IT Talent Training Course

Aug. 2025.

A.I. PROGRAMMING WITH PYTORCH

Instructor :
Daesung Kim

5th Day - Part 01

INDEX

COURSE OBJECTIVES

- Understand the core structure and working principles of CNN (Convolutional Neural Network), which is specialized in image data processing, in a visual and intuitive way. Understand why existing neural networks have limitations in image processing, and be able to explain how CNN maintains spatial information and learns features efficiently.

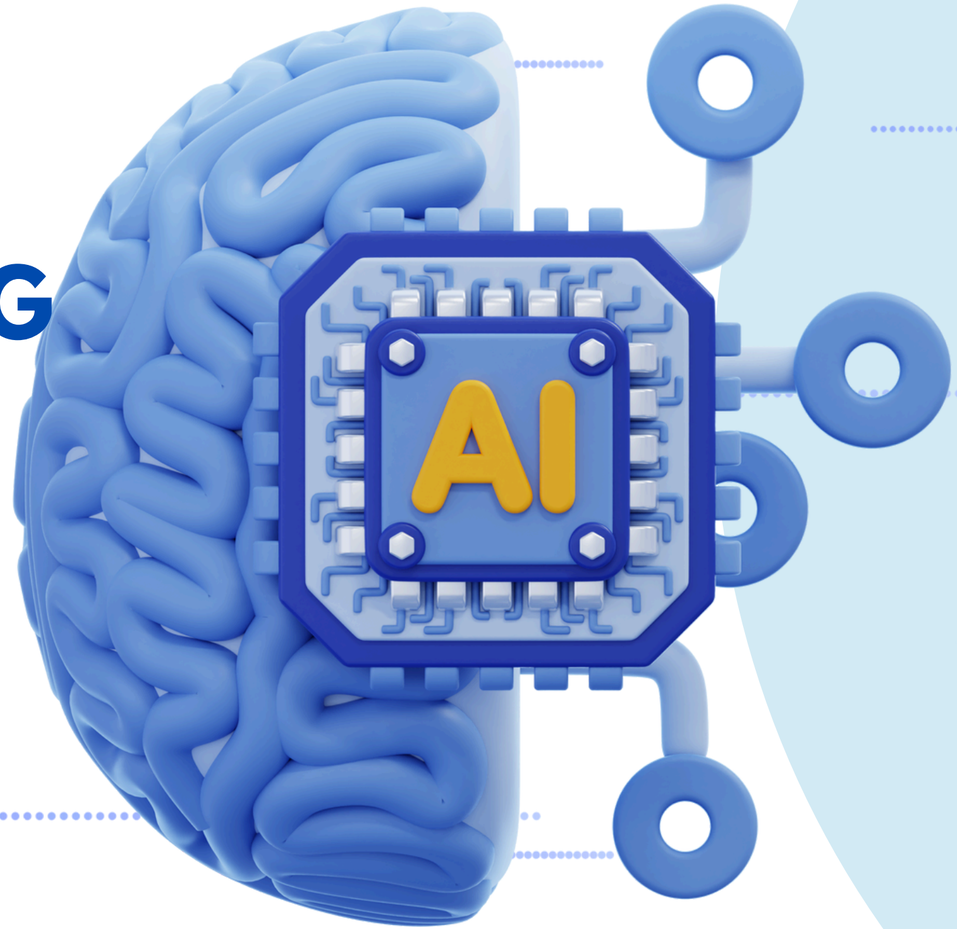
WHY DO WE NEED A NEW KIND OF NEURAL NETWORK?

The linear regression model we covered was great at handling numeric data, but there is much more complex data in the world than just numbers. It is data like images.

From photos taken with smartphones to CCTV footage from the street, our surroundings are filled with image data. Artificial intelligence needs to be able to 'see' and understand these images.

This morning, we will delve deep into the convolutional neural network (CNN), which is used to process this image data and has revolutionized the field of computer vision.

01
LIMITATIONS OF EXISTING NEURAL NETWORKS

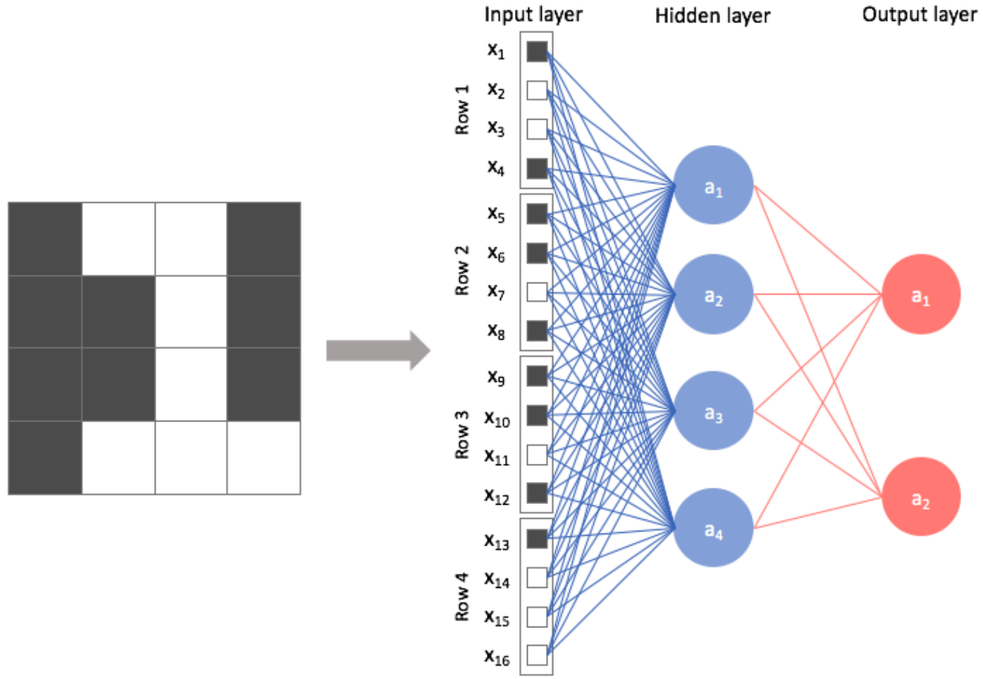


LIMITATIONS - 1

- **Fully Connected Layer (nn.Linear):** The most basic neural network structure, where all neurons in one layer are connected to all neurons in the next layer.
- **Image data structure:** An image is made up of a grid of pixels, and each pixel contains color information (e.g., black and white has 1 channel, color has 3 channels - Red, Green, Blue).
- **Flattening:** The process of flattening a multidimensional array (e.g., a 2D image) into a 1D vector.

1. Loss of Spatial Information

- To input an image into a fully connected neural network, the 2D or 3D image data must be converted into a long 1D vector. This is called **'flattening'**.
- For example, let's imagine that we have a black and white handwritten digit image of size 4*4 pixels. To input this image into a FCN, we need to convert it into a single long vector with 4*4 = 16 digits.



LIMITATIONS - 2

- There is a fatal loss of information in this process. Just as important as the value of a pixel in an image is the spatial information of which pixels are next to each other. A human eye, a cat's ear, or a car wheel only have meaning when several pixels are grouped together in a specific shape. But the moment you flatten the data into one dimension, the information about which pixels were originally above, below, or next to each other disappears. The neural network simply receives a list of 16 independent numbers.

LIMITATIONS - 3

2. Explosion of Parameters

The second problem is much more realistic and serious. The number of weights (parameters) that the model has to learn increases exponentially.

- Let's say we're dealing with a small black and white image that's 28 pixels wide and 28 pixels high.
  - Input vector size: 28×28=784
  - If the number of neurons in the first hidden layer is 512, how many parameters are needed between the input layer and the first hidden layer?
  - 784×512+512(bias)=401,920
- Now let's consider a realistic-sized color (3-channel) image that's 224 pixels wide and 224 pixels high.
  - Input vector size: 224×224×3=150,528
  - If the number of neurons in the first hidden layer is 512?
  - 150,528×512+512(bias)=77,070,848. About 77 million parameters are required in just one layer!
- This large number of parameters **leads to the following problems:**
  - **A huge amount of computation:** It takes too much time and computing resources to train the model.
  - **Overfitting:** The model is too complex, so it is easy for it to be overly optimized for training data and perform poorly on new data.

To solve these two problems, the **convolutional neural network (CNN)** was born from the question, "Is there a way to drastically reduce the number of parameters while maintaining the spatial features of the image?"

LIMITATIONS - 3

```
import torch
import torch.nn as nn

# Function to count the number of parameters in the model
def count_parameters(model):
return sum(p.numel() for p in model.parameters() if p.requires_grad)

# --- Scenario 1: 28x28 grayscale image ---
# Input dimension: 28 * 28 = 784
# Number of hidden neurons: 512
fcn1 = nn.Linear(in_features=784, out_features=512)
print(f"FCN parameters for 28x28 grayscale image: {count_parameters(fcn1):,}")

# --- Scenario 2: 224x224 color image ---
# Input dimension: 224 * 224 * 3 = 150,528
# Number of hidden neurons: 512
fcn2 = nn.Linear(in_features=150528, out_features=512)
print(f"Number of FCN parameters for 224x224 color image: {count_parameters(fcn2):,}")
```

```
FCN parameters for 28x28 grayscale image: 401,920
FCN parameters for 224x224 grayscale image: 77,070,848
```

As you can see, the results of the code execution are exactly the same as what was calculated in theory. Considering that only one layer requires tens of millions of parameters, it is clear how inefficient it is to apply FCN directly to images.

SUMMARY

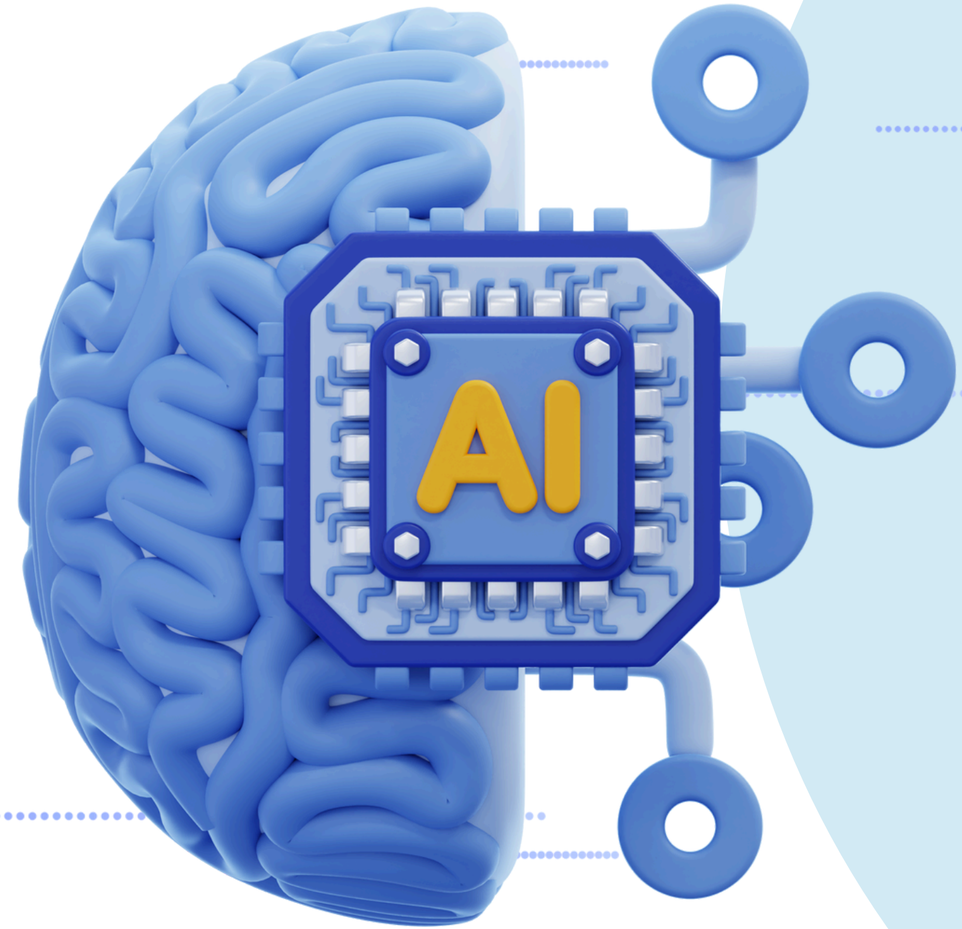
Summary

- Applying images to a conventional fully connected neural network (FCN) requires 'flattening' the data into a one-dimensional vector.
- In this process, spatial information between pixels (top, bottom, side, etc.) is lost.
- In addition, even if the image size increases slightly, the number of parameters to be learned increases exponentially, making learning inefficient and increasing the risk of overfitting.

Glossary of terms:

- Fully Connected Neural Network (FCN): A neural network in which all neurons in one layer are connected to all neurons in the next layer. nn.Linear is a representative example.
- Spatial Information: A two-dimensional or three-dimensional positional relationship between data elements. In images, it refers to the arrangement of pixels.
- Parameter: Values that the model optimizes itself during the learning process. It mainly refers to weights and biases.

02
CONVOLUTION OPERATION



THE CORE OF CNN: CONVOLUTION OPERATION - 1

- **Matrix Operations:** Basic understanding of matrix addition and multiplication. (NumPy, PyTorch Tensor Practice)
- **Feature:** A value or attribute that represents data well. Machine learning models make predictions based on this feature.

CNN solves the problem of FCN by looking at it in a 'local' and 'shared' way. The core operation that implements this idea is Convolution.

1. Kernel / Filter

The main character of the convolution operation is a small matrix called a kernel or filter. You can think of these kernels as 'feature detectors' designed to detect certain 'features' in the image.

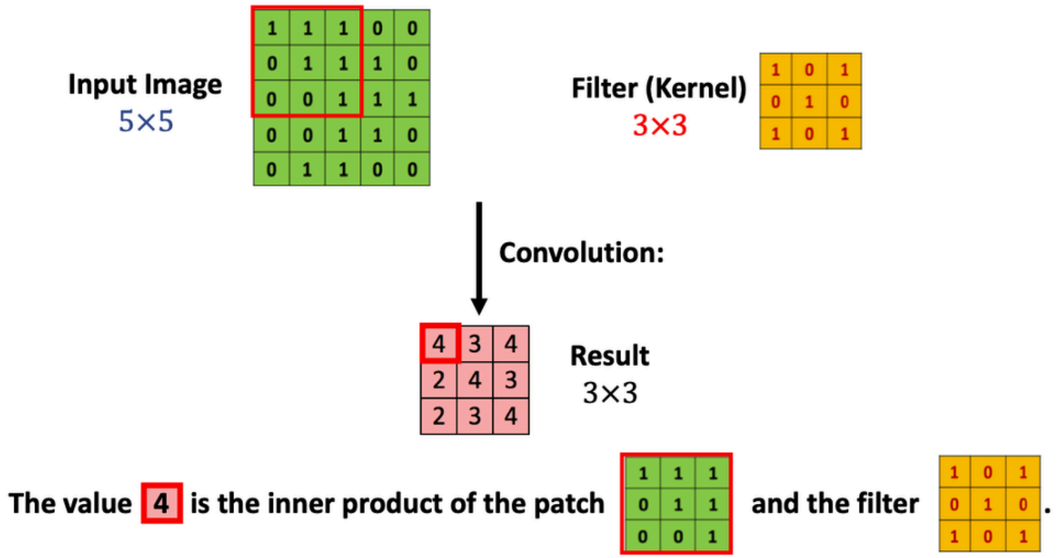
For example, one kernel might detect vertical lines, another might detect horizontal lines, and another might detect green blobs.

THE CORE OF CNN: CONVOLUTION OPERATION - 2

2. Convolution operation process

The convolution operation is a simple process that follows:

- **Sliding:** The kernel moves from the top left to the bottom right of the input image at regular intervals.
- **Element-wise Product and Sum:** The kernel multiplies each element of the kernel by a small area of the image (called the receptive field) and adds all the results.
- **Feature Map Generation:** The value calculated in step 2 becomes a pixel of the output feature map or Activation Map.



When the kernel scans the entire image, a complete feature map is generated. This feature map is like a map that shows 'where and how strongly the feature that the kernel is looking for appears in the input image'.

THE CORE OF CNN: CONVOLUTION OPERATION - 3

Key Idea:

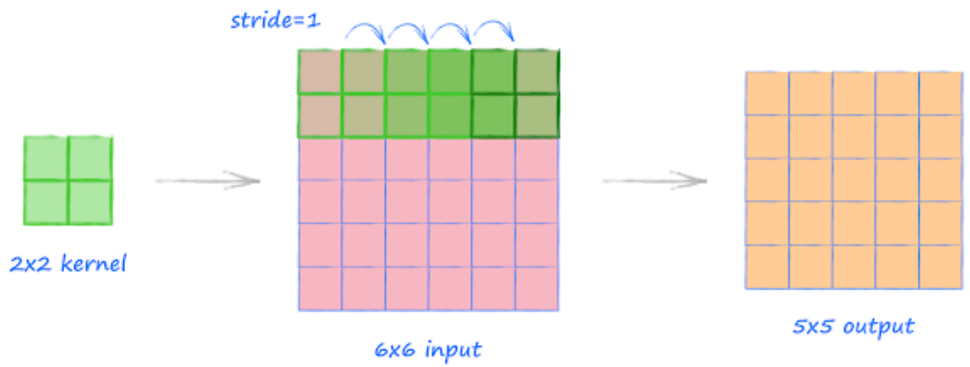
- **Locality:** Instead of looking at the entire image at once like FCN, it looks at a 'local' part of the image through a small window called a kernel. This is based on the fact that features in an image (e.g. the human eye) are not made up of individual pixels, but rather of a combination of surrounding pixels.
- **Parameter Sharing:** A kernel that finds vertical lines in the upper left of an image can also find vertical lines in the lower right. Thus, a single kernel (a single set of parameters) is reused over the entire image. This 'parameter sharing' dramatically reduces the number of parameters to learn compared to FCN.

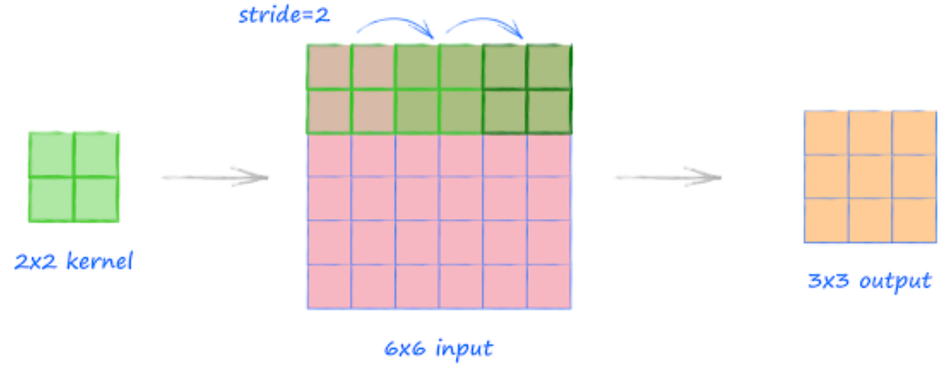
THE CORE OF CNN: CONVOLUTION OPERATION - 4

3. Main parameters: Stride, Padding, Channel

There are three important concepts that allow for more precise control over the convolution operation.

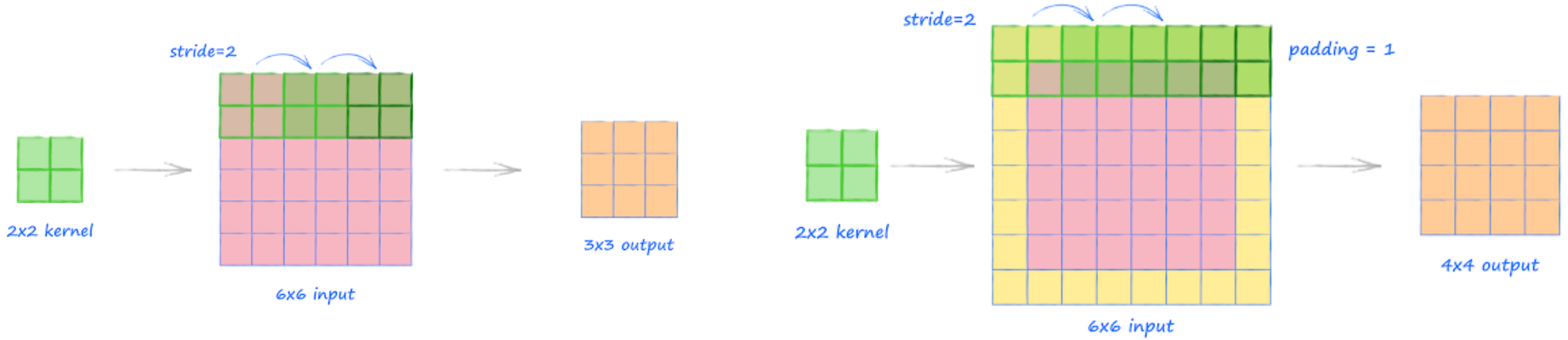
- **Stride:** The step size that determines how many pixels the kernel moves at a time. A stride of 1 means that it moves one pixel at a time, while a stride of 2 means that it moves every two pixels. As the stride increases, the output feature map size decreases and the amount of computation decreases.





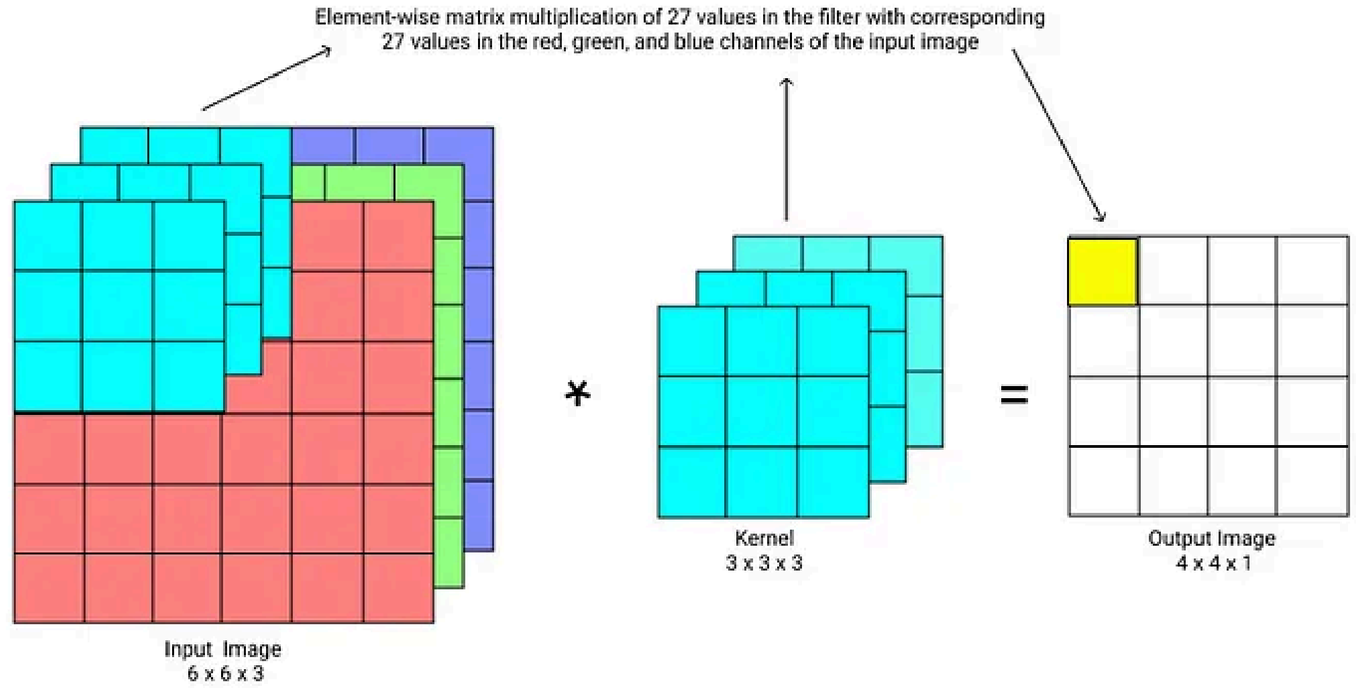
THE CORE OF CNN: CONVOLUTION OPERATION - 5

- **Padding:** When performing convolution operations, there is a problem that the size of the feature map becomes smaller than the input image. In particular, pixels at the corners or borders of the image participate in the operation less often than the center, which may result in information loss. To prevent this, the edges of the input image are surrounded by a specific value (usually 0) and this is called padding. Adding padding can keep the size of the feature map the same as the input (this is called 'Same' padding).



THE CORE OF CNN: CONVOLUTION OPERATION - 6

- **Channel:** Black and white images have one channel, but color images have three channels: R, G, and B. If the input image has multiple channels, the kernel must also have the corresponding number of channels (depth). For example, a kernel that processes a three-channel color image would be a 3D cube with three channels. The operation performs convolution on each channel and then adds the results to create a single number. In addition, multiple kernels can be used to detect different features (vertical lines, horizontal lines, red, etc.) at the same time. The number of kernels used is the number of channels in the output feature map.



Output size calculation formula:

- When the input size is I×I, the kernel size is K×K, the padding is P, and the stride is S, the size of the output feature map is calculated as follows:

$$O = \frac{I - K + 2P}{S} + 1$$

EXAMPLE

Let's use PyTorch's nn.Conv2d to see for ourselves how the size of the feature map changes after a convolution operation.

[Code] Day05 - 01_Convolution_operation.py

SUMMARY

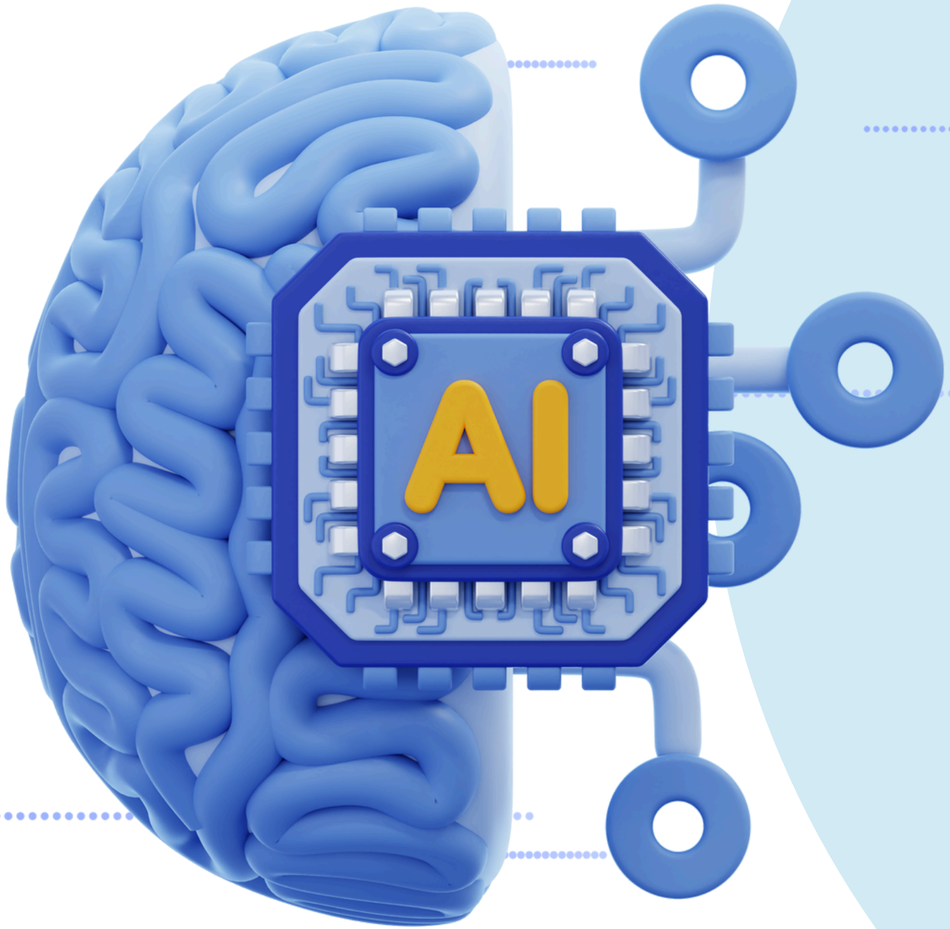
Summary

- Convolutional operation is the process of generating a feature map by sliding a small feature detector called a 'kernel' over an image.
- It only looks at 'local' information of the image and 'shares' a single kernel, drastically reducing the number of parameters.
- Stride is the movement step of the kernel, Padding is a border to prevent information loss, and Channel is the depth of the image and kernel.

Glossary of terms:

- Convolution: A core operation that extracts features from input data using kernels.
- Kernel/Filter: A small weight matrix for detecting specific features.
- Feature Map: The result of the convolution operation. A map showing where specific features appear in the image.
- Receptive Field: The area of the input image that the kernel sees at one time.
- Parameter Sharing: A core principle of CNN that reuses a single kernel for all locations in the image.

03
POOLING LAYER



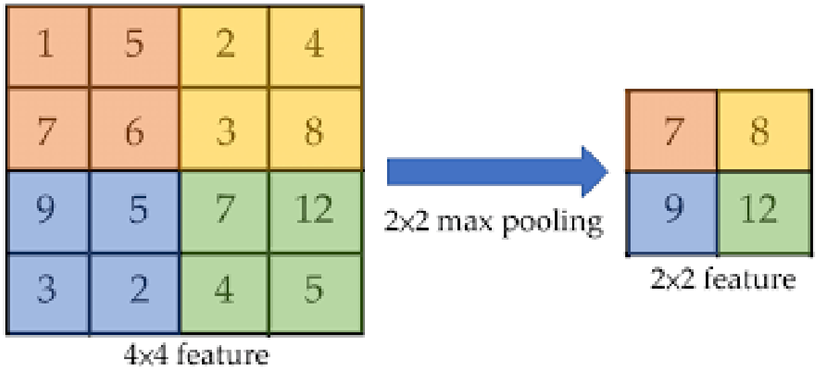
REDUCE SIZE AND KEEP ESSENCE: POOLING LAYER

The feature maps that pass through the convolution layer can still be large. Also, we need to be able to recognize the same cat even if the features we find change slightly in the image (e.g. the cat's eyes move slightly to the left of the center of the picture). The role of the pooling layer is to solve these two problems.

Pooling is a downsampling process that reduces the width and height of the feature map. Like the convolution, a kernel (usually 2×2) slides over the feature map, but without weights, it selects only one value according to a certain rule.

MAX POOLING

This is the most widely used pooling method. It keeps only the largest value within the area covered by the kernel and discards the rest.



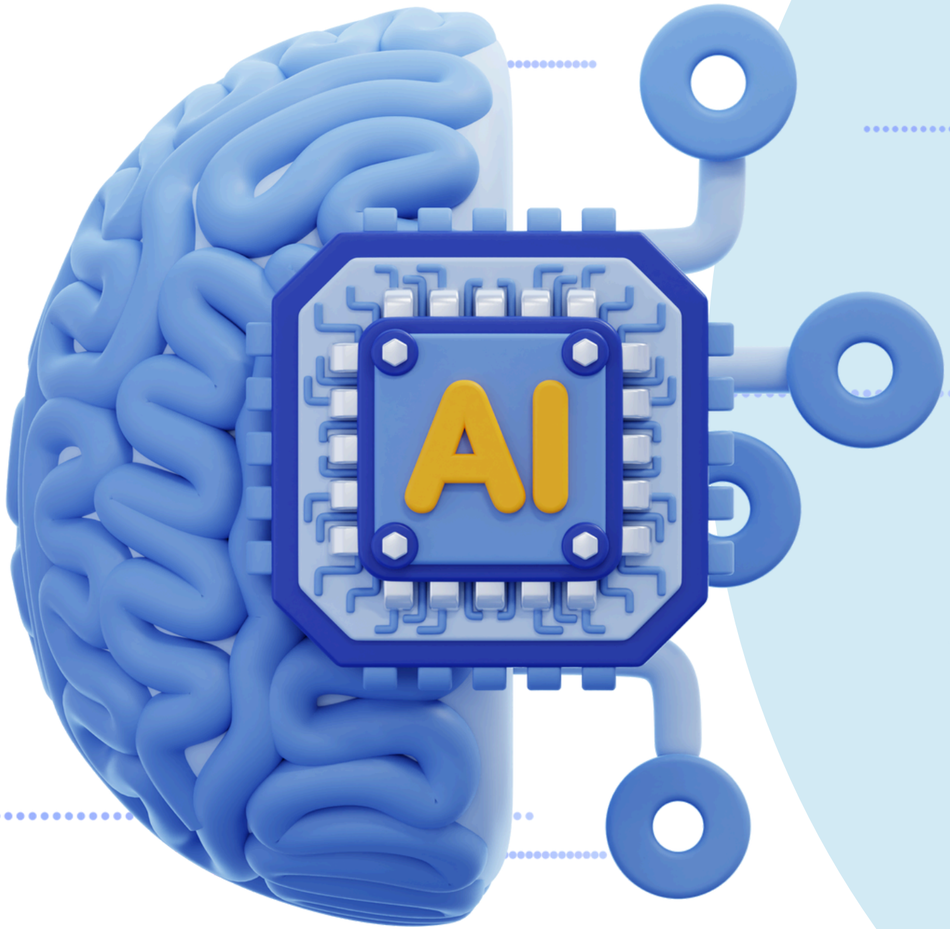
Max pooling has two effects:

- **Increased computational efficiency:** By reducing the size of the feature map (usually by half), the number of parameters and computations to be processed in the next layer is reduced.
- **Securing translation invariance:** Since only the strongest signal (largest value) in a specific region is retained, the pooling result is likely to remain the same even if the location of the feature changes by one or two pixels. This makes the model more robust.

In addition, there are average pooling methods that take the average value of the region, but max pooling is generally preferred because it preserves the features the best.

04

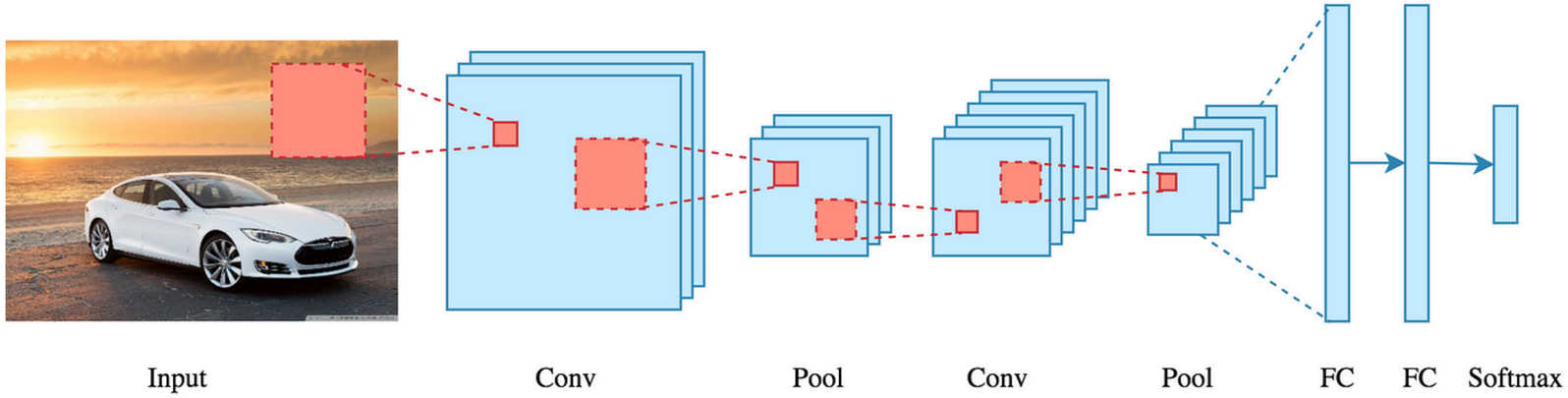
HIERARCHICAL FEATURE LEARNING OF CNNAYER



THE BIG PICTURE: HIERARCHICAL FEATURES OF CNNS

A typical CNN is largely divided into two parts.

- **Feature Extractor:** It is made by stacking multiple blocks of [Convolution Layer -> Activation Function (ReLU) -> Pooling Layer].
- **Classifier:** This is the fully connected layer (FCN) part that receives the extracted features and ultimately predicts the class of the image (e.g. 'cat', 'dog').



THE BIG PICTURE: HIERARCHICAL FEATURES OF CNNS

This structure learns the feature hierarchy on its own through learning.

- **Shallow Layers:** The convolutional layers at the beginning of the network mainly learn simple, low-level features. For example, kernels are learned to detect vertical lines, horizontal lines, diagonals, chunks of specific colors, and boundaries between light and dark.
- **Intermediate Layers:** The simple feature maps extracted from the early layers are input and combined to learn more complex, intermediate-level features. For example, lines and curves are combined to detect 'eye shapes', 'circular shapes of car wheels', and 'window patterns of buildings'.
- **Deep Layers:** As the network goes further back, the intermediate features are combined again to learn very complex, high-level, and abstract features. The 'eye', 'nose', and 'mouth' features are combined to form a feature called 'human face', and the 'wheel', 'door', and 'window' features are combined to form a feature representing the entire object called 'car'.

Starting from such simple patterns and combining increasingly complex and meaningful patterns is like building small parts out of Lego blocks and using those parts to build a large structure. This hierarchical feature learning ability is the fundamental reason why CNNs are so powerful in image recognition.

Finally, this high-level feature map is **flattened** and passed as input to a classifier (FCN), which calculates which class the image most likely belongs to based on this feature vector and makes a final prediction.

IT Talent Training Course

Aug. 2025.

A.I. PROGRAMMING WITH PYTORCH

Instructor :
Daesung Kim

5th Day - Part 02

INDEX

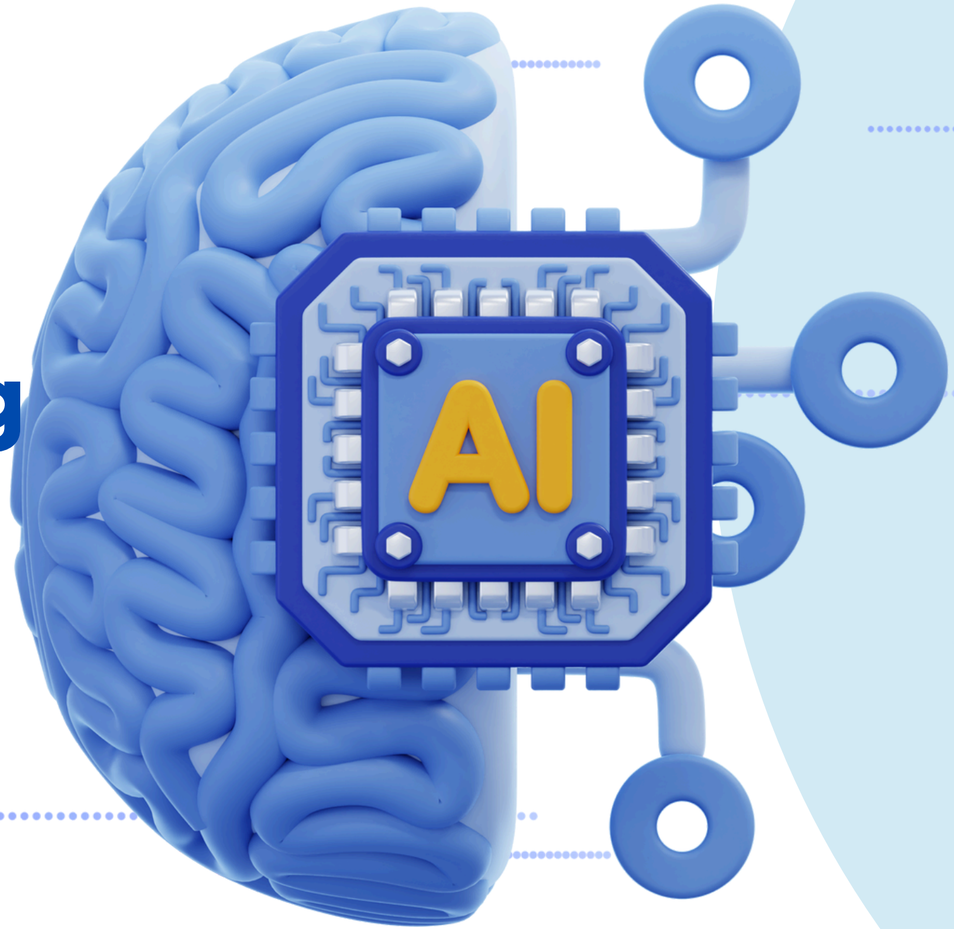
COURSE OBJECTIVES

This course provides an experience of implementing the theoretical concepts of CNN learned in the morning session into actual code to complete a deep learning model that classifies image data from start to finish. This is a comprehensive hands-on training that mobilizes all the knowledge of PyTorch learned from Day 1 to Day 4 (Tensor, nn.Module, DataLoader, Optimizer, learning loop).

- **Key contents:** Preparing a dataset using torchvision, designing a CNN model architecture using nn.Conv2d and nn.MaxPool2d, and the entire process of training the model and evaluating its performance through a completed learning pipeline.

01

Preparing the lab environment and loading the dataset



REQUIRED BACKGROUND KNOWLEDGE

- **Google Colab:** Setting up a GPU runtime
- **PyTorch:** Basic tensor manipulation
- **torchvision:** A library for image data processing
- **DataLoader:** Feeding batch data
- **Data Normalization:** The importance of data preprocessing

In the morning theory class, we learned the concepts of how CNNs recognize images. Now it's time to put our ideas into a living program with some actual code. In this lab, we'll build an image classifier using the CIFAR-10 dataset, which contains 10 types of small color images.

1. INTRODUCTION TO TORCHVISION LIBRARY

PyTorch includes a very useful library called torchvision, which is specifically designed for computer vision research. This library provides three main functions:

- **Datasets:** It allows you to easily download and load datasets widely used in deep learning research, such as CIFAR-10, MNIST, and ImageNet, with just a few lines of code.
- **Models:** It provides pre-trained popular model architectures, such as AlexNet, VGG, and ResNet, so that you can use powerful models right away without having to implement everything yourself.
- **Transforms:** It provides various tools to transform (preprocess) image data into a form suitable for input to a deep learning model. For example, you can easily resize an image, crop a specific part, or flip the data.

2. CIFAR-10 DATASET PREPARATION AND PREPROCESSING - 1

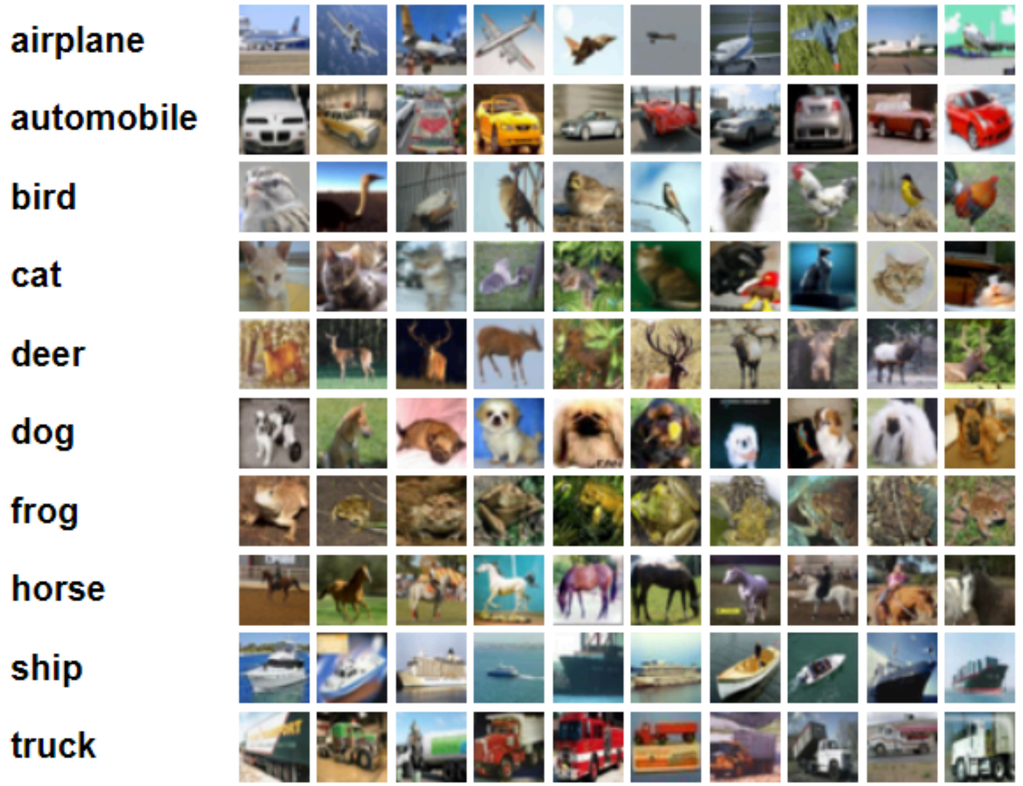
The CIFAR-10 dataset consists of 60,000 color images of 32x32 pixels in size, and is divided into 10 classes (classifications): 'airplane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', and 'truck'.

Before training a model, **preprocessing is essential** to transform the image data into a form that the model can learn better. We will apply two main preprocessing steps.

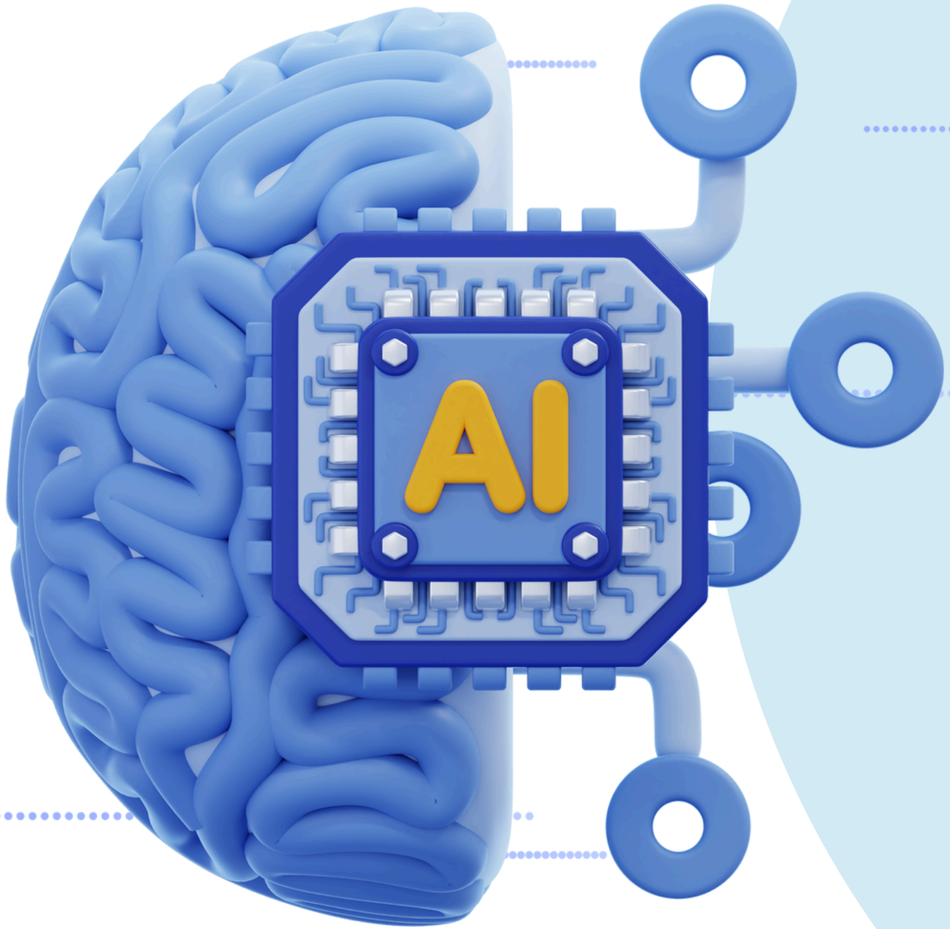
- **transforms.ToTensor():** Images loaded with the Python Image Library (PIL) have pixel values ranging from 0 to 255. ToTensor() converts them to Tensor objects that PyTorch can handle, and changes the pixel value range to real numbers ranging from 0.0 to 1.0. It also changes the dimension order of the image from (height, width, channels) to (channels, height, width). This is the input format required by the CNN layer in PyTorch.
- **transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)):** Data **Normalization** is a very important technique to stabilize model learning and improve performance. It is common to adjust the pixel values of each channel (Red, Green, Blue) to have a mean of 0 and a standard deviation of 1, but here we will simply adjust all pixel values to a range between -1 and 1. This is done through the operation (input value - 0.5) / 0.5, and it increases learning efficiency by concentrating the data distribution toward the center.

2. CIFAR-10 DATASET PREPARATION AND PREPROCESSING - 2

[Code] Day05 - 02_Visualizing_CIFAR-10.py



02
CNN model definition



DEFINING A CNN MODEL - 1

- **nn.Module:** The base class for all PyTorch models
- **__init__, forward:** Two core methods of the model class
- **nn.Conv2d:** A 2D convolutional layer
- **nn.MaxPool2d:** A 2D max pooling layer
- **nn.Linear:** A fully connected layer
- **nn.ReLU:** An activation function

Now it's time to design an actual CNN model by combining the convolution and pooling layers we learned in the morning. It's the same way we created the model by inheriting nn.Module on Day 4, but this time we use nn.Conv2d and nn.MaxPool2d instead of nn.Linear.

DEFINING A CNN MODEL - 2

The structure of the model we will create is as follows.

- Input -> Conv1 -> ReLU -> Pool1 -> Conv2 -> ReLU -> Pool2 -> Flatten -> FC1 -> ReLU -> FC2 -> ReLU -> FC3 -> Output

The most important part is to understand how the shape of the data (tensor) changes as it passes through each layer.

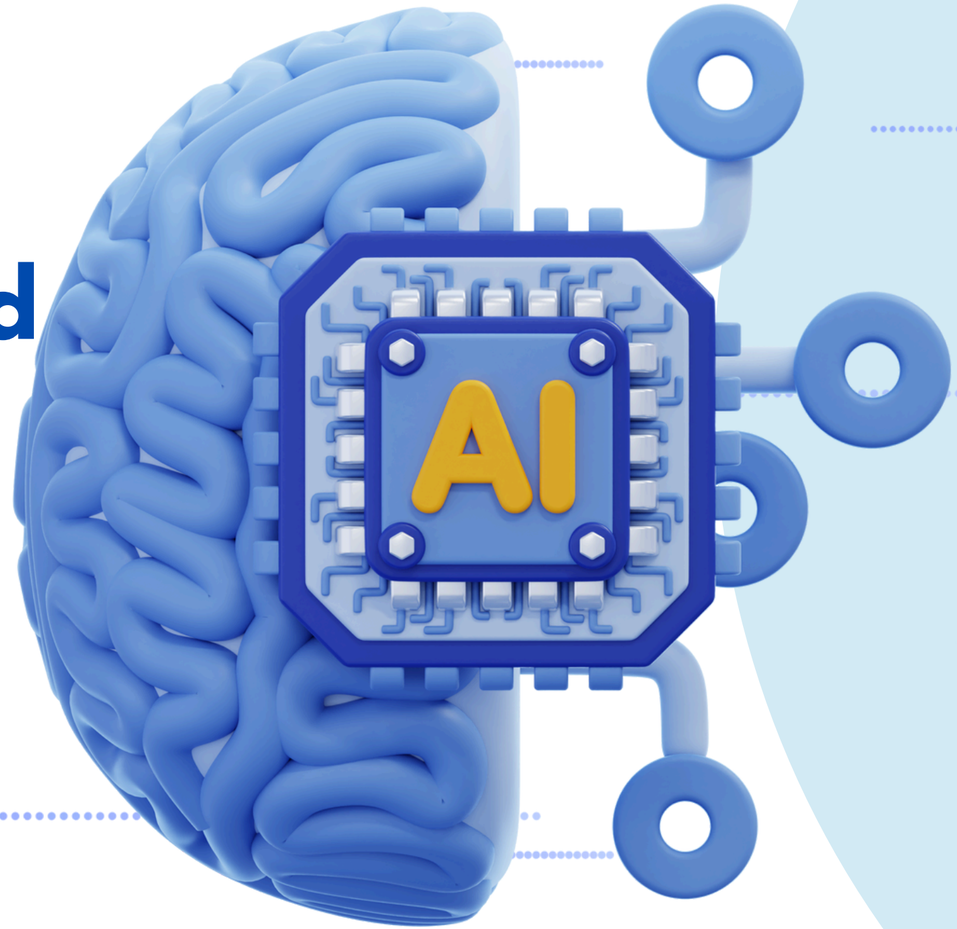
1. **Input:** CIFAR-10 image (channels, height, width) = (3, 32, 32). Since we process in batches, it becomes (batch size, 3, 32, 32).
2. **nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5):**
   - in_channels=3: The number of channels in the input data (3 since it is an RGB image)
   - out_channels=6: The number of filters (kernels) to use. This will be the number of channels in the output feature map.
   - kernel_size=5: The size of the filter is 5x5.
   - The size of the output tensor is: (32 - 5 + 1) = 28. Therefore, it becomes (batch size, 6, 28, 28).
3. **nn.MaxPool2d(kernel_size=2, stride=2):**
   - Moves a 2x2 window by 2 spaces and leaves only the largest value.
   - The size of the output tensor is halved in width and height. (28 / 2) = 14. Therefore, it becomes (batch size, 6, 14, 14).

DEFINING A CNN MODEL - 3

4. **nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5):**
    - It takes the output channels (6) of the previous layer as input and outputs 16 feature maps.
    - The size of the output tensor is: (14 - 5 + 1) = 10. Therefore, it becomes (batch size, 16, 10, 10).
5. **nn.MaxPool2d(2, 2):**
    - Apply max pooling again.
    - The size of the output tensor is: (10 / 2) = 5. So it becomes (batch size, 16, 5, 5).
6. **Flatten:** Now we need to flatten the 3D feature map (16, 5, 5) into a 1D vector to input it to the fully connected layer (nn.Linear). The size of this vector will be 16 * 5 * 5 = 400.
7. **The nn.Linear layers:** We design them to take 400 nodes and output the scores for 10 classes.

[Code] Day05 - 03_Defining_CNN_model_classes.py

03
Defining loss function and optimizer



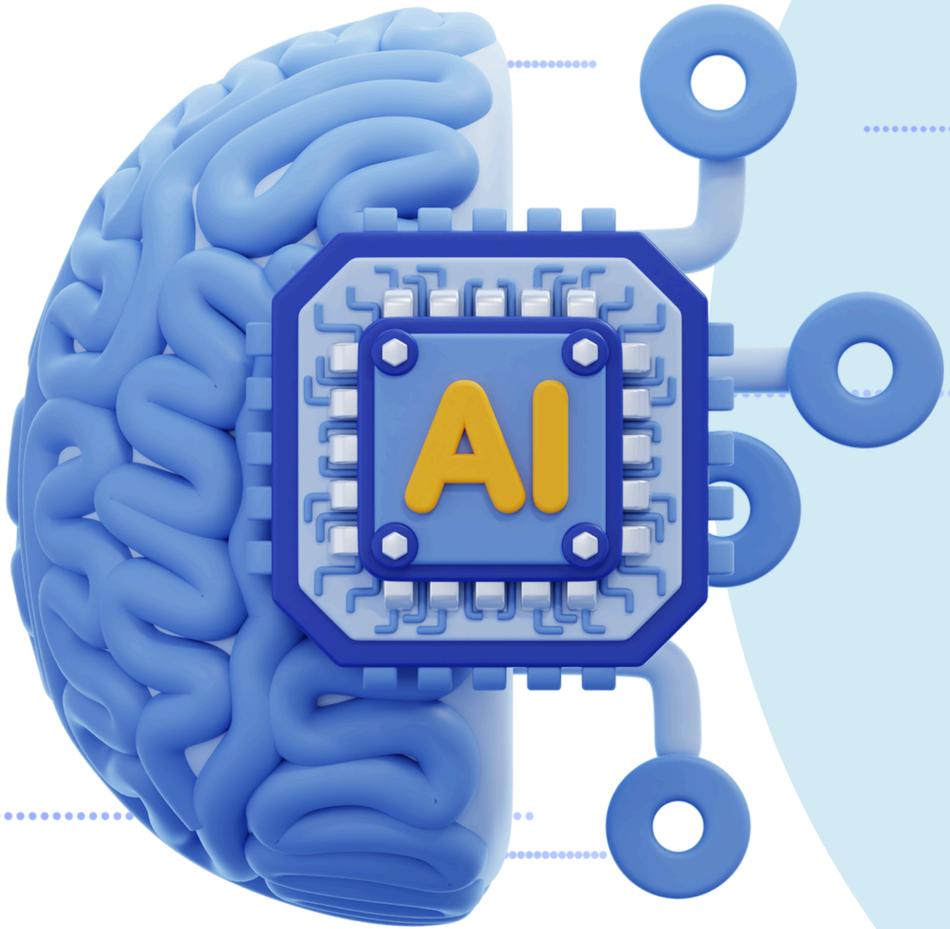
DEFINING THE MODEL TRAINING METHOD

- **Loss Function:** A function that measures how wrong the model's prediction is
- **Cross-Entropy:** A standard loss function for multi-class classification problems
- **Optimizer:** An algorithm that updates the model's parameters based on the calculated loss
- **Stochastic Gradient Descent (SGD):** A typical optimizer
- **Learning Rate:** A value that determines how large the parameter updates will be

Now that we've defined the structure of the model, we need to define a way to 'learn' the model. This is exactly what we learned on Day 4.

- **Loss Function:** We're solving a 'multi-class classification' problem where we have to match one of 10 classes. The most standard loss function used for this problem is nn.CrossEntropyLoss. This function calculates the loss value by comparing the 10 scores (logit) output by the model with the actual correct label.
- **Optimizer:** It updates the model's learnable parameters (mainly the weights and biases of nn.Conv2d and nn.Linear) in a direction that reduces the loss (error) calculated by the loss function. We will use **momentum** added to the most basic optim.SGD (stochastic gradient descent). Momentum is a technique that helps learning proceed faster and more stably by providing 'inertia' to the gradient descent process.

04
Neural network training and evaluation



MODEL TRAINING ACCORDING TO LEARNING LOOP STRUCTURE - 1

- **Training Loop:** The process of training the model by repeating the entire dataset
- **Epoch:** A unit that uses the entire training dataset at once
- **Batch:** A unit that divides the data into small groups and processes them
- **optimizer.zero_grad(), loss.backward(), optimizer.step():** The three stages of the training loop
- **Model Evaluation:** The process of verifying the model's performance with untrained data

Finally, everything is ready. Now, we will train the model according to the standard training loop structure that we completed on Day 4. We will train the model by repeatedly looking at the entire training dataset multiple times, that is, we will train for multiple epochs.

MODEL TRAINING ACCORDING TO LEARNING LOOP STRUCTURE - 2

Let's recap the 5 core steps of the training loop.

1. Get a batch of input data and labels from the trainloader (and send it to the GPU).
2. optimizer.zero_grad(): We don't want gradients from previous batches to remain, so we initialize the gradients to zero.
3. outputs = net(inputs): We pass the data through the model to get the outputs. (Forward propagation)
4. loss = criterion(outputs, labels): We compare the predicted values to the actual results to compute the loss.
5. loss.backward(): We compute the gradient for each parameter with respect to the loss. cite_start
6. optimizer.step(): Based on the computed gradients, the optimizer updates the parameters of the model.

[Code] Day05 - 04_CNN_model_training.py

EVALUATE THE MODEL WITH TEST DATA

The model training is complete! But to know how well this model works, we need to evaluate its performance on new data that has never been seen during training, the test dataset.

Since we don't need to compute gradients during evaluation, we run the code in the torch.no_grad() context to avoid unnecessary computation and reduce memory usage.

[Code] Day05 - 05_CNN_model_test.py

SUMMARY

Summary

- We used the torchvision library to load the CIFAR-10 image dataset and preprocessed the data using ToTensor and Normalize transformations.
- We designed a simple CNN model architecture directly as an nn.Module class by combining nn.Conv2d and nn.MaxPool2d layers.
- We set nn.CrossEntropyLoss and optim.SGD with momentum, which are suitable for multi-class classification problems, as the loss function and optimizer.
- We trained the CNN model by applying the standard learning loop learned on Day 4 as it is, and confirmed that the loss decreased during the learning process.
- We evaluated the trained model on a test dataset that we had not seen before, and measured the overall accuracy and the accuracy per class.

Glossary of terms:

- torchvision: A computer vision library for PyTorch, providing datasets, models, and image transformation functions.
- CIFAR-10: A 32x32 color image dataset with 10 classes.
- Data Normalization: A preprocessing technique that stabilizes model learning by adjusting the distribution of data to a specific range (e.g. -1 to 1) or a standard normal distribution.
- nn.Conv2d: A PyTorch layer that performs convolution operations on 2D images.
- nn.MaxPool2d: A pooling layer that reduces the size of feature maps and emphasizes important features.
- nn.CrossEntropyLoss: A loss function that is mainly used in multi-class classification problems.
- Epoch: A unit that learns by passing through the entire training dataset once.