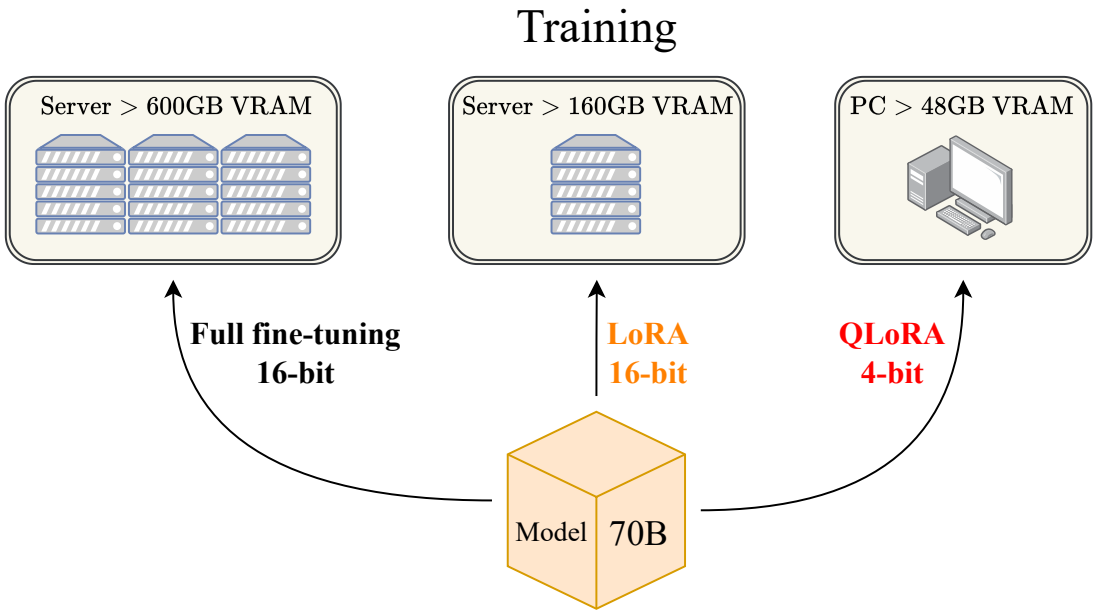
AI VIET NAM – AI COURSE 2024

Tutorial: Low-rank Adaptation Techniques in Fine-tuning a Large Language Model

Minh-Nam Tran, Anh-Khoi Nguyen, Dinh-Thang Duong, Quang-Vinh Dinh

I. Giới thiệu

Low-Rank Adaptation (LoRA) Techniques (Tạm dịch: Kỹ thuật điều chỉnh hạng thấp) là một phương pháp hiệu quả trong việc fine-tuning các mô hình Deep Learning, đặc biệt là trong huấn luyện các mô hình ngôn ngữ lớn (LLMs). Thay vì cập nhật toàn bộ các tham số của mô hình, LoRA tập trung vào việc tìm kiếm một cấu trúc có hạng thấp cho các ma trận trọng số. Phương pháp này không chỉ giảm thiểu số lượng tham số cần điều chỉnh mà vẫn duy trì hoặc thậm chí cải thiện hiệu năng của mô hình, mà còn cho phép huấn luyện LLMs một cách tối ưu về mặt tài nguyên và thời gian. Qua đó, mô hình có thể tận dụng tối đa kiến thức ban đầu từ dữ liệu lớn, đồng thời dễ dàng thích ứng với các nhiệm vụ cụ thể mới chỉ với một số lượng nhỏ tham số được cập nhật.



Hình 1: Ảnh minh họa về huấn luyện mô hình bằng tinh chỉnh đầy đủ (full fine-tuning), LoRA và QLoRA với các thông số VRAM được tham khảo tại đây.

Trong tutorial này, chúng ta sẽ áp dụng **LoRA** và **QLoRA** trên mô hình **Llama3.2** để fine-tuning cho bài toán phân tích cảm xúc tiếng Việt.

Theo đó, bài viết được bố cục như sau:

- **Phần I:** Giới thiệu về nội dung bài viết.
- **Phần II:** Tóm tắt về bài toán Parameter Efficient Fine-tuning (PEFT) và nhóm kỹ thuật Low-rank Adaptaion (LoRA).
- **Phần III:** Hướng dẫn cách cài đặt và huấn luyện một mô hình LLM ứng dụng các kỹ thuật LoRA.
- **Phần IV:** Trích dẫn tài liệu.

II. Low-rank Adaptation

Trước khi đi vào các hướng dẫn chi tiết, bài viết này sẽ giới thiệu về các kỹ thuật tinh chỉnh (fine-tuning) hiệu quả về tham số, đặc biệt tập trung vào hai phương pháp Low-rank Adaptation (LoRA) và Quantized LoRA (QLoRA), cho phép tinh chỉnh các mô hình ngôn ngữ lớn với tài nguyên hạn chế.

II.1. Tóm tắt về Parameter Efficient Fine-tuning

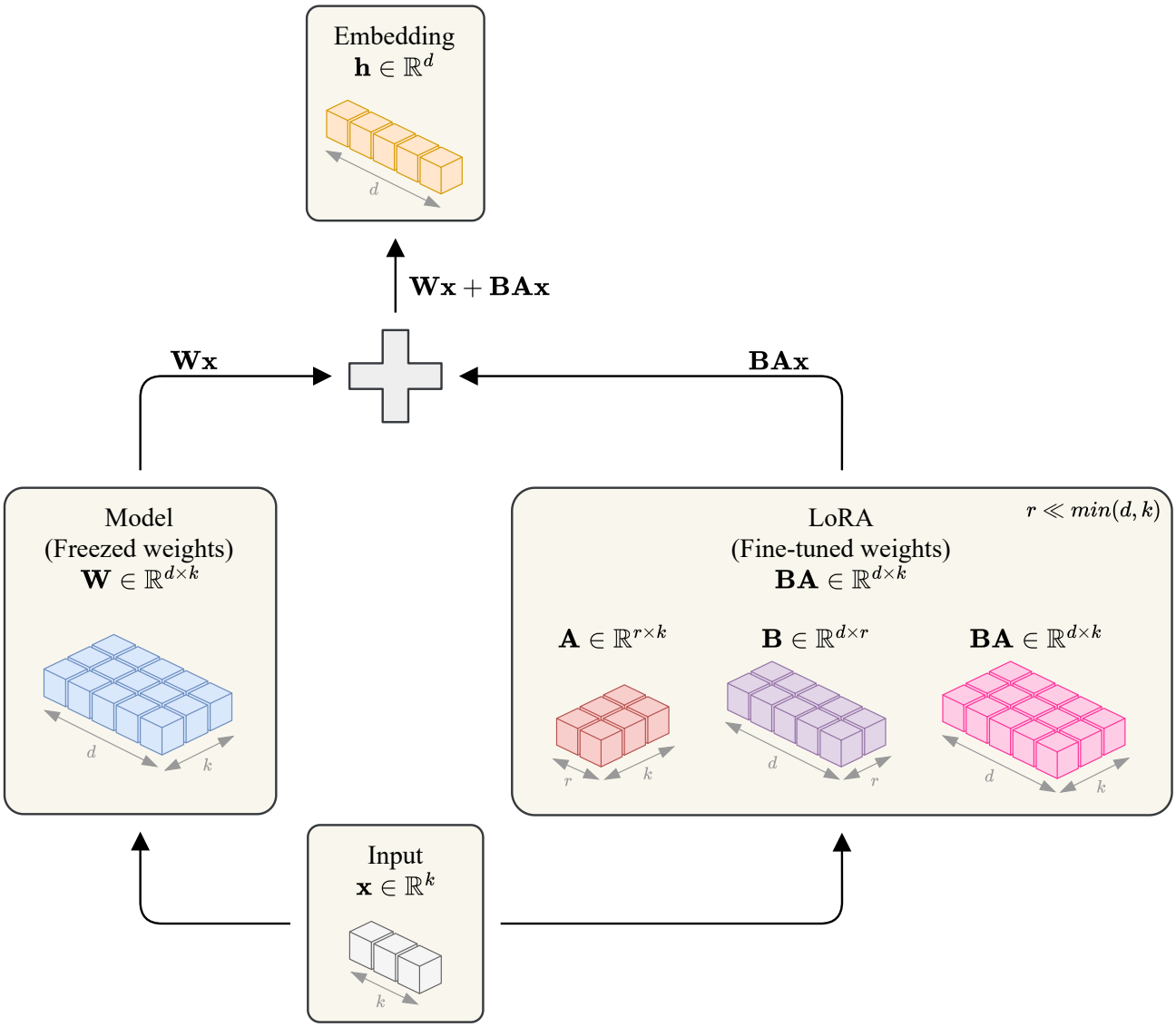
Thông thường, khi muốn tinh chỉnh một mô hình ngôn ngữ lớn (như GPT, LLaMa hay Mixtral) cho một nhiệm vụ cụ thể, ta phải huấn luyện lại toàn bộ tham số của mô hình. Việc này có nhiều nhược điểm như:

- Tốn rất nhiều tài nguyên, cần GPU mạnh, bộ nhớ lớn, thời gian huấn luyện lâu.
- Dễ bị quá khớp (overfitting) với dữ liệu huấn luyện, khiến mô hình khó tổng quát tốt.
- Lưu trữ không hiệu quả, mỗi phiên bản fine-tune đều phải lưu trữ toàn bộ mô hình gốc rất lớn.

Để giải quyết các vấn đề trên, người ta phát triển các kỹ thuật gọi chung là Parameter Efficient Fine-tuning (PEFT). Thay vì cập nhật tất cả các tham số, PEFT chỉ cập nhật một số lượng nhỏ tham số quan trọng. Nhờ vậy, mô hình giảm đáng kể tài nguyên tính toán và bộ nhớ, hạn chế quá khớp khi dữ liệu ít, và vẫn giữ được hiệu suất tốt như tinh chỉnh truyền thống. Một trong những kỹ thuật PEFT nổi bật nhất hiện nay chính là LoRA.

II.2. LoRA

Ý tưởng chính của LoRA xuất phát từ quan sát khi tinh chỉnh một mô hình lớn, ta không cần thay đổi toàn bộ ma trận trọng số khổng lồ của toàn mô hình. Thay vào đó, ta có thể biểu diễn sự thay đổi bằng tích của hai ma trận nhỏ hơn nhiều. Điều này giúp giảm đáng kể số lượng tham số cần cập nhật.



Hình 2: Minh hoạ về các thành phần trong LoRA.

Giả sử bạn có một ma trận trọng số ban đầu là $\mathbf{W} \in \mathbb{R}^{d \times k}$. Thay vì cập nhật trực tiếp $\mathbf{W}$, LoRA thêm vào một phần cập nhật nhỏ có dạng hạng thấp như sau:

$$\mathbf{W}' = \mathbf{W} + \mathbf{BA}$$

Trong đó:

- $\mathbf{W}'$ là ma trận trọng số sau khi áp dụng LoRA.
- $\mathbf{W}$ là ma trận trọng số gốc, không thay đổi trong quá trình tinh chỉnh mô hình.
- $\mathbf{B} \in \mathbb{R}^{d \times r}$ và $\mathbf{A} \in \mathbb{R}^{r \times k}$ hai ma trận có hạng thấp (low-rank) với các giá trị trong $\mathbf{B}$ được khởi tạo bằng 0, trong khi các giá trị trong $\mathbf{A}$ được khởi tạo ngẫu nhiên bằng Gaussian ($\mathbf{A} \sim \mathcal{N}(0, \sigma^2)$). Rank $r \ll min(d, k)$ là hạng được chọn trước, thường nhỏ hơn nhiều so với kích thước của ma trận gốc. Hai ma trận này sẽ được học trong quá trình tinh chỉnh.

Để kiểm soát mức độ ảnh hưởng của phần cập nhật này, ta thêm vào một tỷ lệ giữa hệ số điều chỉnh là alpha (α) và rank (r):

$$\mathbf{W}' = \mathbf{W} + \left(\frac{\alpha}{r}\right) \times \mathbf{BA}$$

Ví dụ:

- Khi $\alpha = r$, LoRA có ảnh hưởng bình thường đến $\mathbf{W}$.
- Nếu $\alpha > r$ thì LoRA ảnh hưởng mạnh hơn, giúp mô hình học nhanh hơn nhưng có thể bị quá khớp.
- Nếu $\alpha < r$ thì tác động của LoRA yếu hơn, giúp mô hình ổn định hơn nhưng có thể học chậm hơn.

Với kỹ thuật này, LoRA giúp tiết kiệm bộ nhớ đáng kể vì chỉ cần lưu trữ hai ma trận nhỏ thay cho toàn bộ mô hình khổng lồ, dẫn tới việc huấn luyện nhanh sẽ hơn do ít tham số cần cập nhật hơn, và đặc biệt là vẫn giữa được hiệu suất cạnh tranh với việc tinh chỉnh toàn bộ.

II.3. QLoRA

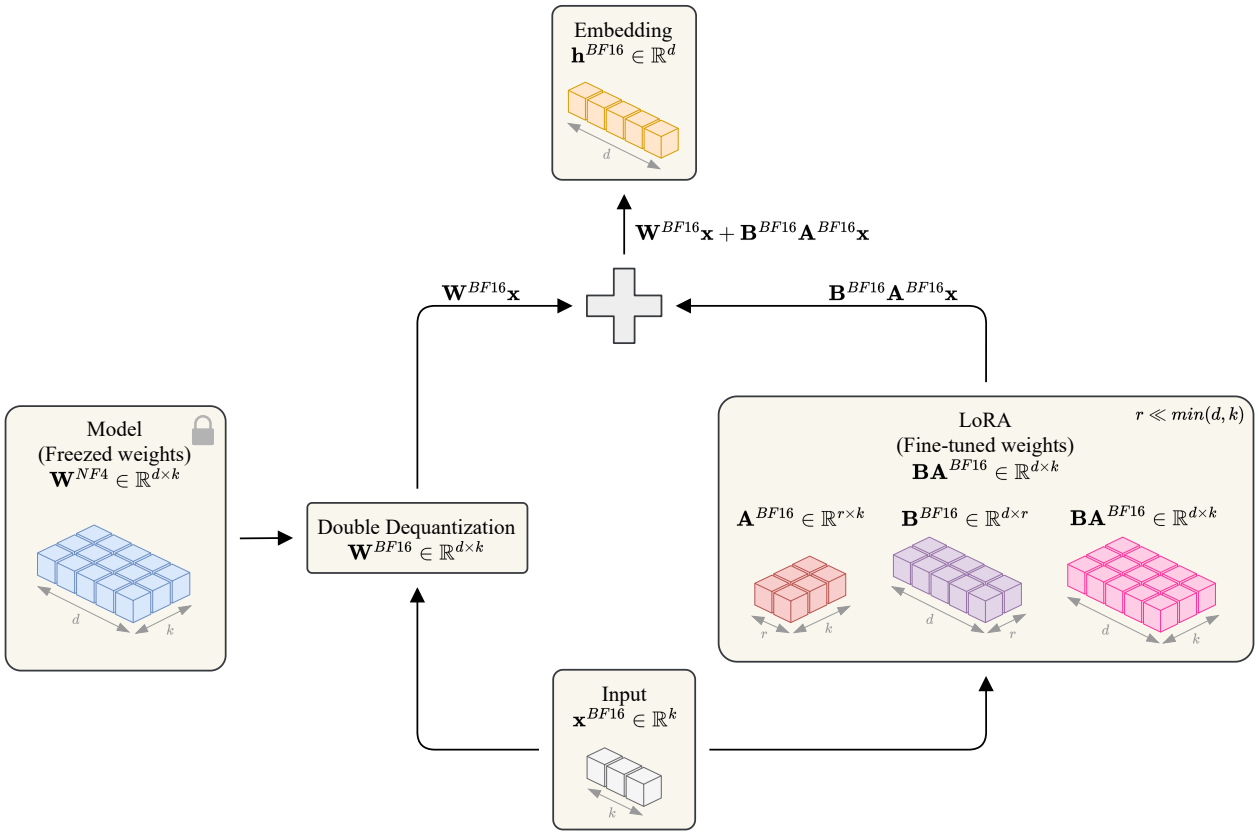
Mặc dù LoRA đã giảm đáng kể tài nguyên cần thiết cho việc tinh chỉnh mô hình, QLoRA đẩy hiệu quả này lên một tầm cao mới bằng cách kết hợp LoRA với kỹ thuật lượng tử hóa (quantization).

Trong đó, lượng tử hóa là quá trình chuyển một giá trị có độ chính xác cao (ví dụ như 32-bit) thành một giá trị có độ chính xác thấp hơn (ví dụ như 4-bit, 8-bit) để tiết kiệm bộ nhớ và tăng tốc tính toán. Ngược lại, giải lượng tử hóa (dequantization) là quá trình khôi phục lại các giá trị gốc gần đúng từ giá trị lượng tử hóa.

Đối với quá trình lượng tử hoá của QLoRA, hai kĩ thuật sau được áp dụng để tối ưu quá trình lượng tử hoá:

1. Blockwise k-bit quantization:
    - Trọng số ban đầu của layer $\mathbf{W}$ được chia nhỏ thành các khối (blocks) có kích thước cố định (ví dụ: 64 phần tử).
    - Mỗi khối sẽ được chuẩn hóa độc lập bằng cách chia cho giá trị tuyệt đối lớn nhất của khối, đưa tất cả các giá trị trong mỗi khối về khoảng [-1, 1].
    - Sau khi chuẩn hóa, mỗi khối được lượng tử hóa riêng biệt xuống mức k-bit (thường là 4-bit), giúp hạn chế tác động tiêu cực từ các giá trị ngoại lai (outliers) và nâng cao độ chính xác của lượng tử hóa.
    - Ví dụ minh họa: Giả sử ta có một ma trận trọng số $\mathbf{W}$ kích thước 8×8, ta trải phẳng thành vector một chiều 64 phần tử và chia thành 4 khối, mỗi khối có 16 phần tử. Sau đó mỗi khối được chuẩn hóa và lượng tử hóa độc lập với độ chính xác 4-bit. Kỹ thuật này giúp giảm đáng kể tác động tiêu cực từ outliers và tăng độ chính xác khi lượng tử hóa.
2. NormalFloat (NF4):

- NF4 là kỹ thuật lượng tử hóa 4-bit được tối ưu đặc biệt cho phân phối chuẩn thường gặp trong trọng số mạng nơ-ron.
- Các mức lượng tử hóa NF4 được xác định dựa vào các điểm phân vị (quantiles) của phân phối chuẩn, từ đó phân bố nhiều mức lượng tử hơn quanh giá trị trung bình (0), nơi mà phần lớn các trọng số mô hình tập trung.
- Điều này cho phép NF4 biểu diễn chính xác hơn các giá trị trọng số quanh giá trị trung bình, làm giảm sai số lượng tử hóa và tăng hiệu suất mô hình sau khi lượng tử hóa.
- Giả sử ta có một tập trọng số theo phân phối chuẩn, ta sẽ xác định 16 mức lượng tử NF4 dựa trên các phân vị của phân phối chuẩn này, sau đó chuẩn hóa trọng số về khoảng [-1, 1] và lượng tử hóa trọng số về mức NF4 gần nhất. Kỹ thuật này đảm bảo rằng trọng số quanh 0 được biểu diễn chính xác hơn, tối ưu hóa hiệu quả lượng tử hóa.



Hình 3: Minh hoạ về các thành phần trong QLoRA.

Ý tưởng của QLoRA là kết hợp hai chiến lược sau:

1. Nén trọng số ban đầu $\mathbf{W}$ của mô hình xuống dạng lượng tử hóa 4-bit (thường là NF4 - NormalFloat4) để tiết kiệm bộ nhớ.

2. Chỉ tinh chỉnh các ma trận của LoRA thay vì cập nhật toàn bộ trọng số gốc $\mathbf{W}$. Trong quá trình này, trọng số gốc đã được nén $\mathbf{W}$ của mô hình được giải lượng tử hóa về dạng có

độ chính xác cao hơn là 16-bit (thường là BF16 - BrainFloat16) rồi cộng thêm phần điều chỉnh từ LoRA để được trọng số sau khi áp dụng QLoRA ($\mathbf{W}'$).

Như vậy, trọng số sau khi áp dụng QLoRA ($\mathbf{W}'$) được biểu diễn bởi công thức:

$$\mathbf{W}' = \underbrace{\text{doubleDequant}(\mathbf{W})}_{\substack{\text{phần trọng số gốc (4-bit NF4)} \\ \text{đã giải nén ra BF16}}} + \underbrace{\frac{\alpha}{r}\mathbf{BA}}_{\text{phần LoRA}}$$

Với chiến lược đầu tiên, QLoRA sẽ lượng tử hoá các giá trị trong $\mathbf{W}$ về dạng có độ chính xác thấp hơn nhưng tiết kiệm bộ nhớ hơn. Lấy ví dụ cho trường hợp đưa 32-bit Floating Point (FP32) xuống 8-bit Int8 (có phạm vi biểu diễn từ $[-127, 127]$), giả sử chúng ta có một tensor X^{FP32} dưới dạng số thực 32-bit:

$$X^{FP32} = [0.3, -0.2, 1.5, -0.8]$$

- **Bước 1: Tính toán hằng số lượng tử hoá c (quantization constant)**

  Hằng số lượng tử hoá được tính công thức:

  $$c = \frac{127}{\text{absmax}(X^{FP32})} = \frac{127}{1.5} = 84.67$$

  Trong đó, $\text{absmax}(X^{FP32})$ là giá trị tuyệt đối lớn nhất trong tensor X^{FP32}.

- **Bước 2: Thực hiện lượng tử hoá**

  Tensor X^{FP32} sẽ được nén thành X^{Int8} theo công thức:

  $$X^{Int8} = \text{round}(c \times X^{FP32})$$

  Áp dụng công thức này cho từng phần tử của X^{FP32}:

  $$X^{Int8} = \text{round}(84.67 \times [0.3, -0.2, 1.5, -0.8]) = [25, -17, 127, -68]$$

- **Bước 3: Thực hiện giải lượng tử hóa**

  Để khôi phục lại giá trị ban đầu, ta thực hiện dequantization bằng công thức:

  $$X^{FP32} = \text{dequant}(c^{\text{FP32}}, X^{\text{Int8}}) = \frac{X^{Int8}}{c}$$

  Áp dụng công thức này cho X^{Int8}:

  $$X^{FP32} = \frac{[25, -17, 127, -68]}{84.67} = [0.295, -0.201, 1.5, -0.803]$$

Để tiết kiệm bộ nhớ hơn nữa, QLoRA lượng tử hoá cả hằng số lượng tử hoá c, do đó trong **chiến lược thứ hai**, ta cần thực hiện 2 lần giải lượng tử hoá (double dequantization), với trước

tiên là giải lượng tử hóa cho hằng số c, sau đó dùng hằng số c này để giải lượng tử hóa cho thành phần còn lại.

$$X^{\text{BF16}} = \text{doubleDequant}(c_1^{\text{FP32}}, c_2^{k\text{-bit}}, X^{k\text{-bit}}) = \text{dequant}(\text{dequant}(c_1^{\text{FP32}}, c_2^{8\text{-bit}}), X^{4\text{-bit}})$$

Với chiến lược như thế, QLoRA đại diện cho một bước tiến đáng kể so với LoRA về sử dụng hiệu quả bộ nhớ. Với những đổi mới kỹ thuật như 4-bit NormalFloat, Double Quantization và Paged Optimizers (không được đề cập trong bài viết này, đọc thêm tại đây [1]), QLoRA đã mở ra khả năng tinh chỉnh các mô hình cực lớn trên phần cứng thông thường, giúp ích cho dự án nghiên cứu hoặc ứng dụng thực tế với nguồn lực hạn chế tiếp cận được các mô hình mô hình ngôn ngữ có kích thước lớn.

III. Cài đặt chương trình

III.1. Dataset

Bộ dữ liệu UIT-VSFC: Vietnamese Students' Feedback Corpus for Sentiment Analysis [2] gồm hơn 16.000 câu phản hồi từ sinh viên về nhiều khía cạnh của quá trình học tập và giảng dạy. Các phản hồi này được thu thập từ các khảo sát sinh viên tại một trường đại học Việt Nam trong khoảng thời gian từ năm 2013 đến 2016.

Bộ dữ liệu được gán nhãn theo hai nhiệm vụ:

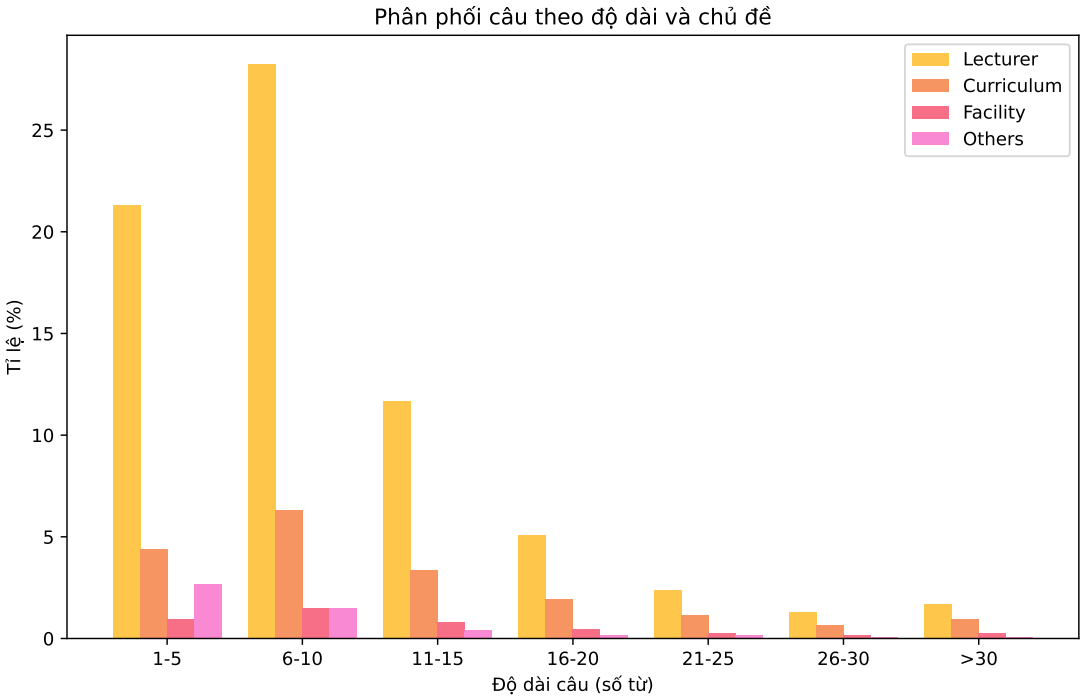
1. Phân tích cảm xúc (Sentiment-based classification): Các câu phản hồi được phân loại thành Positive (tích cực), Negative (tiêu cực), và Neutral (trung lập).
2. Phân loại theo chủ đề (Topic-based classification): Các câu phản hồi được chia thành bốn nhóm chính:
    - Lecturer (Giảng viên).
    - Curriculum (Chương trình học).
    - Facility (Cơ sở vật chất).
    - Others (Khác).

Sentiment/Topic	**Positive (%)**	**Negative (%)**	**Neutral (%)**	**Total (%)**
Lecturer	33.57	25.38	1.81	71.76
Curriculum	3.40	14.39	1.00	18.79
Facility	0.11	4.21	0.08	4.40
Others	1.61	2.01	1.43	5.04
Total	49.69	45.99	4.32	100.00

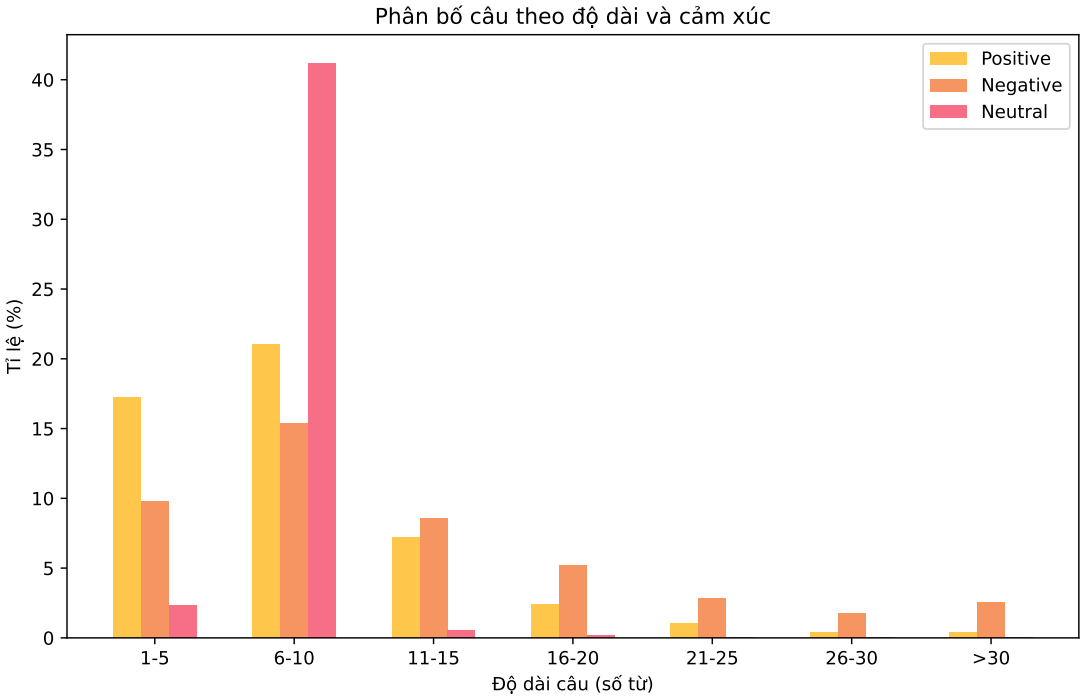
Bảng 1: Bảng thống kê giữa Chủ đề và Cảm xúc.

Tập dữ liệu	**Tỷ lệ (%)**	**Số lượng câu**
Tập huấn luyện	70%	11.426
Tập phát triển	10%	1.538
Tập kiểm thử	20%	3.166
Tổng	100%	16.130

Bảng 2: Bảng thống kê các tập dữ liệu.



Hình 4: Biểu đồ phân phối độ dài câu theo các chủ đề.



Hình 5: Biểu đồ phân phối độ dài câu theo các cảm xúc.

Trong đó, một số đặc điểm của bộ dữ liệu này gồm:

- Đa số là câu ngắn (1-15 từ chiếm hơn 80%).
- Chứa nhiều từ viết tắt, biểu tượng cảm xúc đặc trưng của sinh viên (ví dụ: "ok hết .", "nhiệt tình , cô dễ thương =) .", "đánh giá cao colonsmile .").
- Có sự mất cân bằng giữa các lớp, đặc biệt là lớp trung tính và chủ đề cơ sở vật chất.

III.2. Cài đặt mã nguồn

Ở mục này, chúng ta sẽ chạy code với Jupyter Notebook (các notebook được cung cấp ở phần V. Phụ Lục. Bài hướng dẫn dưới đây sẽ sử dụng file `1. LoRA_SentimentAnalysis_Llama3.2-1B.ipynb`.

III.2.1. Cài đặt và sử dụng thư viện

Đầu tiên, ta cài đặt các thư viện thông qua pip như sau:

```
!pip install -qq --upgrade pip
!pip install -qq --upgrade peft transformers accelerate bitsandbytes datasets
                                    trl huggingface_hub evaluate
```

Sau đó, nếu bạn dùng Google Colab, chúng ta sẽ đăng nhập vào HuggingFace thông qua việc setup HuggingFace token (link) và chạy code cell dưới đây.

```
from google.colab import userdata
from huggingface_hub import login

login(token=userdata.get('HF_TOKEN'))
```

Cuối cùng, chúng ta sẽ import các module cần dùng từ các thư viện đã cài đặt.

```
import os
import torch
import numpy as np
import evaluate

from peft import PeftModel, PeftConfig, LoraConfig, TaskType, get_peft_model,
                                    get_peft_config
from transformers import AutoModelForCausalLM, AutoTokenizer
from transformers import DataCollatorForLanguageModeling, Trainer,
                                    TrainingArguments
from datasets import load_dataset
from trl import SFTTrainer
import warnings

warnings.filterwarnings("ignore")
```

III.2.2. Hyper-parameters

Chúng ta sẽ thiết lập các hyper-parameter và cấu hình cho huấn luyện, bao gồm tên mô hình sử dụng (trong bài là `Llama-3.2-1B-Instruct`), vị trí thư mục cache, số bước huấn luyện tối đa, số bước đánh giá và các tham số huấn luyện khác như `batch_size`.

```
base_model_id = "meta-llama/Llama-3.2-1B-Instruct"
cache_dir = "./cache"

MAX_TRAIN_STEPS = 5_000
NUM_EVAL_STEPS = 500
MAX_TRAIN_SAMPLES = 20_000
MAX_EVAL_SAMPLES = 2_000

training_args = TrainingArguments(
    output_dir="./output",
    # num_train_epochs=1,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=8,
    logging_dir="./logs",
    logging_steps=10,
    save_steps=NUM_EVAL_STEPS,
    max_steps=MAX_TRAIN_STEPS,
    eval_steps=NUM_EVAL_STEPS,
    eval_strategy="steps",
    overwrite_output_dir=True,
    save_total_limit=2,
    report_to="none",
    push_to_hub=False,
)
```

III.2.3. Load backbone model

Tiếp theo, chúng ta tải mô hình Llama 3.2 1B Instruct đã được và tokenizer tương ứng từ Hugging Face. Đồng thời, ta chuyển mô hình đã load sang GPU CUDA nếu có, hoặc sử dụng CPU nếu không có.

```
base_model = AutoModelForCausalLM.from_pretrained(base_model_id,
                                        trust_remote_code=True, torch_dtype=
                                        torch.bfloat16, cache_dir=cache_dir)
tokenizer = AutoTokenizer.from_pretrained(base_model_id, trust_remote_code=
                                        True, cache_dir=cache_dir)
base_model = base_model.to('cuda' if torch.cuda.is_available() else 'cpu')
```

Cài đặt padding token là EOS token để khi train, mô hình sẽ dùng EOS token làm padding token, đảm bảo quá trình parallel training.

```
if tokenizer.pad_token is None or tokenizer.pad_token_id is None:
    print("Pad token is not set. Setting it to EOS token.")
    tokenizer.pad_token = tokenizer.eos_token
    tokenizer.pad_token_id = tokenizer.eos_token_id
else:
    print(f'Pad token: {tokenizer.pad_token}')
    print(f'Pad token id: {tokenizer.pad_token_id}')
```

```

print(f'EOS token: {tokenizer.eos_token}')
print(f'EOS token id: {tokenizer.eos_token_id}')
```

Cuối cùng là cài đặt instruction template. Ở bài này, chúng ta sử dụng chat template của Llama 3.2 1B Instruct.

```
if tokenizer.chat_template is None:
    tokenizer.chat_template = """{{- bos_token }}
{%- if not date_string is defined %}
    {%- if strftime_now is defined %}{%- set date_string = strftime_now("%d %b %Y") %}{%- else %}{%- set date_string = "26 Jul 2024" %}{%- endif %}
{%- endif %}

{#- This block extracts the system message, so we can slot it into the right place. #}
{%- if messages[0]['role'] == 'system' %}
    {%- set system_message = messages[0]['content']|trim %}
    {%- set messages = messages[1:] %}
{%- else %}
    {%- set system_message = "" %}
{%- endif %}

{#- System message #}
{{- "<|start_header_id|>system<|end_header_id|>\n\n" }}
{{- "Cutting Knowledge Date: December 2023\n" }}
{{- "Today Date: " + date_string + "\n\n" }}
{{- system_message }}
{{- "<|eot_id|>" }}

{%- for message in messages %}
    {{- '<|start_header_id|>' + message['role'] + '<|end_header_id|>\n\n'+ message['content'] | trim + '<|eot_id|>' }}
{%- endfor %}
{%- if add_generation_prompt %}
    {{- '<|start_header_id|>assistant<|end_header_id|>\n\n' }}
{%- endif %}
"""
```

Để minh hoạ cách chat template hoạt động, ta đưa vào một đoạn conversation sau và áp dụng chat template:

```
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Hello, how are you?"},
    {"role": "assistant", "content": "I'm just a computer program, but I'm here to help you!"},
]
print(tokenizer.apply_chat_template(conversation=messages, tokenize=False), end="\n\n")
```

Kết quả khi chạy apply chat template cho conversation ví dụ là:

<|begin_of_text|><|start_header_id|>system<|end_header_id|>

Cutting Knowledge Date: December 2023
Today Date: 28 Feb 2025

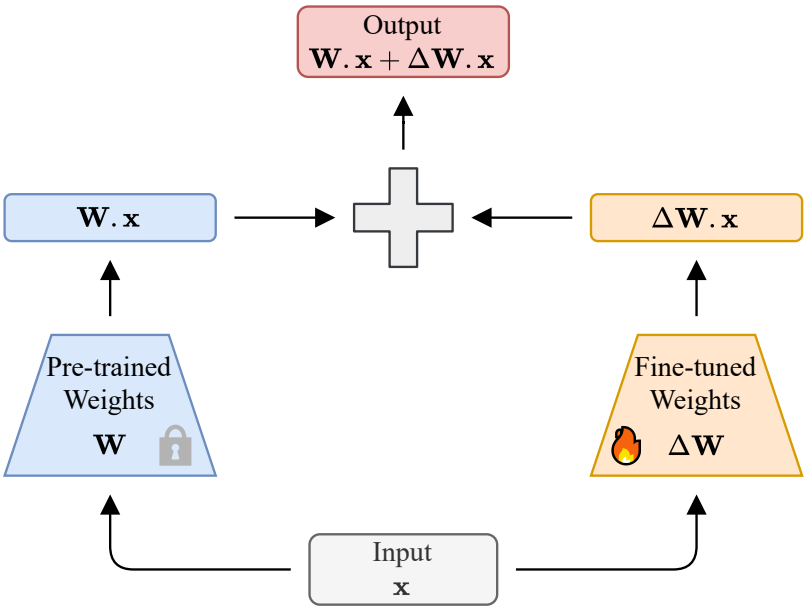
You are a helpful assistant.<|eot_id|><|start_header_id|>user<|end_header_id|>

Hello, how are you?<|eot_id|><|start_header_id|>assistant<|end_header_id|>

I'm just a computer program, but I'm here to help you!<|eot_id|>

III.2.4. Áp dụng LoRA vào mô hình backbone

Ta có thể sử dụng thư viện PEFT để áp dụng LoRA vào các mô hình ngôn ngữ lớn, phục vụ cho việc huấn luyện trên các GPU có ít memory (minh họa ở Hình 6).



Hình 6: Trực quan LoRA.

Để áp dụng LoRA vào mô hình đã load và xem tỉ lệ các tham số mới được thêm vào so với tổng tham số, ta chạy code cell dưới đây:

```
peft_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,
    inference_mode=False,
    r=8,
    lora_alpha=32,
    lora_dropout=0.1
)
peft_model = get_peft_model(base_model, peft_config)
peft_model.print_trainable_parameters()
```

Đối với mô hình Llama 3.2 1B Instruct, ta có được kết quả sau khi áp dụng LoRA:

```
trainable params: 851,968 || all params: 1,236,666,368 || trainable%: 0.0689
```

III.2.5. Load dữ liệu huấn luyện

Ta tải bộ dữ liệu `vietnamese_students_feedback` từ Hugging Face (thông tin về bộ dữ liệu xem tại phần V. Phụ Lục) và lấy một phần nhỏ để huấn luyện và đánh giá mô hình (20.000 mẫu để train và 2.000 để test).

```
dataset = load_dataset("uitnlp/vietnamese_students_feedback", cache_dir=
                                        cache_dir)

for split in dataset:
    if split == "train":
        MAX_TRAIN_SAMPLES = min(MAX_TRAIN_SAMPLES, len(dataset[split]))
        dataset[split] = dataset[split].select(range(MAX_TRAIN_SAMPLES))
    else:
        MAX_EVAL_SAMPLES = min(MAX_EVAL_SAMPLES, len(dataset[split]))
        dataset[split] = dataset[split].select(range(MAX_EVAL_SAMPLES))
    print(f"{split}: {len(dataset[split])}")
```

Tiếp theo, chúng ta sẽ trích xuất các labels từ tập dữ liệu của mình:

```
label_set = set([item["sentiment"] for split in dataset for item in dataset[
                                        split]])
all_labels = dataset['train'].features['sentiment'].names

label2id = {label: i for i, label in enumerate(all_labels)}
id2label = {i: label for i, label in enumerate(all_labels)}
```

Chúng ta sẽ huấn luyện mô hình thông qua format hỏi như sau:

```
USER_PROMPT_TEMPLATE = """Predict the sentiment of the following input
                                        sentence.
The response must begin with "Sentiment: ", followed by one of these keywords:
                                         "positive", "negative", or "neutral",
                                        to reflect the sentiment of the input
                                        sentence.

Sentence: {input}"""
```

Sau đó, chúng ta sẽ chuẩn bị dữ liệu huấn luyện bằng cách apply prompt template và tokenize trước khi đưa dữ liệu vào mô hình:

```
def tokenize_function(examples):
    results = {
        "input_ids": [],
        "labels": [],
        "attention_mask": [],
    }

    for i in range(len(examples['sentence'])):
        cur_input = examples['sentence'][i]
        cur_output_id = examples['sentiment'][i]

```

```
        cur_prompt = USER_PROMPT_TEMPLATE.format(input=cur_input)
        cur_output = id2label[cur_output_id]

        input_messages = [
            {"role": "system", "content": "You are a helpful assistant. You must fulfill the user request."},
            {"role": "user", "content": cur_prompt},
        ]
        input_output_messages = input_messages + [{"role": "assistant", "content": f"Sentiment: {cur_output}"}]

        input_prompt_tokenized = tokenizer.apply_chat_template(conversation=input_messages, return_tensors="pt", add_generation_prompt=True)[0]
        input_output_prompt_tokenized = tokenizer.apply_chat_template(conversation=input_output_messages, return_tensors="pt")[0]

        input_ids = input_output_prompt_tokenized
        label_ids = torch.cat([
            torch.full_like(input_prompt_tokenized, fill_value=-100),
            input_output_prompt_tokenized[len(input_prompt_tokenized):]
        ])

        assert len(input_ids) == len(label_ids)

        results["input_ids"].append(input_ids)
        results["labels"].append(label_ids)
        results['attention_mask'].append(torch.ones_like(input_ids))

    return results

col_names = dataset['train'].column_names
tokenized_dataset = dataset.map(
    tokenize_function,
    batched=True,
    remove_columns=col_names,
    num_proc=os.cpu_count(),
)
tokenized_dataset
```

Để kiểm tra input format có đúng hay không, ta sẽ decode một mẫu dữ liệu đã được mã hoá:

```
print(tokenizer.decode(tokenized_dataset['train'][0]['input_ids'], skip_special_tokens=False))
```

và thu được kết quả như sau:

<|begin_of_text|><|start_header_id|>system<|end_header_id|>

Cutting Knowledge Date: December 2023
Today Date: 27 Feb 2025

You are a helpful assistant. You must fulfill the user request.<|eot_id|><|start_header_id|>user<|end_header_id|>

Predict the sentiment of the following input sentence.
The response must begin with "Sentiment: ", followed by one of these keywords: "positive", "negative", or "neutral", to reflect the sentiment of the input sentence.

Sentence: slide giáo trình đầy đủ.<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Sentiment: positive<|eot_id|>

III.2.6. Tạo data collator

Trong quá trình huấn luyện, các mẫu dữ liệu sẽ có độ dài không đồng đều. Do đó, ta cần phải sử dụng padding token để các câu có độ dài bằng nhau.

```python
class RightPaddingDataCollator(DataCollatorWithPadding):
    """The default data collator pads only inputs, not including the labels."""

    def __init__(self, tokenizer, max_length: int = 1024):
        super().__init__(tokenizer, max_length=max_length)

    def __call__(self, features: List[Dict[str, Any]]) -> Dict[str, Any]:
        input_ids, labels, attention_mask = [], [], []
        max_batch_len = max(len(f["input_ids"]) for f in features)

        for sample in features:
            # Convert to torch tensors
            cur_input_ids = torch.tensor(sample["input_ids"], dtype=torch.long)
            cur_labels = torch.tensor(sample["labels"], dtype=torch.long)
            cur_attention_mask = torch.ones_like(cur_input_ids)

            # Next, we pad the inputs and labels to the maximum length within the batch
            pad_token_id = self.tokenizer.pad_token_id
            padding_length = max_batch_len - len(cur_input_ids)
            cur_input_ids = torch.cat([cur_input_ids, torch.full((padding_length,), fill_value=pad_token_id, dtype=torch.long)])
            cur_labels = torch.cat([cur_labels, torch.full((padding_length,), fill_value=-100, dtype=torch.long)])
            cur_attention_mask = torch.cat([cur_attention_mask, torch.zeros((padding_length,), dtype=torch.long)])

```

```
            # Truncate the inputs and labels to the maximum length
            cur_input_ids = cur_input_ids[:max_batch_len]
            cur_labels = cur_labels[:max_batch_len]
            cur_attention_mask = cur_attention_mask[:max_batch_len]

            # Append to the return lists
            input_ids.append(cur_input_ids)
            labels.append(cur_labels)
            attention_mask.append(cur_attention_mask)

        # Return formatted batch.
        return {
            "input_ids": torch.stack(input_ids),
            "labels": torch.stack(labels),
            "attention_mask": torch.stack(attention_mask)
        }


data_collator = RightPaddingDataCollator(tokenizer)
```

III.2.7. Metrics

Ta cài đặt các metrics đánh giá (accuracy, F1, precision, recall) và define một hàm để tính toán các metrics này trong quá trình train và test.

```
accuracy_metric = evaluate.load("accuracy")
f1_metric = evaluate.load("f1")
precision_metric = evaluate.load("precision")
recall_metric = evaluate.load("recall")


def preprocess_logits_for_metrics(logits, labels):
    if isinstance(logits, tuple):
        logits = logits[0]
    return logits.argmax(dim=-1)


def compute_metrics(eval_preds):
    preds, labels = eval_preds

    if isinstance(preds, tuple):
        preds = preds[0]

    idx = 0
    for i in range(len(labels[0])):
        if labels[0][i] == -100:
            idx = i
        else:
            break
    # Slice the labels and preds to remove the prompt tokens
    preds = preds[:, idx:]

    # Replace -100 in the preds as we can't decode them
    preds = np.where(preds != -100, preds, tokenizer.pad_token_id)
```

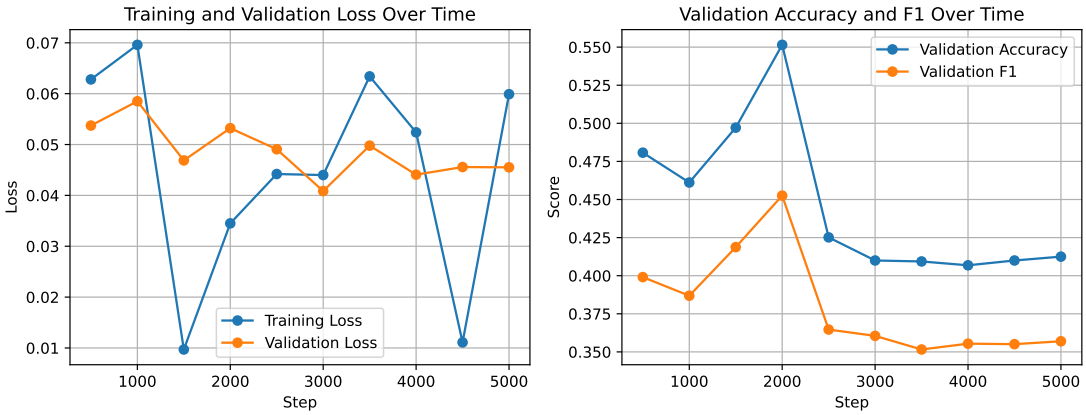
```
30
31      processed_preds = []
32      for pred in preds:
33          end_pred_idx = np.where(pred == tokenizer.eos_token_id)[0]
34          if len(end_pred_idx) > 0:
35              end_pred_idx = end_pred_idx[0]
36              processed_preds.append(pred[:end_pred_idx])
37          else:
38              processed_preds.append(pred)
39
40      # Decode generated summaries into text
41      decoded_preds = tokenizer.batch_decode(processed_preds,
                                               skip_special_tokens=True)
42
43      # Replace -100 in the labels as we can't decode them
44      labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
45
46      # Decode reference summaries into text
47      decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)
48
49      # Convert the decoded predictions and labels to label ids
50      int_preds, int_labels = [], []
51      for p, l in zip(decoded_preds, decoded_labels):
52          l = l.split(":")[-1].strip()
53          cur_label_id = label2id[l]
54          int_labels.append(cur_label_id)
55          try:
56              p = p.split(":")[-1].strip()
57              cur_pred_id = label2id[p]
58          except Exception as e:
59              cur_pred_id = (cur_label_id + 1) % len(label2id)
60          int_preds.append(cur_pred_id)
61
62      accuracy_results = accuracy_metric.compute(predictions=int_preds,
                                               references=int_labels)
63      f1_results = f1_metric.compute(predictions=int_preds, references=
                                               int_labels, average="macro")
64      precision_results = precision_metric.compute(predictions=int_preds,
                                               references=int_labels, average="macro")
65      recall_results = recall_metric.compute(predictions=int_preds, references=
                                               int_labels, average="macro")
66
67      return {
68          **accuracy_results,
69          **f1_results,
70          **precision_results,
71          **recall_results
72      }
```

III.2.8. Huấn luyện mô hình

Ta sử dụng `SFTTrainer` của thư viện trl và chạy `trainer.train()` để huấn luyện mô hình. Code cài đặt như sau:

```
trainer = SFTTrainer(
    model=peft_model,
    args=training_args,
    train_dataset=tokenized_dataset['train'],
    eval_dataset=tokenized_dataset['validation'],
    preprocess_logits_for_metrics=preprocess_logits_for_metrics,
    compute_metrics=compute_metrics,
    processing_class=tokenizer,
    data_collator=data_collator,
)
trainer.train()
```

Kết quả huấn luyện được hiển thị ở Hình 7:



Hình 7: Kết quả huấn luyện mô hình Llama 3.2 1B Instruct với LoRA trên bộ dữ liệu UIT-VSFC.

Chúng ta tiếp tục đánh giá mô hình trên tập test như sau:

```
# Evaluate the model on the test set
trainer.evaluate(tokenized_dataset['test'])
```

và thu được kết quả

```
{'eval_loss': 0.05903154984116554,
 'eval_accuracy': 0.45356917245735945,
 'eval_f1': 0.3842498243276324,
 'eval_precision': 0.449602388563955,
 'eval_recall': 0.40293078144740546,
 'eval_runtime': 18.9385,
 'eval_samples_per_second': 83.586,
 'eval_steps_per_second': 10.455}
```

III.2.9. Inference

Để sử dụng mô hình, ta thiết kế hai hàm sau:

```
def inference(model, tokenizer, input_sentence):
    tokenizer.pad_token_id = tokenizer.eos_token_id

    user_prompt = USER_PROMPT_TEMPLATE.format(input=input_sentence)
```

```
    messages = [
        {"role": "system", "content": "You are a helpful assistant. You must fulfill the user request."},
        {"role": "user", "content": user_prompt},
    ]
    input_prompt = tokenizer.apply_chat_template(conversation=messages, add_generation_prompt=True, tokenize=False)
    inputs = tokenizer(input_prompt, return_tensors="pt", add_special_tokens=False)
    inputs = {k: v.to(model.device) for k, v in inputs.items()}

    output_ids = model.generate(**inputs, max_new_tokens=16, pad_token_id=tokenizer.eos_token_id)
    output_ids = output_ids[:, inputs['input_ids'][0].shape[-1]:output_ids.shape[-1]]
    results = tokenizer.batch_decode(output_ids, skip_special_tokens=True)
    return results[0]

def batch_inference(model, tokenizer, input_sentences):
    tokenizer.padding_side = "left"
    tokenizer.pad_token_id = tokenizer.eos_token_id

    user_prompts = [USER_PROMPT_TEMPLATE.format(input=input_sentence) for input_sentence in input_sentences]
    messages_list = [
        [
            {"role": "system", "content": "You are a helpful assistant. You must fulfill the user request."},
            {"role": "user", "content": user_prompt},
        ]
        for user_prompt in user_prompts
    ]
    input_prompts = [tokenizer.apply_chat_template(conversation=messages, add_generation_prompt=True, tokenize=False) for messages in messages_list]

    inputs = tokenizer(input_prompts, return_tensors="pt", padding=True, add_special_tokens=False)
    inputs = {k: v.to(model.device) for k, v in inputs.items()}

    output_ids = model.generate(**inputs, max_new_tokens=16, pad_token_id=tokenizer.eos_token_id)
    output_ids = output_ids[:, inputs['input_ids'][0].shape[-1]:output_ids.shape[-1]]
    results = tokenizer.batch_decode(output_ids, skip_special_tokens=True)
    return results
```

Ta có thể đánh giá sentiment của một câu thông qua hàm `inference` hay nhiều câu cùng một lúc với hàm `batch_inference`.

```
inference(peft_model, tokenizer, "The weather is nice today.")
# Output: 'Sentiment: positive'

batch_inference(peft_model, tokenizer, ["I love this product.", "I hate this product. It is because the quality is
```

```
                                        extremely bad."])
# Output:  ['Sentiment: positive', 'Sentiment: negative']
```

III.2.10. Đánh giá mô hình với few-shot learning

Ngoài việc huấn luyện mô hình với LoRA, ta có thể đánh giá mô hình gốc sử dụng few-shot learning:

```
USER_FEWSHOT_PROMPT_TEMPLATE = """Predict the sentiment of the following input
                                        sentence.
The response must begin with "Sentiment: ", followed by one of these keywords:
                                        "positive", "negative", or "neutral",
                                        to reflect the sentiment of the input
                                        sentence.

Here are a few examples:

{few_shot_examples}

Sentence: {input}"""

def evaluate_few_shot(model, tokenizer, eval_dataset, few_shot_examples,
                                        batch_size=8, print_example=False):
    model.eval()
    all_predictions = []
    all_labels = []

    # Format the few-shot examples for the prompt
    formatted_few_shot_examples = ""
    for i, example in enumerate(few_shot_examples):
        formatted_few_shot_examples += f"Sentence: {example['sentence']}\
                                        nSentiment: {id2label[example['
                                        sentiment']]}\n"
        if i < len(few_shot_examples) - 1:
            formatted_few_shot_examples += "\n"


    for i in range(0, len(eval_dataset), batch_size):
        batch = eval_dataset[i:i + batch_size]

        # Replace USER_PROMPT_TEMPLATE with USER_FEWSHOT_PROMPT_TEMPLATE
        user_prompts = [USER_FEWSHOT_PROMPT_TEMPLATE.format(input=sentence,
                                        few_shot_examples=
                                        formatted_few_shot_examples) for
                                        sentence in batch['sentence']]

        messages_list = [
            [
                {"role": "system", "content": "You are a helpful assistant.
                                        You must fulfill the user request."},
                {"role": "user", "content": user_prompt},
            ]
            for user_prompt in user_prompts
        ]
```

```
36
37          input_prompts = [tokenizer.apply_chat_template(conversation=messages,
                                        add_generation_prompt=True, tokenize=
                                        False) for messages in messages_list]
38          inputs = tokenizer(input_prompts, return_tensors="pt", padding=True,
                                        add_special_tokens=False)
39          inputs = {k: v.to(model.device) for k, v in inputs.items()}
40
41          output_ids = model.generate(**inputs, max_new_tokens=16, pad_token_id=
                                        tokenizer.eos_token_id)
42          output_ids = output_ids[:, inputs['input_ids'][0].shape[-1]:output_ids
                                        .shape[-1]]
43          predictions = tokenizer.batch_decode(output_ids, skip_special_tokens=
                                        True)
44
45          if print_example:
46              print_example = False
47              print(f"### Prompt:\n{user_prompts[0]}")
48              print(f"### Model Output:\n{predictions[0]}")
49              print(f"### Label:\n{id2label[batch['sentiment'][0]]}")
50              print()
51
52          pred_ids = []
53          true_labels = batch['sentiment']
54          for p, l in zip(predictions, true_labels):
55              try:
56                  label_id = l
57                  p = p.split(":")[-1].strip()
58                  pred_id = label2id[p]
59              except Exception as e:
60                  pred_id = (l + 1) % len(label2id)
61              pred_ids.append(pred_id)
62
63          all_predictions.extend(pred_ids)
64          all_labels.extend(true_labels)
65
66      accuracy_metric = evaluate.load("accuracy")
67      f1_metric = evaluate.load("f1")
68      precision_metric = evaluate.load("precision")
69      recall_metric = evaluate.load("recall")
70
71      metrics = {
72          'accuracy': accuracy_metric.compute(predictions=all_predictions,
                                        references=all_labels),
73          'f1': f1_metric.compute(predictions=all_predictions, references=
                                        all_labels, average='macro'),
74          'precision': precision_metric.compute(predictions=all_predictions,
                                        references=all_labels, average='macro')
                                        ,
75          'recall': recall_metric.compute(predictions=all_predictions,
                                        references=all_labels, average='macro')
76      }
77
78      results = {}
79      for metric_name, metric_dict in metrics.items():
```

```
        results.update(metric_dict)

    return results
```

Ví dụ few-shot learning với số example = 4

> Predict the sentiment of the following input sentence.
> The response must begin with "Sentiment: ", followed by one of these keywords: "positive", "negative", or "neutral", to reflect the sentiment of the input sentence.
>
> Here are a few examples:
>
> Sentence: các dụng cụ thực hành không được cung cấp đầy đủ .
> Sentiment: negative
>
> Sentence: nhưng buổi thực hành hết sức bổ ích và tính ứng dụng cực cao .
> Sentiment: positive
>
> Sentence: giảng viên giảng dạy hay , vui tính .
> Sentiment: positive
>
> Sentence: khả năng truyền đạt , giao tiếp rất kém , kiến thức không vững , thiếu khả năng tương tác với sinh viên .
> Sentiment: negative
>
> Sentence: nói tiếng anh lưu loát .

Ta sẽ đánh giá mô hình Llama 3.2 1B Instruct với số shot lần lượt là 1, 2, 4, và 8 shots.

```
# Pick a list of shot from the train set
shuffled_train_dataset = dataset['train'].shuffle()
sampled_few_shot_examples = list(shuffled_train_dataset.select(range(10)))

n_shots = [1, 2, 4, 8]
for n in n_shots:
    few_shot_examples = sampled_few_shot_examples[:n]
    few_shot_results = evaluate_few_shot(
        model=base_model,
        tokenizer=tokenizer,
        eval_dataset=dataset['test'],
        few_shot_examples=few_shot_examples,
        batch_size=16,
        print_example=True,
    )
    print(f"*** Few-shot evaluation results with {n} shots:")
    for metric_name, value in few_shot_results.items():
        print(f"* {metric_name}: {value:.4f}")
    print()
```

Kết quả đánh giá nằm ở Bảng 3.

n_shots	accuracy	f1	precision	recall
1	0.6999	0.4825	0.6933	0.5085
2	0.7795	0.5412	0.7117	0.5604
4	0.8743	0.6346	0.8628	0.6362
8	0.8042	0.5867	0.8922	0.5910

Bảng 3: Kết quả sau khi huấn luyện với Few-shot.

Nhìn chung, ta có thể thấy càng tăng số lượng example trong prompt, độ chính xác của mô hình càng tăng.

III.2.11. Thay đổi Rank của LoRA

Chúng ta sẽ thí nghiệm việc thay đổi rank của LoRA với các giá trị $r \in \{1, 2, 4, 8, 16, 32, 64, 128\}$. Ta viết hàm `train_lora` như sau:

```python
def train_lora(base_model, tokenizer, training_args, lora_rank, dataset):
    peft_config = LoraConfig(
        task_type=TaskType.CAUSAL_LM, inference_mode=False, r=lora_rank,
                                      lora_alpha=32, lora_dropout=0.1
    )
    cur_peft_model = get_peft_model(base_model, peft_config)
    cur_peft_model.print_trainable_parameters()

    trainer = SFTTrainer(
        model=cur_peft_model,
        args=training_args,
        train_dataset=dataset['train'],
        eval_dataset=dataset['validation'],
        preprocess_logits_for_metrics=preprocess_logits_for_metrics,
        compute_metrics=compute_metrics,
        processing_class=tokenizer,
        data_collator=data_collator,
    )
    trainer.train()
    return cur_peft_model
```

và chạy code sau để huấn luyện mô hình LoRA nhiều lần với rank khác nhau.

```python
ranks = [1, 2, 4, 8, 16, 32, 64, 128]
# ranks = [1, 2]
rank_results = pd.DataFrame(columns=['rank', 'accuracy', 'f1', 'precision', '
                                     recall'])
for rank in ranks:
    print(f'*** Train with rank {rank}')
    cur_trained_model = train_lora(base_model, tokenizer, training_args, rank,
                                   tokenized_dataset)
    cur_results = evaluate_zero_shot(
        model=cur_trained_model,
        tokenizer=tokenizer,
        eval_dataset=dataset['test'],
        batch_size=8
```

```
    )

    # add current results to rank_results
    rank_results.loc[len(rank_results)] = [rank, cur_results['accuracy'],
                                           cur_results['f1'], cur_results['
                                           precision'], cur_results['recall']]
rank_results
```

Kết quả tổng hợp được hiển thị ở Bảng 4.

rank	accuracy	f1	precision	recall
1	0.9261	0.7957	0.8844	0.7606
2	0.9299	0.8094	0.8766	0.7772
4	0.9292	0.8102	0.8815	0.7767
8	0.9267	0.8015	0.8628	0.7715
16	0.9299	0.8163	0.8799	0.7842
32	0.9274	0.8077	0.8625	0.7788
64	0.9305	0.8168	0.8803	0.7846
128	0.9280	0.8137	0.8623	0.7862

Bảng 4: Hiệu suất với các rank khác nhau.

IV. Câu hỏi trắc nghiệm

1. Mục tiêu chính của LoRA là:
   (a) Giảm số lượng tham số cần fine-tune.
   (b) Tăng tốc inference của mô hình.
   (c) Thay thế toàn bộ kiến trúc transformer.
   (d) Loại bỏ nhu cầu sử dụng GPU.

2. Trong LoRA, ma trận trọng số được phân rã thành:
   (a) Hai ma trận hạng thấp.
   (b) Một ma trận sparse và một ma trận dense.
   (c) Một tensor và một attention map.
   (d) Một ma trận và một vector bias.

3. Tham số chính điều chỉnh độ phức tạp của LoRA là:
   (a) learning rate.
   (b) rank (thường ký hiệu là r).
   (c) số epoch.
   (d) batch size.

4. LoRA chủ yếu được áp dụng vào:
   (a) Layer normalization
   (b) Positional embedding.
   (c) Activation functions.
   (d) Linear layer (thường là attention projection).

5. Trong quá trình inference, mô hình LoRA:
   (a) Cộng kết quả của adapter với trọng số gốc.
   (b) Sử dụng riêng adapter, bỏ qua trọng số gốc.
   (c) Dùng một phiên bản pruned của mô hình.
   (d) Hoạt động như mô hình lượng tử hóa.

6. QLoRA cải tiến điều gì so với LoRA?
   (a) Tăng số lượng tham số được fine-tune.
   (b) Dùng lượng tử hóa để giảm VRAM sử dụng.
   (c) Thay thế hoàn toàn attention layer.
   (d) Loại bỏ nhu cầu huấn luyện.

7. QLoRA thường sử dụng kỹ thuật lượng tử hóa nào?
   (a) FLOAT16.
   (b) BFLOAT16.
   (c) 4-bit NormalFloat (NF4).
   (d) 8-bit FP.
8. Kỹ thuật double quantization trong QLoRA dùng để:
   (a) Nén bảng mã lượng tử hóa.
   (b) Lượng tử hóa trọng số hai lần để tăng độ chính xác.
   (c) Tăng tốc quá trình giải mã.
   (d) Loại bỏ noise trong gradient.
9. Kỹ thuật lượng tử hóa 4-bit như NF4 trong QLoRA giúp:
   (a) Loại bỏ hoàn toàn sai số làm tròn.
   (b) Áp dụng tốt hơn trên mô hình nhỏ
   (c) Tăng hiệu suất mà vẫn giữ được độ chính xác.
   (d) Dự đoán nhanh hơn nhưng độ chính xác giảm mạnh.
10. Vì sao lượng tử hóa 8-bit (INT8) vẫn phổ biến trong nhiều ứng dụng?
    (a) Không cần thay đổi phần cứng.
    (b) Cân bằng tốt giữa hiệu suất và độ chính xác.
    (c) Là chuẩn bắt buộc cho transformer.
    (d) Là lượng tử hóa duy nhất hỗ trợ gradient descent.

V. Tài liệu tham khảo

[1] T. Dettmers, A. Pagnoni, A. Holtzman **and** L. Zettlemoyer, *QLoRA: Efficient Finetuning of Quantized LLMs*, 2023.

[2] K. V. Nguyen, V. D. Nguyen, P. X. V. Nguyen, T. T. H. Truong **and** N. L.-T. Nguyen, "UIT-VSFC: Vietnamese Students' Feedback Corpus for Sentiment Analysis," **in***2018 10th International Conference on Knowledge and Systems Engineering (KSE)* 2018, **pages** 19–24.

VI. Phụ lục

1. **Coding:** Các Jupyter Notebook được cung cấp tại đây.
2. **Datasets:** Thông tin về UIT-VSFC Dataset xem tại đây.

- Hết -