IT Talent Training Course

Aug. 2025.

A.I. PROGRAMMING WITH PYTORCH

Instructor :
Daesung Kim

3rd Day - Part 01

INDEX

01 Tensor

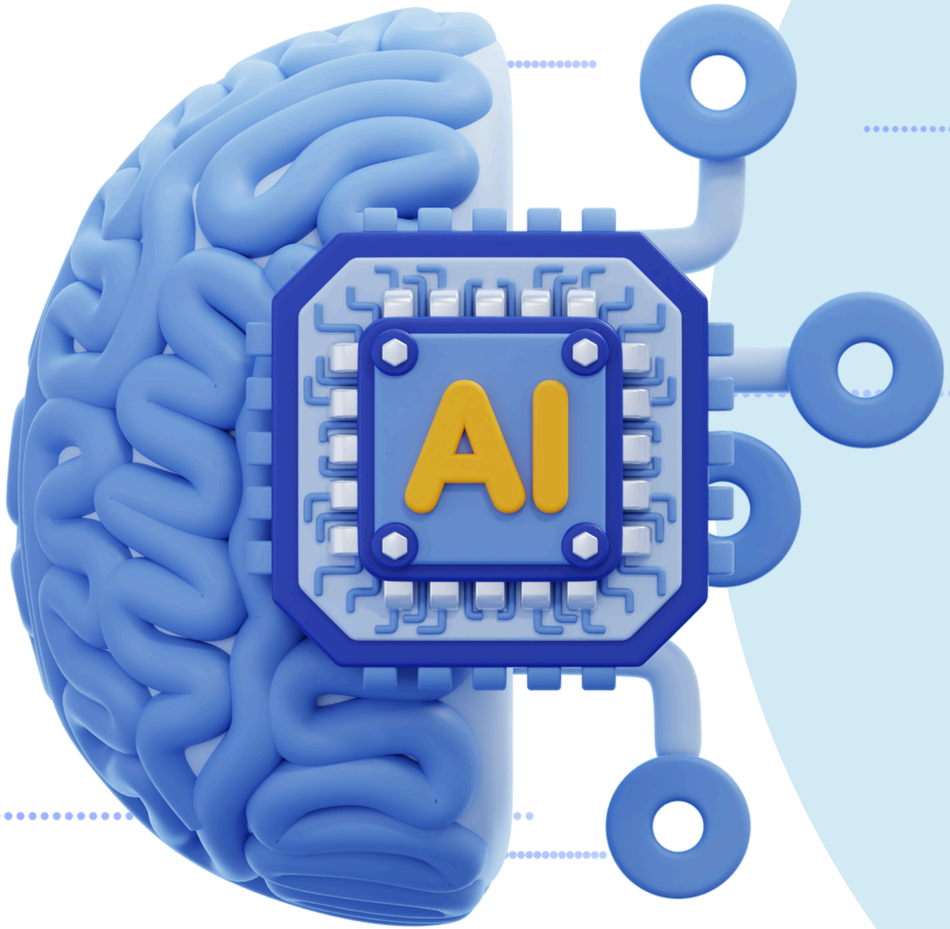
02 Accelerate computation using GPU

03 Tensor manipulation and key operations

COURSE OBJECTIVES

- Understand the relationship between NumPy ndarray and PyTorch Tensor, and explain why tensors are used in deep learning.
- Create PyTorch tensors in various ways and check basic properties (shape, dtype, device).
- Practice activating GPUs in the Google Colab environment and moving tensors between CPU and GPU.
- See and experience the dramatic improvement in large-scale computation speed when using GPUs through code.
- Learn various tensor operations (indexing, slicing, view, squeeze/unsqueeze, permute) and main computation methods.

01
TENSOR



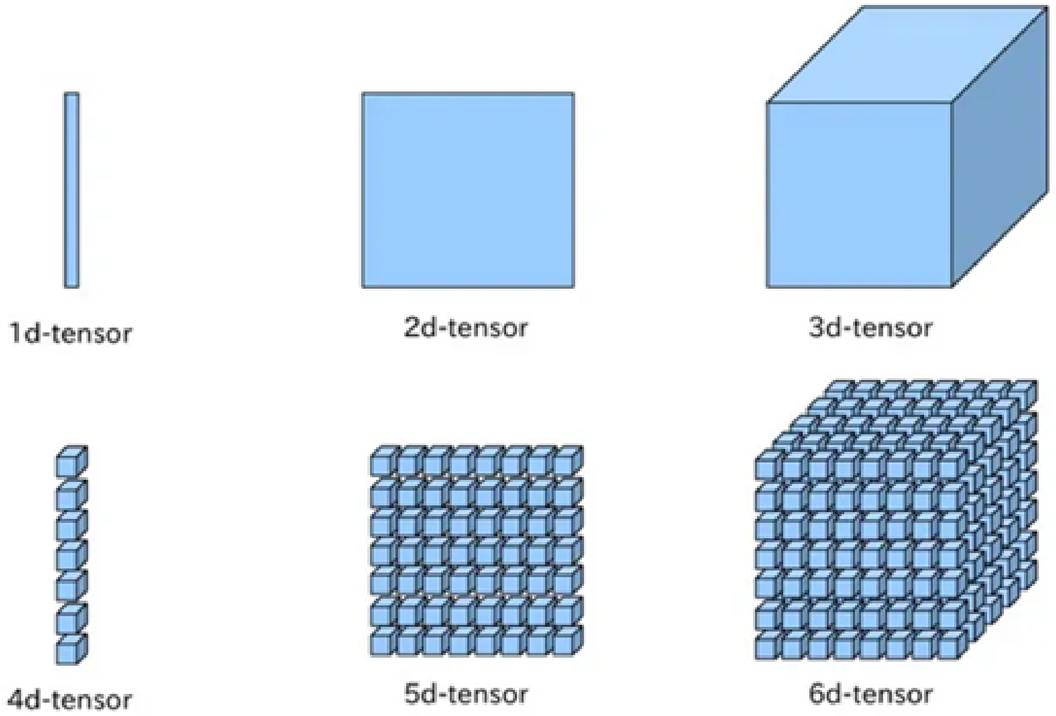
NUMPY TO PYTORCH TENSORS

Tensors in PyTorch are very similar to NumPy's ndarrays - almost like twins. However, tensors have two powerful weapons that ndarrays don't.

1. **GPU-powered parallel processing:** Deep learning involves a much larger amount of matrix multiplication and addition than you can imagine. It is difficult for CPUs alone to handle these calculations. GPUs are specialized in processing these simple repetitive calculations simultaneously (parallel processing) with thousands of small cores, which increases the learning speed of deep learning models by tens or hundreds of times.
2. **Automatic differentiation function (autograd):** The principle of deep learning models 'learning' is to find optimal parameter values through 'differentiation'. PyTorch provides a magical function called autograd that automatically calculates this complex differentiation process. (We will cover this in detail in the afternoon session!)

PYTORCH TENSOR CREATION AND PROPERTIES - 1

Tensors are the most basic unit of data representation in PyTorch. Beyond scalars (0-dimensional), vectors (1-dimensional), and matrices (2-dimensional), all multi-dimensional data can be represented as tensors.



PYTORCH TENSOR CREATION AND PROPERTIES - 2

1. Creating tensors in various ways

PyTorch tensors can be easily created from Python lists or NumPy arrays.

[Code] Day03 - 01_Creating tensors.py

PYTORCH TENSOR CREATION AND PROPERTIES - 2

2. Properties of a tensor

A tensor has properties that include not only its data values, but also its shape, data type (dtype), and information about the device on which the data is stored.

```
# Use the randn_tensor created earlier
tensor = randn_tensor


print(f"Tensor Shape: {tensor.shape}") # Shape
print(f"Tensor DataType: {tensor.dtype}") # Data type
print(f"Tensor Device: {tensor.device}") # Storage device (default is CPU)
```

PYTORCH TENSOR CREATION AND PROPERTIES - 2

3. Create a tensor that follows the properties of another tensor

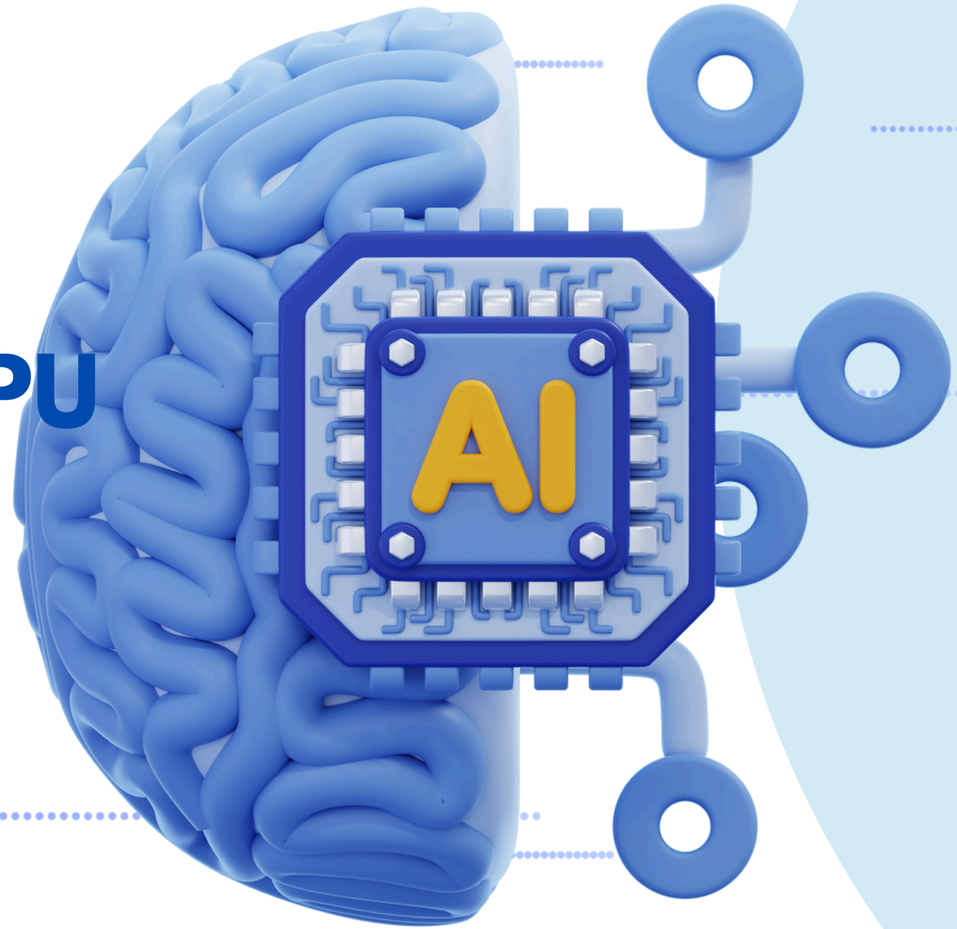
In real projects, there are many cases where you need to create a tensor that has the same shape and properties (dtype, device) as an existing tensor, but different values (e.g., filled with 0 or 1). In this case, it is very convenient to use the _like function.

```
# Original Tensor x_data
x_data = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
print(f"Original Tensor (x_data) shape: {x_data.shape}, dtype: {x_data.dtype}")

# Create tensors with the same shape and dtype as x_data
ones_like_x = torch.ones_like(x_data) # Same as x_data, but with value 1
zeros_like_x = torch.zeros_like(x_data) # Same as x_data, but with value 0
rand_like_x = torch.rand_like(x_data) # Same as x_data, but with random values

print("\n ones_like(x_data):\n", ones_like_x)
print("\n zeros_like(x_data):\n", zeros_like_x)
print("\n rand_like(x_data):\n", rand_like_x)
```

02
ACCELERATE COMPUTATION USING GPU



ENABLING GPUS IN GOOGLE COLAB

- **CPU (Central Processing Unit) vs GPU (Graphics Processing Unit):** Understanding the Difference Between Sequential and Parallel Processing.
- **Google Colab:** Web-based Python Development Environment.

From the Colab top menu, go to Edit > Note Settings > Change Hardware Accelerator to GPU and save.

MOVING TENSORS TO GPU

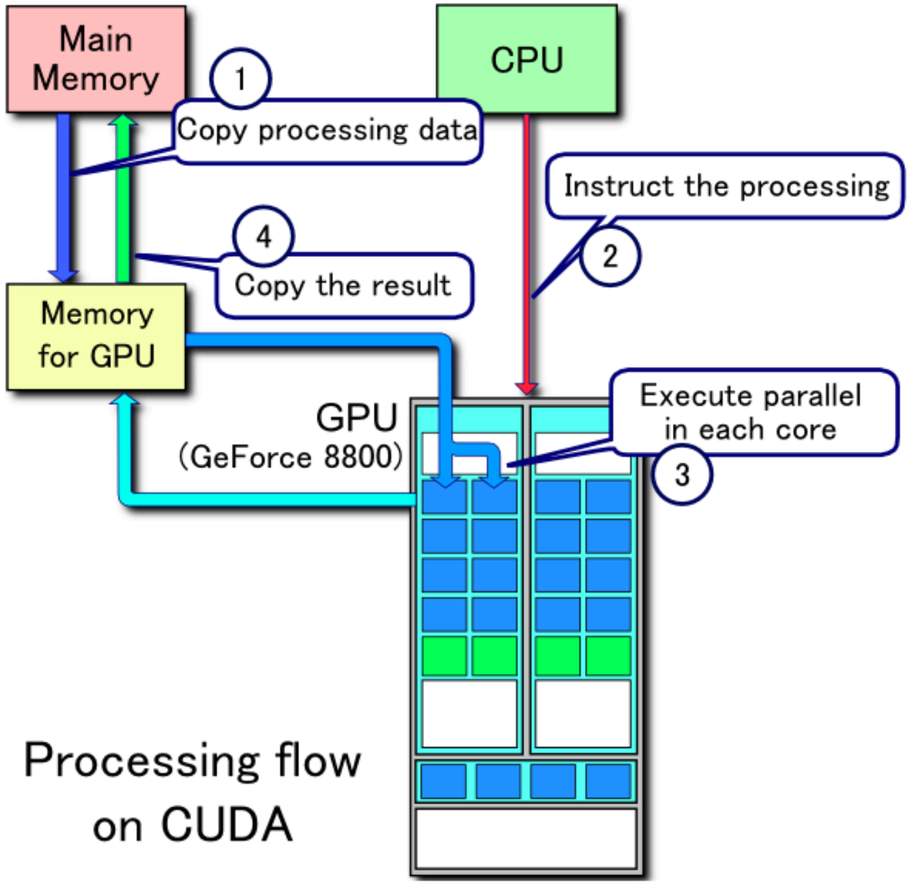
Explicitly move tensors from CPU memory to GPU memory using the .to() method.

```
import torch

# 1. Check if GPU is available and set the device
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Current device: {device}")

# 2. Create tensor on CPU and move to GPU
cpu_tensor = torch.ones(2, 3)
gpu_tensor = cpu_tensor.to(device) # Move to device using .to()

print(f"cpu_tensor device: {cpu_tensor.device}")
print(f"gpu_tensor device: {gpu_tensor.device}")
```



CPU VS GPU COMPUTATIONAL SPEED COMPARISON

Let's check the performance of the GPU directly by multiplying two huge matrices.

```
import time

# Create two very large tensors (on the CPU)
size = 5000
tensor_a = torch.randn(size, size)
tensor_b = torch.randn(size, size)

# --- Measure CPU computation time ---
start_time = time.time()
result_cpu = torch.matmul(tensor_a, tensor_b)
end_time = time.time()
print(f"CPU time: {end_time - start_time:.4f} seconds")

# --- Measure GPU computation time ---
if device == "cuda":
    tensor_a_gpu = tensor_a.to(device)
    tensor_b_gpu = tensor_b.to(device)

    torch.cuda.synchronize() # Synchronize for accurate timing
    start_time = time.time()
    result_gpu = torch.matmul(tensor_a_gpu, tensor_b_gpu)
    torch.cuda.synchronize()
    end_time = time.time()

    print(f"GPU time: {end_time - start_time:.4f} seconds")
```

BRINGING GPU TENSORS BACK TO THE CPU

If you want to visualize the results of a computation done on the GPU (e.g., Matplotlib) or convert them to a NumPy array, you must bring them back to the CPU. Use the .cpu() method.

```
# The tensor gpu_tensor on the GPU (created in the previous exercise)
if 'gpu_tensor' in locals() and gpu_tensor.is_cuda:
print(f"Original tensor is on: {gpu_tensor.device}")

# Copy to CPU with .cpu() method
back_to_cpu_tensor = gpu_tensor.cpu()
print(f"Moved back tensor is on: {back_to_cpu_tensor.device}")

# After coming to the CPU, it can be converted to a NumPy array
numpy_from_gpu = back_to_cpu_tensor.numpy()
print("\nSuccessfully converted to NumPy array:\n", numpy_from_gpu)

else:
print("GPU tensor not available for this exercise. Please run previous cells with GPU runtime.")
```

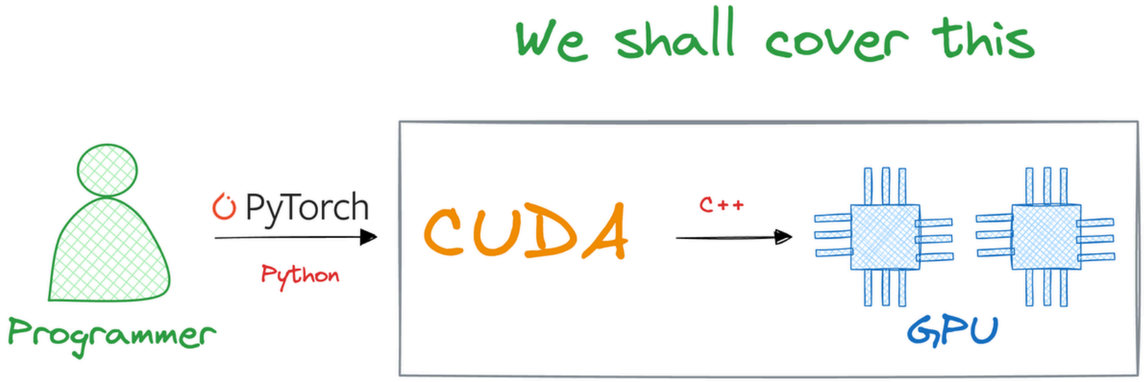
SUMMARY

Summary

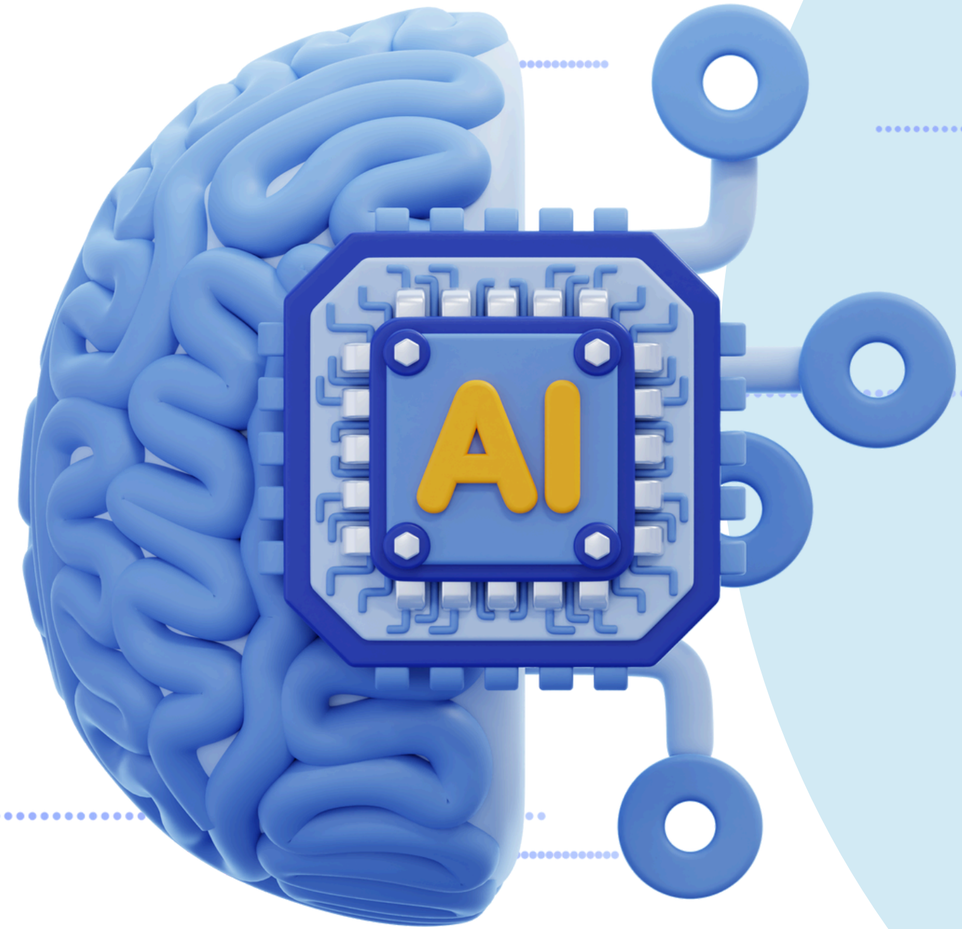
- You can send a tensor to the GPU with .to(device) and get it back with .cpu(). All operations are possible only on the same device, and GPUs have an overwhelming performance advantage in large-scale parallel operations. If you want to work with CPU-based libraries such as NumPy conversion, you must use .cpu().

Glossary of terms:

- CUDA: NVIDIA's GPU-based parallel computing platform.
- .to(device): Method to move/copy a tensor to a specified device.
- .cpu(): Method to move/copy a tensor to the CPU.



03
TENSOR MANIPULATION AND KEY OPERATIONS



INDEXING, SLICING

The usage is exactly the same as NumPy.

```
t = torch.tensor([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print("The entire last column of t:", t[:, -1])
print("The first two rows of t, from column 1 to column 2:\n", t[:2, 1:3])
```

CHANGE THE SHAPE OF A TENSOR

- view(): Reshapes a tensor. (Similar to NumPy's reshape)
- unsqueeze(dim): Adds a dimension of size 1 at the specified dim position.
- squeeze(dim): Removes a dimension of size 1 at the specified dim position. (If not specified, removes all dimensions of size 1.)

[Code] Day03 - 02_Change_the_shape_tensor.py

CHANGE THE ORDER OF TENSOR DIMENSIONS

While view reads data in order and rearranges it into a new shape, permute changes the order of the dimensions themselves. For example, image data is usually loaded in the order (height, width, channels), but PyTorch models take input in the order (channels, height, width). In this case, permute is essential.

```
# Virtual image data in the order of (height, width, channels) (size 10x12, 3 channels)
image_tensor = torch.rand(10, 12, 3)
print(f"Original image tensor shape (H, W, C): {image_tensor.shape}")


# permute(2, 0, 1) : 2nd dimension -> 0th, 0th dimension -> 1st, 1st dimension -> 2nd
# (H, W, C) -> (C, H, W)
permuted_tensor = image_tensor.permute(2, 0, 1)
print(f"Permuted tensor shape (C, H, W): {permuted_tensor.shape}")
```

COMBINING TENSORS

- torch.cat([t1, t2], dim=...): Concatenate along existing dimensions.
- torch.stack([t1, t2], dim=...): Create a new dimension and stack it.

```
t1 = torch.ones(2, 3)
t2 = torch.zeros(2, 3)

# cat(dim=1): (2, 3) + (2, 3) -> (2, 6)
cat_dim1 = torch.cat([t1, t2], dim=1)
print(f"cat(dim=1) shape: {cat_dim1.shape}\n", cat_dim1)

# stack(dim=0): Stack two (2, 3) -> (2, 2, 3)
stack_dim0 = torch.stack([t1, t2], dim=0)
print(f"\nstack(dim=0) shape: {stack_dim0.shape}\n", stack_dim0)
```

KEY MATHEMATICAL OPERATIONS AND AGGREGATE FUNCTIONS

In addition to basic arithmetic operations, aggregate functions such as sum, average, and maximum are essential when calculating the model's loss or evaluating its performance.

```
t = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)

# Matrix multiplication
t_mul = torch.tensor([[1, 0], [0, 1], [1, 1]])
print("Matrix multiplication (matmul):\n", torch.matmul(t, t_mul))

# Aggregation function
print("\nTotal sum:", t.sum())
print("Total average:", t.mean())

# Operation with dim=0 (row direction)
print("\nSum by column:", t.sum(dim=0))
print("Average by column:", t.mean(dim=0))

# Operation with dim=1 (column direction)
print("\nRow Sum:", t.sum(dim=1))
print("Average by row:", t.mean(dim=1))

# Maximum value and its location(index)
print("\nTotal maximum:", t.max())
print("Maximum value of each column:", t.max(dim=0).values)
print("Location of maximum value of each column(index):", t.max(dim=0).indices)
```

SUMMARY

Summary

- Tensor manipulation is a core technology for controlling the data flow of a model. You can change the shape with view, add/remove dimensions with unsqueeze/squeeze, and change the order of dimensions with permute. You can merge tensors with cat/stack, and perform various calculations with aggregate functions such as sum/mean/max and matmul.

Glossary of terms:

- view(): Methods that change the shape of a tensor.
- unsqueeze()/squeeze(): Methods that add or remove dimensions with a size of 1.
- permute(): Methods that explicitly change the order of dimensions of a tensor.
- torch.cat()/torch.stack(): Functions that concatenate or stack tensors.
- Aggregation functions: Functions that summarize multiple values of a tensor into a single value, such as sum(), mean(), and max().

IT Talent Training Course

Aug. 2025.

A.I. PROGRAMMING WITH PYTORCH

Instructor :
Daesung Kim

3rd Day - Part 02

INDEX

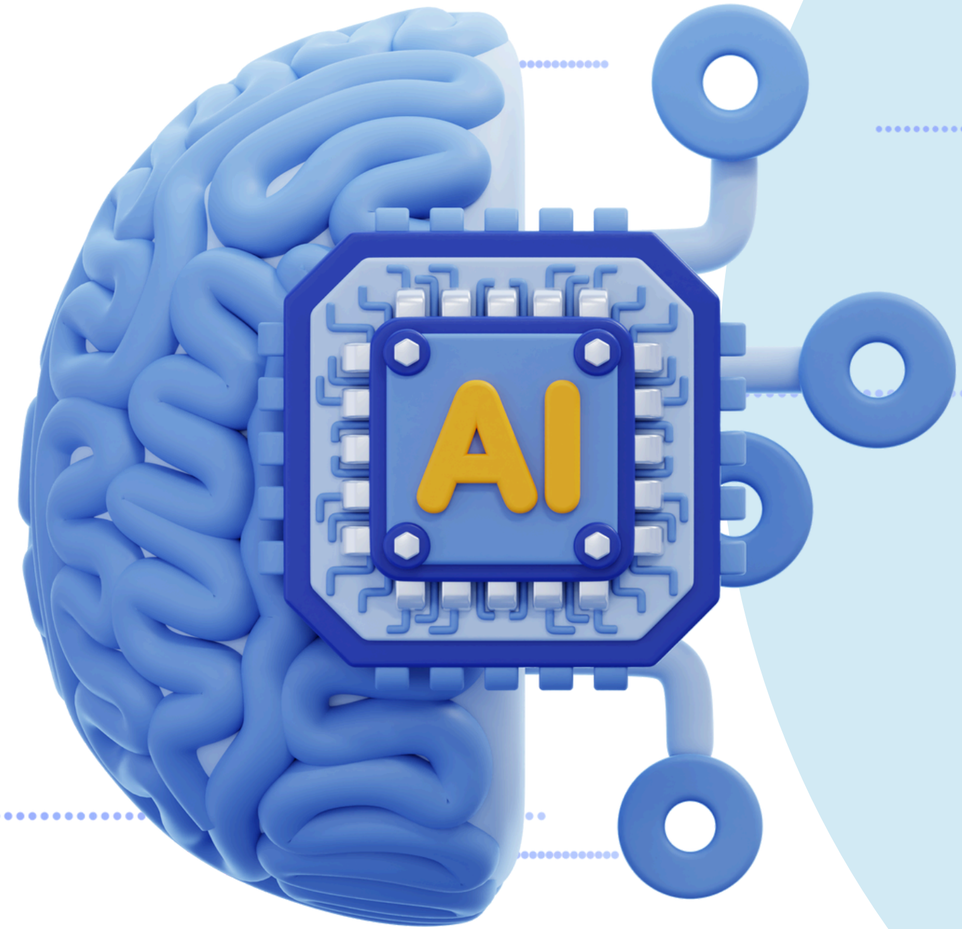
01 Principles of deep learning

02 Autograd

03 Linear regression

04 Integrated Practice Example

01
Principles of Deep Learning



LOSS FUNCTION

Training an AI model is the process of making the model's predictions as close to the actual correct answer as possible. So how can we measure 'close' or 'far'? We use a **loss function** or **cost function.**

The loss function is a function that calculates the difference between the model's predicted value and the actual correct answer (y), that is, the **error**. The larger this error is, the worse the model's performance is, and the smaller the error is, the better the performance is. Therefore, the goal of deep learning training is **to minimize the value of this loss function.**

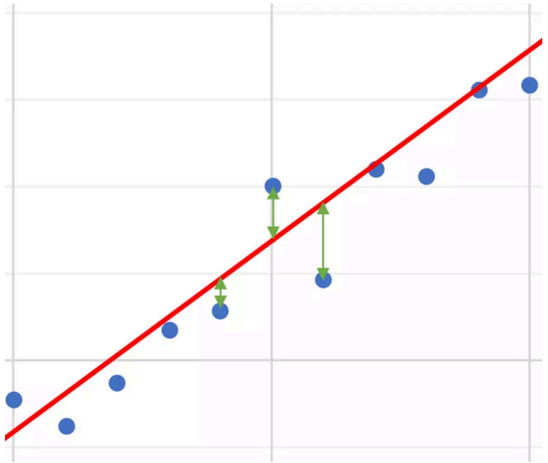
- One of the most representative loss functions is **Mean Squared Error (MSE)**. It is the value obtained by squaring the errors of each data point, adding them together, and then taking the average.

$$\text{Loss (MSE)} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

$\hat{y}_i$: Model's prediction for the i-th data

y_i : Actual correct answer for the i-th data

n : Number of total data



The reason we square the errors instead of just adding them is to focus on the size of the error itself, whether the predicted value is larger or smaller (+ or -) than the true value, and to impose a larger penalty on larger errors.

GRADIENT DESCENT

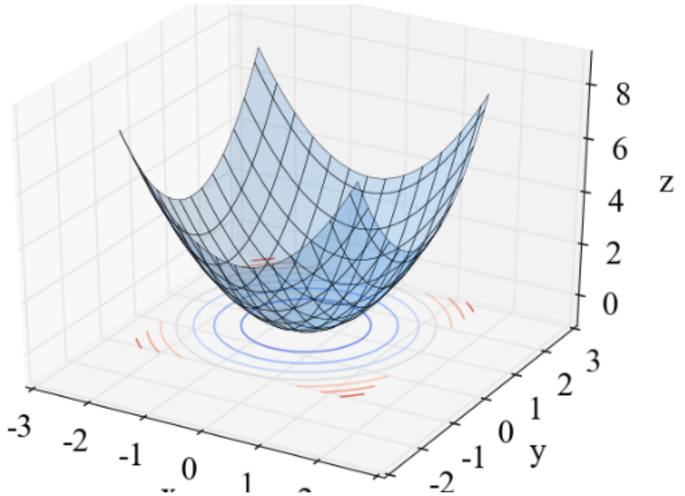
How can we find the values of parameters (W and b) that minimize the loss?

This is where the concept of **Gradient Descent** comes in. Imagine that you are standing on a mountainside covered in thick fog. Your goal is to get down to the lowest point (the valley). You can't see the entire terrain because of the fog, but you can tell which direction is the steepest downhill from your current location. If you take just one step in that steepest downhill direction, you will eventually reach the lowest point.

- **Height of the mountain:** Loss
- **Current position:** Current parameter (W,b) values
- **Direction of steepest descent:** opposite of the gradient of the loss function
- **Size of one step:** Learning rate

$$W_{\text{new}} = W_{\text{old}} - \alpha \frac{\partial \text{Loss}}{\partial W}$$

$$b_{\text{new}} = b_{\text{old}} - \alpha \frac{\partial \text{Loss}}{\partial b}$$



Alpha is the **learning rate**, an important hyperparameter that determines how much to update the parameters at a time, or the 'step size'. If the learning rate is too large, the minimum point may be missed, and if it is too small, the learning speed may be very slow.

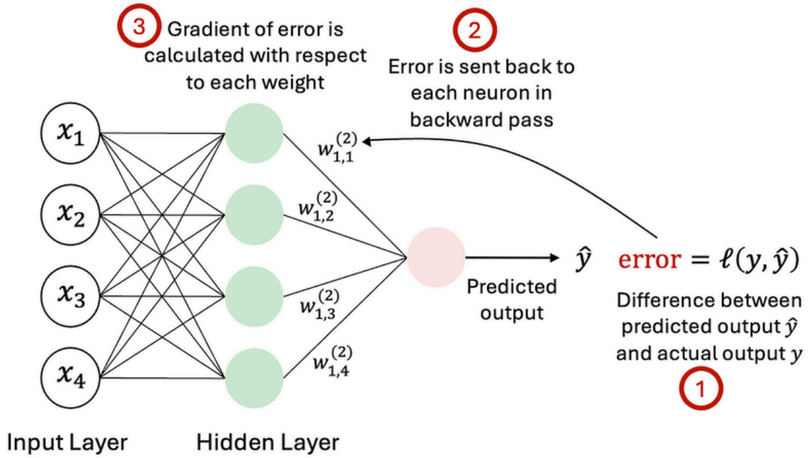
BACKPROPAGATION

To apply gradient descent, we need to know the gradient of each parameter with respect to the loss. How can we efficiently calculate this gradient when there are many parameters complicatedly intertwined, such as in a deep learning model?

The algorithm used here is **Backpropagation**. As the name suggests, it means **propagating** the error backwards.

- **Forward Propagation:** Input data passes through the neural network to create the final prediction value.
- **Loss Calculation:** The loss is calculated using the predicted value and the correct value.
- **Backward Propagation:** Starting from the calculated loss, we calculate how much each parameter contributed to the final loss (i.e., whether it is 'responsible') from the output layer to the input layer using the **Chain Rule**. This 'degree of responsibility' is the gradient we want.

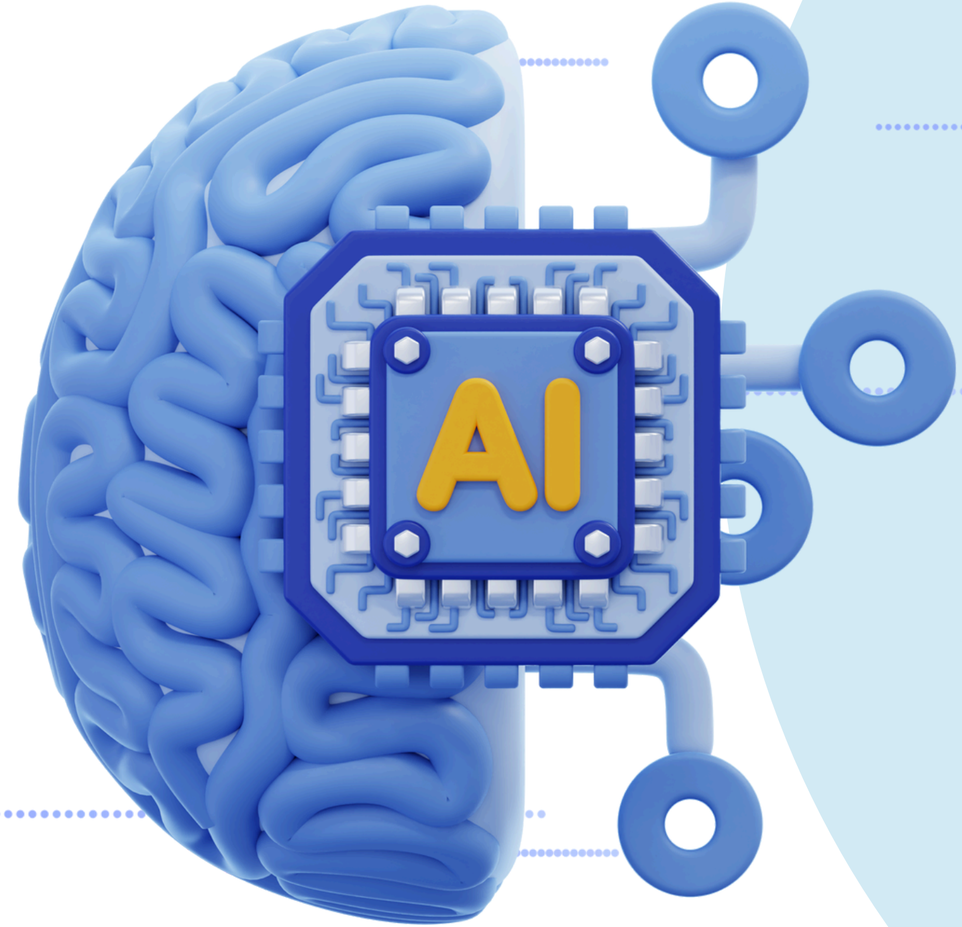
You don't need to understand all the complicated mathematical formulas of backpropagation. The important thing is to have the intuition that it is "**a process of efficiently calculating the responsibility of each component (parameter) for the final error."** And remember that **PyTorch automatically does** this complicated calculation.



02
Autograd

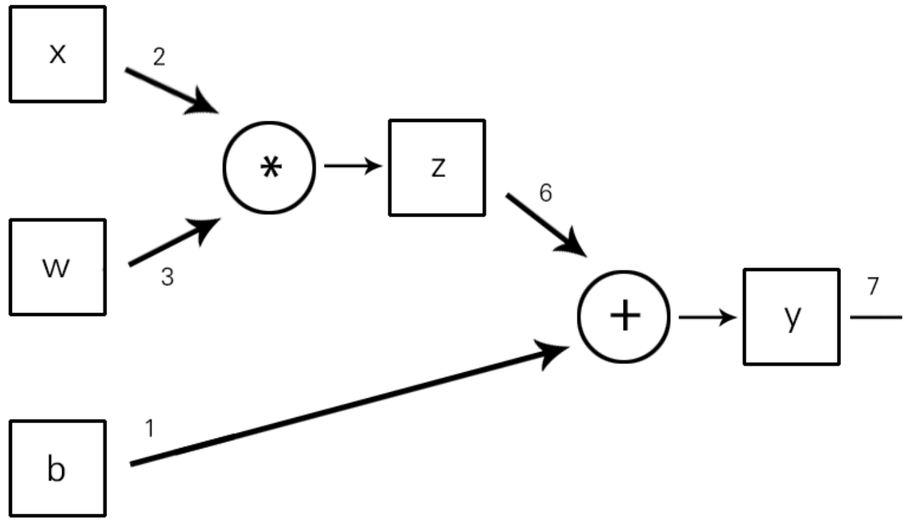
PyTorch solves the complex backpropagation process through an automatic differentiation engine called torch.autograd.

- Gradient Descent
- Loss Function
- Backpropagation



COMPUTATIONAL GRAPH

PyTorch records all operations applied to tensors in a data structure called a **computational graph**. For example, if there is an operation such as y = Wx + b, PyTorch internally stores the process of multiplying x and W, and then adding the result to b to create y as a graph.



Since we have this graph, it becomes possible to later compute the derivatives of each operation (backpropagation), starting from y and tracing backwards to W and b.

TRACING, REQUIRES_GRAD=TRUE

To tell PyTorch which tensor it should compute the gradient for, you need to set the requires_grad=True property when creating the tensor.

```
# Parameters that need to be updated through training
W = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

# Input data (not the target of training)
x = torch.randn(1)

print(W.requires_grad) # True
print(b.requires_grad) # True
print(x.requires_grad) # False
```

To tell PyTorch Parameters whose values must change through learning (weights W, bias b, etc.) must have requires_grad=True.which tensor it should compute the gradient for, you need to set the requires_grad=True property when creating the tensor.

CALCULATING SLOPE

The command to run the gradient calculation is very simple: just call the .backward() method on the loss tensor.

```
# ... (calculate loss via forward propagation code) ...
loss = torch.mean((y_pred - y_real)**2)

# This one line performs backpropagation along the computation graph
loss.backward()
```

The moment this loss.backward() is called, PyTorch walks backwards through the computational graph and computes the gradients of all tensors that have requires_grad=True set.

.GRAD PROPERTY - 1

In PyTorch, when you call loss.backward() multiple times, the new gradient is not overwritten in the .grad property, but is added to the existing value (accumulated). This is the intended behavior for certain models, such as RNNs, but in most cases, you want to compute a new gradient for every training step (epoch or iteration).

Therefore, before performing the next backpropagation, we must always initialize the previous gradient to 0.

```
# Check the gradient accumulation phenomenon
w = torch.tensor([2.0], requires_grad=True)
for i in range(2):
    loss = w * w
    loss.backward()
    print(f"{i+1}th backward w.grad: {w.grad}") # The first is 2*w=4, the second is 4+4=8
    # w.grad.zero_() # If this code is not present, the gradient continues to accumulate.
```

Now we can use this .grad value to update the parameters directly according to the gradient descent formula.

.GRAD PROPERTY - 2

Gradients are stored by default only in leaf node tensors that have requires_grad=True set. Leaf nodes are tensors that users create themselves. The gradients of intermediate tensors (non-leaf tensors) created as a result of operations are used only during the computation process for memory efficiency and are immediately discarded.

```
x = torch.tensor(1.0, requires_grad=True) # leaf node
y = x * 2 # intermediate node
z = y * 3 # intermediate node
z.backward()

print(x.grad) # tensor(6.) -> gradient of x is stored
# print(y.grad) # AttributeError -> gradient of intermediate node y is not stored
```

In most cases, we only need the gradients for the model's parameters (the leaf nodes that the user creates), so this isn't a big deal.

.GRAD PROPERTY - 3

loss.backward() can only be called on **scalars (tensors with one element)** by default, because the loss is a single measurement over the entire batch of predictions. Functions like torch.mean() that we used are used to condense multiple error values into a single scalar value.

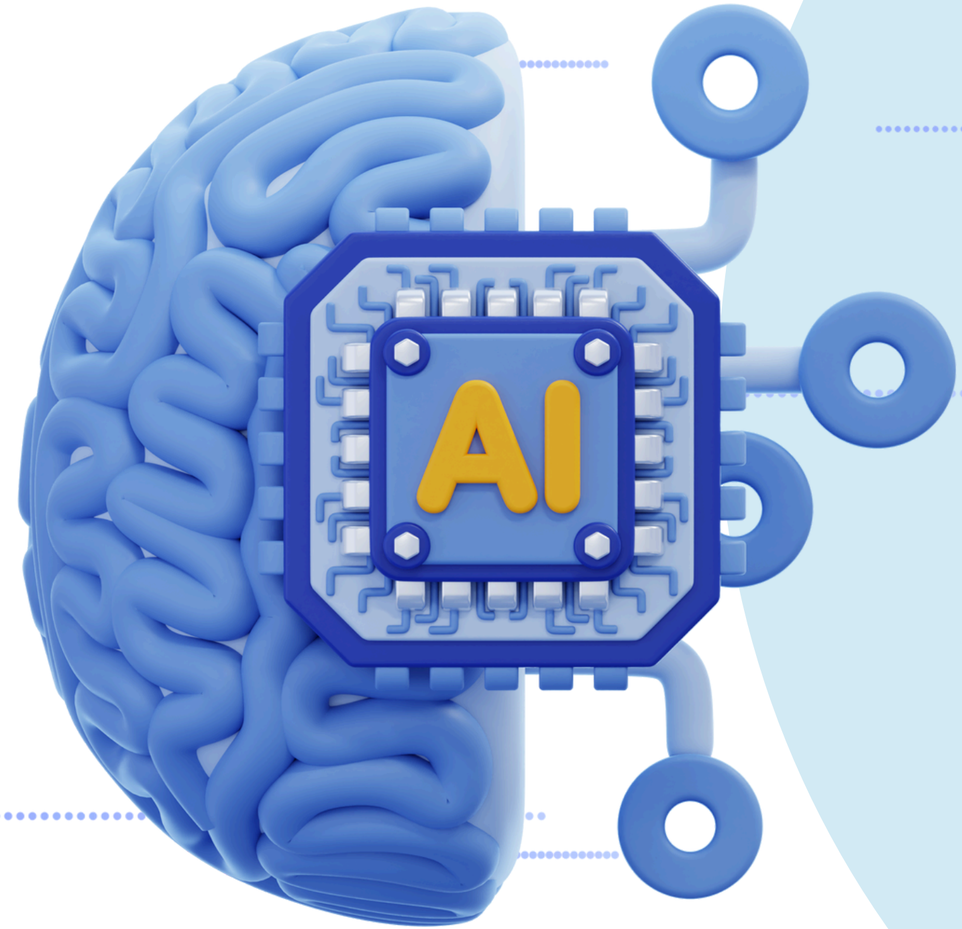
You will also often see the .item() method used during training to print out loss values or save them for visualization.

```
# loss has a single value (a scalar) and is connected to the computational graph
# print(loss) -> tensor(15.234, grad_fn=<MeanBackward0>)

# .item() extracts only pure Python numeric values from the tensor
# It disconnects from the computational graph, so it can be used for logging values or passing them to external libraries
print(loss.item()) # 15.234
```

It is a good idea to get into the habit of using .item() when you only need the values, as using tensors directly connected to a computational graph can take up unnecessary memory.

03
Linear regression



LINEAR REGRESSION - 1

Now, let's put everything we've learned together and train a simple linear regression model using only PyTorch tensor operations.

[Practice Objective]

- Given data with the relationship y = 2x + 1,
- Start with random values for W and b,
- and use gradient descent to make W closer to 2 and b closer to 1.

[Code] Day03 - 03_Linear_Regression.py

LINEAR REGRESSION - 2

[Code Explanation]

1. **Data Preparation:** Create data that follows y = 2x + 1 but has some noise (torch.randn). This is because real data is not always clean.
2. **Parameter Initialization:** Start with random values for W and b. The most important part is to set requires_grad=True so that PyTorch can track their movements.
3. **Training Loop:**
   - Forward: Compute the predicted value y_pred with the current W, b.
   - Loss Computation: Compute the difference between the predicted value and the actual value Y as MSE.
   - Gradient Initialization: Before calling loss.backward(), the .grad value calculated in the previous epoch must be initialized to zero_(). Otherwise, the gradient will keep adding up and learning will proceed in the wrong direction.
   - Backpropagation: Call loss.backward() to compute W.grad and b.grad.
   - Parameter Update: Translate the gradient descent formula directly into code inside the torch.no_grad() context manager. The operations inside this block are not included in the computational graph, which prevents unnecessary tracking.
4. **Check the results:** As training progresses, we can see that the loss gradually decreases, and W and b are very close to 2 and 1, respectively.

MULTIPLE LINEAR REGRESSION

This time, let's look at the case where there are multiple input features. We can see how the concept of matrix multiplication is applied.

[Practice Objective]

- Learn data with the relationship y = W1*x1 + W2*x2 + b (i.e. y = [2, 3] * [x1, x2] + 0.5).

[Code] Day03 - 04_Multiple_linear_regression.py

SUMMARY

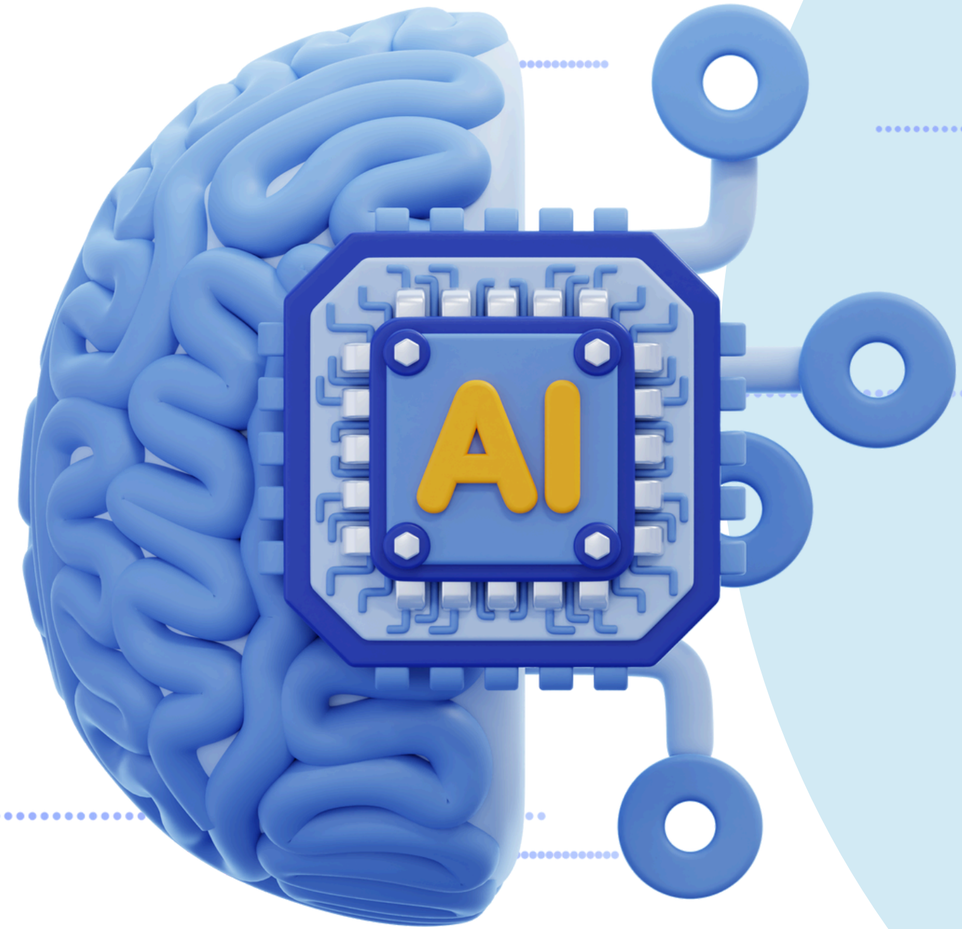
Summary

- The goal of deep learning is to find parameters that minimize the loss function.
- Gradient Descent is a method to gradually update parameters in the direction of decreasing the loss by using the gradient of the loss function.
- Backpropagation is an algorithm that efficiently calculates the gradient of each parameter in a complex neural network.
- PyTorch's autograd automates all of this process. We just set the tensor with requires_grad=True, call loss.backward(), and update the parameters with the gradient value stored in .grad.

Glossary of terms:

- Loss Function: A function that represents the difference between the model's predicted value and the actual correct value. (e.g. MSE)
- Gradient Descent: An optimization algorithm that updates parameters in the opposite direction of the gradient to minimize the loss.
- Learning Rate: A value that determines how much parameters are updated at a time in gradient descent.
- Backpropagation: An algorithm that calculates the gradient of each parameter by going backwards from the error of the output layer toward the input layer.
- Computational Graph: A graph that records the order and relationship of tensor operations. The basis of autograd.
- autograd: PyTorch's automatic differentiation engine.
- requires_grad: A property that determines whether to calculate the gradient of a tensor.
- backward(): A method that is called on the loss tensor to perform backpropagation along the computational graph and calculate the gradient.
- .grad: A property that stores the calculated gradient value for each parameter tensor after calling backward().

04
Integrated Practice Example



LEARNING A SECOND-ORDER POLYNOMIAL REGRESSION MODEL USING ONLY TENSOR OPERATIONS - 1

Project Goal :

- Fit a curve model to data with a quadratic function relationship in the form of $y = ax^2 + bx + c$.
- Train the model parameters (a, b, c) directly using PyTorch's basic tensor operations and autograd.
- Use Matplotlib, which you learned on Day 2, to visually check the data and the results of the trained model.

Project execution steps

1. Data generation and visualization
   - First, we generate training data by adding some noise to the real function relationship $y = 0.5x^2 - 2x + 1$.
   - We can see that the generated data has a curved shape by checking the distribution with a scatter plot in matplotlib.

2. Model parameter initialization
   - The model we will train is $y = w2*x^2 + w1*x + b$.
   - We will create the parameters w2, w1, and b that we need to find through training as tensors with random values.
   - **Key point:** Since these parameters are the targets of training, they must be set to requires_grad=True.

LEARNING A SECOND-ORDER POLYNOMIAL REGRESSION MODEL USING ONLY TENSOR OPERATIONS - 2

3. Implement training loop

- Set the learning rate and the total number of training times (epochs).
- We will implement a for loop with the following logic.
  a. Prediction (Forward Pass): We will compute the predicted value y_pred using the current parameters (w2, w1, b).
  b. Loss Calculation: Calculate the mean square error (MSE) between the predicted value and the actual value Y.
  c. Gradient Calculation:
     - Initialize using .grad.zero_() so that no gradients from previous learning remain.
     - Call loss. backward() to calculate the gradients for each parameter.
  d. Parameter Update:
     - Use the gradient descent formula inside the torch.no_grad() block to directly update w2, w1, b.

4. Check and visualize the results

- After the learning is complete, compare the final w2, w1, b values with the actual values (0.5, -2, 1).
- Overlay the final curve graph created with the learned parameters on the scatter plot of the original data to visually check how well the model explains the data.

LEARNING A SECOND-ORDER POLYNOMIAL REGRESSION MODEL USING ONLY TENSOR OPERATIONS - 3

Challenge

- Observe how the loss changes and the final result changes as you change the learning rate and the number of epochs.
- Modify the code to create a model that learns a 3rd degree polynomial ($y = w3*x^3 + w2*x^2 + w1*x + b$).