D

Appendix D — Fine-Tuning Script

D.1 Step 1: Installation of Dependencies

```
!pip install -q transformers datasets peft accelerate
    bitsandbytes
```

This command installs the core libraries required for fine-tuning the LLaMA-3.2-3B model in the Colab environment:

- `transformers`: HF's library for loading and using pre-trained models and tokenizers.
- `datasets`: Provides access to standard datasets and preprocessing tools.
- `peft`: PEFT library enables the LoRa technique, reducing memory and compute requirements.
- `accelerate`: Simplifies device placement and supports multi-GPU or CPU training with minimal configuration.
- `bitsandbytes`: Allows for 4-bit and 8-bit quantization of model weights, enabling more memory-efficient training on limited hardware.

D.2 Step 2: Authentication to Hugging Face

```
from huggingface_hub import login
login("YOUR_TOKEN")  # Replace with own HF token
```

Authenticates access to Hugging Face Hub to download the base model. In this documentation, the used token was removed and replaced with the placeholder “YOUR_TOKEN”, as tokens are sensitive data and should not be shared.

D.3 Step 3: Tokenizer Setup

```
model_name = "meta-llama/Llama-3.2-3B-Instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token  # Sets padding token
```

This step loads the tokenizer associated with the LLaMA-3.2-3B-Instruct model from the HF repository. As the model is subject to usage restrictions, access had to be explicitly requested first and approved via HF’s platform, including acceptance of Meta’s license agreement. The tokenizer converts input text into token IDs that the model can process. Since the original tokenizer does not include a dedicated padding token, the code assigns the end-of-sequence (`eos_token`) as the padding token. This is necessary for batching inputs of varying lengths during training or inference. Padding ensures that all sequences in a batch have the same length by filling shorter ones with the pad token, allowing for efficient parallel processing without affecting model outputs [48]. After the first fine-tuning set-up, this step requires loading the generated adapter as well, which is described at the end of this chapter.

D.4 Step 4: Model Setup

```
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    device_map="auto",          # Auto GPU allocation
    torch_dtype=torch.float16,  # precision
    load_in_4bit=True           # 4-bit quantization
)
```

Here, the base model `model_name` is configured with three optimizations aimed at efficient performance:

- device_map=“auto”: The model dynamically allocates layers across available GPUs/CPUs.
- torch_dtype=torch.float16: Enables mixed-precision training, reducing memory usage by 50% while retaining numerical stability.

- load_in_4bit=True: Applies 4-bit quantization via the QLoRA method (via bitsandbytes), which reduces memory requirements. Though quantization leads to minor accuracy degradation, evaluations show that LLM performance remains close to full-precision fine-tuning across a range of tasks [49]. Therefore, this was accepted for feasible training.

D.5 Step 5-6: Data Upload

```
from Google.colab import files
uploaded = files.upload()  # Upload JSONL file
shutil.move("big_bang.jsonl", "/content/big_bang.jsonl")  # Fix
    path
```

This enables the user to upload a JSONL file for fine-tuning and relocates the training dataset within Google Colab for better access during the following step. In this example, the file “big_bang.jsonl” is being uploaded.

D.6 Step 7: Dataset Loading

```
dataset = load_dataset(
    "json",
    data_files={"train": "/content/big_bang.jsonl"},
    split="train"
)
```

The dataset for fine-tuning is loaded using HF’s `datasets` library and prepared for training.

D.7 Step 8: Prompt Formatting & Tokenization

```
def format_prompt(example):
    return {"text": f"### Instruction:\nClassify...\n\n### Input
    :\n{example['input']}\n\n### Response:\n{example['output']}"}

dataset = dataset.map(format_prompt)  # Applies formatting

tokenized_dataset = dataset.map(
    lambda x: tokenizer(
        x["text"],
        truncation=True,
        padding="max_length",
        max_length=512
    ),
```

```
    batched=True
)
```

This step prepares the dataset for training by formatting and tokenizing it. The formatting function converts each input–output pair from the JSONL file into an instruction-following prompt, where the task is to classify a given input to a specific EA smell. This format follows the conventions used in instruction-tuned models and helps the model to understand how to map inputs to expected responses. After formatting, each example is tokenized, meaning converted from text into a sequence of token IDs the model can process. The parameter `max_length=512` sets a fixed upper limit for the tokenized input length. This value was chosen as a trade-off between computational efficiency and expressiveness. On the one hand, it is short enough to ensure fast training and compatibility with memory constraints in the Colab environment. On the other hand, it is long enough to accommodate the majority of input texts along with the prompt structure without frequent truncation. Longer sequences would increase memory consumption and training time, while shorter ones risk truncating meaningful information.

D.8 Step 9: LoRA Configuration & Training

```
lora_config = LoraConfig(
    r=8,              # Rank of adaptation matrices
    lora_alpha=32,    # Scaling factor
    target_modules=[  # Modules to adapt
        "q_proj", "k_proj", "v_proj",
        "o_proj", "gate_proj",
        "up_proj", "down_proj"
    ],
    lora_dropout=0.05,
    task_type=TaskType.CAUSAL_LM
)

model = get_peft_model(model, lora_config)  # Wraps model

training_args = TrainingArguments(
    output_dir="./llama3-colab-ft",
    per_device_train_batch_size=2,
    gradient_accumulation_steps=4,  # Effective batch size = 8
    num_train_epochs=4,
    fp16=True,                      # Mixed precision
    logging_steps=1
)
```

```
23
24 trainer = Trainer(
25     model=model,
26     args=training_args,
27     train_dataset=tokenized_dataset,
28     data_collator=DataCollatorForLanguageModeling(tokenizer, mlm
       =False)
29 )
30 trainer.train()  # Starts fine-tuning
```

This step defines and applies the LoRA-based PEFT strategy. A rank (`r`) of 8 and scaling factor (`lora_alpha`) of 32 are used to adapt specific layers within the model: the projection matrices of the attention mechanism (`q_proj`, `k_proj`, `v_proj`, `o_proj`) and components of the feed-forward network (`gate_proj`, `up_proj`, `down_proj`). This configuration modifies only a small fraction of the model's parameters, typically around 0.1%, while the rest remain frozen. A dropout rate of 0.05 is applied to mitigate overfitting, and the `CAUSAL_LM` task type ensures proper autoregressive behavior for language modeling.

The model is then wrapped using the PEFT framework, enabling efficient fine-tuning by training only the newly added LoRA parameters. Training is conducted using HF's `Trainer` API, with an effective batch size of 8, achieved via gradient accumulation over 4 steps with a per-device batch size of 2. Training runs for 4 epochs using mixed-precision (FP16) arithmetic to reduce memory usage and speed up computation. A language modeling data collator is used with `mlm=False`, ensuring dynamic padding without applying masked language modeling, which is appropriate for causal tasks. After running this step, training commences, and the model's loss is recorded at each step to evaluate training progress. The evaluation of the model's loss is described in Chapter 4.2.7.

D.9 Step 10: Model Saving

```
1 model.save_pretrained("llama3-lora-ft")
2 tokenizer.save_pretrained("llama3-lora-ft")
```

After the successful fine-tuning, this saves the LoRA adapters separately from the base model.

D.10 Step 11: Model Upload

```
model.push_to_hub("Forwhatt/LLM_ea_smells", commit_message="Fine-tuning for Big Bang")
tokenizer.push_to_hub("Forwhatt/LLM_ea_smells")
```

This code uploads the adapter to its own HF repository, so it can be accessed again at any time. Colab does not allow the storage of active sessions, meaning that a fine-tuning step is not saved if the adapter is not uploaded to HF.
In the next round of fine-tuning, Step 3 had to be adapted to support continued training on the previously fine-tuned model. Unlike the initial setup, where only the base model was loaded, this updated configuration also loads the LoRA adapter weights from the created HF repository. This ensures that the model retains the knowledge gained from the earlier training phase.

The key change is the addition of the `PeftModel.from_pretrained()` method, which wraps the base model with the previously fine-tuned adapter stored at `adapter_repo`. This setup allows fine-tuning to resume from the intermediate state captured by the adapter, rather than needing to start from scratch.

```
# ALTERNATIVE STEP 3: Set up model and tokenizer
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import PeftModel
import torch

base_model_name = "meta-llama/Llama-3.2-3B-Instruct"  # Original Model
adapter_repo = "Forwhatt/LLM_ea_smells"

tokenizer = AutoTokenizer.from_pretrained(base_model_name, use_auth_token=True)
tokenizer.pad_token = tokenizer.eos_token

model = AutoModelForCausalLM.from_pretrained(
    base_model_name,
    device_map="auto",
    torch_dtype=torch.float16,
    load_in_4bit=True,
    use_auth_token=True
)

# Load LoRA adapter
model = PeftModel.from_pretrained(model, adapter_repo, use_auth_token=True)
```