Lecture 26: Classifying Images with Deep Convolutional Neural Networks (CNNs)

Dr. Huiping Cao

Outline

- The building blocks of CNN architectures
- Implementing deep CNNs in PyTorch

Convolutional Neural Networks (CNNs) - background

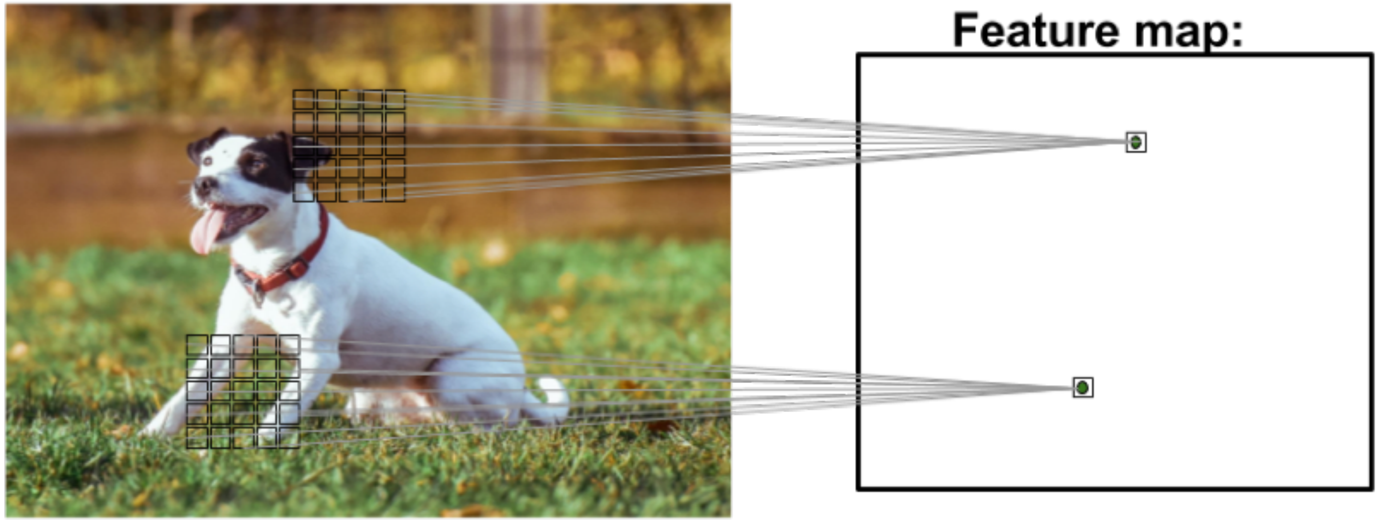
- CNNs are a family of models that were originally inspired by how the **visual cortex of the human brain works** when recognizing objects.
- The development of CNNs goes back to the 1990s.
  - *Handwritten Digit Recognition with a Back-Propagation Network* by *Y. LeCun*, and colleagues, 1989, published at the *Neural Information Processing Systems (NeurIPS)*
  - 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.
- CNNs have outstanding performance for image classification tasks.

Convolutional architecture as feature extraction layers

- Successfully extracting **salient** (**relevant**) **features** is key to the performance of any machine learning algorithm.
  - Input features that may come from a domain expert.
  - based on computational feature 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.
- It is common to consider CNN layers as feature extractors
  - The early layers (those right after the input layer) extract **low-level features** from raw data
  - 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.

Feature hierarchy

- Deep CNNs, construct a so-called **feature hierarchy** by combining the low-level features in a layer-wise fashion to form high-level features.
- CNN computes **feature maps** from an input image, where each element comes from a **local patch of pixels (called local receptive field)** in the input image.



CNNs do well in image-related tasks

- **Two major ideas:**
  - **Sparse connectivity:** A single element in the feature map is connected to only a small patch of pixels.
  - **Parameter sharing**: The same weights are used for different patches of the input image
- **Replacing** a conventional, fully connected MLP with a convolution layer.
  - Decrease the number of weights (parameters) in the network.
  - Improvement in the ability to capture ***salient*** **features**

Convolutional Neural Networks (CNNs)

- CNNs are composed of several **convolutional** (conv) layers, **subsampling** layers, and one or more **Fully Connected (FC)** layers.
    - Subsampling layers are also known as **Pooling** (P) layers.
- Pooling layers **do not have any learnable parameters**.
- Convolutional and fully connected layers have **weights and biases** that are optimized during training

Convolution operation

- Convolution is a simplified name for ***discrete convolution*** operation. It is a fundamental operation in a CNN.
- Math notations:
    - $\mathbf{A}_{n_1 \times n_2}$: a two-dimensional array of size $n_1 \times n_2$
    - Brackets [] are used to denote the indexing for vector elements or matrix elements. E.g., **A**[*i*, *j*]: the element at index *i*, *j* of matrix **A**.
    - Special symbol $*$: convolution operation between two vectors or matrices.
- Operations on 1D tensors and 2D tensors.

Discrete convolution on one-dimensional data

- Let **x** be a vector having *n* elements. **x** is the input (also called *signal*)
- Let **w** be a vector having *m* elements. **w** is called the **filter** or **kernel**.
- **y** = **x** $*$ **w**: convolution for two one-dimensional vectors **x** and **w.** Mathematically defined as

$$\mathbf{y} = \mathbf{x} * \mathbf{w}\ ;\ \mathbf{y}[i] = \sum_{k=-\infty}^{+\infty} \mathbf{x}[i-k]\mathbf{w}[k]$$

- Two odd things in the definition:
  - (1) $-\infty$ to $+\infty$ indices
  - and (2) negative indexing for **x**
- To solve these two issues, **zero padding** and **flip** *w* are utilized.

Padding (Zero-padding)

- **Theoretically**: assume that **x** and **w** are filled with infinite zeros from both the left and right sides. The output **y** also has infinite size with lots of zeros. This is not very useful in real applications.
- **Practically**, **x** is padded only with a finite number of zeros.
- Parameter *p*: number of zeros padded on each side of **x**.

Original $\mathbf{x}$:			1	2	3	4	5	6	7	8		
Padding with $\mathbf{p} = \mathbf{2}$:	0	0	1	2	3	4	5	6	7	8	0	0

- After padding: $\mathbf{x}^p$ has $n + 2p$ elements.

Convolution definition with padding

- Practical definition

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \,;$$

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

$$= \mathbf{x}^p[i + m - 1]\mathbf{w}[0] + \mathbf{x}^p[i + m - 2]\mathbf{w}[1] + \ \dots + \mathbf{x}^p[i]\mathbf{w}[m - 1]$$

  - **x** and **w** are indexed in different directions.
  - Flip **w** and get a rotated filter $\mathbf{w}^r$.
  - $\mathbf{y}[i] = \mathbf{x}[i{:}\, i + m - 1] \cdot \mathbf{w}^r$ where $\mathbf{x}[i + 1 : i + m]$ is a patch of **x** with size *m*.

Example: padding size p=0

$\mathbf{x}$

1	2	3	4	5	6	7	8

$*$

$\mathbf{w}$

1	2	3	4

$\mathbf{w}^r$

4	3	2	1

$\mathbf{y}[1] =$

1	2	3	4	5	6	7	8
4	3	2	1				

$1 \times 4 + 2 \times 3 + 3 \times 2 + 4 \times 1 = 4 + 6 + 6 + 4 = 20$

$\mathbf{y}[2] =$

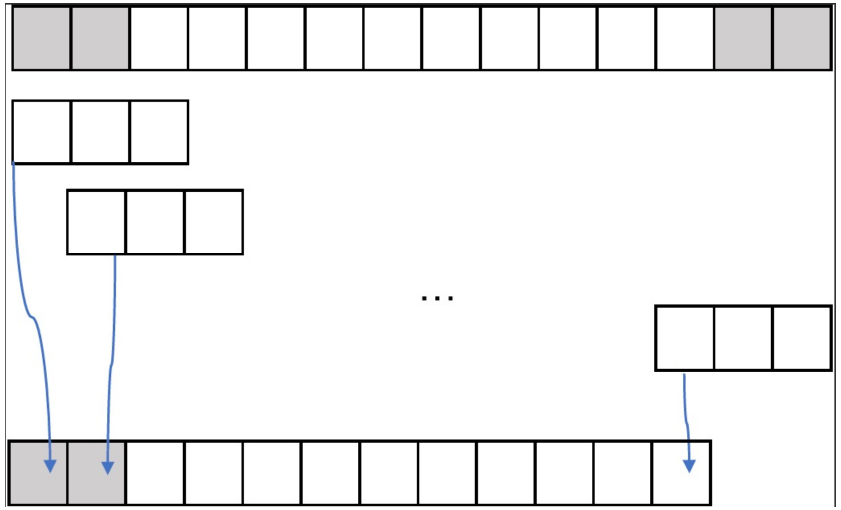
1	2	3	4	5	6	7	8
	4	3	2	1			

$2 \times 4 + 3 \times 3 + 4 \times 2 + 5 \times 1 = 8 + 9 + 8 + 5 = 30$

$$\begin{aligned}\mathbf{y}[4] &= 40\\ \mathbf{y}[5] &= 50\\ \mathbf{y}[6] &= 60\end{aligned}$$

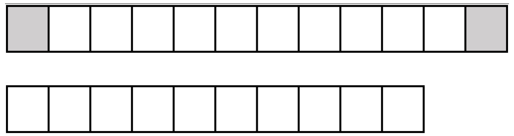
Different types of padding

- **Full** mode: $p = m - 1$

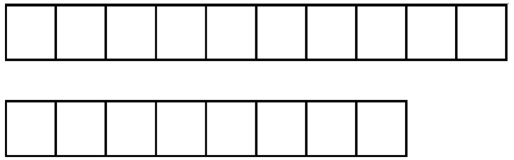


Different types of padding

- **same** mode: the size of the output is the same as the size of the input. *p* is computed based on the filter size.



- **valid** mode: *p* = 0.



- In practice, the *same* mode is the most commonly used.
- *Valid* mode decrease the tensor size, which may make the performance worse.

Stride and size of the output

- **Stride**: the number of cells that $\mathbf{w}^r$ is shifted each time. In the previous example, stride *s* is 1. If *s* = 2, in the previous example, we can only get three elements in **y**: 10, 40, 60.
- **The size of the convolutional output**:
  Given input vector of size *n*, filter of size *m*, padding parameter *p*, and stride parameter *s*, the size of the output resulting from $\mathbf{x} * \mathbf{w}$ is

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

  - Example1: $n = 10, m = 5, p = 2, s = 1, \text{then } o = \left\lfloor \frac{10+4-5}{1} \right\rfloor + 1 = 10$
  - Example2: $n = 10, m = 3, p = 2, s = 2, \text{then } o = \left\lfloor \frac{10+4-3}{2} \right\rfloor + 1 = 6$

Convolution in 2D

- Let **X** be a matrix with $n_1 \times n_2$. It is input.
- Let **W** be a matrix with $m_1 \times m_2$. It is input. **W** is called the filter or **kernel**.
- **Y** = **X** $*$ **W**: convolution for two two-dimensional matrices **X** and **W**
- Mathematically defined as

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

Padding & strides

- Zero-padding, rotating the filter matrix, and the use of strides are all applicable to 2D convolutions.
- **Zero-padding** for two dimensions, e.g., p = (1, 1)

$$\mathbf{X}^{padded} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 2 & 0 \\ 0 & 5 & 0 & 1 & 0 \\ 0 & 1 & 7 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

- **Strides** for two dimensions, e.g, s = (2, 2)

Rotating the filter matrix

- **Rotating the filter matrix** is different from matrix transpose. For example,

$$\mathbf{W} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \mathrm{W}^{\mathrm{T}} = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}, \mathrm{W}^{\mathrm{r}} = \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

- Example: Given the above $\mathbf{X}^{padded}$ and **W**, and s=(2,2), we get the following $Y = \begin{pmatrix} 24 & 18 \\ ? & ? \end{pmatrix}$.
- Let $\odot$ calculate the sum of the element-wise product.

Example

$$\mathbf{X}^{padded} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 2 & 0 \\ 0 & 5 & 0 & 1 & 0 \\ 0 & 1 & 7 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad \mathbf{W} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

$$y[0][0] = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 5 & 0 \end{pmatrix} \odot \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = (0\times 9 + 0\times 8 + 0\times 7) + (0\times 6 + 2\times 5 + 1\times 4) +$$

$$(0\times 3 + 5\times 2 + 0\times 4) = 24$$

$$y[0][1] = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 1 & 0 \end{pmatrix} \odot \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = 18$$

Subsampling

- Subsampling is typically applied in two forms of pooling operations: **max-pooling** and **mean-pooling** (or **average-pooling**)
- The pooling layer is denoted by $\mathbf{P}_{n_1 \times n_2}$
- **Pooling size:** the number of adjacent pixels in each dimension.
- ***max-pooling*** takes the maximum value from a neighborhood of pixels, and ***mean-pooling*** computes their average.

Subsampling advantages

- Max-pooling: small changes in a local neighborhood do not change the result of max-pooling, Thus, it helps generate features that are more robust to noise.
  - Example, the following two input matrices result in the same output by using max-pooling for $\mathbf{P}_{2\times 2}$
  - Given $X_1 = \begin{pmatrix} 10 & 255 & 125 & 0 \\ 70 & 255 & 105 & 25 \\ 255 & 0 & 150 & 0 \\ 0 & 255 & 10 & 10 \end{pmatrix}, X_2 = \begin{pmatrix} 100 & 95 & 100 & 100 \\ 100 & 255 & 50 & 125 \\ 255 & 80 & 30 & 100 \\ 40 & 30 & 150 & 20 \end{pmatrix}$
  - The result of max-pooling is the same: $\begin{pmatrix} 255 & 125 \\ 255 & 150 \end{pmatrix}$
- Pooling decreases the size of features, which results in higher computational efficiency. In addition, reducing the number of features may reduce the degree of overfitting.

CNN

- The most important operation in a traditional neural network is the matrix-vector multiplication.
- For image dataset, we need to work with **multiple input or color channels.**
  - We can use a rank-3 tensor or a three-dimensional array $X_{n_1 \times n_2 \times C_{in}}$ where C_{in} is the number of **input channels**. For color images, C_{in} is 3 representing the R, G, B color channels. For grayscale images, we have C_{in} =1.
  - Number of output feature maps: C_{out}.
  - We perform the convolution operation **for each channel** separately and then **add the results** together using the matrix summation
    - The convolution associated with each channel (c) has its own kernel matrix as W[:, :, c]

CNN - Read images

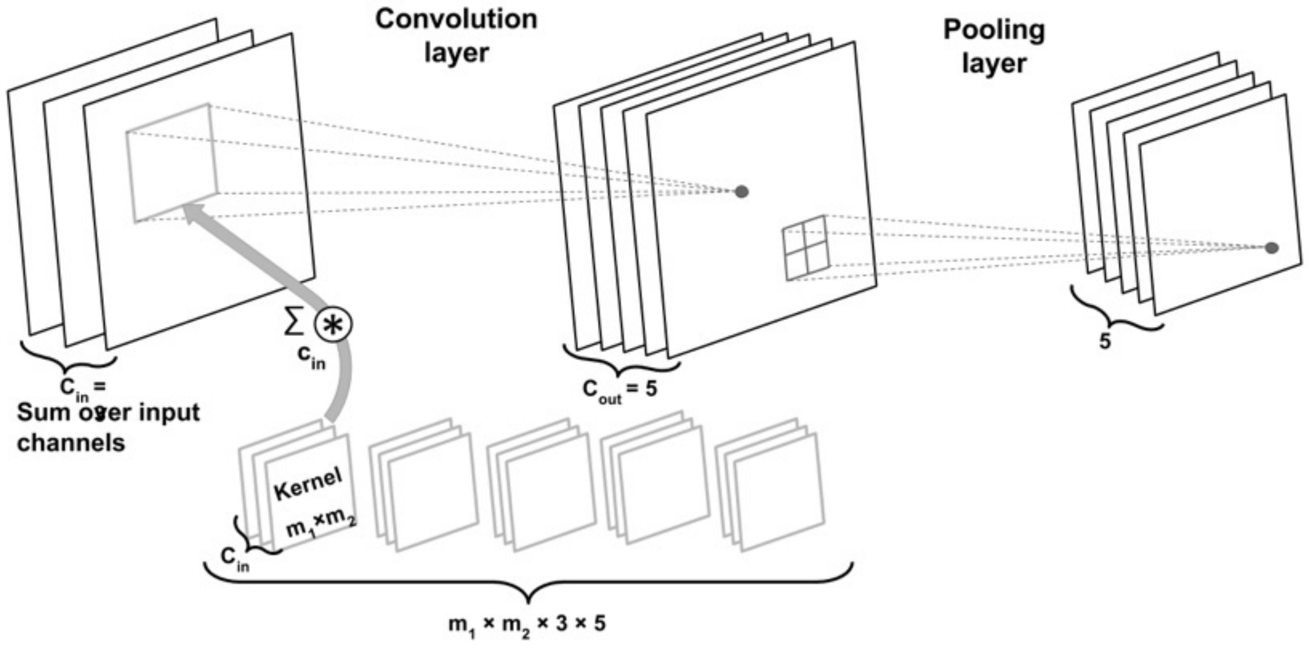
```
import torch
from torchvision.io import read_image
img = read_image('cat_dog_images/cat-01.jpg')
print('Image shape:', img.shape)
print('Number of channels:', img.shape[0]) #Number of channels: 3
print('Image data type:', img.dtype) #Image data type: torch.uint8
print(img[:, 100:102, 100:102])
```

Output:

```
Image shape: torch.Size([3, 900, 1200])
Number of channels: 3
Image data type: torch.uint8
tensor([[[17, 18],
         [16, 16]],
        [[17, 18],
         [16, 16]],
        [[17, 18],
         [16, 16]]], dtype=torch.uint8)
```

Kernel matrix

- **Kernel matrix**: each one has size $m_1 \times m_2$, it has C_{in} such kernels where each one represents one $m_1 \times m_2$ kernel matrix. If we want to get C_{out} feature maps, kernel matrix is of size $m_1 \times m_2 \times C_{in} \times C_{out}$.



How many trainable parameters

- For one convolutional layer
  - Kernel: $m_1 \times m_2 \times C_{in} \times C_{out}$
  - Bias: C_{out}
- If the input tensor size is $(n_1 \times n_2 \times C_{in})$ and we need to create a fully connected layer
  - $(n_1 \times n_2 \times C_{in}) \times (n_1 \times n_2 \times C_{out}) = (n_1 \times n_2)^2 \times C_{in} \times C_{out}$

NN Tuning and NN capacity

- The **size of a weight matrix** need to be tuned.
- The **number of layers** needs to be tuned.
- **Capacity** of a network refers to the level of complexity of the function that it can learn. When the capacity is too small, we may have *underfitting* issue. When the capacity is large, we may have *overfitting* issue.

Regularizing a NN with dropout

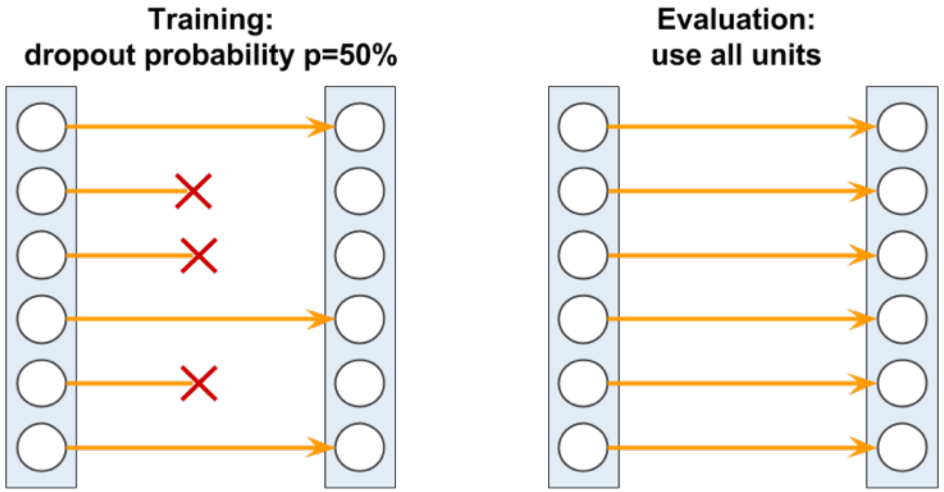
- One way is to **build a network with a relatively large capacity** (in practice, we want to choose a capacity that is slightly larger than necessary).
- Apply **one or multiple regularization schemes** to achieve good generalization performance on new data.
- Regularization
  - L1 and L2 regularization can prevent or reduce the effect of overfitting by adding a penalty to the loss that results in shrinking the weight parameters during training.
- Other regularization strategy: **dropout**. It works well for deep NNs and can effectively prevent overfitting.

Dropout

- Dropout is applied to the **hidden** units of higher layers.
- During the **training** phase: a fraction of the hidden units is randomly dropped at every iteration with probability p_{drop}. This probability is determined by the user and the common choice is *p* = 0.5.
  - When dropping a certain fraction of input neurons, the weights associated with the remaining neurons are rescaled to account for the missing (dropped) neurons. TensorFlow and other tools scale the activations during training.
- The **random dropout forces** the network to learn **a redundant representation** of the data.
- The network is forced to learn **more general and robust** patterns from the data.

Dropout

- During the **testing** phase: all neurons contribute to computing the pre-activations of the next layer.
- Example:



Dropout

- Considered as the **consensus** (averaging) of an ensemble of models.
- Dropout offers a workaround, with an efficient way to **train many models at once** and compute **their average predictions** at test or prediction time.

References

- Chapter 14: By Sebastian Raschka , Yuxi (Hayden) Liu , Vahid Mirjalili: Machine Learning with PyTorch and Scikit-Learn, Packt.