Popular Machine Learning Methods: Idea, Practice and Math

Convolutional Neural Networks

Yuxiao Huang

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

Spring 2022

Reference

- This set of slices was largely built on the following 7 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
NND Neural Network Design (2nd Edition)
LFD Learning From Data
RL Reinforcement Learning: An Introduction (2nd Edition)

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

Code Example & Case Study

- See related code example in github repository: /p3_c2_s3_convolutional_neural_networks/code_example
- See related case study in github repository: /p3_c2_s3_convolutional_neural_networks/case_study

Table of Contents

Learning Objectives: Expectation

- It is **expected** to understand
  - the architecture and idea of Convolutional Neural Networks (CNNs)
  - the good practices for building CNNs
  - the idea of and good practices for transfer learning using state-of-the-art pretrained CNNs

Learning Objectives: Recommendation

- It is **recommended** to understand
  - the architecture and idea of some state-of-the-art pretrained CNNs:
    - AlexNet
    - GoogLeNet
    - ResNet
    - SENet

Fashion MNIST Dataset

Figure 1: Kaggle competition: Fashion MNIST dataset. Picture courtesy of Kaggle.

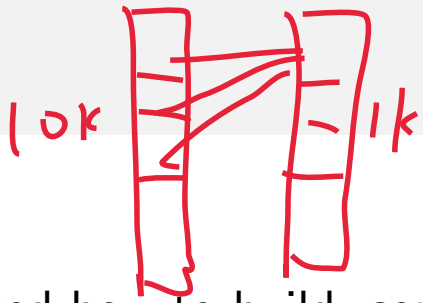
- Fashion MNIST dataset: a dataset of Zalando's article images:
  - features: 28×28 (i.e., 784) pixels (taking value in $[0, 255]$) in a grayscale image
  - target: the article of clothing in each image:
    - 0: T-shirt/top
    - 1: Trouser
    - 2: Pullover
    - 3: Dress
    - 4: Coat
    - 5: Sandal
    - 6: Shirt
    - 7: Sneaker
    - 8: Bag
    - 9: Ankle boot

CIFAR-10

Figure 2: Kaggle competition: CIFAR-10 dataset. Picture courtesy of Kaggle.

- CIFAR-10 dataset: a dataset for image classification:
  - features: 32×32 (i.e., 1024) pixels (taking value in $[0, 255]$) in a color image
  - target: the object in each image:
    - 0: airplane
    - 1: automobile
    - 2: bird
    - 3: cat
    - 4: deer
    - 5: dog
    - 6: frog
    - 7: horse
    - 8: ship
    - 9: truck

Why CNNs?



- In /p3_c2_s1_deep_neural_networks, we discussed how to build, compile and train Fully Connected Feedforward Neural Networks (FNNs).
- Let us apply a FNN to a hypothetical image, where:
  - the input image has 100×100 pixels
  - the first hidden layer of the FNN has 1000 perceptrons
- Then the number of parameters (weights and biases) on the first hidden layer can be calculated as

$$p^0 p^1 + p^1 = 10^4 \times 10^3 + 10^3 = 10^7 + 10^3. \tag{1}$$

  Here:
  - $p^0 = 10^4$ is the number of perceptrons on the input layer
  - $p^1 = 10^3$ is the number of perceptrons on the first hidden layer
- That is, there are over 10 million parameters on the first hidden layer alone!
- As a result, FNN is too computationally expensive to be suitable for computer vision.

Biological Neurons, Visual Cortex and Receptive Fields

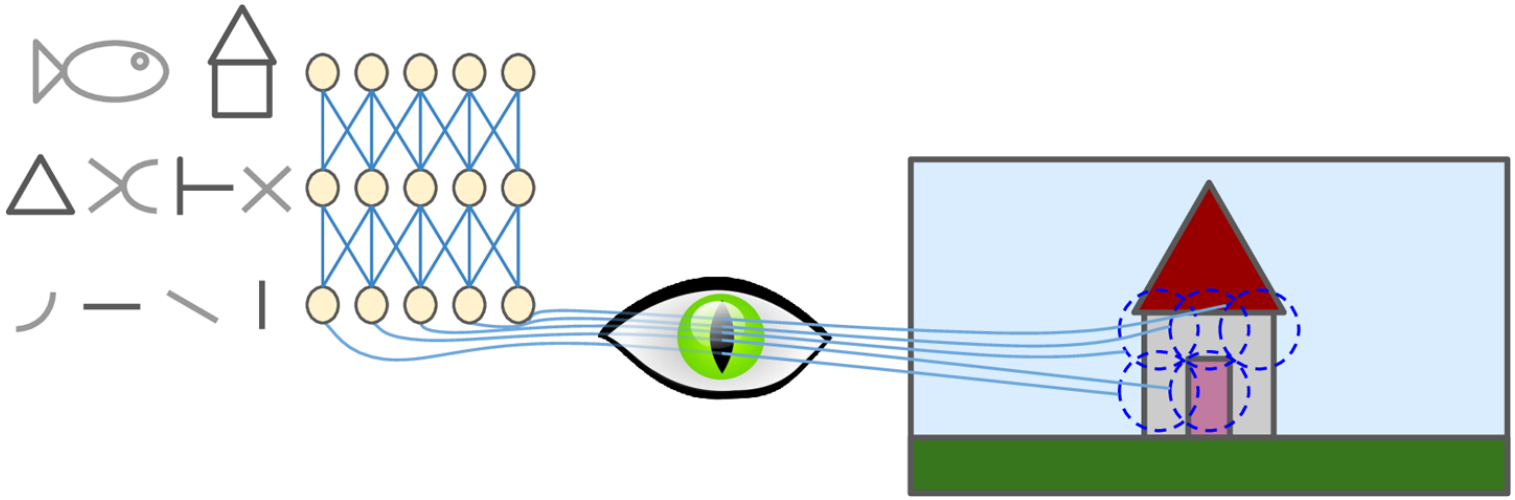


Figure 3: Biological neurons, visual cortex and receptive fields. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- 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 Architecture of CNNs

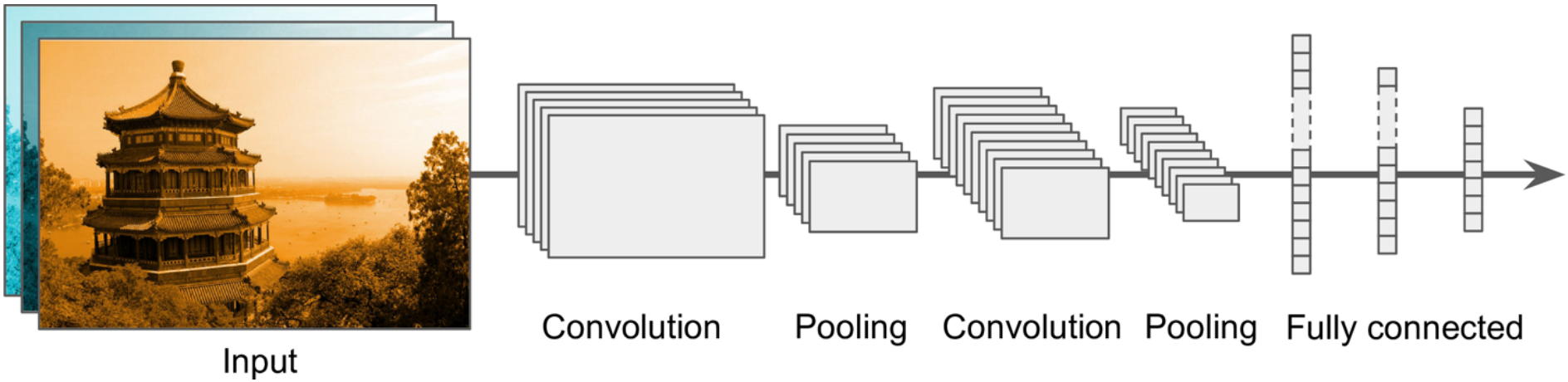


Figure 4: Typical architecture of CNNs. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Below are three key components in the typical architecture of CNNs:
  - Convolutional Layers (more on this later)
  - Pooling Layers (more on this later)
  - Fully Connected Layers (the same as those in FNNs)

Convolutional Layer

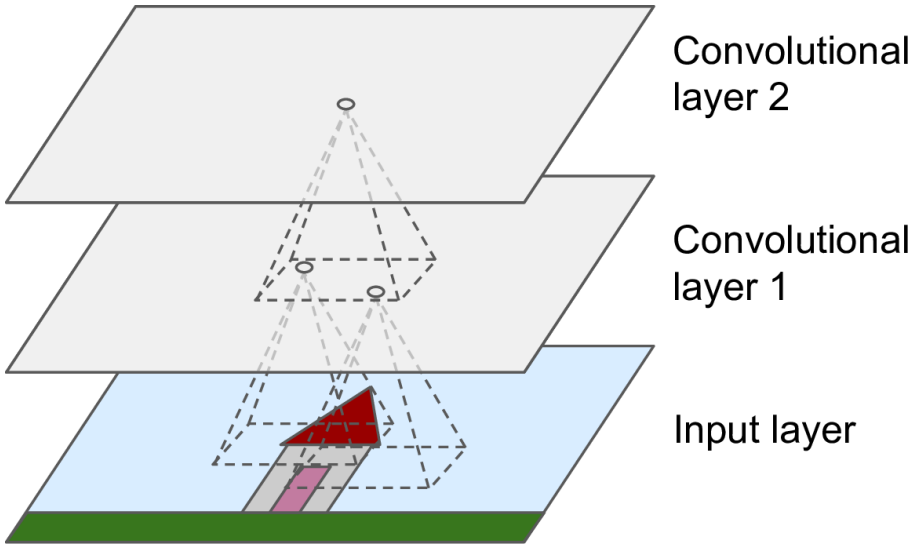


Figure 5: Convolutional layer. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Similar to biological neurons (see fig. 3), 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 CNNs, as they not only result in significantly fewer parameters (more on this later) but also allow capturing the hierarchical structure in real-world images.

Convolutional Layer

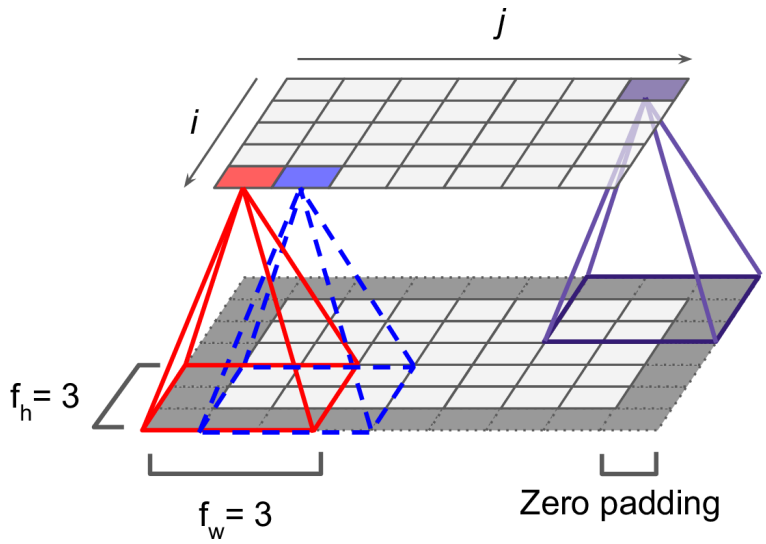


Figure 6: Adjacent layers in a CNN. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- 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 allow adjacent layers to have the same number of rows and columns, we usually add zeros around a layer, a step often called *Zero Padding* (more on this later).

Convolutional Layer

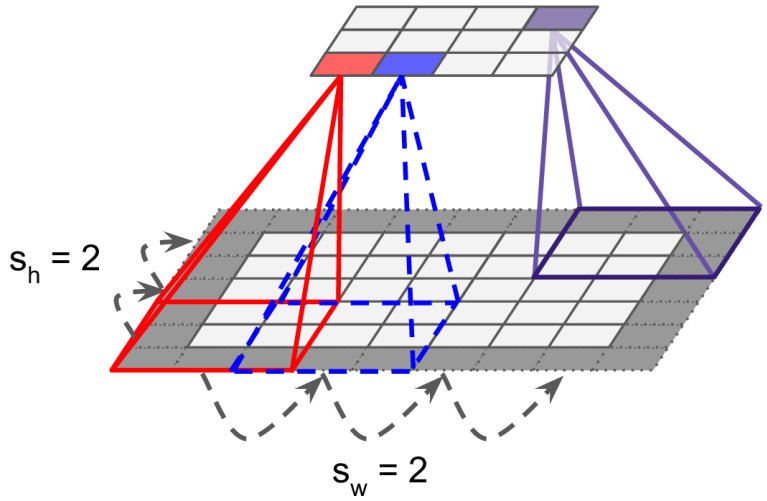


Figure 7: Adjacent layers in a CNN, with a stride of 2. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- The shift from one receptive field to the next is called the *Stride*.
- In fig. 6 the stride is 1, whereas in fig. 7 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 is the height of the receptive field
  - f_w is the width of the receptive field
- We call such weight matrix 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 is the height of the layer
  - n is the 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

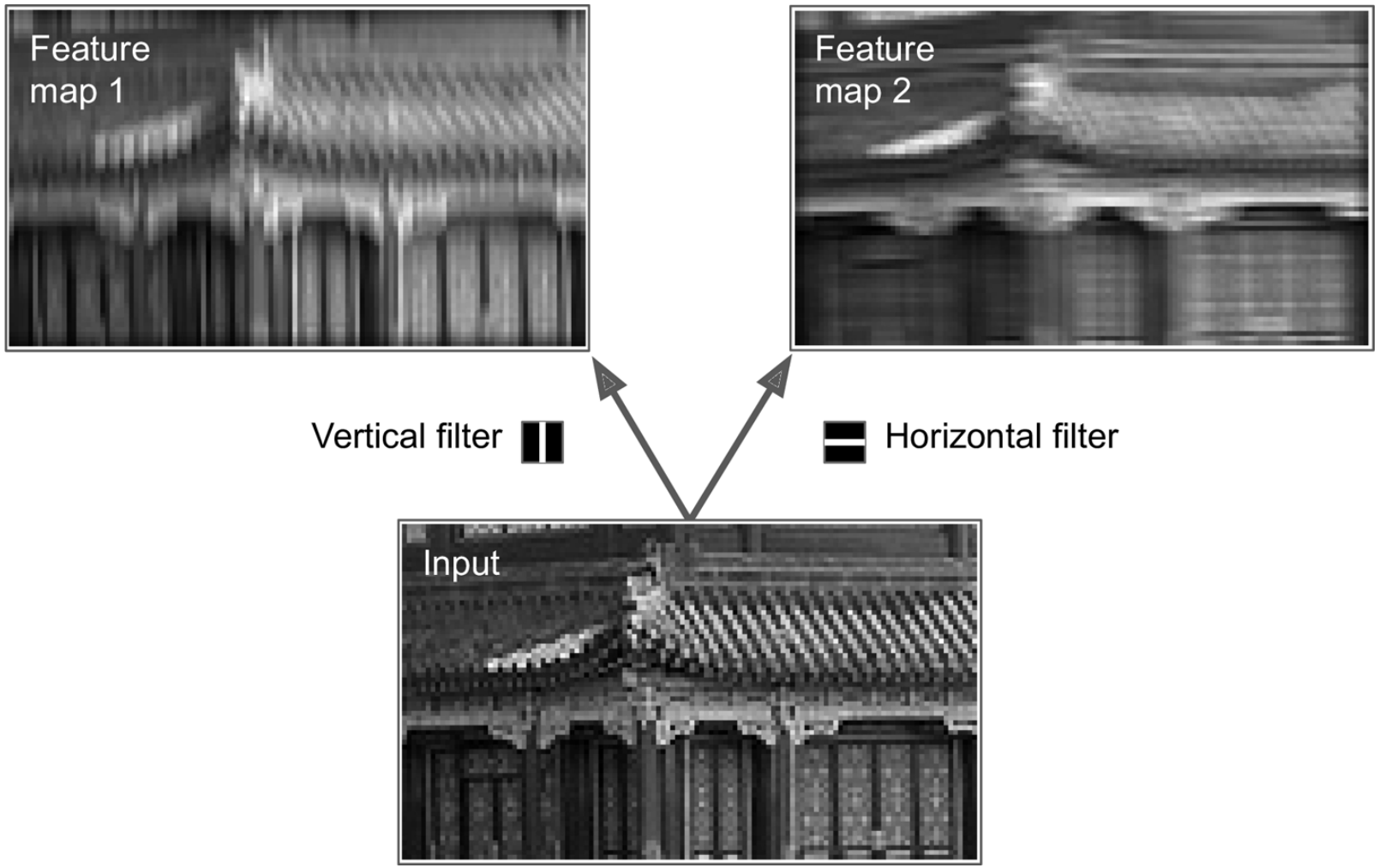


Figure 8: Filter and feature map. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

Filter and Feature Map

- The bottom panel in fig. 8 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. 8, 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):
  - each sublayer has the same filter and bias
  - 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 CNNs
  - it makes CNNs 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

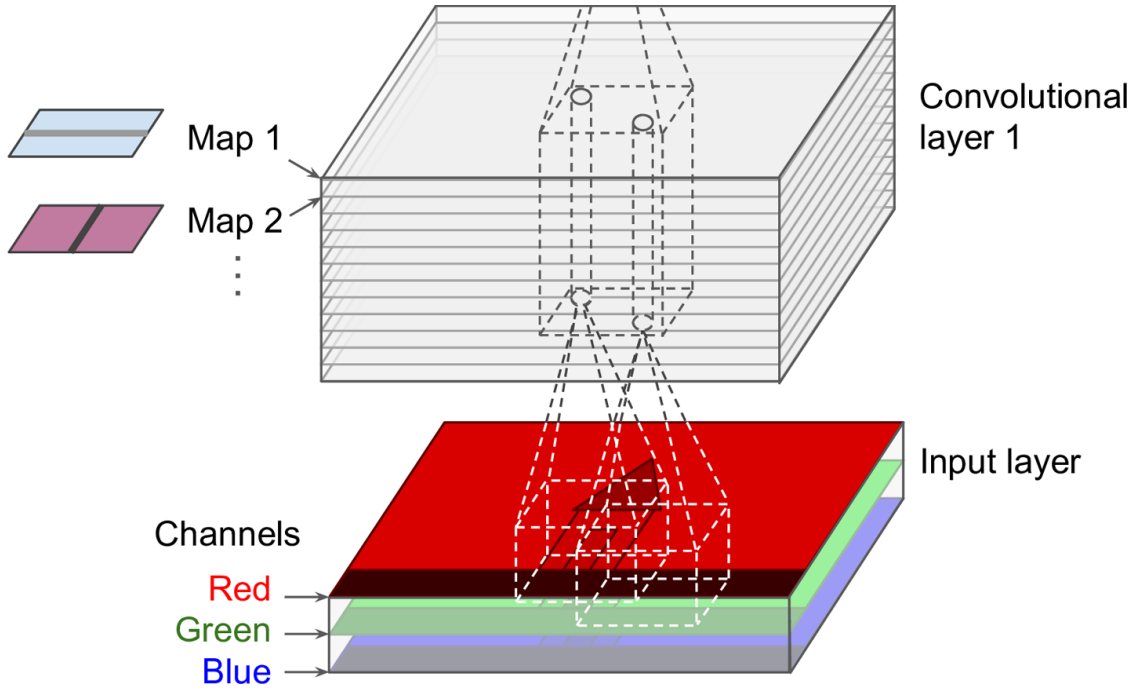


Figure 9: An input image with 3 channels and a convolutional layer with multiple sublayers. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- 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

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

  Here:
  - 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 are the vertical and horizontal strides
  - p^{l-1} is 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

Convolutional Layer: Code Example

- See /p3_c2_s3_convolutional_neural_networks/code_example:
  1. cell 17

Padding

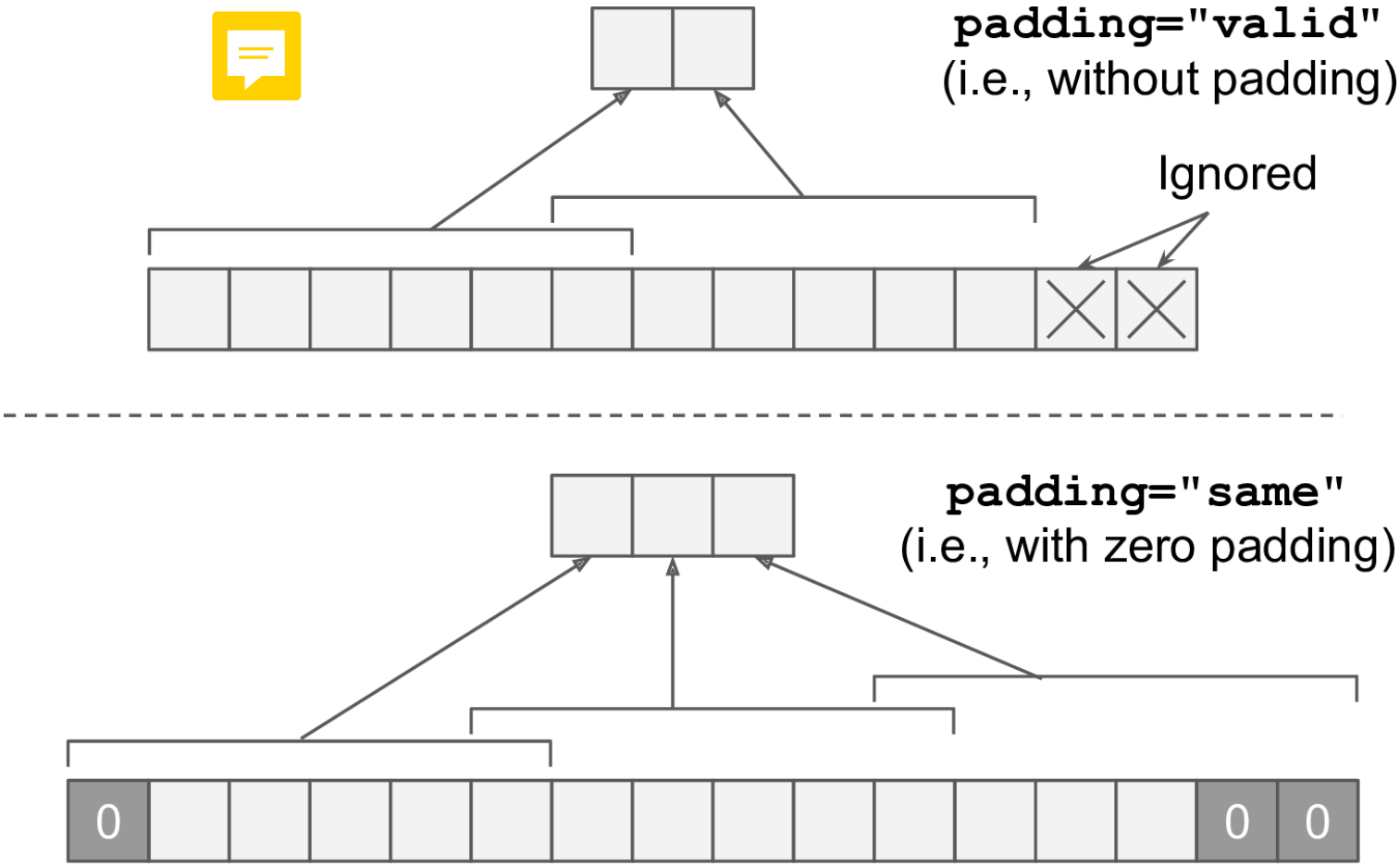


Figure 10: The ‘`valid`’ and ‘`same`’ paddings. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

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. 10 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. 10 shows an example

Pooling Layer

- A Pooling Layer follows a convolutional layer.
- The number of sublayers of a pooling layer is the same as the number of sublayers of its input 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 its input 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 identity 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)
  - a pooling layer uses mean as the activation function is called an *Average Pooling Layer*.

Max Pooling Layer

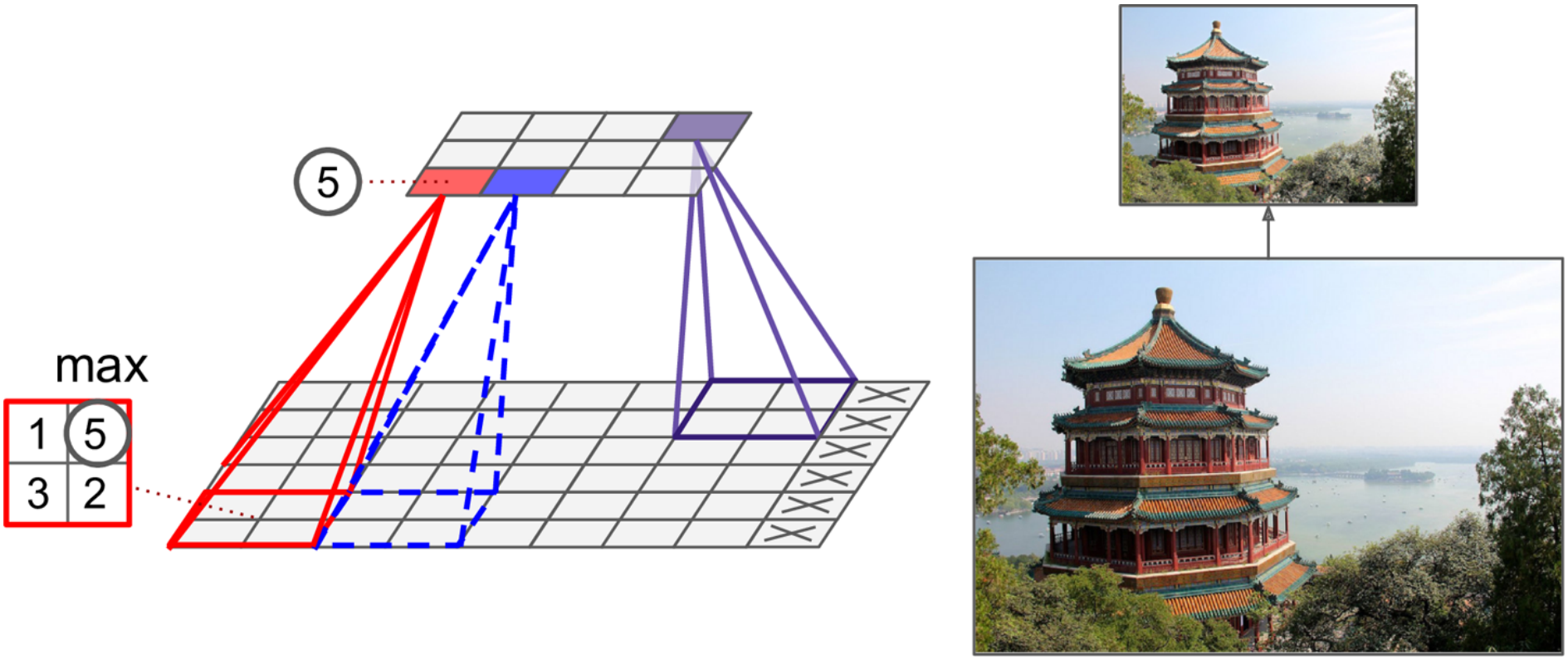


Figure 11: A convolutional layer and a max pooling layer (with a 2×2 pooling kernel, a stride of 2 and no padding). Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

Invariance to Small Translations

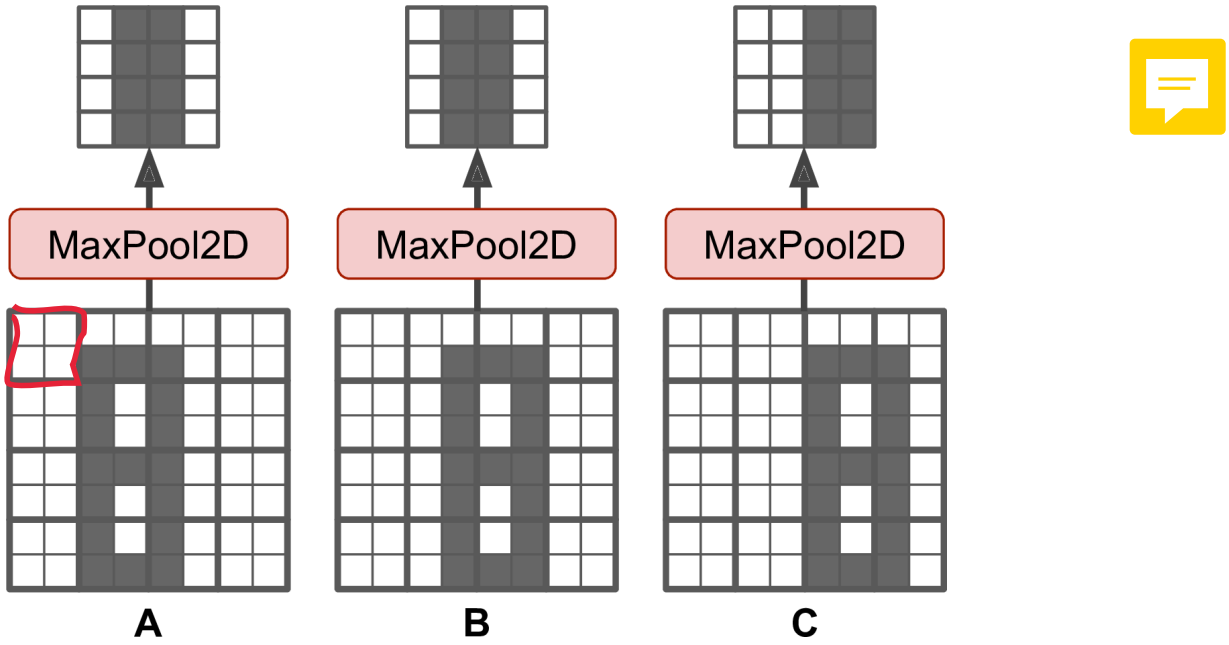


Figure 12: Invariance to small translations. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- 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. 12, 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 Layer

- 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. 12 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

Takeaway

- Pros:
  - reduces both time and space complexity
  - the invariance to small translations could be helpful for cases where prediction does not depend too much on small translations
- Cons:
  - it may drop too many outputs of the convolutional layer
  - the invariance to small translations could be harmful for cases where prediction actually depends on small translations

Max Pooling Layer: Code Example

- See /p3_c2_s3_convolutional_neural_networks/code_example:
  1. cell 17

Typical Architecture of CNNs

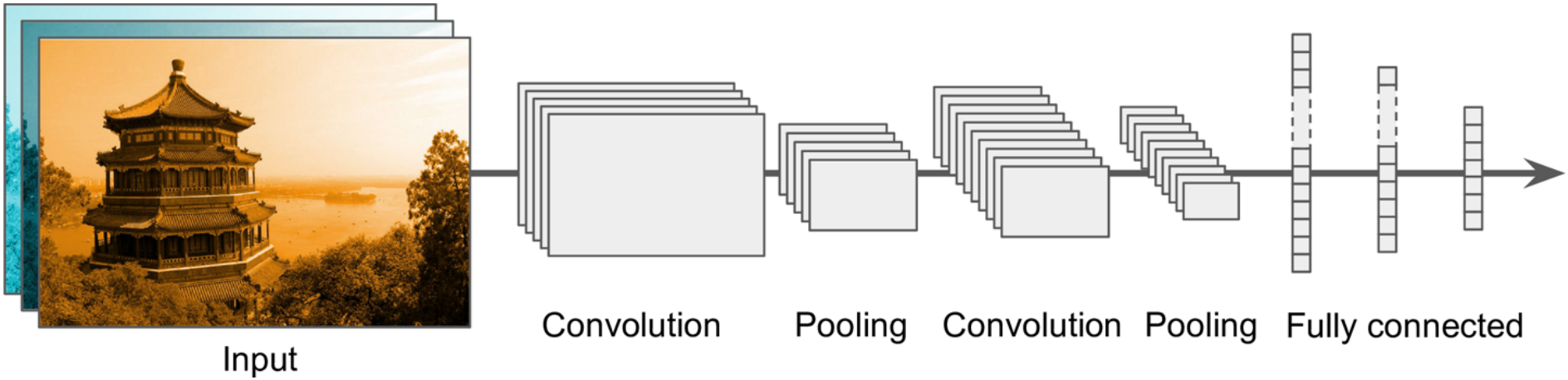


Figure 4: Typical architecture of CNNs. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Fig. 4 shows the following typical architecture of CNNs:
  1. Input layer
  2. Repeat:
     1. one or multiple convolutional layers which usually get smaller but deeper when iteration progresses (as 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 (as the input image usually only has three channels)
- For the other convolutional layers:
  - if we were to use larger filters, we could add too many parameters (as 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.

Building CNNs: Code Example

- See /p3_c2_s3_convolutional_neural_networks/code_example:
  1. cell 17

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

State-of-the-Art Pretrained Models

- Below are some of the 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)
- Here we will focus on the four winners in the list above:
  - AlexNet
  - GoogLeNet
  - ResNet
  - SENet
- See HML: Chapter 14 for a very nice introduction of the remaining pretrained CNNs in the list above.
- See keras.applications for other state-of-the-art pretrained models.

AlexNet

- AlexNet was the winner of the 2012 ImageNet ILSVRC challenge, with a top-five error rate of 17%.
- AlexNet was also the first to have convolutional layers directly followed by convolutional layers (rather than pooling layers).
- Figs. 13 and 14 show the architecture of AlexNet, where:
  - fig. 13 shows the first 5 layers
  - fig. 14 shows the last 6 layers

AlexNet: Architecture (First Five Layers)

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	–	–	–	–

Figure 13: The first 5 layers in AlexNet. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

AlexNet: Architecture (Last Six Layers)

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

Figure 14: The last 6 layers in AlexNet. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

AlexNet: Regularization

- AlexNet uses three kinds of regularization methods:
  - Dropout:
    - applied after layers F9 and F10 (see fig. 14)
    - with dropout rate 0.5
  - Data Augmentation
  - Local Response Normalization

Data Augmentation

- In /p2_c1_data_preprocessing we discussed one kind of data augmentation method named *Oversampling*.
- Oversampling generates data for the minority classes in the training data (ones with fewer samples) to address the *Class Imbalance* problem (where it is much less likely to predict a new sample as the minority classes).
- In AlexNet, data augmentation generates new images for all the images in the training data to address the overfitting problem.
- This is done by:
  - randomly shifting the training images by various offsets
  - randomly flipping the training images horizontally
  - randomly changing the lighting conditions
- This allows the model to be more robust to variations in the position, orientation, and lighting in the pictures.

Local Response Normalization: Idea

- AlexNet also uses a normalization method named *Local Response Normalization* (LRN) directly after layers C1 and C3 (see fig. 14).
- The idea of LRN is that, if a perceptron on a sublayer of a convolutional layer is strongly activated, we want to inhibit perceptrons at the same entry on adjacent sublayers.
- This forces different sublayers to explore a wider range of features, which in turn addresses overfitting.

Local Response Normalization: Math

- Concretely, in LRN we normalize a perceptron's output as

$$b_i = a_i \left(k + \alpha \sum_{j=j_{\text{low}}}^{j_{\text{high}}} a_j^2 \right)^{-\beta} \quad \text{where} \quad \begin{cases} j_{\text{high}} = \min\left(i + \frac{r}{2}, f_n - 1\right), \\ j_{\text{low}} = \max\left(0, i - \frac{r}{2}\right). \end{cases} \tag{3}$$

  Here:

  - b_i is the normalized output of a perceptron on sublayer i of a convolutional layer
  - a_i is the output (i.e., activation) of the perceptron
  - k (the *Bias*), α, β and γ (*Depth radius*) are hyperparameters, where (by default):
    - $k = 0.75$
    - $\alpha = 10^{-3}$
    - $\beta = 0.75$
    - $\gamma = 5$
  - f_n is the number of sublayers of the convolutional layer

- Eq. (3) says that if a_i is large (perceptron on sublayer i is strongly activated), b_j will be small (perceptron at the same entry on adjacent sublayers will be inhibited), as a_i^2 will enter the denominator in eq. (3) when calculating b_j.

GoogLeNet

- GoogLeNet was the winner of the 2014 ImageNet ILSVRC challenge, with a top-five error rate of 7%.
- A key reason for the improved performance is that:
  - on the one hand, GoogLeNet is much deeper than AlexNet
  - on the other hand, GoogLeNet uses subnetworks named *Inception Modules*, which allow it to have fewer parameters than AlexNet (by a factor of 10)

Inception Module: Architecture

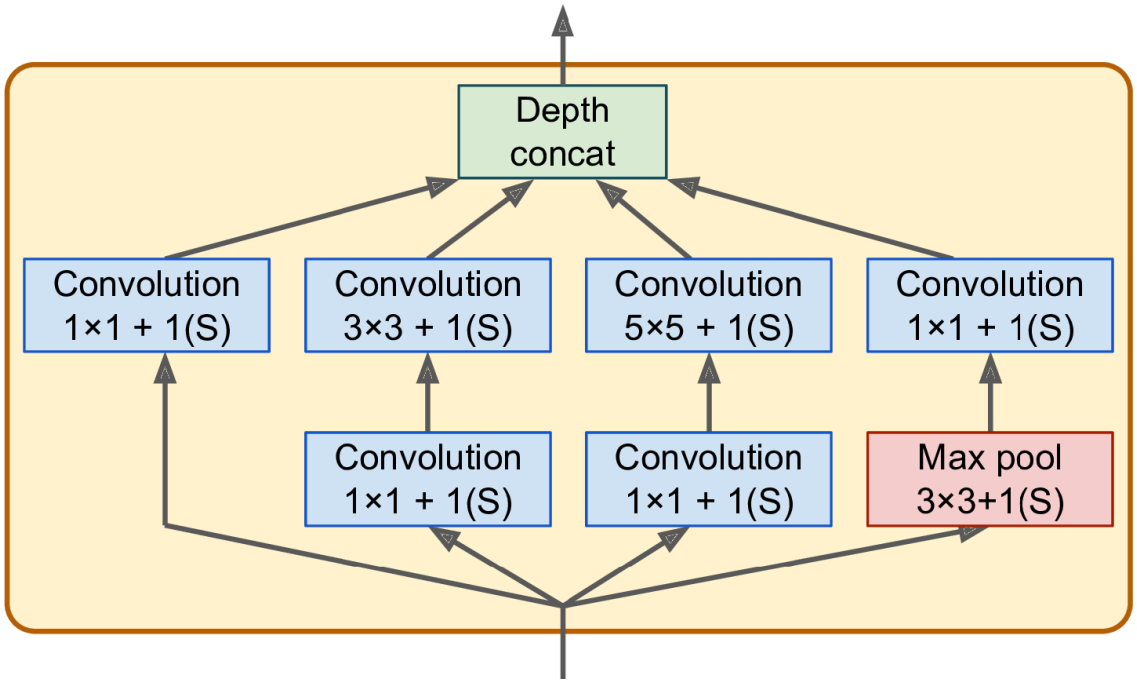


Figure 15: The inception module in GoogLeNet. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- In fig. 15, $a \times a + 1(S)$ means that:
  - the filter (a.k.a., convolution kernel) is $a \times a$
  - the stride is 1
  - the padding is `same`
- the *Depth Concat* layer concatenates all the outputs along the depth dimension.

Inception Module: Idea

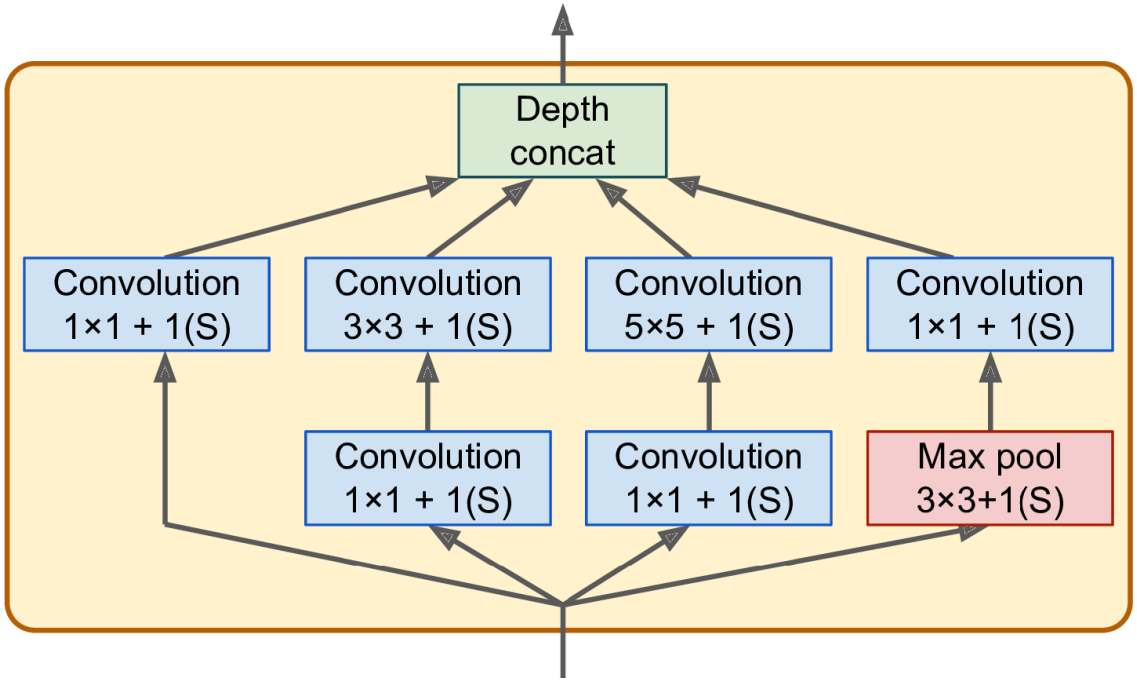


Figure 15: The inception module in GoogLeNet. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- In fig. 15, the convolutional layer with 1×1 filters:
  - can capture pattern along the depth
  - can act as *Bottleneck Layers* by outputting fewer feature maps than their inputs (so as to speed up training and improve generalization)
  - (with convolutional layer with 3×3 and 5×5 filters) can act as two-layer neural networks, which can capture more complex patterns

GoogLeNet: Architecture

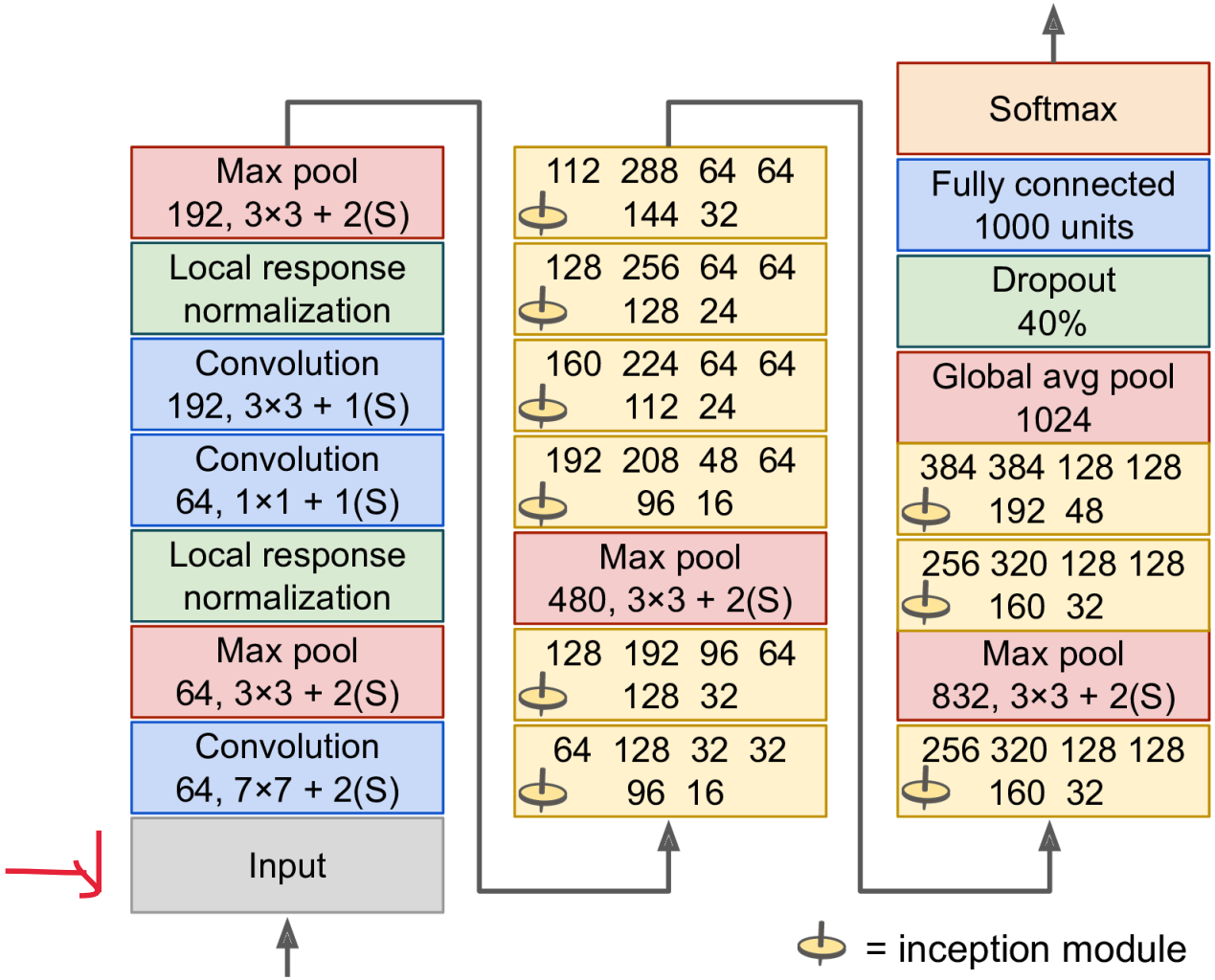


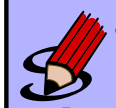
Figure 16: The architecture of GoogLeNet. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

GoogLeNet: Architecture

- In fig. 16:
  1. the first convolutional layer uses a large kernel size to preserve more information from the input image
  2. both the first convolutional layer and the max pooling layer divide the size of the input image by 4, so as to reduce the number of parameters
  3. the first local response normalization layer is used to address overfitting
  4. the next two convolutional layers act as a two-layer neural network, which can capture more complex features
  5. the same as the first local response normalization layer, the second layer is also used to address overfitting
  6. the same as the first max pooling layer, the second also divides the size of the input by 4, so as to reduce the number of parameters
  7. the next building block is a stack of 9 inception modules discussed earlier, interleaved with two max pooling layers (to reduce the number of parameters)
  8. the next *Global Average Pooling* layer outputs the mean of each feature map, to reduce the number of parameters and handle overfitting
  9. the next dropout layer is used to handle overfitting
  10. the output layer (with the softmax function) outputs the probability distribution over all the classes

ResNet

- ResNet was the winner of the 2015 ImageNet ILSVRC challenge, with a top-five error rate under 3.6%.
- A key reason for the improved performance is that:
  - on the one hand, ResNet uses an extremely deep CNN with 152 layers
  - on the other hand, ResNet uses *Skip Connections* (a.k.a., *Shortcut Connections*), so as to reduce the number of parameters



Takeaway

- A general trend in deep learning is that
  - models are getting deeper and deeper
  - parameters are getting fewer and fewer

Residual Learning

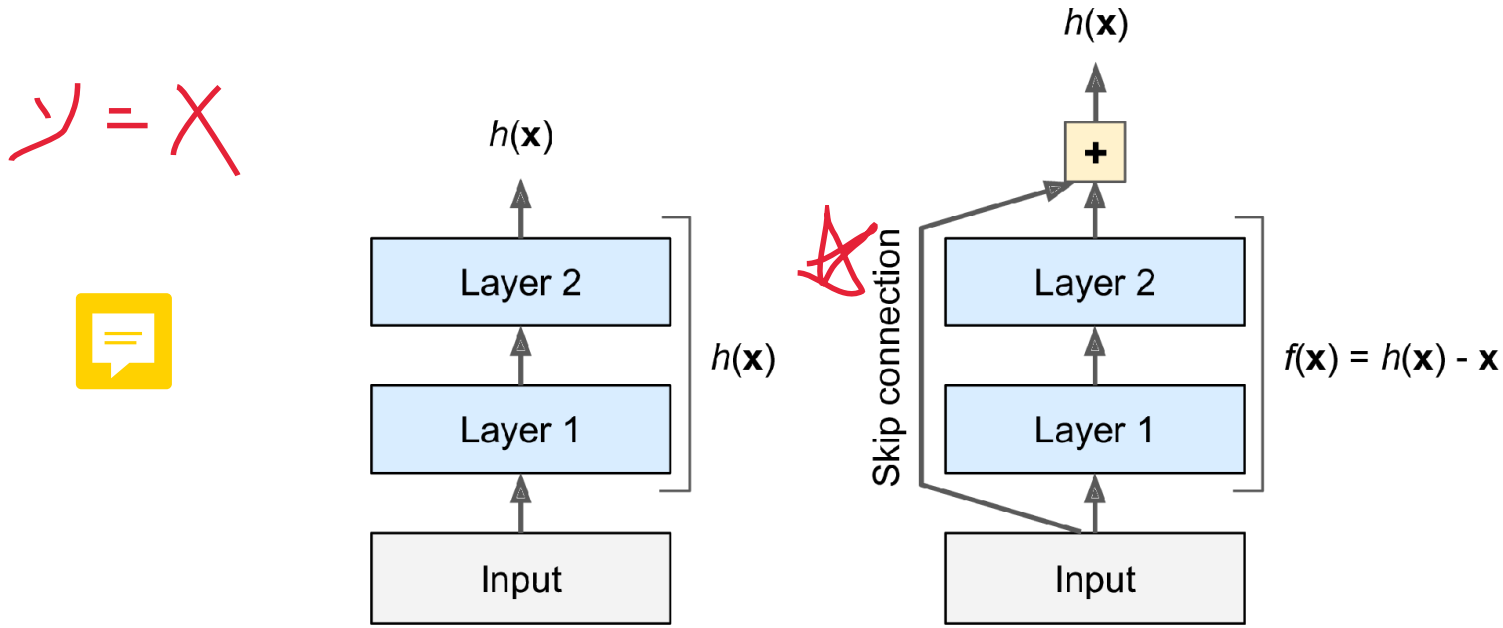


Figure 17: Residual learning. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Fig. 17 shows an oversimplified example illustrating the idea of residual learning:
  - it may take a while for layers 1 and 2 to model a function close to the identity function (where the output equals the input), as the weights are initialized close to zero, and in turn the output is close to zero
  - however, by adding a skip connection, we can add the input directly to the output, hence even when layers 1 and 2 have not started learning, they have already modeled the identity function fairly well (in turn speeds up training significantly)

Deep Residual Networks

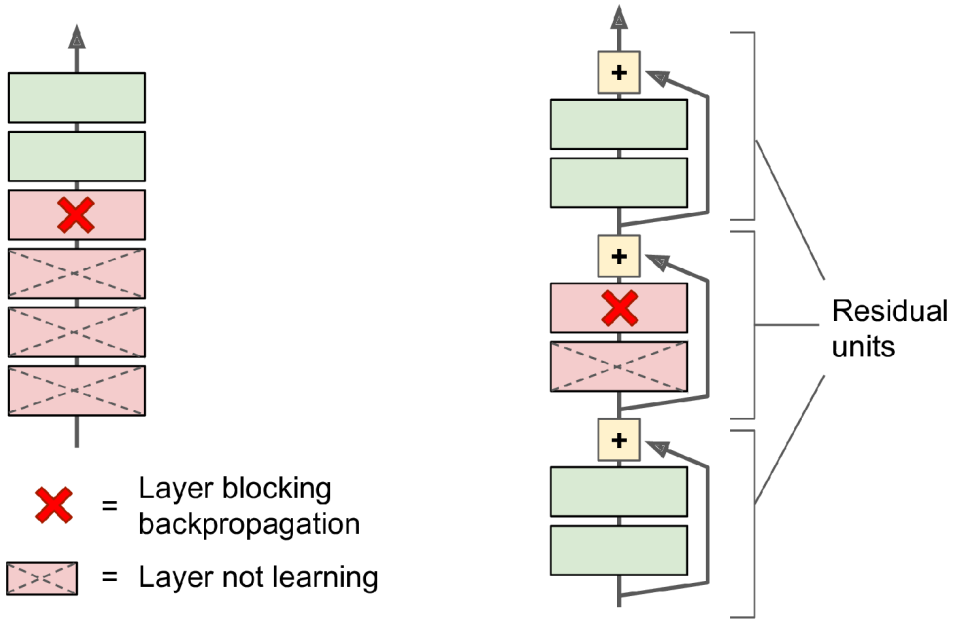


Figure 18: Deep residual networks. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- The key building blocks in ResNet are *Deep Residual Networks* (hence the name).
- Here, deep residual networks are a stack of *Residual Units*, where each residual unit is a small CNN with many skip connections (as shown in the right panel in fig. 18).
- The idea of deep residual networks is very similar to the idea of residual learning: the skip connections allow the convolutional layers to model functions close to the identity function, even when they have not started learning yet.

ResNet

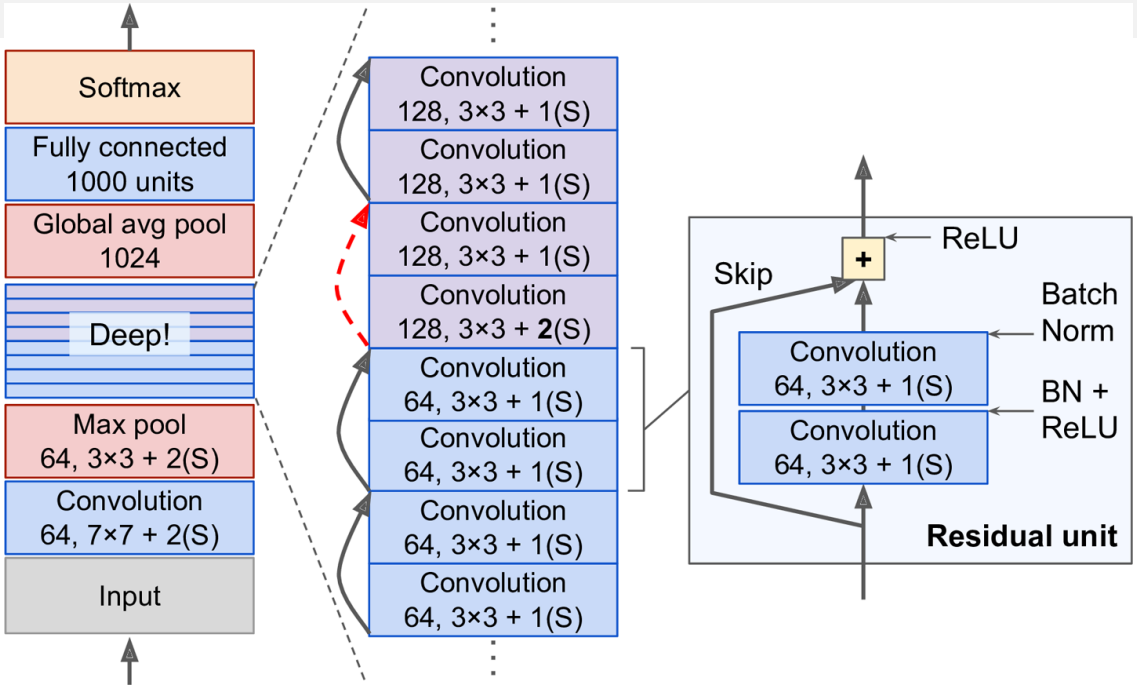


Figure 19: The architecture of ResNet. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- The lower and higher layers in ResNet are almost the same as those in GoogLeNet.
- However, unlike GoogLeNet which uses a stack of inception modules in the middle, ResNet uses a deep stack of simple residual units (as shown in fig. 19).
- Here, each residual unit is composed of two convolutional layers, where each layer:
  - uses ReLU as the activation function
  - is followed by a batch normalization layer

Skip Connection

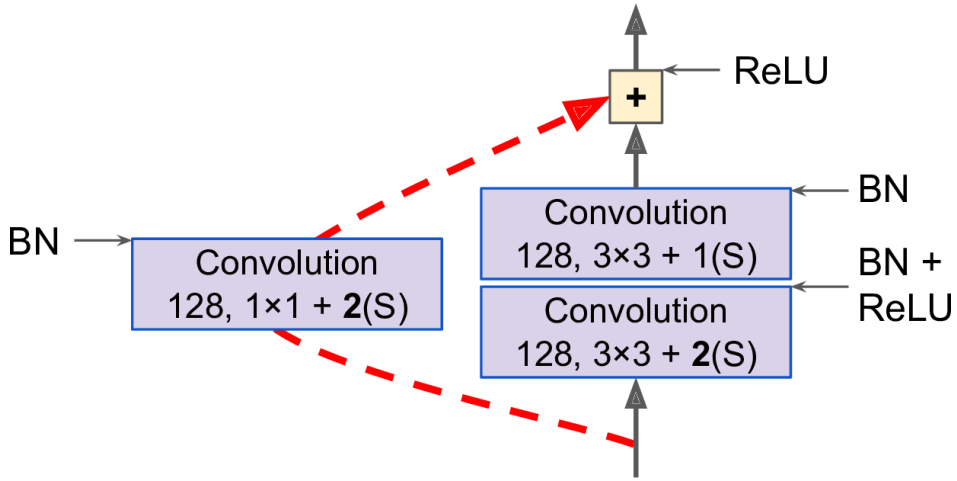


Figure 20: The skip connection. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition).*

- Fig. 19 shows that in every few residual units:
  - the number of sublayers in a convolutional layer is doubled
  - the height and width of the sublayers are halved (with stride 2)
- In this case, we cannot directly add the inputs to the outputs of the residual unit (as they do not have the same shape).
- To address this problem, we feed the inputs to a convolutional layer (see fig. 20) with:
  - 1×1 kernel (to mimic the identity function)
  - stride 2 (to produce the same shape as that of the outputs)
  - the same number of sublayers as those in the second convolutional layer

SENet

- SENet (Squeeze-and-Excitation Network) was the winner of the 2017 ImageNet ILSVRC challenge, with a top-five error rate of 2.25%.
- SENet can either extend the inception networks or ResNets:
  - the extension of the inception networks is called SE-Inception
  - the extension of the ResNets is called SE-ResNet
- The extension is done by adding a small neural network named *SE Block* to every inception module or residual unit (see fig. 21).

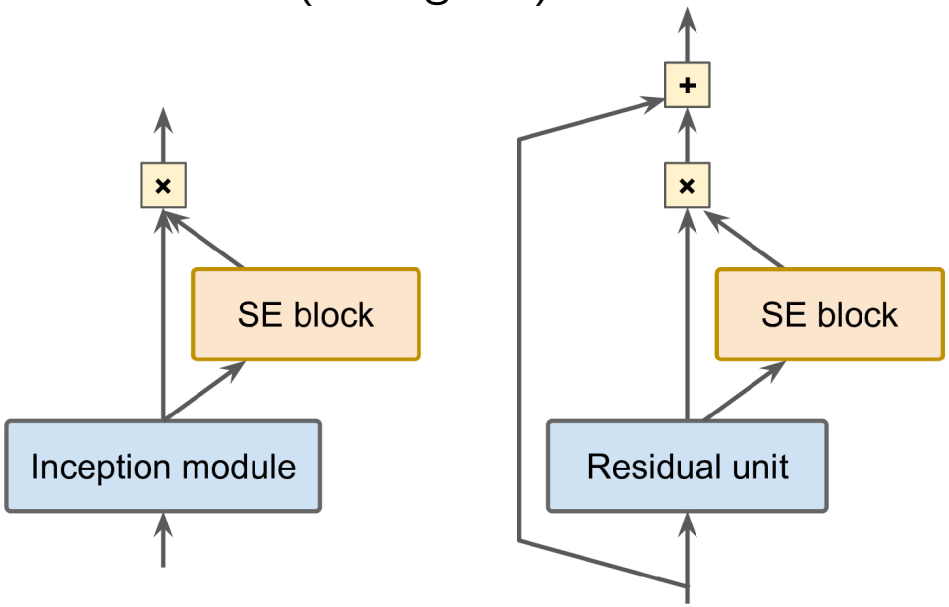


Figure 21: The SE block. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition).*

SE Block: Idea

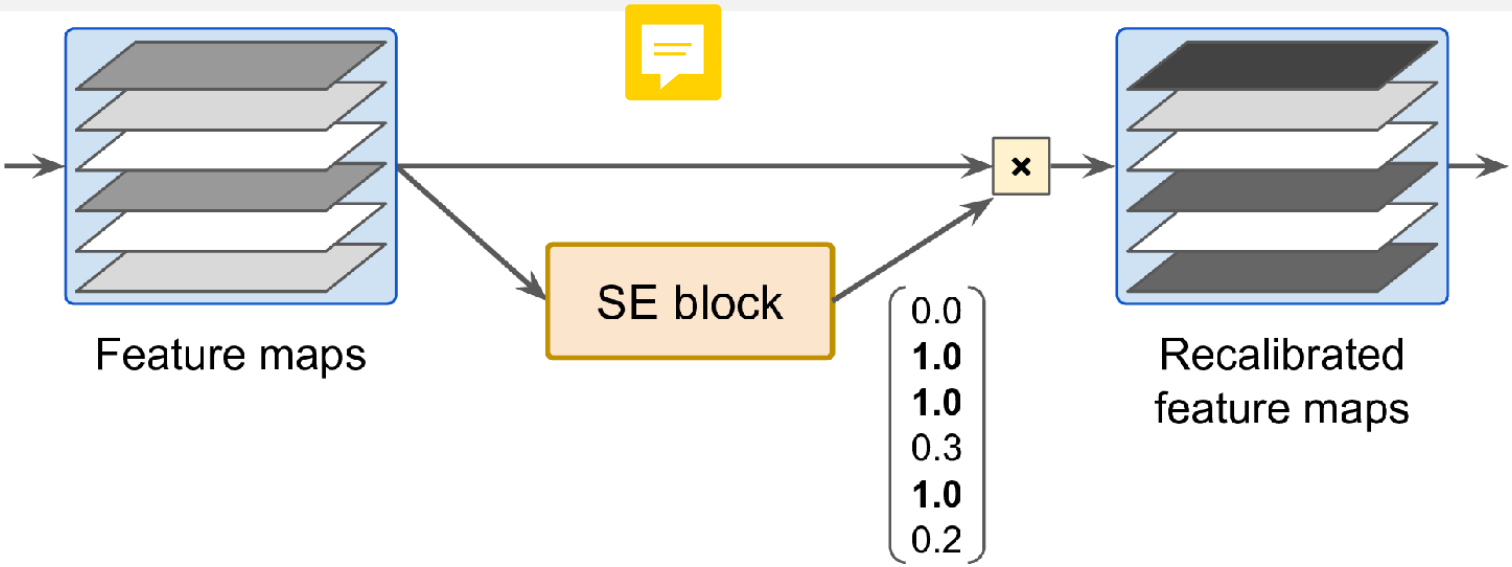


Figure 22: Recalibrating the feature maps using the SE block. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- The SE block takes as input feature maps and outputs recalibrated feature maps (see fig. 22).
- Concretely, the SE block only aims to capture the pattern along the depth dimension, that is, which features are usually activated together.
- For example:
  - a SE block may detect eyes and nodes usually appear together in images
  - if the SE block finds the eyes feature map activated, it will boost the nodes feature map, which in turn, improves prediction accuracy

SE Block: Architecture

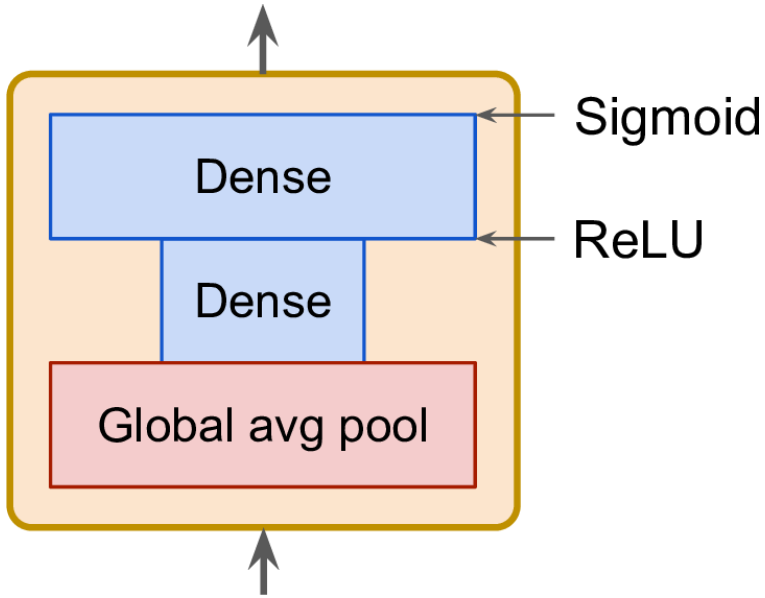


Figure 23: Architecture of the SE block. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Fig. 23 shows the three components in a SE block:
  - a global average pooling layer: to calculate the mean of each feature map
  - a dense hidden layer: to "squeeze" information by using much fewer perceptrons than the global pooling layer (e.g., by a factor of 16) so as to force the SE block to learn a general representation of the feature combinations
  - a dense output layer with sigmoid activation: to "excite" the feature maps that are usually activated together by producing a recalibration vector (with values between 0 and 1), which will be multiplied by the feature maps (see fig. 22)

Transfer Learning

- As we discussed previously, while in theory we can implement our own DNNs 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 DNNs to make it suitable for our data, an approach 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 CNNs will usually capture simple features that are common in many data (hence can be reused).

Building CNNs with Pretrained CNNs

- Here we can simply follow the good practice (for building DNNs with pretrained DNNs) discussed previously (also shown below).

Good practice

- To build 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 our data is to the data where the model was pretrained, the more lower layers of the pretrained model we should reuse as the base.
- It is even possible to reuse all the hidden layers of a pretrained model, 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 model.

Training CNN with Pretrained CNN

- Here we can simply follow the good practice (for training DNNs with pretrained DNNs) 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

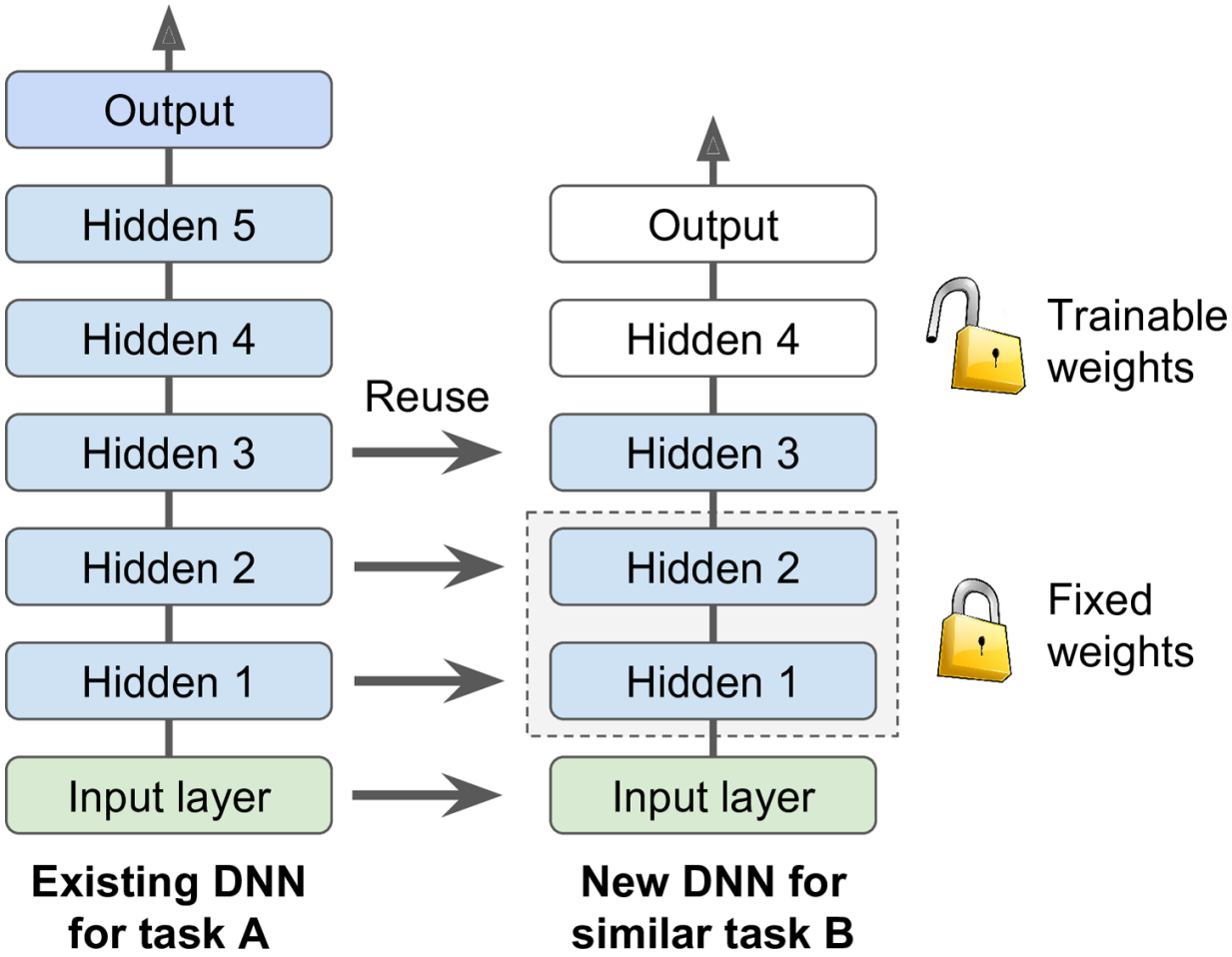


Figure 24: Building and training DNN with pretrained model. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition).*

Building CNN with Pretrained CNN: Code Example

- See /p3_c2_s3_convolutional_neural_networks/case_study:
  1. cells 14 to 17
  2. cell 20

Freezing the Pretrained Layers: Code Example

- See /p3_c2_s3_convolutional_neural_networks/case_study:
  1. cell 21

Unfreezing the Pretrained Layers: Code Example

- See /p3_c2_s3_convolutional_neural_networks/case_study:
  1. cell 27