

Fine-Tuning

Giải thích chi tiết và dễ hiểu về **Fine-Tuning** trong lĩnh vực mô hình ngôn ngữ (LLM):

- Fine-Tuning là gì
- Transformers
- Fine-Tuning trong hệ thống RAG
- Dự án RAG Retriever + Generator Fine-Tuning
- Flan-T5
- Flan-T5 với RAG

Fine-Tuning là gì

Fine-tuning là quá trình **đào tạo lại (huấn luyện tiếp)** một mô hình ngôn ngữ đã được huấn luyện trước (pre-trained) trên **dữ liệu cụ thể của bạn**, để mô hình:

- Hiểu tốt hơn về lĩnh vực bạn quan tâm (ví dụ: y tế, pháp lý, kỹ thuật...),
- Trả lời chính xác và phù hợp hơn với yêu cầu của ứng dụng thực tế,
- Tuân theo phong cách viết, giọng điệu hoặc cấu trúc riêng.

Ví dụ dễ hiểu

Giả sử bạn có một mô hình GPT đã học hàng tỷ câu văn từ Internet. Nếu bạn muốn nó:

- Trả lời theo cách **ngghiêm túc, ngắn gọn, kỹ thuật**,
- Hoặc chuyên trả lời **câu hỏi về luật Việt Nam**,
thì bạn sẽ **fine-tune** nó bằng cách huấn luyện thêm trên tập dữ liệu nhỏ gồm các ví dụ bạn mong muốn.

Fine-tuning khác với Prompt Engineering thế nào?

Prompt Engineering	Fine-Tuning
Thay đổi prompt để điều khiển đầu ra	Huấn luyện lại mô hình để thay đổi hành vi
Không tốn chi phí đào tạo lại	Tốn tài nguyên (GPU, RAM, thời gian)
Dễ làm, nhưng hiệu quả có giới hạn	Mạnh hơn, dùng cho yêu cầu phức tạp
Không cần dữ liệu huấn luyện	Cần dữ liệu huấn luyện có nhãn

Có mấy loại Fine-Tuning?

1. Full Fine-Tuning (ít dùng với LLM lớn)

- Cập nhật **toàn bộ trọng số (weights)** của mô hình.
- Yêu cầu **nhiều tài nguyên** → không hiệu quả với mô hình lớn như LLaMA-2-13B, GPT-J...

2. PEFT (Parameter-Efficient Fine-Tuning) – [rất phổ biến hiện nay]

Gồm các kỹ thuật như:

- **LoRA** (Low-Rank Adaptation)
- **QLoRA** (LoRA kết hợp nén và huấn luyện trên GPU yếu hơn)
- **Prefix Tuning, Adapter Tuning...**

Ưu điểm:

- Chỉ fine-tune **rất ít tham số** (~0.1%-1%) → tiết kiệm chi phí và tài nguyên.
- Có thể lưu nhiều “bản cập nhật nhỏ” cho các mục tiêu khác nhau.

QLoRA là gì?

QLoRA = LoRA + Quantization (4-bit)

Mục tiêu là fine-tune mô hình lớn (ví dụ: LLaMA-2-13B) **trên laptop hoặc GPU yếu**.

- **Quantization**: giảm độ chính xác xuống 4-bit để tiết kiệm RAM/GPU.
- **LoRA**: chèn các lớp nhỏ để học thêm nhưng không thay đổi mô hình gốc.

Rất phổ biến để fine-tune mô hình lớn với ngân sách thấp!

Khi nào nên fine-tune?

Trường hợp	Có nên fine-tune?
Muốn mô hình hiểu văn bản nội bộ, tài liệu chuyên ngành	Có
Muốn mô hình trả lời theo phong cách riêng	Có
Chỉ cần thay đổi nhẹ câu trả lời	<input type="checkbox"/> Dùng prompt thôi là đủ
Không có dữ liệu huấn luyện	<input type="checkbox"/> Không fine-tune được

Các công cụ hỗ trợ Fine-tuning

- **Hugging Face Transformers + PEFT + TRL** (Python, rất phổ biến)
- **LoRA, QLoRA với bitsandbytes**
- **OpenAI Fine-tuning API** (với GPT-3.5-Turbo)
- **Axolotl, SFTTrainer, AutoTrain...**

Transformers

Transformers là một kiến trúc mạng nơ-ron (neural network architecture) được giới thiệu bởi Google trong bài báo nổi tiếng năm 2017: **"Attention is All You Need"**.

Nó **thay thế hoàn toàn RNN/LSTM** trong việc xử lý chuỗi dữ liệu (như văn bản), và **trở thành nền tảng** cho hầu hết các mô hình ngôn ngữ hiện đại (LLM).

Ý tưởng cốt lõi: Attention

Điểm mạnh của Transformers là **cơ chế Attention**, cụ thể là **Self-Attention**.

Giả sử bạn có câu:
"The cat sat on the mat because it was tired."

Từ **"it"** cần hiểu là đang nói đến **"the cat"**.
Cơ chế **attention** giúp mô hình xác định được từ nào trong câu **liên quan nhất** đến từ hiện tại.

Cấu trúc của Transformer

Có 2 phần chính:

1. Encoder (bộ mã hóa)

- Dùng trong BERT, T5 (phần mã hóa).
- Hiểu toàn bộ ngữ cảnh của chuỗi đầu vào.

2. Decoder (bộ giải mã)

- Dùng trong GPT, T5 (phần sinh văn bản).
- Dự đoán từ tiếp theo dựa trên các từ trước đó.

Tóm tắt:

Mô hình	Dùng phần nào?
BERT	Encoder
GPT	Decoder
T5	Encoder + Decoder

Thành phần chính trong mỗi layer

1. Multi-Head Self Attention

- Cho phép mô hình "chú ý" đến nhiều phần khác nhau của câu cùng lúc.

2. Feed-Forward Neural Network

- Một MLP đơn giản sau mỗi attention.

3. Layer Normalization

- Giúp mô hình ổn định trong quá trình huấn luyện.

4. Residual Connections

- Giúp tránh mất thông tin và tăng hiệu quả học.

Vị trí từ (Positional Encoding)

Transformers **không có khái niệm tuần tự** như RNN.

→ Phải **thêm thông tin vị trí** bằng Positional Encoding để mô hình biết thứ tự từ trong câu.

Vì sao Transformers lại mạnh?

- **Huấn luyện song song** (không tuần tự như RNN) → nhanh hơn rất nhiều.
- **Tăng khả năng học ngữ cảnh xa** (không bị "quên" từ đầu câu).
- **Học được từ dữ liệu lớn**, dẫn đến khả năng tổng quát mạnh mẽ.

Hugging Face Transformers là gì?

Đây là **thư viện mã nguồn mở** giúp bạn dễ dàng:

- Sử dụng các mô hình transformer như BERT, GPT, T5, LLaMA...
- Dùng để **fine-tune**, huấn luyện, đánh giá mô hình.
- Tích hợp với **datasets**, **tokenizers**, và **PEFT** (fine-tuning hiệu quả).

Ví dụ sử dụng nhanh (với Hugging Face):

```
from transformers import pipeline

qa = pipeline("question-answering", model="distilbert-base-cased-distilled-squad")

qa({
    "context": "The cat sat on the mat because it was tired.",
    "question": "Why did the cat sit on the mat?"
})
```

Tóm tắt dễ nhớ

Thuật ngữ	Ý nghĩa ngắn gọn
Transformer	Kiến trúc xử lý chuỗi mạnh mẽ, thay thế RNN
Attention	Cơ chế "chú ý" đến phần quan trọng của chuỗi
Self-Attention	Mỗi từ chú ý đến các từ khác trong chuỗi

Thuật ngữ	Ý nghĩa ngắn gọn
Encoder / Decoder	Mã hóa / Sinh văn bản
Hugging Face	Thư viện dễ dùng để tận dụng mô hình này

Fine-Tuning trong hệ thống RAG

Fine-tuning đóng vai trò rất quan trọng trong việc **nâng cao chất lượng** của hệ thống **RAG (Retrieval-Augmented Generation)**. Dưới đây là cách **ứng dụng Fine-Tuning** cho từng phần trong hệ thống RAG, kèm theo ví dụ và hướng dẫn triển khai.

Tổng quan về hệ thống RAG

Một hệ thống RAG gồm 2 thành phần chính:

- Retriever:** Truy xuất các đoạn văn bản liên quan từ kho dữ liệu.
- Generator (LLM):** Sinh câu trả lời dựa trên văn bản đã truy xuất.

Fine-tuning có thể được áp dụng cho **retriever**, **generator**, hoặc cả hai để cải thiện độ chính xác và tính hữu ích của câu trả lời.

1. Fine-Tuning Retriever

Mục tiêu:

- Giúp retriever tìm đúng các đoạn văn bản **liên quan hơn** đến câu hỏi của người dùng.

Cách làm:

- Sử dụng các cặp dữ liệu `query <-> relevant passage`.
- Huấn luyện mô hình bi-encoder (ví dụ: `sentence-transformers`) để embedding câu hỏi và tài liệu gần nhau trong không gian vector.

Công cụ:

- `sentence-transformers` (Hugging Face)
- Datasets: Haystack, BEIR, custom Q&A pairs

Ví dụ:

```
from sentence_transformers import SentenceTransformer, InputExample, losses
from torch.utils.data import DataLoader

train_examples = [
```

```

InputExample(texts=["What is the capital of France?", "Paris is the capital of France."]),
...
]
model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
train_dataloader = DataLoader(train_examples, batch_size=16)
train_loss = losses.MultipleNegativesRankingLoss(model)

model.fit(train_objectives=[(train_dataloader, train_loss)], epochs=1)

```

2. Fine-Tuning Generator (LLM)

Mục tiêu:

- Giúp LLM sinh ra câu trả lời chính xác hơn **dựa trên context truy xuất được**.

Cách làm:

- Sử dụng tập dữ liệu gồm: `context`, `question`, và `answer`.
- Fine-tune LLM như `T5`, `GPT-2`, `Mistral`, `LLaMA` trên các cặp `input: context + question → output: answer`.

Công cụ:

- Hugging Face Transformers
- PEFT / LoRA / QLoRA để fine-tune mô hình lớn nhẹ hơn

Ví dụ:

```

from transformers import T5Tokenizer, T5ForConditionalGeneration, Trainer, TrainingArguments

tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small")

def preprocess(example):
    input_text = f"question: {example['question']} context: {example['context']}"
    target_text = example["answer"]
    return tokenizer(input_text, truncation=True, padding="max_length", max_length=512), \
        tokenizer(target_text, truncation=True, padding="max_length", max_length=64)

# Sử dụng Trainer API để fine-tune

```

3. Fine-Tuning kết hợp Retriever + Generator

- Một số framework (như [Haystack](#), [RAG](#)) hỗ trợ fine-tune **toàn bộ pipeline** end-to-end.
- Tuy nhiên, việc này **đắt đỏ** và phức tạp hơn → nên ưu tiên fine-tune từng phần riêng trước.

Khi nào nên Fine-Tune trong RAG?

Dấu hiệu	Giải pháp
Truy xuất sai tài liệu	Fine-tune retriever
Sinh câu trả lời sai / không khớp	Fine-tune generator
Cần tối ưu cho domain cụ thể	Fine-tune cả hai

Tools hỗ trợ Fine-tuning RAG

- [LangChain](#) + [OpenAI](#) / [HuggingFace](#)
- [Haystack](#)
- [Hugging Face Transformers](#) + [PEFT](#)
- [TruLens](#) hoặc [Promptfoo](#) để đánh giá chất lượng sau fine-tuning

Thành phần	Fine-tuning giúp gì?
Retriever	Tìm đúng đoạn tài liệu liên quan
Generator	Sinh câu trả lời chính xác hơn từ context
Toàn hệ thống	Tối ưu hóa end-to-end, tăng chất lượng RAG

Tóm tắt:

Thành phần	Fine-tuning giúp gì?
Retriever	Tìm đúng đoạn tài liệu liên quan
Generator	Sinh câu trả lời chính xác hơn từ context
Toàn hệ thống	Tối ưu hóa end-to-end, tăng chất lượng RAG

Dự án RAG Retriever + Generator Fine-Tuning

Dưới đây là một **template RAG đầy đủ (Retriever + Generator)** có hỗ trợ **Fine-Tuning cả hai phần: Retriever (bi-encoder) và Generator (T5 hoặc LLaMA)**. Mình sẽ chia thành từng bước rõ ràng:

Cấu trúc thư mục đề xuất

```
rag_finetuning_project/  
├─ data/  
│   └─ train.jsonl  
├─ retriever/  
│   ├── train_retriever.py  
│   └─ model/  
├─ generator/  
│   ├── train_generator.py  
│   └─ model/  
├─ rag_pipeline.py  
└─ requirements.txt
```

1. Dataset format

`data/train.jsonl`:

```
{"question": "What is the capital of France?", "context": "Paris is the capital of France.", "answer": "Paris"}  
{"question": "What is the currency of Japan?", "context": "The yen is the official currency of Japan.", "answer":  
"yen"}  
...
```

2. Fine-Tune Retriever (`retriever/train_retriever.py`)

Sử dụng `sentence-transformers`.

```

from sentence_transformers import SentenceTransformer, InputExample, losses
from torch.utils.data import DataLoader
import json

# Load data
examples = []
with open("../data/train.jsonl", "r") as f:
    for line in f:
        ex = json.loads(line)
        examples.append(InputExample(texts=[ex["question"], ex["context"]]))

# Load model
model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")
train_dataloader = DataLoader(examples, batch_size=16)
train_loss = losses.MultipleNegativesRankingLoss(model)

# Train
model.fit(train_objectives=[(train_dataloader, train_loss)], epochs=1, output_path="model/")

```

3. Fine-Tune Generator (generator/train_generator.py)

Sử dụng `T5` từ Hugging Face Transformers.

```

from transformers import T5Tokenizer, T5ForConditionalGeneration, Trainer, TrainingArguments
from datasets import load_dataset
import json

def load_data():
    data = []
    with open("../data/train.jsonl", "r") as f:
        for line in f:
            ex = json.loads(line)
            input_text = f"question: {ex['question']} context: {ex['context']}"
            data.append({"input": input_text, "label": ex["answer"]})
    return data

data = load_data()
from datasets import Dataset
dataset = Dataset.from_list(data)

tokenizer = T5Tokenizer.from_pretrained("t5-small")

```

```

model = T5ForConditionalGeneration.from_pretrained("t5-small")

def preprocess(examples):
    model_input = tokenizer(examples["input"], truncation=True, padding="max_length", max_length=512)
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(examples["label"], truncation=True, padding="max_length", max_length=32)
    model_input["labels"] = labels["input_ids"]
    return model_input

tokenized = dataset.map(preprocess, batched=True)

args = TrainingArguments(
    output_dir="model/",
    evaluation_strategy="no",
    per_device_train_batch_size=4,
    num_train_epochs=3,
    save_strategy="epoch",
)

trainer = Trainer(
    model=model,
    args=args,
    train_dataset=tokenized,
    tokenizer=tokenizer,
)

trainer.train()

```

4. Tích hợp Retriever + Generator (`rag_pipeline.py`)

```

from sentence_transformers import SentenceTransformer, util
from transformers import T5Tokenizer, T5ForConditionalGeneration

retriever = SentenceTransformer("retriever/model")
generator = T5ForConditionalGeneration.from_pretrained("generator/model")
tokenizer = T5Tokenizer.from_pretrained("t5-small")

# Fake corpus (ở thực tế sẽ là hàng ngàn documents)
corpus = [
    "Paris is the capital of France.",
    "Tokyo is the capital of Japan.",

```

```

    "The yen is the currency of Japan.",
]

corpus_embeddings = retriever.encode(corpus, convert_to_tensor=True)

def rag_answer(question):
    question_embedding = retriever.encode(question, convert_to_tensor=True)
    top_hit = util.semantic_search(question_embedding, corpus_embeddings, top_k=1)[0][0]
    context = corpus[top_hit['corpus_id']]

    input_text = f"question: {question} context: {context}"
    input_ids = tokenizer(input_text, return_tensors="pt", truncation=True).input_ids
    output_ids = generator.generate(input_ids)
    return tokenizer.decode(output_ids[0], skip_special_tokens=True)

# Ví dụ:
print(rag_answer("What is the capital of France?"))

```

requirements.txt

```

transformers
datasets
sentence-transformers
torch

```

Tùy chọn nâng cao

- **Thay T5 bằng LLaMA/Mistral** với QLoRA → sử dụng `PEFT`.
- **Sử dụng LangChain hoặc Haystack** để xây dựng UI / chatbot interface.
- **Sử dụng FAISS hoặc Weaviate** cho truy vấn triệu tài liệu.

Flan-T5

Flan-T5 là một mô hình ngôn ngữ do **Google** huấn luyện, thuộc họ **T5 (Text-To-Text Transfer Transformer)**.

Tên đầy đủ: "**Fine-tuned L**anguage **N**et **T5**"

Nó là **T5 được huấn luyện lại (fine-tune)** để **trả lời câu hỏi tốt hơn, chính xác hơn, và hiểu lệnh tốt hơn**.

Cấu trúc cơ bản

Flan-T5 sử dụng mô hình **T5 gốc** – một Transformer với kiến trúc **Encoder-Decoder**:

- **Encoder**: Hiểu văn bản đầu vào.
- **Decoder**: Sinh văn bản đầu ra.

Và mọi thứ đều được xử lý dưới dạng **text → text**.

Ví dụ:

Input	Output
"Translate English to French: Hello"	"Bonjour"
"Summarize: Hôm nay trời đẹp..."	"Trời đẹp hôm nay."
"What is the capital of Japan?"	"Tokyo"

Flan-T5 có gì đặc biệt?

“ Flan-T5 = T5 + fine-tuning thêm **hàng loạt tác vụ hướng lệnh (instruction tuning)** ”

Những điểm cải tiến:

1. **Instruction tuning**:

Flan-T5 được huấn luyện để hiểu **lệnh rõ ràng**, ví dụ như:

- “Tóm tắt văn bản sau...”
- “Dịch sang tiếng Đức...”
- “Viết lại đoạn văn cho dễ hiểu...”

2. **Đa nhiệm (multi-task learning)**:

Huấn luyện trên **hơn 1000 loại tác vụ NLP** khác nhau.

3. **Nhiều kích thước**:

- `flan-t5-small`, `flan-t5-base`, `flan-t5-large`, `flan-t5-xl`, `flan-t5-xxl` (lớn nhất ~11B tham số)

4. Dễ dùng, mã nguồn mở:

- Có sẵn trên Hugging Face
- Chỉ cần vài dòng code để sử dụng.

Ví dụ dùng Flan-T5 với Python

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

model = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-base")
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-base")

input_text = "Translate English to Vietnamese: How are you?"
inputs = tokenizer(input_text, return_tensors="pt")
outputs = model.generate(**inputs)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

Kết quả: "Bạn khỏe không?"

Tóm tắt

Tiêu chí	Flan-T5
Thuộc loại	Transformer (T5 - Encoder-Decoder)
Do ai tạo	Google
Mục tiêu	Giải quyết đa nhiệm NLP theo lệnh
Cách hoạt động	Text in → Text out
Ưu điểm chính	Hiểu tốt câu lệnh, đa nhiệm, dễ fine-tune

Flan-T5 với RAG

Dưới đây là một **bài thực hành chi tiết** giúp bạn xây dựng hệ thống **RAG (Retrieval-Augmented Generation)** sử dụng mô hình **Flan-T5** để trả lời câu hỏi dựa trên tài liệu PDF. Hệ thống này sẽ:

- **Trích xuất nội dung** từ tệp PDF.
- **Chia nhỏ văn bản** thành các đoạn (chunk) và tạo **vector embedding** cho từng đoạn.
- **Lưu trữ các vector** trong cơ sở dữ liệu FAISS để truy vấn nhanh chóng.
- **Truy xuất các đoạn văn bản liên quan** đến câu hỏi người dùng.
- **Sử dụng Flan-T5** để sinh câu trả lời dựa trên ngữ cảnh truy xuất được.

Môi trường và thư viện cần thiết

Trước tiên, hãy cài đặt các thư viện cần thiết:

```
pip install transformers sentence-transformers faiss-cpu pypdf
```

1. Trích xuất văn bản từ PDF

```
from pypdf import PdfReader

def extract_text_from_pdf(pdf_path):
    reader = PdfReader(pdf_path)
    text = ""
    for page in reader.pages:
        text += page.extract_text()
    return text
```

2. Chia văn bản thành các đoạn nhỏ (chunk)

```
def split_text(text, chunk_size=500, overlap=50):
    chunks = []
    start = 0
    while start < len(text):
        end = start + chunk_size
        chunks.append(text[start:end])
        start += chunk_size - overlap
```

```
return chunks
```

3. Tạo vector embedding cho từng đoạn văn bản

```
from sentence_transformers import SentenceTransformer

def create_embeddings(chunks):
    model = SentenceTransformer('all-MiniLM-L6-v2')
    embeddings = model.encode(chunks)
    return embeddings, model
```

4. Lưu trữ embeddings trong FAISS5. Truy xuất các đoạn văn bản liên quan đến câu hỏi

```
def retrieve_relevant_chunks(question, model, index, chunks, top_k=3):
    question_embedding = model.encode([question])
    distances, indices = index.search(np.array(question_embedding), top_k)
    retrieved_chunks = [chunks[i] for i in indices[0]]
    return retrieved_chunks
```

6. Sinh câu trả lời bằng Flan-T5

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

def generate_answer(question, context, model_name="google/flan-t5-base"):
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

    prompt = f"Context: {context}\n\nQuestion: {question}\n\nAnswer:"
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True)
    outputs = model.generate(**inputs, max_new_tokens=100)
    answer = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return answer
```

7. Kết hợp tất cả thành một pipeline

```
def rag_pipeline(pdf_path, question):
    # Bước 1: Trích xuất và xử lý văn bản
    text = extract_text_from_pdf(pdf_path)
    chunks = split_text(text)

    # Bước 2: Tạo embeddings và index
    embeddings, embed_model = create_embeddings(chunks)
```

```
index = create_faiss_index(np.array(embeddings))

# Bước 3: Truy xuất các đoạn liên quan
relevant_chunks = retrieve_relevant_chunks(question, embed_model, index, chunks)
context = " ".join(relevant_chunks)

# Bước 4: Sinh câu trả lời
answer = generate_answer(question, context)

return answer
```

Ví dụ sử dụng

```
pdf_path = "duong-luoi-bo.pdf"
question = "Tranh chấp ở Biển Đông bắt đầu từ khi nào?"
answer = rag_pipeline(pdf_path, question)
print("Câu trả lời:", answer)
```

Gợi ý nâng cao

- **Tăng hiệu suất:** Sử dụng `faiss-gpu` nếu bạn có GPU.
- **Cải thiện chất lượng câu trả lời:** Fine-tune Flan-T5 trên tập dữ liệu của bạn.
- **Giao diện người dùng:** Tích hợp với Gradio hoặc Streamlit để tạo giao diện thân thiện.