Introduction to Deep Learning for Computer Vision

Professor M. Shahidur Rahman

DoCSE, SUST

rahmanms@sust.edu

Outline

- Understanding convolutional neural networks (convnets)
- Using data augmentation to mitigate overfitting
- Using a pretrained convnet to do feature extraction
- Fine-tuning a pretrained convent

Convolutional neural networks

- Computer vision is the problem domain that led to the initial rise of deep learning between 2011 and 2015
- *Convolutional neural networks* started getting remarkably good results on image classification
  - the ICDAR 2011 Chinese character recognition competition
  - Hinton's group winning the high-profile ImageNet large-scale visual recognition challenge 2012
- Convolutional neural networks (*convnets)*, the type of deep learning model that is now used almost universally in computer vision applications.

Introduction to convnets

- In chapter 2, using a densely connected network where our test accuracy then was 97.8%.
- Even though the convnet will be basic, its accuracy will blow our densely connected model from chapter 2 out of the water.
- We'll build the model using the Functional API (instead of the previous sequential model)

Instantiating a small convnet

```
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)
```

Training the convnet on MNIST images

```
from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype("float32") / 255
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

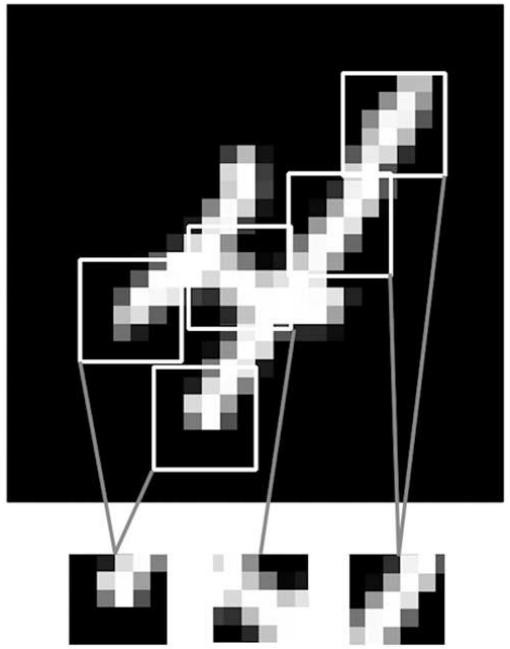
Evaluating the convnet

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print(f"Test accuracy: {test_acc:.3f}")
Test accuracy: 0.991
```

- Whereas the densely connected model from chapter 2 had a test accuracy of 97.8%, the basic convnet has a test accuracy of 99.1%
- We decreased the error rate by about 60% (relative).

The convolution operation

- The fundamental difference between a densely connected layer and a convolution layer:
  - Dense layers learn global patterns in their input feature space (for example, for a MNIST digit, patterns involving all pixels),
  - Convolution layers learn local patterns—in the case of images, patterns found in small 2D windows of the inputs

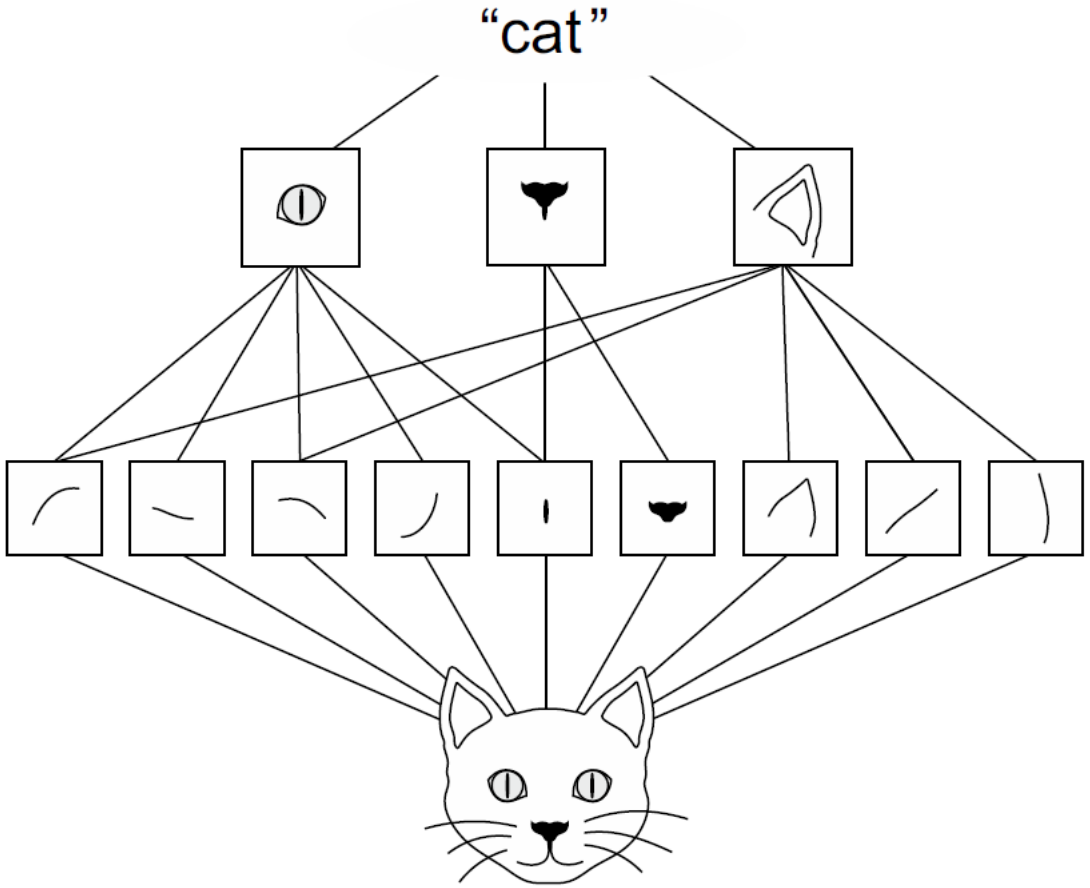


The convolution operation…

This key characteristic gives convnets two interesting properties:

- *The patterns they learn are translation-invariant*. After learning a certain pattern in the lower-right corner of a picture, a convnet can recognize it anywhere: for example, in the upper-left corner.
  - A densely connected model would have to learn the pattern anew if it appeared at a new location. This makes convnets data-efficient: they need fewer training samples to learn representations that have generalization power.
- *They can learn spatial hierarchies of patterns*. A first convolution layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of the features of the first layers, and so on.
  - This allows convnets to efficiently learn increasingly complex and abstract visual concepts, because *the visual world is fundamentally spatially hierarchical*.

The convolution operation…

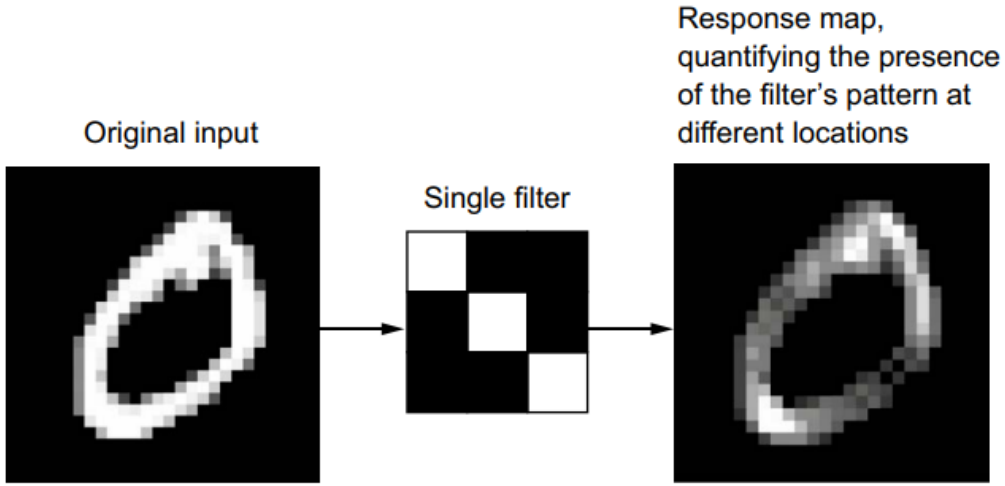


The convolution operation…

- Convolutions operate over rank-3 tensors called feature maps, with two spatial axes (height and width) as well as a depth axis (also called the channels axis).
- For an RGB image, the dimension of the depth axis is 3, because the image has three color channels: red, green, and blue. For a black-and-white picture, like the MNIST digits, the depth is 1 (levels of gray).
- The convolution operation extracts patches from its input feature map and **applies the same transformation to all of these patches**, producing an output feature map. This output feature map is still a rank-3 tensor: it has a width and a height.
- Its depth can be arbitrary, because the output depth is a parameter of the layer, and the different channels in that depth axis no longer stand for specific colors as in RGB input; rather, stand for filters. Filters encode specific aspects of the input data: at a high level, a single filter could encode the concept “presence of a face in the input,” for instance.

The convolution operation…

- In the MNIST example, the first convolution layer takes a feature map of size (28, 28, 1) and outputs a feature map of size (26, 26, 32): it computes 32 filters over its input.
- Each of these 32 output channels contains a 26 × 26 grid of values, which is a response map of the filter over the input, indicating the response of that filter pattern at different locations in the input



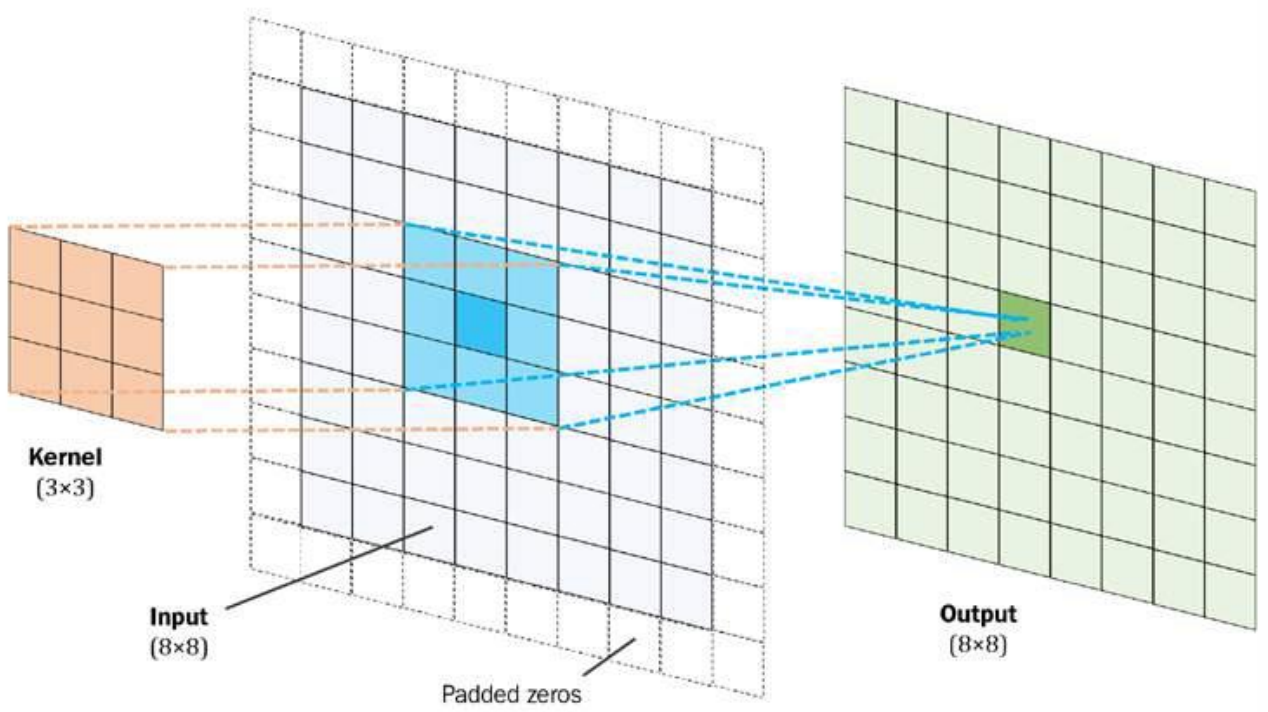
The convolution operation…

Convolutions are defined by two key parameters:

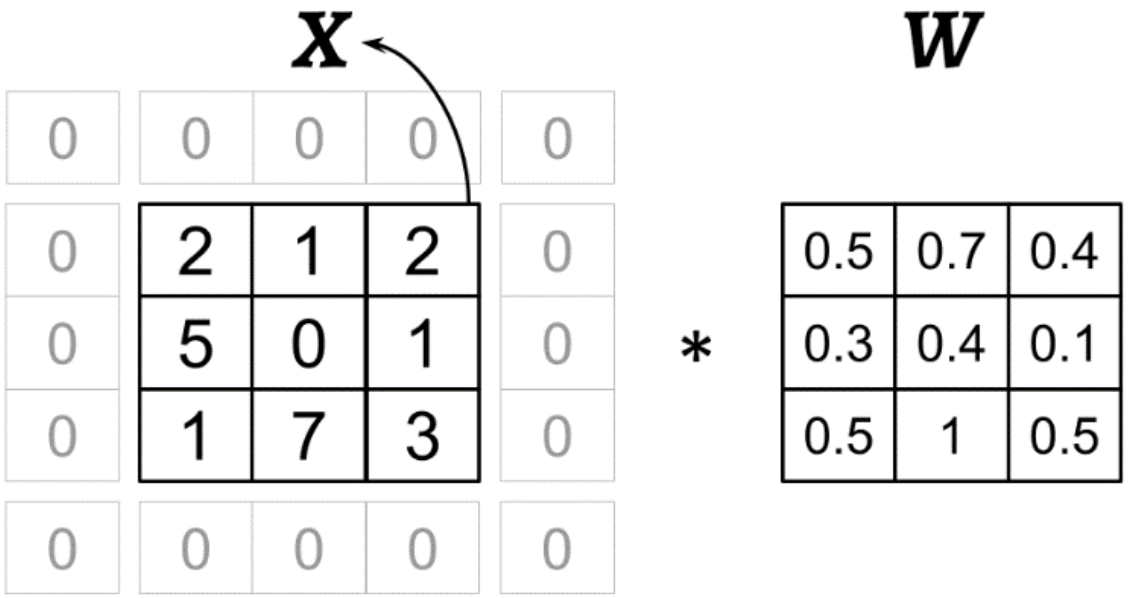
- Size of the patches extracted from the inputs—These are typically 3 × 3 or 5 × 5. In the example, they were 3 × 3, which is a common choice.
- Depth of the output feature map—This is the number of filters computed by the convolution.
- In Keras Conv2D layers, these parameters are the first arguments passed to the layer: Conv2D(output_depth, (window_height, window_width)).

How convolution works (Input depth = 1)

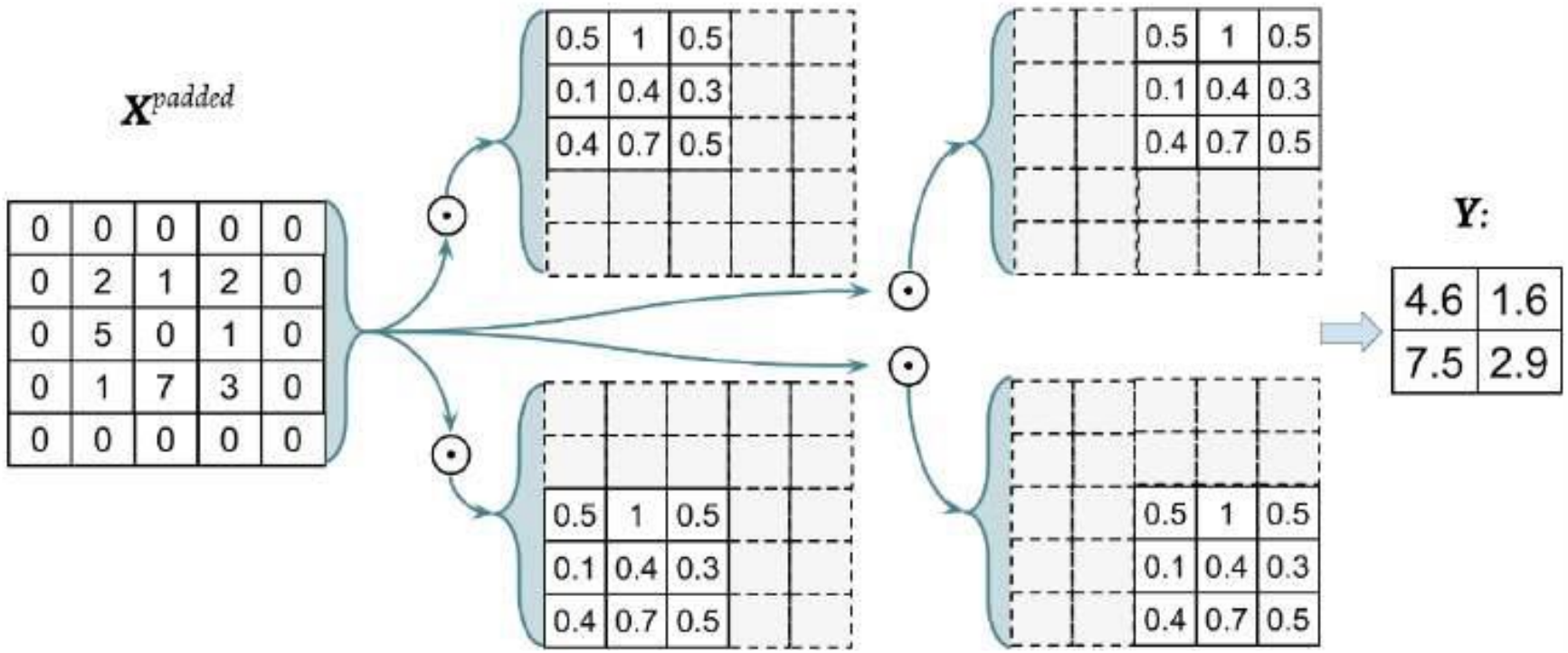
- A convolution works by sliding these windows of size 3 × 3 or 5 × 5 over the input feature map, stopping at every possible location, and extracting the patch of surrounding features.



Convolution: Sum of element-wise products



2D Convolution

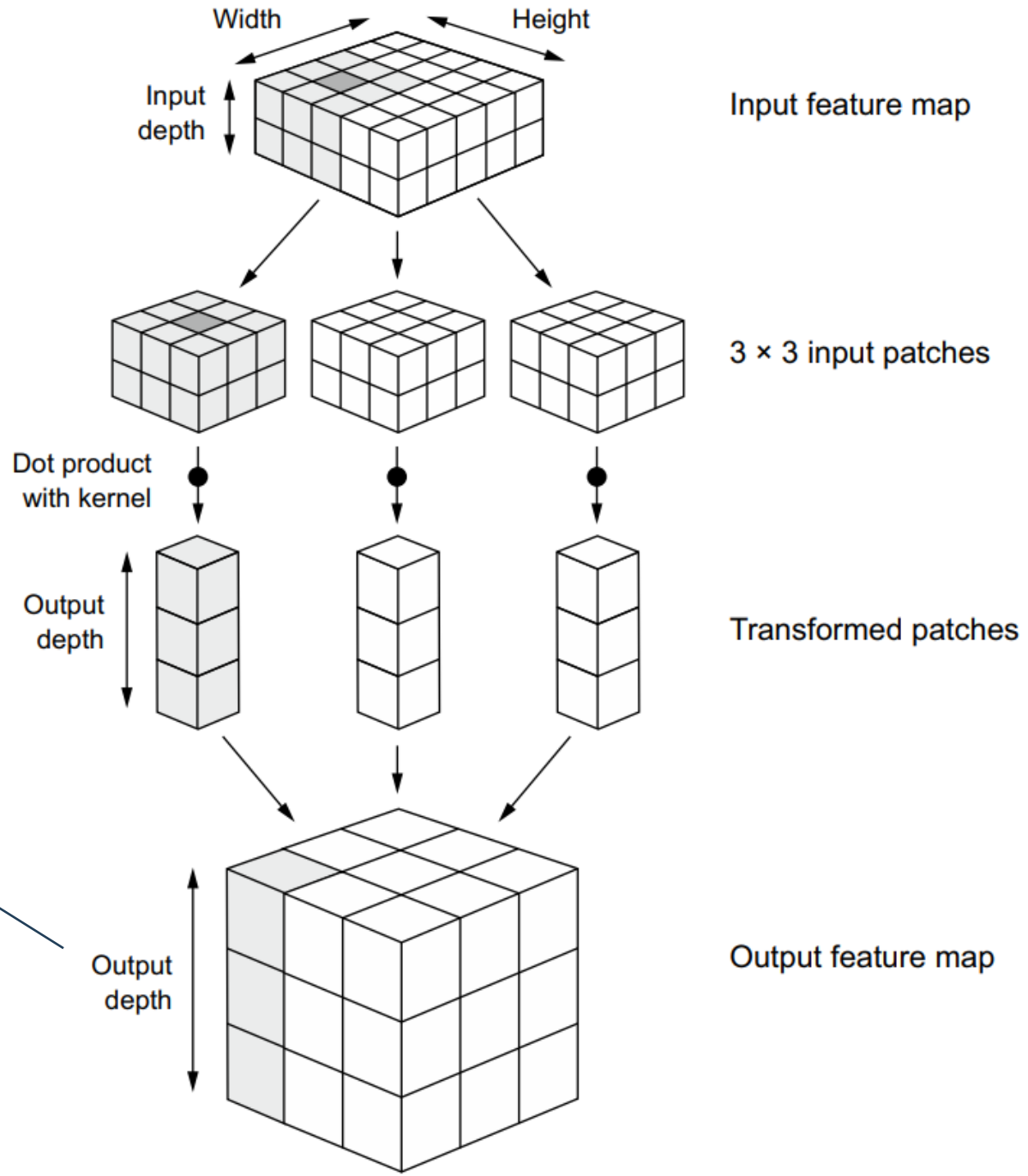


Convolution: Compute the sum of the element-wise product, which is denoted by the ⊙ operator.

2D Convolution (Input Depth > 1)

- A convolution works by sliding these windows of size 3 × 3 or 5 × 5 over the 3D input feature map, stopping at every possible location, and extracting the 3D patch of surrounding features (shape (window_height, window_width, input_depth)).
- Each such 3D patch is then transformed into a 1D vector of shape (output_depth), which is done via a tensor product with a **learned** weight matrix, called the convolution kernel— the same kernel is reused across every patch.
- All of these vectors (one per patch) are then spatially reassembled into a 3D output map of shape (height, width, output_ depth).
- Every spatial location in the output feature map corresponds to the same location in the input feature map (for example, the lower-right corner of the output contains information about the lower-right corner of the input).

How convolution works

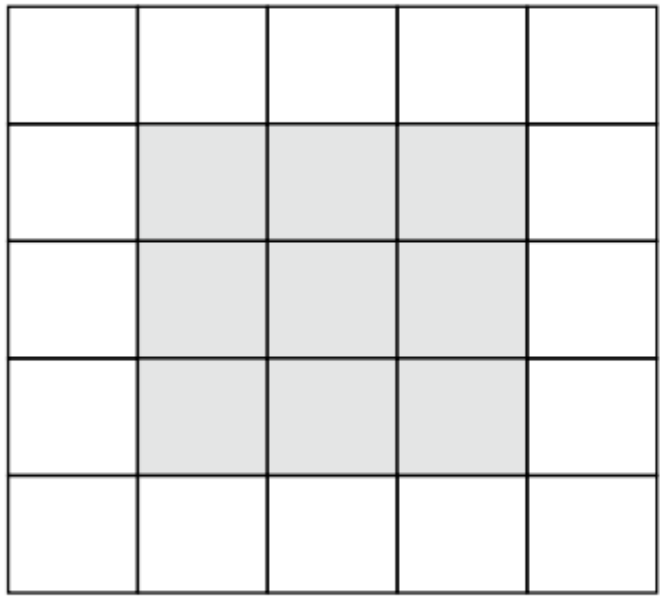


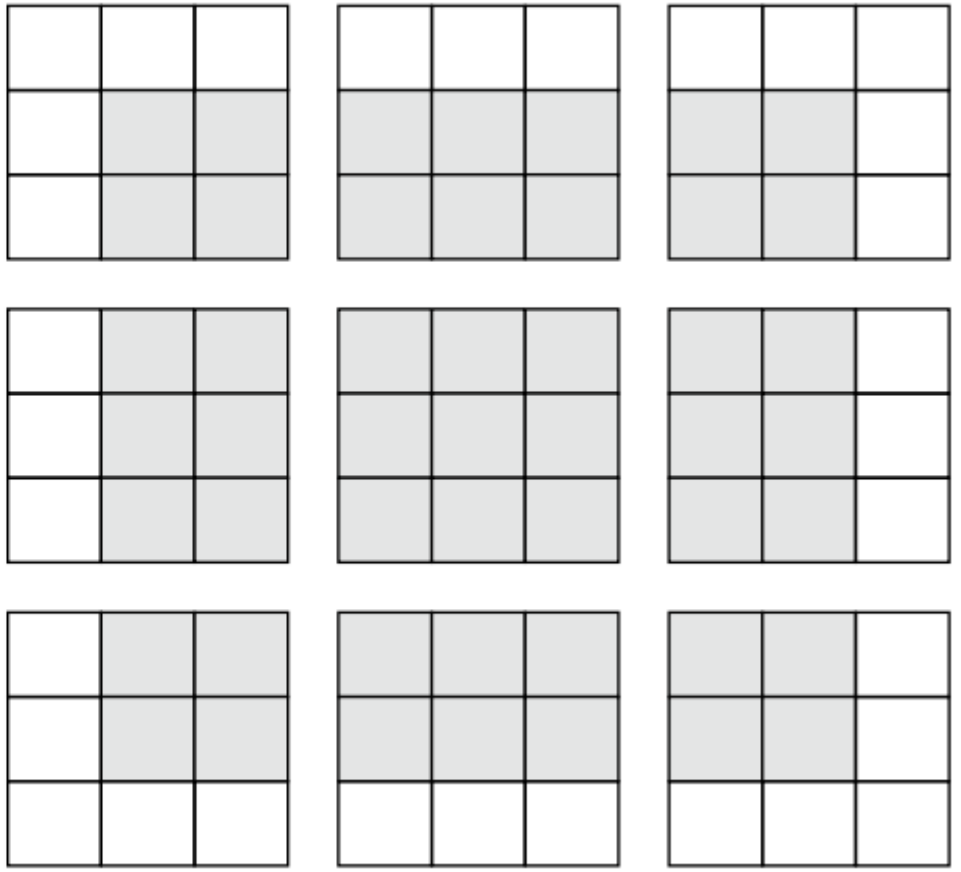
No. of Filters = No. of output feature maps = Output depth

Understanding Border Effects

- The output width and height may differ from the input width and height for two reasons:
  - Border effects, which can be countered by padding the input feature map
  - The use of strides
- Consider a 5 × 5 feature map (25 tiles total). There are only 9 tiles around which you can center a 3 × 3 window, forming a 3 × 3 grid. Hence, the output feature map will be 3 × 3.
- It shrinks by exactly two tiles alongside each dimension, in this case. You can see this border effect in action in the earlier example: you start with 28 × 28 inputs, which become 26 × 26 after the first convolution layer.

Valid locations of 3 × 3 patches in a 5 × 5 input feature map

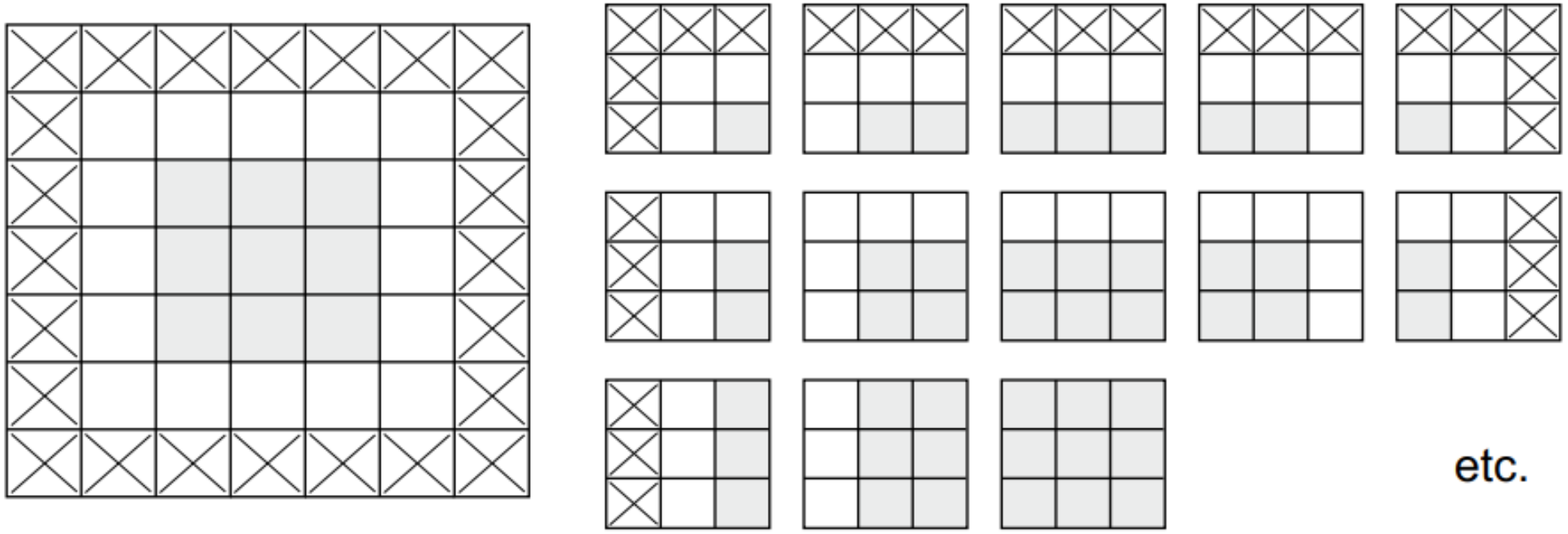




Understanding Border affects and Padding

- If you want to get an output feature map with the same spatial dimensions as the input, you can use padding.
  - Padding consists of adding an appropriate number of rows and columns on each side of the input feature map so as to make it possible to fit center convolution windows around every input tile.
  - For a 3 × 3 window, you add one column on the right, one column on the left, one row at the top, and one row at the bottom.
- In Conv2D layers, padding is configurable via the padding argument, which takes two values:
  - "valid", which means no padding (only valid window locations will be used), and
  - "same", which means “pad in such a way as to have an output with the same width and height as the input.” The padding argument defaults to "valid".

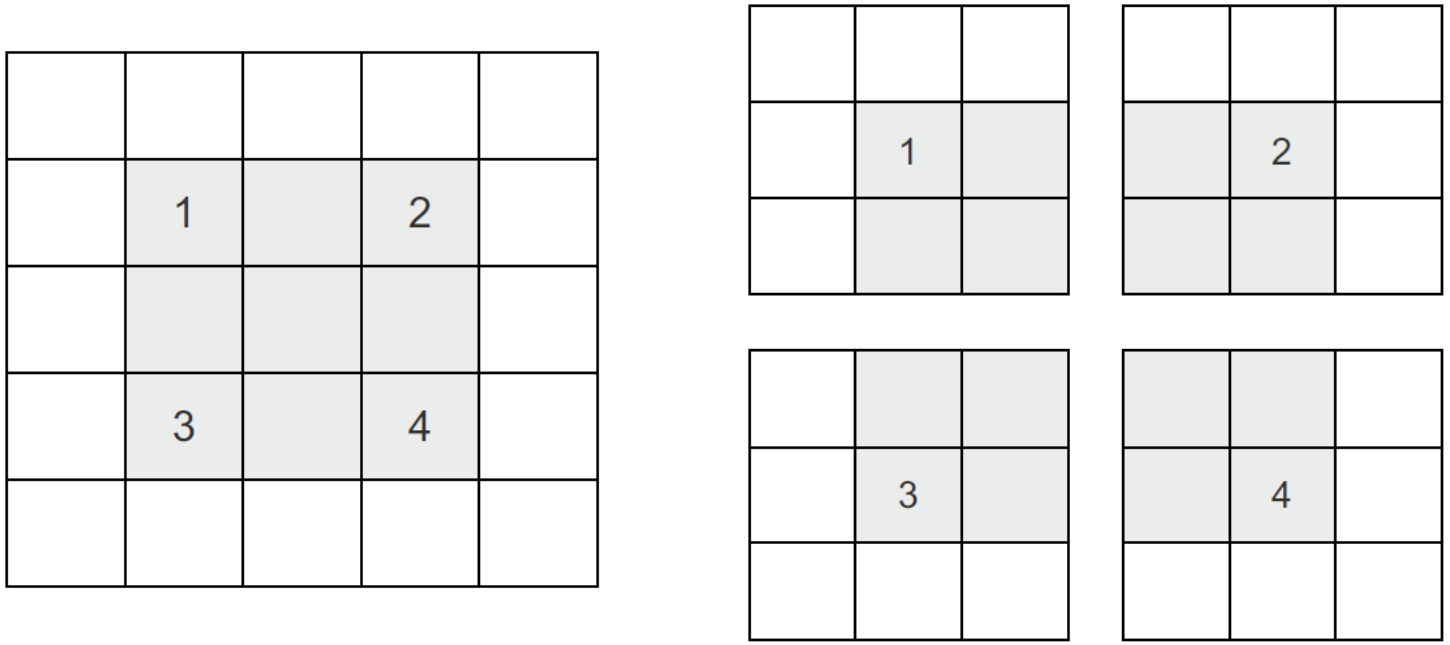
Padding a 5 × 5 input in order to be able to extract 25 3 × 3 patches



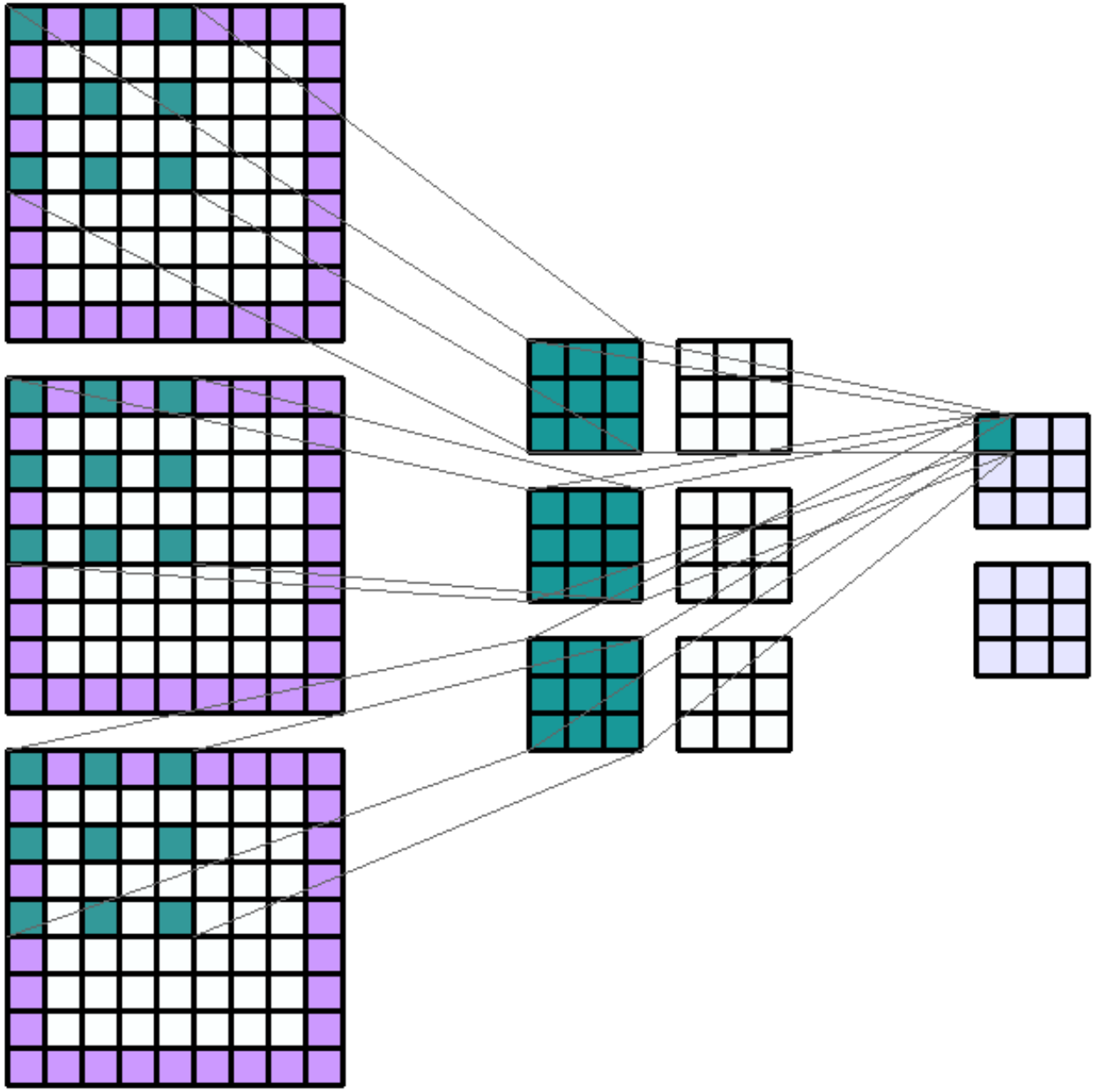
Understanding Convolution Strides

- The other factor that can influence output size is the notion of *strides*.
- The distance between two successive windows is a parameter of the convolution, called its *stride*, which defaults to 1. It's possible to have *strided convolutions*: convolutions with a stride higher than 1.
  - The patches extracted by a 3 × 3 convolution with stride 2 over a 5 × 5 input (without padding) is shown here.
- Strided convolutions are rarely used in classification models, but they come in handy for some types of models (discussed later)
- In classification models, instead of strides, we tend to use the *max-pooling* operation to downsample feature maps.

3 × 3 convolution patches with 2 × 2 strides



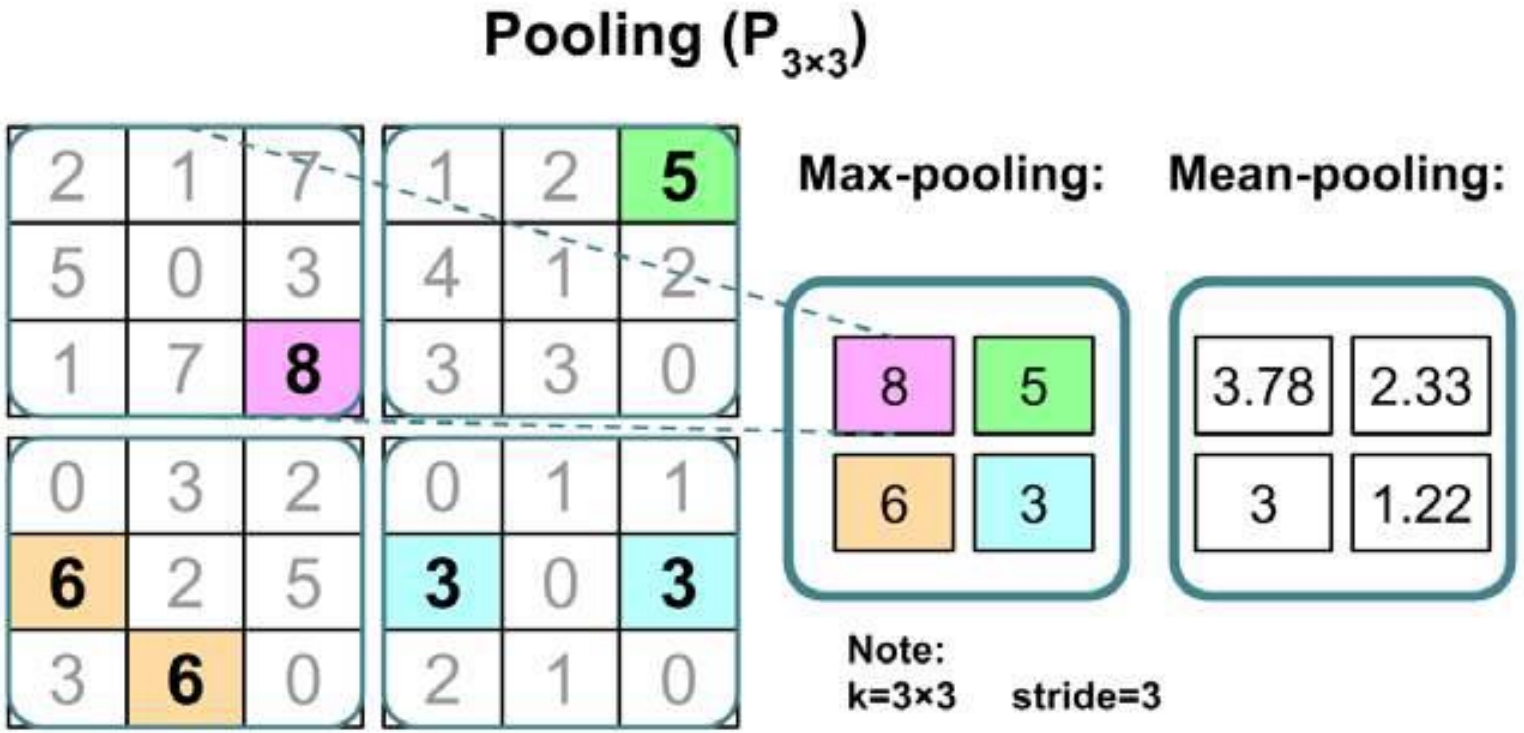
Input depth = 3
No. of filters =
Output depth = 2
Strides = 2 x 2



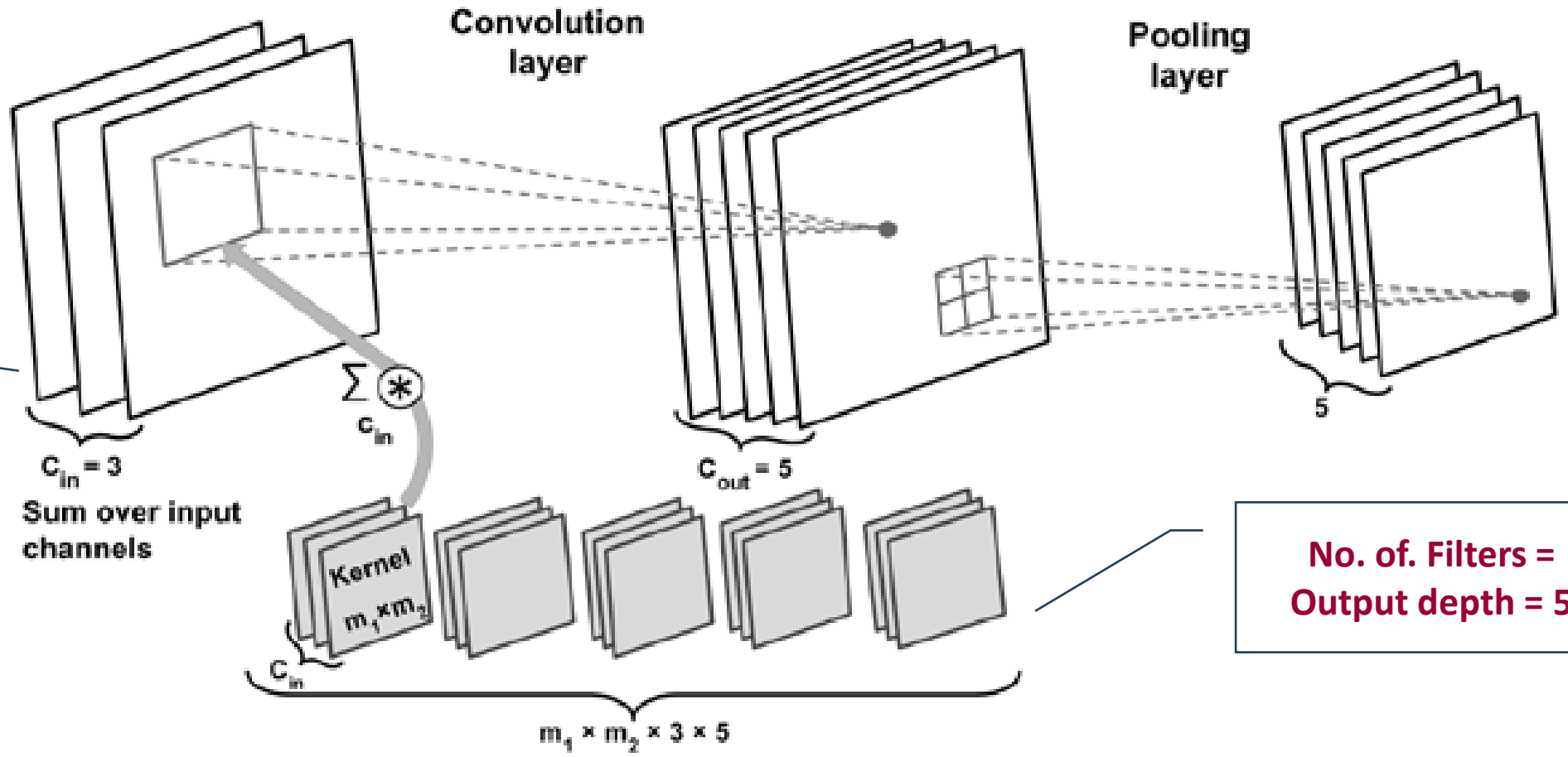
The max-pooling operation

- Max pooling consists of extracting windows from the input feature maps and outputting the max value of each channel.
- A big difference from convolution is that max pooling is usually done with 2 × 2 windows and stride 2, in order to downsample the feature maps by a factor of 2. On the other hand, convolution is typically done with 3 × 3 windows and no stride (stride 1).
- Max pooling isn't the only way to downsampling. Other than strides, you can use average pooling, where each local input patch is transformed by taking the average value of each channel over the patch.
- Max pooling is more informative to look at the maximal presence of different features than at their average presence.
- The most reasonable subsampling strategy is to first produce dense maps of features (via unstrided convolutions) and then look at the maximal activation of the features over small patches, rather than looking at sparser windows of the inputs (via strided convolutions) or averaging input patches, which could cause you to miss or dilute feature-presence information.

Max/ Mean Pooling



A typical Convolution cycle



Input depth = 3

No. of. Filters = Output depth = 5

Overlapping versus non-overlapping pooling

- Traditionally, Pooling is typically performed on nonoverlapping neighborhoods, which can be done by setting the stride parameter equal to the pooling size.
- For example, a non-overlapping pooling layer, $Pn_1 \times n_2$, requires a stride parameter $ss = (n_1, n_2)$.
- On the other hand, overlapping pooling occurs if the stride is smaller than the pooling size.

Training a convnet from scratch on a small dataset

- We'll focus on classifying images as dogs or cats in a dataset containing 5,000 pictures of cats and dogs (2,500 cats, 2,500 dogs).
- We'll use 2,000 pictures for training, 1,000 for validation, and 2,000 for testing.
- We'll start by naively training a small convnet on the 2,000 training samples, without any regularization, to set a baseline for what can be achieved. This will get us to a classification accuracy of about 70%.
- At that point, the main issue will be overfitting. Then we'll introduce *data augmentation*, a powerful technique for mitigating overfitting in computer vision. By using data augmentation, we'll improve the model to reach an accuracy of 80–85%.

The relevance of deep learning for small-data problems

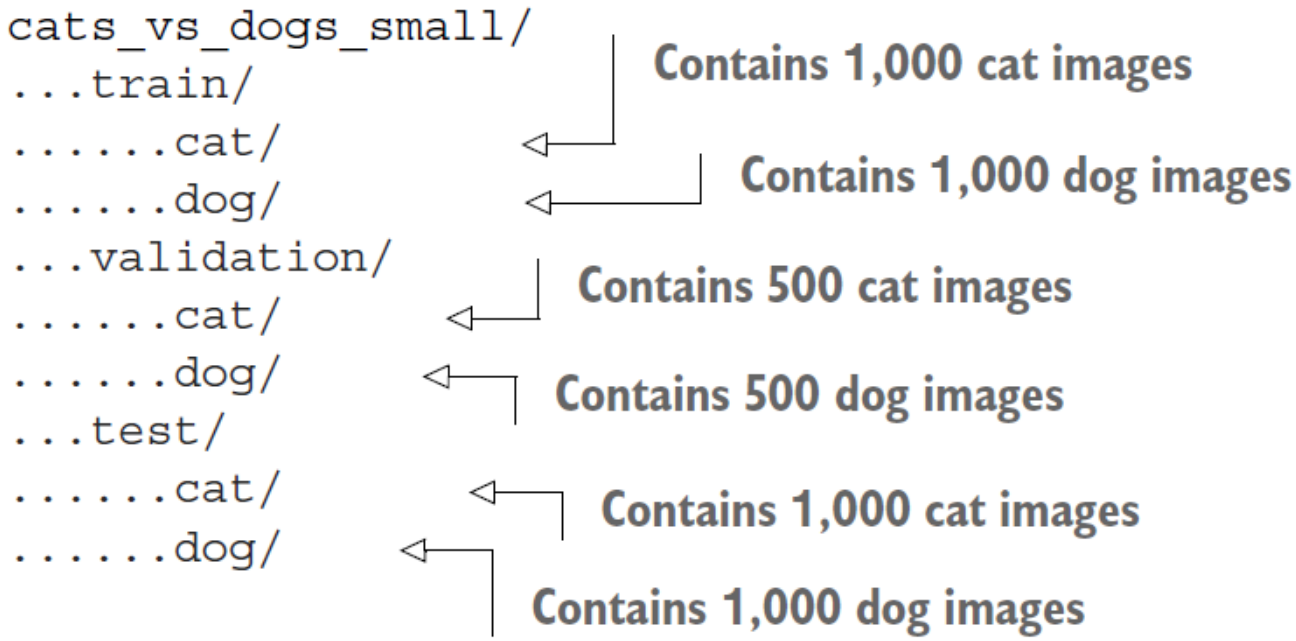
- What qualifies as “enough samples” to train a model is relative
- It isn’t possible to train a convnet to solve a complex problem with just a few tens of samples, but a few hundred can potentially suffice if the model is small and well regularized and the task is simple.
- Convnets learn local, translation-invariant features, they’re highly data-efficient on perceptual problems.
- Training a convnet from scratch on a very small image dataset will yield reasonable results despite a relative lack of data, without the need for any custom feature engineering.
- What’s more, deep learning models are by nature highly repurposable: an image-classification or speech-to-text model trained on a large-scale dataset (pre-trained models) can be reused it on a significantly different problem with only minor changes.

Downloading the data

- The Dogs vs. Cats dataset that we will use isn't packaged with Keras. It was made available by Kaggle as part of a computer vision competition in late 2013.
- You can download the original dataset from www.kaggle.com/c/dogs-vs-cats/data. You can also use the Kaggle API to download the dataset in Colab.
- This dataset contains 25,000 images of dogs and cats (12,500 from each class) and is 543 MB (compressed).
- We'll create a new dataset containing three subsets: a training set with 1,000 samples of each class, a validation set with 500 samples of each class, and a test set with 1,000 samples of each class. (Having more data available would make the problem easier, so it's good practice to learn with a small dataset.)

Samples from the Dogs vs. Cats dataset. Sizes weren't modified: the samples come in different sizes, colors, backgrounds, etc.

Directory Structure of the Subsets



```
import os, shutil, pathlib

original_dir = pathlib.Path("train")
new_base_dir = pathlib.Path("cats_vs_dogs_small")

def make_subset(subset_name, start_index, end_index):
    for category in ("cat", "dog"):
        dir = new_base_dir / subset_name / category
        os.makedirs(dir)
        fnames = [f"{category}.{i}.jpg"
                  for i in range(start_index, end_index)]
        for fname in fnames:
            shutil.copyfile(src=original_dir / fname,
                            dst=dir / fname)

make_subset("train", start_index=0, end_index=1000)
make_subset("validation", start_index=1000, end_index=1500)
make_subset("test", start_index=1500, end_index=2500)
```

Path to the directory where the original dataset was uncompressed

Directory where we will store our smaller dataset

Create the training subset with the first 1,000 images of each category.

Create the validation subset with the next 500 images of each category.

Create the test subset with the next 1,000 images of each category.

Utility function to copy cat (and dog) images from index start_index to index end_index to the subdirectory new_base_dir/{subset_name}/cat (and /dog). The "subset_name" will be either "train", "validation", or "test".

Building the model

- Unlike the first example, we're dealing with bigger images and a more complex problem, we'll make our model larger, accordingly: it will have two more Conv2D and MaxPooling2D stages.
- This serves both to augment the capacity of the model and to further reduce the size of the feature maps so they aren't overly large when we reach the Flatten layer.
- Here, because we start from inputs of size 180 pixels × 180 pixels (a arbitrary choice), we end up with feature maps of size 7 × 7 just before the Flatten layer.
- As it's a binary-classification problem, we'll end the model with a single unit (a Dense layer of size 1) and a sigmoid activation.
- We will start the model with a Rescaling layer, which will rescale image inputs (from originally in the [0, 255] range) to the [0, 1] range.

A small convnet for dogs vs. cats classification

```
from tensorflow import keras
from tensorflow.keras import layers

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

The model expects RGB images of size 180 × 180.

Rescale inputs to the [0, 1] range by dividing them by 255.

```
>>> model.summary()
Model: "model_2"
```

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 180, 180, 3)]	0
rescaling (Rescaling)	(None, 180, 180, 3)	0
conv2d_6 (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d_2 (MaxPooling2	(None, 89, 89, 32)	0
conv2d_7 (Conv2D)	(None, 87, 87, 64)	18496
max_pooling2d_3 (MaxPooling2	(None, 43, 43, 64)	0
conv2d_8 (Conv2D)	(None, 41, 41, 128)	73856
max_pooling2d_4 (MaxPooling2	(None, 20, 20, 128)	0
conv2d_9 (Conv2D)	(None, 18, 18, 256)	295168
max_pooling2d_5 (MaxPooling2	(None, 9, 9, 256)	0
conv2d_10 (Conv2D)	(None, 7, 7, 256)	590080
flatten_2 (Flatten)	(None, 12544)	0
dense_2 (Dense)	(None, 1)	12545

Configuring the model for training

```
model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              metrics=["accuracy"])
```

Data preprocessing

- As we know, data should be formatted into preprocessed floatingpoint tensors before being fed into the model. The steps for getting the JPEG files into the model are roughly as follows:
  1. Read the picture files.
  2. Decode the JPEG content to RGB grids of pixels.
  3. Convert these into floating-point tensors.
  4. Resize them to a shared size (we'll use 180 × 180).
  5. Pack them into batches (we'll use batches of 32 images).
- Keras features the utility function image_dataset_from_directory(), which sets up a data pipeline that can automatically turn image files on disk into batches of preprocessed tensors.

Using image_dataset_from_directory to read images

```
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)
```

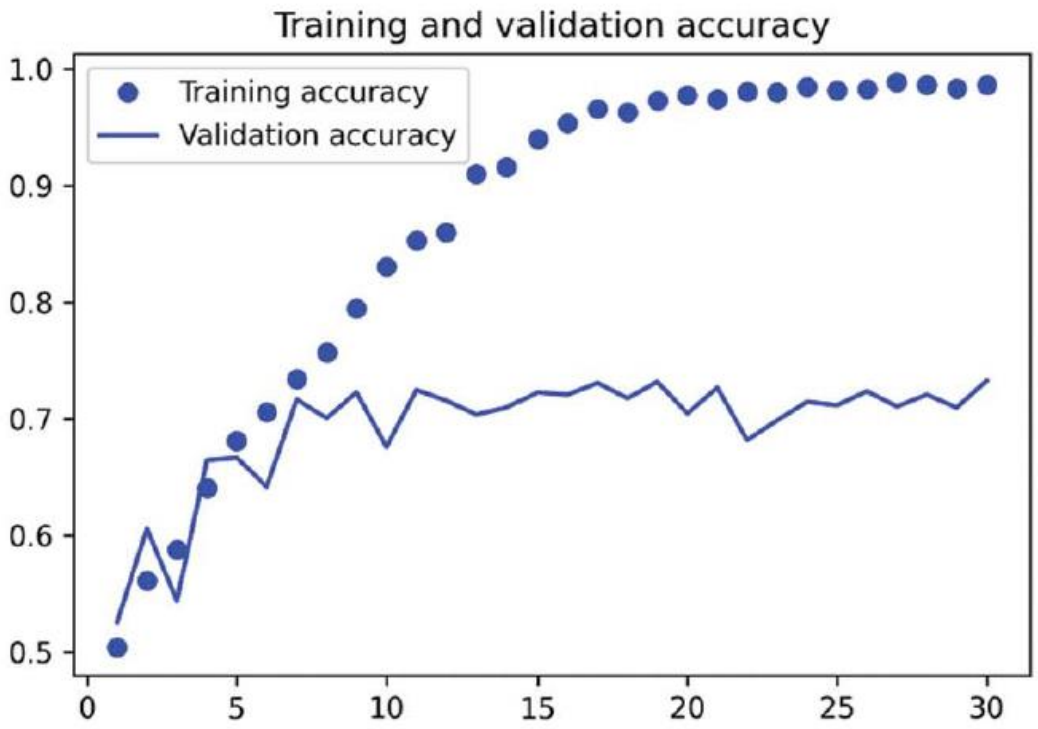
Fitting the model using a **Dataset**

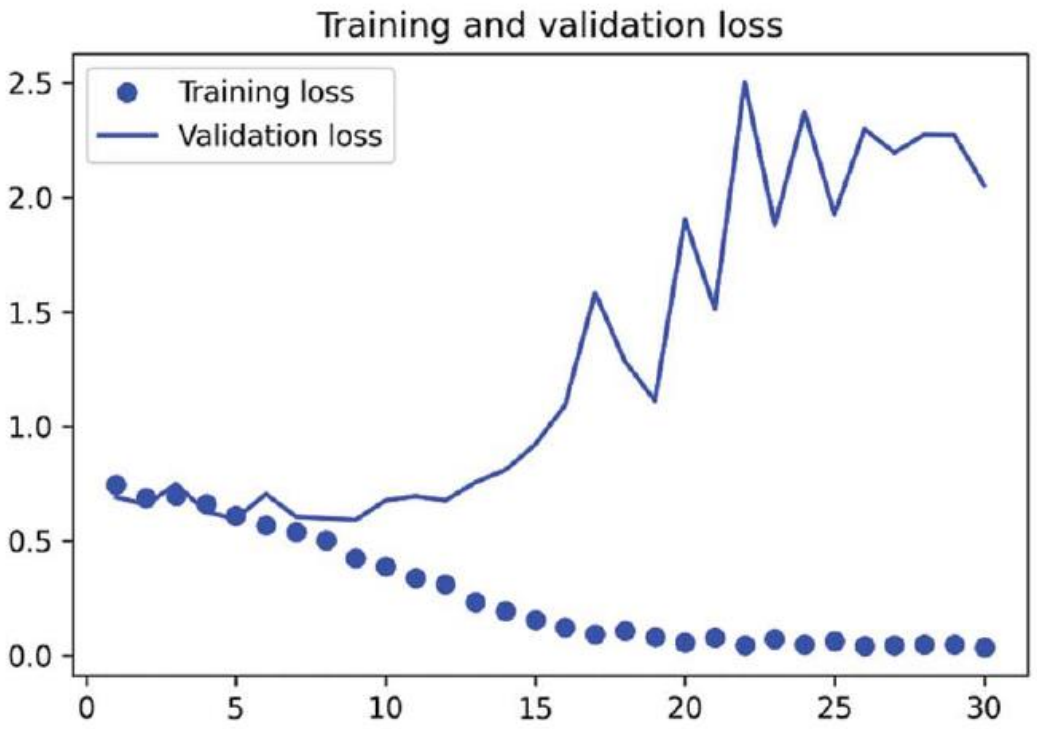
```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch.keras",
        save_best_only=True,
        monitor="val_loss")
]

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

Displaying curves of loss and accuracy during training

```
import matplotlib.pyplot as plt
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)
plt.plot(epochs, accuracy, "bo", label="Training accuracy")
plt.plot(epochs, val_accuracy, "b", label="Validation accuracy")
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 metrics for a simple convnet





- Overfitting is apparent. The training accuracy increases linearly over time, until it reaches nearly 100%, whereas the validation accuracy peaks at 75%.

Evaluating the model on the test set

```
test_model = keras.models.load_model("convnet_from_scratch.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

- We get a test accuracy of 69.5%. (Due to the randomness of neural network initializations, you may get numbers within one percentage point of that.)
- Because we have relatively few training samples (2,000), overfitting will be our number one concern.
- We will now work with a overfitting technique specific to computer vision and used almost universally when processing images with deep learning models: *data augmentation*

Using data augmentation

- Data augmentation takes the approach of generating more training data from existing training samples by *augmenting* the samples via a number of random transformations that yield believable-looking images.
- The goal is that, at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data so it can generalize better.
- In Keras, this can be done by adding a number of *data augmentation layers* at the start of your model (before the rescaling layer).

Define a data augmentation stage to add to an image model

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

- RandomFlip("horizontal")—Applies horizontal flipping to a random 50% of the images that go through it
- RandomRotation(0.1)—Rotates the input images by a random value in the range [–10%, +10%] (these are fractions of a full circle—in degrees, the range would be [–36 degrees, +36 degrees])
- RandomZoom(0.2)—Zooms in or out of the image by a random factor in the range [-20%, +20%]

Displaying some randomly augmented training images

```
plt.figure(figsize=(10, 10))
for images, _ in train_dataset.take(1):
    for i in range(9):
        augmented_images = data_augmentation(images)
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(augmented_images[0].numpy().astype("uint8"))
        plt.axis("off")
```

We can use take(N) to only sample *N* batches from the dataset. This is equivalent to inserting a break in the loop after the *N*th batch.

Apply the augmentation stage to the batch of images.

Display the first image in the output batch. For each of the nine iterations, this is a different augmentation of the same image.

Generating variations of a dog
via random data augmentation

Adding dropout layer

- If we train a new model using this data-augmentation configuration, the model will never see the same input twice. But the inputs come from a small number of original images—we can't produce new information.
- As such, this may not be enough to completely get rid of overfitting.
- To further fight overfitting, we'll also add a Dropout layer to our model right before the densely connected classifier.

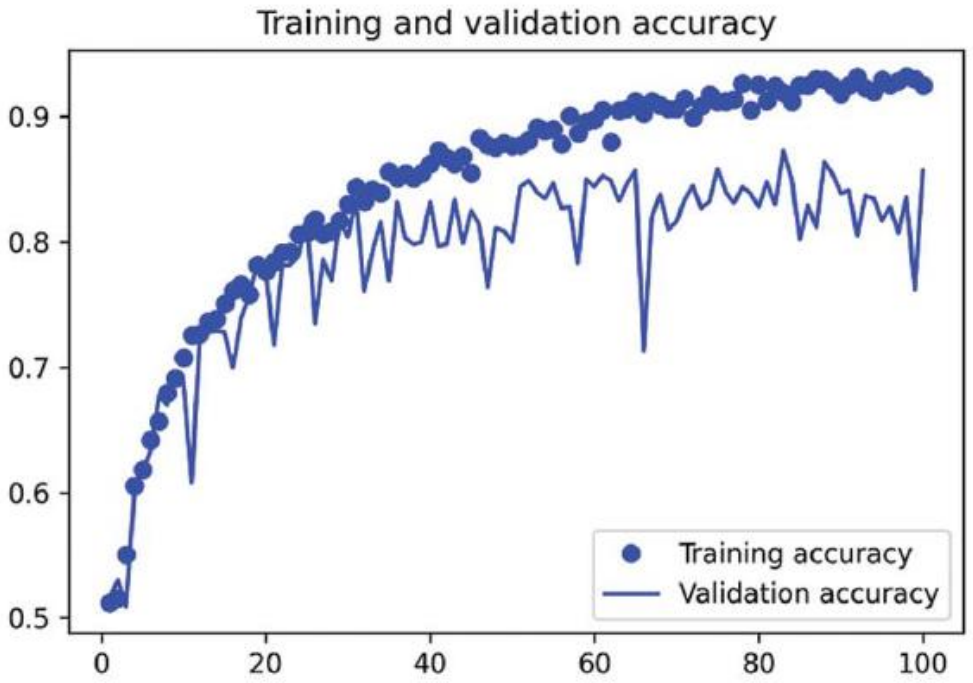
Applying image augmentation and dropout

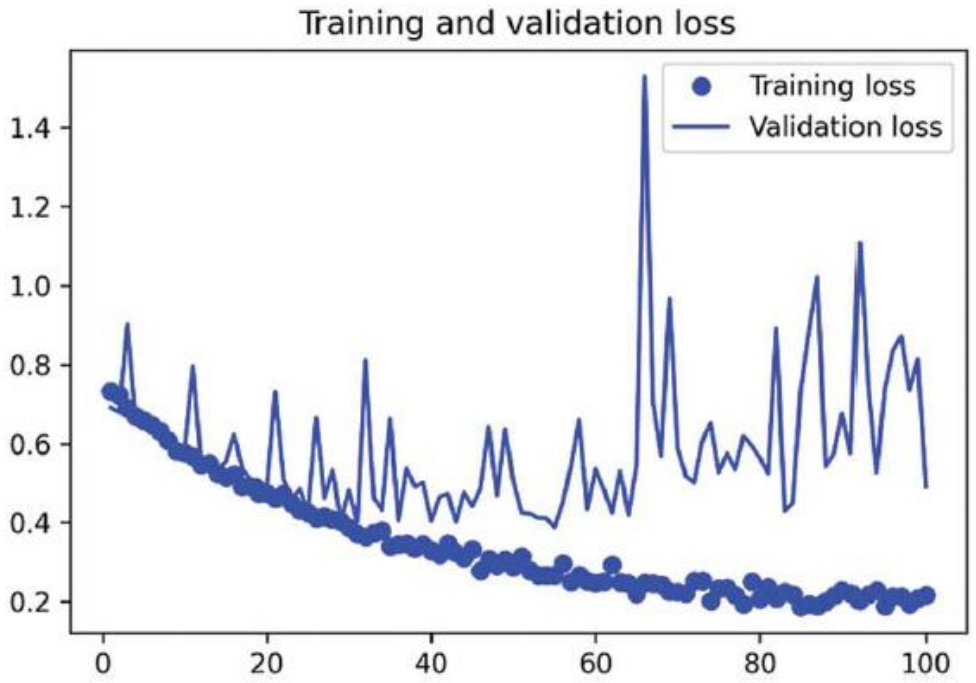
```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = layers.Rescaling(1./255)(x)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
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)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              metrics=["accuracy"])
```

Training the regularized convnet

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch_with_augmentation.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=100,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

Training and validation metrics with data augmentation





- Thanks to data augmentation and dropout, we start overfitting much later, around epochs 60–70 (compared to epoch 10 for the original model). The validation accuracy ends up consistently in the 80–85% range.

Evaluating the model on the test set

```
test_model = keras.models.load_model(
    "convnet_from_scratch_with_augmentation.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

- We get a test accuracy of 83.5%. It's starting to look good!
- By further tuning the model's configuration (such as the number of filters per convolution layer, or the number of layers in the model), we might be able to get an even better accuracy, likely up to 90% (difficult to go any higher with so little data).

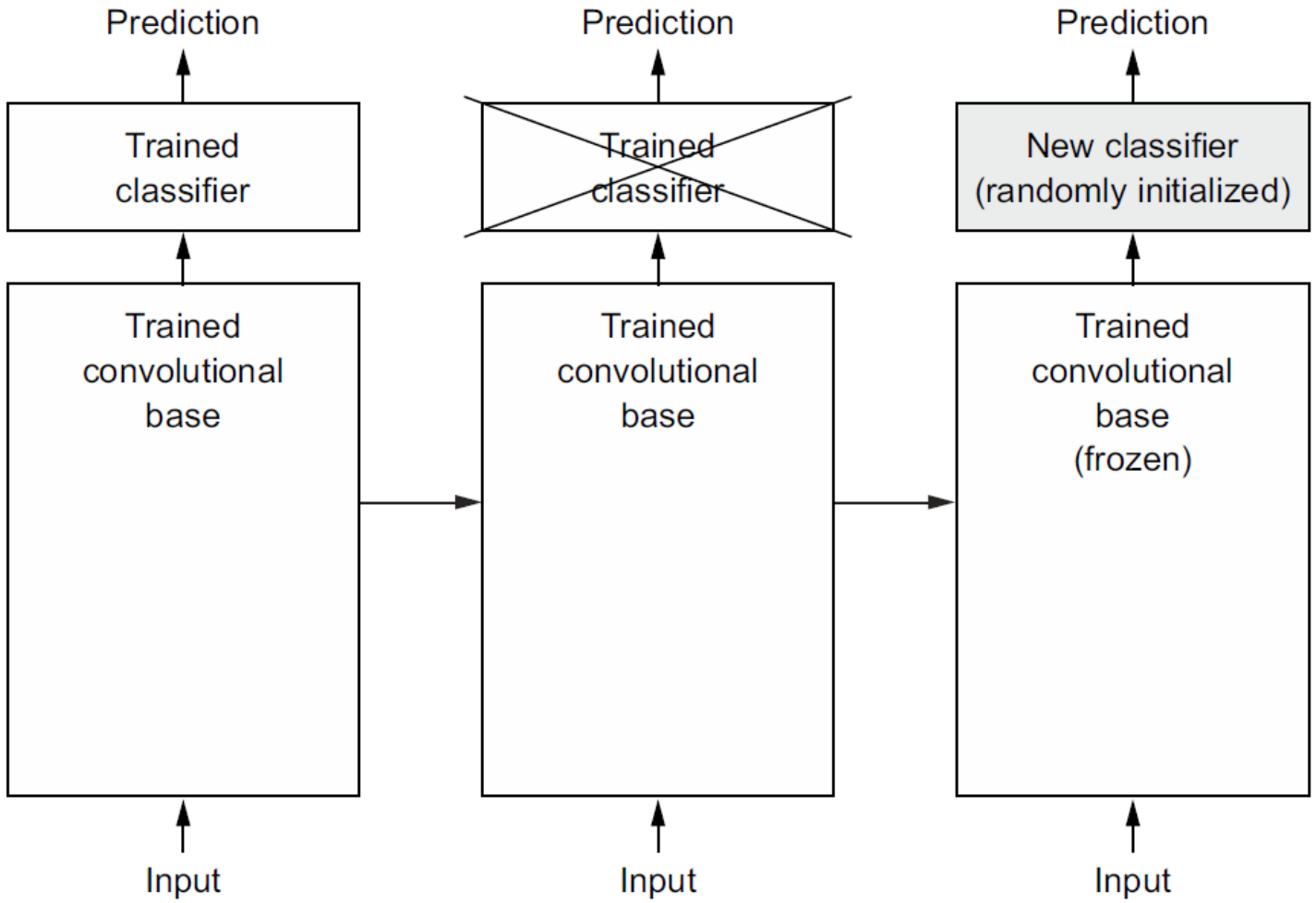
Leveraging a pretrained model

- A common and highly effective approach to deep learning on small image datasets is to use a pretrained model.
- A *pretrained model* is a model 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, the spatial hierarchy of features learned by the pretrained model can effectively act as a generic model of the visual world
- A large convnet trained on the ImageNet dataset (1.4 million labeled images and 1,000 different classes) containing many animal classes, including different species of cats and dogs, and you can thus expect it to perform well on the dogs-versus-cats classification problem.
- There are two ways to use a pretrained model: *feature extraction* and *fine-tuning*.

Feature extraction with a pretrained model

- Feature extraction consists of using the representations learned by a previously trained model to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.
- Previous convnet start with a series of pooling and convolution layers, and they end with a densely connected classifier. The first part is called the *convolutional base* of the model.
- In case of convnets, feature extraction consists of taking the convolutional base of a previously trained network, running the new data through it, and training a new classifier on top of the output.

Swapping classifiers while keeping the same convolutional base



Why only reuse the convolutional base?

- The representations learned by the convolutional base are likely to be more generic and more reusable regardless of the computer vision problem at hand
- The representations learned by the classifier will necessarily be specific to the set of classes on which the model was trained
- Additionally, representations found in densely connected layers no longer contain any information about where objects are located in the input image; these layers get rid of the notion of space, whereas the object location is still described by convolutional feature maps.
- For problems where object location matters, densely connected features are largely useless.

Why only reuse the convolutional base?...

- 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”).
- So 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.

Using VGG16 model as convolutional base

- VGG16 architecture, developed by Karen Simonyan and Andrew Zisserman in 2014 (an older but similar with our current model)
- The VGG16 model comes prepackaged with Keras (keras.applications). Many other image-classification models (all pretrained on the ImageNet dataset) are available as part of keras.applications:
  - Xception
  - ResNet
  - MobileNet
  - EfficientNet
  - DenseNet etc.

Instantiating the VGG16 convolutional base

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

Instantiating the VGG16 convolutional base

We pass three arguments to 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 we intend to use our own densely connected classifier (with only two classes: cat and dog), we don't need to include it.
- input_shape is the shape of the image tensors that we'll feed to the network. This argument is purely optional: if we don't pass it, the network will be able to process inputs of any size.

VGG16 Summary

```
>>> conv_base.summary()
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_19 (InputLayer)        [(None, 180, 180, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 180, 180, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 180, 180, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 90, 90, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 90, 90, 128)       73856
_________________________________________________________________
```

VGG16 Summary…

```
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 22, 22, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 11, 11, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 11, 11, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 11, 11, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 11, 11, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 5, 5, 512)         0
=================================================================
```

- The final feature map has shape (5, 5, 512). That's the feature map on top of which we'll stick a densely connected classifier.

There are two ways we could proceed:

- Run the convolutional base over our dataset, record its output to a NumPy array on disk, and then use this data as input to a standalone, densely connected classifier
  - This solution is fast and cheap to run, because it only requires running the convolutional base once for every input image. However, this technique won't allow us to use data augmentation.
- Extend the model we have (conv_base) by adding Dense layers on top, and run the whole thing from end to end on the input data.
  - This will allow us to use data augmentation, because every input image goes through the convolutional base every time it's seen by the model. But this technique is far more expensive than the first.

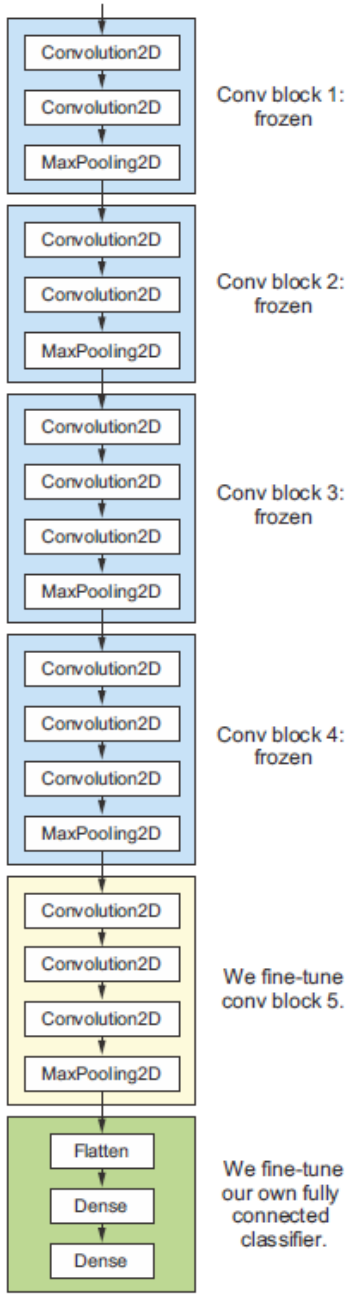
Fine-tuning a pretrained model

- 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.

Fine-tuning the last convolutional block of the VGG16 network

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

Freezing all layers until the fourth from the last



Fine-tuning a pretrained model…

The steps for fine-tuning a network are as follows:

- Add our custom network on top of an already-trained base network.
- Freeze the base network.
- Train the part we added.
- Unfreeze some layers in the base network. (Note that you should not unfreeze “batch normalization” layers, which are not relevant here since there are no such layers in VGG16.)
- Jointly train both these layers and the part we added.

Fine tune the Model

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

callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="fine_tuning.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

Evaluate this model on the test data

```
model = keras.models.load_model("fine_tuning.keras")
test_loss, test_acc = model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```