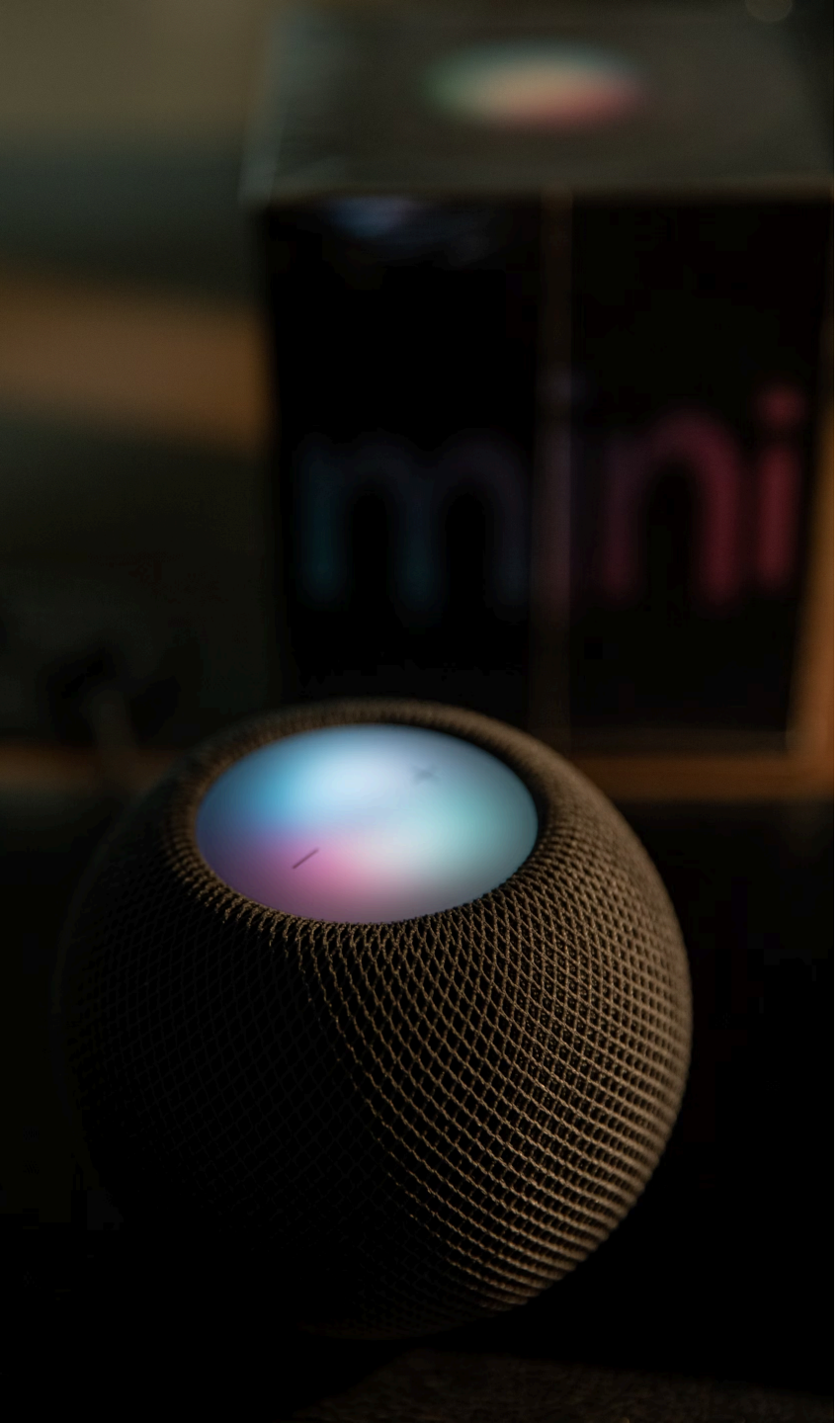
Finetuning LLMs

Microteaching by Amir Yunus

Materials

- Github: https://github.com/AmirYunus/finetune_LLM
- Lecture Notes available at the wiki: https://github.com/AmirYunus/finetune_LLM/wiki
- Code implemented in this lecture (**lab.ipynb**) in the github repository



Discussion

> In the future, students will learn **solely through AI conversations**, eliminating the need for human interaction and peer learning.

Studies have shown that ...

- AI-driven learning results in [1]:
  - Higher grades vs traditional learning
  - More consistent performance
- Student feedback on AI-driven learning:
  - More engaging
  - Improved retention

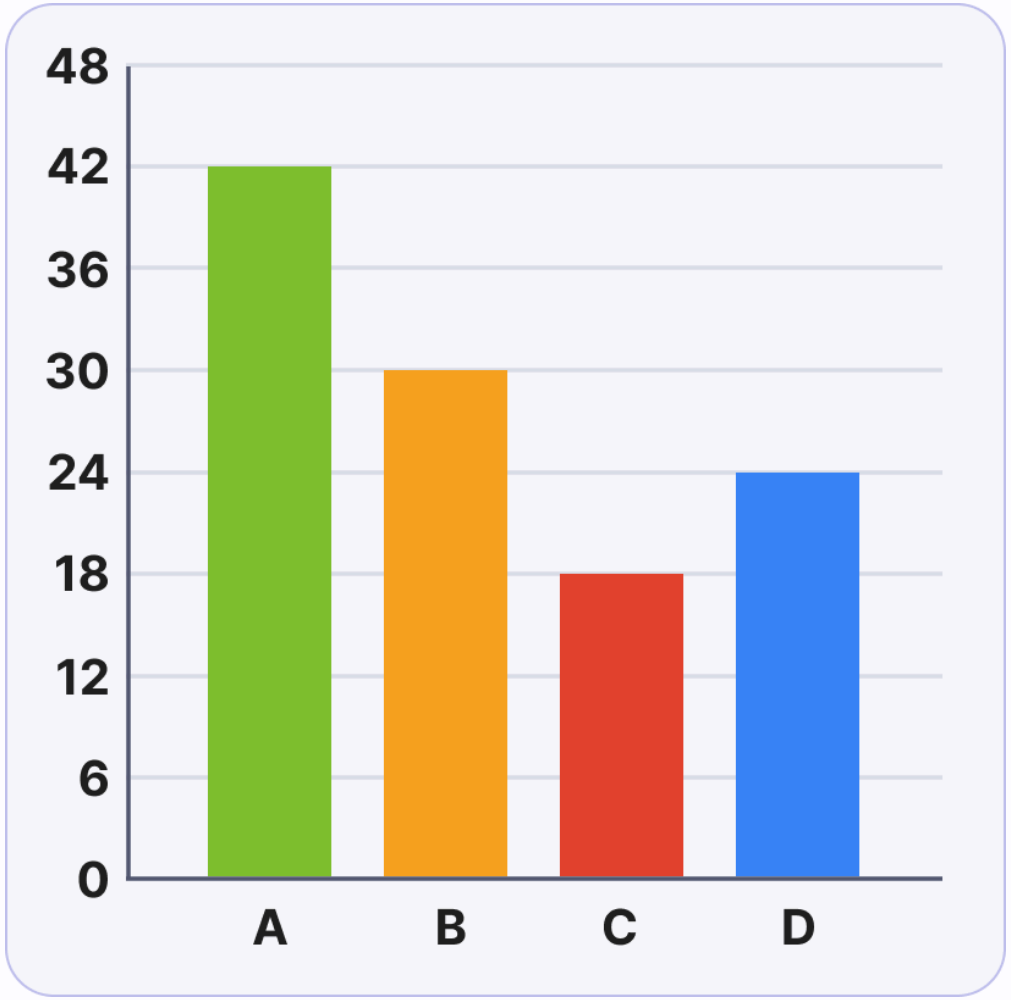
[1] M. J. Khan, O. Jian, and M. Joe, "Personalized learning through AI," *Advances in Engineering Innovation*, vol. 5, no. 1, pp. Dec. 2023. doi: 10.54254/2977-3903/5/2023039.

Poll

Would you feel comfortable with your child learning solely from an AI?

- A. Yes
- B. Yes, with some restrictions
- C. No, unless under supervision
- D. Never

Example Poll Results



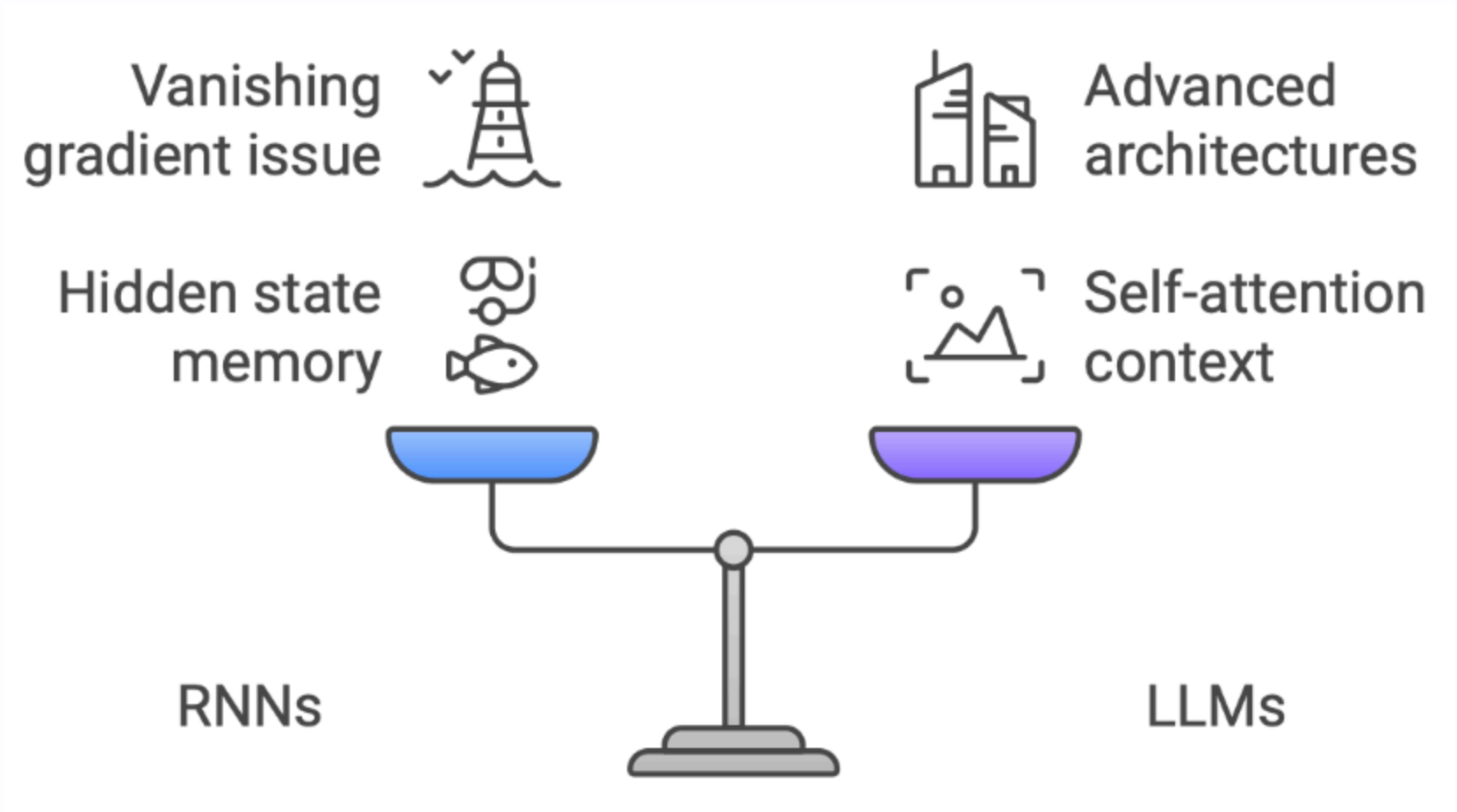
Learning Objectives

By the end of this lesson, you will be able to:

1. Understand the significance of LLM finetuning
2. Describe the finetuning process
3. Perform finetuning on a pre-trained LLM
4. Export the finetuned model and use it for inference

Recall

> Can anyone explain how the architecture of an RNN might relate to the way LLMs process sequences of text?


Vanishing gradient issue
Advanced architectures
Hidden state memory
Self-attention context
RNNs
LLMs

What is LLM Finetuning?

- Adapting pre-trained large language models (LLMs) for specific tasks.
- Utilises extensive knowledge embedded in models for distinct applications.

Deep Learning (DL) vs LLM Finetuning

- **Model Size:** Traditional DL uses smaller models vs LLMs with billions of parameters
- **Parameter Efficiency:** DL retrains many parameters vs LLMs use techniques like Low-Rank Adaptation (LoRA)
- **Context:** DL limited to shorter sequences vs LLMs maintain longer context

Deep Learning Finetuning Practices

- **Layer Freezing:** Early layers frozen, later layers trained
- **Learning Rate:** Lower rates prevent drastic weight changes
- **Data Augmentation:** Enhances training data diversity

LLM Finetuning Techniques

- **Adapter Layers:** Small neural layers inserted into pre-trained model while original parameters remain frozen.
- **Prompt Engineering:** Optimises input prompts to guide model responses without modifying weights.
- **Low-Rank Adaptation (LoRA):** Uses low-rank matrices for efficient weight updates by decomposing updates into smaller matrices.

The Finetuning Pipeline

1. Create virtual environment to isolate dependencies
2. Import required libraries
3. Load pre-trained model and tokenizer
4. Add LoRA adapters for efficient parameter tuning
5. Preprocess data for training
6. Fine-tune model on task-specific dataset
7. Run inference to test fine-tuned model
8. Export LoRA adapters
9. Save model in optimised format

Hands-on Lab

But before we start ...

Hardware Requirements

- Modern GPU with CUDA support (e.g. RTX 30 series / 40 series / A100)
- Minimum 16GB RAM for handling large datasets/models
- CUDA Toolkit (e.g. CUDA 12.4)
- Linux OS (e.g. Ubuntu 22.04)

  Note: PyTorch requires CUDA 12.4 which is only available on Ubuntu 22.04

Virtual Environment Setup

Install Dependencies:

```
sudo apt update
sudo apt install cmake libcurl4-openssl-dev -y
```

Create environment with Conda:

```
conda create --prefix=venv python=3.11 pytorch-cuda=12.4
      pytorch cudatoolkit xformers -c pytorch -c nvidia -c xformers -y
```

Activate environment:

```
conda activate ./venv
```

Install Packages

```
python -m pip install "unsloth[colab-new] @ git+https://github.com/unslothai/unsloth.git"
python -m pip install --no-deps trl peft accelerate bitsandbytes
python -m pip install jupyter
```

Import Required Libraries

```
from unsloth import FastLanguageModel
from unsloth import is_bfloat16_supported
from unsloth.chat_templates import get_chat_template
from unsloth.chat_templates import standardize_sharegpt
from unsloth.chat_templates import train_on_responses_only

from datasets import load_dataset
from trl import SFTTrainer
from transformers import TrainingArguments
from transformers import DataCollatorForSeq2Seq
from transformers import TextStreamer

import torch
```

Load Pre-Trained Model and Tokenizer

```
max_seq_length = 2048
dtype = None
load_in_4bit = True

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/Llama-3.2-3B-Instruct",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
)
```

Adding LoRA Adapters

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

Load Dataset

```
dataset = load_dataset("mlabonne/FineTome-100k", split="train")
```

Convert Dataset (1/2)

```
dataset = standardize_sharegpt(dataset)
```

Example conversion from ShareGPT format:

```
{"from": "system", "value": "You are an assistant"}
{"from": "human", "value": "What is 2+2?"}
{"from": "gpt", "value": "It's 4."}
```

To Hugging Face format:

```
{"role": "system", "content": "You are an assistant"}
{"role": "user", "content": "What is 2+2?"}
{"role": "assistant", "content": "It's 4."}
```

Convert Dataset (2/2)

The final conversation structure uses special tokens (Llama 3.1 format) from the Hugging Face format:

```
<|begin_of_text|>
<|start_header_id|>system<|end_header_id|>[system message]<|eot_id|>
<|start_header_id|>user<|end_header_id|>[user message]<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>[assistant message]<|eot_id|>
<|end_of_text|>
```

Configure the Tokenizer

```
tokenizer = get_chat_template(
    tokenizer,
    chat_template = "llama-3.1",
)
```

Format the Dataset for Training

```
def formatting_prompts_func(examples):
    convos = examples["conversations"]
    texts = [tokenizer.apply_chat_template(convo,
                    tokenize=False,
                    add_generation_prompt=False) for convo in convos]
    return {"text": texts}

dataset = dataset.map(formatting_prompts_func, batched=True)
```

Initialize the SFTTrainer

```
trainer = SFTTrainer(model = model,
    tokenizer = tokenizer,
    train_dataset = dataset,
    dataset_text_field = "text",
    max_seq_length = max_seq_length,
    data_collator = DataCollatorForSeq2Seq(tokenizer = tokenizer),
    dataset_num_proc = 2,
    packing = False,

    args = TrainingArguments(per_device_train_batch_size = 2,
        gradient_accumulation_steps = 4,
        warmup_steps = 5,
        max_steps = 60,
        learning_rate = 2e-4,
        fp16 = not is_bfloat16_supported(),
        bf16 = is_bfloat16_supported(),
        logging_steps = 1,
        optim = "adamw_8bit",
        weight_decay = 0.01,
        lr_scheduler_type = "linear",
        seed = 42,
        output_dir = "outputs",
        report_to = "none",),)
```

Train the Model on Responses Only

```
trainer = train_on_responses_only(
    trainer,
    instruction_part = "<|start_header_id|>user<|end_header_id|>\n\n",
    response_part = "<|start_header_id|>assistant<|end_header_id|>\n\n",
)
```

Fine-Tune the Model

```
trainer.train()
```

[Post-Training] Set Model for Inference

```
FastLanguageModel.for_inference(model)
```

Create an Input for Inference

```
messages = [
    {"role": "user",
     "content": "Continue the fibonnaci sequence: 1, 1, 2, 3, 5, 8,"},
]
```

Prepare the Input for the Model

```
inputs = tokenizer.apply_chat_template(
    messages,
    tokenize=True,
    add_generation_prompt=True,
    return_tensors="pt",
).to("cuda")
```

Create a TextStreamer Object for Real-Time Output

```
text_streamer = TextStreamer(
    tokenizer,
    skip_prompt=True
)
```

Generate Inference from the Model

```
_ = model.generate(
    input_ids = inputs,
    streamer = text_streamer,
    max_new_tokens = 128,
    use_cache = True,
    temperature = 1.5,
    min_p = 0.1
)
```

Example output:

```
Here is the continued Fibonacci sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34
```

Save the LoRA Adapter

```
model.save_pretrained("lora_model")
tokenizer.save_pretrained("lora_model")
```

Save the Model in GGUF Format

```
model.save_pretrained_gguf("model_16bit_gguf", tokenizer, quantization_method="f16")
```

Recap (1/3)

- LLM finetuning adapts pre-trained models for specific tasks
  - Customizes model behavior
  - Improves performance on target domains
- Key benefits:
  - More efficient than training from scratch
  - Cost-effective solution
  - Enables quick adaptation to new requirements
  - Accessible to smaller organizations

Recap (2/3)

Finetuning pipeline:

1. Create virtual environment
2. Import required libraries
3. Load pre-trained model
4. Add LoRA adapters
5. Preprocess data
6. Fine-tune model
7. Run inference
8. Save model

Recap (3/3)

Challenges to consider:

- Dataset quality is crucial
- Computational requirements
- Risk of overfitting
- Ethical considerations
- Bias detection and mitigation