LLMs for text classification and generation

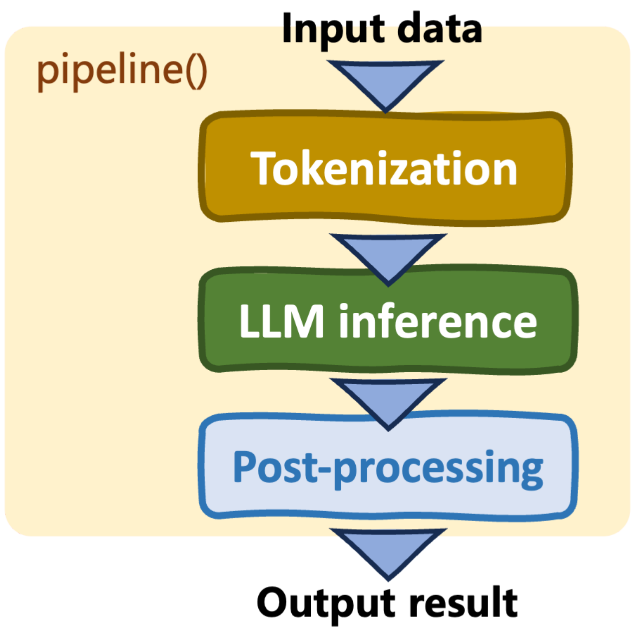
INTRODUCTION TO LLMS IN PYTHON

Iván Palomares Carrascosa, PhD
Senior Data Science & AI Manager

Loading a pre-trained LLM

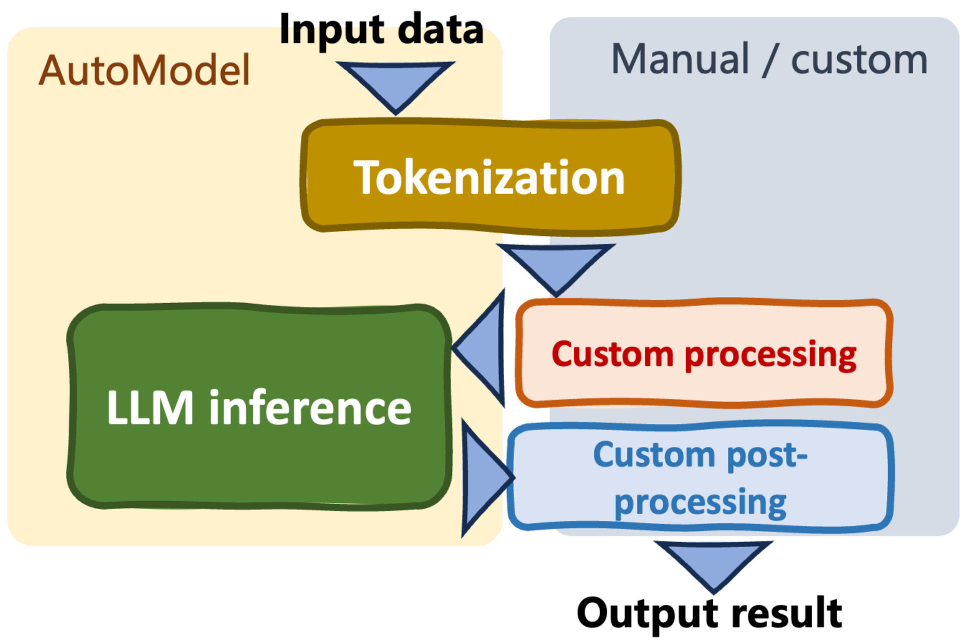
Pipelines: `pipeline()`

- Simple, high-level interface
- Automatic model and tokenizer selection
- More abstraction = less control
- Limited task flexibility



Auto classes (`AutoModel` **class)**

- Flexibility, control and customization
- Complexity: manual set-ups
- Support very diverse language tasks
- Enable model fine-tuning



The AutoModel and AutoTokenizer classes

```python
import torch.nn as nn
from transformers import AutoModel, AutoTokenizer

model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)

text = "I am an example sequence for text classification."

class SimpleClassifier(nn.Module):
    def __init__(self, input_size, num_classes):
        super(SimpleClassifier, self).__init__()
        self.fc = nn.Linear(input_size, num_classes)
    def forward(self, x):
        return self.fc(x)
```

`from_pretrained()`

- Load pre-trained model weights and tokenizer as specified in `model_name`
- `model_name` : **model checkpoint:**
  - A unique model version with specific architecture, configuration, and weights
- `AutoModel` does not provide task-specific head

The AutoModel and AutoTokenizer classes

```
inputs = tokenizer(
  text, return_tensors="pt", padding=True,
  truncation=True, max_length=64)
outputs = model(**inputs)
pooled_output = outputs.pooler_output
print("Hidden states size: ", outputs.last_hidden_state.shape)
print("Pooled output size: ", pooled_output.shape)

classifier_head = SimpleClassifier(
  pooled_output.size(-1), num_classes=2)
logits = classifier_head(pooled_output)
probs = torch.softmax(logits, dim=1)
print("Predicted Class Probabilities:", probs)
```

```
Hidden states size:  torch.Size([1, 11, 768])
Pooled output size:  torch.Size([1, 768])
```

```
Predicted Class Probabilities:
tensor([[0.4334, 0.5666]], grad_fn=<SoftmaxBackward0>)
```

- **Tokenize** inputs
- Get model's **hidden states** in `outputs`
  - `pooler_output` : high-level, aggregated representation of the sequence
  - `last_hidden_states` : raw unaggregated hidden states
  - Forward pass through **classification head** to obtain **class probabilities**

Auto class for text classification

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer

model_name = "nlptown/bert-base-multilingual-uncased-sentiment"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(
  model_name)

text = "The quality of the product was just okay."
inputs = tokenizer(text, return_tensors="pt")
outputs = model(**inputs)
logits = outputs.logits

predicted_class = torch.argmax(logits, dim=1).item()
print(f"Predicted class index: {predicted_class + 1} star.")
```

```
Predicted class index: 3 star.
```

`AutoModelForSequenceClassification` **class:**

- Provides pre-configured model with a classification head
- No need to manually add model head
- `outputs` already passed through head's linear layer
  - Access raw class logits and return "most likely" class

Auto class for text generation

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model_name = "gpt2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

prompt = "This is a simple example for text generation,"
inputs = tokenizer.encode(
  prompt, return_tensors="pt")
output = model.generate(inputs, max_length=26)

generated_text = tokenizer.decode(
  output[0], skip_special_tokens=True)
print("Generated Text:")
print(generated_text)
```

```
Generated Text:
This is a simple example for text generation, but it's also
a good way to get a feel for how the text is generated.
```

`AutoModelForCausalLM` **class:**

- Pre-configured model for causal (auto-regressive) language generation, e.g.: `"gpt2"`
- Model head for next-word prediction
- `generate()` takes prompt and generates up to `max_length` subsequent tokens
- **Raw outputs are decoded** before printing extended **prompt with generated text**

Exploring a dataset for text classification

```python
from datasets import load_dataset
from torch.utils.data import DataLoader

dataset = load_dataset("imdb")
train_data = dataset["train"]
dataloader = DataLoader(train_data, batch_size=2, shuffle=True)

for batch in dataloader:
    for i in range(len(batch["text"])):
        print(f"Example {i + 1}:")
        print("Text:", batch["text"][i])
        print("Label:", batch["label"][i])
```

```
Example 1:
Text: Much worse than the original. It was actually *painf(...)
Label: tensor(0)
Example 2:
Text: I have to agree with Cal-37 it's a great movie, spec(...)
Label: tensor(1)
```

- `load_dataset()` : loads a dataset from Hugging Face hub
  - **imdb:** review sentiment classification
- `DataLoader` **class:** simplifies iterating, batch processing and parallelism
  - Iterating through review-sentiment examples

Exploring a dataset for text generation

```python
from datasets import load_dataset

dataset = load_dataset("stanfordnlp/shp", "askculinary")
train_data = dataset["train"]
print(train_data[i])

for i in range(5):
    example = train_data[i]
    print(f"Example {i + 1}:")
    print("Title:", example["post_id"])
    print("Paragraph:", example["history"])
    print()
```

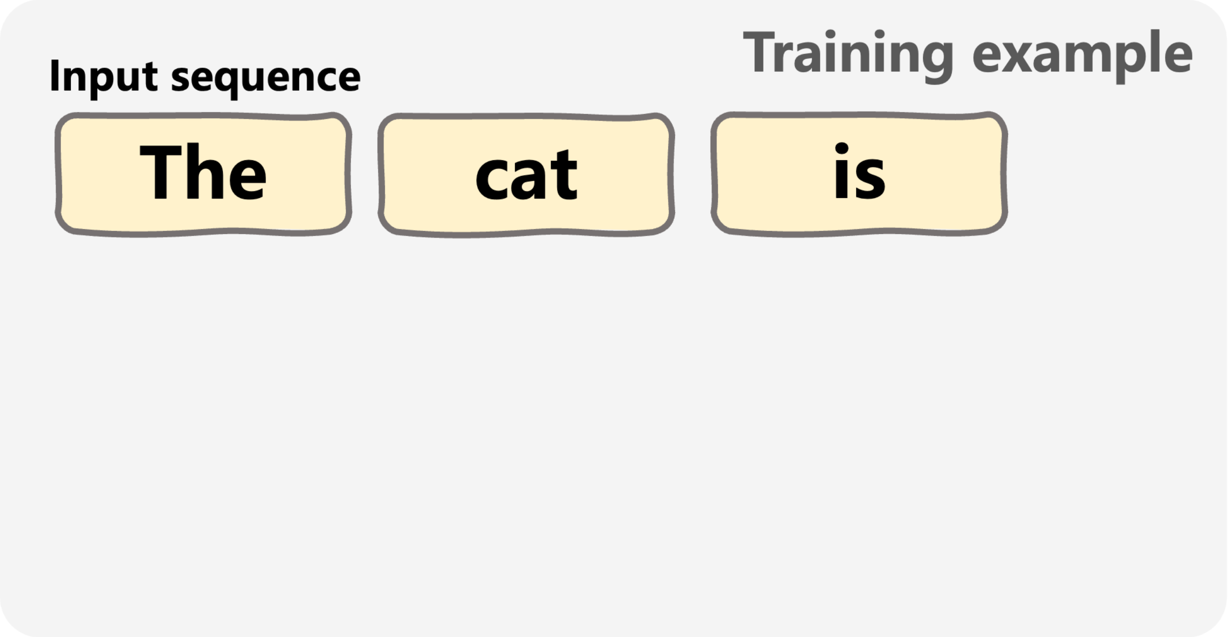
```
Example 1:
Title: himc90
Paragraph: In an interview right before receiving the 2013
 Nobel prize in physics, Peter Higgs stated that he (...)

Example 2 (...)
```

- Using a dataset from *standfordnlp* catalogue
  - Suitable for **text generation** and **generative QA**
- Display some text information in data instances

How text generation LLM training works

Input + target (labels) pairs

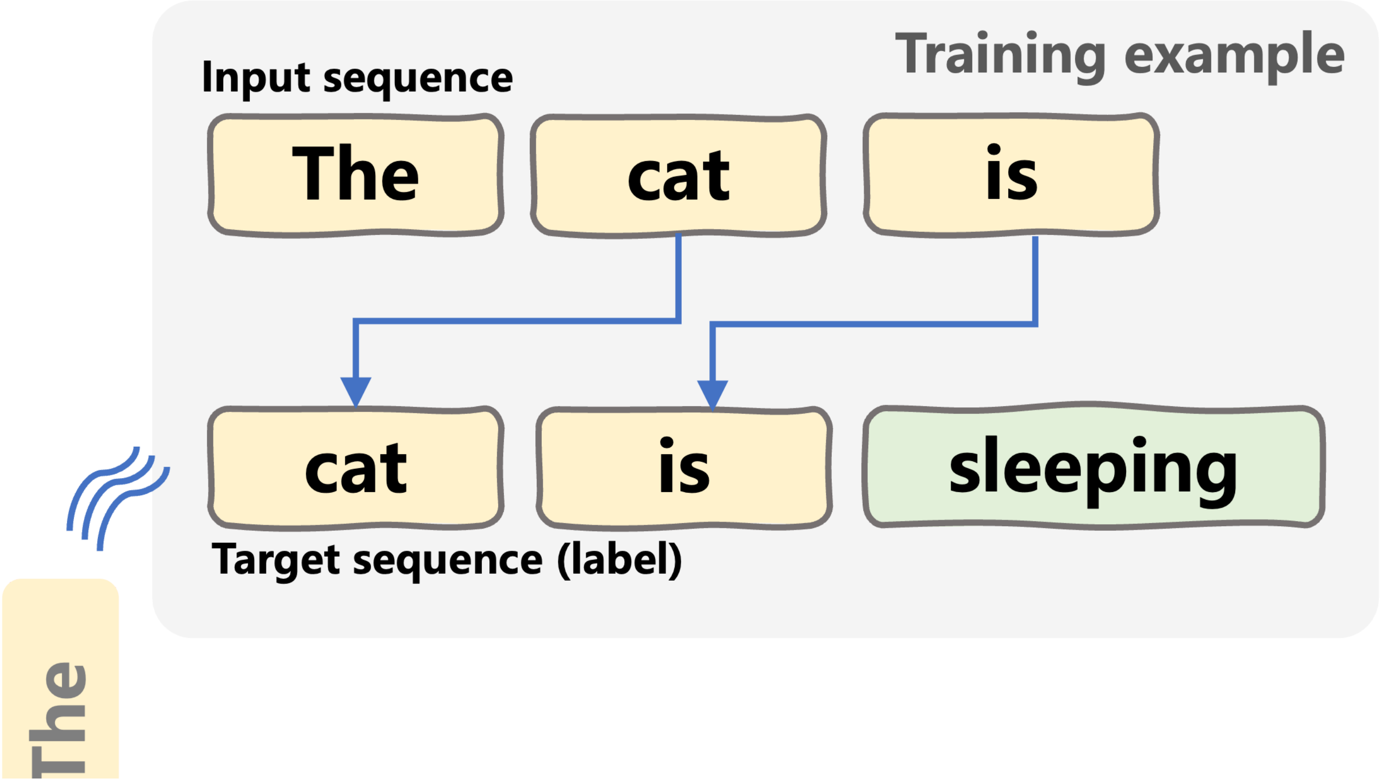
- **Input sequences:** a segment of the text, e.g. "*the cat is*" from "*the cat is sleeping on the mat*"



How text generation LLM training works

Input + target (labels) pairs

- **Input sequences:** a segment of the text, e.g. "*the cat is*" from "*the cat is sleeping on the mat*"
- **Target sequences:** tokens shifted one position to the left, e.g. "*cat is sleeping*"



Let's practice!

INTRODUCTION TO LLMS IN PYTHON

LLMs for text summarization and translation

INTRODUCTION TO LLMS IN PYTHON

Iván Palomares Carrascosa, PhD
Senior Data Science & AI Manager

Inside text summarization

> **Goal:** create a summarized version of a text, preserving important information

- **Inputs:** Original text
- **Target (labels):** summarized text

The majestic mountains stood tall, their peaks shrouded in mist as the first rays of sunlight touched them. A serene river flowed gracefully at the base, reflecting the vibrant colors of the surrounding flora. Birds chirped in harmony, creating a symphony that echoed through the picturesque valley.

Input sequence

Summarized sequence

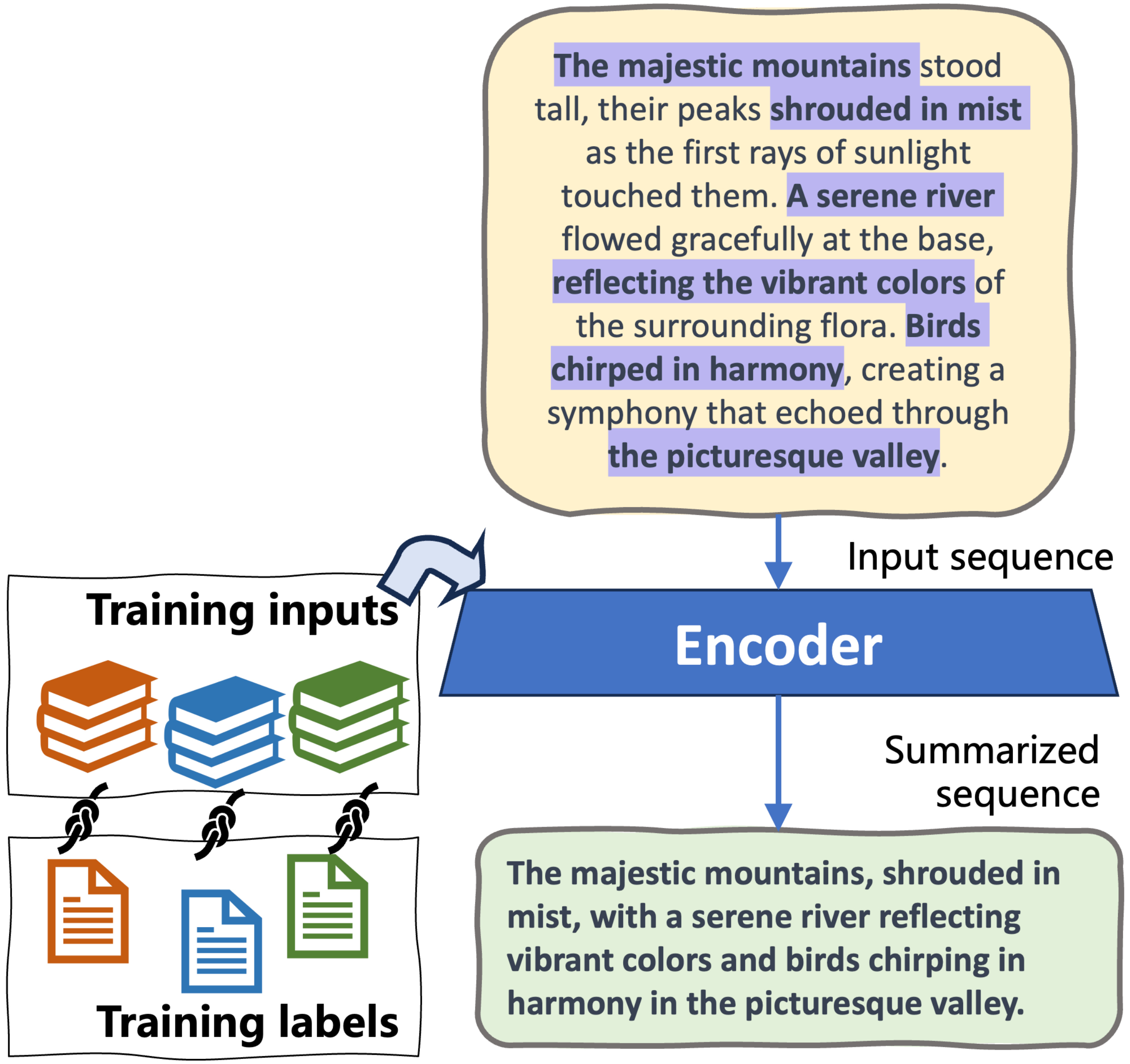
The mist-covered mountains, touched by morning sun, overlooked a serene river reflecting vibrant flora, accompanied by a harmonious bird symphony in the picturesque valley."

Inside text summarization

> **Goal:** create a summarized version of a text, preserving important information

- **Inputs:** Original text
- **Target (labels):** summarized text

Extractive summarization: select, extract, and combine parts of the original text



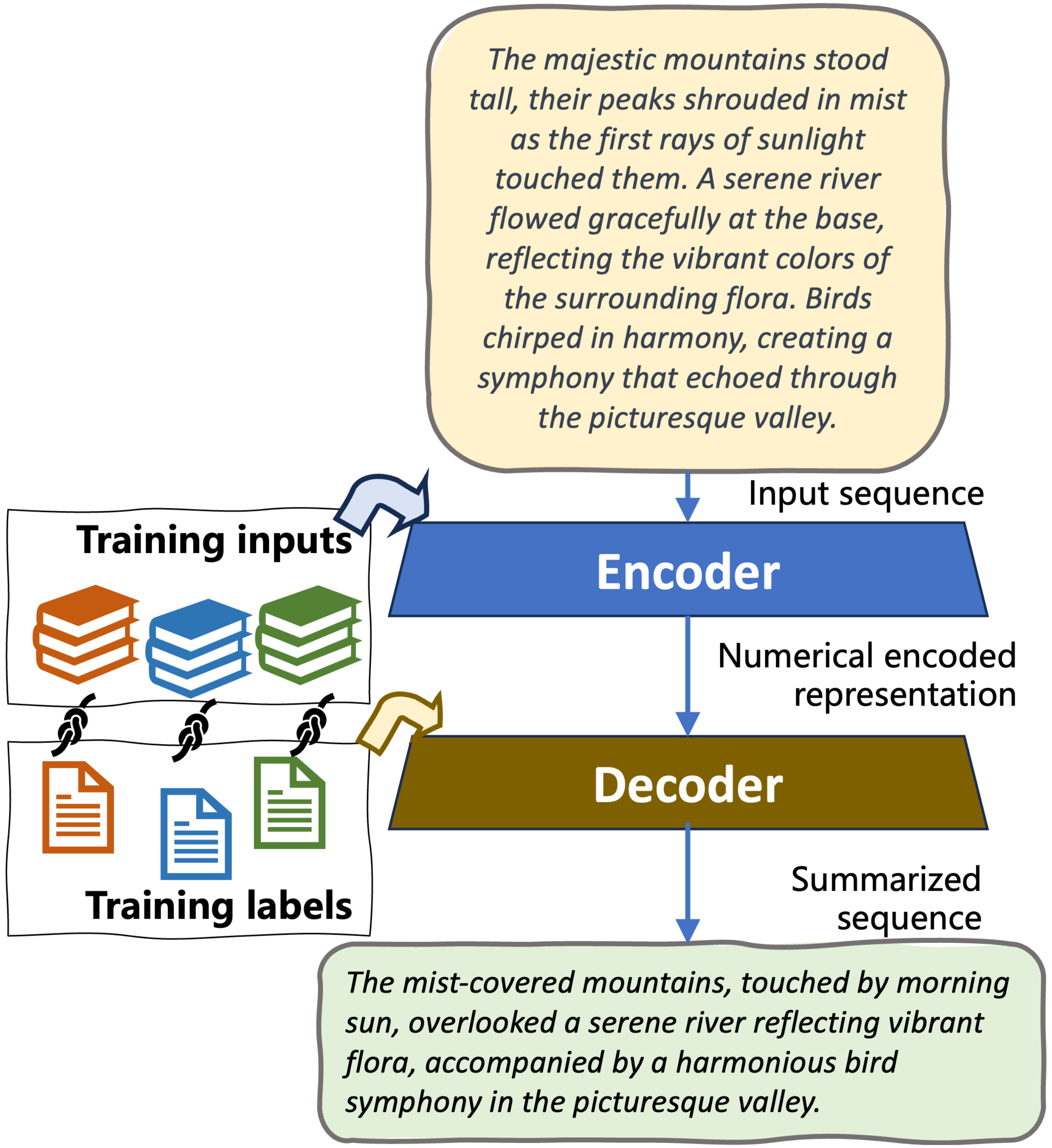
Inside text summarization

> **Goal:** create a summarized version of a text, preserving important information

- **Inputs:** Original text
- **Target (labels):** summarized text

Extractive summarization: select, extract, and combine parts of the original text

Abstractive summarization: generate a summary word by word



Exploring a text summarization dataset

```
from datasets import load_dataset

dataset = load_dataset("ILSUM/ILSUM-1.0", "English")
print(f"Features: {dataset['train'].column_names}")
```

```
Features: ['id', 'Article', 'Heading', 'Summary']
```

Two main text attributes

- *Long text*: input sequence for the LLM
  - `'Article'` in the example
- *Summarized text*: target, training label
  - `'Summary'` in the example

```
example = dataset["train"][21]
example['Article']
```

```
This is how an Apple Watch saved a man's life after detecting
accident. It all started when Gabe Burdett was waiting for his
father Bob at their pre-designated location for some mountain
biking at the Riverside State Park when he received a text
alert from his dad's Apple Watch, saying it had detected a
"hard fall".Burdett, from city of Spokane in Washington State
later received another update from the Watch, saying his father
had reached Sacred Heart Medical Center."We drove straight
there but he was gone when we arrived. I get another (...)
```

```
example['Summary']
```

```
Dad flipped his bike at the bottom of Doomsday, hit his head
and was knocked out until sometime during the ambulance ride.
The watch had called 911 with his location and EMS had him
scooped up and to the hospital in under a 1/2hr.
```

Loading a pre-trained LLM for summarization

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

model_name = "t5-small"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

input_ids = tokenizer.encode(
  "summarize: " + example["Article"],
  return_tensors="pt", max_length=512, truncation=True
)

summary_ids = model.generate(input_ids, max_length=150)
summary = tokenizer.decode(
  summary_ids[0], skip_special_tokens=True)

print("Original Text:")
print(example["Article"])
print("\nGenerated Summary:")
print(summary)
```

- Import and use `AutoModelForSeq2SeqLM`
- Load `t5-small` : versatile for various tasks
- Add a task-specific prefix to the input text: `"summarize:""`
- `.generate()` passes the tokenized input to the model
- `.decode()` post-processes output embedding back into text

Loading a pre-trained LLM for summarization

```python
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

model_name = "t5-small"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

input_ids = tokenizer.encode(
  "summarize: " + example["Article"],
  return_tensors="pt", max_length=512, truncation=True
)

summary_ids = model.generate(input_ids, max_length=150)
summary = tokenizer.decode(
  summary_ids[0], skip_special_tokens=True)

print("Original Text:")
print(example["Article"])
print("\nGenerated Summary:")
print(summary)
```

```
Original Text:
This is how an Apple Watch saved a man's life after detecting
accident. It all started when Gabe Burdett was waiting for his
father Bob at their pre-designated location for some mountain
biking at the Riverside State Park when he received a text
alert from his dad's Apple Watch, saying it had detected a
"hard fall".Burdett, from city of Spokane in Washington State
later received another update from the Watch, saying his father
had reached Sacred Heart Medical Center."We drove straight
there but he was gone when we arrived. I get another (...)
```

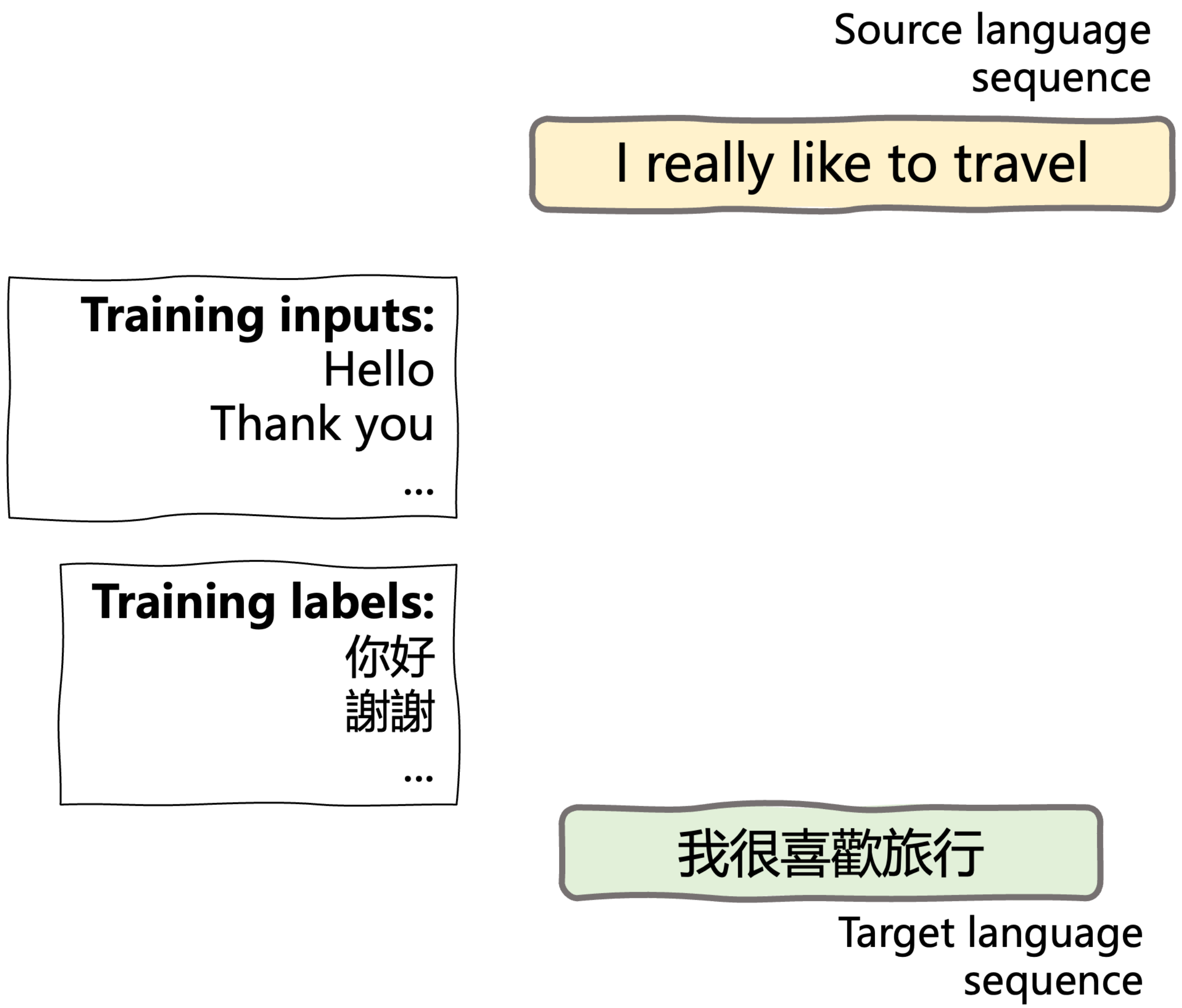
```
Generated Summary:
a man was waiting for his father when he received a text alert
from his dad's apple watch. the watch notified 911 with the
location and within 30 minutes, emergency medical services took
the injured Bob to the hospital. the watch notified 911 with
the location and within 30 minutes, emergency medical services
took the injured Bob to the hospital.
```

[1] Due to space limitations, only the first 50% of the original input text is shown in the slide

Inside language translation

> **Goal:** produce translated version of a text, conveying same meaning and context

- **Inputs:** text in source language
- **Target (labels):** target language translation

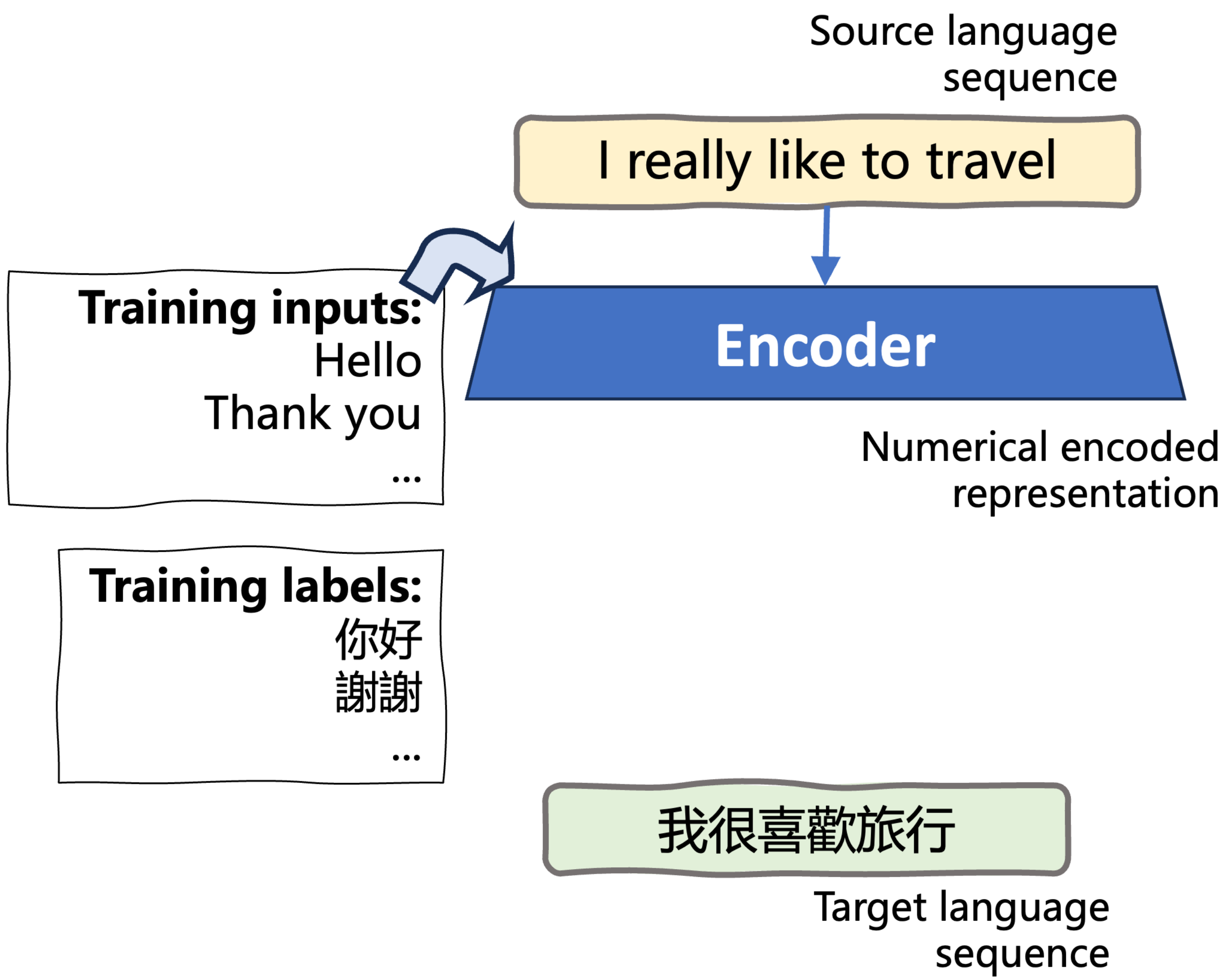


Inside language translation

> **Goal:** produce translated version of a text, conveying same meaning and context

- **Inputs:** text in source language
- **Target (labels):** target language translation

Encode source language sequence



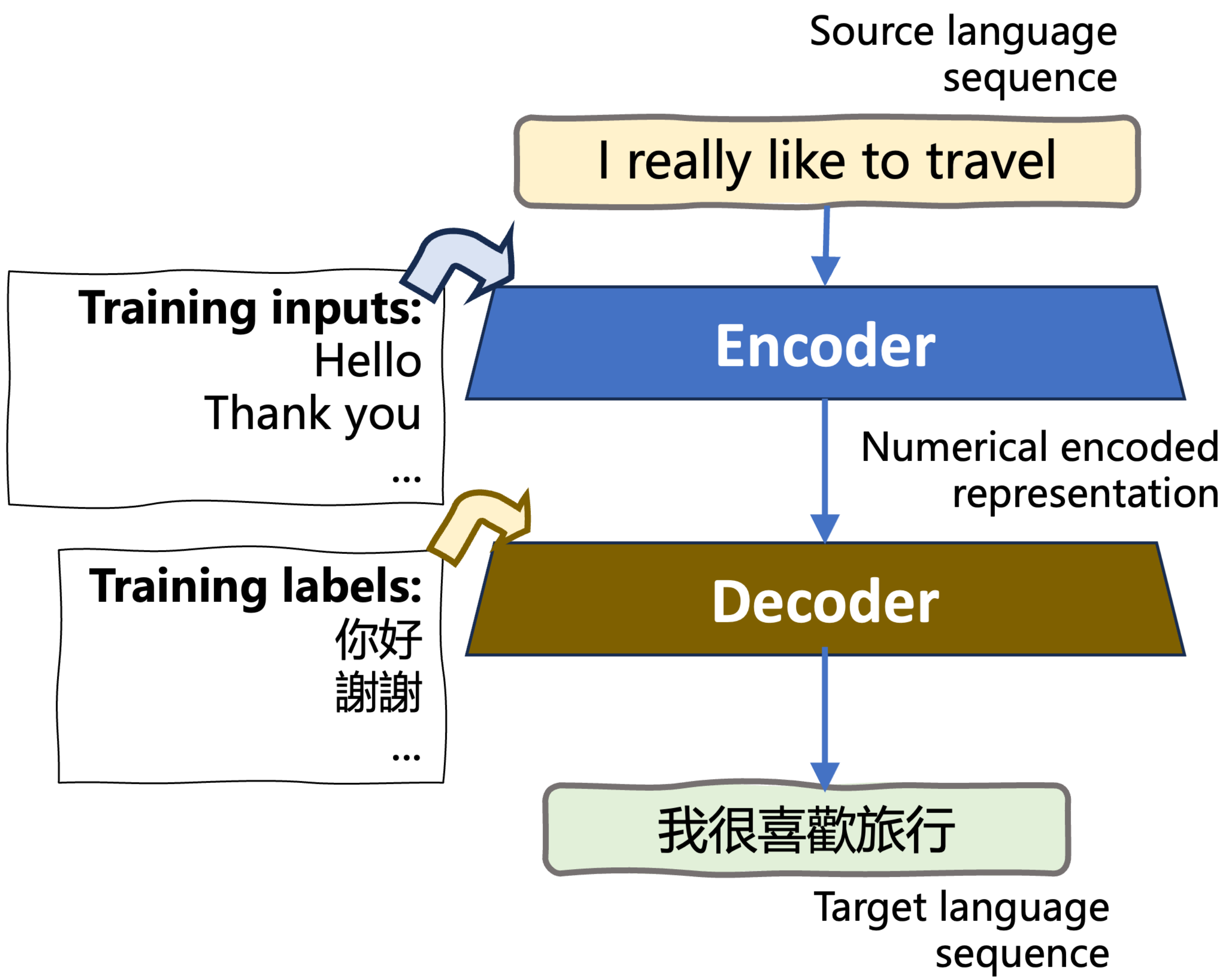
Inside language translation

> **Goal:** produce translated version of a text, conveying same meaning and context

- **Inputs:** text in source language
- **Target (labels):** target language translation

Encode source language sequence

Decode into **target language sequence**, using learned *language patterns* and *associations*



Exploring a language translation dataset

```
from datasets import load_dataset

dataset = load_dataset("techiaith/legislation-gov-uk_en-cy")

sample_data = dataset["train"]

input_example = sample_data.data['source'][0]
target_example = sample_data.data['target'][0]
print("Input (English):", input_example)
print("Target (Welsh):", target_example)
```

```
Input (English): 2 Regulations under section 1: supplementary
Target (Welsh): 2 Rheoliadau o dan adran 1: atodol
```

- Load **English-Welsh** bilingual dataset
  - `Dataset` object
- Extract a training example
  - `source` : English sequences
  - `target` : Welsh sequences

Loading a pre-trained LLM for translation

```python
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

model_name = "Helsinki-NLP/opus-mt-en-cy"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

input_seq = "2 Regulations under section 1: supplementary"
input_ids = tokenizer.encode(input_seq, return_tensors="pt")
translated_ids = model.generate(input_ids)
translated_text = tokenizer.decode(
  translated_ids[0], skip_special_tokens=True)

print("Predicted (Welsh):", translated_text)
```

```
Predicted (Welsh):
  2 Rheloiad o dan adran 1:aryary " means "i
```

- Import and use `AutoModelForSeq2SeqLM`
- Load `Helsinki-NLP` model for English-Welsh translation
- Tokenize English sequence (`.encode()`) and pass it to the model (`.generate()`)
- Decode and print Welsh translation

Let's practice!

INTRODUCTION TO LLMS IN PYTHON

LLMs for question answering

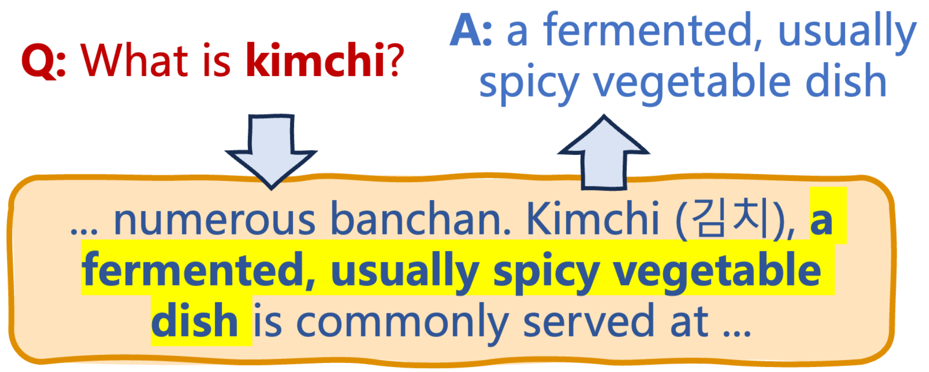
INTRODUCTION TO LLMS IN PYTHON

Iván Palomares Carrascosa, PhD
Senior Data Science & AI Manager

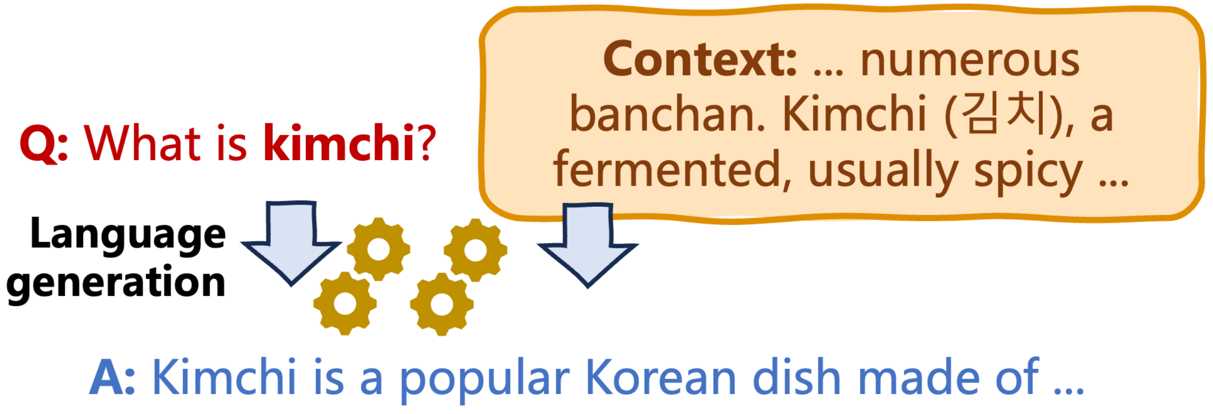
Types of question answering (QA) tasks

QA task type	Architecture
Extractive	Encoder-only
Open Generative	Encoder-Decoder
Closed generative	Decoder-only

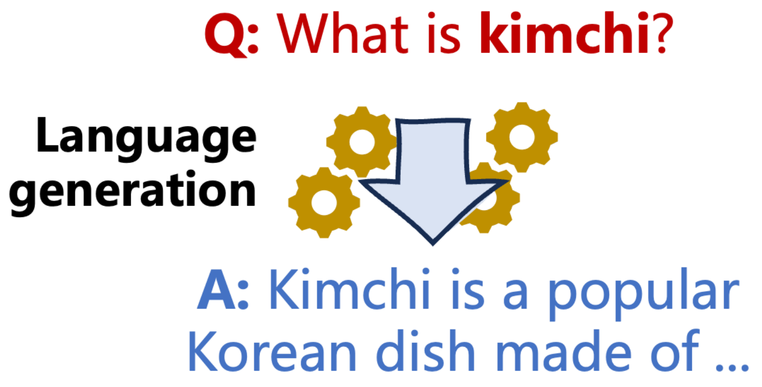
Extractive QA: The LLM *extracts* the answer to a question from a provided context



Open Generative QA: The LLM *generates* the answer based on a context



Closed Generative QA: The LLM fully generates the answer, no context provided



Exploring a QA dataset

```python
from datasets import load_dataset
mlqa = load_dataset(
  "xtreme", name="MLQA.en.en")
print(mlqa)
```

```
DatasetDict({
   test: Dataset({
      features: ['id', 'title', 'context',
                'question', 'answers'],
      num_rows: 11590
   })
   validation: Dataset({
      features: ['id', 'title', 'context',
                'question','answers'],
      num_rows: 1148
   })
})
```

- Load English subset of the *xtreme* dataset for extractive QA
  - `DatasetDict` object.
  - *Test* and *validation* `Dataset` objects.
- Relevant features:
  - `'context'`
  - `'question'`
  - `'answers'`

Exploring a QA dataset

Example instance in the dataset:

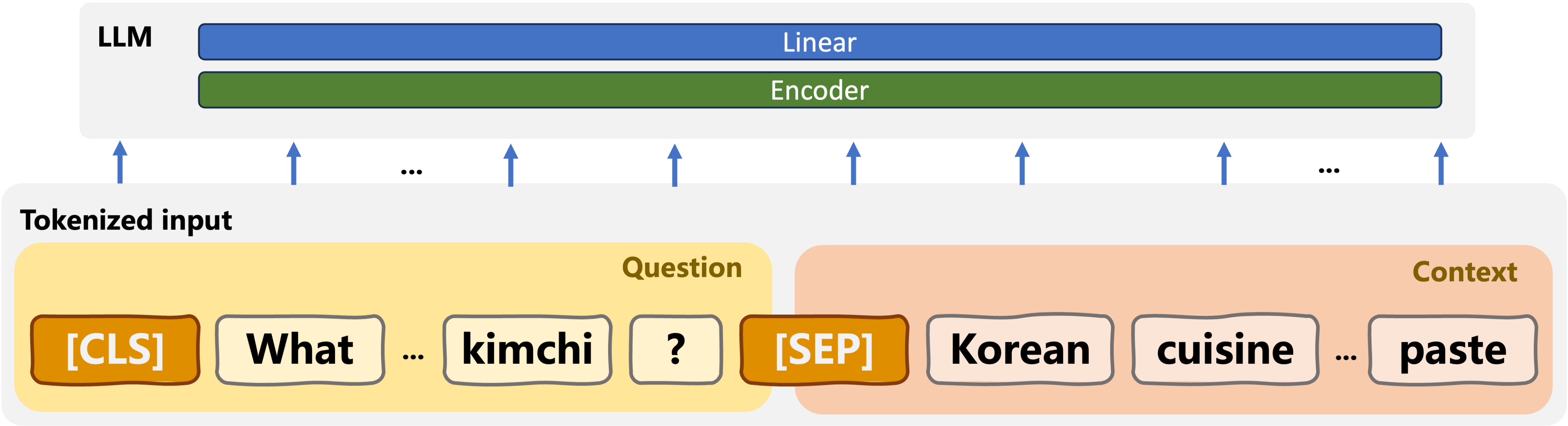
```
print("Question:" , mlqa["test"]["question"][53])
print("Answer:" , mlqa["test"]["answers"][53])
print("Context:" , mlqa["test"]["context"][53])
```

```
Question: what is a kimchi?

Answer: {'answer_start': [271], 'text': ['a fermented, usually spicy vegetable dish']}

Context: Korean cuisine is largely based on rice, noodles, tofu, vegetables, fish and meats. Traditional Korean
meals are noted for the number of side dishes, banchan, which accompany steam-cooked short-grain rice. Every
meal is accompanied by numerous banchan. Kimchi, a fermented, usually spicy vegetable dish is commonly served
at every meal and is one of the best known Korean dishes. Korean cuisine usually involves heavy seasoning with
sesame oil, doenjang, a type of fermented soybean paste, soy sauce, salt, garlic, ginger, and gochujang, a hot
pepper paste. Other well-known dishes are Bulgogi, grilled marinated beef, Gimbap, and Tteokbokki , a spicy
snack consisting of rice cake seasoned with gochujang or a spicy chili paste.
```

Extractive QA: framing the problem

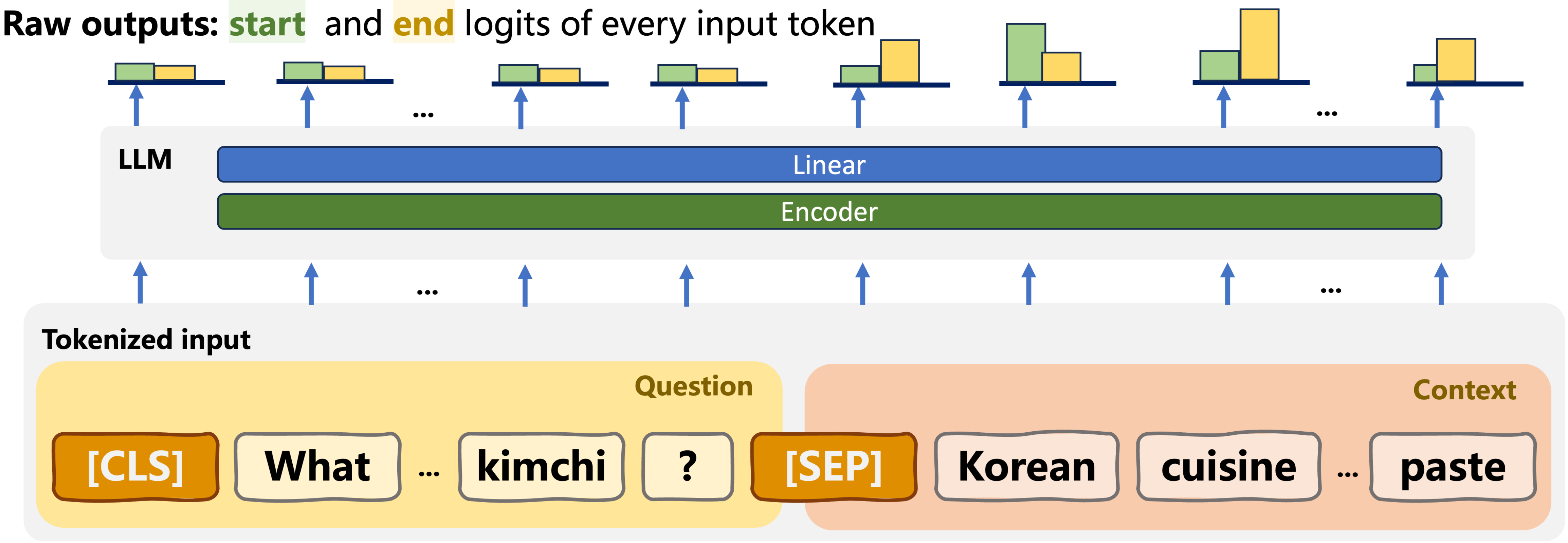
> **Supervised learning:** span classification

Extractive QA: framing the problem



Supervised learning: span classification

Extractive QA: framing the problem



Supervised learning: span classification

Prediction result: answer span given by: [*start position*, *end position*]

- Answer span obtained from *most likely* raw outputs (**logits**)

Extractive QA: tokenizing inputs

```python
from transformers import AutoTokenizer

model_ckp = "deepset/minilm-uncased-squad2"
tokenizer = AutoTokenizer.from_pretrained(model_ckp)

question = "How is the taste of wasabi?"
context = """Japanese cuisine captures the essence of \
a harmonious fusion between fresh ingredients and \
traditional culinary techniques, all heightened \
by the zesty taste of the aromatic green condiment \
known as wasabi."""
inputs = tokenizer(question, context,
                   return_tensors="pt")
```

Tokenization results:

Tensor	Description
input_ids	Integer
attention_mask	Boolean
token_type_ids	**0**: Question, **1**: Context

Extractive QA: loading and using model

```
from transformers import AutoModelForQuestionAnswering

model = AutoModelForQuestionAnswering.
  from_pretrained(model_ckp)

with torch.no_grad():
  outputs = model(**inputs)

start_idx = torch.argmax(outputs.start_logits)
end_idx = torch.argmax(outputs.end_logits) + 1

answer_span = inputs["input_ids"][0]
                    [start_idx:end_idx]
answer = tokenizer.decode(answer_span)
```

- Custom model class: `AutoModelForQuestionAnswering`
- Inference on example input:
  - `**inputs` unpacks and extracts tokenized inputs
- Raw outputs post-processing
  - `start_logits` , `end_logits` answer start/end likelihoods per input token
  - `start_idx` , `end_idx` : positions of input tokens delimiting answer span

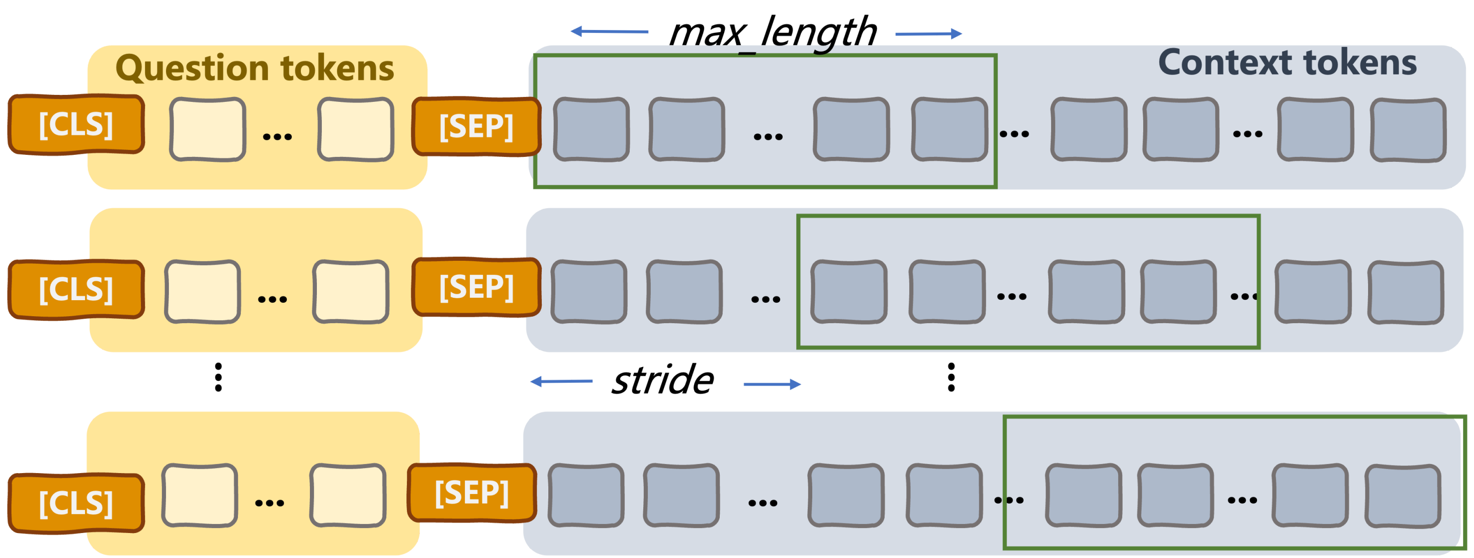
Managing long context sequences

```
long_exmp = tokenizer(example_qt, example_ct,
                      return_overflowing_tokens=True,
                      max_length=100, stride=25)

for idx, window in enumerate(long_exmp["input_ids"]):
  print("Tokens in window ", idx, ": ", len(window))
```

```
No. tokens in window  0 :  100
No. tokens in window  1 :  100
[...]
No. tokens in window  8 :  50
```

```
for window in long_exmp["input_ids"]:
  print(tokenizer.decode(window), "\n")
```

```
[CLS] what is a kimchi? [SEP] Korean cuisine is l[...]
[CLS] what is a kimchi? [SEP] steam-cooked short-[...]
```



Sliding window parameters:

- `max_length` : sliding window size
- `stride` : stride size between windows

Let's practice!

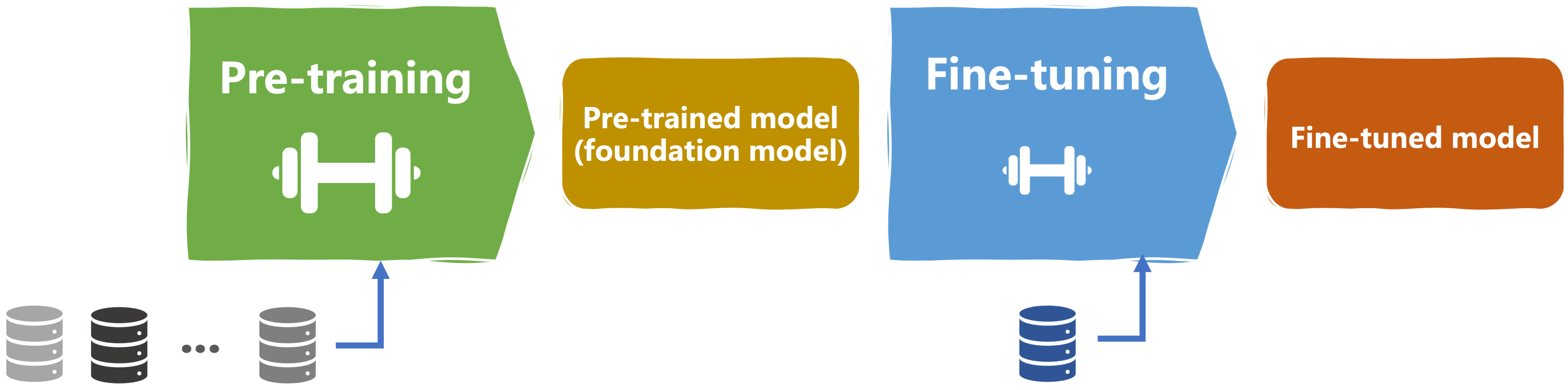
INTRODUCTION TO LLMS IN PYTHON

LLM fine-tuning and transfer learning

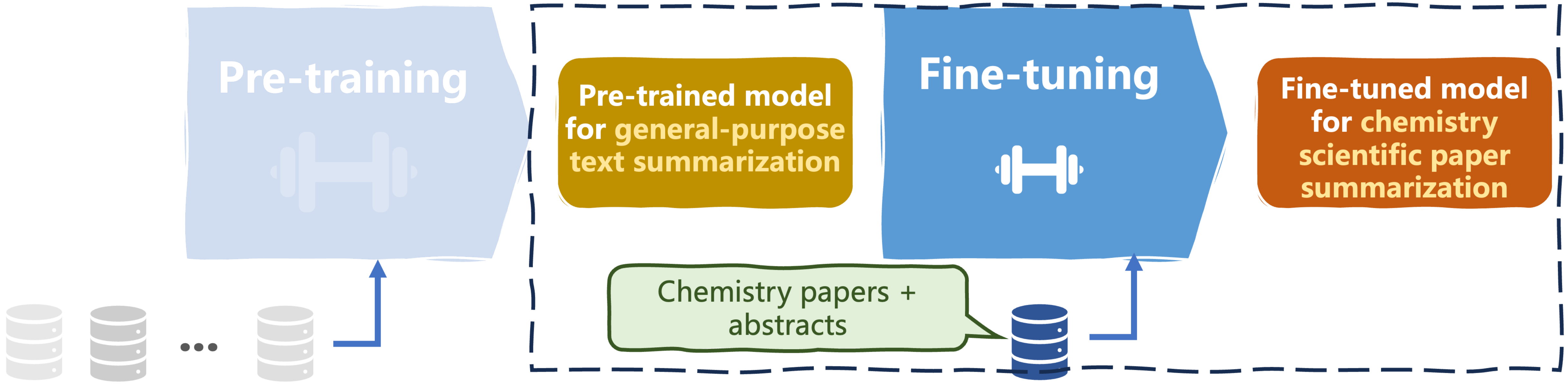
INTRODUCTION TO LLMS IN PYTHON

Iván Palomares Carrascosa, PhD
Senior Data Science & AI Manager

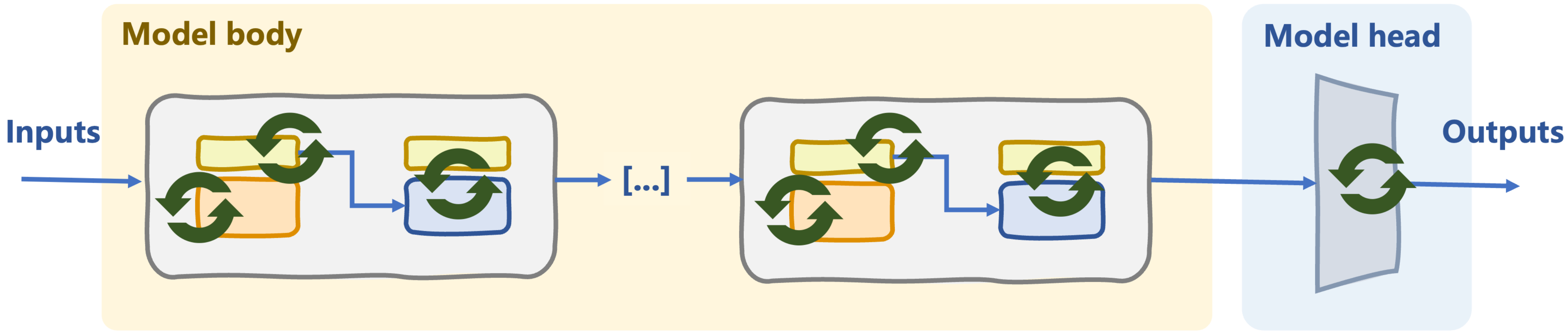
Revisiting the LLM lifecycle



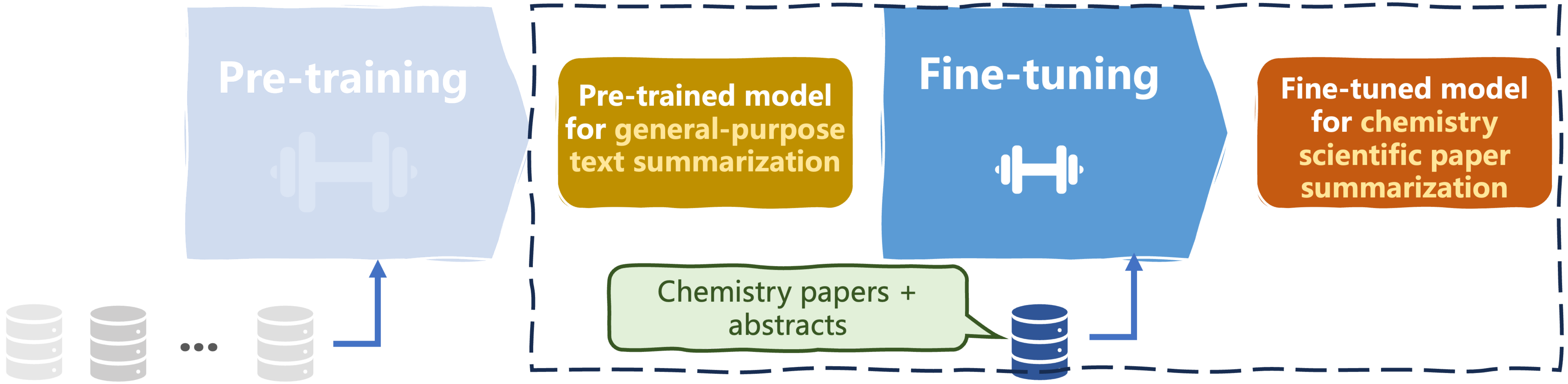
Revisiting the LLM lifecycle



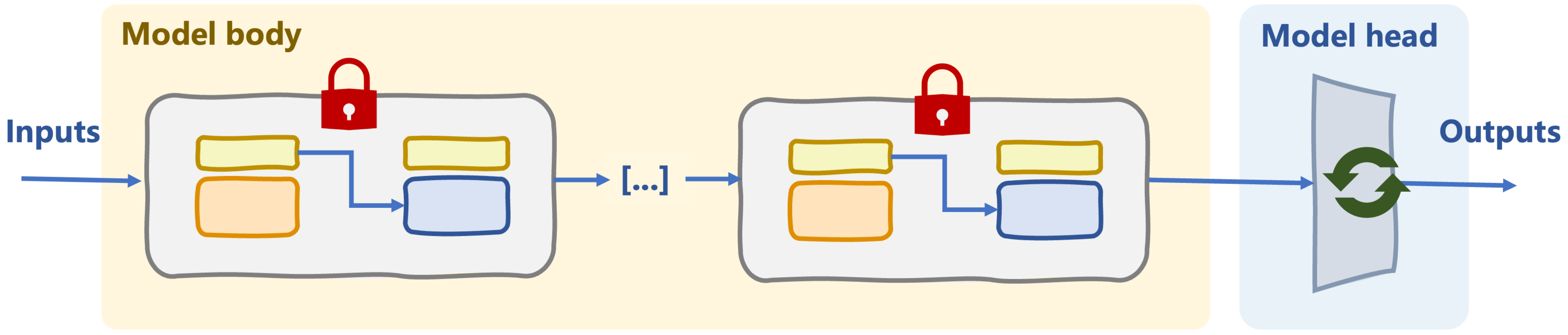
Full fine-tuning: The entire model weights are updated; more computationally expensive



Revisiting the LLM lifecycle



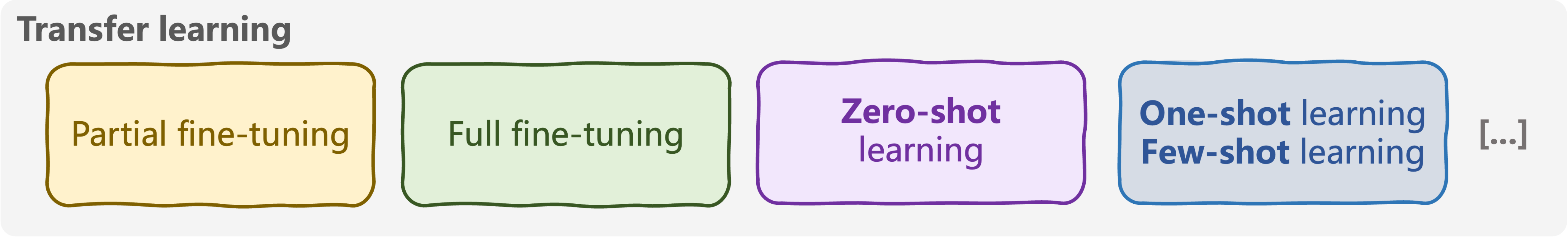
Partial fine-tuning: Lower (body) layers fixed; only task-specific layers (head) are updated



Demystifying transfer learning

Transfer learning: a model trained on one task is adapted for a different but related task

- In pre-trained LLMs, fine-tune on a smaller dataset for a specific task



Zero-shot learning: perform tasks never "seen" during training

One-shot, few-shot learning: adapt a model to a new task with one or a few examples only

Fine-tuning a pre-trained Hugging Face LLM

```
import torch
from transformers import AutoModelForSequenceClassification,
 AutoTokenizer
from datasets import load_dataset

model_name = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(
  model_name, num_labels=2)

def tokenize_function(examples):
    return tokenizer(
      examples["text"], padding="max_length", truncation=True)

data = load_dataset("imdb")
tokenized_data = data.map(tokenize_function, batched=True)
```

- Load BERT-based model for **text classification** and associated **tokenizer**
- **Tokenize dataset** used for fine-tuning
  - IMDB reviews dataset
  - `truncation=True` truncates input sequences beyond model's `max_length`
  - `batched=True` to process examples in batches rather than individually

Fine-tuning a pre-trained Hugging Face LLM

```
from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir="./smaller_bert_finetuned",
    per_device_train_batch_size=8,
    num_train_epochs=3,
    evaluation_strategy="steps",
    eval_steps=500,
    save_steps=500,
    logging_dir="./logs",
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["test"],
)

trainer.train()
```

`TrainingArguments` **class:** customize training settings

- Output directory, batch size per GPU, epochs, etc.

`Trainer` **class:** manage training and validation loop

- Specify model, training arguments, training and validation sets

`trainer.train()` : execute training loop

Inference and saving a fine-tuned LLM

```
example_input = tokenizer("I am absolutely amazed with this
                          new and revolutionary AI device",
                          return_tensors="pt")
output = model(**example_input)
predicted_label = torch.argmax(output.logits, dim=1).item()
print("Predicted Label:", predicted_label)
```

```
Predicted Label: 0
```

```
model.save_pretrained("./my_bert_finetuned")
tokenizer.save_pretrained("./my_bert_finetuned")
```

- After fine-tuning, **inference is performed as usual**
  - Tokenize inputs, pass them to the LLM, obtain and post-process outputs
- Fine-tuned model and tokenizer can be saved using `.save_pretrained()`

Let's practice!

INTRODUCTION TO LLMS IN PYTHON