IT Talent Training Course

Aug. 2025.

A.I. PROGRAMMING WITH PYTORCH

Instructor :
Daesung Kim

4th Day – Part 01

INDEX

01 nn.module

02 practice

03 Integrated practice example

COURSE OBJECTIVES

- On the third day, you will learn how to create a neural network manually using Tensors and autograd, step by step, in the form of a **reusable and easily extendable 'model' using nn.Module**, a sophisticated and powerful tool provided by PyTorch.

CONCEPT REVIEW: WHAT IS AN ARTIFICIAL NEURAL NETWORK? - 1

Before learning about nn.Module, let's review what the 'artificial neural network' we are going to create is fundamentally. An artificial neural network (ANN) is a computational model created by imitating the way the human brain processes information. Just as the 'neurons', the basic units of the brain, are connected to each other and send and receive signals, artificial neural networks also process information by forming layers of 'artificial neurons (nodes).

The smallest unit: Perceptron and neuron

The most basic component of a neural network is an **artificial neuron or node**. A neuron receives multiple input signals, multiplies each signal by a **weight**, adds them all together, and adds a **bias**. Then, it passes this value through a special function called an **activation function** to create the final output signal.

- Weight (w): A value that indicates how important each input signal is. It is adjusted to the optimal value through learning.
- Bias (b): A value that controls how easily a neuron is activated. It can be thought of as a kind of 'base score'.
- Activation function (f): A function that determines the final output value of a neuron, and plays a very important role in providing 'non-linearity' so that the neural network can learn complex patterns.

$$y = f(\sum_i (w_i x_i) + b)$$

CONCEPT REVIEW: WHAT IS AN ARTIFICIAL NEURAL NETWORK? - 2

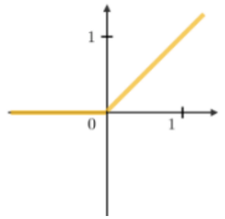
The 'Activation Function' that breathes life into neural networks

If there is no activation function, each layer of the neural network simply repeats linear operations (y=Wx+b). Even if you compose several linear functions, it will only end up as one large linear function. Such a model can never learn complex data patterns that cannot be separated by a straight line.

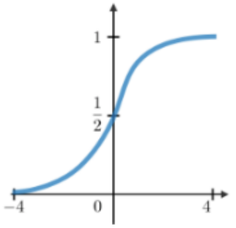
The activation function **adds nonlinearity to the results** of these linear calculations, making the neural network much more flexible and able to approximate complex functions (i.e., able to learn complex patterns).

- **ReLU (Rectified Linear Unit):**
  - If the input is greater than 0, it outputs as is, and if it is less than 0, it outputs 0. It is one of the most widely used activation functions in modern deep learning because it is simple to calculate and performs well.
- **Sigmoid:**
  - It compresses the output to a value between 0 and 1, allowing it to be interpreted as a probability. It is mainly used in the output layer of binary classification problems.

$$g(z) = \max(0, z)$$



$$g(z) = \frac{1}{1 + e^{-z}}$$



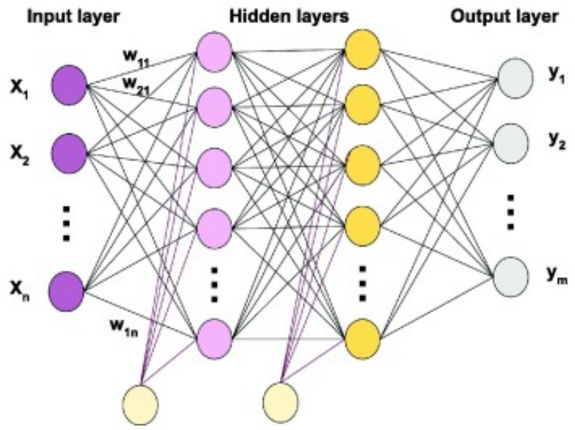
CONCEPT REVIEW: WHAT IS AN ARTIFICIAL NEURAL NETWORK? - 3

'Layers' and 'neural networks' created by gathering neurons

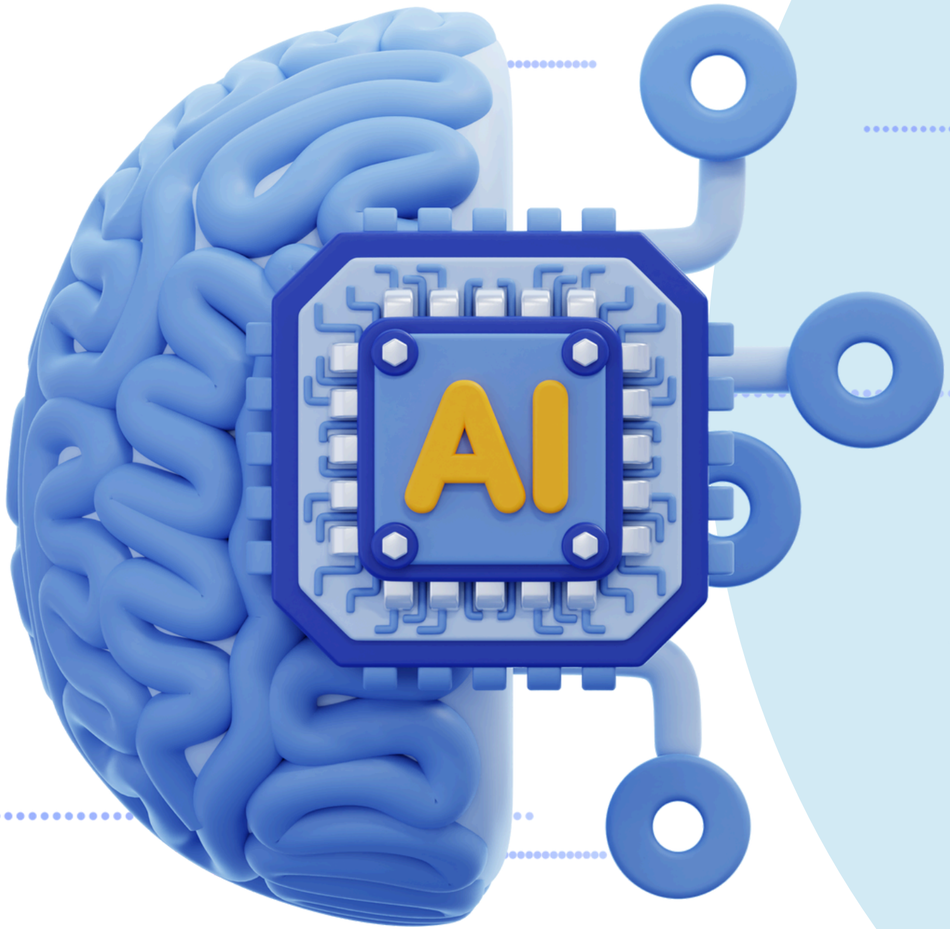
These artificial neurons are grouped together to form a **layer**. Neural networks are usually built by stacking multiple layers.

- **Input Layer:** The layer where data first enters.
- **Hidden Layer:** All layers located between the input layer and the output layer. The deeper (more) these hidden layers are stacked, the more they are called 'deep neural networks (DNNs)'. The actual learning that extracts and combines the features of the input data takes place in the hidden layers.
- **Output Layer:** The layer that outputs the final prediction value of the model.

A neural network in which information flows in one direction only from the input layer through the hidden layers to the output layer is called a **feedforward neural network (FNN)**, and this is the most basic neural network structure.



01
NN.MODULE



WHY USE NN.MODULE?

On the afternoon of the third day, we created two tensors called w and b directly to create a linear regression model y = w*x + b, and set requires_grad=True. This is fine when the model is simple. But what if our model consists of tens or hundreds of w and b pairs (parameters)?

- **Increased complexity:** Creating and managing all parameters one by one is very cumbersome and error-prone.
- **Difficulty in understanding the structure:** As the code becomes longer, it is difficult to see at a glance how the model is structured as a whole.
- **Limitations in reusability and management:** It is difficult to save or load learned parameters, move the entire model from CPU to GPU, etc.

torch.nn.Module is a **PyTorch class** that is the **foundation of all neural network models** and was created to solve these problems. It encapsulates the layers and parameters that make up the model, as well as the order in which the computation is performed (forward pass) when data is input, into a single 'object' for clean management.

STRUCTURE AND CORE METHODS OF NN.MODULE - 1

When we create our own model class by inheriting nn.Module, we usually implement (override) two core methods.

__init__(self): Defining the components of the model

- The __init__ method is a 'constructor' that is called once when the model object is first created. This is where we **define and initialize the components (parts) required for our model**. For example, we create 'layers' that will make up the model, such as linear layers and activation functions, here.

- **Introduction to the main layers:**
  - nn.Linear(in_features, out_features): Fully-Connected Layer, which performs matrix multiplication and bias addition (y=Wx+b) on the input data.
    - in_features: Number of features in the input tensor
    - out_features: Number of features in the output tensor
    - Internally, the weight (W) and bias (b) tensors are automatically generated and managed as special tensors called nn.Parameter. nn.Parameter has requires_grad=True set by default, so autograd automatically tracks it.

STRUCTURE AND CORE METHODS OF NN.MODULE - 2

Example)

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        # Define a linear layer with 10 input features and 5 output features
        self.layer1 = nn.Linear(10, 5)
        # Define a linear layer with 5 input features and 1 output feature
        self.layer2 = nn.Linear(5, 1)
```

STRUCTURE AND CORE METHODS OF NN.MODULE - 3

forward(self, x): Defining the flow of data

- The forward method is where we define how the model will receive input data x, perform calculations, and return the final result.
- You can think of it as a process of assembling pre-made parts (layers) in order in __init__.
- When you call a model object like a function (model(input_data)), PyTorch internally executes this forward method and returns the result.

Example) following __init__

```
# ... __init__ part is the same as above ...

def forward(self, x):
# Pass the input x through layer1.
x = self.layer1(x)
# Pass the output of layer1 through layer2 and return the final result.
x = self.layer2(x)
return x
```

REIMPLEMENTING A LINEAR REGRESSION MODEL WITH NN.MODULE

Let's change the linear regression model we created on Day 3 into a sophisticated class form using nn.Module.

- **Practice Objective:**
  - Define a simple linear regression model with one input feature and one output feature as an nn.Module class.

[Code] Day04 - 01_linear_regression_model_nnModule.py

Expected results from running the code:

```
LinearRegressionModel(
(linear): Linear(in_features=1, out_features=1, bias=True)
)

Input shape: torch.Size([10, 1])
Output shape: torch.Size([10, 1])
linear.weight tensor([[-0.2345]]) # Randomly initialized weights (w)
linear.bias tensor([0.6789]) # Randomly initialized bias (b)
```

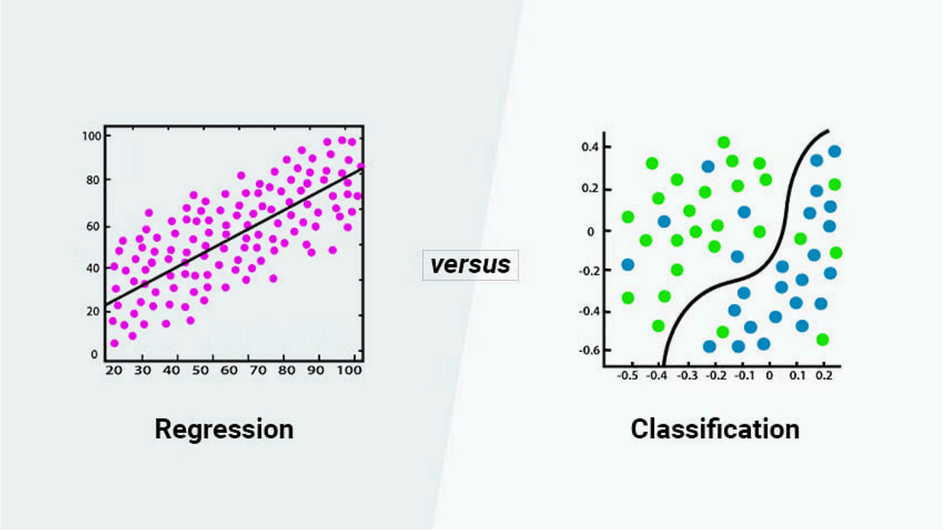
As you can see, we no longer need to create and manage w and b ourselves. nn.Module and nn.Linear handle everything for us. The code is much more concise, and the structure of the model is easier to understand.

LOSS FUNCTION: THE MODEL'S REPORT CARD - 1

Now that we have a model, we need a 'report card' to evaluate how well the model's predictions are performing. This is where the loss function or cost function comes in.

The loss function is a function that calculates the difference (error) between the model's predictions and the actual correct answer (target). The goal of deep learning training is to make the value of this loss function as close to 0 as possible, i.e. minimize the loss.

PyTorch has implemented commonly used loss functions in the torch.nn package.



LOSS FUNCTION: THE MODEL'S REPORT CARD - 2

1. Regression : nn.MSELoss(Mean Squared Error)

Regression is a problem of predicting continuous values such as stock prices, height, and weight. The most widely used loss function at this time is**MSE**.

The calculation formula is the average of the squared values of (correct answer - predicted value) of each data sample. The reason for squaring is to make the size of the error positive, and to give a bigger penalty the larger the error.

$$L_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

```
# Create a mean squared error loss function object
loss_fn = nn.MSELoss()

# Create random predictions and correct answers
predictions = torch.randn(10, 1) # Assume these are the model's predictions
predictions
targets = torch.randn(10, 1) # Assume these are the actual correct answers

# Calculate the loss
loss = loss_fn(predictions, targets)
print(f"MSE Loss: {loss.item()}")
```

LOSS FUNCTION: THE MODEL'S REPORT CARD - 3

2. Classification : nn.CrossEntropyLoss

Classification is a problem of guessing which category of given data belongs to among pre-determined categories such as dog, cat, or car. The standard loss function used in this case is **Cross-Entropy Error**.

Cross-Entropy measures how different the probability distribution predicted by the model is from the probability distribution of the actual correct answer. The more the model assigns a high probability to the correct class and a low probability to other classes, the smaller the cross-entropy loss value.

```
# Create a cross entropy loss function object
loss_fn = nn.CrossEntropyLoss()

# Assume the problem is to classify 3 classes
# BATCH_SIZE=5, NUM_CLASSES=3
predictions = torch.randn(5, 3) # Model's predictions for 5 samples (raw scores)
targets = torch.tensor([1, 0, 2, 1, 0]) # Actual correct class indices for 5 samples

# Calculate the loss
loss = loss_fn(predictions, targets)
print(f"Cross Entropy Loss: {loss.item()}")
```

Important note:
PyTorch's nn.CrossEntropyLoss is a combination of nn.LogSoftmax and nn.NLLLoss. This means that even if the raw values (raw scores or logits) output from the last layer of the model are received as input, it internally applies the Softmax function to convert them to probabilities and then calculates the loss. Therefore, when using this loss function, **there is no need to add a Softmax layer at the end of the model**.

SUMMARY

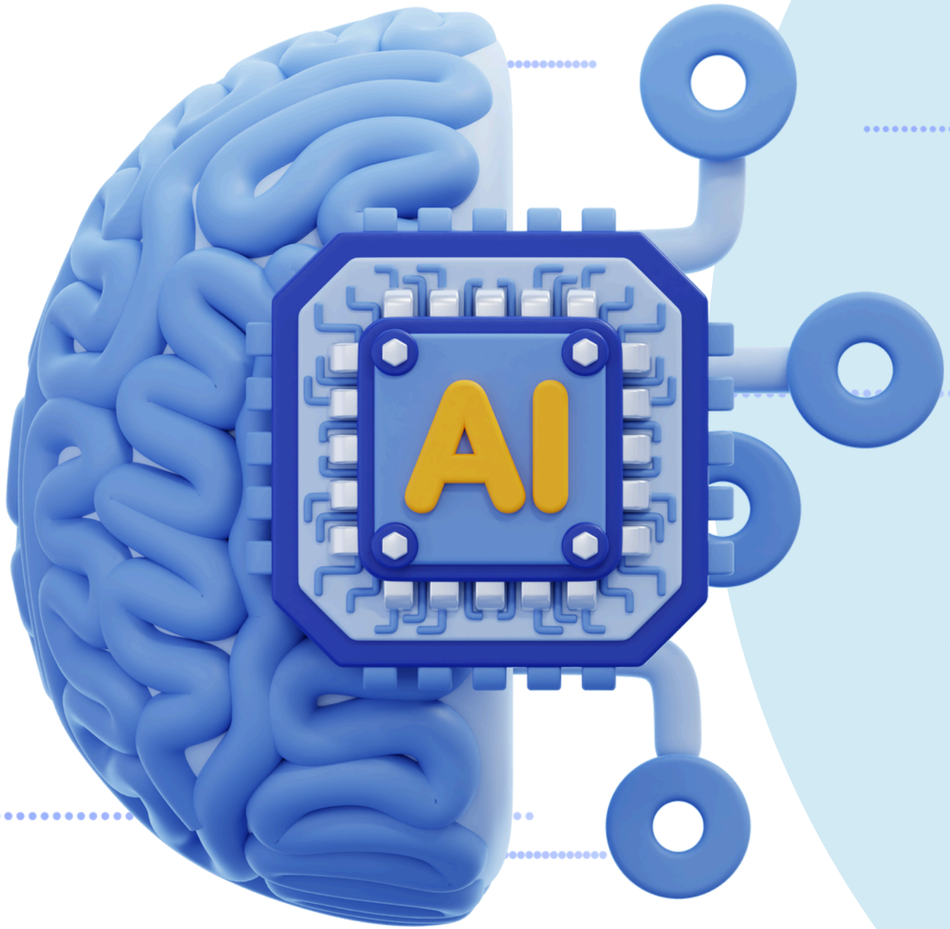
Summary

- Artificial neural networks are made up of neurons that form a layer, and learn complex patterns by securing nonlinearity through activation functions. You can inherit torch.nn.Module to define the structure of the model in __init__ and the flow of data in forward to perform systematic modeling. You should select a loss function that suits the problem type, such as nn.MSELoss for regression problems and nn.CrossEntropyLoss for classification problems.
- I completed a mini-project that synthesized the morning's learning content, designed a neural network model that classifies nonlinear data, and verified its performance before learning.

Glossary of terms:

- Artificial Neural Network (ANN): A machine learning model inspired by the brain.
- Activation Function: A nonlinear function that determines the output value of a neuron (e.g. ReLU, Sigmoid).
- nn.Module: A base class that all neural network models in PyTorch inherit from. It encapsulates the structure, parameters, and computational flow of the model.
- nn.Linear: A fully connected layer that applies a linear transformation (y=Wx+b) to the input.
- Loss Function: A function that computes the error between the predicted value and the actual value.
- Optimizer: An algorithm that updates the learnable parameters of the model using the computed gradient (e.g. SGD, Adam).

02
PRACTICE



REIMPLEMENTING A LINEAR REGRESSION MODEL WITH NN.MODULE - 1

1. Regression : nn.MSELoss(Mean Squared Error)

Reimplementing a linear regression model with nn.Module

```
import torch
import torch.nn as nn

# 1. Define a model class
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super(LinearRegressionModel, self).__init__()
        # Define a linear layer with 1 input feature and 1 output feature.
        self.linear = nn.Linear(in_features=1, out_features=1) #

    def forward(self, x):
        # Pass the linear layer defined in __init__ and return the predicted value.
        return self.linear(x)

# 2. Create a model object and check the structure
model_reg = LinearRegressionModel()
print("--- Linear Regression Model ---")
print(model_reg)
```

REIMPLEMENTING A LINEAR REGRESSION MODEL WITH NN.MODULE - 2

2. Creating a classification model with hidden layers and activation functions

This time, let's create a slightly more complex model. Let's define a simple multilayer perceptron (MLP) classification model with a hidden layer and a nonlinear activation function (nn.ReLU). This model takes two input features and classifies them into one of three classes.

```
# 1. Define the MLP classifier model class
class MLPClassifier(nn.Module):
def __init__(self):
super(MLPClassifier, self).__init__()
# Input layer (input_features=2 -> hidden_features=10)
self.hidden_layer = nn.Linear(2, 10)
# Activation function
self.activation = nn.ReLU() #
# Output layer (hidden_features=10 -> output_classes=3)
self.output_layer = nn.Linear(10, 3)

def forward(self, x):
# Define the data flow. x = self.hidden_layer(x) # Pass through hidden layer
x = self.activation(x) # Pass through activation function
x = self.output_layer(x) # Pass through output layer
return x

# 2. Create model object and check structure
model_clf = MLPClassifier()
print("\n--- MLP classifier model ---")
print(model_clf)
```

APPLYING A LOSS FUNCTION - 1

Let's apply a **loss function**, a "report card" to evaluate how well the model's predictions were correct.

1. Applying nn.MSELoss to a regression model

For regression problems that predict continuous values, such as stock prices or height, Mean Squared Error is used.

```
# Create a virtual data
X_reg = torch.randn(10, 1) # Input (10 samples, 1 feature)
y_reg = 2 * X_reg + 1 + torch.randn(10, 1) * 0.1 # Actual correct answer (y = 2x + 1 + noise)

# 1. Predictions of the regression model
predictions_reg = model_reg(X_reg)

# 2. Define the MSE loss function and calculate the loss
loss_fn_mse = nn.MSELoss() #
loss_reg = loss_fn_mse(predictions_reg, y_reg)

print(f"Predicted values of the linear regression model (first 5): \n{predictions_reg[:5].data}")
print(f"Actual correct answers (first 5): \n{y_reg[:5].data}")
print(f"\nMean squared error (MSE Loss): {loss_reg.item()}")
```

APPLYING A LOSS FUNCTION - 2

Let's apply a **loss function**, a "report card" to evaluate how well the model's predictions were correct.

2. Applying nn.CrossEntropyLoss to a classification model

For classification problems that match categories such as dogs and cats, **Cross-Entropy Error** is used. PyTorch's nn.CrossEntropyLoss is convenient because it takes the raw output (logit) of the model as input and internally calculates Softmax.

```
# Create virtual data
X_clf = torch.randn(5, 2) # Input (5 samples, 2 features)
y_clf = torch.tensor([0, 1, 2, 1, 0]) # Actual correct class (one of 0, 1, 2)

# 1. Predictions of classification model (raw scores)
predictions_clf = model_clf(X_clf)

# 2. Define Cross Entropy loss function and calculate loss
loss_fn_ce = nn.CrossEntropyLoss() #
loss_clf = loss_fn_ce(predictions_clf, y_clf)

print(f"\nPredictions of MLP classification model (raw scores): \n{predictions_clf.data}")
print(f"Actual correct class: \n{y_clf.data}")
print(f"\nCross Entropy Error (Cross Entropy Loss): {loss_clf.item()}")
```

LOSS FUNCTION: THE MODEL'S REPORT CARD

2. Creating a classification model with hidden layers and activation functions

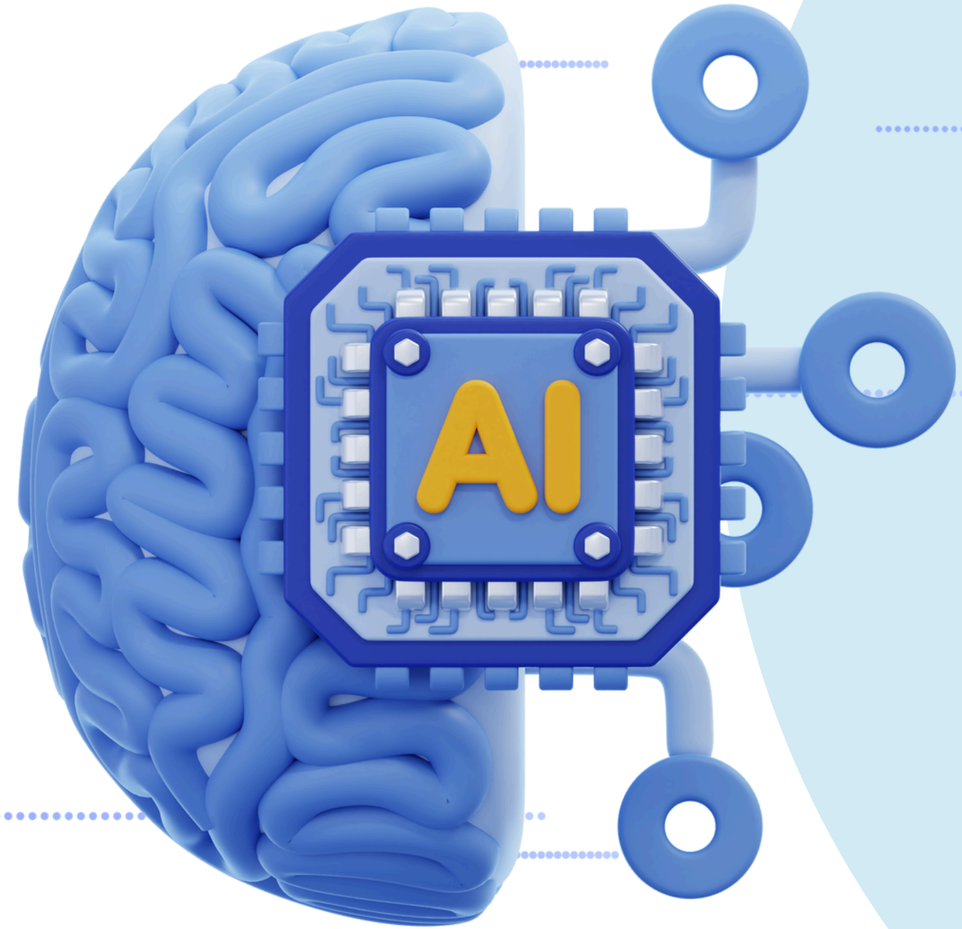
This time, let's create a slightly more complex model. Let's define a simple multilayer perceptron (MLP) classification model with a hidden layer and a nonlinear activation function (nn.ReLU). This model takes two input features and classifies them into one of three classes.

```
# 1. Define the MLP classifier model class
class MLPClassifier(nn.Module):
def __init__(self):
super(MLPClassifier, self).__init__()
# Input layer (input_features=2 -> hidden_features=10)
self.hidden_layer = nn.Linear(2, 10)
# Activation function
self.activation = nn.ReLU() #
# Output layer (hidden_features=10 -> output_classes=3)
self.output_layer = nn.Linear(10, 3)

def forward(self, x):
# Define the data flow. x = self.hidden_layer(x) # Pass through hidden layer
x = self.activation(x) # Pass through activation function
x = self.output_layer(x) # Pass through output layer
return x

# 2. Create model object and check structure
model_clf = MLPClassifier()
print("\n--- MLP classifier model ---")
print(model_clf)
```

03
INTEGRATED PRACTICE EXAMPLE



REIMPLEMENTING A LINEAR REGRESSION MODEL WITH NN.MODULE - 1

We will work on a mini-project to build a simple neural network from scratch to classify non-linearly distributed data using all of the nn.Module, nn.Linear, nn.ReLU, and nn.CrossEntropyLoss that we learned in the morning session.

Goal:
Create a neural network model that classifies data with a 'moon-shaped' distribution that can never be divided by a straight line.

REIMPLEMENTING A LINEAR REGRESSION MODEL WITH NN.MODULE - 2

1. Data preparation and visualization

First, let's create the data to be classified and check it visually. We will create two crescent-shaped clusters of data using the sklearn library.

```
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt

# 1. Data generation
X_project, y_project = make_moons(n_samples=200, noise=0.15, random_state=42)
X_project = torch.tensor(X_project, dtype=torch.float32)
y_project = torch.tensor(y_project, dtype=torch.int64)

# 2. Data visualization
plt.figure(figsize=(8, 6))
plt.scatter(X_project[:, 0], X_project[:, 1], c=y_project, cmap=plt.cm.coolwarm)
plt.title("Moon-shaped Toy Dataset")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()

print(f"Input shape: {X_project.shape}")
print(f"Target shape: {y_project.shape}")
```

REIMPLEMENTING A LINEAR REGRESSION MODEL WITH NN.MODULE - 3

2. Defining the model, loss function, and optimizer

To classify this nonlinear data, we define a model similar to the MLPClassifier we created earlier. And along with the loss function, we will also set up the **Optimizer** that we will learn in the afternoon. The optimizer is responsible for updating the parameters (weights and biases) of the model based on the calculated gradient.

```
import torch.optim as optim

# 1. Define the model (2 outputs since we classify into 2 classes)
class MoonClassifier(nn.Module):
def __init__(self):
super().__init__()
self.layer1 = nn.Linear(2, 16)
self.activation = nn.ReLU()
self.layer2 = nn.Linear(16, 2) # Print scores for classes 0 and 1

def forward(self, x):
return self.layer2(self.activation(self.layer1(x)))

# 2. Instantiate the model, loss function, and optimizer
project_model = MoonClassifier()
project_loss_fn = nn.CrossEntropyLoss()
# optim.SGD: Stochastic Gradient Descent, the most basic optimizer [cite: 109]
# lr(learning_rate): How much to change the parameters Learning rate to decide whether to update
project_optimizer = optim.SGD(project_model.parameters(), lr=0.1)

print("--- Mini Project Model ---")
print(project_model)
```

REIMPLEMENTING A LINEAR REGRESSION MODEL WITH NN.MODULE - 4

3. Prediction and loss calculation (before training)

Now, let's see what predictions our model, which has not been trained yet, makes and how large the error (loss) is. Reducing this loss value is 'learning', and this process will be covered in detail in the afternoon session.

```
# 1. Predictions before training
initial_predictions = project_model(X_project)

# 2. Calculate loss before training
initial_loss = project_loss_fn(initial_predictions, y_project)

# Use the index of the largest value in the prediction result as the class prediction
_, predicted_classes = torch.max(initial_predictions, 1)

# 3. Calculate accuracy before training
initial_accuracy = (predicted_classes == y_project).sum().float() / len(y_project)

print(f"Pre-training, model's predictions (first 10): {predicted_classes[:10]}")
print(f"Actual answers (first 10): {y_project[:10]}")
print(f"\nLoss before training (Initial Loss): {initial_loss.item():.4f}")
print(f"Initial Accuracy: {initial_accuracy.item():.4f}")
```

SUMMARY

Summary

- Artificial neural networks are made up of neurons that form a layer, and learn complex patterns by securing nonlinearity through activation functions.
- You can inherit torch.nn.Module to define the structure of the model in __init__ and the flow of data in forward to perform systematic modeling.
- You should select a loss function that suits the problem type, such as nn.MSELoss for regression problems and nn.CrossEntropyLoss for classification problems.
- I completed a mini-project that synthesized the morning's learning content, designed a neural network model that classifies nonlinear data, and verified its performance before learning.

Glossary of terms:

- Artificial Neural Network (ANN): A machine learning model inspired by the brain's working principles.
- Activation Function: A nonlinear function that determines the output value of a neuron (e.g. ReLU, Sigmoid).
- nn.Module: A base class that all neural network models in PyTorch inherit from. It encapsulates the structure, parameters, and computational flow of the model.
- nn.Linear: A fully connected layer that applies a linear transformation (y=Wx+b) to the input.
- Loss Function: A function that calculates the error between the predicted value and the actual value.
- Optimizer: An algorithm that updates the model's learnable parameters using the computed gradients (e.g. SGD, Adam).

IT Talent Training Course

Aug. 2025.

A.I. PROGRAMMING WITH PYTORCH

Instructor :
Daesung Kim

4th Day – Part 02

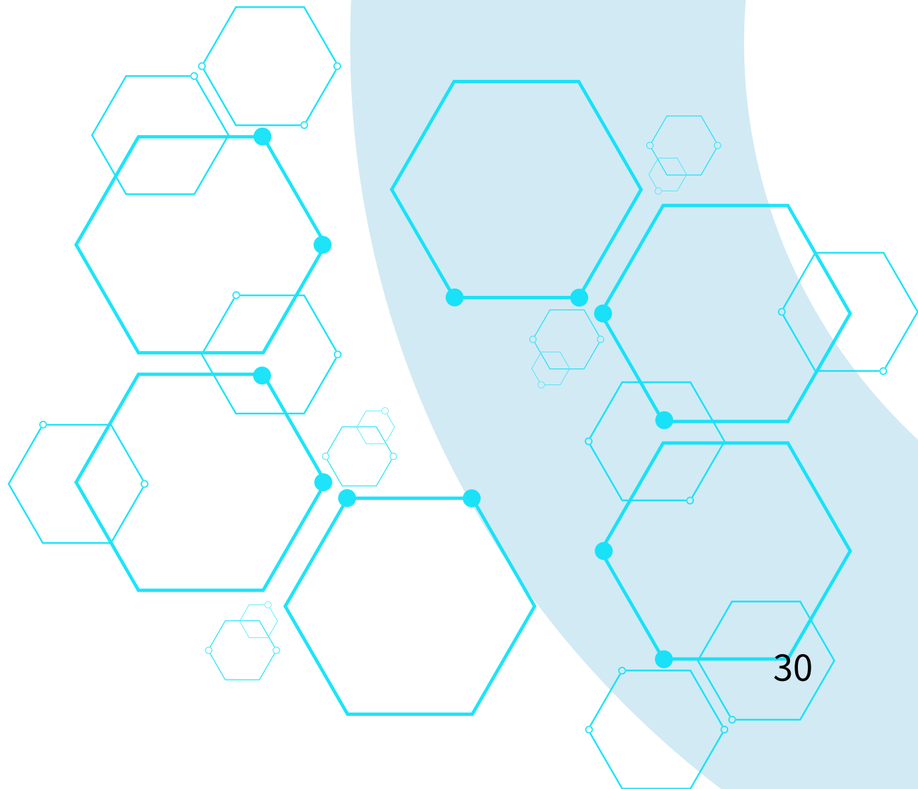
INDEX

01 Optimizer

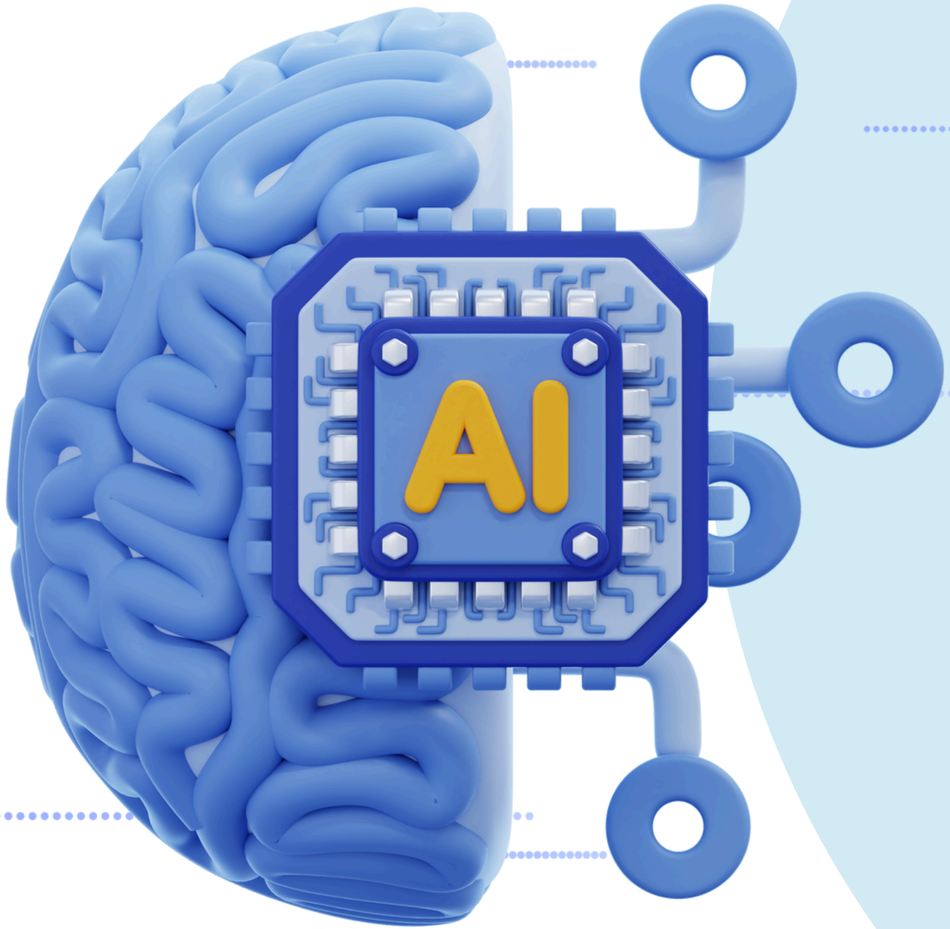
02 Training Loop

03 Dataset, DataLoader

04 Practice



01
Optimizer



WHAT IS AN OPTIMIZER?

Do you remember the third day of practice? We calculated the gradient with loss.backward() and then updated the parameters by writing the following code ourselves.

```
w = w − learning_rate * w.grad
b = b − learning_rate * b.grad
```

This method is good for understanding the principle of gradient descent, but **when the model becomes complex** and has millions of parameters, it becomes **nearly impossible to write** this update code for all parameters.

Optimizer is an object that **automates this parameter update process**. We just tell the optimizer 'which parameters to update' and 'which optimization algorithm to use'. Then, the optimizer automatically updates all parameters according to the specified algorithm using the gradients (w.grad, b.grad, etc.) calculated by loss.backward().

REPRESENTATIVE OPTIMIZERS: SGD AND ADAM

The torch.optim package implements a variety of optimization algorithms. Among them, I will introduce two of the most basic and widely used ones.

- **optim.SGD (Stochastic Gradient Descent):**
  - This is the most basic gradient descent. The reason it is called 'stochastic' is because it updates the parameters using a small amount of data (mini-batch) rather than the entire dataset.
  - It is much faster than using the entire data, and sometimes noise can have a positive effect on the learning process.
  - Conceptually, it is closest to the w = w - lr * w.grad that we implemented manually on Day 3.

- **optim.Adam (Adaptive Moment Estimation):**
  - It is one of the most widely used optimizers at present, and in most cases, it is faster and more stable than SGD.
  - The core idea of Adam is 'Adaptive Learning Rate'. Unlike SGD, which applies the same learning rate to all parameters, Adam adjusts the learning rate for each parameter individually.
  - Parameters that are not updated frequently are trained quickly with a larger learning rate, and parameters that are updated too frequently are trained at a smaller learning rate to help them converge stably.
  - Unless there is a specific reason, using the Adam optimizer can be a good starting point.

USING OPTIMIZERS IN PYTORCH

Creating an optimizer is very simple.

- We tell it what parameters of the model we want to update (model.parameters())
- and specify any necessary hyperparameters, such as the learning rate (lr).

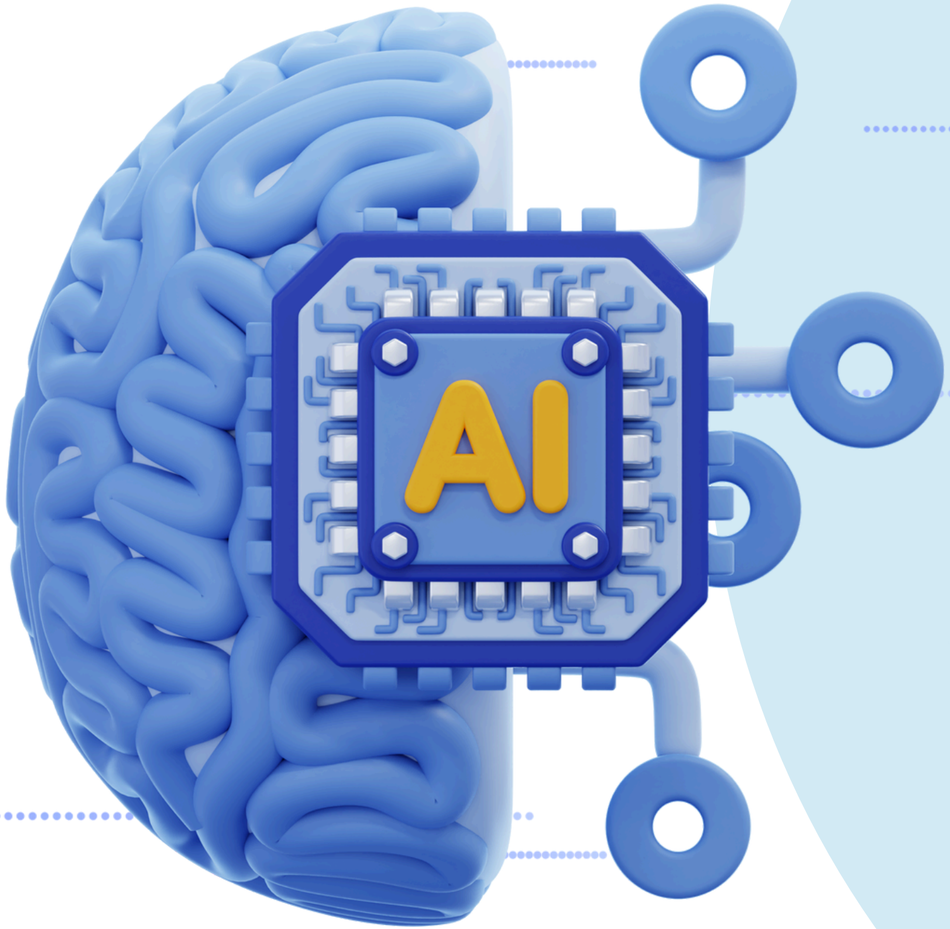
```
# Assuming we have a model created in the morning session
model = LinearRegressionModel(input_dim=1, output_dim=1)

# Create an SGD optimizer
# model.parameters() contains all the learnable parameters (w, b, etc.) in the model.
optimizer_sgd = torch.optim.SGD(model.parameters(), lr=0.01)

# Create an Adam optimizer
optimizer_adam = torch.optim.Adam(model.parameters(), lr=0.01)
```

Now we are ready to perform all parameter updates with just one command: optimizer.step().

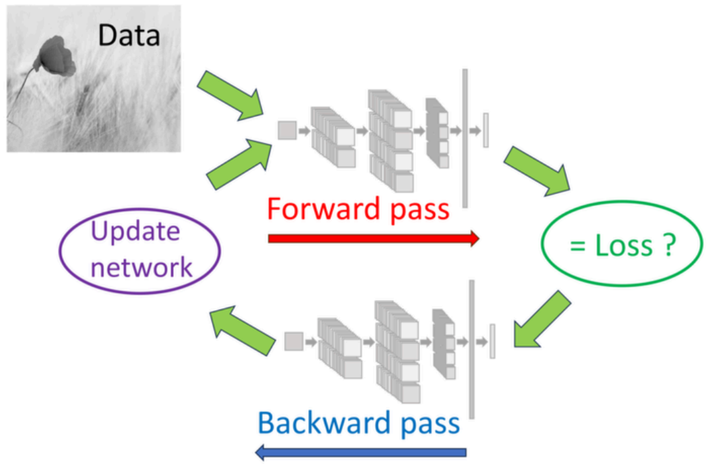
02
Training Loop



PYTORCH STANDARD LEARNING LOOP

- **nn.Module:** The base class of PyTorch models. It defines the forward propagation logic via the forward() method.
- **Compute the loss:** Compute the error using the loss function, such as loss = loss_function(predictions, labels) .
- **loss.backward():** It automatically computes the gradient of each parameter with respect to the loss.

With the optimizer, the training process in PyTorch has a very formalized 3-step flow. These 3 steps are the core patterns that will be used in almost all the training codes of deep learning models that you will build in the future, so it is important to be familiar with them.



1. INITIALIZE SLOPE

optimizer.zero_grad()

- **Role:** Clears all the gradients (.grad property) of the parameters computed in the previous loop to 0.
- **Reason:** By default, PyTorch calculates the gradients anew and 'accumulates' them each time loss.backward() is called. If the gradients are not initialized, the gradients of the current batch as well as the previous batches will continue to accumulate, which will cause the parameters to be updated in the wrong direction. Therefore, it is essential to initialize the gradients at the beginning of every training loop.

2. PERFORMING BACKPROPAGATION

loss.backward()

- **Role:** Compute the gradient for each parameter of the model based on the loss computed with the data of the current batch.
- This step is the same as what we learned in Day 3. The computed gradient is stored in the .grad property of each parameter tensor.

3. PARAMETER UPDATE

optimizer.step()

- **Role:** Updates the parameters of the model that the optimizer maintains internally.
- **Behavior:** The optimizer automatically performs the operation w = w - lr * w.grad for all parameters based on the gradient values stored in the .grad property of each parameter, and the learning rate (lr) and optimization algorithm (e.g., SGD, Adam) specified at creation time.

TRAINING LOOP

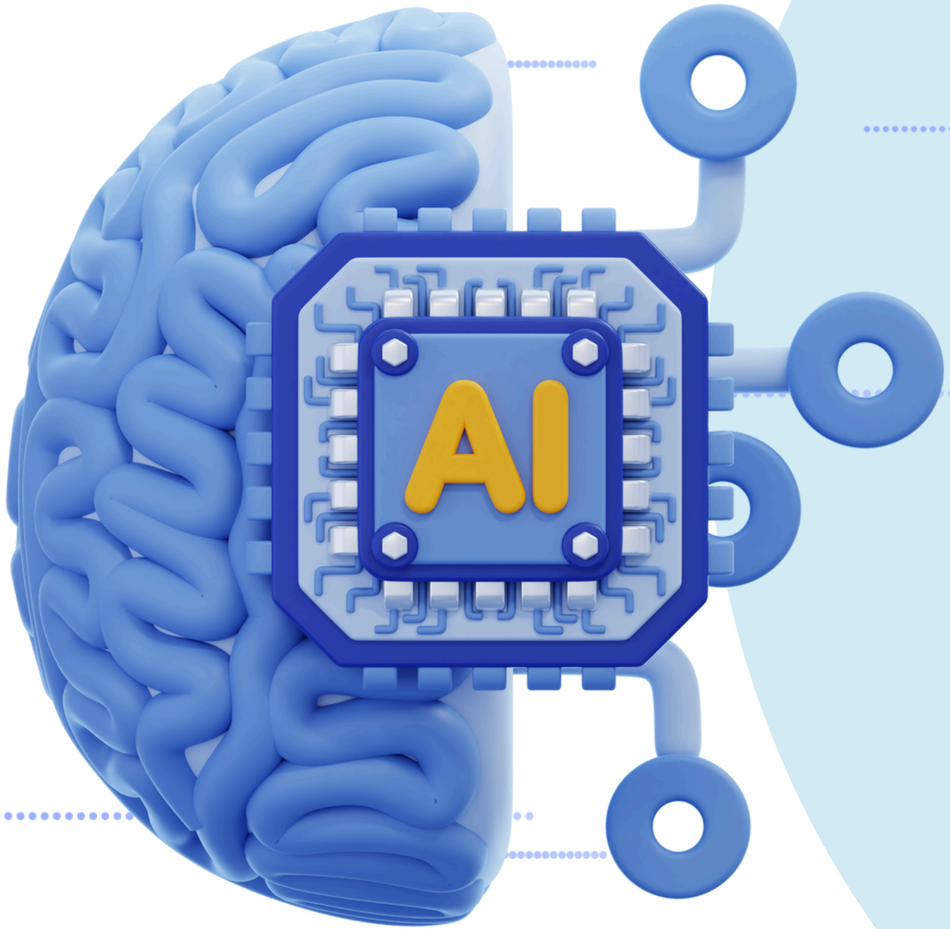
These three steps - gradient initialization - backpropagation - parameter update - make up one learning loop.

```
# Pseudocode (Full code in the tutorial!)
for inputs, labels in data_loader:
# 1. Forward: Compute predictions
predictions = model(inputs)

# 2. Compute loss
loss = loss_function(predictions, labels)

# 3. Step 3 of the training loop
optimizer.zero_grad() # Initialize gradients
loss.backward() # Backpropagation (compute new gradients)
optimizer.step() # Update parameters
```

03
Dataset, DataLoader



WHY DO WE NEED A DATA LOADER?

Imagine training on a huge dataset that is tens or hundreds of gigabytes in size. It is impossible to fit all of this data into memory at once. Even if it were possible, it would be very inefficient to compute and update the gradient using the entire data at once (batch gradient descent).

We need the following features:

- Divide the entire data into **small batches (mini-batches)** and process them
- **Randomly shuffle the data every epoch** during training to prevent the model from overfitting to the order of the data
- **Process the data in parallel** using multiple CPU cores to minimize the time the GPU spends waiting for the data to load

The **Dataset and DataLoader are tools** that elegantly solve all of these complex and tedious tasks.

CREATE A DATA STATEMENT

Dataset:

Dataset is an abstract class that literally acts as a 'specification' for a dataset. If we inherit the Dataset class and create our own dataset, we only need to implement three methods.

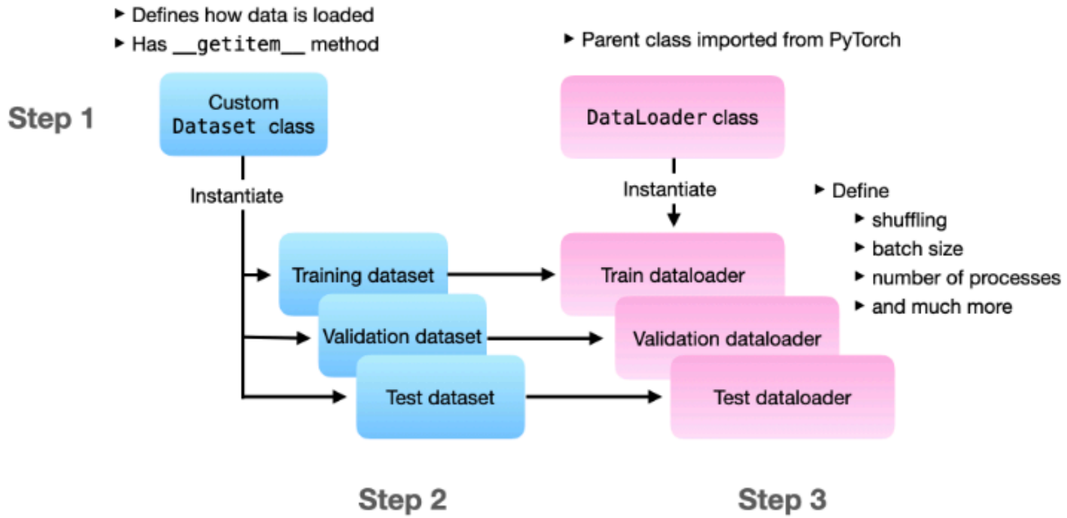
- __init__(self, ...): Sets the information necessary to initialize the dataset, such as the path where the data is stored, the file name, etc. This is not where all the data is loaded, but where we prepare meta information such as the location of the data.
- __len__(self): Returns the total number of data in the dataset. This method is executed when len(my_dataset) is called.
- __getitem__(self, idx): When an index called idx is received, it returns the corresponding data sample (e.g. image and label). This method is executed when my_dataset[i] is called.

AUTOMATED DATA FEEDER

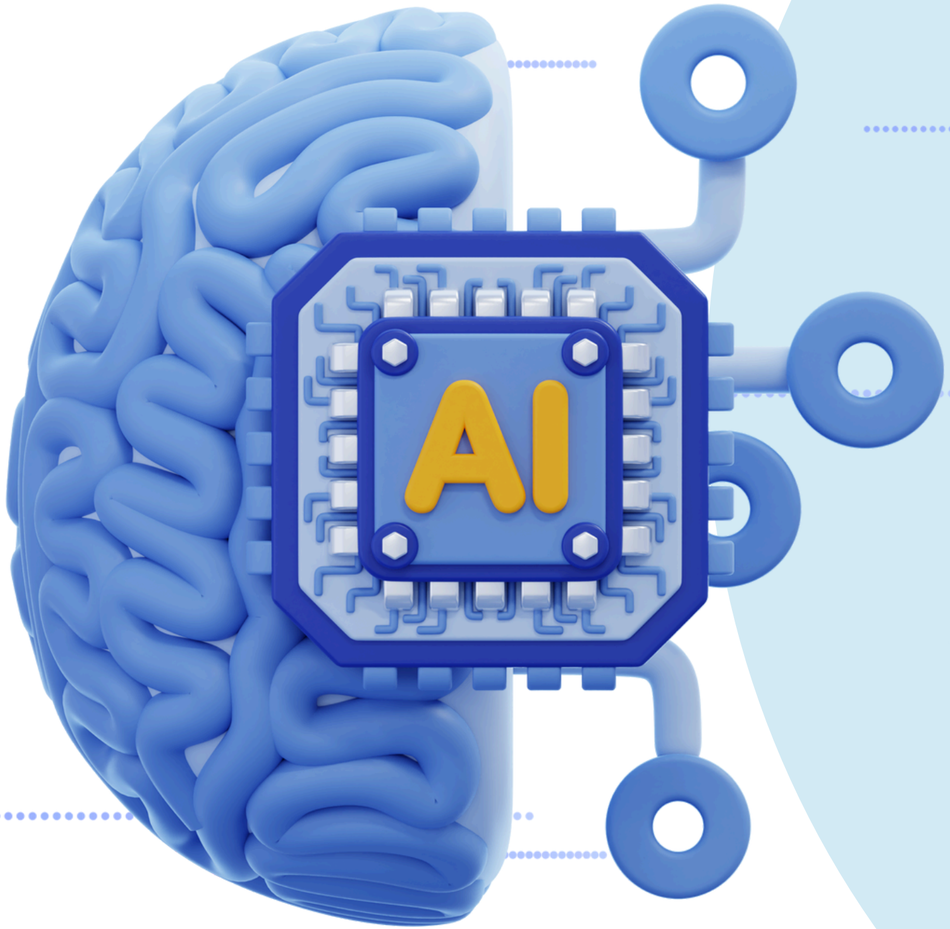
DataLoader:

DataLoader is a powerful iterator that takes a Dataset object as input and bundles and supplies data in the way we want. The main arguments used when creating a DataLoader are as follows:

- dataset: Dataset object to supply data to.
- batch_size: The number of data to bundle and return at once (mini-batch size).
- shuffle: If set to True, the data will be randomly shuffled at each epoch. It is usually set to True during training, and set to False during evaluation for consistent results.
- num_workers: The number of CPU processes to use when loading data. Setting a value greater than 0 can speed up data loading. (Caution may be required in Colab environments.)



04
Practice



1. CREATE VIRTUAL DATA

First, we create some fictitious data with a simple linear relationship to use for practice.

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader

# Create data
# Create data with a relationship similar to y = 2x + 1
X = torch.randn(100, 1) * 10 # 100 samples, 1 feature
noise = torch.randn(100, 1)
y = 2 * X + 1 + noise

print("X shape:", X.shape)
print("y shape:", y.shape)
```

2. IMPLEMENTING A CUSTOM DATASET CLASS

We define a CustomDataset class that inherits torch.utils.data.Dataset and processes the X and y data we created.

```
class CustomDataset(Dataset):
def __init__(self, X, y):
self.X = X
self.y = y

def __len__(self):
# Return the total length of the dataset
return len(self.X)

def __getitem__(self, idx):
# Return the data sample (X, y) corresponding to idx as a tuple
sample = self.X[idx], self.y[idx]
return sample

# Create a Dataset object
dataset = CustomDataset(X, y)

# Test if it works
print("Total number of samples:", len(dataset))
first_sample = dataset[0]
print("First sample (X, y):", first_sample)
```

3. INITIALIZE MODEL, LOSS FUNCTION, OPTIMIZER, AND DATA LOADER

Now we prepare all the components needed for learning.

```
# 1. Model definition (review of morning session content)
class LinearRegressionModel(nn.Module):
def __init__(self):
super().__init__()
self.linear = nn.Linear(in_features=1, out_features=1)

def forward(self, x):
return self.linear(x)

# 2. Hyperparameter setting
learning_rate = 0.01
num_epochs = 100
batch_size = 10

# 3. DataLoader creation
# Wraps the Dataset to bundle and mix data in batch size units. data_loader =
DataLoader(dataset=dataset, batch_size=batch_size, shuffle=True)

# 4. Instantiate model, loss function, and optimizer
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = LinearRegressionModel().to(device)
loss_function = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

print("DataLoader, Model, Loss, Optimizer are ready.")
```

4. IMPLEMENT AND RUN THE FINAL LEARNING LOOP

We use double nested for loops to train in units of 'epochs' and 'batches'.

```
print("--- Training Start ---")
# Training the entire dataset by repeating num_epochs
for epoch in range(num_epochs):
# DataLoader extracts data in batch size for each iteration. for
X_batch, y_batch in data_loader:
# Move data to the same device as the model
X_batch = X_batch.to(device)
y_batch = y_batch.to(device)

# 1. Forward
predictions = model(X_batch)

# 2. Calculate loss
loss = loss_function(predictions, y_batch)

# 3. PyTorch standard 3-step learning
optimizer.zero_grad() # Initialize gradient
loss.backward() # Backpropagation
optimizer.step() # Update parameters
```

```
# Output intermediate results every 10 epochs
if (epoch + 1) % 10 == 0:
print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

print("--- Training Finished ---")

# Check the learned parameters (since y = 2x + 1, weight should be
close to 2 and bias should be close to 1)
# Since model.parameters() is a generator, convert it to a list and
check
trained_params = list(model.parameters())
trained_weight = trained_params[0].item()
trained_bias = trained_params[1].item()

print(f"Trained Weight: {trained_weight:.4f}")
print(f"Trained Bias: {trained_bias:.4f}")
```

SUMMARY

Summary

- Optimizer is an engine that automatically updates model parameters using the gradient calculated by loss.backward().
- Torch.optim.SGD and torch.optim.Adam are representative optimizers.
- PyTorch's standard training loop consists of three steps: optimizer.zero_grad() → loss.backward() → optimizer.step().
- Dataset is a class that defines the specification of the dataset (total length, specific sample access method).
- DataLoader is a convenient tool that receives a Dataset and automates data feeding, such as mini-batch configuration and data shuffling.
- Combining these elements allows you to build a standard and scalable training pipeline that can be applied to any model.

Glossary of terms:

- Optimizer: An object that updates parameters to minimize the model's loss using various algorithms based on gradient descent.
- SGD (Stochastic Gradient Descent): A basic optimization algorithm that updates parameters using mini-batches.
- Adam (Adaptive Moment Estimation): An 'adaptive' optimization algorithm that applies different learning rates to each parameter, and is currently widely used.
- Epoch: A state in which learning is completed using the entire training dataset at once.
- Batch Size: The number of data samples used for one parameter update.
- Dataset: A Pytorch class that defines the total number of datasets (__len__) and how to get data at a specific index (__getitem__).
- DataLoader: An iterator that wraps a Dataset and provides batch unit supply, shuffling, parallel loading, etc.