Machine Learning I (DATS 6202)

Convolutional Neural Network

Yuxiao Huang

Data Science, Columbian College of Arts & Sciences
George Washington University

Spring 2020

Reference

- This set of slices was largely built on the following 8 wonderful books and a wide range of fabulous papers:

HML	Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)
PML	Python Machine Learning (3rd Edition)
ESL	The Elements of Statistical Learning (2nd Edition)
PRML	Pattern Recognition and Machine Learning
LFD	Learning From Data
NND	Neural Network Design (2nd Edition)
NNDL	Neural Network and Deep Learning
RL	Reinforcement Learning: An Introduction

- For most materials covered in the slides, we will specify their corresponding books and papers for further reference.

Code Example & Case Study

- See code example of topics discussed in this section in github repository: /p2_c2_s1_convolutional_neural_network/code_example
- See case study of Kaggle Competition related to this section in github repository: /p2_c2_s1_convolutional_neural_network/case_study

Overview

1. Learning Objectives
2. The Architecture and Idea of CNN
3. Designing CNN
4. Implementing CNN
5. Transfer Learning using Pretrained Models

Learning Objectives

- It is **expected** to understand
  - the architecture and idea of Convolutional Neural Network (CNN)
  - the good practices for designing CNN
  - the idea of and good practices for transfer learning using pretrained CNN
- It is **recommended** to understand
  - the architecture and idea of the state-of-the-art pretrained CNN

Why CNN?

- Previously we discussed how to design, implement, train and fine-tune fully connected deep neural network (DNN).
- Can we apply such DNN to Computer Vision?
- Suppose:
  - the input image has 100×100 pixels
  - the first hidden layer of a DNN has 1000 perceptrons
- **Q:** What is the number of parameters (weights and biases) on the first hidden layer?

Why CNN?

- Previously we discussed how to design, implement, train and fine-tune fully connected deep neural network (DNN).
- Can we apply such DNN to Computer Vision?
- Suppose:
  - the input image has 100×100 pixels
  - the first hidden layer of a DNN has 1000 perceptrons
- **Q:** What is the number of parameters (weights and biases) on the first hidden layer?
- **A:**

$$p^1 p^0 + p^1 = 1000 \times 100 \times 100 + 1000 = 10^7 + 10^3, \tag{1}$$

where p^0 and p^1 are the number of perceptrons on the input and first hidden layer.

- That is, there are over 10 million parameters on the first hidden layer alone!
- As a result, fully connected DNN is too computationally expensive to be useful for computer vision.

Biological Neurons, Visual Cortex and Receptive Fields

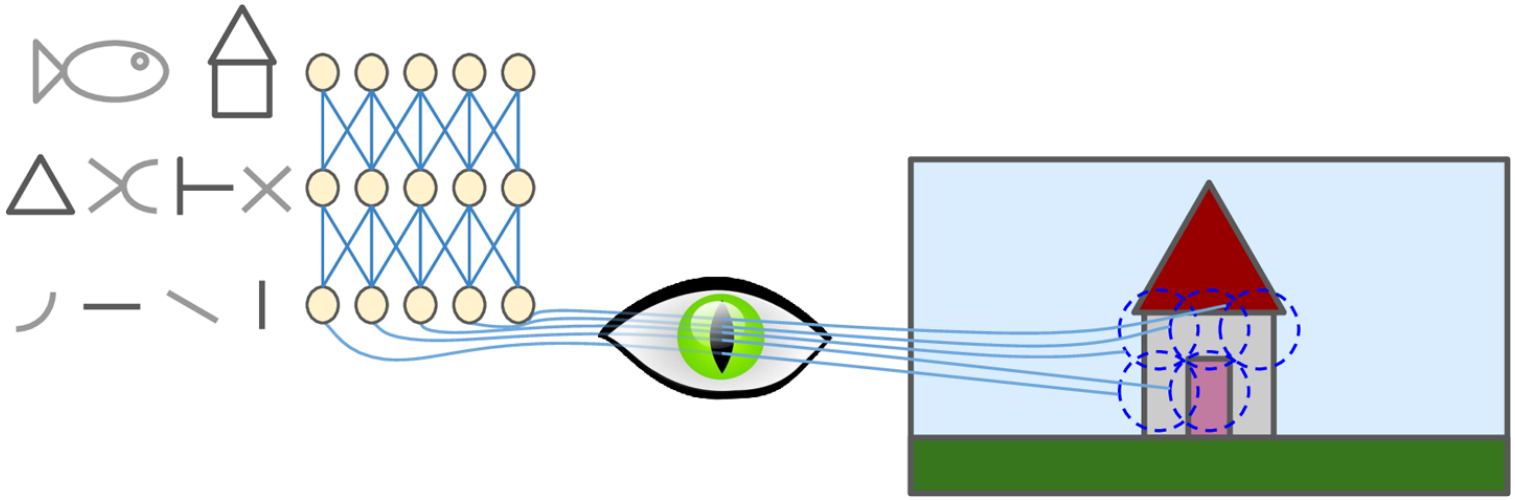

Figure 1: Biological neurons, visual cortex and receptive fields. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Each biological neuron on the bottom layer does not respond to every single pixel, but pixels in a specific area (blue dashed circles), named *Local Receptive Field*.
- The receptive field of each neuron may overlap.
- Biological neurons on the lower layer recognize lower-level (simple) patterns, whereas neurons on the higher layer recognize higher-level (complex) patterns.

Typical CNN Architecture

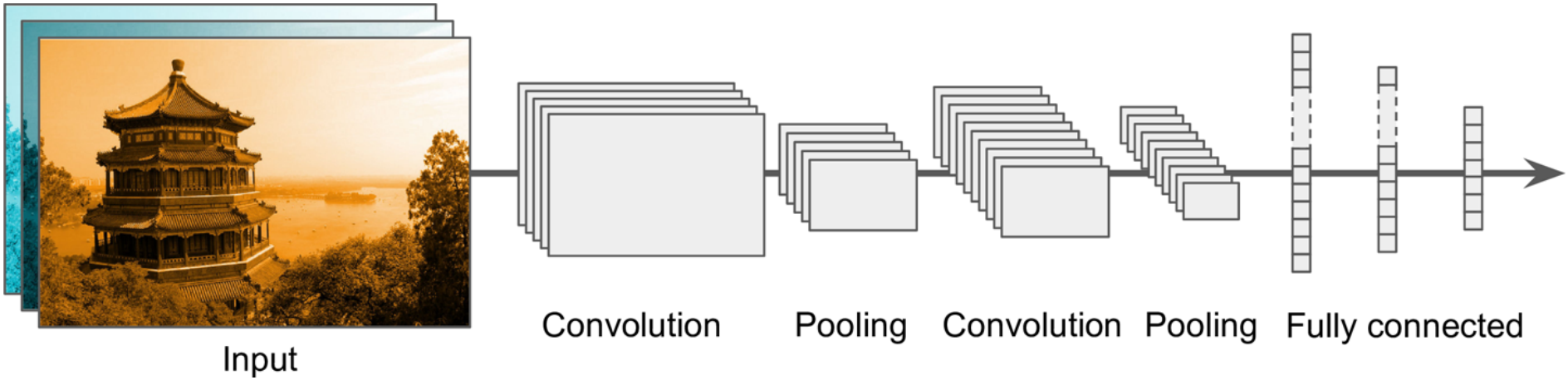

Figure 2: Typical CNN architecture. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Key components in CNN:
  - *Convolutional Layers* (more on this later)
  - *Pooling Layers* (more on this later)
  - fully connected layers

Convolutional Layer

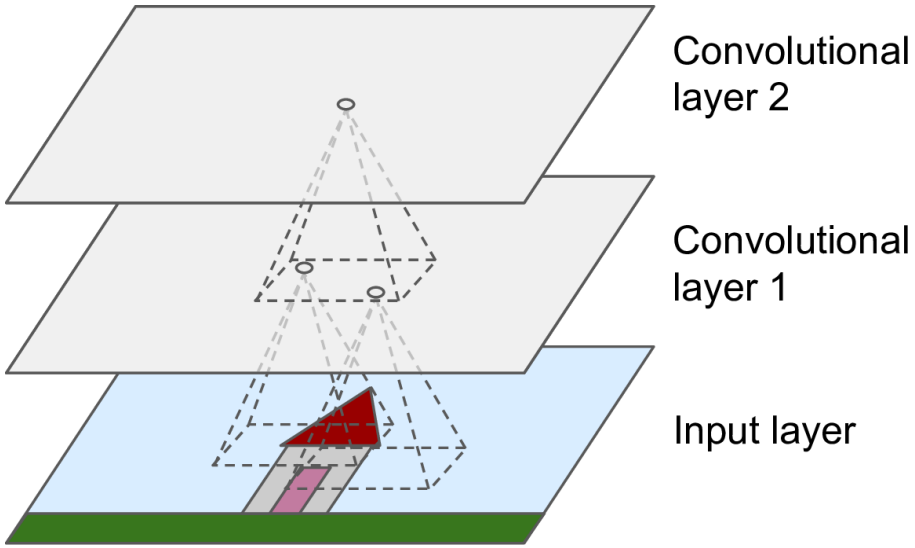

Figure 3: Convolutional layers. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Similar to biological neurons (see fig. 1), each perceptron on a convolutional layer is not connected to every single perceptron on the previous layer, but perceptrons in its receptive field. Moreover, the receptive field of each perceptron may overlap.
- This makes convolutional layers the most important building block in CNN, since they not only result in significantly fewer parameters (more on this later) but also allow capturing the hierarchical structure in real-world image.

Convolutional Layers

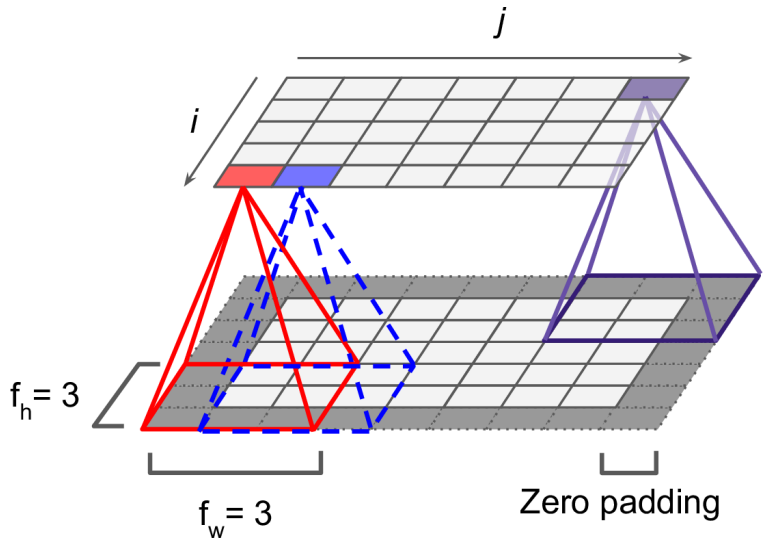

Figure 4: Adjacent layers in a CNN. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Each convolutional layer could comprise a 2d matrix of perceptrons (or even a 3d tensor of perceptrons, more on this later). Perceptron in row i, column j on layer k is only connected to perceptrons in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$ on layer $k - 1$, where f_h and f_w are the height and width of the receptive field.
- To make adjacent layers have the same number of rows and columns, we usually add zeros around a layer. This is called *Zero Padding* (more on this later).

Convolutional Layers

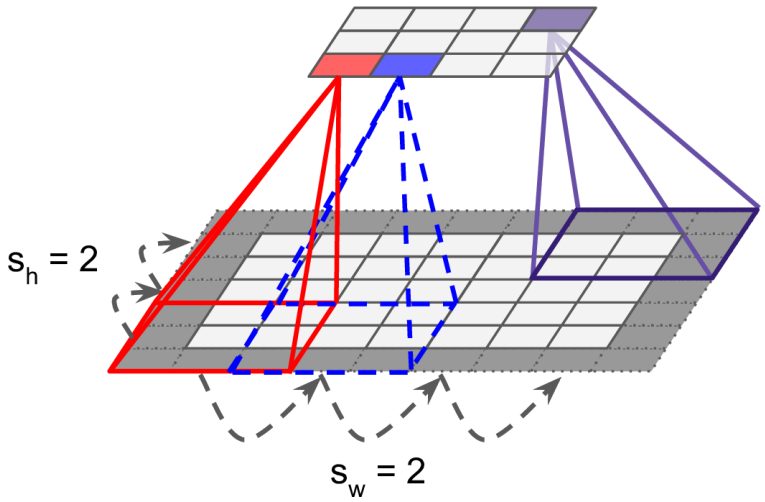

Figure 5: Adjacent layers in a CNN, with a stride of 2. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- The shift from one receptive field to the next is called the *Stride*.
- In fig. 4 the stride is 1, whereas in fig. 5 the stride is 2.
- Perceptron in row i, column j on layer k is connected to perceptrons in rows $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w$ to $j \times s_w + f_w - 1$ on layer $k - 1$, where f_h and f_w are the height and width of the receptive field, while s_h and s_w the vertical and horizontal strides (which may not necessarily be the same).
- Using a larger stride will reduce the number of rows and columns in a convolutional layer, which will significantly reduce the computational cost for training the CNN.

Filter and Feature Map

- Since a perceptron on the k^{th} convolutional layer is only connected to perceptrons in its receptive field on the $(k-1)^{th}$ layer, we can represent the weight of a perceptron on layer k as a $f_h \times f_w$ matrix, where f_h and f_w are the height and width of the receptive field.
- We call such weight of a perceptron the *Filter* (a.k.a., *Convolution Kernel*).
- We can also represent the output (i.e., activation) of all the perceptrons on a layer as a $m \times n$ matrix, where m and n are the height and width of the layer.
- When all the perceptrons on a layer use the same filter and bias, the output of the layer highlights the areas in the input of the layer that activate the filter the most.
- We call such output a *Feature Map*.

Filter and Feature Map

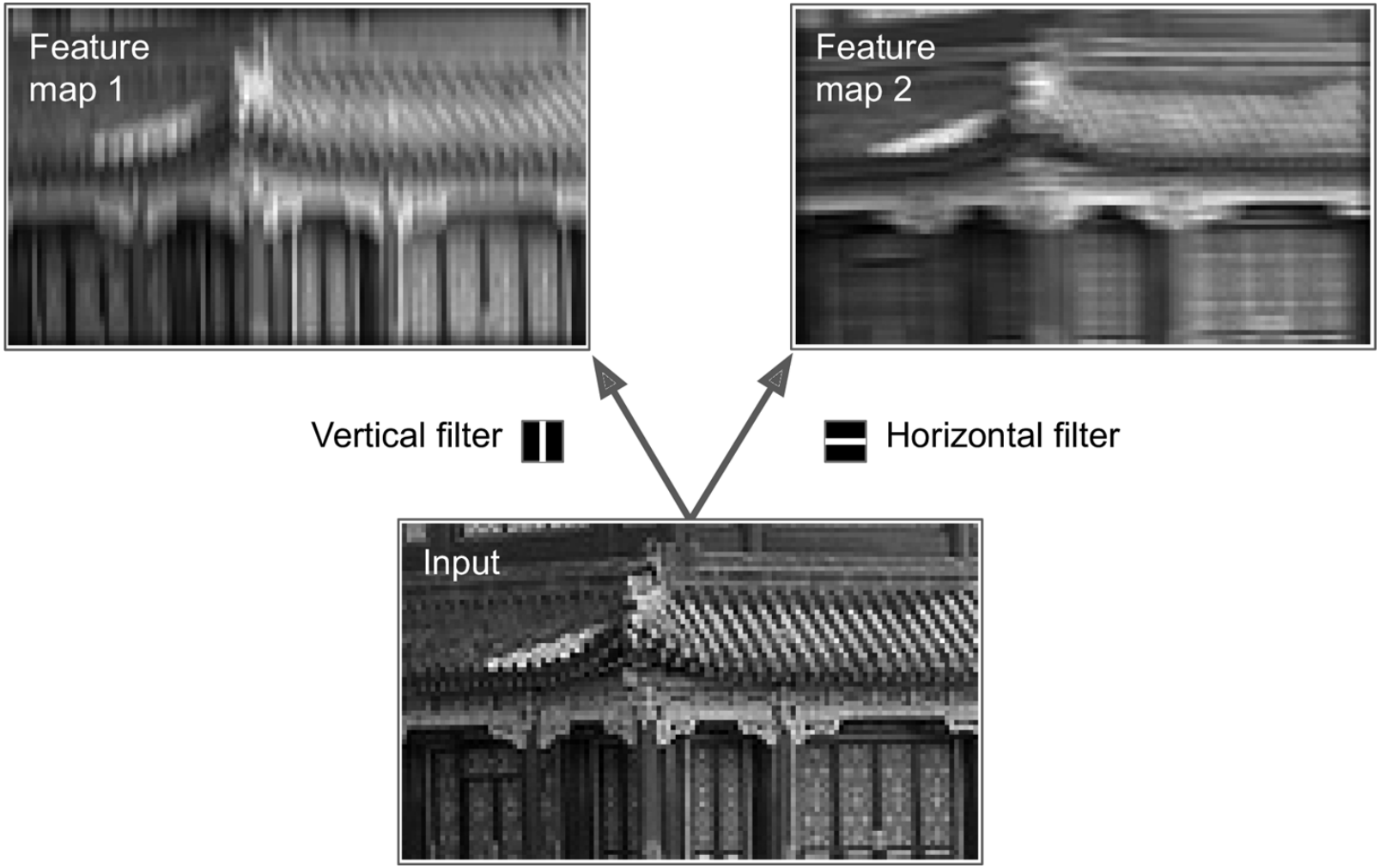

Figure 6: Filter and feature map. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

Filter and Feature Map

- The bottom panel in fig. 6 is the input image.
- The top-left panel is the feature map produced by a convolutional layer, which enhances the vertical white lines but blurs the rest:
  - each perceptron on the layer uses the same vertical filter (the black box with the middle column being white)
  - each entry in the vertical filter is zero except for the middle column (full of 1s)
- The top-right panel is the feature map produced by a convolutional layer, which enhances the horizontal white lines but blurs the rest:
  - each perceptron on the layer uses the same horizontal filter (the black box with the middle row being white)
  - each entry in the horizontal filter is zero except for the middle row (full of 1s)

Stacking Multiple Sublayers

- In fig. 6, a convolutional layer has only one layer (hence a 2d matrix), where the perceptrons have the same filter (vertical or horizontal) and bias.
- In reality, a convolutional layer usually has multiple sublayers (hence a 3d tensor), where each sublayer has the same filter and bias, but different sublayers usually have different filters and biases.
- There are two benefits for perceptrons on the same sublayer (of a convolutional layer) sharing the same filter and bias:
  - it significantly reduces the number of parameters in CNN
  - it makes CNN robust to the location of patterns (since on the one hand different perceptrons of a sublayer correspond to different receptive fields, on the other hand these different receptive fields have the same filter and bias)
- Similar to a convolutional layer, the input image may also have multiple sublayers, one per color channel (e.g., red, green and black, a.k.a., RGB).

Stacking Multiple Sublayers

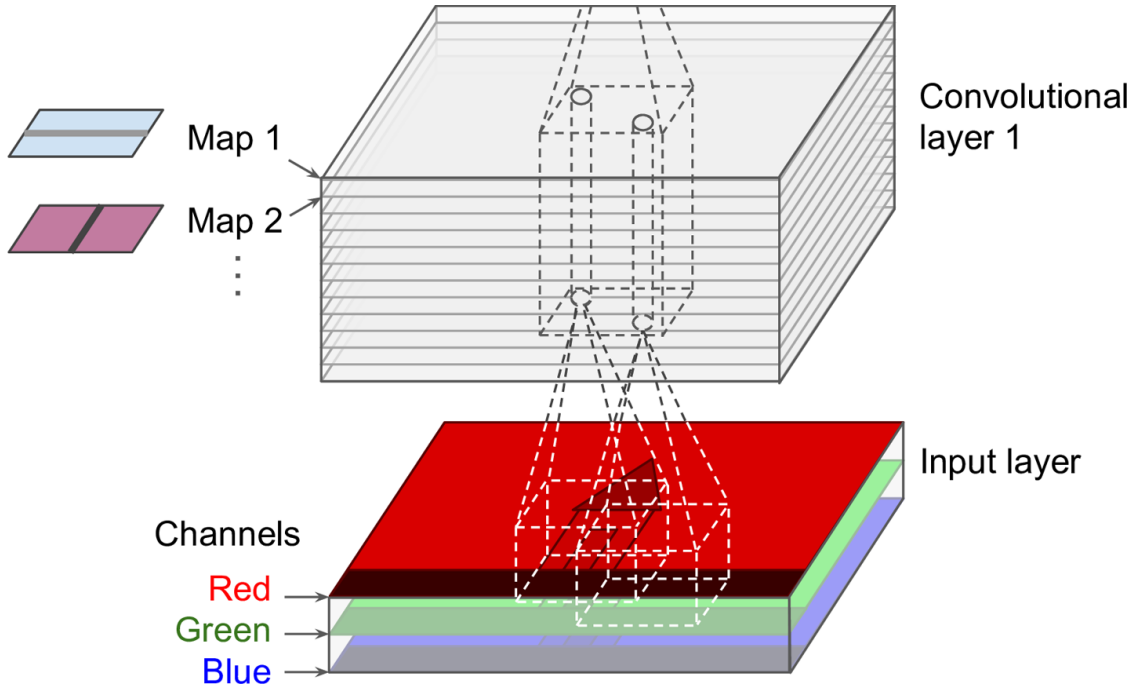

Figure 7: An input image with 3 channels and a convolutional layer with multiple sublayers. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- A perceptron in row i, column j of sublayer k of convolutional layer l is connected to perceptrons in rows $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w$ to $j \times s_w + f_w - 1$ (where f_h and f_w are the height and width of the receptive field, while s_h and s_w the vertical and horizontal strides), across all sublayers of convolutional layer $l - 1$.

Stacking Multiple Sublayers

- The output in row i, column j of sublayer k of convolutional layer l, a^l_{ijk}, is

$$a^l_{ijk} = b^l_k + \sum_{a=0}^{f_h-1} \sum_{b=0}^{f_w-1} \sum_{c=0}^{p^{l-1}-1} a^{l-1}_{i'j'k'} \times w^l_{abck} \quad \text{where} \quad \begin{cases} i' = i \times s_h + a \\ j' = j \times s_w + b \end{cases} \tag{2}$$

  - b^l_k is the bias of the perceptron that outputs a^l_{ijk}
  - f_h and f_w are the height and width of the receptive field, s_h and s_w the vertical and horizontal strides, and p^{l-1} the number of sublayers of convolutional layer $l-1$
  - $a^{l-1}_{i'j'k'}$ is the output in row i', column j' of sublayer k' of convolutional layer $l-1$ (or channel k' if layer $l-1$ is the input layer)
  - w^l_{abck} is the connection weight between perceptron in row i', column j' of sublayer c of convolutional layer $l-1$, and perceptron in row i, column j of sublayer k of convolutional layer l

Implementing Convolutional Layer

- The code below shows how to implement a convolutional layer using `keras.layers.Conv2D`:

```
1  # A convolutional layer
2  keras.layers.Conv2D(filters=64,
3                      kernel_size=[2, 2],
4                      strides=[1, 1],
5                      padding='same',
6                      activation='relu')
```

- Line 2: `filters` is the number of different filters (a.k.a., convolutional kernels)
- Line 3: `kernel_size` is the height and width of the receptive field, can be simplified as kernel_size=2 (when the height and width are the same)
- Line 4: `strides` is the stride along the height and width of the receptive field, can be simplified as strides=1 (when the two kinds of stride are the same)
- Line 5: `padding` is the type of padding, which is either 'same' or 'valid' (more on this later)
- Line 6: `activation` is the activation of the convolutional layer

Padding

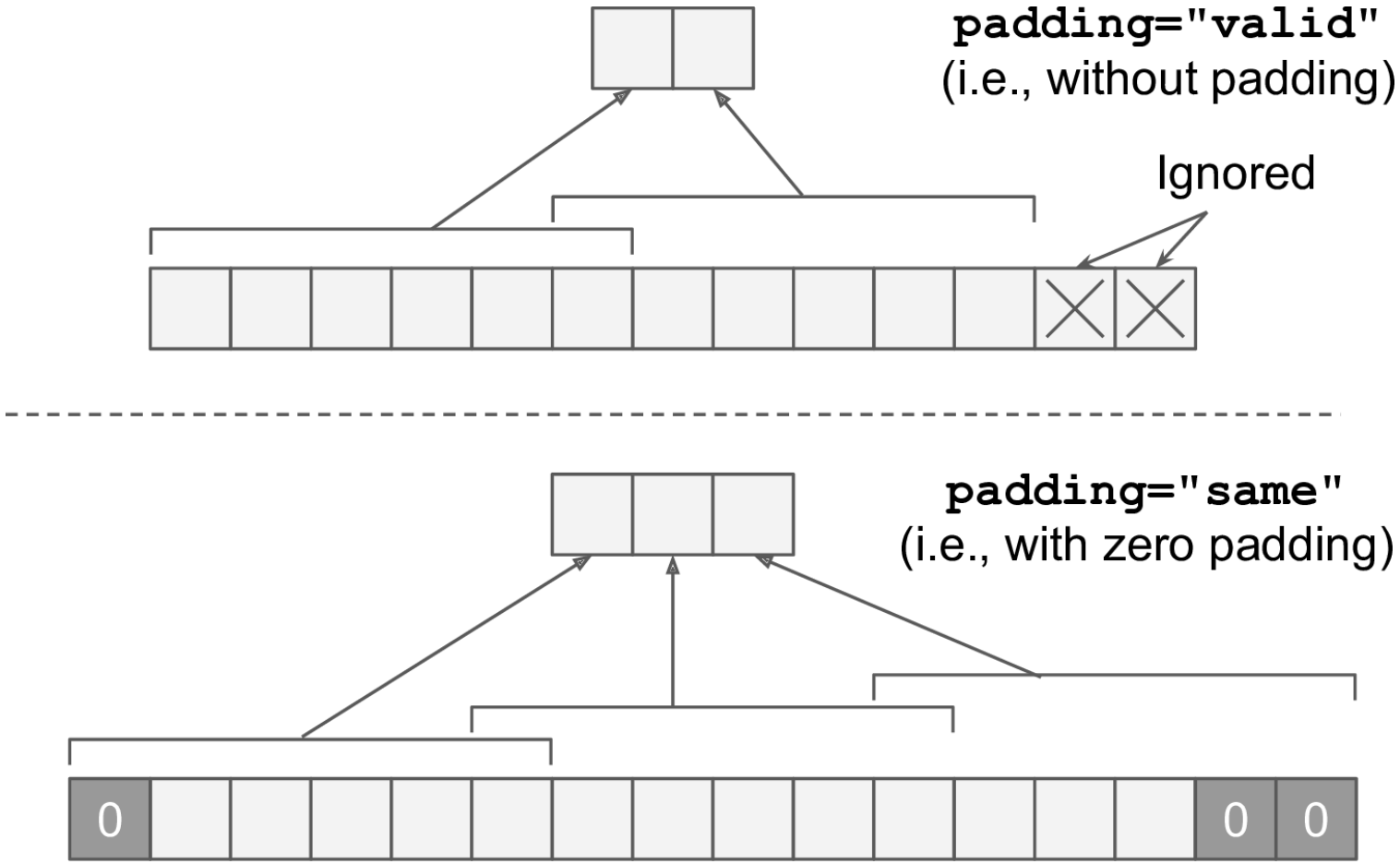

Figure 8: The 'valid' and 'same' paddings. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

Padding

- If '`valid`' on convolutional layer k:
  - we will not use zero padding for layer $k-1$
  - we may ignore some rows and columns on the bottom and right of layer $k-1$
  - the receptive filed of each perceptron on layer k lies strictly within valid entries inside layer $k-1$, hence the name *valid*
  - the top panel in fig. 8 shows an example
- If '`same`' on convolutional layer k:
  - we will use zero padding for layer $k-1$
  - we will set the number of perceptrons on layer k to the number of perceptrons on layer $k-1$, divided by the stride, rounded up
  - we will add zeros as evenly as possible around the perceptrons on layer $k-1$
  - when `strides=1`, the number of perceptrons on layer k is the same as the number of perceptrons on layer $k-1$ (hence the name *same*)
  - the bottom panel in fig. 8 shows an example

Pooling Layers

- A *Pooling Layer* follows a convolutional layer.
- The number of sublayers of a pooling layer is the same as the number of sublayers of the convolutional layer.
- A perceptron on sublayer k of the pooling layer is connected to the ones in the perceptron's receptive field on sublayer k of the convolutional layer (where the receptive field is determined by its size, stride and padding type).
- This allows a pooling layer to subsample (i.e., shrink) a convolutional layer to reduce the number of parameters in a CNN, and in turn, the time and space complexity.
- The same as perceptrons on the input layer, a perceptron on a pooling layer does not have weights. However, unlike perceptrons on the input layer that use the linear function as the activation function, a perceptron on a pooling layer uses `max` or `mean` as the activation function.
- A pooling layer uses max as the activation function is called a *Max Pooling Layer* (which is the most widely used pooling layer), whereas a pooling layer uses mean as the activation function is called an *Average Pooling Layer*.

Max Pooling Layer

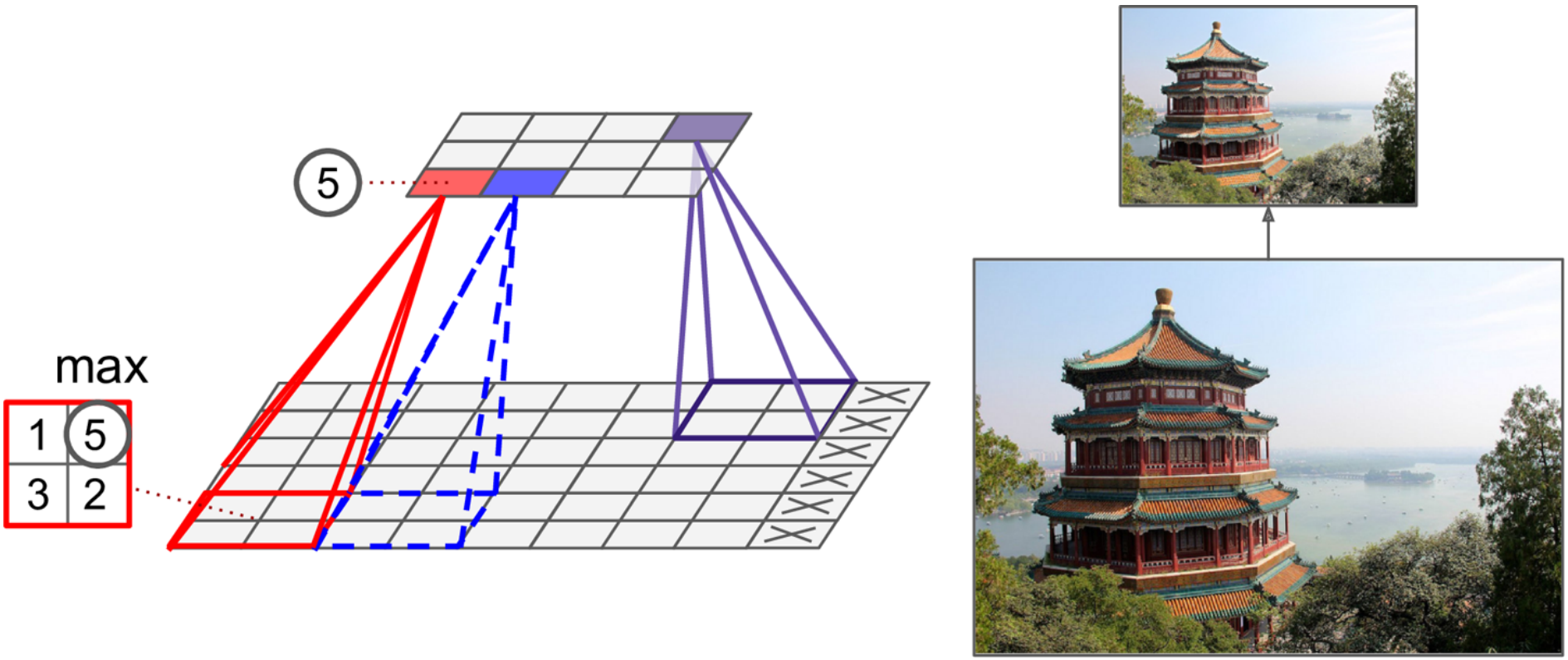

Figure 9: A convolutional layer and a max pooling layer (with a 2×2 pooling kernel, a stride of 2 and no padding). Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

Invariance to Small Translations

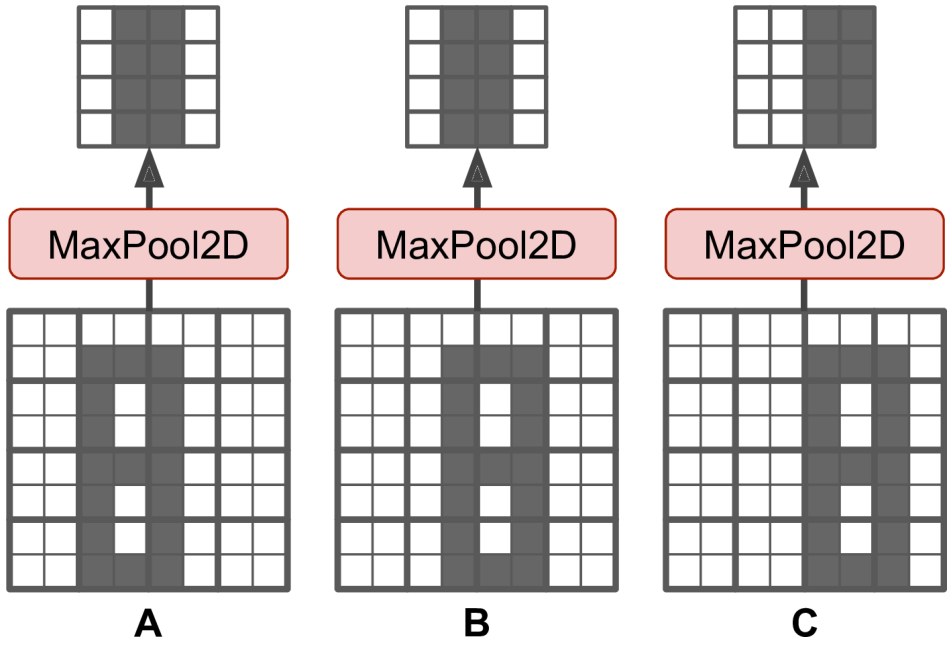

Figure 10: Invariance to small translations. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Besides reducing both time and space complexity (as discussed earlier), a max pooling layer also introduces some level of *invariance* to small translations.
- In fig. 10, images B and C are obtained by shifting A by one and two pixels.
- When passing the three images to a max pooling layer (with a 2×2 pooling kernel and stride 2), the output of images A and B are exactly the same, and they are 50% the same as that of image C.

Pros and Cons of Max Pooling Layers

- Pros:
  - reduces both time and space complexity
  - the invariance to small translations could be helpful for cases (e.g., classification) where prediction does not depend too much on small translations
- Cons:
  - it may drop too many outputs of the convolutional layer (e.g., the max pooling layer in fig. 10 will drops 75% output of the convolutional layer)
  - the invariance to small translations could be harmful for cases (e.g., *Semantic Segmentation* which classifies each pixel in an image based on the object the pixel belongs to) where prediction actually depends on small translations

Implementing Max Pooling Layer

- The code below shows how to implement a max pooling layer using `keras.layers.MaxPooling2D`:

```
1  # A max pooling layer
2  keras.layers.MaxPooling2D(pool_size=(2, 2), padding='valid')
```

- Line 2: `pool_size` is the height and width of the receptive field, can be simplified as pool_size=2 (when the height and width are the same).
- Line 2: `padding` (the same as parameter `padding` in `keras.layers.Conv2D`) is the type of padding, which is either 'valid' (the default hence no need to be specified) or 'same'.

Typical CNN Architecture

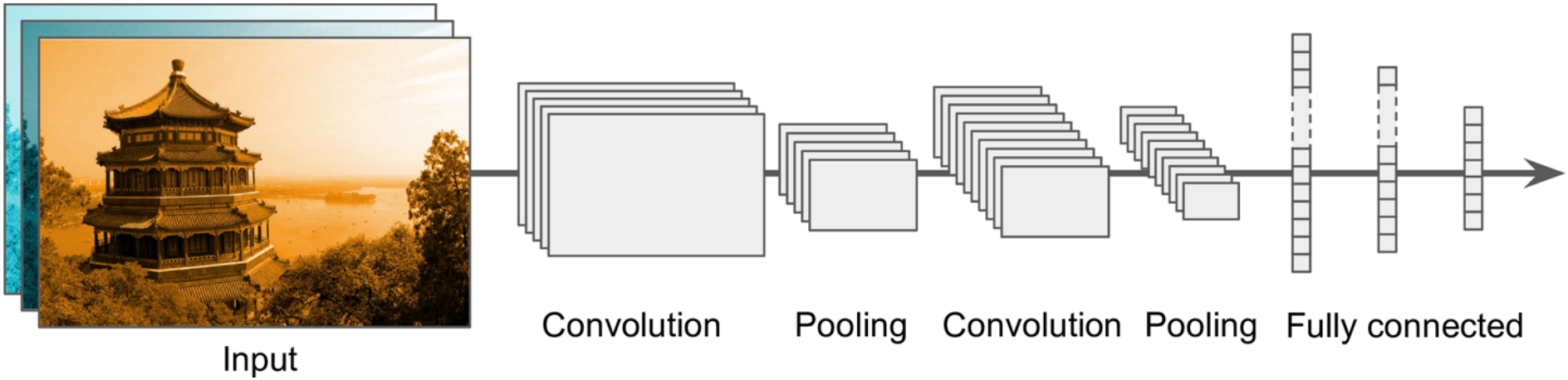

Figure 11: Typical CNN architecture. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

Typical CNN Architecture

1. Input layer
2. Repeat:
   1. one or multiple convolutional layers which usually get smaller but deeper when iteration progresses (since the number of features often gets smaller but the number of ways to combine the features often gets larger)
   2. a pooling layer where the number of sublayers is the same as the number of sublayers of its input convolutional layer
3. Repeat:
   1. fully connected feedforward layers (which usually get smaller when iteration progresses)
4. Output layer

Typical CNN Architecture

- **Good Practice**:
  - for the first convolutional layer:
    - it is usually recommended to use larger filters (e.g., 5×5), with a stride of 2 or more
    - this will usually keep a good number of samples in the input image, without adding too many parameters (since the input image usually only has three channels)
  - for the other convolutional layers:
    - if we also used larger filters, we could add too many parameters (since the previous layer usually has many sublayers)
    - instead, it is usually recommended to use smaller filters (e.g., 3×3), which will significantly reduce the number of parameters in a CNN
    - it is also recommended to double the number of filters after each pooling layer (this often will not increase the number of perceptrons on the convolutional layer since the pooling layer often has much fewer number of perceptrons)
  - for all the convolutional layers, it is usually recommended to use `ReLU` as the activation function

Implementing CNN

- The code below shows how to implement CNN using the *Sequential* API for the Fashion MNIST dataset (with 60,000 28 $\times$ 28 images):

```
model = keras.models.Sequential([
    keras.layers.Conv2D(32, 5, activation='relu', padding='same',
                        input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(64, 2, activation='relu', padding='same'),
    keras.layers.Conv2D(64, 2, activation='relu', padding='same'),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 2, activation='relu', padding='same'),
    keras.layers.Conv2D(128, 2, activation='relu', padding='same'),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation='softmax')
])
```

Implementing CNN

- Lines 2 and 3: A convolutional layer with:
  - 32 as the number of different filters (convolutional kernels)
  - 5 as the height and width of the receptive field
  - 1 as the stride along the height and width of the receptive field
  - `ReLU` as the activation
  - `'same'` as the padding
  - [28, 28, 1] as the input shape, since the input is a 28×28 grayscale (with 1 channel) image
- Line 4: A max pooling layer with:
  - 2 as the height and width of the receptive field
  - `None` as the stride
  - `'valid'` as the padding
- Lines 5 to 7: Two deeper but smaller convolutional layers, followed by a max pooling layer.
- Lines 8 to 10: Two even deeper convolutional layers, followed by a max pooling layer.
- Lines 11 to 16: A fully connected feedforward DNN where each hidden layers is followed by a dropout layer.

Pretrained Models

- In the past decade, many state-of-the-art CNNs have been developed, leading to amazing advances in computer vision.
- We can see this progress from the improvement of the proposed models in competitions such as the ILSVRC ImageNet challenge (from 2010 to 2017), where the *Top-Five Error Rate* for image classification dropped from 26% to less than 2.3%:
  - ImageNet has 1,000 classes, some of which (e.g., 120 dog breeds) are very difficult to separate
  - the top-five error rate is the proportion of testing images where the top-five predictions do not include the correct class

Further Reading

- See HML Chapter 14 for a very nice introduction of the following state-of-the-art pretrained CNNs:
  - LeNet-5 (1998)
  - AlexNet (2012 ImageNet ILSVRC winner)
  - GoogLeNet (2014 ImageNet ILSVRC winner)
  - VGGNet (2014 ImageNet ILSVRC runner-up)
  - ResNet (2015 ImageNet ILSVRC winner)
  - Xception (2016)
  - SENet (2017 ImageNet ILSVRC winner)
- See keras.applications for other state-of-the-art pretrained models.

Transfer Learning

- As we discussed previously, while in theory we can implement our own DNN from scratch, in reality we are not recommended to do so, when there are state-of-the-art DNNs pretrained on similar data.
- Instead, we are suggested to tweak the pretrained DNN to make it suitable for our data. This approach is called *Transfer Learning*.
- Transfer learning will not only speed up designing, training and fine-tuning the DNN considerably, but also require significantly less training data.
- It turns out that transfer learning works particularly well in computer vision, since the lower layers of a pretrained CNN will usually capture simple features that are common in many data (hence can be reused).

Designing CNN with Pretrained CNN

- Here we can simply follow the good practice (for designing DNN with pretrained DNN) discussed previously (also shown below).
- **Good Practice**:
  - to design a DNN with pretrained model, we should
    1. reuse the lower layers of the pretrained model as the base
    2. add extra layers (that work for our data) on top of the base
  - the more similar the data are, the more lower layers of a pretrained DNN we should reuse
  - it is even possible to reuse all the hidden layers of a pretrained DNN, when the data are similar enough
  - we should resize our data so that the number of features in the resized data is the same as the number of perceptrons on the input layer of the pretrained DNN

Training CNN with Pretrained CNN

- Here we can simply follow the good practice (for training DNN with pretrained DNN) discussed previously (also shown below).
- **Good Practice**:
  - to train a DNN with pretrained model, we should
    1. freeze all the reused layers of the pretrained DNN (i.e., make their weights non-trainable so that backpropagation will not change them) then train the DNN
    2. unfreeze one or two top hidden layers of the pretrained DNN (the more training data we have the more top hidden layers we can unfreeze) and reduce the learning rate when doing so (thus the fine-tuned weights on the lower layers will not change significantly)
  - If the above steps do not produce an accurate DNN
    - if we do not have sufficient data, we can drop the top hidden layers and repeat the above steps
    - otherwise, we can replace (rather than drop) the top hidden layers or even add more hidden layers

Designing and Training DNN with Pretrained DNN

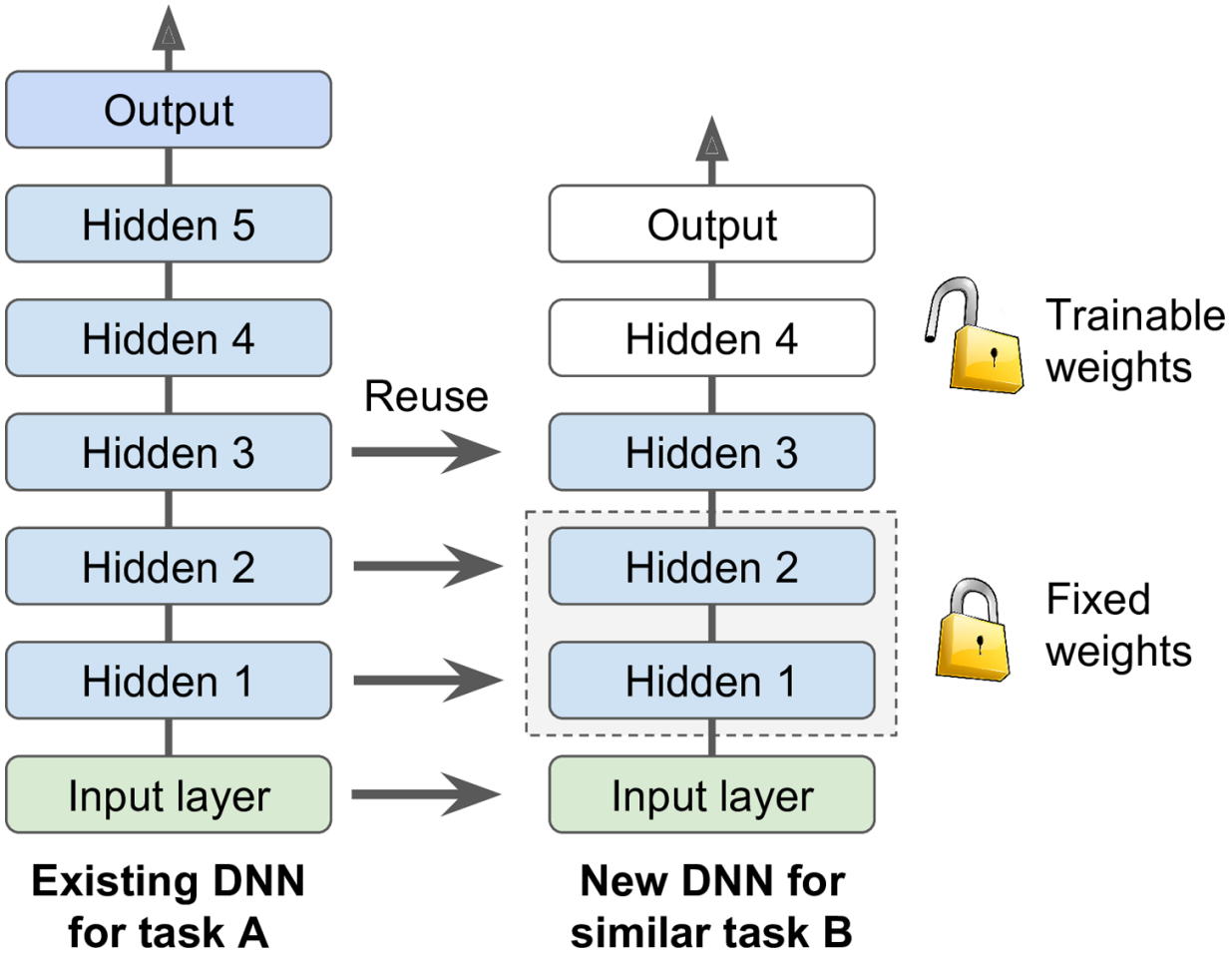

Figure 12: Designing and training DNN with pretrained DNN.

Implementing CNN with Pretrained CNN

- The code below shows how to build a CNN on top of the pretrained Xception (after removing its output layer) using `keras.applications`:

```
1  # Add the pretrained layers
2  pretrained_model = keras.applications.xception.Xception(include_top
       =False, weights='imagenet')
3
4  # Add GlobalAveragePooling2D layer
5  average_pooling = keras.layers.GlobalAveragePooling2D()(
       pretrained_model.output)
6
7  # Add the output layer
8  output = keras.layers.Dense(n_classes, activation="softmax")(
       average_pooling)
9
10 # Get the model
11 model = keras.Model(inputs=pretrained_model.input, outputs=output)
```

- Line 2: `include_top` determines whether to include the output layer of Xception.
- Line 3: `weights` determines where Xception was pretrained.

Freezing the Pretrained Layers

- The code below shows how to freeze the pretrained layers:

```
1 # For each layer in the pretrained model
2 for layer in pretrained_model.layers:
3     # Freeze the layer
4     layer.trainable = False
```

- Line 4: `layer.trainable` means whether the layer's parameters should be trainable (changable).
- The code below shows how to compile the CNN (we need to compile a model after freezing its layers):

```
1 # Compile the model
2 model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.01),
3               loss='sparse_categorical_crossentropy',
4               metrics=['accuracy'])
```

- Line 2: We use `Adam` optimizer with `learning rate` 0.01.

Unfreezing the Pretrained Layers

- The code below shows how to unfreeze the pretrained layers:

```
1 # For each layer in the pretrained model
2 for layer in pretrained_model.layers:
3     # Unfreeze the layer
4     layer.trainable = True
```

- Line 4: `layer.trainable` means whether the layer's parameters should be trainable (changable).
- The code below shows how to compile the CNN (we need to compile a model after unfreezing its layers):

```
1 # Compile the model
2 model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
3               loss='sparse_categorical_crossentropy',
4               metrics=['accuracy'])
```

- Line 2: We use `Adam` optimizer with `learning_rate` 0.001 (one tenth of the previous learning rate).