IT Talent Training Course

Aug. 2025.

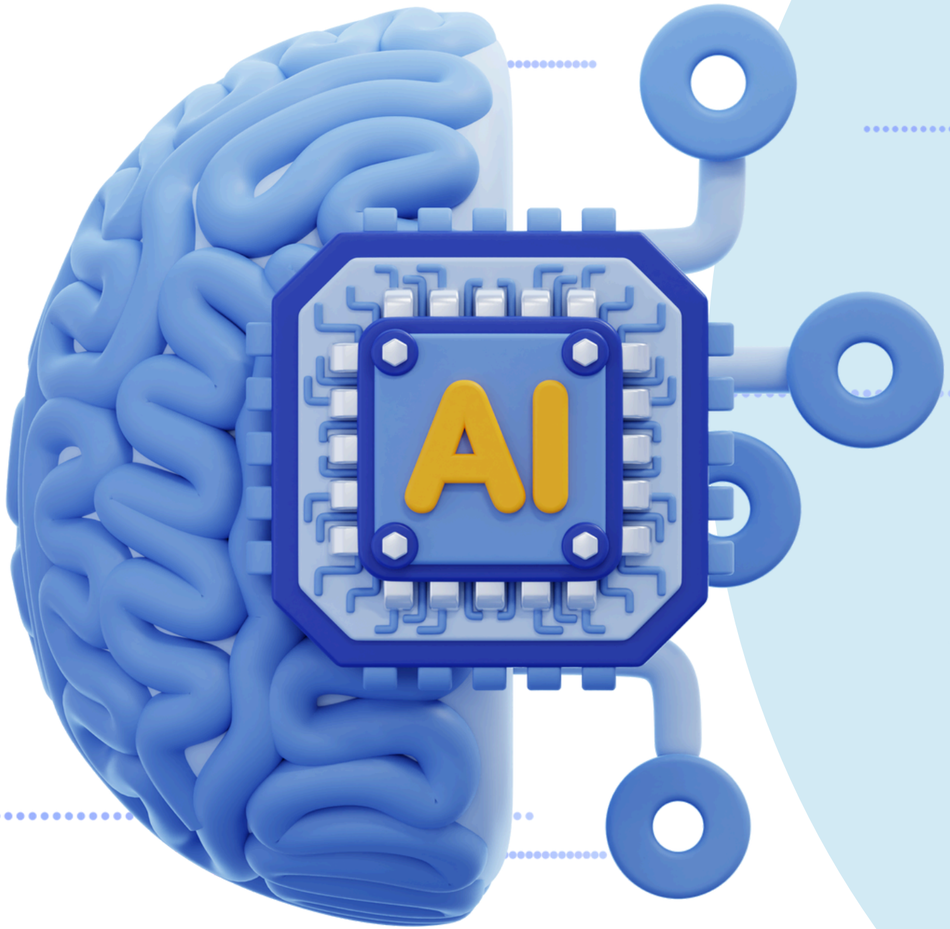
A.I. PROGRAMMING WITH PYTORCH

Instructor :
Daesung Kim

6th Day - Part 01

INDEX

01
SEQUENTIAL DATA



LIMITATIONS OF CNN AND THE NEED FOR NEW MODELS - 1

- **Neural Network Basic Structure:** Concept of Input Layer, Hidden Layer, Output Layer
- **Fully Connected Neural Network (FCN):** Structure in which neurons in each layer are connected to all neurons in the next layer
- **CNN (Convolutional Neural Network):** Convolution and pooling operations used to extract spatial features of images

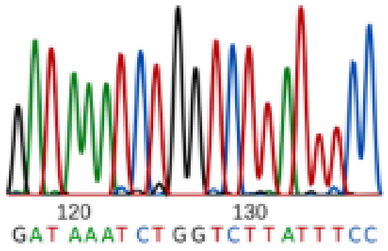
The CNN we have covered so far has learned the 'spatial relationship' between each pixel of an image and its surrounding pixels. But let's think about the following data.

- **Text:** In the sentence "I ate breakfast today," the order of words is very important. "I ate breakfast today" does not make sense.
- **Voice:** Someone's speech is a sound wave signal that changes continuously over time.
- **Stock data:** Yesterday's stock price and today's stock price are deeply related.

In this way, data in which each element of the data is not independent and influences each other according to temporal or logical order is called **Sequential Data**.

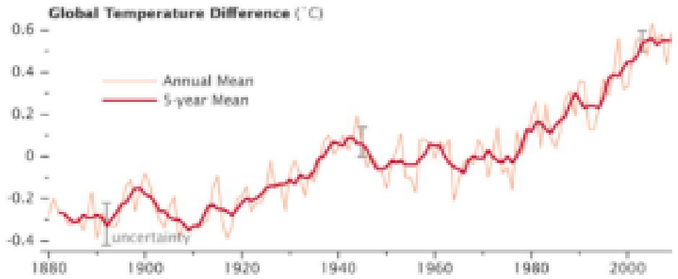
LIMITATIONS OF CNN AND THE NEED FOR NEW MODELS - 2

Existing models such as CNN and FCN do not properly consider the order information of the input data. For example, if you put a sentence into FCN, the order information of each word disappears, and even if you put it into CNN, it only identifies the relationship with limited surrounding words, making it difficult to understand the long-term context of the entire sentence. At this point, the need for a new architecture specialized in sequential data processing arose, and the answer is **RNN (Recurrent Neural Network)**.



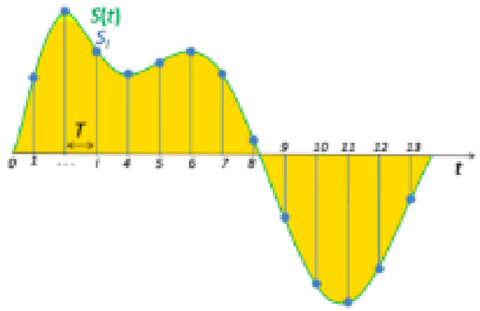
DNA 염기 서열

(Sequential Data)



세계 기온 변화

(Temporal Sequence)

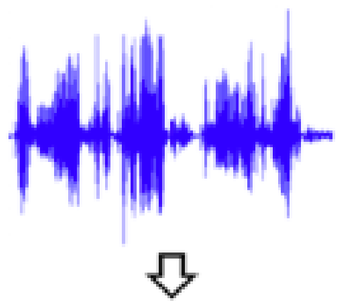


샘플링된 소리 신호

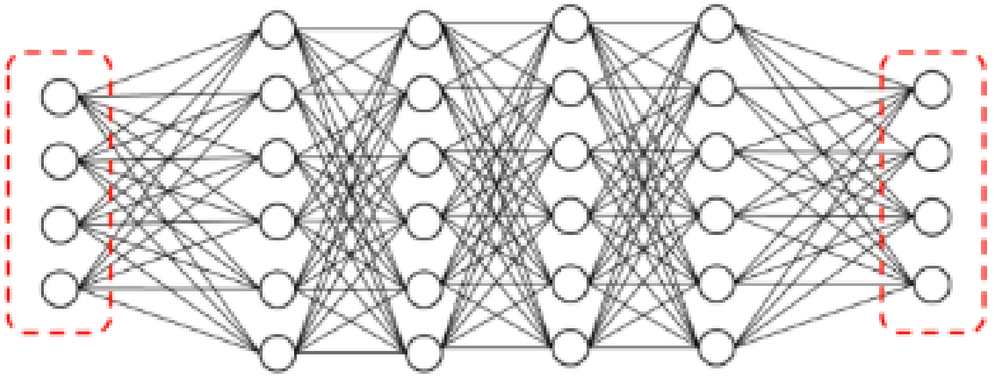
(Time Series)

LIMITATIONS OF CNN AND THE NEED FOR NEW MODELS - 3

Sequential data is data that has meaning in its order and whose meaning is lost when the order changes. The reason for using recurrent neural networks is to receive input as sequential data or to output sequential data. If it is a Time Series with a certain time difference, it is considered as a Step, unlike Temporal Sequence where the x-axis represents a specific time.



Fixed-Length Vector?



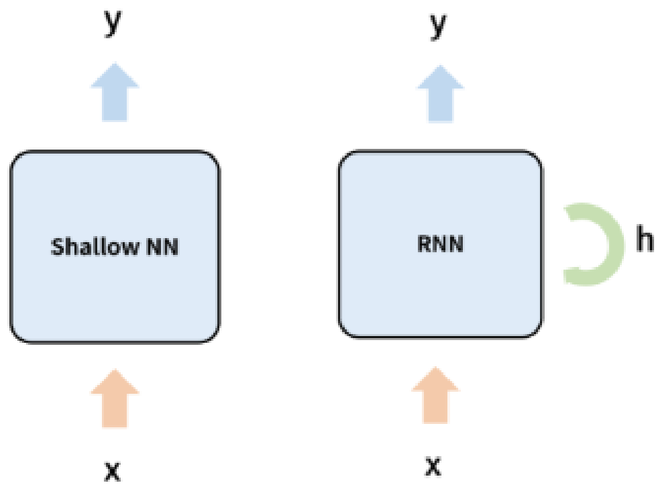
One-Hot Vector?

The number of possible sentences that can be printed is infinite.

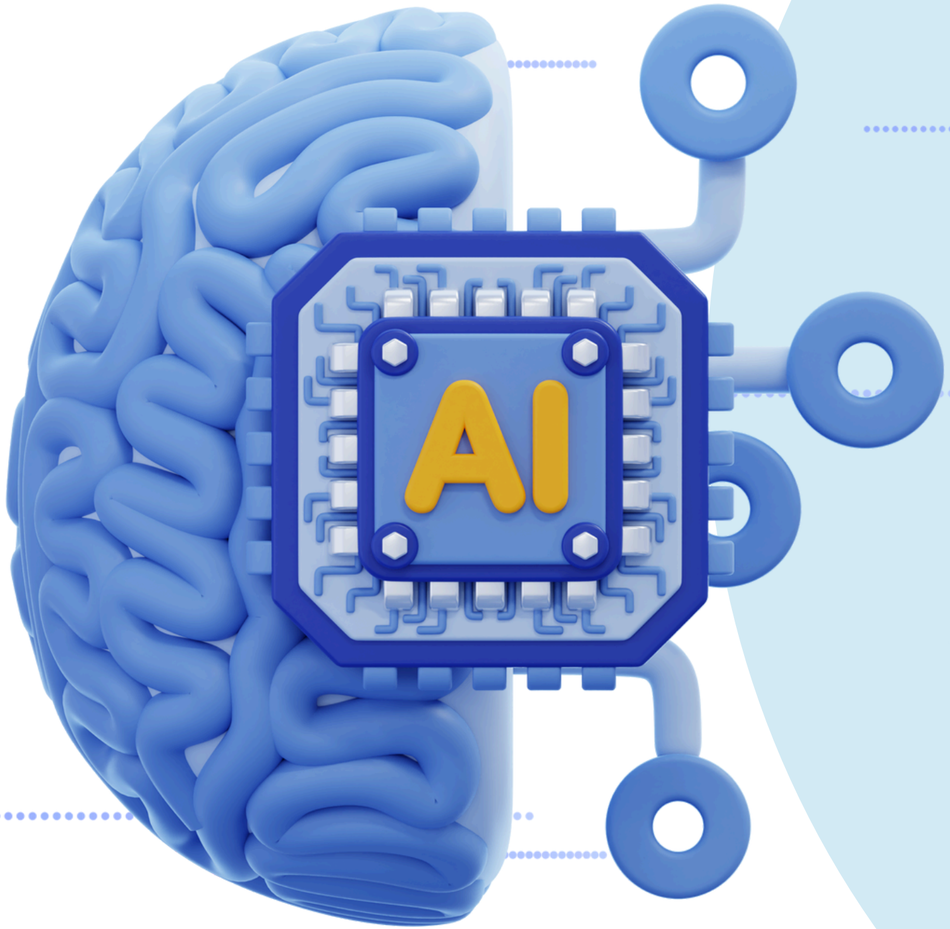
The length of the input voice is different each time.

LIMITATIONS OF CNN AND THE NEED FOR NEW MODELS - 4

In order to process sequential data, you must be able to remember the contents of the previously input data. A system that does not store previous input data is called a memoryless system. Unlike the previous network structure, RNN uses the n-1th step hidden layer in the input layer by concatenating it with the nth data. By doing so, **it is affected by all previous inputs**.



02
RECURRENT NEURAL NETWORK

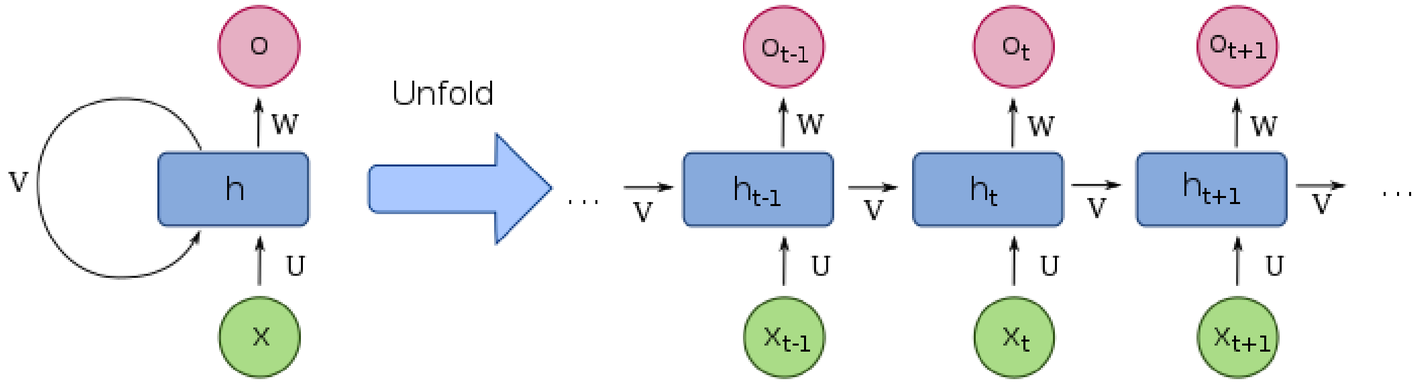


NEURAL NETWORKS WITH MEMORY - 1

- **Sequential data:** The concept of data where temporal order is important
- **Hidden Layer:** The middle layer of a neural network that processes input data and extracts features
- **Activation Function:** (e.g. tanh, ReLU) A function that adds nonlinearity to a neural network

1. Core idea of RNN: cyclic structure

The core idea of RNN(Recurrent Neural Network) is in its structure that is 'circular' as its name suggests. When processing data of the current time step (t), it utilizes the result (memory) processed in the previous time step (t−1).



NEURAL NETWORKS WITH MEMORY - 2

2. How RNN works

What operations specifically occur inside an RNN? The hidden state h_t at time step t is calculated using the following formula.

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$$

- x_t : Input vector at time step t
- h_{t-1} : Hidden state vector at previous time step t−1
- h_t : Hidden state vector at current time step t
- W_x : Weight matrix multiplied by input x_t
- W_h : Weight matrix multiplied by previous hidden state h_{t-1}
- b : Bias vector
- tanh : Hyperbolic tangent activation function. It compresses the result between -1 and 1.

The important thing here is that the same weight matrices W_x, W_h and bias b are shared across all timesteps. That is, the RNN learns to combine past information with current information by applying the 'same rules' over time.

The final output y_t can be computed using the hidden state h_t (the activation function may vary depending on the problem).

$$y_t = W_y h_t + b_y$$

NEURAL NETWORKS WITH MEMORY - 3

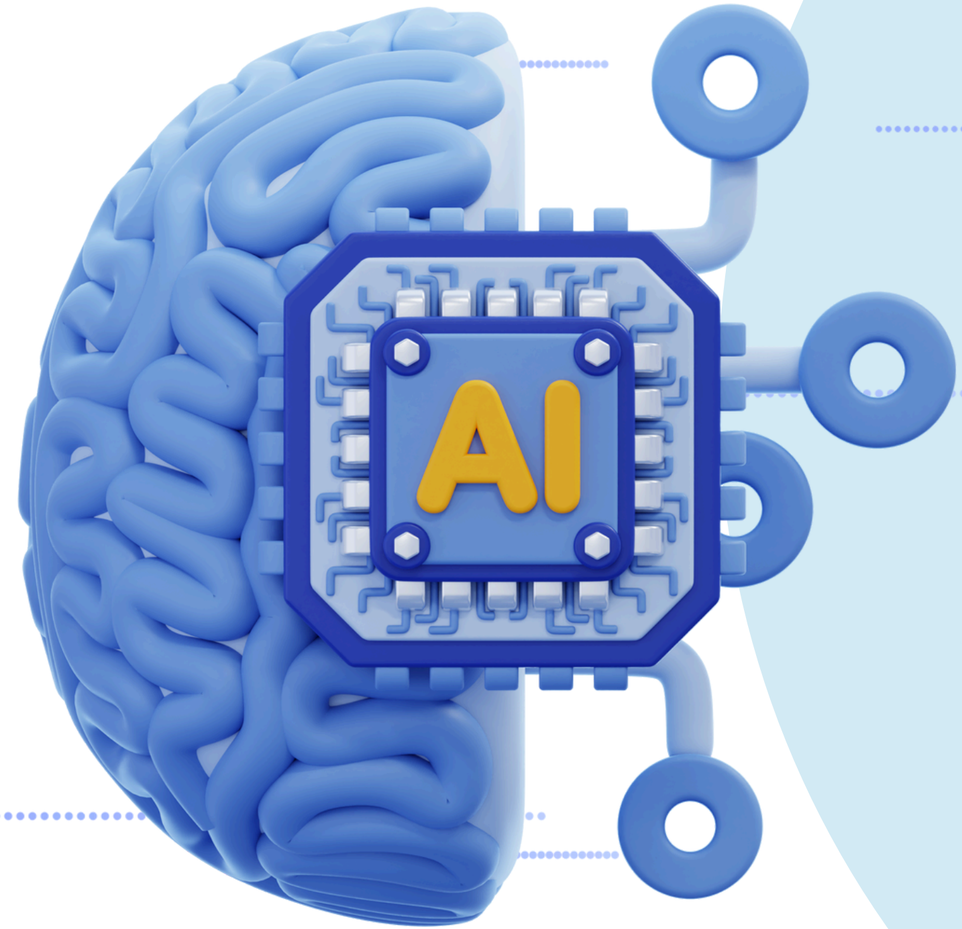
3. Long-Term Dependency Problem

RNNs can theoretically summarize all information from the beginning of a sequence to the end. However, in reality, as the length of the sequence increases, important information from the beginning is gradually diluted or distorted as it is transmitted backwards, which causes the **long-term dependency problem**.

For example, in the sentence "I was born in France and spent my childhood there. ... (omission) ... so I speak ___ fluently," the word "France" that fills the blank appears at the beginning of the sentence. It is very difficult for RNNs to successfully transmit this long-distance information to the end.

The fundamental cause of this problem occurs during the backpropagation process. As time goes by, the gradient continues to multiply, becoming too small to approach 0 (vanishing gradient) or too large to diverge (exploding gradient). In particular, the problem of **vanishing gradients is more frequent**, which prevents the model from learning anything from distant information.

03
LONG SHORT-TERM MEMORY



INNOVATION FOR LONG-TERM MEMORY

- **Structure of RNN and Long-Term Dependency Problem:** Concept of Vanishing/Exploding Gradient
- **Sigmoid function:** Outputs the result value between 0 and 1 and acts as a 'gate'
- **Element-wise product of vectors (Hadamard product):** Operation of multiplying the elements in the same position in two matrices/vectors of the same size

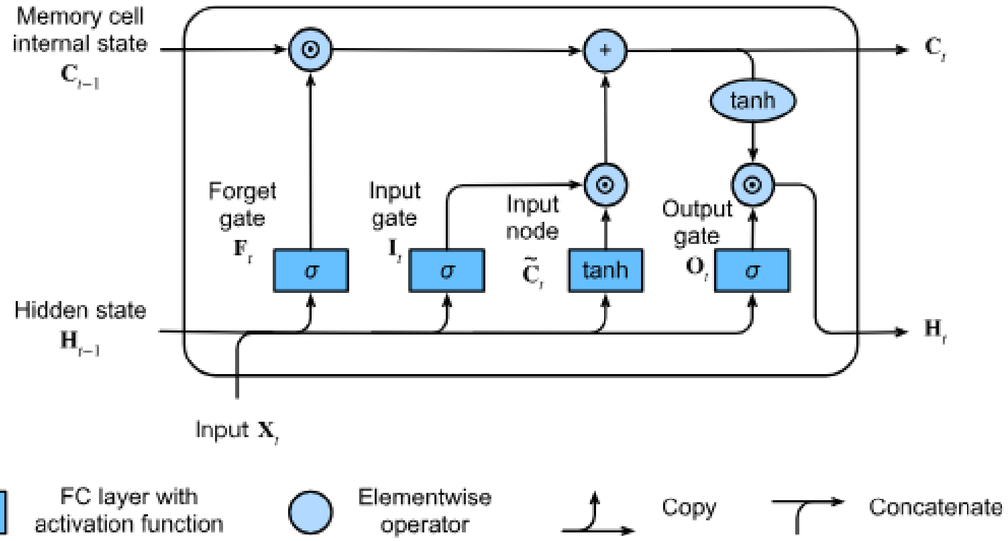
1. Core idea of LSTM: Cell state and gate

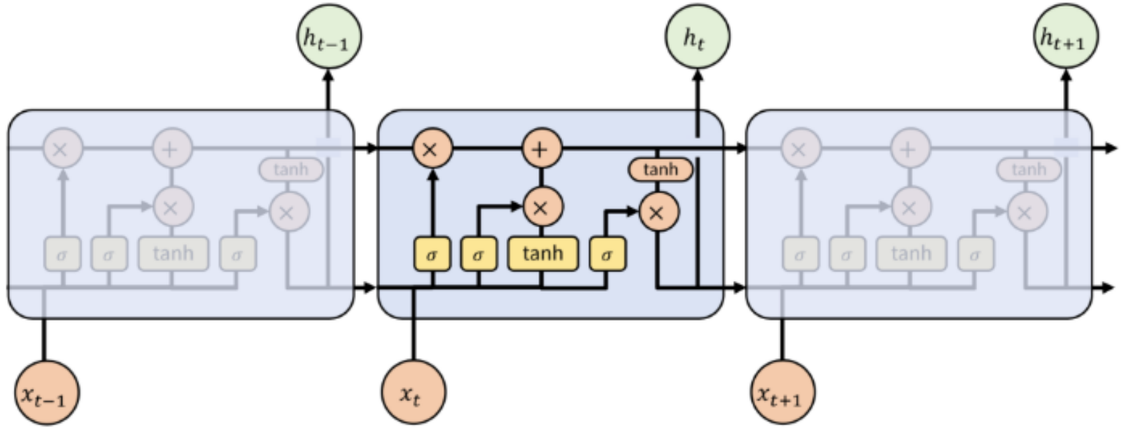
LSTM (Long Short-Term Memory) was proposed in 1997 to solve the long-term dependency problem. LSTM is a special type of RNN that is designed to effectively remember information for much longer periods of time than the basic RNN structure.

- **Cell State:** In addition to the hidden state of the existing RNN, we added a separate information path called the 'cell state'. This cell state is like an 'information highway' that penetrates the entire network and allows information to flow for a long time without major changes. It can be likened to a 'conveyor belt'.
- **Gates:** We created three 'gates' that precisely control the addition or removal of information from this information highway (cell state). These gates consist of a sigmoid function and an element-wise multiplication operation, and control the flow of information with values between 0 (blocking information) and 1 (passing information).

INNOVATION FOR LONG-TERM MEMORY

2. LSTM structure





1) Cell State (Cell State, C_t)

This is the most essential part of LSTM. The horizontal line across the top of the picture is the cell state. It has a structure where information can flow without being altered, with only a few linear operations.

INNOVATION FOR LONG-TERM MEMORY

2) Forget Gate (Forget Gate, F_t)
This gate determines 'how much to forget' about past information. It receives the previous hidden state (h_{t-1}) and the current input (x_t) and passes it through a sigmoid function. The output value is between 0 and 1, and the closer it is to 1, the more the information is 'completely remembered', and the closer it is to 0, the more it is 'completely forgotten'.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

3) Input Gate (i_t, $\tilde{C}_t$)
Next, it decides 'what new information to store in the cell state'. This process is done in two steps.
First, the input gate decides which value to update (i_t) through the sigmoid function.
Second, it creates a vector of new candidate values ($\tilde{c}_t$) through the tanh activation function.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

We are ready to update the previous cell state (C_{t-1}). Forget f_t of previous information and add i_t of new candidate information.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

INNOVATION FOR LONG-TERM MEMORY

4) Output Gate (o_t)

Finally, we decide what to output. This output is based on the cell state, but it is a filtered version.

- First, the sigmoid gate decides which part of the cell state to output (o_t).
- Second, we pass the cell state (C_t) through the tanh function to scale it between -1 and 1.
- Finally, we multiply these two values to output the final hidden state (h_t). This h_t is passed to the next time step or used to predict the final outcome.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

Through this sophisticated gate mechanism, LSTMs have the ability to retain necessary information for a long time and forget unnecessary information without losing or exploding gradients, effectively solving the long-term dependency problem.

SUMMARY

Summary

- Sequential data: Data where order is important, such as text and time series data. It is difficult to process with existing FCNs and CNNs.
- RNN (Recurrent Neural Network): A model that utilizes the 'hidden state (memory)' of the previous time step for the current calculation through a circular structure. However, when the sequence becomes long, the 'long-term dependency problem (memory loss)' occurs.
- LSTM (Long Short-Term Memory): A model designed to solve the long-term dependency problem of RNNs. It enables long-term memory through the information highway called 'cell state' and 'forgetting, input, output gates' that precisely control the flow of information.

Glossary of terms:

- Sequential Data: Data where there is a sequential relationship between data elements.
- Hidden State (h_t): A vector that summarizes the information of each time step in RNN/LSTM. Contains past contextual information.
- Long-Term Dependency Problem: When the sequence becomes long, information from the beginning is not properly transmitted to the back. Vanishing Gradient is the main cause.
- Cell State (C_t): The core of LSTM. A 'memory highway' designed to preserve information for a long time.
- Gate: A mechanism that controls the flow of information to a value between 0 (block) and 1 (pass) using the sigmoid function. LSTM has three gates: forget, input, and output.

IT Talent Training Course

Aug. 2025.

A.I. PROGRAMMING WITH PYTORCH

Instructor :
Daesung Kim

6th Day - Part 02

INDEX

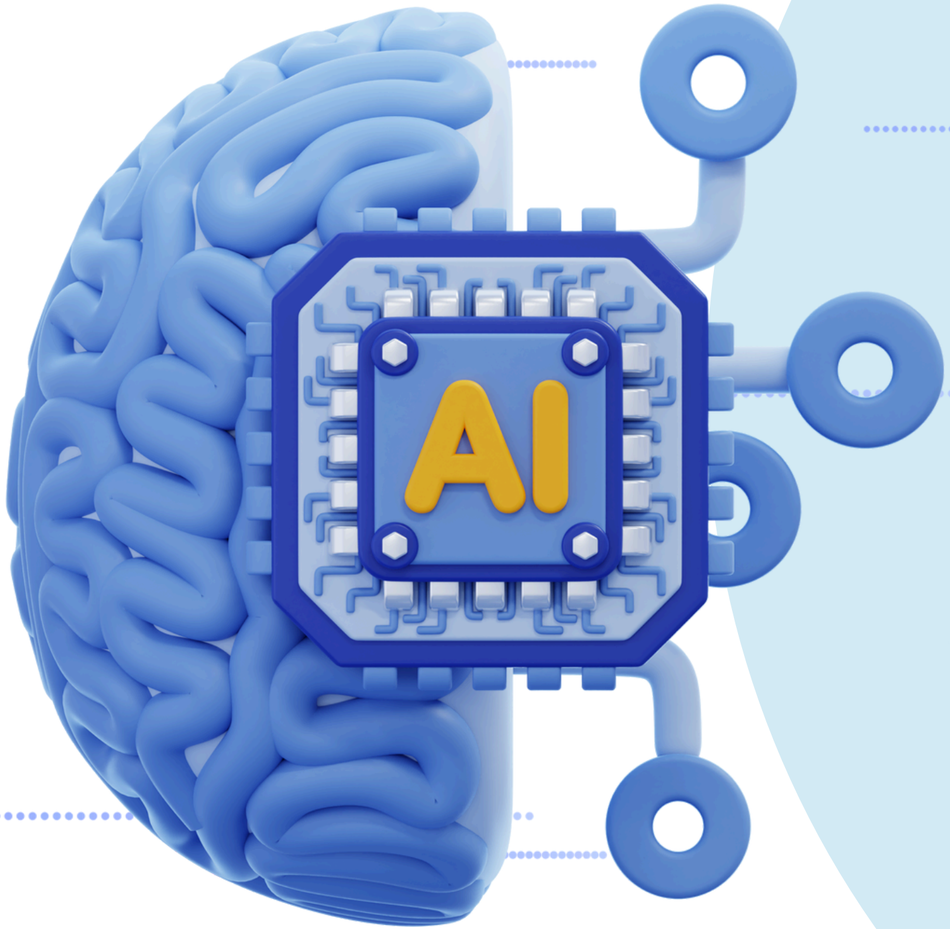
COURSE OBJECTIVES

- Understand how to process variable-length sequential data (text).
- You can convert text data into tensors (one-hot vectors) that neural network models can understand.
- You can build and train character-level RNN models directly using PyTorch.
- Experience the entire process of predicting new data using the trained model and evaluating its performance.
- Acquire the core principle of deep learning that 'the structure of the data determines the architecture of the model'.

- **Python Programming:** Ability to use basic data structures such as strings, lists, and dictionaries
- **PyTorch Basics:** Tensor creation and manipulation, model definition using nn.Module, understanding of basic learning loops (optimizer, loss)
- **RNN/LSTM Principles (Morning Lecture):** The core idea of RNN is to remember past information through hidden state
- **One-Hot Encoding:** A technique to express categorical data as a vector consisting of 0 and 1

01

Why character-level RNNs?



OVERVIEW

Yesterday, we built a CNN that 'sees' images. An important feature of CNNs is that the input data is all the same size (e.g. 32x32 pixels). However, not all data in the world is structured like this. In particular, let's think about text data that we use in our daily lives.

- "Kim", "Jackson", "Satoshi", "Schmidhuber"

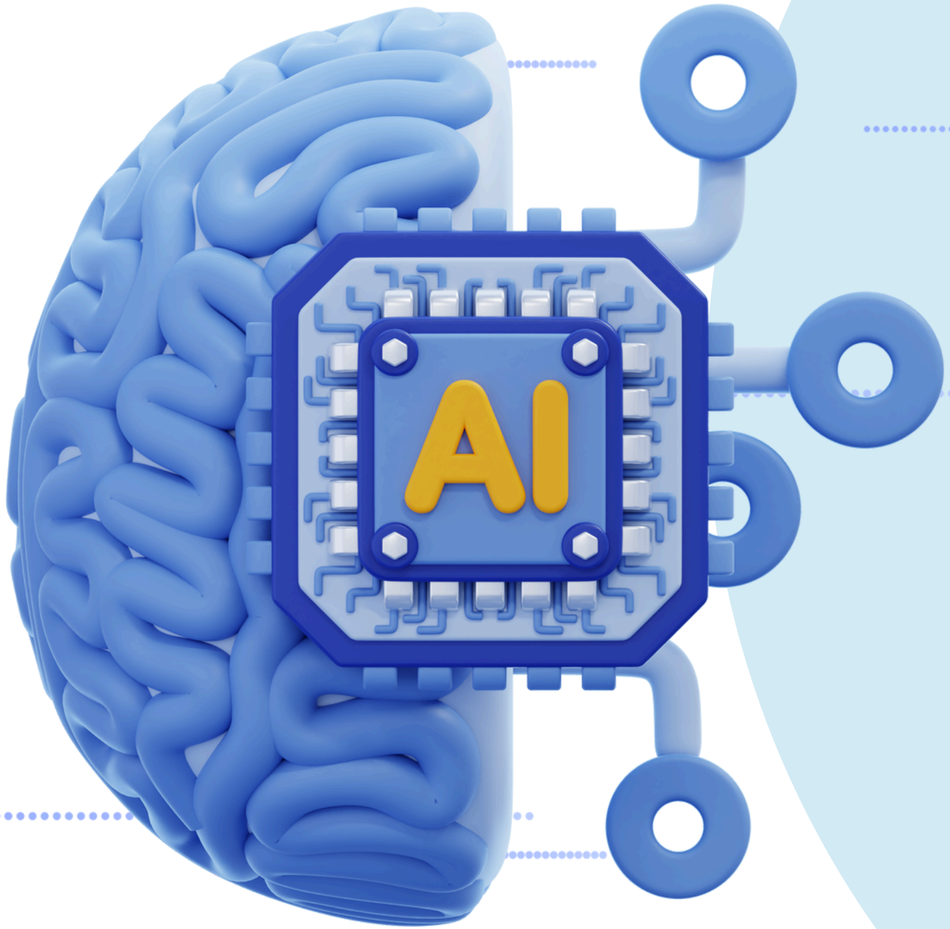
These names are all different lengths. Data that has temporal order or continuity and has variable lengths is called 'sequential data'.

Models like CNN have difficulty directly processing such variable-length data. This is where RNNs (recurrent neural networks) come in. **RNNs are specialized in understanding order and context** by remembering information from previous stages (hidden state) through their internal “circular” structure and processing it together with the input of the current stage.

PROJECT OVERVIEW: PREDICTING NATIONALITY BY NAME

- **Goal:** Given a person's surname, create a classification model that predicts which language the name is in among 18 languages.
- **Input:** A string of names (e.g. "Satoshi")
- **Output:** One of 18 nationalities (e.g. "Japanese")
- **Dataset:** Text files containing names in various languages (data provided by the PyTorch tutorial)
- **Key idea:** The assumption that the combination and order of letters in names reflects patterns in specific languages (e.g. if a name starts with 'Sch', it is likely to be German).

02
Data preparation and preprocessing



UNDERSTANDING AND LOADING THE DATASET STRUCTURE

Before training a deep learning model, the most important and time-consuming process is 'data preprocessing'. The computer cannot directly understand the string 'S-m-i-t-h'. We need to convert it into a tensor format, which is a number that the model can calculate.

This course is based on the official PyTorch "NLP From Scratch" tutorial.

First, let's download the data we'll be using for practice and take a look at its structure.

```
# Download and unzip data in Google Colab environment
!wget https://download.pytorch.org/tutorial/data.zip
!unzip data.zip

# Import required libraries
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import os
import unicodedata
import string

# Check data file path
# Glob is useful for retrieving a list of files using a file path pattern.
print(glob.glob('data/names/*.txt'))
```

As a result of execution, you can see that there are 18 text files under the data/names/ folder, such as Arabic.txt, Chinese.txt, and Czech.txt. Inside each file, the names of the corresponding language are listed one per line.

CONVERT TEXT TO TENSOR - 1

Step 1: Convert name (string) to ASCII code

There are many languages in the world (e.g. À, á, ä), and to handle them consistently, we convert Unicode characters to basic ASCII characters. This reduces the number of characters that the model has to handle.

```
all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

# Convert Unicode string to ASCII (e.g. Ślusàrski -> Slusarski)
def unicode_to_ascii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

print(f"Original: Ślusàrski => Converted: {unicode_to_ascii('Ślusàrski')}")
```

CONVERT TEXT TO TENSOR - 2

Step 2: Build all character sets and clean up the data

Now read all the text files and create a dictionary of the form {language: [names...]} and a list of all languages.

```
# A dictionary that creates a list of names for each language
category_lines = {}
# A list of all languages
all_categories = []

# A function that reads a file and returns one line at a time
def readLines(filename):
lines = open(filename, encoding='utf-8').read().strip().split('\n')
return [unicode_to_ascii(line) for line in lines]

for filename in glob.glob('data/names/*.txt'):
category = os.path.splitext(os.path.basename(filename))[0]
all_categories.append(category)
lines = readLines(filename)
category_lines[category] = lines

n_categories = len(all_categories)

print(f"Found {n_categories} languages: {all_categories}")
print(f"Italian Sample name: {category_lines['Italian'][:5]}")
```

CONVERT TEXT TO TENSOR - 3

Step 3: Representing letters as one-hot vectors

This is the most essential transformation. We map each letter to a unique integer index, and then create a one-hot vector from that index.

What is a one-hot vector?

It is a vector that has 1 in only one position and 0 in all other positions. For example, if the entire set of letters (all_letters) is 'abc' and we want to represent 'b':

- Index of 'a': 0, Index of 'b': 1, Index of 'c': 2
- One-hot vector of 'b': [0, 1, 0]

This representation method has the advantage of making categorical data (letters) into a numeric form that is easy for computers to process, and does not impose an artificial order or size relationship between each letter.

CONVERT TEXT TO TENSOR - 4

```
import torch

# Find the index of each letter in all_letters (e.g. "a" = 0)
def letter_to_index(letter):
return all_letters.find(letter)

# Convert a letter to a one-hot tensor of size <1 x n_letters>
def letter_to_tensor(letter):
tensor = torch.zeros(1, n_letters)
tensor[0][letter_to_index(letter)] = 1
return tensor

# Convert a name (line) to a tensor of size <line_length x 1 x n_letters>
def line_to_tensor(line):
tensor = torch.zeros(len(line), 1, n_letters)
for i, letter in enumerate(line):
tensor[i][0][letter_to_index(letter)] = 1
return tensor

# Test
print("J One-hot tensor of :")
print(letter_to_tensor('J'))
print("\nTensor size of entire Jackson :")
print(line_to_tensor('Jackson').size())
```

Note that the tensor here has the size <line_length x 1 x n_letters> .

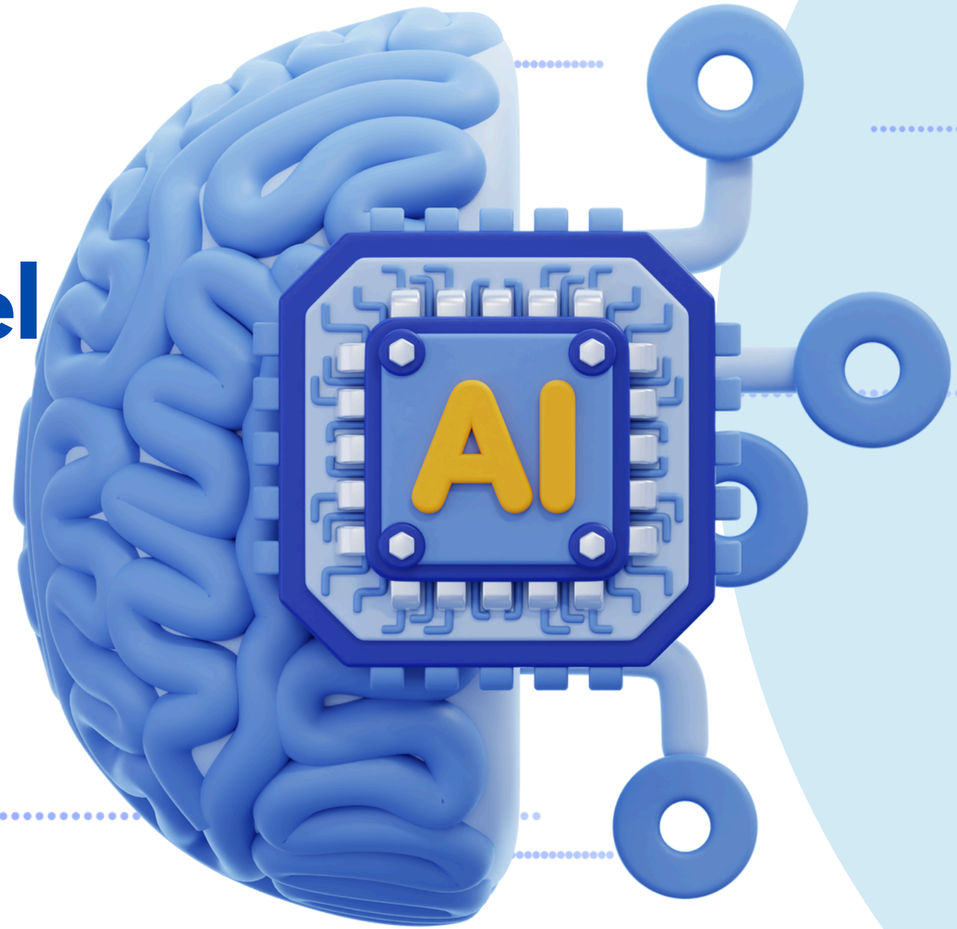
- **line_length:** length of the name (variable)
- **1:** batch size (process one data at a time)
- **n_letters:** size of the entire character set (dimension of the one-hot vector)

FUNCTION TO GENERATE TRAINING DATA PAIRS

At each training step, we will randomly select a (language, name) pair and feed it to the model. We will create a helper function for this.

```
import random

def random_training_example():
    # 1. Randomly select a category
    category = random.choice(all_categories)
    # 2. Randomly select a line from the list of names in that language
    line = random.choice(category_lines[category])
    # 3. Convert categories and lines to tensors
    category_tensor = torch.tensor([all_categories.index(category)], dtype=torch.long)
    line_tensor = line_to_tensor(line)
    return category, line, category_tensor, line_tensor

# Test 10
for i in range(10):
    category, line, category_tensor, line_tensor = random_training_example()
    print(f'category = {category} / line = {line}')
```

03
Building a character-level RNN model



VISUALIZING MODEL ARCHITECTURE

Now that we have our data ready, let’s define an RNN model to process this data using PyTorch’s nn.Module.

Our model takes input sequentially, one character at a time. As it processes each character, the RNN outputs two things:

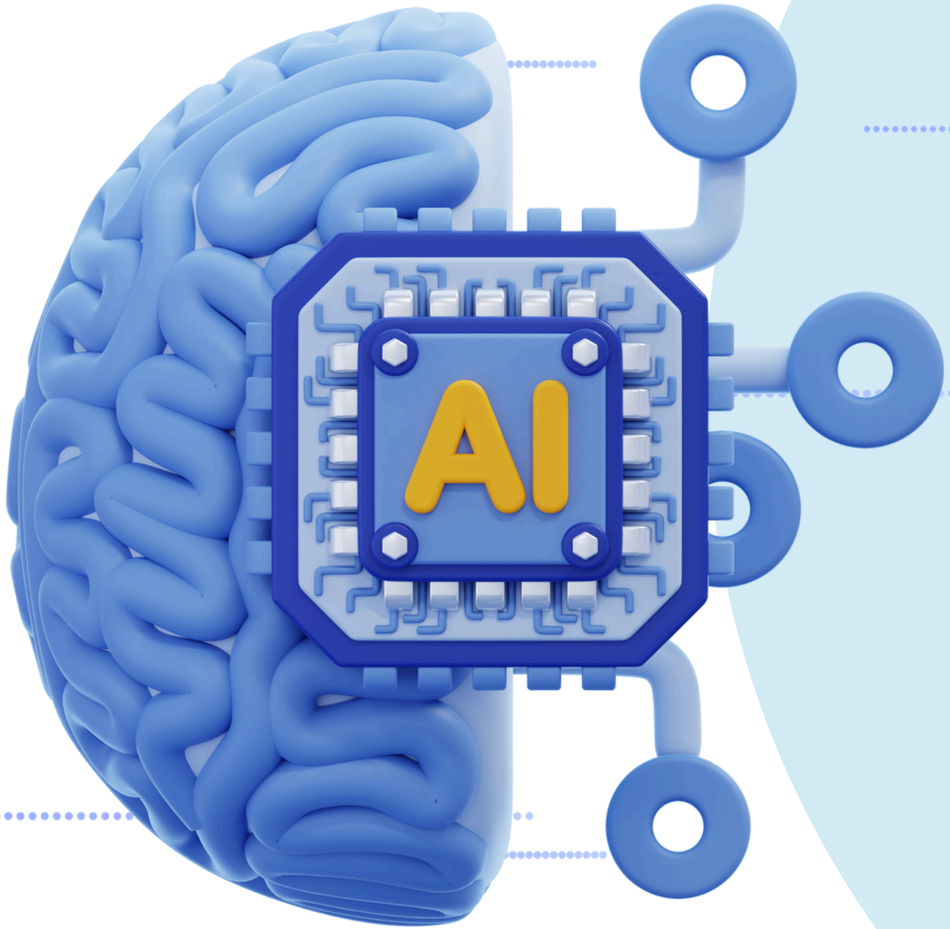
- **output:** the prediction for the current timestep.
- **next_hidden:** the 'memory' (hidden state) to use when processing the next character.

DEFINING MODEL CLASSES USING NN.MODULE

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size

        # Merge the input (one-hot vector of characters) and the hidden state and pass it to the next step.
        # This is how basic RNNs work.
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        # LogSoftmax is useful for computing probabilities in classification problems.
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input_tensor, hidden_tensor):
        # Merge the input and hidden tensors into one.
        combined = torch.cat((input_tensor, hidden_tensor), 1)

        # Pass the combined tensor through each linear layer.
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)

        return output, hidden

    def initHidden(self):
        # Initialize the first hidden state to 0 at the beginning of training.
        return torch.zeros(1, self.hidden_size)

# Create a model instance
n_hidden = 128 # The size of the hidden state is a hyperparameter.
rnn = RNN(n_letters, n_hidden, n_categories)
```

- **input_size:** n_letters, i.e. the size of the one-hot vector representing a single letter.
- **hidden_size:** the size of the 'memory'. The larger this value, the more complex the model can learn, but the greater the risk of overfitting.
- **output_size:** n_categories, i.e. the number of languages to predict.

04
Model Training



LOSS FUNCTION AND OPTIMIZER

Now that we have our model and data ready, it's time to train our model by setting up the learning loop we learned on Day 4.

- Loss Function: A function that measures how wrong the model's predictions are. Since the last layer of our model is LogSoftmax, we use the NLLLoss (Negative Log Likelihood Loss) that is paired with it.
- Optimizer: An algorithm that updates the model's weights (parameters) in a way that reduces the loss. Here, we use SGD (Stochastic Gradient Descent).

```
learning_rate = 0.005 # Learning rate is also an important hyperparameter.

# Defining the loss function
criterion = nn.NLLLoss()
# Defining the optimizer
optimizer = torch.optim.SGD(rnn.parameters(), lr=learning_rate)
```

LEARNING HELPER FUNCTIONS

To make the overall training code cleaner, we create a function that performs a single training step.

```
def train(category_tensor, line_tensor):
    # 1. Initialize hidden state
    hidden = rnn.initHidden()

    # 2. Initialize gradient
    optimizer.zero_grad()

    # 3. Input each letter of the name in order to the model
    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    # 4. Calculate loss with the final output (output) and the actual answer (category_tensor)
    loss = criterion(output, category_tensor)

    # 5. Calculate gradient through backpropagation
    loss.backward()

    # 6. Update parameters with optimizer
    optimizer.step()

    # Return loss and final prediction
    return output, loss.item()
```

IMPLEMENTING THE ENTIRE TRAINING LOOP

Now, put all the pieces together and create a full loop that repeats the training a specified number of times.

```
import time
import math

n_iters = 100000 # Total number of training sessions
print_every = 5000
plot_every = 1000

# List for recording loss values
current_loss = 0
all_losses = []

def time_since(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

start = time.time()

for iter in range(1, n_iters + 1):
    # 1. Get random training data
    category, line, category_tensor, line_tensor = random_training_example()
    # 2. Train the model
    output, loss = train(category_tensor, line_tensor)
    current_loss += loss

    # 3. Training results at regular intervals output of power
    if iter % print_every == 0:
        guess, guess_i = category_from_output(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100, time_since(start), loss, line, guess, correct))

    # 4. Store loss values at regular intervals (for visualization)
    if iter % plot_every == 0:
        all_losses.append(current_loss / plot_every)
        current_loss = 0
```

The actual learning may take a few minutes.

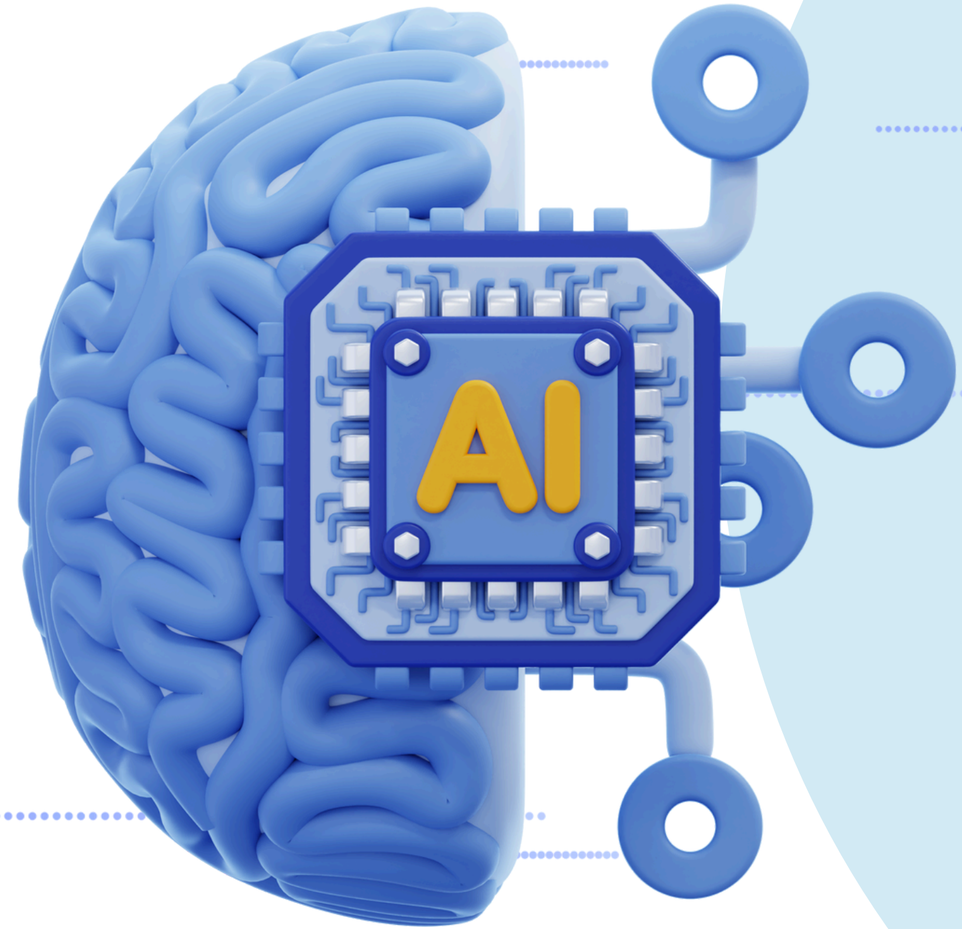
VISUALIZING THE LEARNING PROCESS: EVOLUTION OF LOSS

If the model is trained well, the loss value should gradually decrease. Let's draw a loss graph using matplotlib.

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
plt.title('Training Loss')
plt.xlabel('Iterations (x1000)')
plt.ylabel('Loss')
plt.show()
```

If the graph shows a steady downward trend, it is a sign that learning is progressing successfully.

05
Model Evaluation and Utilization



PERFORMANCE EVALUATION USING CONFUSION MATRIX

A confusion matrix is one of the most intuitive indicators of the performance of a classification model. The rows represent the actual answers, and the columns represent the model's predictions. The more values on the diagonal, the better the model is at guessing the correct answers.

```
# Disable gradient computation for evaluation.
with torch.no_grad():
    # Create a tensor of size n_categories x n_categories for the confusion matrix
    confusion = torch.zeros(n_categories, n_categories)
    n_confusion = 10000

    # Run predictions on many examples
    for i in range(n_confusion):
        category, line, category_tensor, line_tensor = random_training_example()
        output = evaluate(line_tensor) # The evaluate function is a version of the train function with only the training part removed.
        guess, guess_i = category_from_output(output)
        category_i = all_categories.index(category)
        confusion[category_i][guess_i] += 1

    # Normalize by dividing by the sum of each row (expressed as a ratio)
    for i in range(n_categories):
        confusion[i] = confusion[i] / confusion[i].sum()

    # Draw confusion matrix with Matplotlib
    fig = plt.figure(figsize=(8,8))
    ax = fig.add_subplot(111)
    cax = ax.matshow(confusion.numpy())
    fig.colorbar(cax)

    ax.set_xticklabels(['] + all_categories, rotation=90)
    ax.set_yticklabels(['] + all_categories)

    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()
```

If the diagonal of the confusion matrix is brightly lit, it means that our model is fairly accurate at classifying nationalities for most languages. If certain rows or columns are particularly dark, it means that the model is confusing those languages (e.g. Chinese, Korean, Vietnamese).

PREDICT NATIONALITY WITH A NEW NAME

Let's test how the model predicts by entering our own names.

```
def predict(input_line, n_predictions=3):
    print(f'\n> {input_line}')
    with torch.no_grad():
        output = evaluate(line_to_tensor(input_line))

        # Get top N prediction results
        topv, topi = output.topk(n_predictions, 1, True)
        predictions = []

        for i in range(n_predictions):
            value = topv[0][i].item()
            category_index = topi[0][i].item()
            print('(%.2f) %s' % (value, all_categories[category_index]))
            predictions.append([value, all_categories[category_index]])

# Example for Laos students
predict('Sonesingdara') # Example of Laos names
# Other examples
predict('Jackson')
predict('Satoshi')
predict('Kim')
```

By running this code, you can see for yourself how the trained RNN makes inferences based on sequential character patterns. This experience is an integrated course that clearly shows how deep learning theory is translated into concrete code to solve real-world problems.

SUMMARY

Summary

- In this session, we built a character-level RNN model to process text, which is sequential data of different lengths.
- The most important process was data preprocessing, which converts the string 'name' into a one-hot vector tensor that the model can understand.
- We inherited nn.Module and defined our own RNN class, and confirmed the working principle of the model that sequentially processes inputs one character at a time and updates the hidden state.
- We successfully trained the model by applying the learning loop (loss function, optimizer, backpropagation) learned on Day 4, and monitored the learning process through the loss graph.
- Finally, we evaluated the overall performance of the model through the confusion matrix, and completed a practical example of predicting nationality by entering a new name. With this, we have acquired a powerful tool for handling sequential data.

Glossary of terms:

- Sequential Data: Data where temporal order or continuity is meaningful. (Example: text, time series data, speech)
- Character-level RNN: RNN model that receives and processes individual characters as sequential input, not words.
- One-Hot Vector: A vector where only one element of the vector is 1 and the rest are all 0. Widely used to express categorical data.
- Hidden State: An internal vector that RNN uses to 'remember' information from the previous time step. It is passed on to the next time step.
- Confusion Matrix: A table that visualizes the performance of a classification model. It compares the actual class with the class predicted by the model to show how well each class is matched and which classes are confused.
- NLLLoss (Negative Log Likelihood Loss): A type of loss function used in classification problems, especially when the output layer of the model is LogSoftmax.