AI MEDICAL ASSISTANT

Reasoning Medical Specialist

Fine-Tuned DeepSeek R1

On a Medical QA Dataset

NAME – ABHISHEK YADAV

MASTER’S COMPUTER SCIENCE(AI)

DEEP LEARNING FOR NPL PROJECT SUMMARY

PROJECT OVERVIEW

In this capstone project, I have created a medical Q&A system using the DeepSeek-R1 LLM and fine-tune it using a chain-of-thought reasoning dataset. The primary objective is to build an intelligent bot that can effectively engage with users by answering questions and providing insights specific to a chosen industry. This project will not only enhance our technical skills but also provide a deep understanding of the chosen industry's nuances, challenges, and trends.

PROJECT LAYOUT

- Download Model
  - Create and setup huggingface API tokens in colab
- Setup model and tokenizer.
- Setup wandb API
- Test DeepSeek R1 on an medical use-case before fine-tuning
  - Run inference on the model
  - Define a test question
  - Tokenize the input
  - Generate a response
  - Decode the response tokens back to text
- Setup key parms
  - Tokens per input
  - Auto detect the data type
  - Load in 4-bit
- Fine tuning & save
  - Download and prepare dataset(verified medical reasoning data set)
  - Training prompt
  - Setup LoRA
- Evaluation (Before and After testing)

Install required Dependencies.

Purpose:

- Unsloth, TRL, PEFT, xformers for fine-tuning, and CUDA-compatible
- Bitsandbytes allows 4-bit/8-bit model optimization
- Unsloth_zoo provides pre-configured models like deepseek

```
# install unsloth
!pip install --force-reinstall --no-cache-dir --no-deps git+https://github.com/unslothai/unsloth.git

# Install TRL, PEFT, and xformers (specific versions)
!pip install trl==0.14.0 peft==0.14.0 xformers==0.0.28.post3

# Install PyTorch and torchvision (correct index URL from official PyTorch)
!pip install torch==2.5.1 torchvision==0.20.1 --index-url https://download.pytorch.org/whl/cu124

!pip install bitsandbytes

!pip install unsloth_zoo
```

Authentication with HuggingFace and check GPU

Purpose:

- Require to load the model form the HuggingFace server using the HuggingFace API_KEY.
- Check for the available GPU for our google colab.

```
# Check HF token
from google.colab import userdata
hf_token = userdata.get('HF_API')
login(hf_token)
```

```
# Check GPU availability
# Test if CUDA is available
import torch
print("CUDA available:", torch.cuda.is_available())
print("GPU device:", torch.cuda.get_device_name(0) if torch.cuda.is_available() else "No GPU")
```

```
CUDA available: True
GPU device: Tesla T4
```

Test DeepSeek R1 on an medical use-case before fine-tuning

Purpose:

- We are using a DeepSeek R1, A distilled virsion of LLaMA 8B.
- **model_name:** Specifies the name of the pre-trained language model you want to load form hugging Face.
- **max_ sequence_length:** maximum number of tokens (word/pieces) the model can handle in one input.
- **2048** is typical for most modern LLMs and balances performance with memory use.
- **Dtype:** data type for model weights is set to none which allows the FastLanguageModel to automatically decide the optimal format based on hardware configuration.
- **Load_in_4bit:** Enables quantization, greatly reduces memory usage and speeds up inference with negligible accuracy loss.
- **Token:** HuggingFace authentication.

```
model_name = "deepseek-ai/DeepSeek-R1-Distill-Llama-8B"
max_sequence_length = 2048
dtype = None
load_in_4bit = True

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = model_name,
    max_seq_length = max_sequence_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
    token = hf_token
)
```

Define a Custom Prompt Template:

- This **prompt template** instructs the model **how to behave** and **how to format** its output.
- It uses **in-context learning**, giving the model a detailed setup and tone.

Prepare Model for Inference:

- **FastLanguageModel.for_inference(model):** Prepares model for efficient response generation.
- **prompt_style.format(question, ""):**Fills the {} placeholders with your actual question.
- **Tokenizer:** Converts the text into numerical tokens the model understands.
- **return_tensors="pt“:** Output in **PyTorch tensor** format.
- **to("cuda"):**Sends the data to the **GPU** for fast processing.
- **response = tokenizer.batch_decode(outputs):** Converts the generated token IDs **back into readable text**.

```
[ ] # Run Inference on the model

    # Define a test question
    question = """A 61-year-old woman with a long history of involuntary urine loss during activities like coughing or
                  sneezing but no leakage at night undergoes a gynecological exam and Q-tip test. Based on these findings,
                  what would cystometry most likely reveal about her residual volume and detrusor contractions?"""

    FastLanguageModel.for_inference(model)

    # Tokenize the input
    inputs = tokenizer([prompt_style.format(question, "")], return_tensors="pt").to("cuda")

    # Generate a response
    outputs = model.generate (
        input_ids = inputs.input_ids,
        attention_mask = inputs.attention_mask,
        max_new_tokens = 1200,
        use_cache = True
    )

    # Decode the response tokens back to text
    response = tokenizer.batch_decode(outputs)


    print(response)
```

Setup the data for fine tuning

- **load_dataset:** Loads the **top 1000 samples** from a Hugging Face dataset.
- **Dataset:** FreedomIntelligence/medical-o1-reasoning-SFT, Contains medical questions, chain-of-thought reasoning, and final responses.
- **EOS_TOKEN:** Signals to the model **where a response ends**.
- **train_prompt_style:** This template provides a structured context:
  Instruction: Tells the model it's a medical expert.
  Question: The clinical question to be answered.
- **def preprocess_input_data:** Takes a batch of examples with question the medical query, complex_CoT for reasoning steps response for final answer.For each example the function will fills the prompt template with question, thought process and response adds the EOS_TOKEN to the end. And return a sictionary with a single key “texts”.
- **finetune_dataset:** Applies the preprocess_input_data function across all dataset rows in batches . Returns a new formatted prompts ready for training.

```python
# Prepare the data for fine-tuning

def preprocess_input_data(examples):
  inputs = examples["Question"]
  cots = examples["Complex_CoT"]
  outputs = examples["Response"]

  texts = []

  for input, cot, output in zip(inputs, cots, outputs):
    text = train_prompt_style.format(input, cot, output) + EOS_TOKEN
    texts.append(text)

  return {
      "texts" : texts,
  }
```

Setup/Apply LoRA finetuning to the model

What is LoRA?

- **LoRA (Low-Rank Adaptation)** is a method to fine-tune large language models efficiently:
- It **freezes** most of the original model weights.
- Only a **few new trainable parameters** (LoRA adapters) are added to specific layers.
- This drastically **reduces GPU memory usage and training time**.
- **model=model:** Base pretrained model (e.g., DeepSeek)
- **r=16: LoRA rank:** controls adapter size and expressiveness
- **target_modules:** LoRA will be applied only to these transformer layers like q_proj, k_proj).
- **lora_alpha=16:** Scaling factor to balance between new and frozen weights.
- **lora_dropout=0:** No dropout applied to LoRA layers (0% randomness)
- **bias="none":** Don't update any bias terms in the original model
- **use_gradient_checkpointing="unsloth":**Saves memory by recomputing layers during backpropagation
- **use_rslora=False:** Disable structured sparsity (a more experimental LoRA variant)
- **loftq_config=None:** Leave LoFTQ quantization config as default (not using it here)

```
model_lora = FastLanguageModel.get_peft_model(
    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 = 3047,
    use_rslora = False,
    loftq_config = None
)
```

Setup Fine-Tuning Trainer with SFTTrainer

What is SFTTrainer?

- SFTTrainer stands for **Supervised Fine-Tuning Trainer**. It is a training utility provided by the **trl** (Transformers Reinforcement Learning) library from Hugging Face, designed specifically for **fine-tuning large language models (LLMs)** with **instruction-based datasets**.

trainer = SFTTrainer(...):

Use the LoRA-modified model, Tokenizer used to encode/decode text, our formatted dataset with prompt-style texts, Tells trainer to read examples from “texts” column then uses maximum token length per saple and then use the proc=1 which shows the number of parallel prepossing process.

args = TrainingArguments(...):

The TrainingArguments block defines how the model is trained. It sets a small batch size with gradient accumulation to simulate a larger batch. Training runs for 1 epoch or 60 steps with a warmup of 5 steps. Mixed precision fp16 or bf16 is used for efficiency. Logging happens every 10 steps, and the adamw_8bit optimizer helps reduce memory use. The learning rate follows a linear schedule, and outputs are saved to the outputs folder. A seed ensures reproducibility.

```python
trainer = SFTTrainer(
    model = model_lora,
    tokenizer = tokenizer,
    train_dataset = finetune_dataset,
    # dataset_text_field = "texts",
    formatting_func = lambda examples: examples["texts"],
    max_seq_length = max_sequence_length,
    dataset_num_proc = 1,

    args = TrainingArguments(
        per_device_train_batch_size = 2,
        gradient_accumulation_steps = 4,
        num_train_epochs = 1,
        warmup_steps = 5,
        max_steps = 60,
        learning_rate = 2e-4,
        fp16 = not is_bfloat16_supported(),
        bf16 = is_bfloat16_supported(),
        logging_steps = 10,
        optim = "adamw_8bit",
        weight_decay = 0.01,
        lr_scheduler_type = "linear",
        seed = 3407,
        output_dir = "outputs",
    ),
)
```

Setup Weights & Biases (W&B) Logging

- **W&B (Weights & Biases)** is a tool used for tracking machine learning experiments (loss curves, metrics, GPU usage, etc.).
- userdata.get("WANDB_API"): retrieves your W&B API key securely (in Colab) and authenticates your session so logs can be sent to your W&B account.
- Starts a new experiment log under the project: **"Fine-tune-DeepSeek-R1-on-Medical-CoT-Dataset"**.
- job_type="training“: labels the run type.
- anonymous="allow“: enables logging even if you're not logged into W&B via browser.

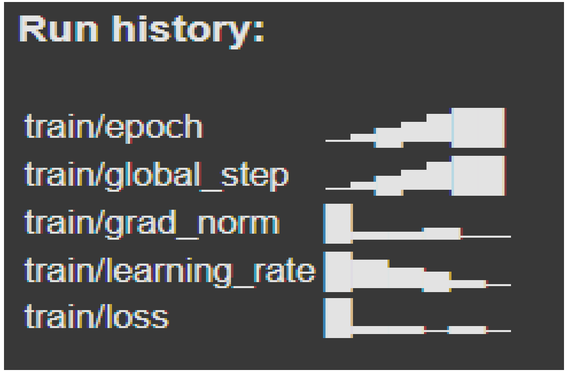
trainer_stats = trainer.train()

- Calls the .train() method of the SFTTrainer.
- This triggers the full training loop.
  Iterates over the dataset.
  Optimises the model using the defined training arguments.
  Logs metrics to the console and W&B.

```python
from google.colab import userdata
wnb_token = userdata.get("WANDB_API")
# Login to WnB
wandb.login(key=wnb_token) # import wandb
run = wandb.init(
    project='Fine-tune-DeepSeek-R1-on-Medical-CoT-Dataset',
    job_type="training",
    anonymous="allow"
)
```

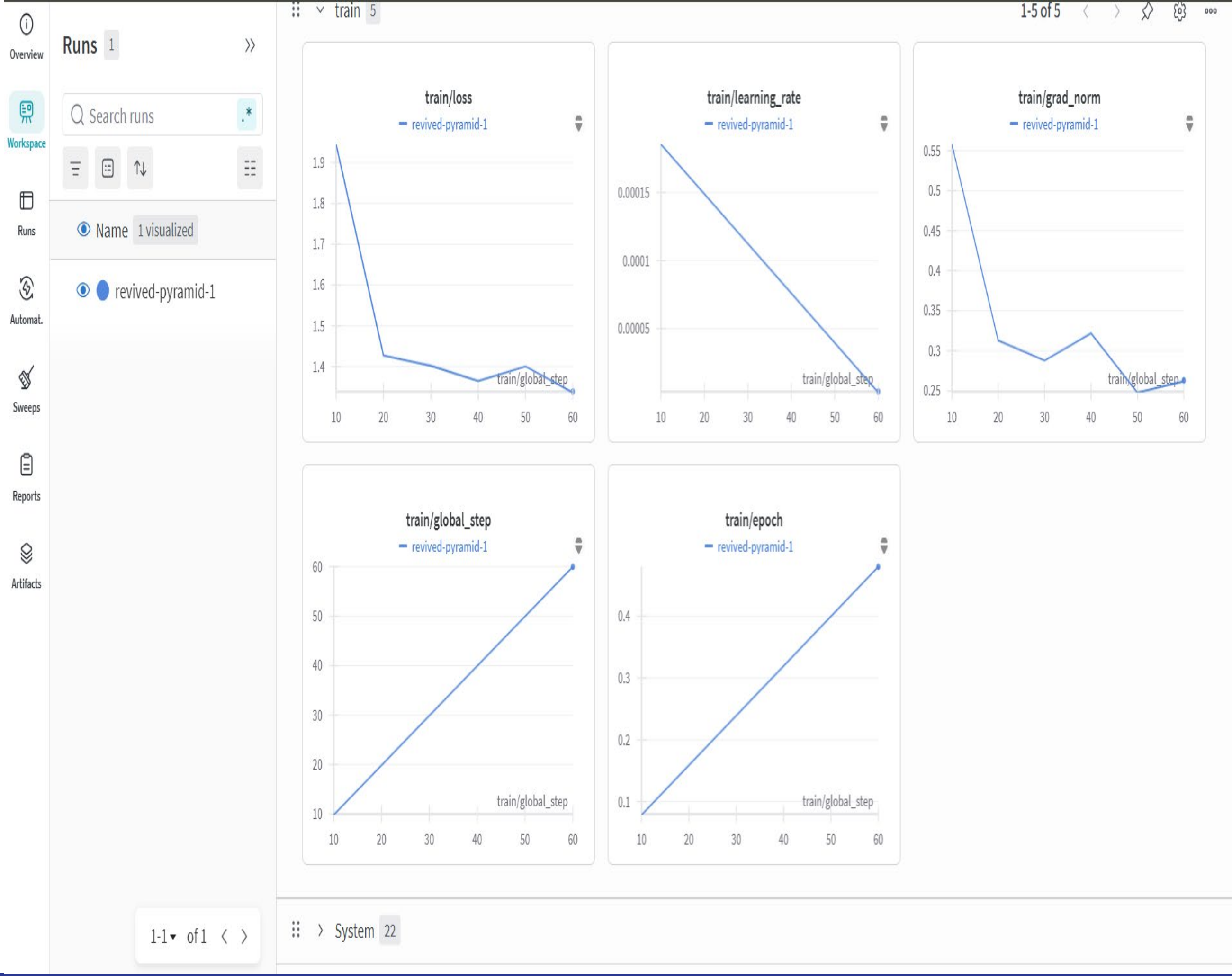
```python
trainer_stats = trainer.train()
```

```python
wandb.finish()
```



Run summary:

total_flos	1.6656601164742656e+16
train/epoch	0.48
train/global_step	60
train/grad_norm	0.26311
train/learning_rate	0.0
train/loss	1.3388
train_loss	1.4811
train_runtime	1044.9609
train_samples_per_second	0.459
train_steps_per_second	0.057



Testing after fine-tuning

- The model is asked to **identify the likely underlying condition** (e.g., colon cancer, poor dental hygiene, etc.)
- Puts the fine-tuned model in **inference mode**, disabling training-specific features like dropout.
- Formats the question using the same **prompt_style** used during training (important for consistency). Converts it to **PyTorch tensors** and moves to **GPU** for faster inference.
- Uses the fine-tuned model to generate a response.
- **max_new_tokens=1200** allows for a long, detailed output.
- **use_cache=True:** makes generation faster by caching intermediate layers.
- Converts generated tokens back into human-readable text.
- Splits the output at "### Answer:“to extract **just the model's answer**.

```python
question = """A 59-year-old man presents with a fever, chills, night sweats, and generalized fatigue,
              and is found to have a 12 mm vegetation on the aortic valve. Blood cultures indicate gram-positive, catalase-negative,
              gamma-hemolytic cocci in chains that do not grow in a 6.5% NaCl medium.
              What is the most likely predisposing factor for this patient's condition?"""


FastLanguageModel.for_inference(model_lora)


# Tokenize the input
inputs = tokenizer([prompt_style.format(question, "")], return_tensors="pt").to("cuda")


# Generate a response
outputs = model_lora.generate (
    input_ids = inputs.input_ids,
    attention_mask = inputs.attention_mask,
    max_new_tokens = 1200,
    use_cache = True
)


# Decode the response tokens back to text
response = tokenizer.batch_decode(outputs)


print(response[0].split("### Answer:")[1])
```

TOOLS AND TECHNOLOGIES

- Google Colab Notebook
- HuggingFace (Model Hosting, Dataset)
- Torch
- Unsloth (optimized Fine-tuning)
- Weight & Biases (Expirement tracking)