

14.2.1 Filters

Neuron Weights

- The weights of a neuron can be represented as a small image the size of the receptive field.
  - We call this image the **filter** or **kernel**.
- All neurons in a given convolutional layer *use the same filter*.
- On the next slide two filters are used, but of size 7×7.
  - One filter is all zero except for the vertical line in the middle, which is all ones.
  - The second filters is all zero except for the horizontal line in the middle, which is all ones.

Feature Map Example



Feature maps

Feature Maps

- A **feature map** is the output of a convolutional layer where all neurons share the same weights (i.e., the same filter) and bias.
- The feature map highlights the areas in an image that activate the filter the most.

Filters are Learned

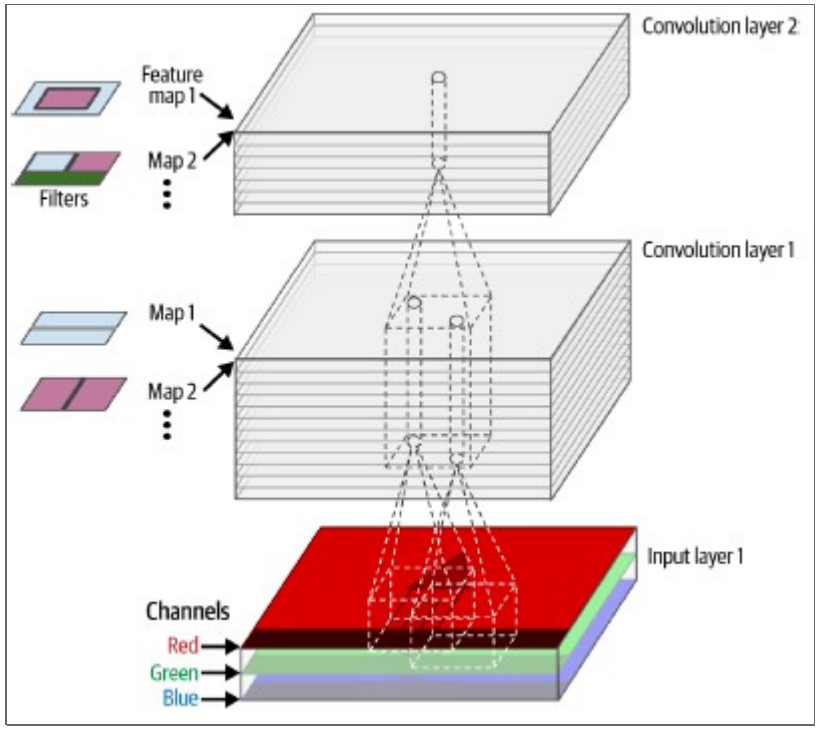
We do not specify filters manually. The convolutional layer learns the best filters for the task at hand.

14.2.2 Stacking Multiple Filter Maps

Multiple Filters

- In practice, a convolutional layer has **multiple filters**.
- The output of each filter is a feature map, so a convolutional layer outputs a stack of feature maps.
  - The output of a convolutional layer is thus a 3D tensor.
- There is one neuron for each pixel in each feature map.
  - All the neurons in the same feature map share the same parameters (i.e. same kernel and bias).
- A neuron's receptive field is as before but now it *extends across all feature maps* of the previous layer.

Multiple Filters



Multiple filters

Multiple Filters

A convolutional layer simultaneously applies multiple filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

Parameter Sharing

- The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model.
- Once a CNN has learned to recognize a pattern in one location, it can recognize it in any other location.
  - With a fully connected layer, the network would have to learn the pattern anew for each new position.

14.2.3 Implementing Convolutional Layers with Keras

Load and Preprocess Some Images

Load two sample images, crop them to 70x120 pixels, and scale their pixel intensities to the 0-1 range:

```
from sklearn.datasets import load_sample_images
import tensorflow as tf

images = load_sample_images()["images"]
# See chapter 13 for these preprocessing layes
images = tf.keras.layers.CenterCrop(height=70, width=120)(images)
images = tf.keras.layers.Rescaling(scale=1 / 255)(images)
```

A 4D-Tensor

- The shape of `images` is `(2, 70, 120, 3)`, which is a 4D tensor.
- The first dimension is the number of images. There are 2 images.
- The second and third dimensions are the height and width of each image. They are 70 pixels high and 120 pixels wide.
- The fourth dimension is the number of color channels per pixel. There are 3 channels: red, green, and blue.
- In general a batch of images is represented by a 4D tensor of shape `(batch size, height, width, channels)`.

Creating a Convolutional Layer

- The `tf.keras.layers.Conv2D` class creates a convolutional layer.
  - Note: the 2D refers to the number of *spatial* dimensions; the layer itself processes 4D tensors.

```
# Create the convolutional layer; 32 filters of size 7x7
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7)
# Apply the convolutional layer to the images
fmaps = conv_layer(images)
```

- The number of filters is the number of output feature maps.
- `kernel_size=7` is equivalent to `kernel_size=(7, 7)`.

Output of a Convolutional Layer

- Remember that the input was a 4D tensor of shape `(2, 70, 120, 3)`.
- The output is a 4D tensor of shape `(2, 64, 114, 32)`.
  - The first dimension is the batch size.
  - The second and third dimensions are the height and width of the feature maps.
  - The fourth dimension is the number of feature maps/filters.

“Valid” Padding

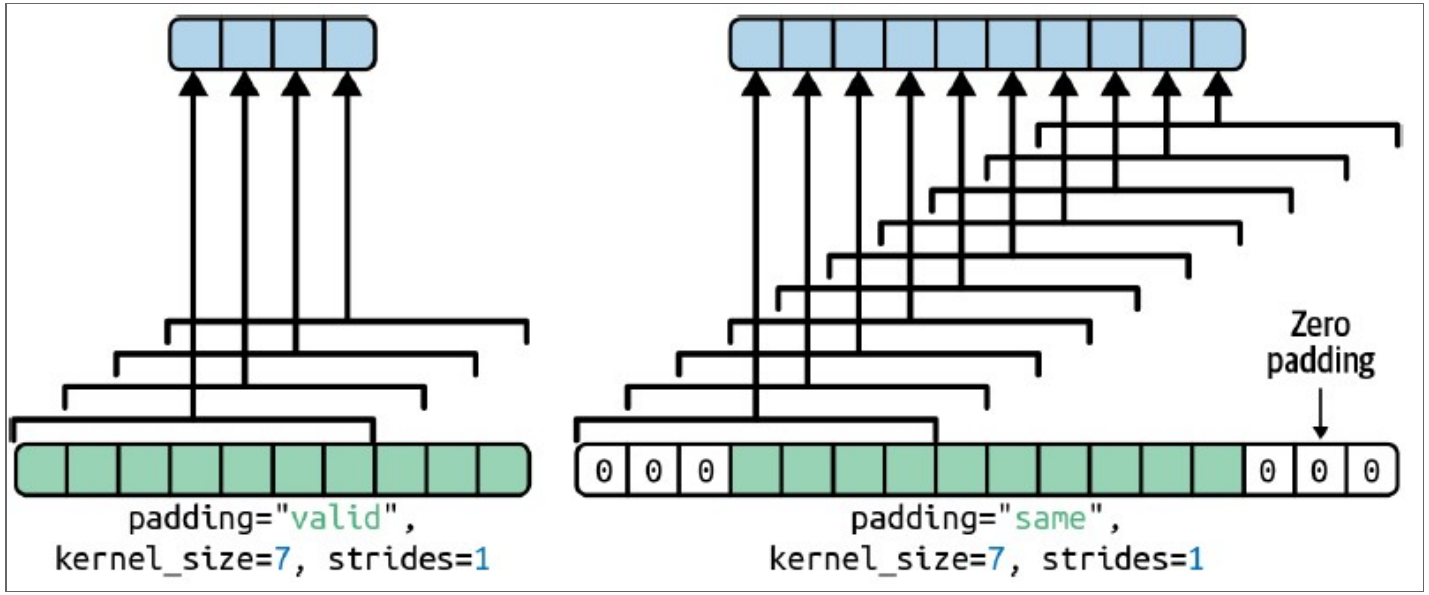
- By default, a convolutional layer uses *no padding at all*, which is called **valid padding**.
- With valid padding, the receptive field of a neuron does not go beyond the edges of the input image.
  - Some pixels are lost, depending on the filter size.

“Same” Padding

- If we set `padding="same"`, the convolutional layer uses just enough padding to ensure that the output has the same height and width as the input.

```
# Use "same" padding
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7,
                                    padding="same")
fmaps = conv_layer(images)
fmaps.shape # gives (2, 70, 120, 32)
```

Valid vs. Same Padding

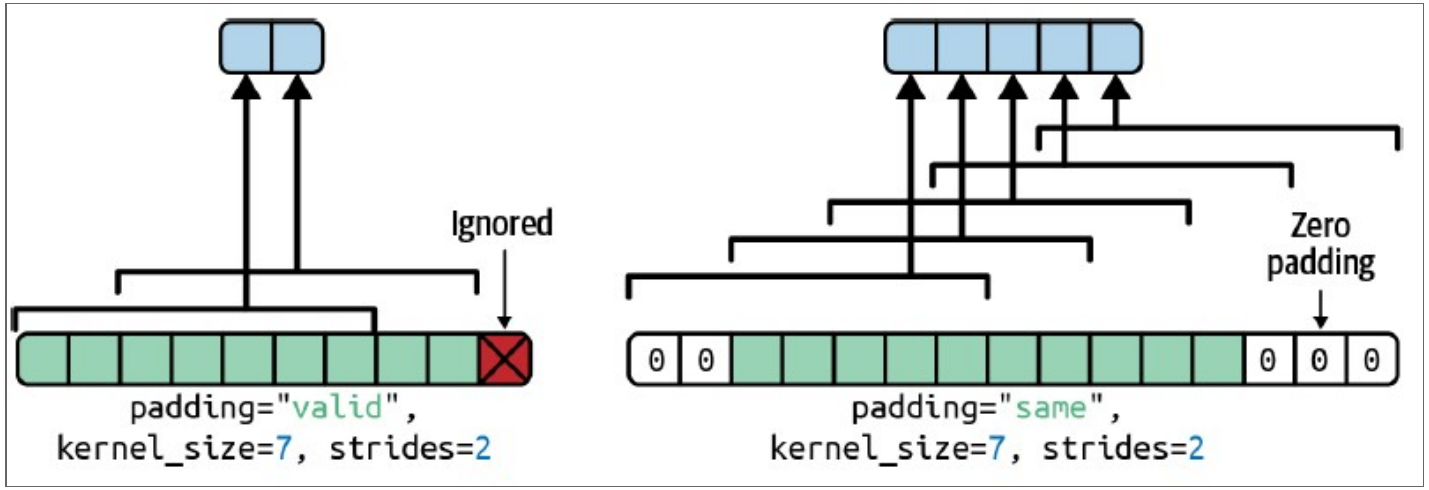


Valid vs. same padding

Using Strides

- If you set `strides` to a value greater than 1 (in any direction) then the output will be smaller than the input.
- E.g. with `strides=2` the output will be half the height and half the width of the input.

Valid vs. Same Padding with Strides



Valid vs. same padding with strides

Weights of a Convolutional Layer

```
# First line repeated from above
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7,
                                    padding="same")
kernels, biases = conv_layer.get_weights()
kernels.shape  # gives (7, 7, 3, 32)
biases.shape   # gives (32,)
```

- `kernels.shape` is `(kernel_height, kernel_width, input_channels, output_channels)`.
- `biases.shape` is `(output_channels,)`.

Weights of a Convolutional Layer

- The height and width of the input images *does not appear in the shape of the kernels*.
- A convolutional layer can be applied to images of *any size*, provided they have the right number of channels (and as long as they are at least as large as the kernels).
  - In this case the number of input channels should be 3.

Activation Functions

- The `tf.keras.layers.Conv2D` class does not apply any activation function by default.
- Typically you will want to add an activation function.
  - Otherwise, stacking multiple convolutional layers would be equivalent to a single convolutional layer.
  - This is because the convolutional layer itself performs a linear transformation.
- You will also want to specify a kernel initializer (e.g. using He initialization).

14.2.4 Memory Requirements

Convolutional Layer in Numbers

- The reverse pass of backpropagation requires all the intermediate values computed during the forward pass.
- Example: convolutional layer with 200 5×5 filters with stride 1 and "same" padding with RGB images of size 150×100 as input.
- Number of parameters (kernels + biases):
  $5 \times 5 \times 3 \times 200 + 200 = 15,200$ (this is small compared to a fully connected layer).

Convolutional Layer in Numbers

- Number of floating point multiplications?
  - 200 feature maps, 150×100 neurons.
  - Each neuron requires $5 \times 5 \times 3 = 75$ multiplications.
  - Total multiplications: $200 \times 150 \times 100 \times 75 = 225,000,000$
- Memory requirements? Suppose we use 32-bit floats, i.e. 4 bytes per float.
  - The size of the output in bytes is thus:
    $200 \times 150 \times 100 \times 4 = 12,000,000$ = 12 MB.
  - If the batch contains 100 images, the memory requirements will be 1.2 GB.

Convolutional Layer in Numbers

- During training, each layer needs to keeps its output in memory as it is needed for the reverse pass.
- During inference, each layer can release its output after it has been used by the next layer.
  - Thus, you only need the RAM required by two consecutive layers.

14.3 Pooling Layers

Pooling Layers

The purpose of a **pooling layer** is to *subsample* the original images. This reduces:

- computational load
- memory usage
- number of parameters (and hence reduces the risk of overfitting)

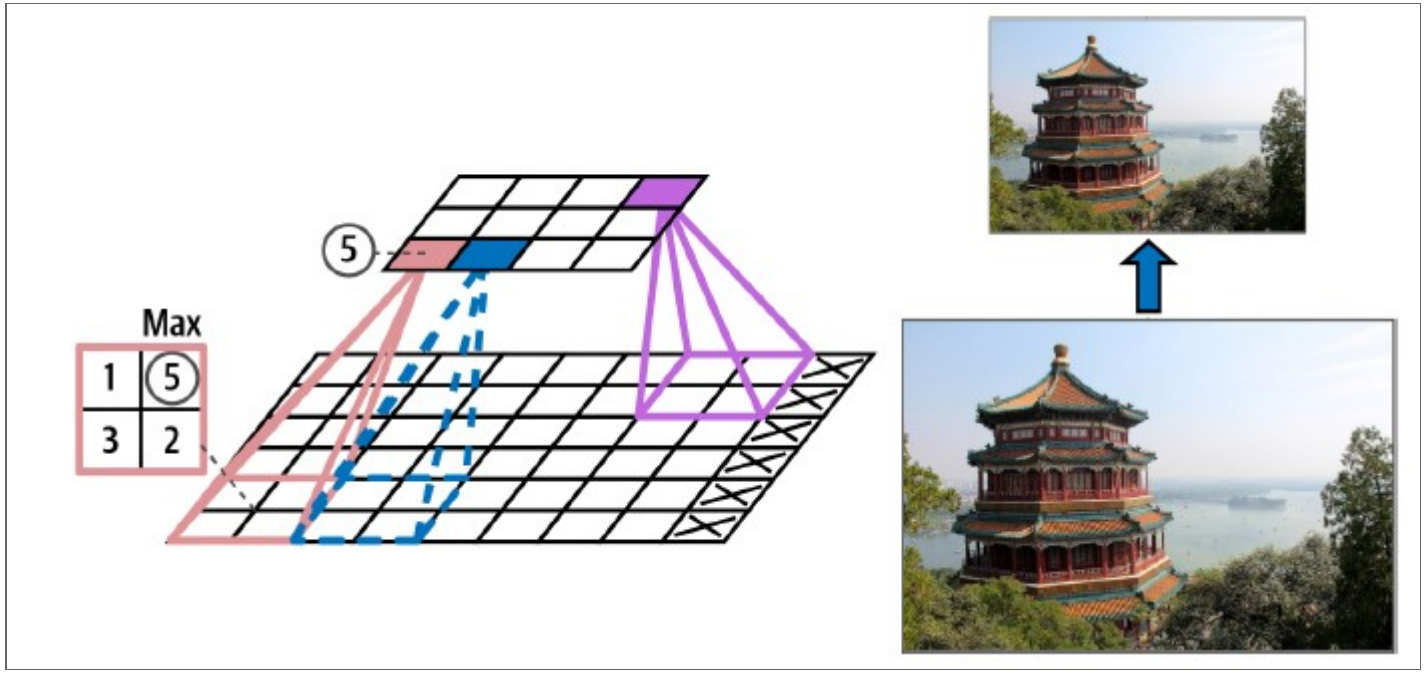
Pooling Layer Connections

- Each neuron in a pooling layer is connected to (the outputs) of a limited number of neurons in the previous layer, located within a small rectangular receptive field.
- You must define the
  - size
  - stride
  - padding type

Pooling Layer has No Weights

- A pooling layer has *no weights*.
- A pooling layer simply *aggregates* (i.e. summarizes) the values of the neurons in its receptive field.
- The most common type of pooling layer is the **max pooling layer**.
  - It simply returns the maximum value of the neurons in its receptive field.

Max Pooling Layer Example



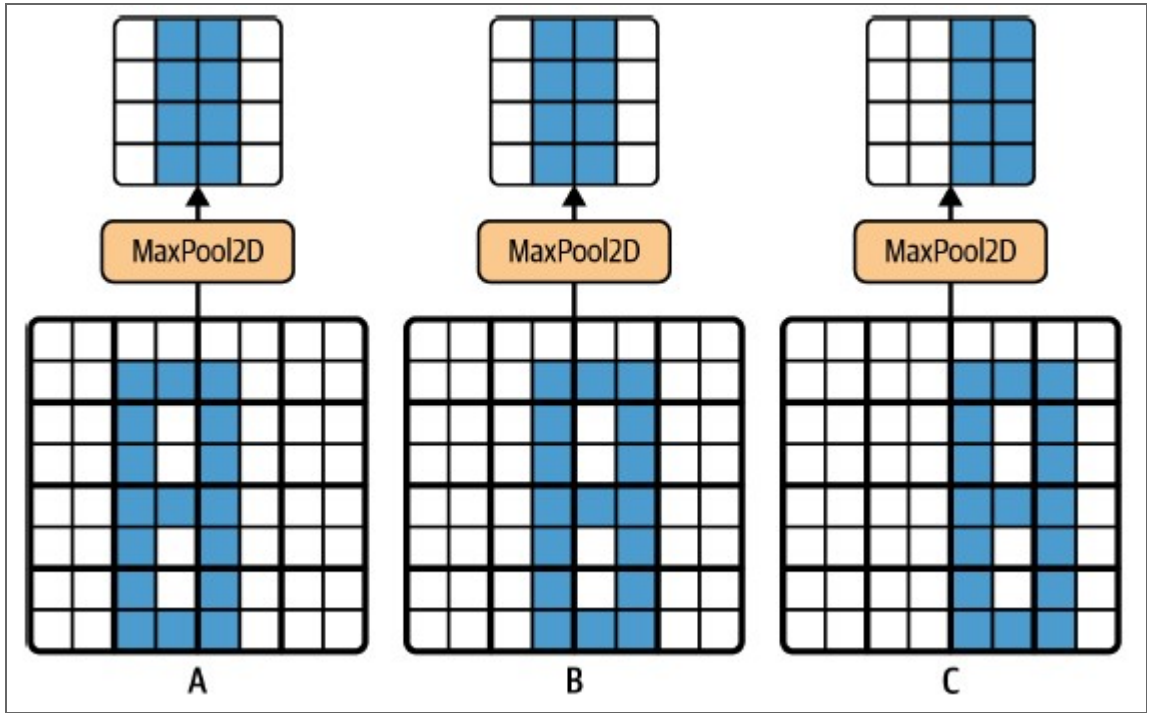
Max pooling layer with kernel 2×2 and stride 2, without padding

Important Note

A pooling layer typically works on every input channel independently, so the output depth (i.e. the number of channels) is the same as the input depth.

Translation Invariance

- Using a max pooling layer introduces some level of *invariance* to small translations.



Translation invariance

Downsides of Max Pooling

- Max pooling is very destructive. With a kernel size of 2×2 and a stride of 2, the output is 4 times smaller than the input.
  - In this case, max pooling simply removes 75% of the input values.
- In some applications translation invariance is not desirable.
  - E.g. in semantic segmentation (i.e. classifying each pixel in an image), we do not want translation invariance: if the input is shifted one pixel to the right, then the output should also be shifted one pixel to the right.

14.4 Implementing Pooling Layers with Keras

Pooling Layers in Keras

- The `tf.keras.layers.MaxPool2D` class creates a max pooling layer.

```
max_pool = tf.keras.layers.MaxPool2D(pool_size=2)
```

- The code above creates a max pooling layer with a kernel size of 2×2 and a stride of 2. It uses `valid` padding by default.

Average Pooling Layer

- Instead of using a max pooling layer, you can also use a `AvgPool2D` layer.
  - These are not very common nowadays.

Global Average Pooling Layer

- **Global average pooling** is still very popular.
  - It simply computes the mean of each entire feature map.
  - Thus, it is like average pooling with a kernel size equal to the size of the feature map.
- This layer is extremely destructive and is typically used just before the output layer.

Global Average Pooling Layer

```
# images is a Tensor of shape (2, 70, 120, 3)
global_avg_pool = tf.keras.layers.GlobalAvgPool2D()
global_avg_pool(images)
```

yields

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0.643388  , 0.59718215, 0.5825038 ],
       [0.7630747 , 0.2601088 , 0.10848834]], dtype=float32)>
```

We simply get the average pixel intensity for each of the three channels (for each image).

14.5 CNN Architectures

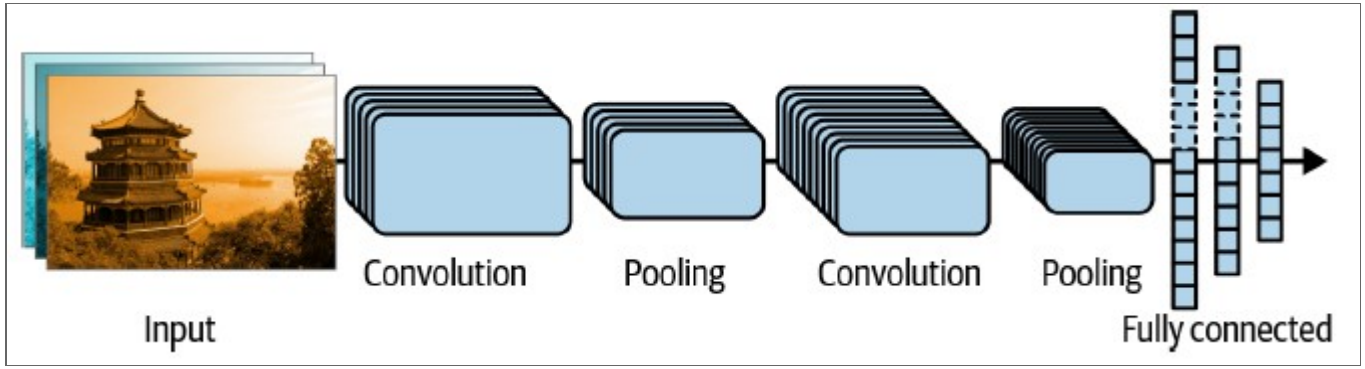
Typical CNN Architecture

- A typical CNN architecture
  - stacks a few convolutional layers (each one followed by a ReLU activation function),
  - then a pooling layer,
  - then another few convolutional layers (+ReLU), then another pooling layer, and so on.
- The image gets smaller as it progresses through the network, but it also typically gets deeper (i.e. with more feature maps).

Typical CNN Architecture (Ctd.)

- At the top of the stack, a regular feedforward network is added.
  - Typically only a few fully connected layers.
  - The final layer outputs the prediction, e.g. a softmax layer for classification.

Typical CNN Architecture



Typical CNN architecture

Basic CNN Example

- We show how to tackle Fashion MNIST using a basic CNN.

Basic CNN Example

- We start by defining a “default” convolution using Python’s `functions.partial` function.
  - The keyword arguments that we supply here will be the default values for all convolutional layers in the model.
  - But, you can still override these values and supply additional arguments when creating a layer.

```
from functools import partial

DefaultConv2D = partial(tf.keras.layers.Conv2D,
  kernel_size=3, padding="same",
  activation="relu", kernel_initializer="he_normal")
```

Basic CNN Example

```
model = tf.keras.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPool2D(), # uses default pool_size of (2,2)
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=64, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=10, activation="softmax")
])
```

Basic CNN Example

- The model is a `Sequential` model.
- Make sure to specify a single channel for the images, as `input_shape=(28,28,1)`.
- After the first layer (which uses a large kernel size), we use max pooling to divide the width and height of the images in half.
- Then we repeat the same structure twice:
  - Two convolutional layers followed by a max pooling layer.

Basic CNN Example (Ctd.)

- The number of filters grows as we progress through the network.
  - Makes sense: number of low-level features is limited but there are many ways to combine them into higher-level features.
- Finally, we add a fully connected network.
  - Start by `Flatten`ing the outputs of the last max pooling layer.
  - Add two fully connected layers with dropout.
  - Add a dense output layer with 10 neurons (one per class), using the softmax activation function.

Results of this Basic CNN

- After compiling with `sparse_categorical_crossentropy` loss and fitting on the training set, you can get an accuracy of more than 92% on the test test.

ImageNet Challenge

- Competition for image classification.
  - Approximately 1.2 million images, each labeled with one of 1,000 different classes.
  - Classes can be really subtle (e.g. 120 different dog breeds).
- Performance is measured using top-five error rate.
  - I.e. the model is “wrong” if the correct answer is not among its top five predictions.

14.5.1 LeNet-5

LeNet-5 Architecture

- Created by Yann Lecun in 1998 for digit recognition.
- Looks very similar to the basic CNN architecture we just saw.
- Major differences:
  - Now we use ReLU activation functions instead of tanh.
  - We use softmax instead of RBF (Radial Basis Functions) for the output layer.

14.5.2 AlexNet

AlexNet

- The winner of the 2012 ImageNet challenge, by a large margin:
  - Top-five error rate of 17%, with second entry achieving 26%.
- Created by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton.

AlexNet Architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU
F9	Fully connected	–	4,096	–	–	–	ReLU
S8	Max pooling	256	6 × 6	3 × 3	2	`valid`	–
C7	Convolution	256	13 × 13	3 × 3	1	`same`	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	`same`	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	`same`	ReLU
S4	Max pooling	256	13 × 13	3 × 3	2	`valid`	–
C3	Convolution	256	27 × 27	5 × 5	1	`same`	ReLU
S2	Max pooling	96	27 × 27	3 × 3	2	`valid`	–
C1	Convolution	96	55 × 55	11 × 11	4	`valid`	ReLU
In	Input	3 (RGB)	227 × 227	–	–	–	–

AlexNet architecture

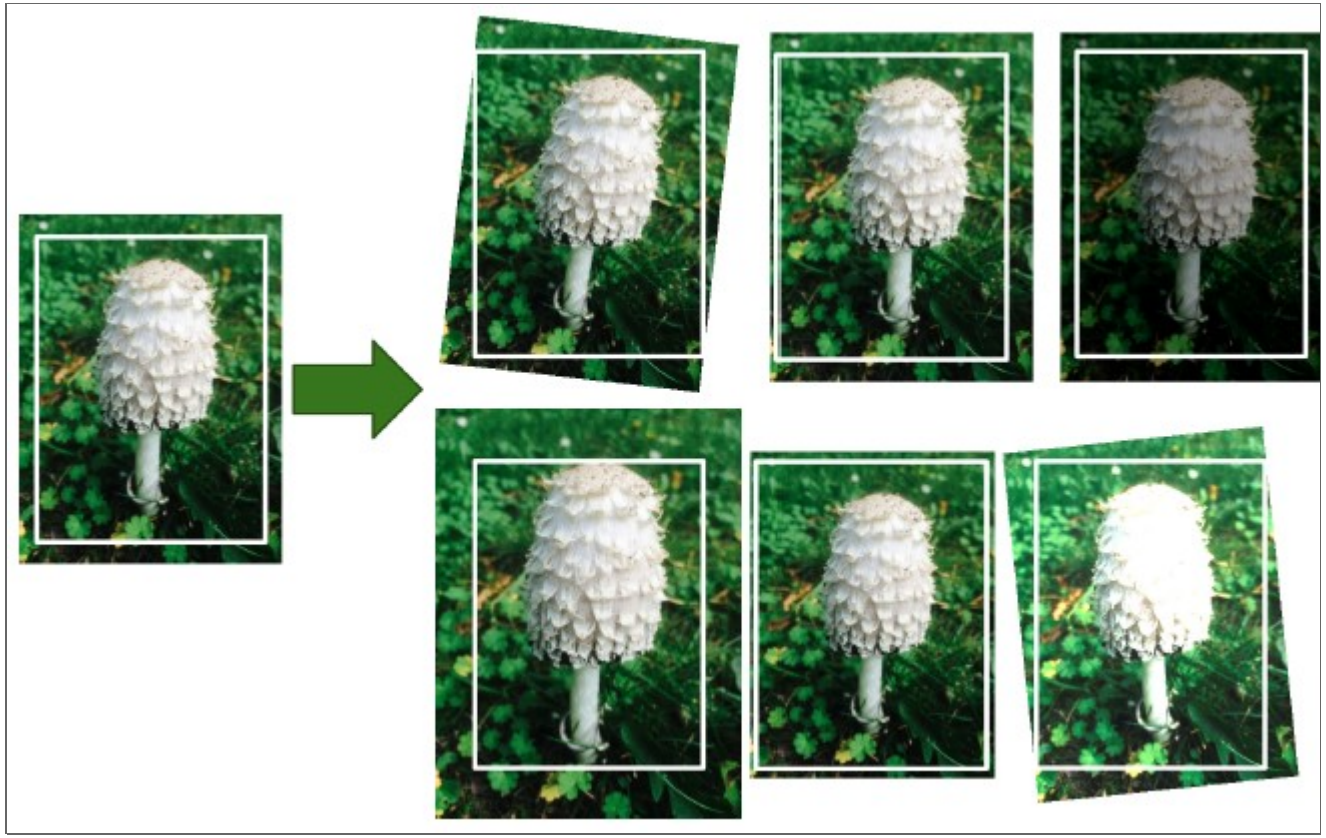
Regularization Techniques in AlexNet

- Dropout of 50% after layers F9 and F10 (the fully connected layers).
  - Of course, only during training!
- They used **data augmentation** by
  - randomly shifting the training images
  - flipping the images horizontally
  - changing the lighting conditions

Data Augmentation

- Data augmentation artificially increases the size of the training set by generating many realistic variants of each training instance.
- The generated instances should still be realistic.
  - Ideally, a human can not tell whether an image is original or augmented.
- The modifications should be *learnable*:
  - Thus, adding white noise will not help.

Data Augmentation



Data augmentation

Data Augmentation in Keras

- Keras provides preprocessing layers for data augmentation.
  - See Chapter 13 for more details.
  - E.g. `RandomCrop`, `RandomRotation`.
- Applying these transformations forces the model to be more tolerant to these changes.
- Data augmentation can also be used with unbalanced datasets. You can use it to generate more instances of the underrepresented classes. This is called *Synthetic Minority Oversampling Technique* (SMOTE).

Local Response Normalization

- With **local response normalization** (LRN), the most strongly activated neurons inhibit other neurons located at *the same position* in *neighboring feature maps*.
  - Such competitive activation has been observed in biological neurons.
  - It encourages feature maps to specialize, pushing them apart and forcing them to explore a wider range of features.

Local Response Normalization

- The formula for LRN is

$$b_i = a_i \left(k + \ \alpha \sum_{j=\max(0,i-r/2)}^{\min(f_n-1,i+r/2)} a_j^2 \right)^{-\beta} \qquad \text{where}$$

- b_i is the normalized output for a neuron (at some position) in feature map i.
- a_i, a_j are the activations of the neurons (at the same position) in feature maps i and j.
- k, α, β and r are hyperparameters: k is the bias, r is the depth radius.
- f_n is the number of feature maps.

LRN in AlexNet

- In AlexNet, LRN was used after the ReLU step of layers C1 and C3.
- The parameters used where $r = 5$, $\alpha = 0.0001$, $\beta = 0.75$ and $k = 2$.
- You can use `tf.nn.local_response_normalization` to implement LRN in Keras.
  - You can also wrap this TensorFlow function in a `Lambda` layer.

14.5.3 GoogLeNet

14.5.4 VGGNet

14.5.5 ResNet

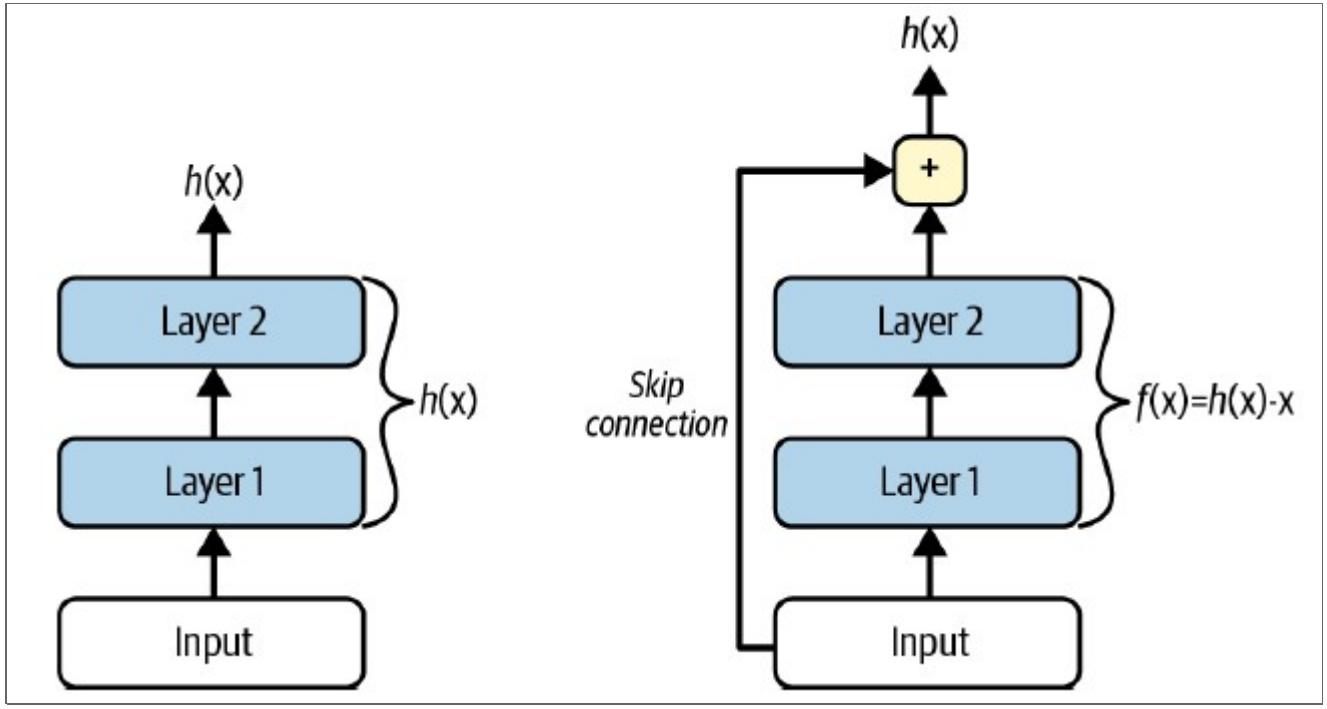
ResNet

- The **Residual Network** or **ResNet** won the 2015 ImageNet challenge.
  - Top-five error rate of under 3.6%.
- The winning variant had 152(!) layers.
- The key to training such a deep network is to use **skip connections**.
  - These connections skip over some layers: the signal feeding into a layer is also added to the output of a layer located a bit higher up the stack.

Skip Connections

- The goal of a neural network is to model/approximate a target function $h(\mathbf{x})$.
- When you add $\mathbf{x}$ to the output of the network, the network instead will model $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$.
  - This is called *residual learning*.
- When you initialize (with weights close to zero), the outputs of the network are also close to zero. When you add a skip connection, the network is close to the identity function.
  - If the target is close to the identity, this will speed up training.

Skip Connections

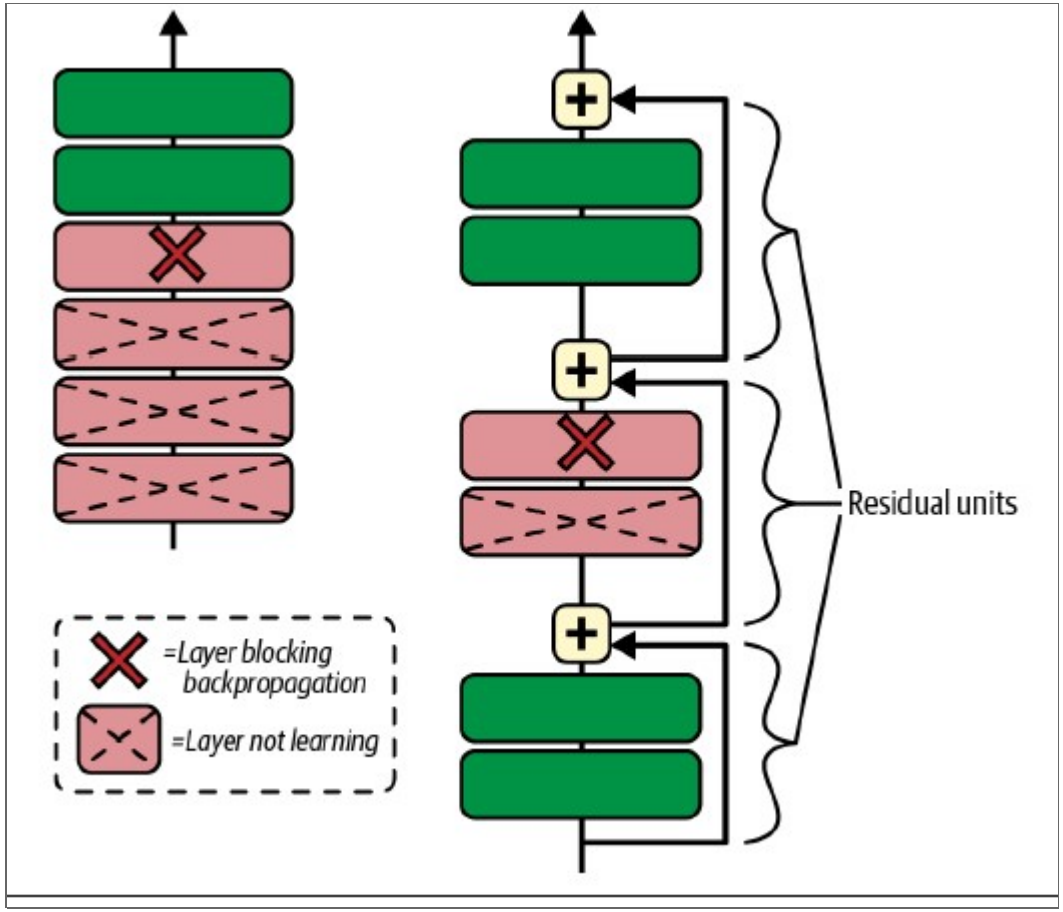


Skip connections

Skip Connections

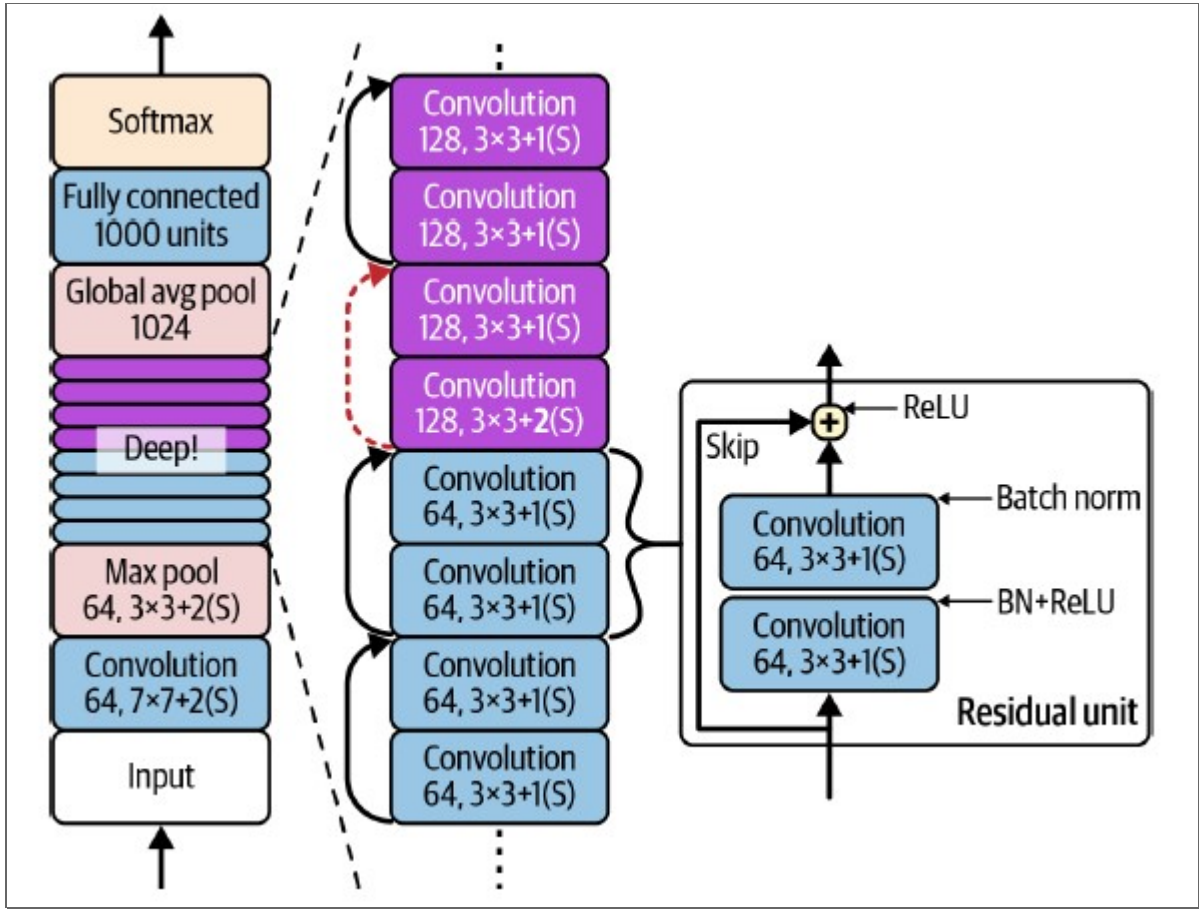
- If you add many skip connections, the network can start making progress even if several layers have not started learning yet.
- Thanks to the skip connections, the signal can easily make its way across the whole network.
  - The network as a whole can be seen as a stack of *residual units* (each is a small network with a skip connection)

Deep Residual Network



Deep residual network

ResNet Architecture



ResNet architecture

ResNet Architecture

- The first two layers start by dividing the image width and height by 4.
  - This reduces the computational load.
  - The first layer uses a large kernel size to preserve much information.
- The network ends with a global average pooling layer and a softmax output layer.

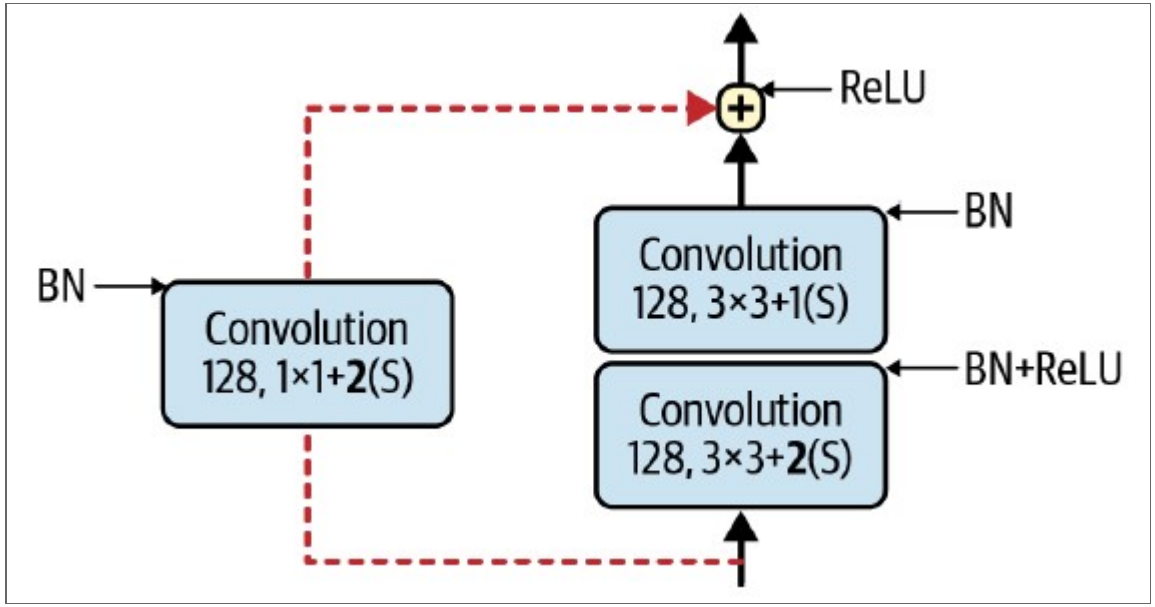
ResNet Architecture

- In between is a very deep stack of residual units.
- Each residual unit
  - is composed of two convolutional layers (no pooling!) with batch normalization and ReLU activation.
  - Kernel size is 3×3, stride 1 and “same” padding (so that spatial dimensions are preserved).

ResNet Architecture

- The number of feature maps is doubled every few residual units, at the same time as the spatial dimensions are halved (using a convolutional layer with stride 2).
  - Now, the outputs cannot be added directly to the outputs of the residual unit, because they do not have the same shape.
  - To solve this, the inputs are passed through a 1x1 convolutional layer with stride 2 and the right number of output feature maps.

Skip Connection when Shapes Differ



Skip connection when shapes differ

14.5.6 Xception

14.5.7 SENet

14.5.8 Other Noteworthy Architectures

14.5.9 Choosing the Right CNN Architecture

- How to choose a model for your project?
- This depends on what matters most for you:
  - Accuracy?
  - Model size? (e.g. for mobile apps)
  - Inference speed? (on CPU or GPU)
- Table 14-3 in the book shows a list of models together with their characteristics.

14.6 Implementing a ResNet-34 CNN using Keras

Implementing ResNet-34 with Keras

- To illustrate that Keras makes it easy to implement complex architectures, we will implement ResNet-34.
- Note: in general, you would simply use a *pretrained model* instead.

Implementing the Residual Unit

```
from functools import partial

DefaultConv2D = partial(tf.keras.layers.Conv2D,
    kernel_size=3, strides=1, padding="same",
    kernel_initializer="he_normal", use_bias=False)
```

Implementing the Residual Unit

```
class ResidualUnit(tf.keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = tf.keras.activations.get(activation)
        self.main_layers = [
            DefaultConv2D(filters, strides=strides),
            tf.keras.layers.BatchNormalization(),
            self.activation,
            DefaultConv2D(filters),
            tf.keras.layers.BatchNormalization()
        ]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                DefaultConv2D(filters, kernel_size=1, strides=strides),
                tf.keras.layers.BatchNormalization()
            ]
```

Implementing the Residual Unit

```
class ResidualUnit(tf.keras.layers.Layer):

  def __init__( ...):
    # code on previous slide

  def call(self, inputs):
      Z = inputs
      for layer in self.main_layers:
          Z = layer(Z)
      skip_Z = inputs
      for layer in self.skip_layers:
          skip_Z = layer(skip_Z)
      return self.activation(Z + skip_Z) # This is the skip connection
```

Implementing the Residual Unit

- This implements Figure 14-19 of the book, see also slide “Skip Connection when Shapes Differ”.
  - In the constructor all the layers are created.
  - The `main_layers` are the ones for the regular path.
  - The `skip_layers` are added when the stride is not equal to one.
- The `call` method makes the inputs go through the main layers and the skip layers (if any).
- The final line implements the skip connection.
  - The outputs from the skip layers are added to the outputs of the main layers.
  - Finally, the activation function is applied.

Implementing ResNet-34

- We can use `Sequential` to implement ResNet-34, since it is just a really deep stack of layers.
  - We can treat each `ResidualUnit` as a layer.

Implementing ResNet-34

```
model = tf.keras.Sequential([
    DefaultConv2D(64, kernel_size=7, strides=2, input_shape=[224, 224,
          3]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
    tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"),
])
prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters

model.add(tf.keras.layers.GlobalAvgPool2D())
model.add(tf.keras.layers.Flatten())
# Assume we want to classify into 10 classes
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

14.7 Using Pretrained Models from Keras

Pretrained Models

- Normally, you will not implement standard models yourself.
  - The `tf.keras.applications` package provides many pretrained models.
- The following line of code loads a pretrained ResNet-50 model, pretrained on ImageNet.

```
model = tf.keras.applications.ResNet50(weights="imagenet")
```

Resizing the Images

- When you want to use the model you just loaded, the images you feed to it must have the correct size.
  - The ResNet-50 model expects images of size 224×224

```
from sklearn.datasets import load_sample_images

images = load_sample_images()["images"]
images_resized = tf.keras.layers.Resizing(height=224, width=224,
                                          crop_to_aspect_ratio=True)(images)
```

Preprocessing the Images

- The pretrained models assume that the images are preprocessed in a specific way:
  - e.g. pixel values between 0 and 1, or -1 and 1.
- Each model provides a `preprocess_input` function that you can use to preprocess the images in the right way.
  - This assumes pixel values between 0 and 255.

```
# images_resized from previous code

inputs = tf.keras.applications.resnet50.preprocess_input(images_resized)
```

Obtain Predictions

- Now we can use the model to obtain predictions.

```
Y_proba = model.predict(inputs)
Y_proba.shape # (2, 1000)
```

- As you can see, we get probabilities for each of the 1000 classes, which isn't very practical.

Decode the Predictions

- The `decode_predictions` function helps to display the top K predictions, together with the name of the class and the estimated probability.

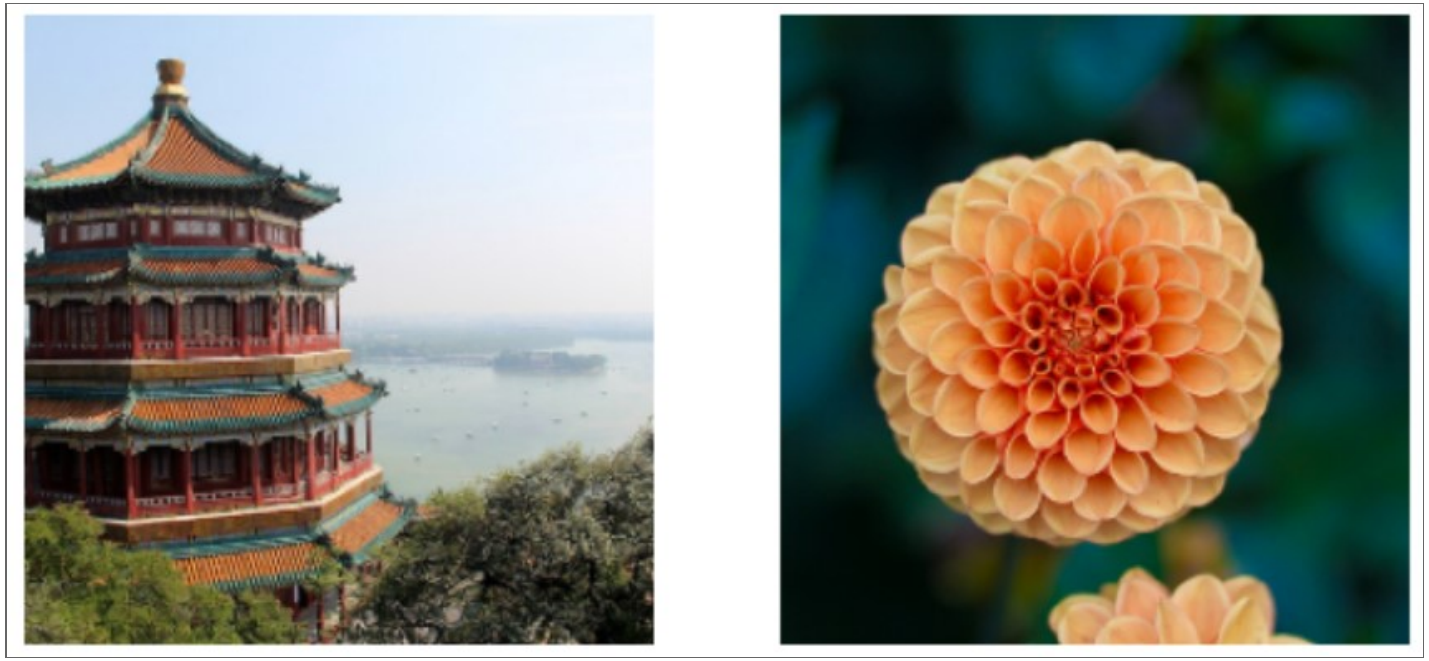
```
top_K = tf.keras.applications.resnet50.decode_predictions(Y_proba,
          top=3)
for image_index in range(len(images)):
    print(f"Image #{image_index}")
    for class_id, name, y_proba in top_K[image_index]:
        print(f"  {class_id} - {name:12s} {y_proba:.2%}")
```

gives

```
Image #0
  n03877845 - palace       54.69%
  n03781244 - monastery    24.72%
  n02825657 - bell_cote    18.55%
Image #1
  n04522168 - vase         32.66%
  n11939491 - daisy        17.81%
```

```
n03530642 - honeycomb    12.06%
```

The Input Images



Input images

14.8 Pretrained Models for Transfer Learning

Transfer Learning

- When you don’t have enough data to train a model from scratch, it is a good idea to reuse the lower layers of a pretrained model.
  - This is called **transfer learning** (as already seen in Chapter 11).

Example

- Let’s reuse a pretrained Xception model to classify images of flowers.
- The flowers dataset is loaded using TensorFlow Datasets.

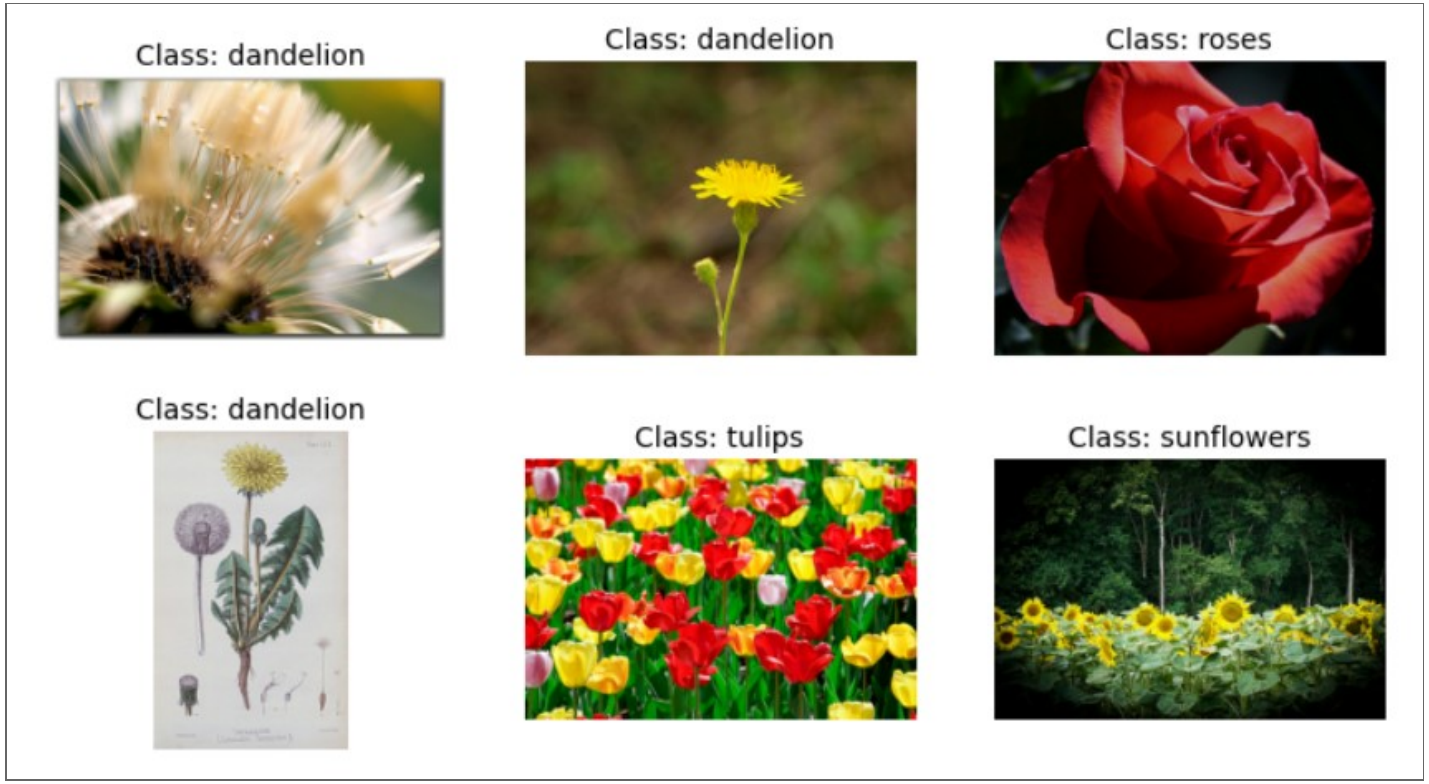
```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True,
                          with_info=True)
dataset_size = info.splits["train"].num_examples # 3670
class_names = info.features["label"].names # ['dandelion', 'daisy', ...]
n_classes = info.features["label"].num_classes # 5
```

Splitting the Dataset

- The loaded dataset only contains training images.
- Let's load it again using a split of 10% for testing, 15% for validating and 75% for training.

```
test_set_raw, valid_set_raw, train_set_raw = tfds.load(
    "tf_flowers",
    split=["train[:10%]", "train[10%:25%]", "train[25%:]"],
    as_supervised=True)
```

Some Example Pictures



Some example pictures with their classes

Preprocessing the Images

- As can be seen from the example pictures, the images have different sizes.
  - We need to resize them to a fixed size and normalize the pixel values.
- Also, the datasets contain individual images.
  - We will need to batch them.
  - We will also shuffle the training dataset, and use prefetching.

Creating the Datasets

```
batch_size = 32
# Resize and preprocess pipeline
preprocess = tf.keras.Sequential([
    tf.keras.layers.Resizing(height=224, width=224,
                             crop_to_aspect_ratio=True),
    tf.keras.layers.Lambda(
      tf.keras.applications.xception.preprocess_input)
])

# Apply the pipeline to the datasets using map.
train_set = train_set_raw.map(lambda X, y: (preprocess(X), y))
train_set = train_set.shuffle(1000,
          seed=42).batch(batch_size).prefetch(1)
valid_set = valid_set_raw.map(lambda X, y: (preprocess(X), y))
valid_set = valid_set.batch(batch_size)
test_set = test_set_raw.map(lambda X, y: (preprocess(X), y))
test_set = test_set.batch(batch_size)
```

Add Data Augmentation

- Since the dataset is not very large, some data augmentation will help. We will
  - randomly flip the images horizontally
  - rotate them a little bit
  - tweak the contrast

```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip(mode="horizontal", seed=42),
    tf.keras.layers.RandomRotation(factor=0.05, seed=42),
    tf.keras.layers.RandomContrast(factor=0.2, seed=42)
])
```

Load Pretrained Model

- Now we can load the pretrained Xception model, pretrained on ImageNet.
  - We set `include_top=False` to exclude the top of the network.
  - This excludes the global average pooling layer and the dense output layer.
- We then add our own global average pooling layer and dense output layer with one unit per class.

Load Pretrained Model

```
n_classes = 5 # from previous code
base_model = tf.keras.applications.xception.Xception(
  weights="imagenet", include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)
model = tf.keras.Model(inputs=base_model.input, outputs=output)
```

Freeze the Weights

- We freeze the weights of the pretrained model.

```
for layer in base_model.layers:
    layer.trainable = False
```

Compile and Train

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=optimizer, metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=3)
```

Unfreeze some Layers

- After training for a few epochs the validation accuracy should be slightly above 80%.
- We are now ready to unfreeze some of the base model's top layer and continue training.

```
# Unfreeze layers from the start of residual unit 7 (out of 14)
for layer in base_model.layers[56:]:
    layer.trainable = True

# Use a much smaller learning rate to avoid damaging the pretrained
          weights
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy",
          optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=10)
```

Results

- After training for 10 more epochs, the test accuracy should be around 92%.
  - With tuning the hyperparameters, lowering the learning rate etc. and training for quite a bit longer we should be able to reach 95% to 97% accuracy.

Errata in the Book

- The book (and the slides) don't actually use the `data_augmentation` model that was introduced.
- This can be fixed as follows:

```
input_ = tf.keras.layers.Input(shape=(224,224,3))
augmented = data_augmentation(input_) # Use data augmentation
base_model_output = base_model(augmented)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model_output)
output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)
model = tf.keras.Model(inputs=[input_], outputs=[output])
```

Everything else stays the same.

14.9 Classification and Localization

Object Localization

- Object localization is a *regression task*. To predict the **bounding box** around an object we can predict
  - the horizontal and vertical coordinates of object's center.
  - the height and width of the bounding box.
- Object localization thus means that we have to predict 4 numbers.

Changes to the Model

- We can add a second dense output layer (with 4 units) to the model.
  - Typically on top of the global average pooling layer.
  - This can be trained using the MSE loss.

Python Code

```
base_model = tf.keras.applications.xception.Xception(weights="imagenet",
                                                     include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
# Output for classification with softmax activation function
class_output = tf.keras.layers.Dense(
    n_classes, activation="softmax")(avg)
# Output for localisation. No activation function.
loc_output = tf.keras.layers.Dense(4)(avg)
# Specify model to have two outputs
model = tf.keras.Model(inputs=base_model.input,
                       outputs=[class_output, loc_output])
# Compile the model. Give the two loss functions and their weights
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
    loss_weights=[0.8, 0.2],  # depends on what you care most about
    optimizer=optimizer, metrics=["accuracy"])
```

Getting the Labels

- Getting the labels typically requires *a lot* of work.
  - Typically the hardest and most expensive part of a machine learning project.
- Many tools are available:
  - Open source: VCC Image Annotator, LabelImg, ...
  - Commercial: Labelbox, Supervisely, ...

Bounding Boxes

- The bounding boxes should be **normalized**:
  - The horizontal and vertical coordinates should be between 0 and 1.
  - The height and width should be between 0 and 1.
- Typically, we predict the square root of the height and width.
  - In this way, small deviations matter more (i.e. have a higher loss) when the bounding box is small.

Bounding Boxes (Extra)

In the loss function we will use the following

$$\left(\sqrt{\hat{w}} - \sqrt{w}\right)^2$$

to penalize errors when predicting the relative width of the bounding box.

Bounding Boxes (Extra)

On the next slide you can see that a difference of 10 pixels matters more (larger loss) when the bounding box is 50 pixels wide than when it is 200 pixels wide when using the formula with the square root. (And the same for a difference of 50 pixels).

Bounding Boxes (Extra)

Picture width (in pixels)	400	400	400	400
Bounding box width (in pixels)	200	50	200	50
w (relative width BB)	0,5	0,125	0,5	0,125
Difference in predicted pixels	10	10	50	50
Predicted with in pixels	210	60	250	100
pred w (predicted relative width)	0,525	0,15	6,25E-01	2,50E-01
(pred w - w)^2	6,25E-04	6,25E-04	1,56E-02	1,56E-02
sqrt(w)	7,07E-01	3,54E-01	7,07E-01	3,54E-01
sqrt(pred w)	7,25E-01	3,87E-01	7,91E-01	5,00E-01
(sqrt(pred w) - sqrt(w))^2	3,05E-04	1,14E-03	6,97E-03	2,14E-02

Bounding boxes example

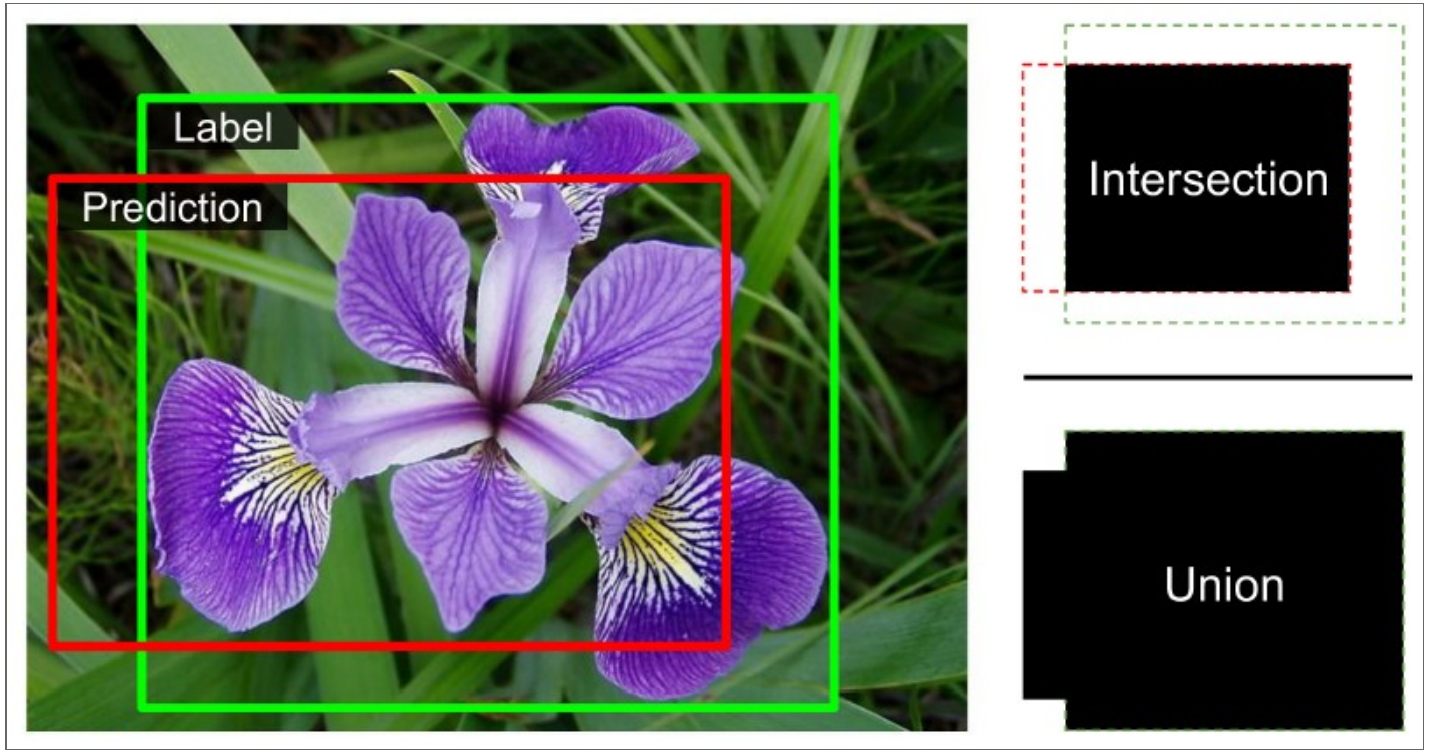
Intersection over Union

- The model is trained using MSE loss for the bounding boxes.
  - However, MSE loss is not a good *metric* for evaluating the model.
  - Compare with classification: we use cross entropy loss for training, but accuracy for evaluation.
- Implemented in `tf.keras.metrics.MeanIoU`.

Intersection over Union

- **Intersection over Union** (IoU) is the most common metric for evaluating object localization.
  - IoU: the area of overlap (intersection) of the actual and predicted bounding box divided by the area of their union.
  - If the bounding boxes do not overlap at all: IoU equals zero.
  - When the bounding boxes are identical: IoU equals one.

Intersection over Union



Intersection over Union

14.10 Object Detection

Object Detection

- **Object detection** is the task of classifying and localizing *multiple objects* in an image.
- Older approach:
  - Use CNN trained to classify and locate a single object (roughly in the center of the image)
  - CNN was also trained to output an *objectness score*: the estimated probability that there is an object centered near the middle.
  - Slide the CNN across the image and make predictions at each step.

Sliding-CNN Approach

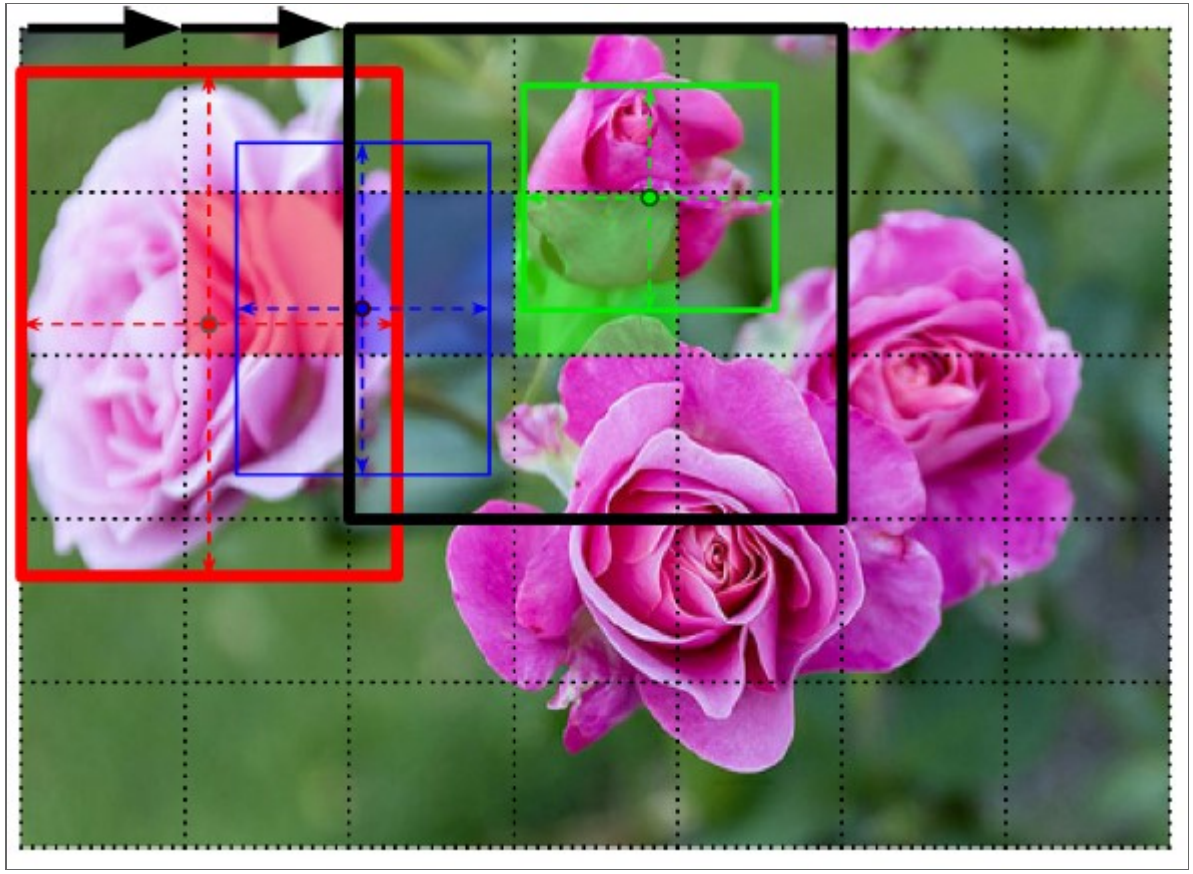


Image chopped in 5 by 7 grid. Sliding-CNN across 3 by 3 regions

Non-Max Suppression

- The sliding-CNN approach is straightforward, but the same object can be detected *multiple times*.
- **Non-max suppression** gets rid of unnecessary bounding boxes.
  - First, remove all bounding boxes with a low objectness score (below some threshold).
  - Next, find the bounding box with the highest objectness score, and remove all other bounding boxes that overlap a lot with it (e.g. IoU more than 60%).
  - Repeat until there are no more bounding boxes to remove.

Sliding-CNN Approach

- The sliding-CNN approach works quite well, but it requires running the CNN many times over the image.
  - This can be slow.

14.10.1 Fully Convolutional Networks

Fully Convolutional Network

- Up till now, we have used CNNs with dense layers at the end.
- In 2015, Long et al. realised that you can *replace* the dense layers with convolutional layers.
  - This is called a **fully convolutional network** (FCN).

Conv instead of Dense Layer

- Suppose we have a CNN with a dense layer at the end.
  - The dense layer has 200 neurons.
  - The input to the dense layer is a (flattened) feature map with 7x7x100 neurons.
  - Each of the 200 neurons in the dense layer computes a weighted sum (plus bias term) of the 7x7x100=4900 activations in the feature map.

Conv instead of Dense Layer (Ctd.)

- Instead of the dense layer, use a convolutional layer with
  - 200 filters
  - kernel size of 7x7 (i.e. equal to the spatial dimensions of input feature map)
  - “valid” padding.
  - Each of the 200 filters computes a weighted sum (plus bias term) of the 7x7x100=4900 activations in the feature map.

Conv instead of Dense Layer (Ctd.)

- Only difference between Dense and convolutional layer.
  - Output of dense layer: `(batch_size, 200)`
  - Output of convolutional layer: `(batch_size, 1, 1, 200)`

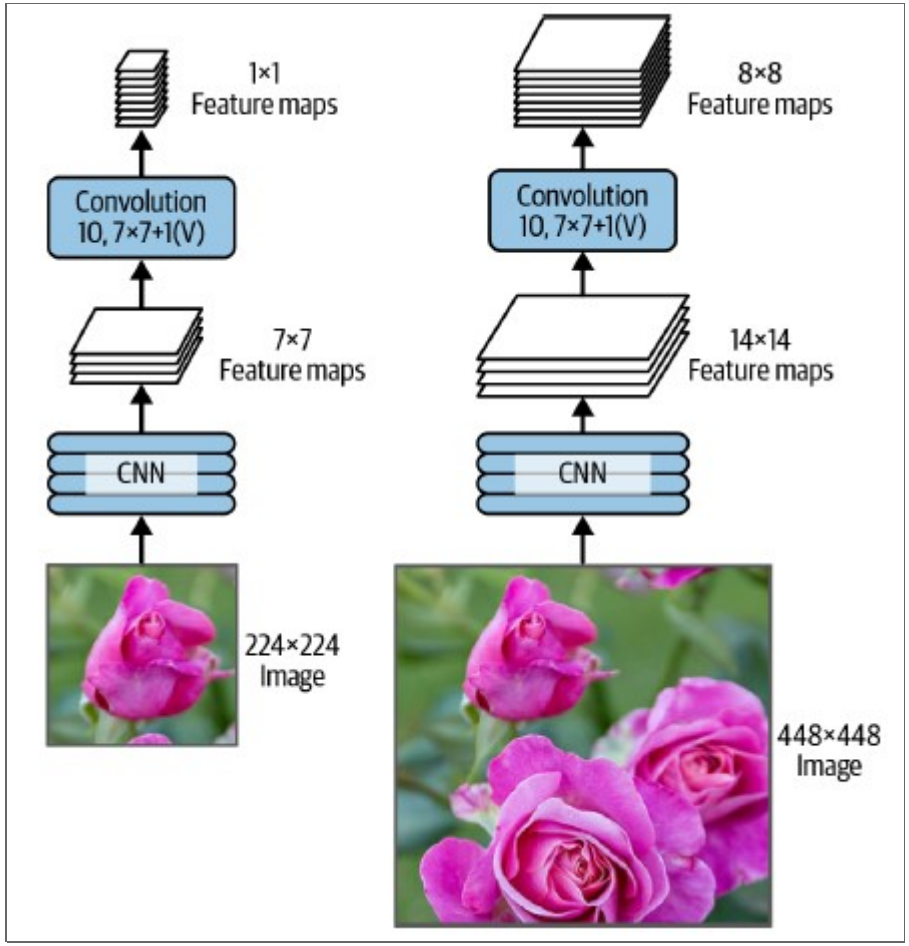
Advantage of Conv Layer

- The advantage of the convolutional layer (and also a pooling layer) is that it can be applied to images of any input size as long as the number of channels is correct.
  - The dense layer requires a fixed input size.
- A **fully convolutional network** (FCN) consists of convolutional and pooling layers only.

Fully Convolutional Network

A fully convolutional network can process images of any size!

Fully Convolutional Network



Fully convolutional network processing a small and large image

Fully Convolutional Network

- Suppose one prediction consists of 10 numbers (which is why we used 10 filters in the last convolutional layer).
- When we process the 448x448 image, we will get 64 (8 times 8) predictions.
  - This is like using the original CNN and sliding it across the images using 8 steps per row and 8 steps per column.

14.10.2 You Only Look Once

YOLO

- **YOLO** (You Only Look Once) is an object detection architecture proposed in 2015.
  - YOLO is fast: it can be run in real time on video.
- YOLO's architecture is very similar to what we've discussed, but with some differences.
  - For each grid cell, YOLO considers only objects whose center lies within the cell.
  - YOLO outputs two bounding boxes per cell (instead of one). This allows it to detect two objects in the same cell.
  - YOLO outputs 20 class probabilities per grid cell.
- Over the years many improvements were made to the original YOLO.

14.11 Object Tracking

Object Tracking

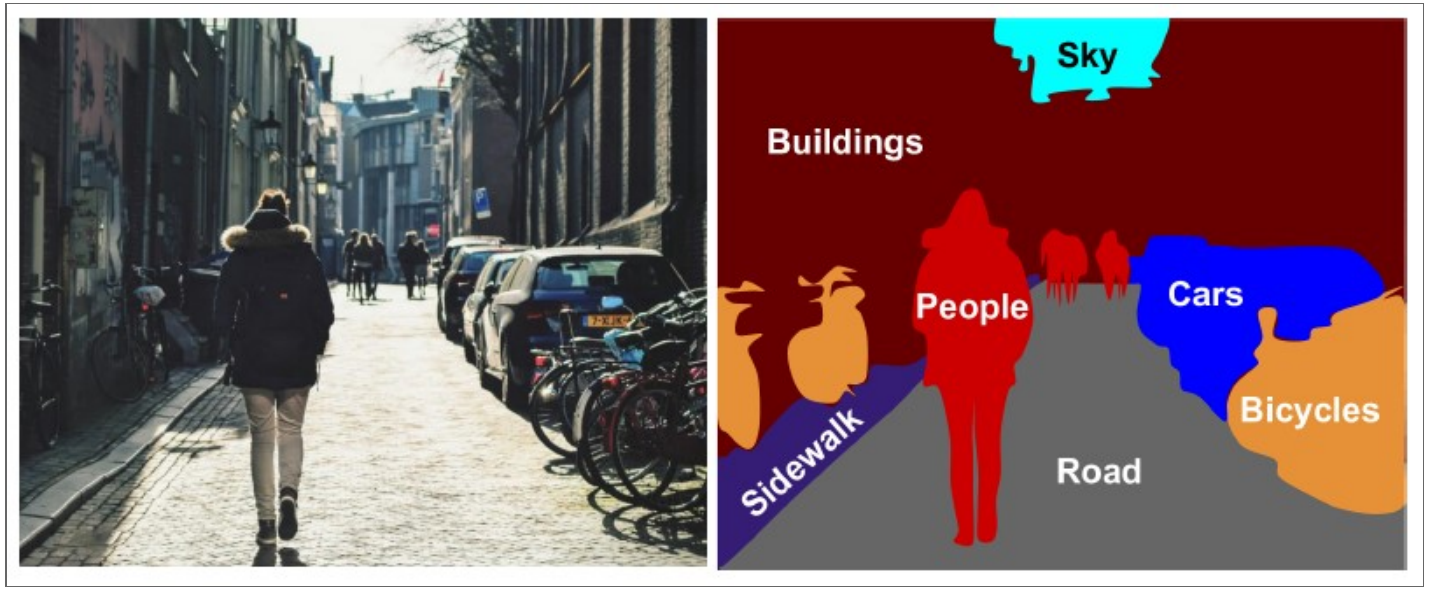
- **Object tracking** means detecting and tracking objects over time.
  - E.g. tracking a person in a video.
- Combinations of “classical” algorithms and deep learning algorithms are used.

14.12 Semantic Segmentation

Semantic Segmentation

- In **semantic segmentation** each *pixel* in the image is classified according to the class it belongs to.
- In the example on the next slide, different instances of the same class are not distinguished.
  - All the bicycles are classified as “bicycle”.

Semantic Segmentation



Semantic segmentation

Instance Segmentation

- In **instance segmentation** objects of the same class are not merged, but are distinguished from each other.
  - E.g. each individual bicycle is classified as a separate instance.
- The Mask R-CNN architecture is a popular architecture for instance segmentation.