DR. MURAT ŞİMŞEK

22.05.2025

GENERATIVE AI FINAL PROJECT REPORT

UMUT AKMAN

230201004

ABSTRACT

This report outlines the process and results of a supervised fine-tuning (SFT) project using the **Llama 3.1 8B model** and the **Azamorn/tiny-codes-csharp dataset**. We leverage Unsloth's `FastLanguageModel` and `trl.SFTTrainer` with LoRA adapters to adapt the base model for C# code generation. Key contributions include data preparation, adapter-based training, runtime monitoring, and deployment-ready artifacts (Hugging Face adapter and GGUF quantized model).

INTRODUCTION

PROBLEM STATEMENT:

Many companies, particularly in regulated or confidential domains, hesitate to use public AI services like ChatGPT or Copilot due to data privacy concerns. They lose out on productivity gains—such as rapid code prototype generation—because they cannot risk exposing proprietary code or project details to external APIs.

PROPOSED SOLUTION:

Provide a locally hosted AI model specialized for C# development. By fine-tuning a base LLaMA 3.1 (8B) model on C# instruction–code pairs and deploying it as a LoRA adapter or GGUF, .NET teams can leverage AI-assisted code generation entirely within their secure environment. This approach accelerates development, preserves data confidentiality, and—when scaled with larger models and richer datasets—can deliver significant value to enterprise workflows.

METHODOLOGY AND MATERIALS

MATERIALS AND ENVIRONMENT

- **Hardware:** Google Colab with NVIDIA A100 Tensor Core GPU
- **Key libraries & tools:**
  - unsloth
  - torch
  - datasets (Hugging Face dataset loading)
  - trl (SFTTrainer)
- **Dataset:** Azamorn/tiny-codes-csharp (instruction and output fields) hosted on Hugging Face
- **Base Model:** meta-llama/Llama-3.1-8B-Instruct

LOADING THE MODEL AND TOKENIZER

```
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/Meta-Llama-3.1-8B",
    max_seq_length=2048,
    dtype=None,
    load_in_4bit=True,
)
```

Llama 3.1 8B is instruction-tuned and supports long contexts, making it suitable for code generation tasks and easy to run locally.

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 = 3407,
    use_rslora = False,
    loftq_config = None,
)
```

LoRA (Low-Rank Adaptation) injects trainable rank-r matrices into attention blocks, freezing base weights. Dramatically lower memory and compute cost during fine-tuning; training only 1-10% of parameters.

DATA PREPARATION

```
alpaca_prompt = """Below is an instruction that describes a task. \
Write an output that completes the request using C#.

### Instruction:
{}

### Output:
{}"""
EOS_TOKEN = tokenizer.eos_token

def formatting_prompts_func(examples):
    instrs = examples["instruction"]
    outs   = examples["output"]
    texts = [alpaca_prompt.format(i, o) + EOS_TOKEN
             for i, o in zip(instrs, outs)]
    return {"text": texts}

dataset = load_dataset("Azamorn/tiny-codes-csharp", split="train")
dataset = dataset.map(formatting_prompts_func, batched=True)
```

TEMPLATE:

Alpaca-style prompt with Instruction and Output fields.

DATASET:

Azamorn/tiny-codes-csharp provides paired examples for supervised fine-tuning. Here is a data snippet:

instruction string	output string

Create a C# script snippet that Transforms Low Transportation: Public Transit Schedules for Decision Making for Beginners. Use if/else or switch/case statements to conditionally perform different actions based on the Consent. Dry-run, then include comments that outline the control flow and how you handle different scenarios.	Here is a possible implementation of this functionality in C#:
	
	// dry run
	var consent = "Low";
	Console.WriteLine(TransformLowTransportationSchedulesForDecisionMaking(consent));
	
	// actual function
	public string TransformLowTransportationSchedulesForDecisionMaking(string consent)
	{
	switch (consent)
	{
	case "High":
	return $"Transformed schedules for high transportation consent level.";
	
	case "Medium":
	return $"Transformed schedules for medium transportation consent level.";
	
	default:
	return $"Transformed schedules for low transportation consent level without personalized information.";
	}
	}
	
	In this example, we use a switch statement to check the value of `consent`. If the value is "High", the function returns a message indicating that the transformed schedules were created for high transportation consent levels. Similarly, if the value is "Medium," the function returns another message indicating that the transformed schedules were created for medium transportation consent levels. Otherwise, the function returns a final message indicating that the transformed schedules were created for low transportation consent levels but without any personalized information.

SETTING UP THE TRAINER

```
class HeartbeatCallback(TrainerCallback):
    def on_step_end(...):
        if state.global_step % 1000 == 0:
            print(f"[Heartbeat] step {state.global_step}")


trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=dataset,
    dataset_text_field="text",
    max_seq_length=2048,
    dataset_num_proc=2,
    packing=False,
    callbacks=[HeartbeatCallback(heartbeat_steps=1000)],
    args=TrainingArguments(
        per_device_train_batch_size=2,
        gradient_accumulation_steps=4,
        warmup_steps=5,
        num_train_epochs=1,
        learning_rate=2e-4,
        optim="adamw_8bit",
        weight_decay=0.01,
        lr_scheduler_type="linear",
        fp16=not is_bfloat16_supported(),
        bf16=is_bfloat16_supported(),
        logging_strategy="steps",
        logging_steps=1,
        save_strategy="steps",
        save_steps=500,
        seed=3407,
        output_dir="outputs",
        report_to="none",
    ),
)
```

Used trl.SFTTrainer for supervised fine-tuning. Heartbeat logs training progress every 1,000 steps and stop idling from Google Colab.

Key hyperparams:

- Batch size per device: 2
- Gradient accumulation: 4
- Epochs: 1
- Warmup steps: 5
- Learning rate: 2e-4
- Optimizer: 8-bit AdamW

These are the recommended configurations from Unsloth.

TRAINING THE MODEL

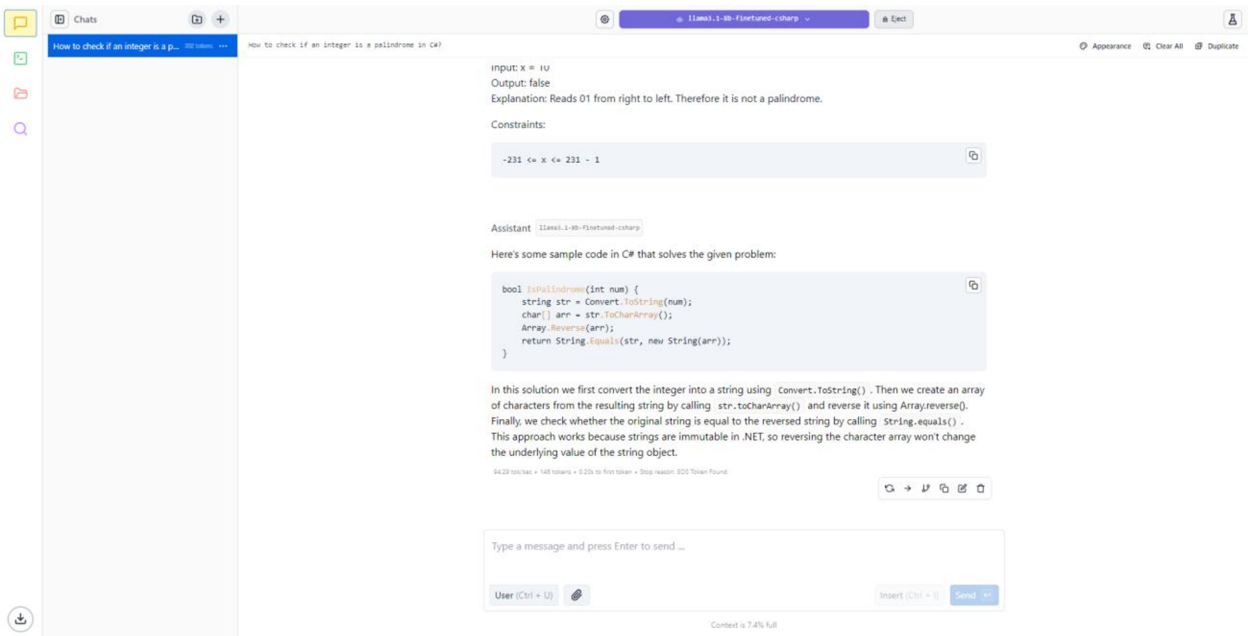
```
trainer_stats = trainer.train()
```

Here is the monitoring results of the training process:

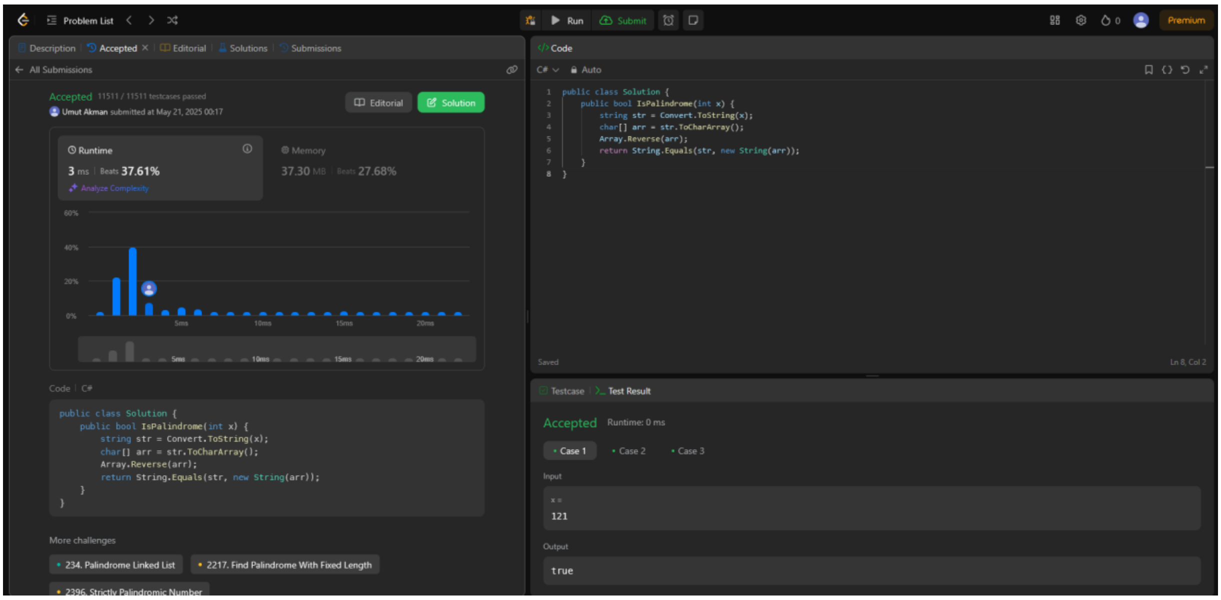
Metric	**Value**
Training runtime	31244 seconds (520 minutes)
Peak GPU memory (total)	7.623 GB

ADDITIONAL STEPS

Quantized our model to a 4-bit q4_k_m GGUF file to make our AI lightweight and easy to run while keeping 95% of our original code quality. Designed for CPU-only inference via llama.cpp or tools like LM Studio to test and run locally. Here is an example, a solution to LeetCode problem 9:



And we can see that it solves the problem:



DISCUSSION

STRENGTHS:

- **Low GPU-memory footprint**: Using 4-bit loading and LoRA adapters keeps peak VRAM usage under 20% of total capacity on Colab's A100.
- **Rapid adaptation**: The model's training loss stabilized within the single epoch, indicating quick convergence given the specialized dataset.
- **High-quality outputs**: Generates syntactically correct C# methods and class structures for a variety of tasks.

CHALLENGES:

- **Output completeness**: Occasionally omits closing braces or trims code blocks, requiring minor manual post-editing.
- **Prompt dependency**: Relies on the Alpaca-style prompt template for best performance; alternative prompting styles may require additional tuning.
- **Dataset limitations**: Publicly available C# instruction–code datasets are small; scaling requires curated, larger corpora.

FUTURE WORK:

1. **Multi-file generation**: Extend the pipeline to generate and link multiple C# files (e.g., data models, services, tests).
2. **Quantization trade-offs**: Evaluate the q5_k_m and q8_0 methods to balance model size, inference speed, and code fidelity.
3. **Enterprise scaling**: Fine-tune with larger models (13B+) and extensive proprietary datasets to deliver on-premise AI for .NET teams in confidential environments.

CONCLUSION

We successfully fine-tuned Llama 3.1 8B using LoRA for C# code generation. The approach demonstrates that adapter-based SFT on a small specialized dataset can yield practical code-generation capabilities. Future work will explore multi-file project generation, more diverse datasets, and evaluation on larger instruction sets.

REFERENCES

1. Unsloth documentation and notebooks (Llama3.1_Alpaca)
2. Azamorn/tiny-codes-csharp dataset on Hugging Face
3. Meta Llama-3.1-8B-Instruct model card