Classifying Images with Deep Convolutional Neural Networks

Aprendizaje Automático

Ingeniería de Robótica Software

Universidad Rey Juan Carlos

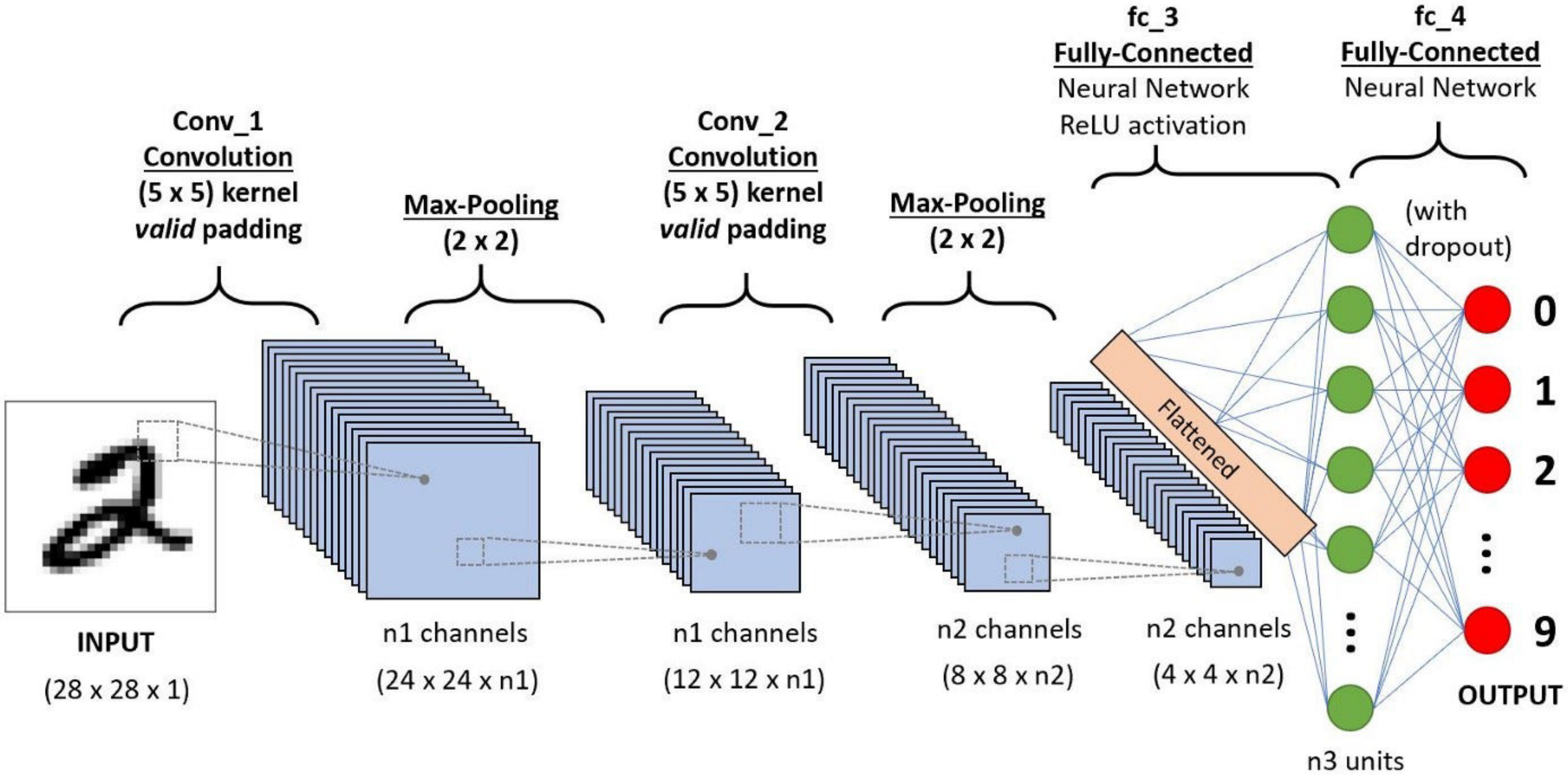
The building blocks of CNNs

- Convolutional Neural Networks (**CNNs**) are a **family of models** that were originally **inspired by how the visual cortex** of the human brain works when **recognizing objects**.
- The **development** of CNNs goes back to the **1990s**, when **Yann LeCun** and his colleagues **proposed a novel NN architecture** for classifying handwritten digits from images.
- **Due to the outstanding performance** of CNNs **for image classification** tasks, this particular type of feedforward NN gained a **lot of attention and led to tremendous improvements in machine learning for computer vision**.
- Several years later, in **2019**, **Yann LeCun received the Turing award** (the most prestigious award in computer science) for his contributions to the field of artificial intelligence (AI), **along with** two other researchers, **Yoshua Bengio and Geoffrey Hinton**.

Understanding CNNs and feature hierarchies

- Successfully **extracting relevant features** is **key to the performance of any machine learning algorithm**, and **traditional machine learning** models **rely on input features that may come from a domain expert or** are based on computational **feature selection or extraction techniques**.
- Certain types of NNs, such as **CNNs**, can **automatically learn the features from raw data that are most useful** for a particular task. For this reason, it's common to consider **CNN layers as feature extractors**:
  - The **early layers** (those right after the input layer) **extract low-level features** from raw data, and the **later layers** (often fully connected layers, as in a multilayer perceptron (MLP)) **use these features to predict** a continuous target value or class label.

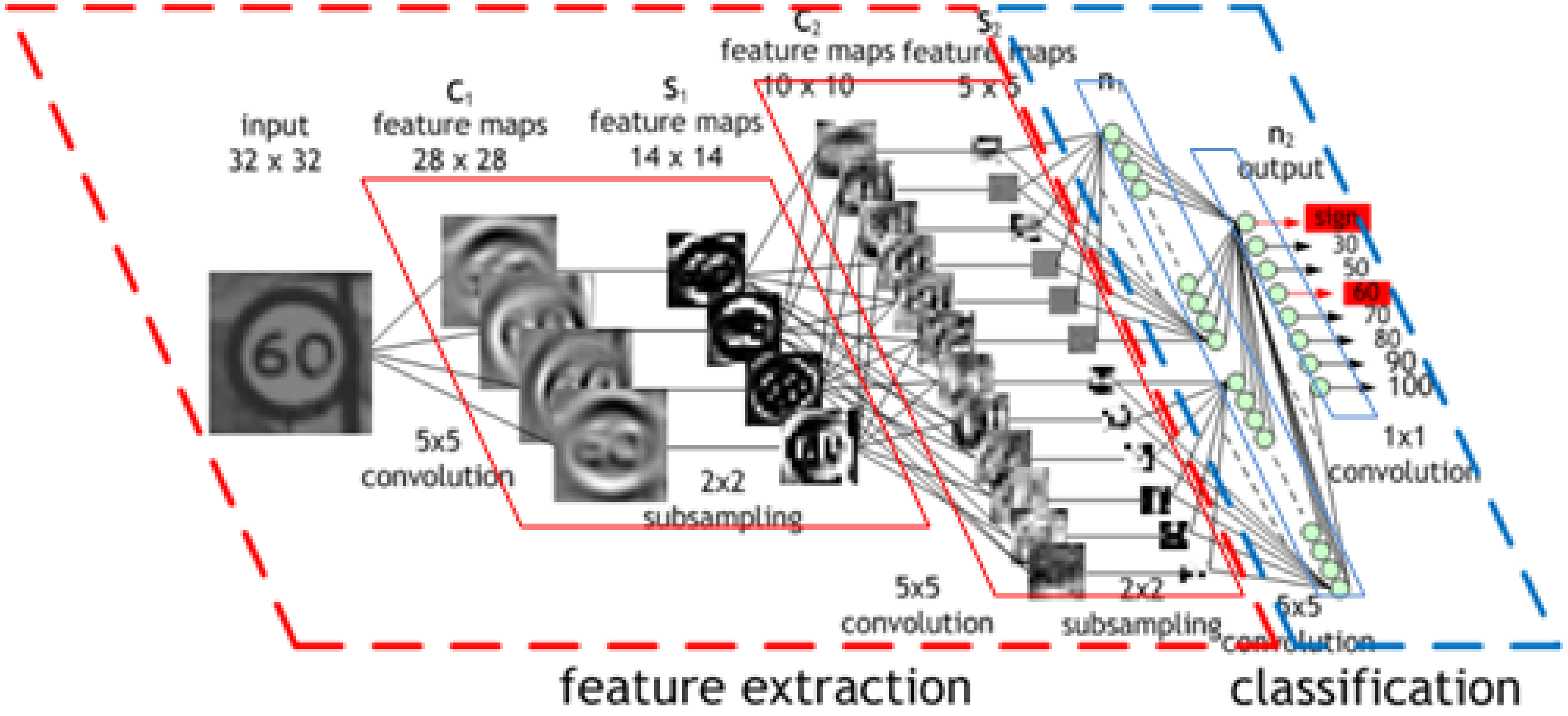
Understanding CNNs and feature hierarchies



Understanding CNNs and feature hierarchies

- Certain types of multilayer NNs, and in particular, **deep CNNs**, **construct** a so-called **feature hierarchy** by combining the **low-level features** in a layer-wise fashion to form **high-level features**.
- For example, if we're dealing with images, then **low-level features, such as edges and blobs**, are **extracted from the earlier layers**, which are combined to form high-level features.
- These **high-level features** can form more complex shapes, such as the **general contours of objects** like buildings, cats, or dogs.

Understanding CNNs and feature hierarchies



Understanding CNNs and feature hierarchies

- A **CNN computes feature maps** from an input image, **where each element comes from a local patch of pixels** in the input image.

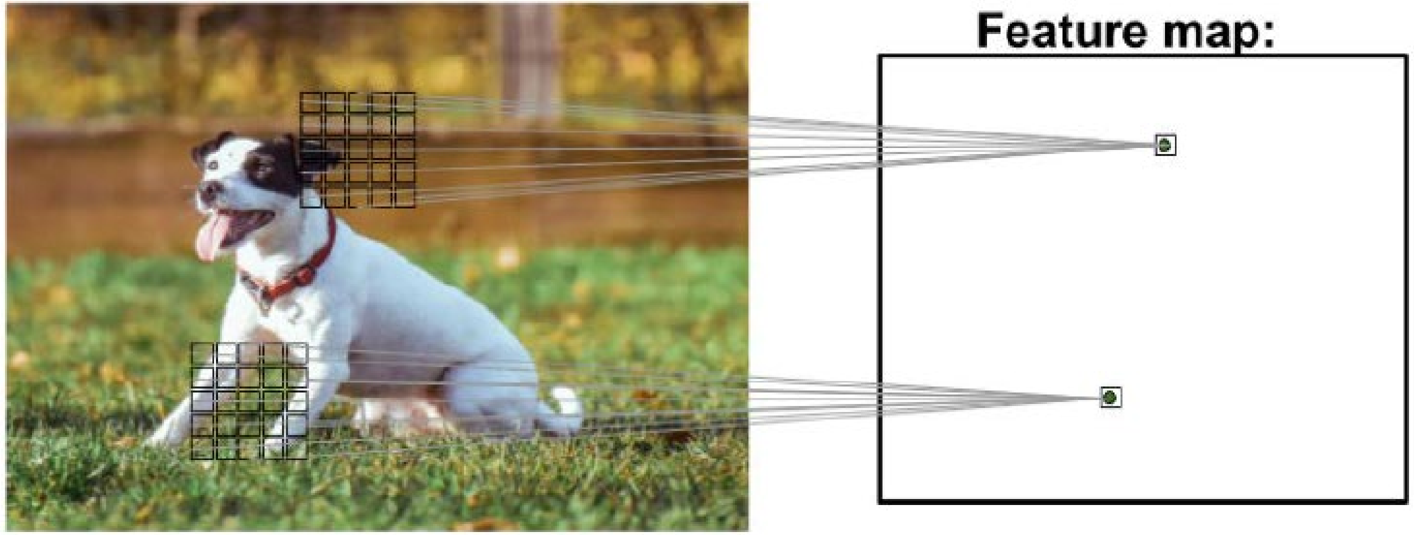


Figure 14.1: Creating feature maps from an image (photo by Alexander Dummer on Unsplash)

Understanding CNNs and feature hierarchies

- **CNNs perform very well on image-related tasks**, and that's largely **due to two** important **ideas**:
  - **<u>Sparse connectivity</u>**: A **single element in the feature map is connected to only a small patch of pixels** (this is very different from connecting to the whole input image, as in the case of MLPs)
  - **<u>Parameter sharing</u>**: The **same weights are used for different patches** of the input image.
- As a direct consequence of these two ideas, replacing a conventional, fully connected MLP with a **convolution layer substantially decreases the number of weights (parameters) in the network**, and we will see an improvement in the ability to capture relevant features.
- In the context of image data, it makes sense to **assume that nearby pixels are typically more relevant to each other than pixels that are far away** from each other.

Understanding CNNs and feature hierarchies

- Typically, **CNNs are composed of several convolutional and subsampling layers** that **are followed by one or more fully connected layers at the end**. The fully connected layers are essentially an MLP.
- **Subsampling layers**, commonly known as **pooling layers**, **do not have any learnable parameters**.
  - There are **no weights or bias** units in pooling layers.
  - However, both the **convolutional and fully connected layers have weights and biases that are optimized during training**.

Performing discrete convolutions

- To understand how convolution operations work, let’s start with a **convolution in one dimension**, which is sometimes **used for** working with certain types of sequence data, such as **text**.
- A **discrete convolution** (or simply convolution) is a fundamental operation in a CNN.

Discrete convolutions in one dimension

- **A discrete convolution for two vectors**, ***x*** and ***w***, is denoted by ***y* = *x* * *w***, in which vector ***x*** is our input (sometimes called signal) and ***w*** is called the **filter or kernel**.
- A discrete convolution is mathematically defined as follows:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]\, w[k]$$

Discrete convolutions in one dimension

- The fact that the **sum runs through indices from –∞ to +∞** seems odd, mainly because in machine learning applications, **we always deal with finite feature vectors**.
- For example, if ***x*** has 10 features with indices 0, 1, 2, ..., 8, 9, then indices –∞:-1 and 10:+∞ are out of bounds for ***x***.
- Therefore, to correctly compute the summation shown in the preceding formula, it is **assumed** that ***x*** and ***w*** are **filled with zeros**.
- This will result in an output vector, ***y***, that also **has infinite size**, with lots of zeros as well.
  - Since this is not useful in practical situations, ***x* is padded only with a finite number of zeros**.

Discrete convolutions in one dimension

- **This process is called zero-padding or** simply **padding**.
- Here, the **number of zeros padded on each side** is denoted by *p*.

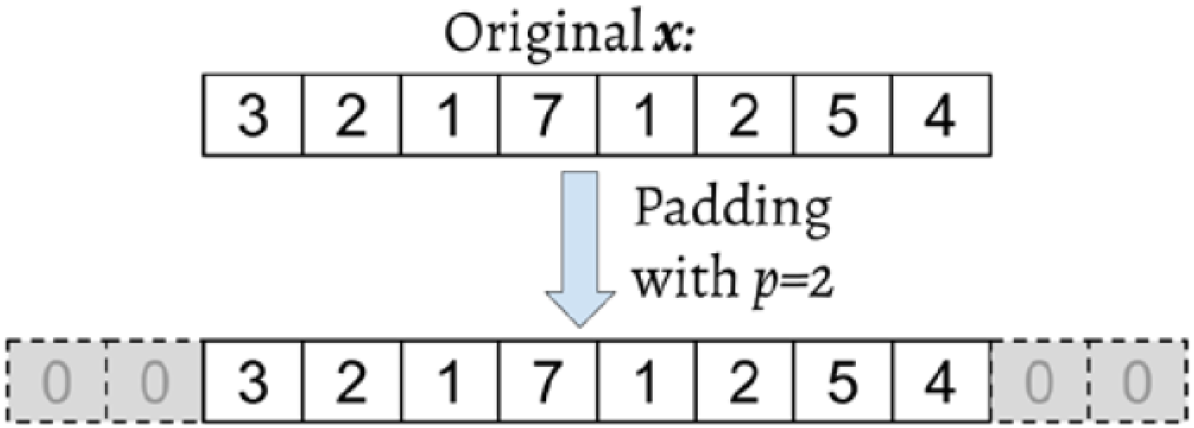


Figure 14.2: An example of padding

Discrete convolutions in one dimension

- Let's assume that the original input, $\boldsymbol{x}$, and filter, $\boldsymbol{w}$, have n and m elements, respectively, where $m \leq n$.
- The **padded vector**, $\boldsymbol{x}^{\boldsymbol{p}}$, has size $n + 2p$. The practical formula for computing a discrete convolution will change to the following:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \rightarrow y[i] = \sum_{k=0}^{k=m-1} x^p[i + m - k]\, w[k]$$

Discrete convolutions in one dimension

- **Example** that the padding size is zero (*p=0*):

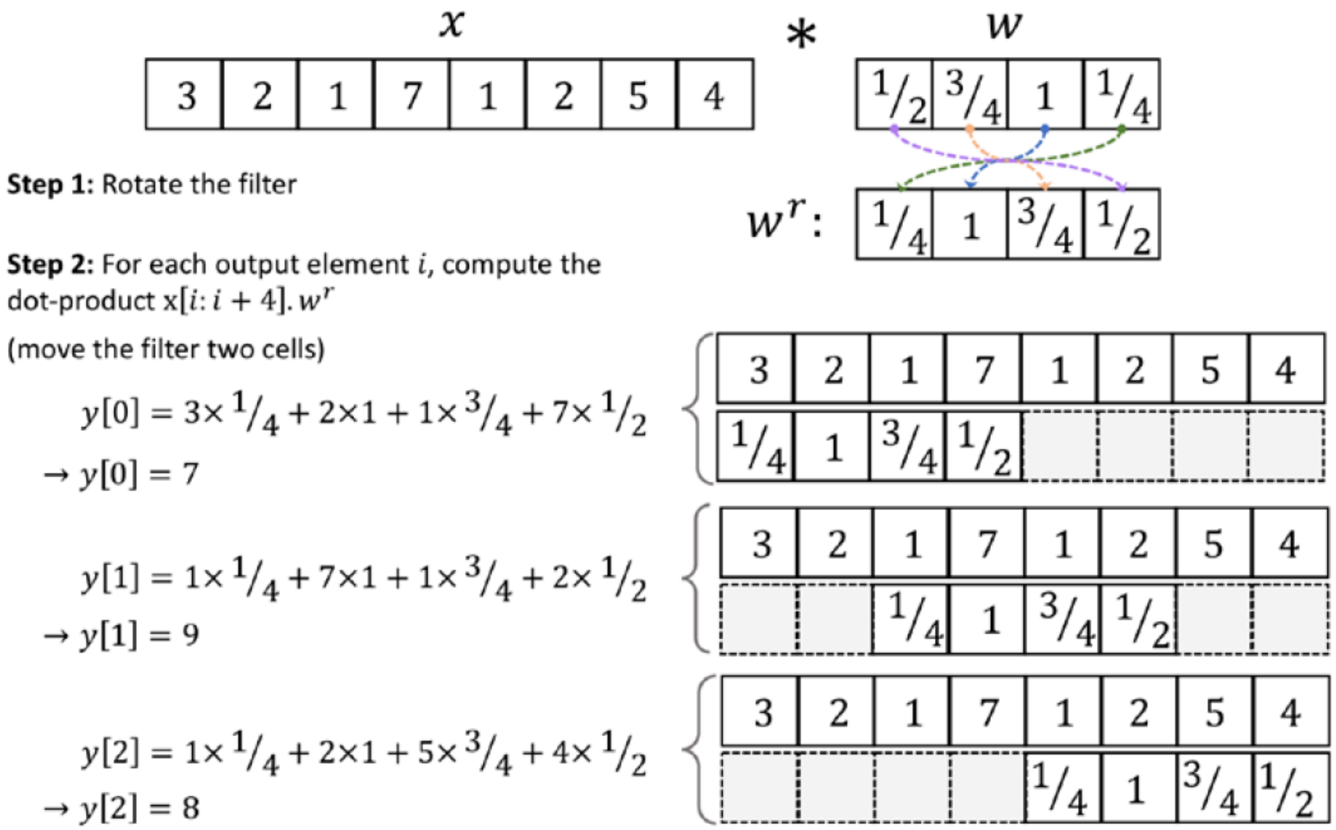


Figure 14.3: The steps for computing a discrete convolution

- Notice that the rotated filter, w^r, is shifted by two cells each time we **shift**.
- This shift is another hyperparameter of a convolution, the **stride**, *s*.
- In this example, the stride is two, *s* = 2.
- Note that the stride has to be a positive number smaller than the size of the input vector.

Discrete convolutions in one dimension

Cross-correlation

Cross-correlation (or simply correlation) between an input vector and a filter is denoted by $\boldsymbol{y} = \boldsymbol{x} \star \boldsymbol{w}$ and is very much like a sibling of a convolution, with a small difference: in cross-correlation, the multiplication is performed in the same direction. Therefore, it is not a requirement to rotate the filter matrix, w, in each dimension. Mathematically, cross-correlation is defined as follows:

$$\boldsymbol{y} = \boldsymbol{x} \star \boldsymbol{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i+k]\, w[k]$$

The same rules for padding and stride may be applied to cross-correlation as well. Note that most deep learning frameworks (including PyTorch) implement cross-correlation but refer to it as convolution, which is a common convention in the deep learning field.

Padding inputs to control the size of the output feature maps

- There are **three modes of padding** that are commonly used in practice: **full**, **same**, and **valid**.
  - In **<u>full</u> mode**, the padding parameter, p, is set to $p = m - 1$. Full padding **increases the dimensions of the output**; thus, it is **rarely used in CNN** architectures.
  - The **<u>same</u> padding mode** is usually used to ensure that the **output vector has the same size as the input vector**, x.
    - In this case, **the padding parameter**, p, is **computed according to the filter size**, along with the requirement that the input size and output size are the same.
  - Finally, computing a convolution in **<u>valid</u> mode** refers to the case where $p = 0$ (no padding).

Padding inputs to control the size of the output feature maps

- The **most commonly used padding** mode in CNNs is **same** padding.
- One of its **advantages** over the other padding modes is that **same padding preserves the size of the vector** which makes designing a network architecture more convenient.

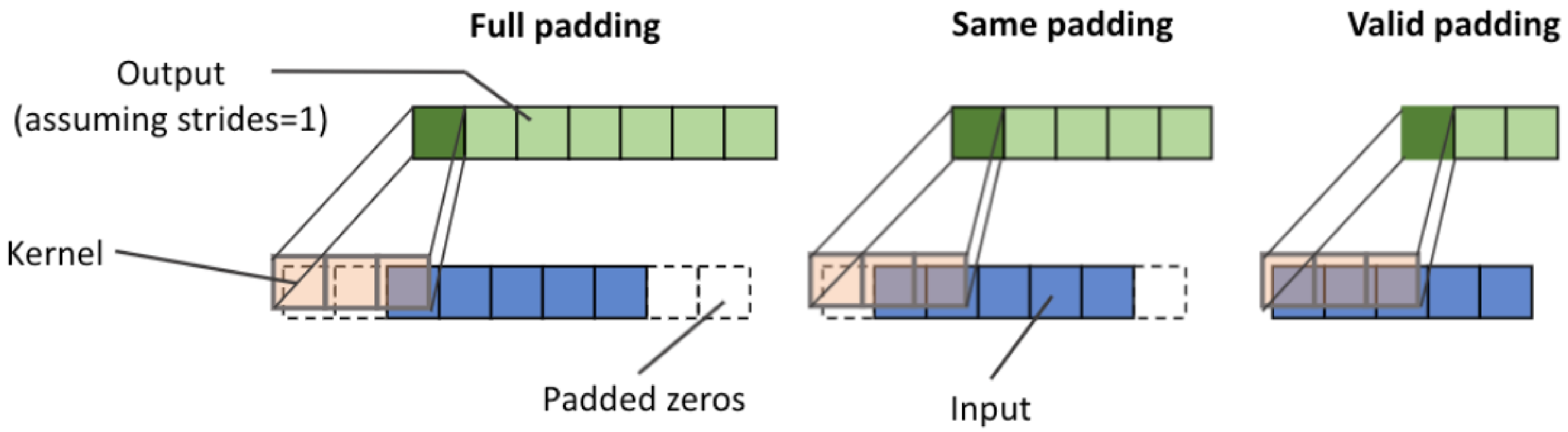


Figure 14.4: The three modes of padding

Determining the size of the convolution output

- Let's assume that the input vector is of size *n* and the filter is of size *m*.
- Then, the **size of the output** resulting from ***y* = *x* * *w***, with padding *p* and stride *s*, would be determined as follows:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

Here, $\lfloor \cdot \rfloor$ denotes the *floor* operation.

Determining the size of the convolution output

Consider the following two cases:

- Compute the output size for an input vector of size 10 with a convolution kernel of size 5, padding 2, and stride 1:

  $$n = 10, m = 5, \quad p = 2, \quad s = 1 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 5}{1} \right\rfloor + 1 = 10$$

  (Note that in this case, the output size turns out to be the same as the input; therefore, we can conclude this to be same padding mode.)

- How does the output size change for the same input vector when we have a kernel of size 3 and stride 2?

  $$n = 10, m = 3, \quad p = 2, \quad s = 2 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 3}{2} \right\rfloor + 1 = 6$$

Performing a discrete convolution in 2D

- When we deal with **2D inputs**, such as a matrix, $\boldsymbol{X}_{n1\times n2}$, and the filter matrix, $\boldsymbol{W}_{m1\times m2}$, where $m1 \le n1$ and $m2 \le n2$, then the matrix $\boldsymbol{Y}$=$\boldsymbol{X}$*$\boldsymbol{W}$ is the result of a 2D convolution between $\boldsymbol{X}$ and $\boldsymbol{W}$. This is defined mathematically as follows:

$$\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W} \rightarrow Y[i,j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i-k_1, j-k_2]\, W[k_1, k_2]$$

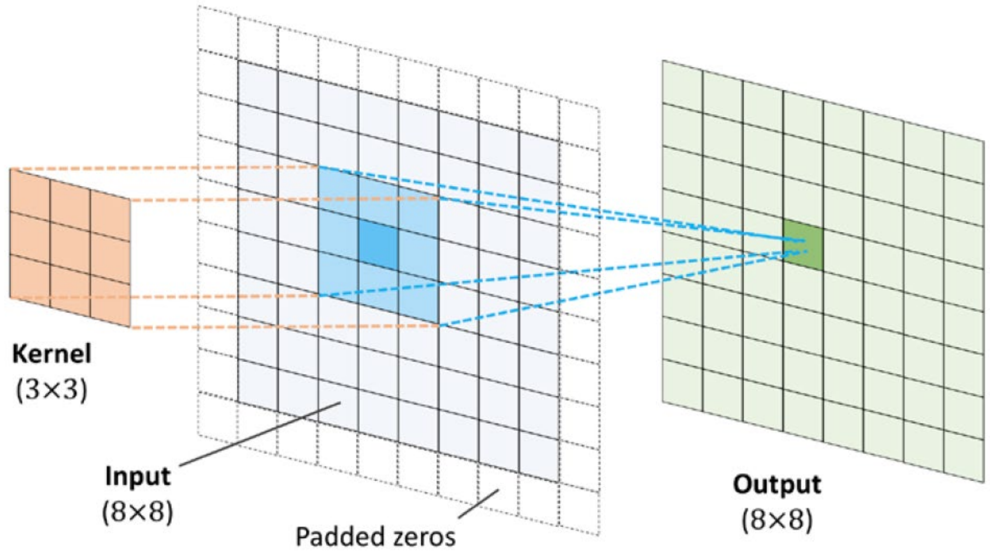


Figure 14.5: The output of a 2D convolution

Performing a discrete convolution in 2D

The following example illustrates the computation of a 2D convolution between an input matrix, $X_{3\times3}$, and a kernel matrix, $W_{3\times3}$, using padding $p = (1, 1)$ and stride $s = (2, 2)$. According to the specified padding, one layer of zeros is added on each side of the input matrix, which results in the padded matrix $\boldsymbol{X}^{\text{padded}}_{5\times5}$, as follows:

$$\boldsymbol{X}: \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 2 & 0 \\ 0 & 5 & 0 & 1 & 0 \\ 0 & 1 & 7 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \boldsymbol{W}: \begin{bmatrix} 0.5 & 0.7 & 0.4 \\ 0.3 & 0.4 & 0.1 \\ 0.5 & 1 & 0.5 \end{bmatrix}$$

$$\boldsymbol{W}^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

Performing a discrete convolution in 2D

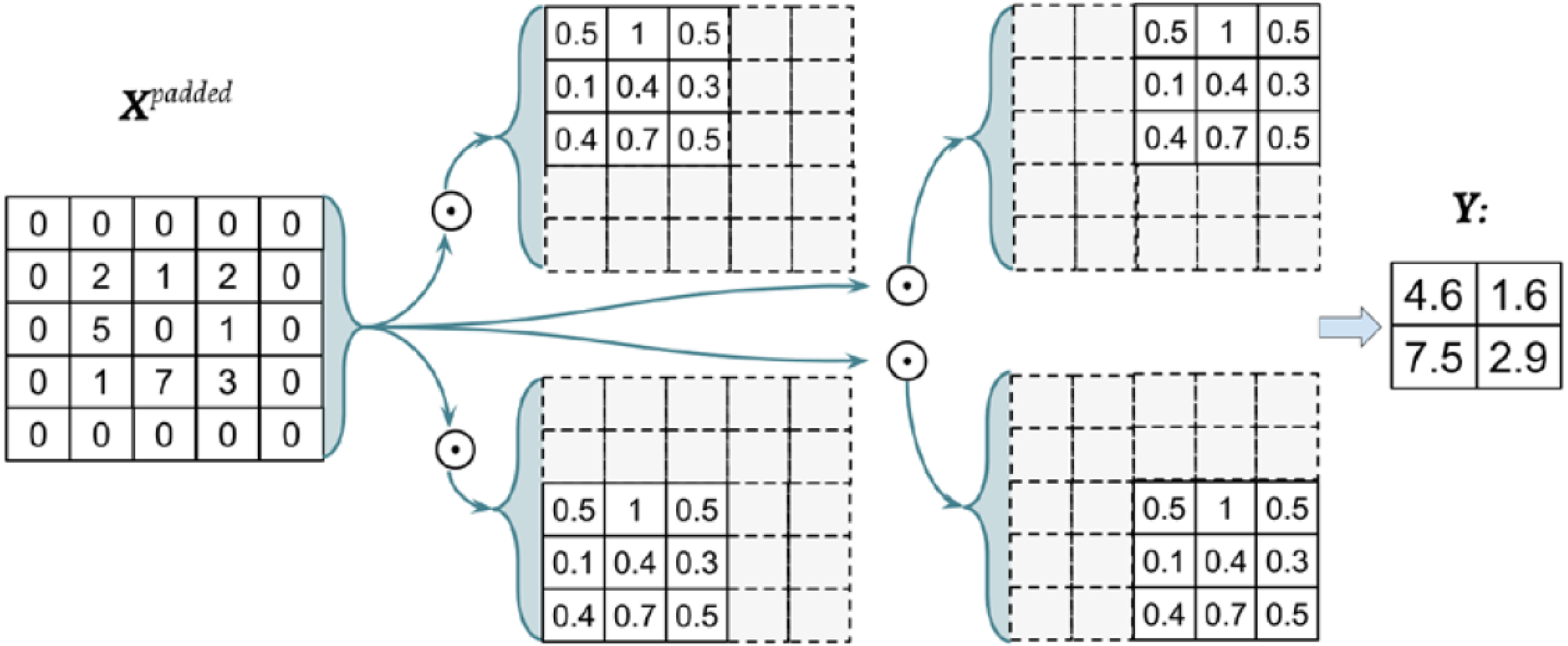


Figure 14.7: Computing the sum of the element-wise product

Subsampling or pooling layers

- **Subsampling** is typically applied in **two forms of pooling** operations in CNNs:
  - **max-pooling**
  - **mean-pooling** (also known as average-pooling)
- The **pooling layer** is usually denoted by $P_{n1 \times n2}$. Here, the subscript determines the **size of the neighborhood** (the number of adjacent pixels in each dimension) where the max or mean operation is performed. We refer to such a neighborhood as the **pooling size**.

Subsampling or pooling layers

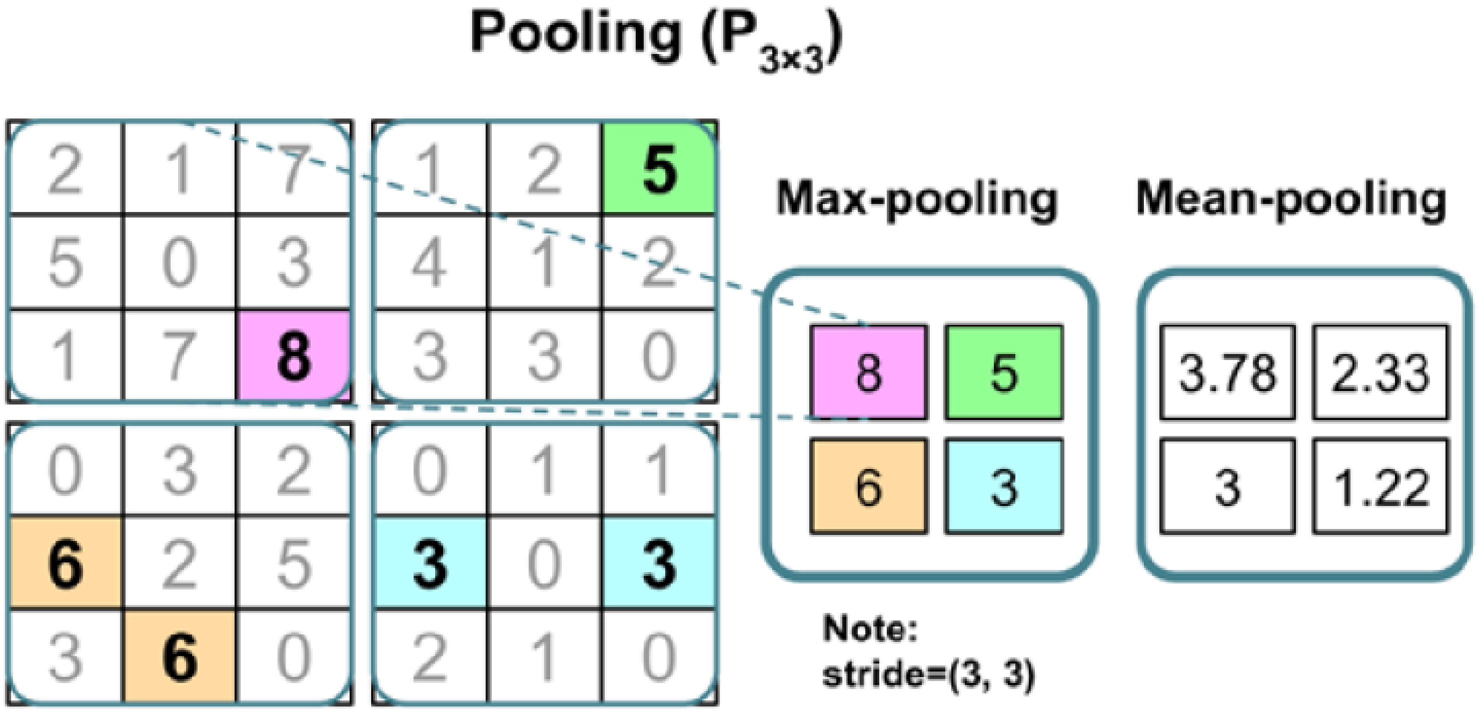


Figure 14.8: An example of max-pooling and mean-pooling

Subsampling or pooling layers

- Pooling (**max-pooling**) introduces a **local invariance**. This means that **small changes in a local neighborhood do not change the result** of max-pooling.
  - Therefore, it helps with **generating features that are more robust to noise** in the input data.
  - Refer to the following **example**, which shows that the max-pooling of two different input matrices, X_1 and X_2, results in the same output:

$$\left.\begin{aligned} \boldsymbol{X}_1 &= \begin{bmatrix} 10 & 255 & 125 & 0 & 170 & 100 \\ 70 & 255 & 105 & 25 & 25 & 70 \\ 255 & 0 & 150 & 0 & 10 & 10 \\ 0 & 255 & 10 & 10 & 150 & 20 \\ 70 & 15 & 200 & 100 & 95 & 0 \\ 35 & 25 & 100 & 20 & 0 & 60 \end{bmatrix} \\ \\ \boldsymbol{X}_2 &= \begin{bmatrix} 100 & 100 & 100 & 50 & 100 & 50 \\ 95 & 255 & 100 & 125 & 125 & 170 \\ 80 & 40 & 10 & 10 & 125 & 150 \\ 255 & 30 & 150 & 20 & 120 & 125 \\ 30 & 30 & 150 & 100 & 70 & 70 \\ 70 & 30 & 100 & 200 & 70 & 95 \end{bmatrix} \end{aligned}\right\} \xrightarrow{\text{max pooling } P_{2\times 2}} \begin{bmatrix} 255 & 125 & 170 \\ 255 & 150 & 150 \\ 70 & 200 & 95 \end{bmatrix}$$

  - **Pooling decreases the size of features**, which results in higher computational efficiency. Also, reducing the number of features may reduce the degree of overfitting as well.

Subsampling or pooling layers

- Traditionally, **pooling is assumed to be non-overlapping**.
- Pooling is typically performed on non-overlapping neighborhoods, which can be done by **setting the stride parameter equal to the pooling size**. For example, a non-overlapping pooling layer, $P_{n1\times n2}$, requires a stride parameter s = (n1, n2).

- **While pooling is still an essential part of many CNN architectures, several CNN architectures have also been developed without using pooling** layers.
  - Instead of using pooling layers to reduce the feature size, researchers use **convolutional layers with a stride of 2**.

Working with multiple input or color channels

- **An input to a convolutional layer** may **contain one or more 2D arrays** or matrices with dimensions N1×N2 (for example, the image height and width in pixels).
- These N1×N2 matrices are called **channels**.
- Conventional implementations of **convolutional layers expect a rank-3 tensor** representation as an input, for example, a three-dimensional array, $\boldsymbol{X}_{N1\times N2\times Cin}$, where **$C_{in}$ is the number of input channels**.
- **For example**, let's consider images as input to the first layer of a CNN. If the **image** is **colored** and uses the **RGB** color mode, then **Cin = 3** (for the red, green, and blue color channels in RGB).
- However, **if the image is in grayscale**, then we have **Cin=1**, because there is only one channel with the grayscale pixel intensity values.

Working with multiple input or color channels

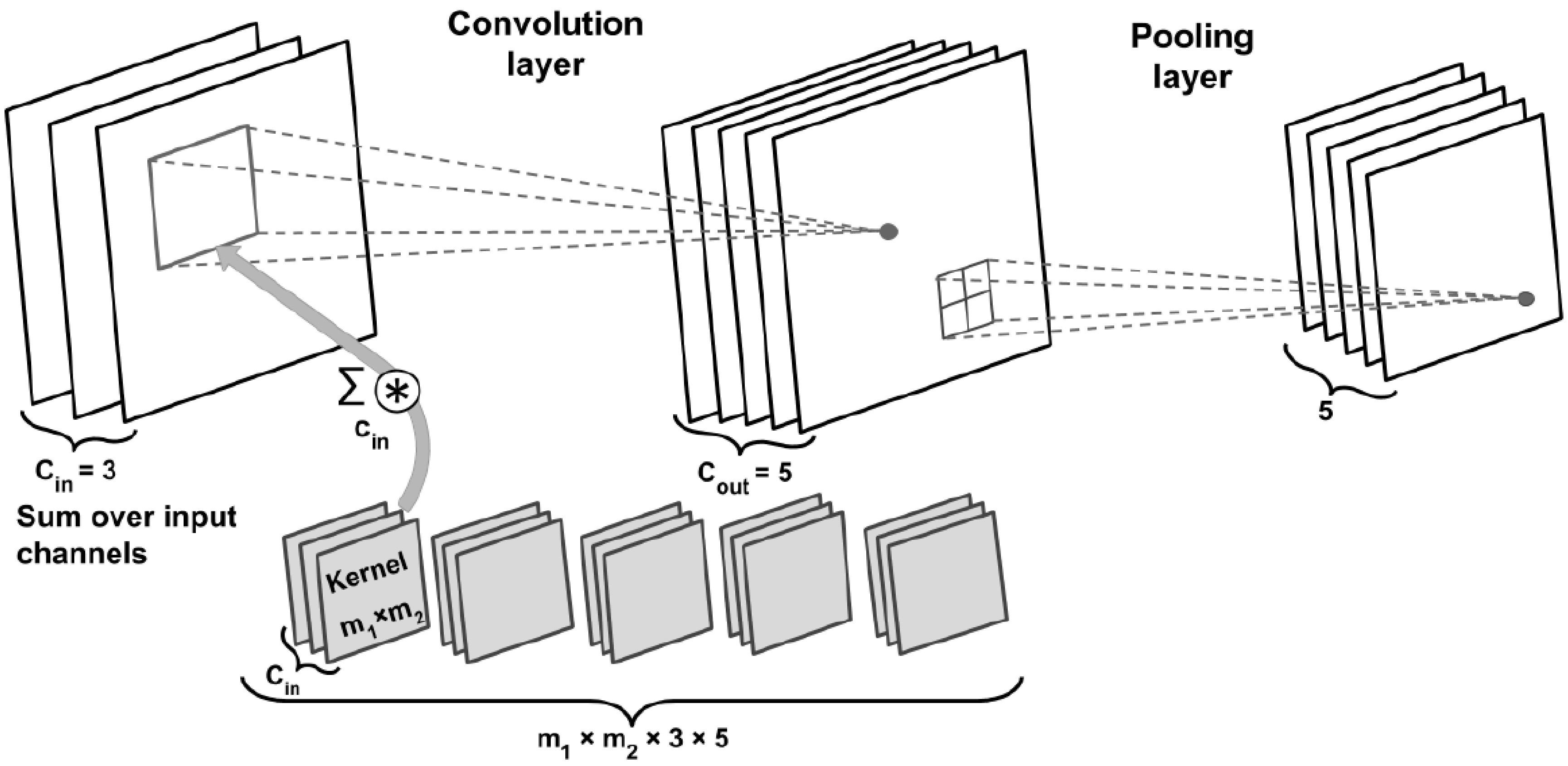


Figure 14.9: Implementing a CNN

Activation functions

- **Different activation functions**, such as **ReLU**, **sigmoid**, and **tanh**.
- Some of these activation functions, like **ReLU**, are mainly **used in the intermediate (hidden) layers** of an NN to add non-linearities to our model.
- Others, like **sigmoid (for binary) and softmax (for multiclass)**, are **added at the last (output) layer**, which **results in class-membership probabilities** as the output of the model.
- If the **sigmoid or softmax activations are not included at the output layer**, then the **model will compute the logits instead of the class-membership probabilities**.

Loss functions for classification

- Focusing on **classification problems**, depending on the type of problem (binary versus multiclass) and the type of output (logits versus probabilities), **we** should **choose the appropriate loss function to train our model**.
  - **Binary cross-entropy** is the loss function for a binary classification (with a single output unit).
  - **Categorical cross-entropy** is the loss function for multiclass classification.

Loss functions for classification

Loss function	Usage	Example *Using probabilities*	Example *Using logits*
`BCELoss` or `BCEWithLogitsLoss`	Binary classification	`BCELoss` *y_true:* 1 *y_pred:* 0.8	`BCEWithLogitsLoss` *y_true:* 1 *y_pred:* 0.8
`NLLLoss` or `CrossEntropyLoss`	Multiclass classification	`NLLLoss` *y_true:* 2 *y_pred:* 0.30 0.15 0.55	`CrossEntropyLoss` *y_true:* 2 *y_pred:* 1.5 0.8 2.1

Loss functions for classification

```
>>> ####### Binary Cross-entropy
>>> logits = torch.tensor([0.8])
>>> probas = torch.sigmoid(logits)
>>> target = torch.tensor([1.0])
>>> bce_loss_fn = nn.BCELoss()
>>> bce_logits_loss_fn = nn.BCEWithLogitsLoss()
>>> print(f'BCE (w Probas): {bce_loss_fn(probas, target):.4f}')
BCE (w Probas): 0.3711
>>> print(f'BCE (w Logits): '
...       f'{bce_logits_loss_fn(logits, target):.4f}')
BCE (w Logits): 0.3711
```

```
>>> ####### Categorical Cross-entropy
>>> logits = torch.tensor([[1.5, 0.8, 2.1]])
>>> probas = torch.softmax(logits, dim=1)
>>> target = torch.tensor([2])
>>> cce_loss_fn = nn.NLLLoss()
>>> cce_logits_loss_fn = nn.CrossEntropyLoss()
>>> print(f'CCE (w Probas): '
...       f'{cce_logits_loss_fn(logits, target):.4f}')
CCE (w Probas): 0.5996
>>> print(f'CCE (w Logits): '
...       f'{cce_loss_fn(torch.log(probas), target):.4f}')
CCE (w Logits): 0.5996
```

Implementing a deep CNN using PyTorch

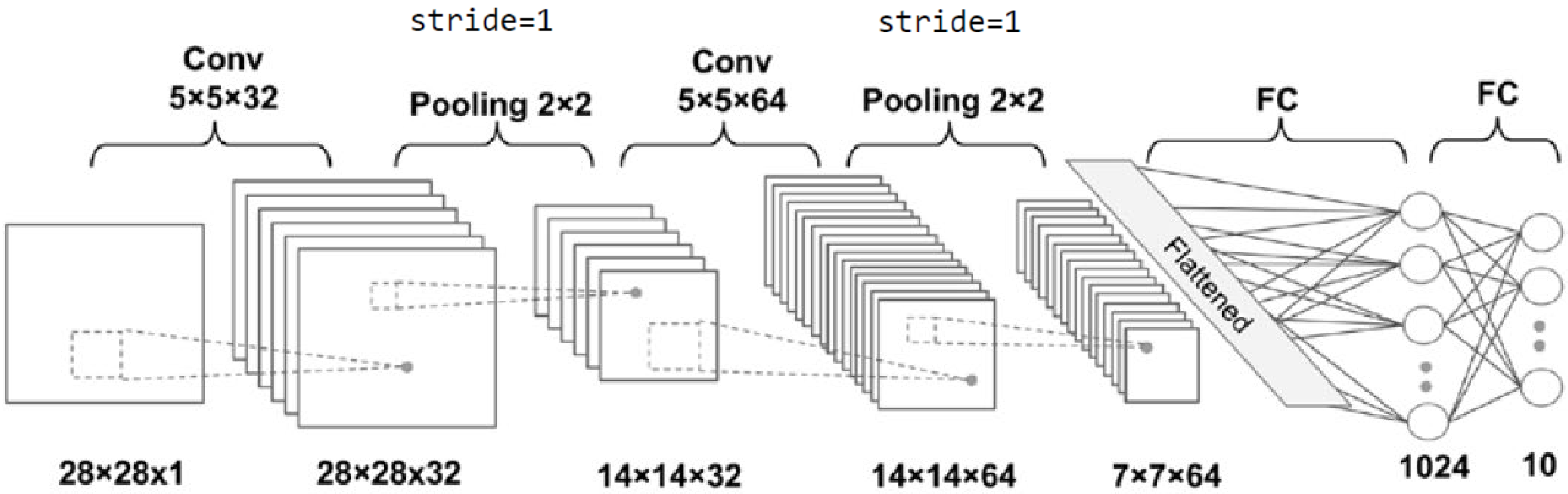


Figure 14.12: A deep CNN

- Input: [*batchsize*×28×28×1]
- Conv_1: [*batchsize*×28×28×32]
- Pooling_1: [*batchsize*×14×14×32]
- Conv_2: [*batchsize*×14×14×64]
- Pooling_2: [*batchsize*×7×7×64]
- FC_1: [*batchsize*×1024]
- FC_2 and softmax layer: [*batchsize*×10]

Inputs are in NCHW: Bathsize, Channel, Height, Width

Implementing a deep CNN using PyTorch

```
>>> import torchvision
>>> from torchvision import transforms
>>> image_path = './'
>>> transform = transforms.Compose([
...     transforms.ToTensor()
... ])
```

```
>>> mnist_dataset = torchvision.datasets.MNIST(
...     root=image_path, train=True,
...     transform=transform, download=True
... )
>>> from torch.utils.data import Subset
>>> mnist_valid_dataset = Subset(mnist_dataset,
...                              torch.arange(10000))
>>> mnist_train_dataset = Subset(mnist_dataset,
...                              torch.arange(
...                                  10000, len(mnist_dataset)
...                              ))
>>> mnist_test_dataset = torchvision.datasets.MNIST(
...     root=image_path, train=False,
...     transform=transform, download=False
... )
```

Implementing a deep CNN using PyTorch

```
>>> from torch.utils.data import DataLoader
>>> batch_size = 64
>>> torch.manual_seed(1)
>>> train_dl = DataLoader(mnist_train_dataset,
...                       batch_size,
...                       shuffle=True)
>>> valid_dl = DataLoader(mnist_valid_dataset,
...                       batch_size,
...                       shuffle=False)
```

Implementing a deep CNN using PyTorch

```
>>> model = nn.Sequential()
>>> model.add_module(
...     'conv1',
...     nn.Conv2d(
...         in_channels=1, out_channels=32,
...         kernel_size=5, padding=2
...     )
... )
>>> model.add_module('relu1', nn.ReLU())
>>> model.add_module('pool1', nn.MaxPool2d(kernel_size=2))
>>> model.add_module(
...     'conv2',
...     nn.Conv2d(
...         in_channels=32, out_channels=64,
...         kernel_size=5, padding=2
...     )
... )
>>> model.add_module('relu2', nn.ReLU())
>>> model.add_module('pool2', nn.MaxPool2d(kernel_size=2))
```

```
>>> x = torch.ones((4, 1, 28, 28))
>>> model(x).shape
torch.Size([4, 64, 7, 7])
```

By providing the input shape as a tuple `(4, 1, 28, 28)` (4 images within the batch, 1 channel, and image size 28×28), specified in this example, we calculated the output to have a shape `(4, 64, 7, 7)`, indicating feature maps with 64 channels and a spatial size of 7×7.

Implementing a deep CNN using PyTorch

```
>>> model.add_module('flatten', nn.Flatten())
>>> x = torch.ones((4, 1, 28, 28))
>>> model(x).shape
torch.Size([4, 3136])
```

```
>>> model.add_module('fc1', nn.Linear(3136, 1024))
>>> model.add_module('relu3', nn.ReLU())
>>> model.add_module('dropout', nn.Dropout(p=0.5))
>>> model.add_module('fc2', nn.Linear(1024, 10))
```

```
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Implementing a deep CNN using PyTorch

```
>>> def train(model, num_epochs, train_dl, valid_dl):
...     loss_hist_train = [0] * num_epochs
...     accuracy_hist_train = [0] * num_epochs
...     loss_hist_valid = [0] * num_epochs
...     accuracy_hist_valid = [0] * num_epochs
...     for epoch in range(num_epochs):
...         model.train()
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)
...             loss = loss_fn(pred, y_batch)
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_hist_train[epoch] += loss.item()*y_batch.size(0)
...             is_correct = (
...                 torch.argmax(pred, dim=1) == y_batch
...             ).float()
...             accuracy_hist_train[epoch] += is_correct.sum()
...         loss_hist_train[epoch] /= len(train_dl.dataset)
...         accuracy_hist_train[epoch] /= len(train_dl.dataset)
...
...         model.eval()
```

Implementing a deep CNN using PyTorch

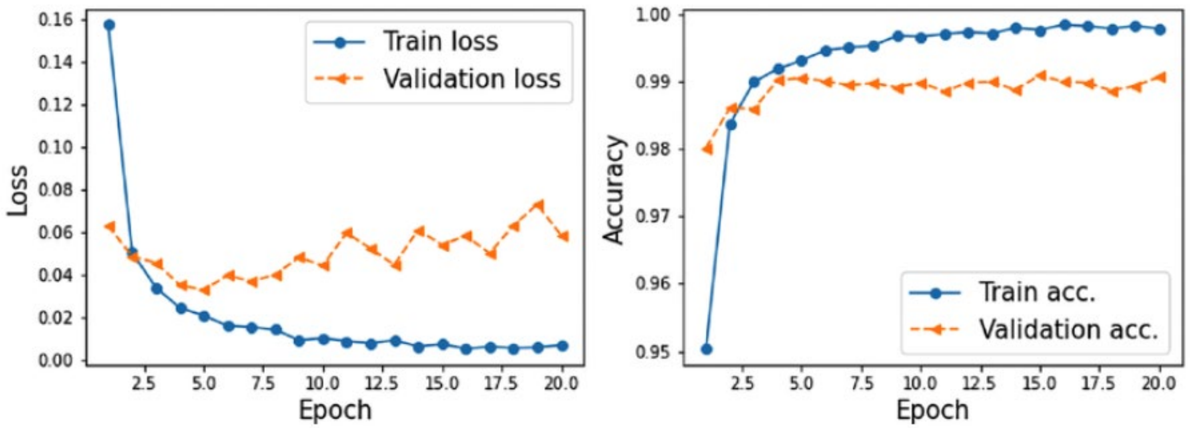
```
...         with torch.no_grad():
...             for x_batch, y_batch in valid_dl:
...                 pred = model(x_batch)
...                 loss = loss_fn(pred, y_batch)
...                 loss_hist_valid[epoch] += \
...                     loss.item()*y_batch.size(0)
...                 is_correct = (
...                     torch.argmax(pred, dim=1) == y_batch
...                 ).float()
...                 accuracy_hist_valid[epoch] += is_correct.sum()
...         loss_hist_valid[epoch] /= len(valid_dl.dataset)
...         accuracy_hist_valid[epoch] /= len(valid_dl.dataset)
...
...         print(f'Epoch {epoch+1} accuracy: '
...               f'{accuracy_hist_train[epoch]:.4f} val_accuracy: '
...               f'{accuracy_hist_valid[epoch]:.4f}')
...     return loss_hist_train, loss_hist_valid, \
...         accuracy_hist_train, accuracy_hist_valid
```

Implementing a deep CNN using PyTorch

```
>>> torch.manual_seed(1)
>>> num_epochs = 20
>>> hist = train(model, num_epochs, train_dl, valid_dl)
Epoch 1 accuracy: 0.9503 val_accuracy: 0.9802
...
Epoch 9 accuracy: 0.9968 val_accuracy: 0.9892
...
Epoch 20 accuracy: 0.9979 val_accuracy: 0.9907
```

```
>>> import matplotlib.pyplot as plt
>>> x_arr = np.arange(len(hist[0])) + 1
>>> fig = plt.figure(figsize=(12, 4))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(x_arr, hist[0], '-o', label='Train loss')
>>> ax.plot(x_arr, hist[1], '--<', label='Validation loss')
```

```
>>> ax.legend(fontsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(x_arr, hist[2], '-o', label='Train acc.')
>>> ax.plot(x_arr, hist[3], '--<',
...         label='Validation acc.')
>>> ax.legend(fontsize=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.set_ylabel('Accuracy', size=15)
>>> plt.show()
```



Implementing a deep CNN using PyTorch

```
>>> pred = model(mnist_test_dataset.data.unsqueeze(1) / 255.)
>>> is_correct = (
...     torch.argmax(pred, dim=1) == mnist_test_dataset.targets
... ).float()
>>> print(f'Test accuracy: {is_correct.mean():.4f}')
Test accuracy: 0.9914
```