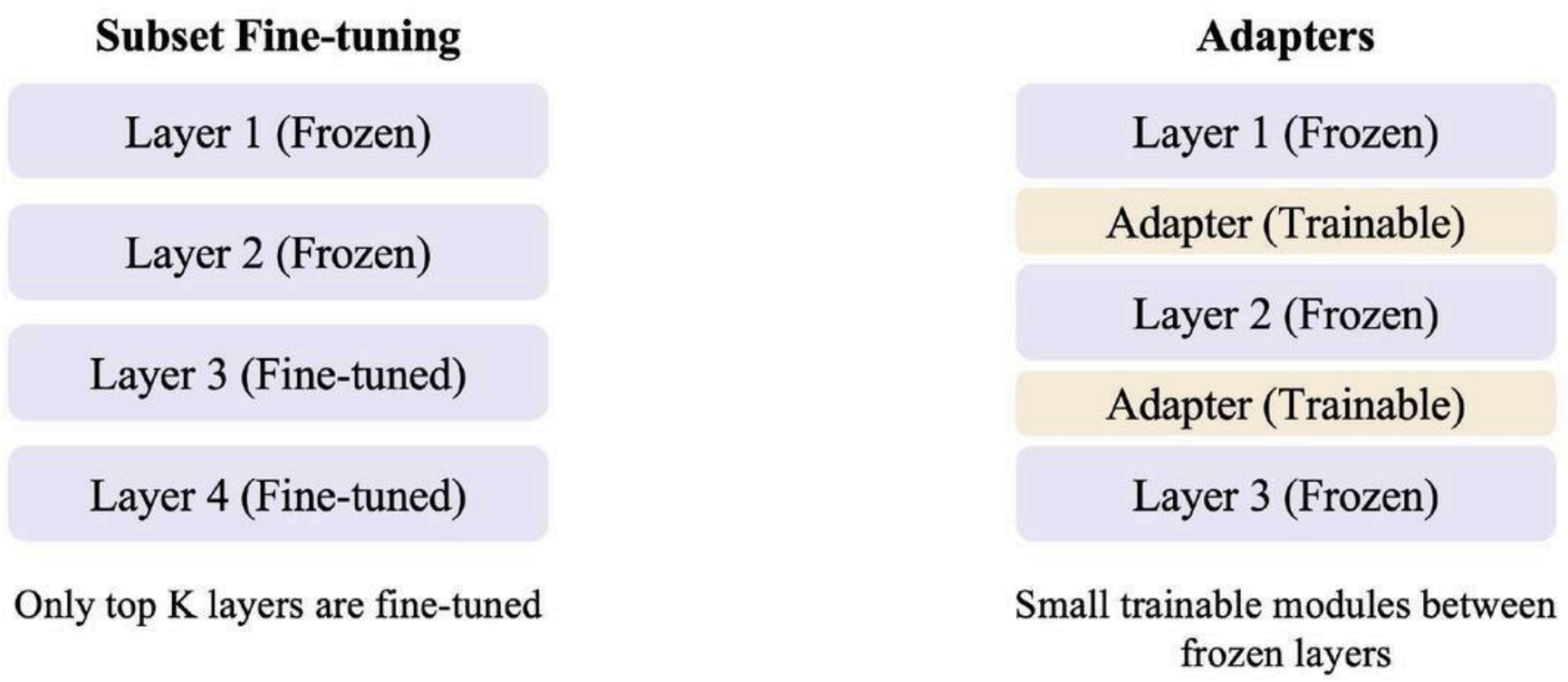
AI VIET NAM – AIO COURSE 2024

PEFT for Multiple-choice Question Answering

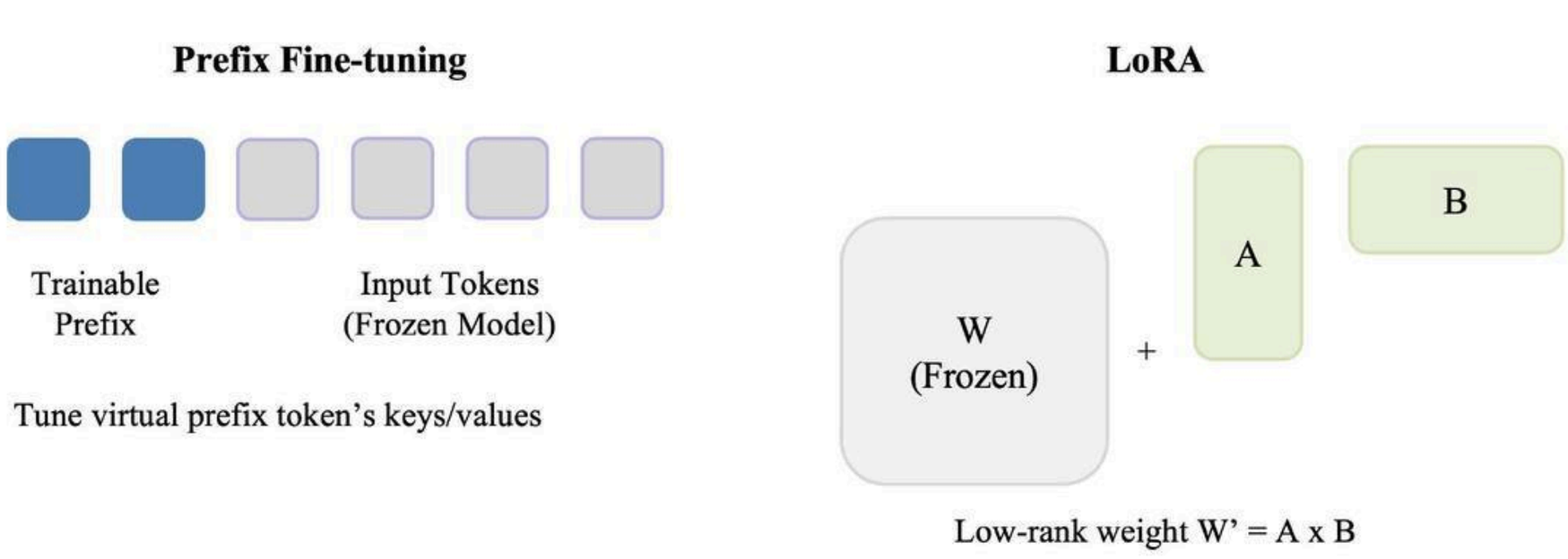
Quoc-Thai Nguyen, Quang-Hien Ho và Quang-Vinh Dinh

Ngày 21 tháng 4 năm 2025

Phần 1. Giới thiệu



Hình 1: Subset Fine-tuning and Adapters Tuning.



Hình 2: Pre-fix Tuning and LoRA.

Trong bối cảnh các mô hình ngôn ngữ lớn (Large Language Models – LLMs) ngày càng phát triển cả về quy mô và sức mạnh, việc tinh chỉnh (fine-tune) toàn bộ mô hình trở nên tốn kém cả về bộ nhớ lẫn chi phí tính toán. **Parameter-Efficient Fine-Tuning (PEFT)** là một nhóm các kỹ thuật nhằm

giảm số lượng tham số cần cập nhật trong quá trình fine-tune, giữ nguyên phần lớn trọng số gốc của mô hình, từ đó giúp tăng hiệu quả huấn luyện và khả năng mở rộng.

Mục tiêu chính của PEFT:

- Tận dụng sức mạnh của mô hình tiền huấn luyện (pre-trained).
- Giảm số lượng tham số cần cập nhật trong quá trình tinh chỉnh.
- Dễ dàng chia sẻ, lưu trữ và tái sử dụng các mô hình đã tinh chỉnh.

1. **Subset Fine-tuning**

   Subset Fine-tuning là một phương pháp tinh chỉnh hiệu quả tham số bằng cách chỉ tinh chỉnh một phần (subset) các tầng của mô hình, giữ nguyên (freeze) các tầng còn lại. Cơ chế hoạt động: Chỉ top-K tầng cuối cùng (thường là những tầng gần output) được fine-tune, các tầng trước đó được giữ nguyên.

   Ưu điểm:

   - Giảm chi phí huấn luyện vì số lượng tham số cập nhật ít hơn.

   - Dễ triển khai trên hầu hết các kiến trúc Transformer hiện có.

   Hạn chế:

   - Không tận dụng toàn bộ cấu trúc mô hình để thích nghi với dữ liệu mới.

   - Dễ bị overfit nếu fine-tune quá ít tầng hoặc underfit nếu chọn tầng không phù hợp.

2. **Adapter-tuning**

   Adapters là các module nhỏ được chèn vào giữa các tầng của mô hình gốc. Mô hình chính được giữ nguyên, chỉ các module Adapter là được huấn luyện.

   Kiến trúc Adapter phổ biến: Down-projection → Non-linearity → Up-projection (thường với bottleneck rank nhỏ, ví dụ: 16, 32)

   Ưu điểm:

   - Có thể dễ dàng huấn luyện nhiều task khác nhau bằng cách thay Adapter.

   - Thích hợp cho multi-task learning hoặc deployment nhiều mô hình nhẹ.

   - Cập nhật ít tham số nhưng vẫn đạt hiệu quả cao.

   Hạn chế:

   - Cần chèn các module vào mô hình, đòi hỏi can thiệp code.

   - Tăng độ trễ nhẹ trong quá trình inference.

3. **Prefix Tuning**

   Prefix Tuning là kỹ thuật tinh chỉnh bằng cách thêm các vector học được (prefix vectors) vào phần đầu của chuỗi đầu vào trong mỗi layer, đồng thời giữ nguyên toàn bộ mô hình gốc.

   Cách hoạt động:

   - Huấn luyện các prefix embedding, thường biểu diễn attention key/value bổ sung.

   - Không can thiệp trực tiếp vào tham số gốc của mô hình.

   Ưu điểm:

   - Số lượng tham số cần huấn luyện cực kỳ nhỏ (thường $<1\%$).

   - Dễ mở rộng sang nhiều tác vụ khác nhau.

Hạn chế:

- Phụ thuộc vào khả năng của mô hình trong việc xử lý prefix (không phải mô hình nào cũng hỗ trợ).

- Cần số lượng prefix tương ứng với mỗi tầng.

4. **Low-rank Adaptation** LoRA là kỹ thuật tinh chỉnh bằng cách chèn ma trận low-rank để mô phỏng sự thay đổi tham số trong các tầng Attention.

   Cách hoạt động:

   - Các ma trận trọng số W được đóng băng.

   - Thay vào đó, huấn luyện hai ma trận A và B có rank thấp sao cho: $W^{'} = W + W, W = AB$

   Ưu điểm:

   - Tối ưu về bộ nhớ: số tham số học được giảm mạnh (rank thấp).

   - Hiệu quả tương đương hoặc tốt hơn full fine-tuning trong nhiều task.

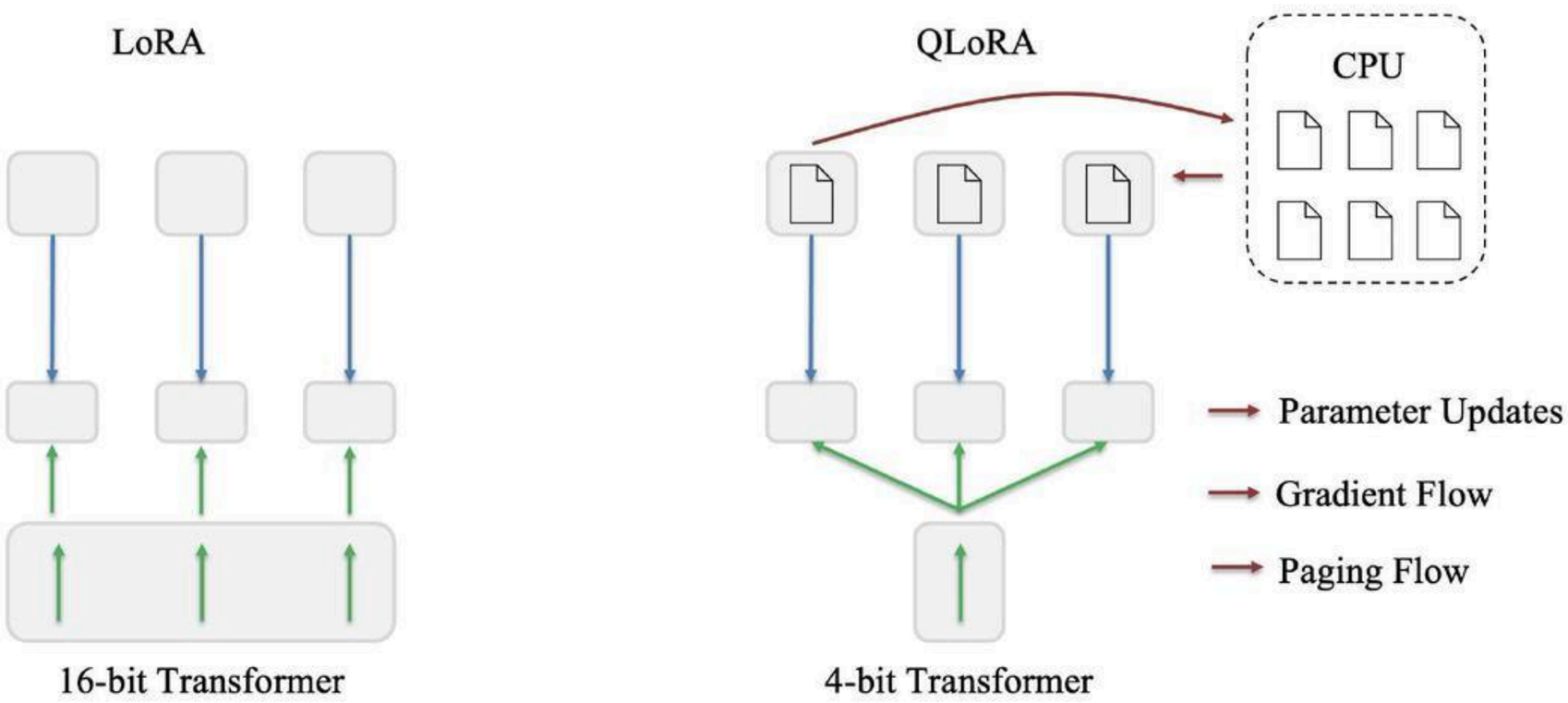
   - Không cần can thiệp quá sâu vào mô hình, dễ áp dụng.

   Hạn chế:

   - Cần chọn rank hợp lý (quá thấp → thiếu năng lực học, quá cao → tốn tài nguyên).

   - Dễ overfit nếu áp dụng không đúng cấu hình.

5. **QLoRA**



Hình 3: LoRA and QLoRA.

QLoRA là viết tắt của Quantized Low-Rank Adapter, là một phương pháp fine-tuning cực kỳ tiết kiệm tài nguyên, cho phép huấn luyện các mô hình ngôn ngữ cực lớn (tới 65B tham số) ngay cả trên GPU 24 hoặc 48 GB (consumer GPUs).

Mục tiêu chính của QLoRA:

- Kết hợp sức mạnh của LoRA với mô hình được lượng tử hóa (quantized).

- Giảm tối đa dung lượng bộ nhớ và chi phí tính toán khi fine-tune.

- Giữ hiệu năng tương đương hoặc tốt hơn so với full fine-tuning trong nhiều tác vụ.

QLoRA kết hợp 3 ý tưởng chính:

- 4-bit Quantization của mô hình gốc

  Trước tiên, mô hình gốc (pre-trained model) được lượng tử hóa về 4-bit bằng kỹ thuật NF4 (Normalized Float 4-bit) – một định dạng mới giữ lại tốt hơn phân phối gốc của trọng số.

  Điều này giúp giảm đáng kể bộ nhớ RAM/GPU, cho phép chạy mô hình lớn hơn rất nhiều.

- Low-Rank Adapter

  Mô hình gốc được giữ nguyên (sau khi lượng tử hóa).

  Các ma trận Low-Rank (A $\times$ B) như trong LoRA sẽ được huấn luyện thêm, nhưng vẫn ở độ chính xác float32 hoặc bfloat16.

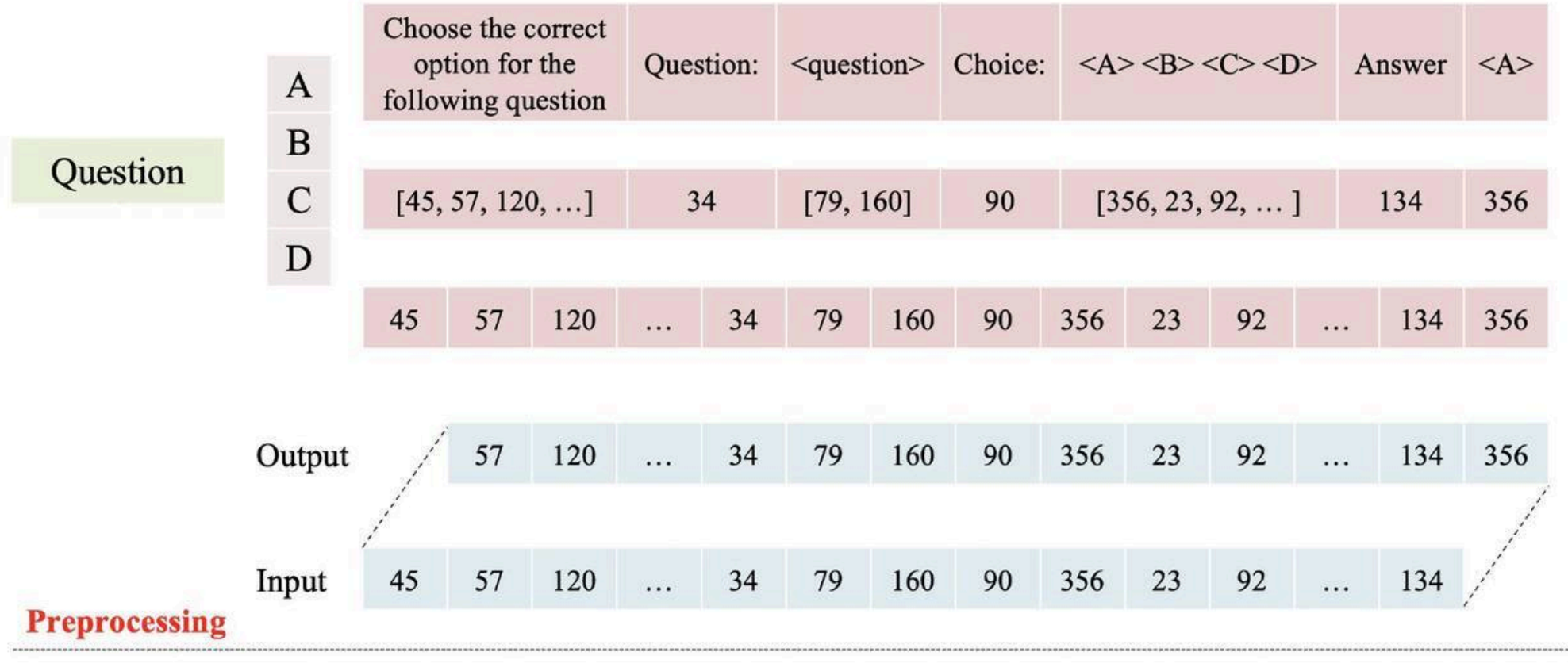
  Như vậy, chỉ có rất ít tham số cần học (low-rank), và không cần cập nhật mô hình gốc.

- Double Quantization

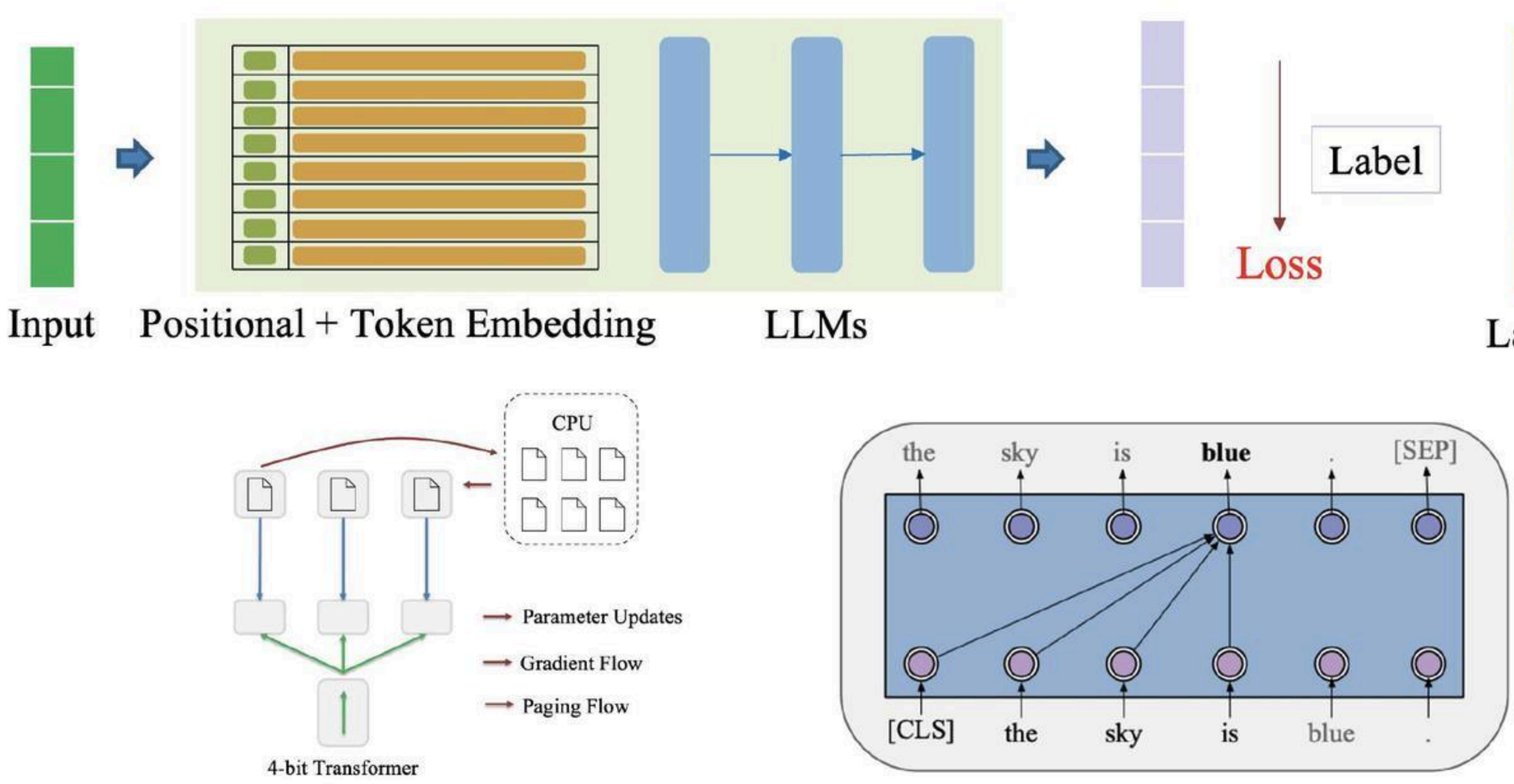
  Tăng cường nén bằng cách tiếp tục lượng tử hóa các giá trị lượng tử hóa – giúp giảm thêm nhu cầu lưu trữ mà không làm mất thông tin.

Phần 2. PEFT for Multiple-choice QA

Mô hình áp dụng các phương pháp trong PEFT áp dụng cho bài toán Multiple-choice QA được thực hiện với 2 phần như hình sau:



Hình 4: Preprocessing Stage.



Hình 5: Training Stage.

Quá trình huấn luyện mô hình như sau:

1. Cài đặt thư viện và tải về bộ dữ liệu

Đầu tiên là cài đặt một số thư viện để có thể chạy được các mô hình ngôn ngữ lớn từ thư viện Unsloth. Và bộ dữ liệu được sử dụng là MedMCQA, với mỗi câu hỏi sẽ có 4 đáp án và một đáp án đúng để làm nhãn.

```
# Install libs
!pip install -q unsloth==2025.3.15 datasets==3.5.0
!pip install transformers bitsandbytes

from datasets import load_dataset

ds = load_dataset("openlifescienceai/medmcqa")
del ds["test"]
```

2. Modeling

Mô hình LLaMA-3.2-1B được tải từ thư viện `unsloth` với kỹ thuật lượng tử hóa 4-bit, giúp giảm dung lượng và tối ưu hóa hiệu suất tính toán trên GPU.

Tiếp theo, kỹ thuật LoRA được áp dụng lên nhiều thành phần của mô hình, bao gồm các modules attention (`q_proj`, `k_proj`, `v_proj`, `o_proj`), các modules feed-forward network (`up_proj`, `down_proj`), và module gating (`gate_proj`).

Bên cạnh đó, gradient checkpointing được kích hoạt giúp tiết kiệm bộ nhớ trong quá trình huấn luyện mà không làm giảm hiệu suất.

Đặc biệt, ReLoRA cũng được kích hoạt với `use_relora=True`, cho phép mô hình định kỳ hợp nhất các tham số LoRA vào mô hình gốc và tái khởi tạo ma trận LoRA. Kỹ thuật này không chỉ giúp cải thiện hiệu quả huấn luyện mà còn duy trì bộ nhớ thấp.

```
from unsloth import FastLanguageModel

max_seq_length = 2048
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/Llama-3.2-1B-bnb-4bit",
    max_seq_length=max_seq_length,
    load_in_4bit=True,
    dtype=None,
)

model = FastLanguageModel.get_peft_model(
    model,
    r=16,
    lora_alpha=16,
    lora_dropout=0,
    target_modules=[
        "q_proj", "k_proj", "v_proj", "up_proj",
        "down_proj", "o_proj", "gate_proj"],
    use_rslora=True,
    use_gradient_checkpointing="unsloth",
    random_state = 42,
    loftq_config = None,
)
print(model.print_trainable_parameters())
```

3. Preprocessing

Mã hoá văn bản thành các vector và kết hợp thiết kế prompt xây dựng đầu vào mô hình qua các bước:

- Lấy dữ liệu từ examples: câu hỏi (`question`), danh sách các lựa chọn (`opa`, `opb`, `opc`, `opd`), đáp án (`cop`).
- Ánh xạ đáp án (`cop`) vào nhãn `id2label`: vì đáp án `cop` có định dạng số nguyên (0, 1, 2, 3), ta ánh xạ chúng vào các nhãn tương ứng (0 -> A, 1 -> B, 2 -> C, 3 -> D).
- Sử dụng `.format()` để kết hợp câu hỏi và các lựa chọn đáp án thành một chuỗi hoàn chỉnh, theo định dạng yêu cầu.
- Cuối cùng, áp dụng hàm `map()` lên toàn bộ dataset để chuẩn hóa dữ liệu, giúp dữ liệu được biến đổi theo quy trình đã định, sẵn sàng cho huấn luyện.

Cấu trúc của prompt:

- Phần hướng dẫn: "Choose the correct option for the following question".
- Phần câu hỏi: "Question: <question>" (nội dung câu hỏi cụ thể thay cho <question>).
- Phần lựa chọn: "Choice: <A> <B> <C> <D>" (các phương án lựa chọn A, B, C, D).
- Phần đáp án: "Answer: <A>" (đáp án đúng, trong ví dụ này là A).

```
data_prompt = """Choose the correct option for the following question.

### Question:
{}

### Choice:
{}

### Answer:
"""

id2label = {
    0: 'A',
    1: 'B',
    2: 'C',
    3: 'D'
}

def formatting_prompt(examples):
    questions = examples["question"]
    opas = examples["opa"]
    opbs = examples["opb"]
    opcs = examples["opc"]
    opds = examples["opd"]
    cops = examples["cop"]

    texts = []
    for idx in range(len(questions)):
        question = questions[idx]
        opa = opas[idx]
        opb = opbs[idx]
        opc = opcs[idx]
        opd = opds[idx]
        answer = id2label[cops[idx]]
        if answer == "A":
            answer = answer + " " + opa
        elif answer == "B":
```

```
            answer = answer + " " + opb
        elif answer == "C":
            answer = answer + " " + opc
        elif answer == "D":
            answer = answer + " " + opd

        choices = f"A. {opa}. B. {opb}. C. {opc}. D. {opd}."
        text = data_prompt.format(question, choices)
        texts.append(text)
    return {"text": texts,}

process_ds = ds.map(formatting_prompt, batched=True)
```

4. Training

Sau khi dữ liệu huấn luyện được chuẩn bị, quá trình tùy chỉnh các tham số huấn luyện được thiết lập thông qua TrainingArguments, bao gồm:

- Giá trị `per_device_train_batch_size` kết hợp với `gradient_accumulation_steps` để đạt số lượng mẫu mà mô hình sẽ xử lý trước khi cập nhật trọng số phù hợp với tài nguyên GPU và số bước huấn luyện bạn mong muốn.
- Sử dụng optimizer 8-bit (`adamw_8bit`) nhằm tiết kiệm bộ nhớ và tăng hiệu suất.
- Scheduler dạng linear kết hợp với `warmup_steps=10` để thực hiện warm-up trước khi học với tốc độ đầy đủ.
- Đánh giá và lưu mô hình mỗi 50 bước thông qua `eval_steps` và `save_steps`.
- Cấu hình tự động tải lại mô hình có hiệu suất tốt nhất vào cuối quá trình huấn luyện bằng `load_best_model_at_end=True`.

Khi các tham số huấn luyện đã được cấu hình, mô hình huấn luyện được bắt đầu với `SFTTrainer`. Các tham số huấn luyện được truyền vào qua đối tượng `args`, và các bộ dữ liệu huấn luyện và kiểm tra được chỉ định qua `train_dataset` và `eval_dataset`.

Thêm vào đó, tham số `dataset_text_field="text"` xác định trường chứa văn bản đầu vào trong dataset.

Cuối cùng, quá trình huấn luyện được thực hiện thông qua lệnh `trainer.train()`.

```
from trl import SFTTrainer
from transformers import TrainingArguments
from unsloth import is_bfloat16_supported

args = TrainingArguments(
        output_dir="med-mcqa-llama-3.2-1B-4bit-lora",
        logging_dir="logs",
        learning_rate=3e-4,
        lr_scheduler_type="linear",
        per_device_train_batch_size=64,
        gradient_accumulation_steps=16,
        num_train_epochs=2,
        eval_strategy="steps",
        save_strategy="steps",
        logging_strategy="steps",
        eval_steps=50,
        save_steps=50,
        logging_steps=50,
        save_total_limit=1,
        load_best_model_at_end=True,
```

```
        fp16=not is_bfloat16_supported(),
        bf16=is_bfloat16_supported(),
        optim="adamw_8bit",
        weight_decay=0.01,
        warmup_steps=10,
        seed=0,
    ),

trainer=SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    args=args,
    train_dataset=process_ds["train"],
    eval_dataset=process_ds["validation"],
    dataset_text_field="text",
)

trainer.train()
```

5. Save Checkpoints

Khi quá trình huấn luyện hoàn tất, mô hình được lưu để sử dụng hoặc chia sẻ sau này. Cụ thể:

- Mô hình sau huấn luyện được lưu cục bộ vào thư mục `"unsloth-llama-traned"` bằng hàm `save_pretrained()`.
- Sau đó, mô hình được đẩy lên Hugging Face Hub thông qua hàm `push_to_hub()`, giúp dễ dàng chia sẻ và tái sử dụng trong các dự án khác.
- Để thực hiện bước này, người dùng cần có tài khoản trên Hugging Face và phải cung cấp `use_auth_token=True` để xác thực quyền truy cập.

```
from huggingface_hub import login

login(token="hf_xxxxxxxxxxxxxxx") # your access token

model.save_pretrained("unsloth-llama-traned")
PEFT_MODEL = "your_huggingface_user_name/Llama-3.2-1B-bnb-4bit-MedMCQA"
model.push_to_hub(PEFT_MODEL, use_auth_token=True)
```

6. Inference

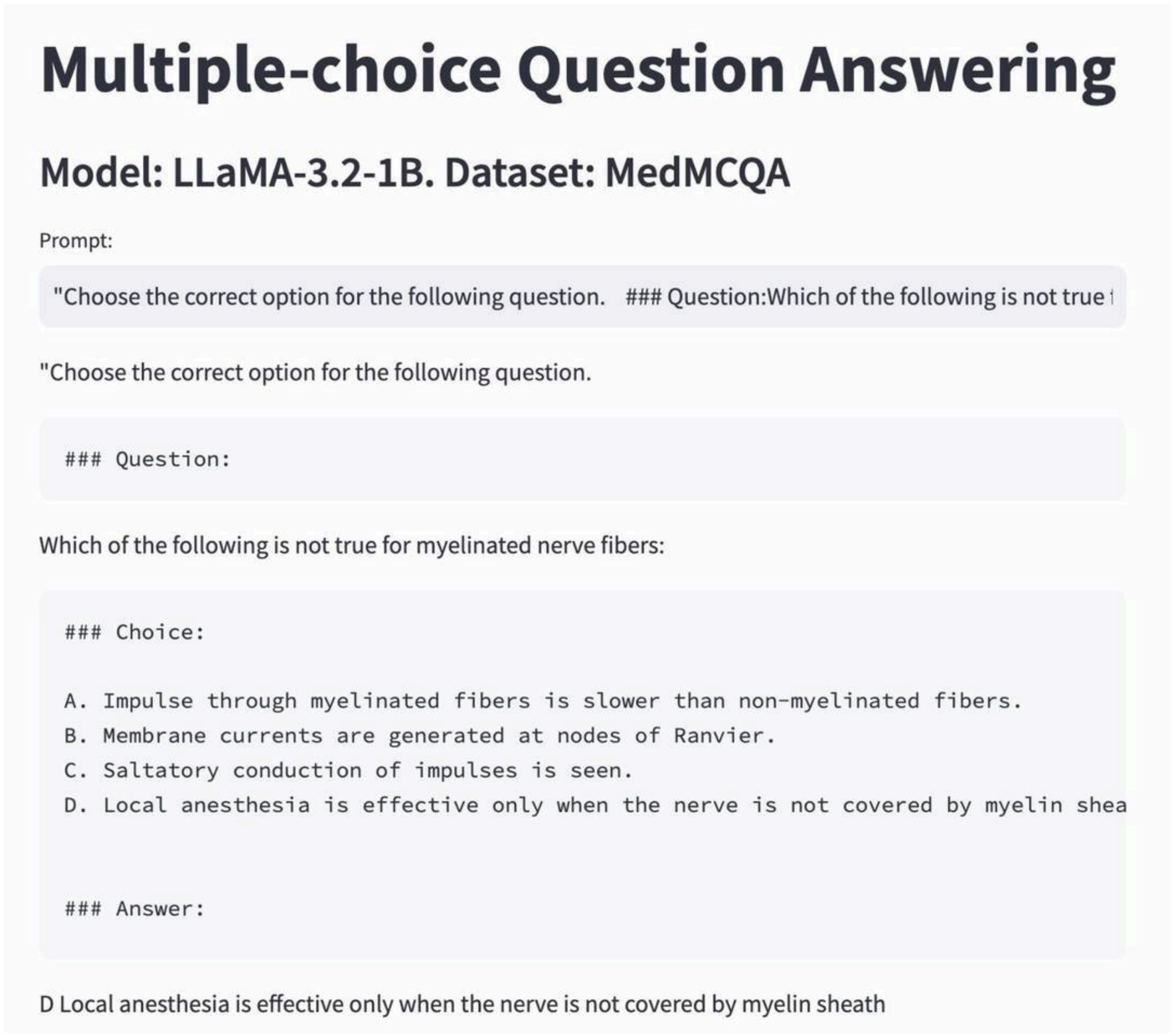
Sau khi huấn luyện, sử dụng mô hình để thực hiện suy luận (inference) trên một ví dụ như sau:

- **Option 1:**
  - Sử dụng `FastLanguageModel.from_pretrained()` để tải mô hình đã huấn luyện với các tham số cấu hình như `max_seq_length` và `load_in_4bit`.
  - `FastLanguageModel.for_inference(model)` giúp tối ưu hóa mô hình cho quá trình suy luận (inference).
  - Dữ liệu đầu vào được tokenized bằng `tokenizer`, sau đó mô hình sinh kết quả dựa trên `generate()`.
  - Kết quả được giải mã bằng `tokenizer.decode()`.
- **Option 2:**
  - Sử dụng `pipeline("text-generation")` trong thư viện transformers để tạo ra một pipeline sẵn có cho tác vụ sinh văn bản.
  - Dữ liệu đầu vào được xử lý bằng pipeline và mô hình sẽ trả về kết quả dựa trên `max_new_tokens`.
  - Kết quả trả về dưới dạng chuỗi văn bản sinh ra từ mô hình.

```python
from transformers import pipeline

#Option 1
max_seq_length = 2048
model,tokenizer = FastLanguageModel.from_pretrained(
    model_name = PEFT_MODEL,
    max_seq_length = max_seq_length,
    load_in_4bit = True,
    dtype = None,
)
FastLanguageModel.for_inference(model)
model.to("cuda")

input_text = process_ds["validation"][0]["text"]

inputs = tokenizer(input_text, return_tensors="pt").to("cuda")

output = model.generate(inputs["input_ids"], max_length=128)

generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
print(f"Answer: {generated_text}")

#Answer: D Local anesthesia is effective only when the nerve is not covered by myelin sheath\n

#Option 2
generator = pipeline("text-generation", model="thainq107/med-mcqa-llama-3.2-1B-4bit-lora")

output = generator([process_ds["validation"][0]["text"]], max_new_tokens=128, return_full_text=False)[0]

# [{"generated_text": "D Local anesthesia is effective only when the nerve is not covered by myelin sheath\n"}]
```

7. Deployment

Triển khai mô hình trên streamlit. Tham khảo về code và demo.



Hình 6: Multiple-choice QA deployment on streamlit.

Phần 3. Câu hỏi trắc nghiệm

Câu hỏi 1 Trong kiến trúc Llama 3.2-1B, chọn nhóm tầng nào để gắn LoRA adapters vừa can thiệp trực tiếp vào Attention và Feed-Forward nhưng vẫn đạt hiệu quả cao về tham số?
a) embed_tokens và lm_head
b) q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj
c) Toàn bộ LayerNorm
d) Chỉ gate_proj và up_proj

Câu hỏi 2 Nếu không đổi token PAD trong labels thành -100 trước khi tính loss, điều gì xảy ra?
a) Mô hình bị ép học dự đoán PAD, loss tăng và chất lượng giảm
b) Trainer tự động bỏ qua PAD
c) Quá trình train nhanh hơn vì ít mask
d) Gây lỗi shape mismatch trong CrossEntropyLoss

Câu hỏi 3 Đặt max_seq_length = 128. Khi prompt + answer dài hơn giới hạn và bị truncation, rủi ro nghiêm trọng nhất đối với bài MCQA là gì?
a) Thiếu <eos> nên mô hình không dừng sinh
b) Mất <bos> làm lệch embedding
c) Bị cắt mất một hoặc nhiều lựa chọn đáp án làm mô hình thiếu dữ liệu suy luận
d) Tăng bộ nhớ vì padding

Câu hỏi 4 Trong bốn cấu hình fine-tune sau, cấu hình nào tiết kiệm VRAM nhất?
a) Full-finetune fp16 + ZeRO-1
b) LoRA bf16
c) LoRA 8-bit + gradient checkpointing
d) QLoRA 4-bit + FlashAttention-v2 + gradient checkpointing

Câu hỏi 5 Với DataCollatorForSeq2Seq, lý do phải truyền model=model là gì?
a) Lấy config.bos_token_id
b) Tự sinh <eos> khi cần
c) Suy luận label_pad_token_id từ model.config.pad_token_id nếu người dùng không cung cấp
d) Không chức năng gì, chỉ giữ API thống nhất

Câu hỏi 6 Đóng băng toàn bộ bias và LayerNorm, chỉ LoRA-hoá các linear nhằm mục đích chính nào?
a) Giảm FLOPs mà không ảnh hưởng hội tụ
b) Tránh "catastrophic forgetting" trong LayerNorm
c) Bias LayerNorm chứa quá ít tham số; cập nhật chúng dễ làm lệch phân phối đặc trưng
d) LoRA không hỗ trợ LayerNorm

Câu hỏi 7 Khi đã fine-tune bằng LoRA và chuyển sang inference, phương thức model.merge_and_unload() chủ yếu giúp gì?
a) Loại bỏ hoàn toàn adapter, trả mô hình về trạng thái ban đầu
b) Gộp trọng số LoRA vào mô hình gốc, giảm overhead khi tính forward
c) Khởi tạo lại gradient để có thể tiếp tục fine-tune
d) Tạo checkpoint mới cho các adapter

Câu hỏi 8 Khi đóng băng trọng số gốc và áp dụng LoRA, ưu điểm lý thuyết cốt lõi đối với không gian

tối ưu hóa là gì?
a) Phẳng hóa loss để gradient dễ đi hơn
b) Giới hạn cập nhật vào tiểu không gian hạng thấp, giảm overfitting và catastrophic forgetting
c) Tăng rank ma trận, cho phép biểu diễn phức tạp hơn
d) Cho phép tính Hessian trực tiếp với chi phí thấp

Câu hỏi 9 Phương pháp nào không thuộc vào PEFT?
a) Adapter Tuning
b) LoRA
c) QLoRA
d) Clustering

Câu hỏi 10 Bộ dữ liệu nào sử dụng cho Multiple-choice QA là?
a) MedMCQA
b) IMDB
c) C4
d) ROOTS

Phần 5. Phụ lục

1. **Hint**: Dựa vào file tải về PEFT for Multiple-choice Question Answering để hoàn thiện các đoạn code.

2. **Solution**: Các file code cài đặt hoàn chỉnh và phần trả lời nội dung trắc nghiệm có thể được tải về tại đây (Lưu ý: Sáng thứ 3 khi hết deadline phần project, admin mới copy các nội dung bài giải nêu trên vào đường dẫn).

3. **Rubric**:

Phần	Kiến Thức	Đánh Giá
1	- Hiểu rõ các mô kỹ thuật fine-tuning - Hiểu rõ mô hình ngôn ngữ lớn và huấn luyện mô hình ngôn ngữ lớn	- Các bước thực hiện trong quá trình fine-tuning LLMs.
2.	- Các phương pháp tối ưu trong fine-tuning LLMs: Prefix-Tuning, LoRA, QLoRA - Phân tích các mô hình và kỹ thuật trong PEFT	- Bài toán Multiple-choice Question Answering - Huấn luyện QLoRA với thư viện unsloth.

- Hết -