

---

# Parameter Efficient Fine-Tuning (PEFT) of a Large Language Model a GPU

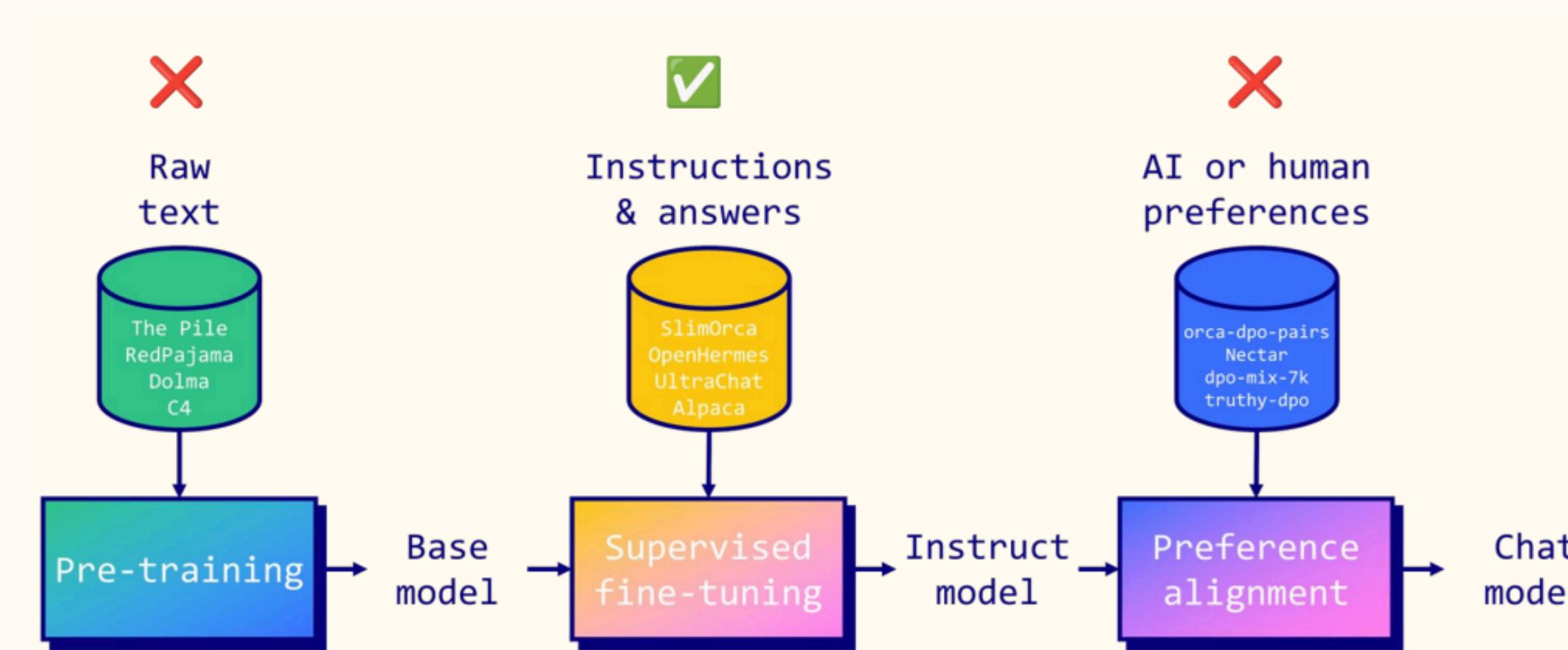
Wasu  
Hugo Legrand

---

## Table of Content

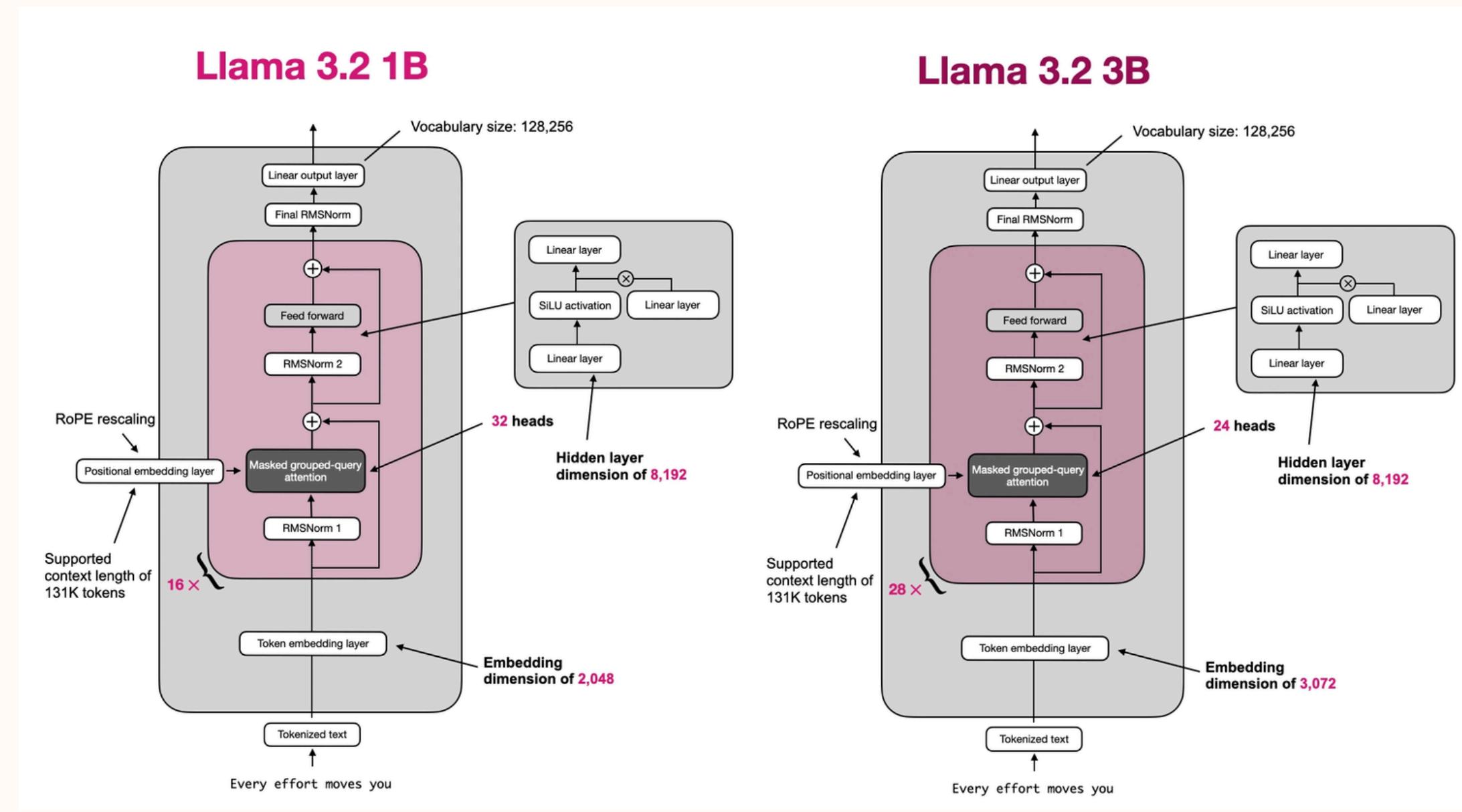
- 1 Introduction
- 2 Model: Llama-3.2
- 3 Technique for PEFT: LoRA
- 4 Additional Technique: Checkpoint
- 5 Results
- 6 Improvement
- 7 Changing foundation LLMs
- 8 Conclusion

## Introduction



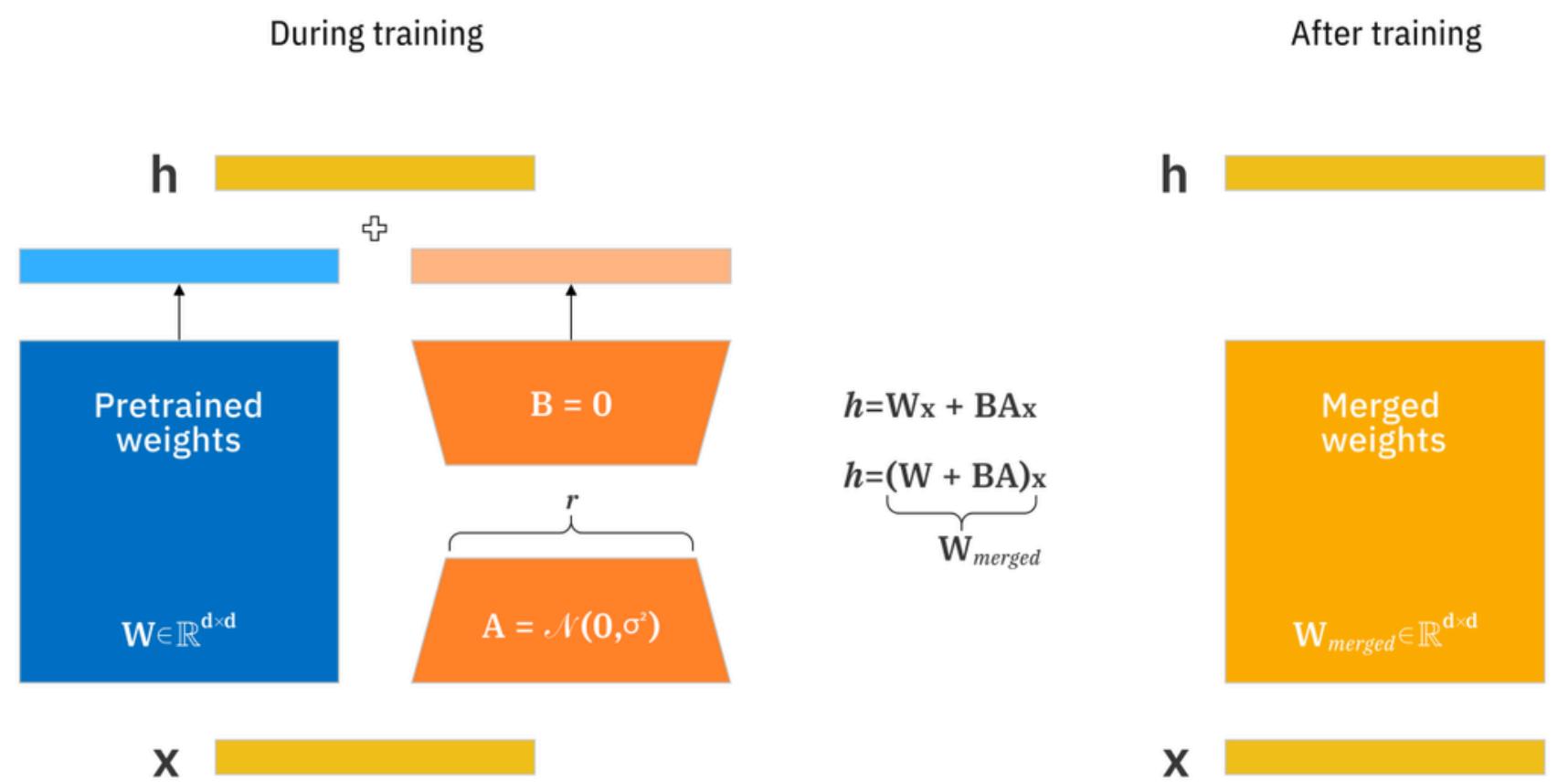
- Dataset: Finetome 100K dataset
- Model: Unslloth/Llama-3.2
- Technique for PEFT: LoRA
- UI: Huggingface Gradio

# Model: Llama-3.2



- Llama 3.2 collection is a collection of pretrained and instruction-tuned generative models in 1B and 3B sizes.
- Llama 3.2 instruction-tuned text only models are optimized for multilingual dialogue use cases, including agentic retrieval and summation tasks.
- Problems
  - Computational Cost
  - Model Collapse
  - Catastrophic forgetting

## Technique for PEFT : LoRA (Low-Rank Adaptation)



- Freeze the original weights and parameters of the model as they are.
- Add a lightweight addition, called a low-rank matrix, which is applied to new inputs to get results specific to the context.
- LoRA utilizes the concept of lower-rank matrices to make the model training process extremely efficient and fast.

---

## Additional Technique: Checkpoint

```
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, # Can make training 5x faster for short sequences.  
    args = TrainingArguments(  
        per_device_train_batch_size = 2,  
        gradient_accumulation_steps = 4,  
        warmup_steps = 5,  
        num_train_epochs = 1, # Set this for 1 full training run.  
        # max_steps =None,  
        learning_rate = 2e-4,  
        fp16 = not is_bfloat16_supported(),  
        bf16 = is_bfloat16_supported(),  
        logging_steps = 1,  
        save_steps = 250,  
        save_total_limit = 2,  
        save_strategy = 'steps',  
        optim = "adamw_8bit",  
        weight_decay = 0.01,  
        lr_scheduler_type = "linear",  
        seed = 3407,  
        output_dir = drive_output_dir,  
        report_to = "none", # Use this for WandB etc  
        resume_from_checkpoint = resume_from_checkpoint,  
    ),  
)
```

- Since fine-tuning the LLM requires high computational resources (like GPU)
- So, We use free-GPU from Colab, T4-GPU.
- However, T4-GPU and allowable session time of Colab is still not enough for training. So, we utilized checkpoint technique.

# Results

link: <https://training-plan-generator.streamlit.app>

The screenshot shows the AI Running Training Plan Generator app interface. On the left, there's a sidebar titled "Training Plan Criteria" with fields for "Your Name" (set to "Runner"), "Goal Race" (set to "5K"), "Current Level" (set to "Beginner"), "Training Duration (weeks)" (set to 8), and "Days per week" (set to 5). Below these is a "Optional Details" button and a large red "Generate Training Plan" button. The main area features a title "AI Running Training Plan Generator" with a running icon, a subtitle "Generate personalized running training plans powered by AI", and a green success message "✓ Training plan generated successfully!". The "Your Training Plan" section details a 5K training plan across four weeks. Week 1 includes an easy run, tempo run, and long run. Week 2 includes tempo runs and easy runs. Week 3 includes interval and tempo runs. Week 4 includes tempo runs and easy runs. The "Plan Summary" section on the right lists the goal (5K), level (Beginner), duration (8 weeks), and training days (5 per week). A "Download as Text" button is also present.

**Training Plan Criteria**

Your Name  
Runner

Goal Race  
5K

Current Level  
Beginner

Training Duration (weeks)  
8

Days per week  
5

Optional Details

Generate Training Plan

**AI Running Training Plan Generator**

Generate personalized running training plans powered by AI

✓ Training plan generated successfully!

**Your Training Plan**

Here is a 5K training plan that meets your requirements:

**Week 1:** Easy Run

- Monday (Easy Run): 10-minute easy run at a slow pace with no intensity. Pace target: 7-8 min/mile
- Wednesday (Tempo Run): 12-minute tempo run at a moderate pace with the same intensity as the previous day's easy run.
- Friday (Long Run): 20-minute long run at an easy to moderate pace, which is about half of your goal race distance.

**Week 2:** Tempo Run and Easy Run

- Monday (Tempo Run): 12-minute tempo run at a moderate pace with a slightly slower intensity than the previous day's easy run. Pace target: 7-8 min/mile
- Tuesday (Easy Run): 10-minute easy run at a slow to moderate pace, which is about half of your goal race distance.
- Wednesday (Tempo Run): 12-minute tempo run at a moderate pace with the same intensity as the previous day's easy run. Pace target: 7-8 min/mile
- Friday (Long Run): 20-minute long run at an easy to moderate pace, which is about half of your goal race distance.

**Week 3:** Interval and Tempo Run

- Monday (Interval Run): 10 minutes on the interval chart with a consistent tempo. Pace target: 7-8 min/mile
- Tuesday (Easy Run): 10-minute easy run at a slow to moderate pace, which is about half of your goal race distance.
- Wednesday (Tempo Run): 12-minute tempo run at a moderate pace with the same intensity as the previous day's easy run. Pace target: 7-8 min/mile
- Friday (Long Run): 20-minute long run at an easy to moderate pace, which is about half of your goal race distance.

**Week 4:** Tempo Run and Easy Run

- Monday (Tempo Run): 12-minute tempo run at a moderate pace with the same intensity as the previous day's easy run. Pace target: 7-8 min/mile
- Tuesday (Easy Run): 10-minute easy run at a slow to moderate pace, which is about half of your goal race distance.
- Wednesday (Tempo Run): 12-minute tempo run at a moderate pace with the same intensity as the previous day's easy run. Pace target: 7-8 min/mile
- Friday (Long Run): 20-minute long run at an easy to moderate pace, which is about half of your goal race distance.

**Plan Summary**

Goal: 5K  
Level: Beginner  
Duration: 8 weeks  
Training Days: 5 per week

Download as Text

Manage app

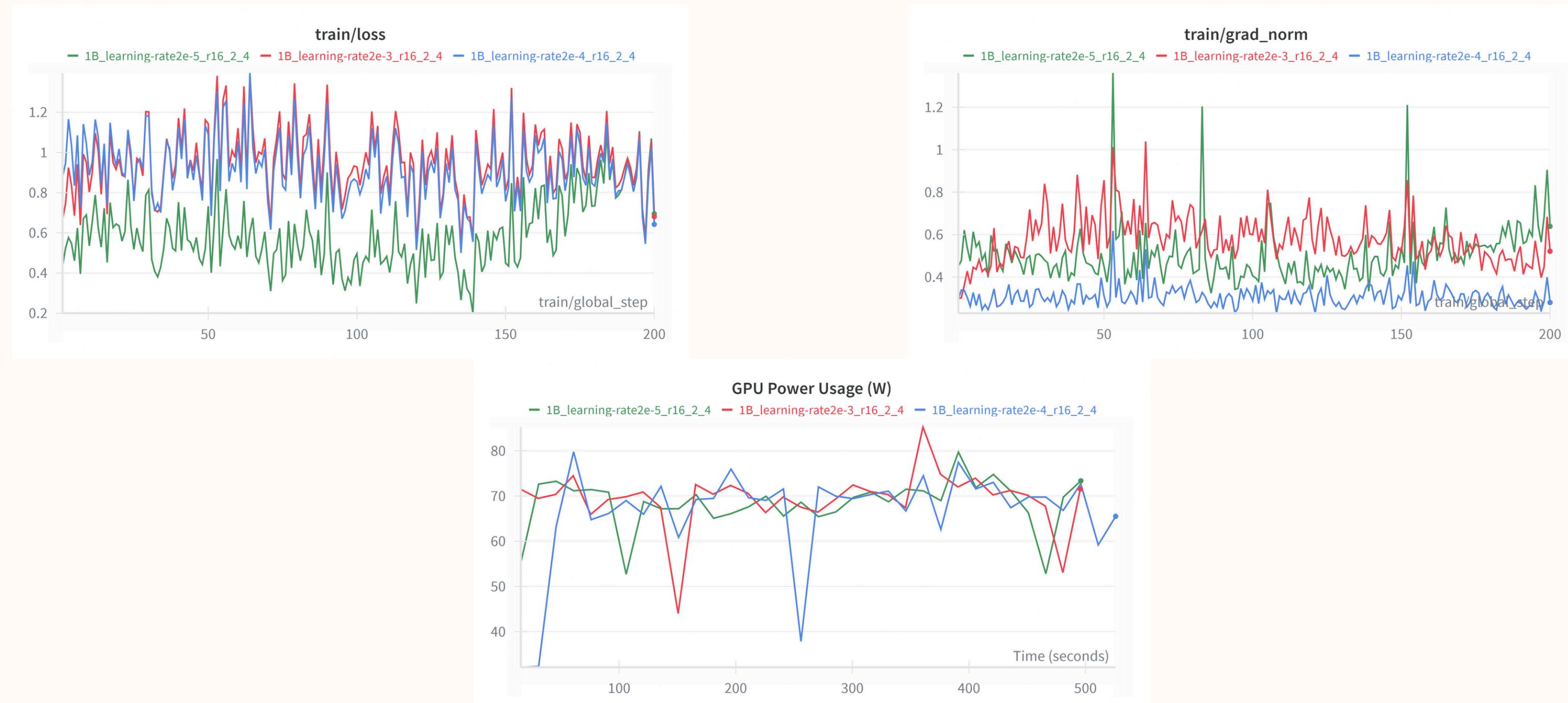
---

## Improvement

- Model-centric-approach
  - Tuning hyperparameters (Rank of LoRA, Learning Rate, etc)
  - Comparing Llama 3.2 1B/3B
- Objective
  - To improve the performance of the LLM.
  - To study the effects of hyperparameters.

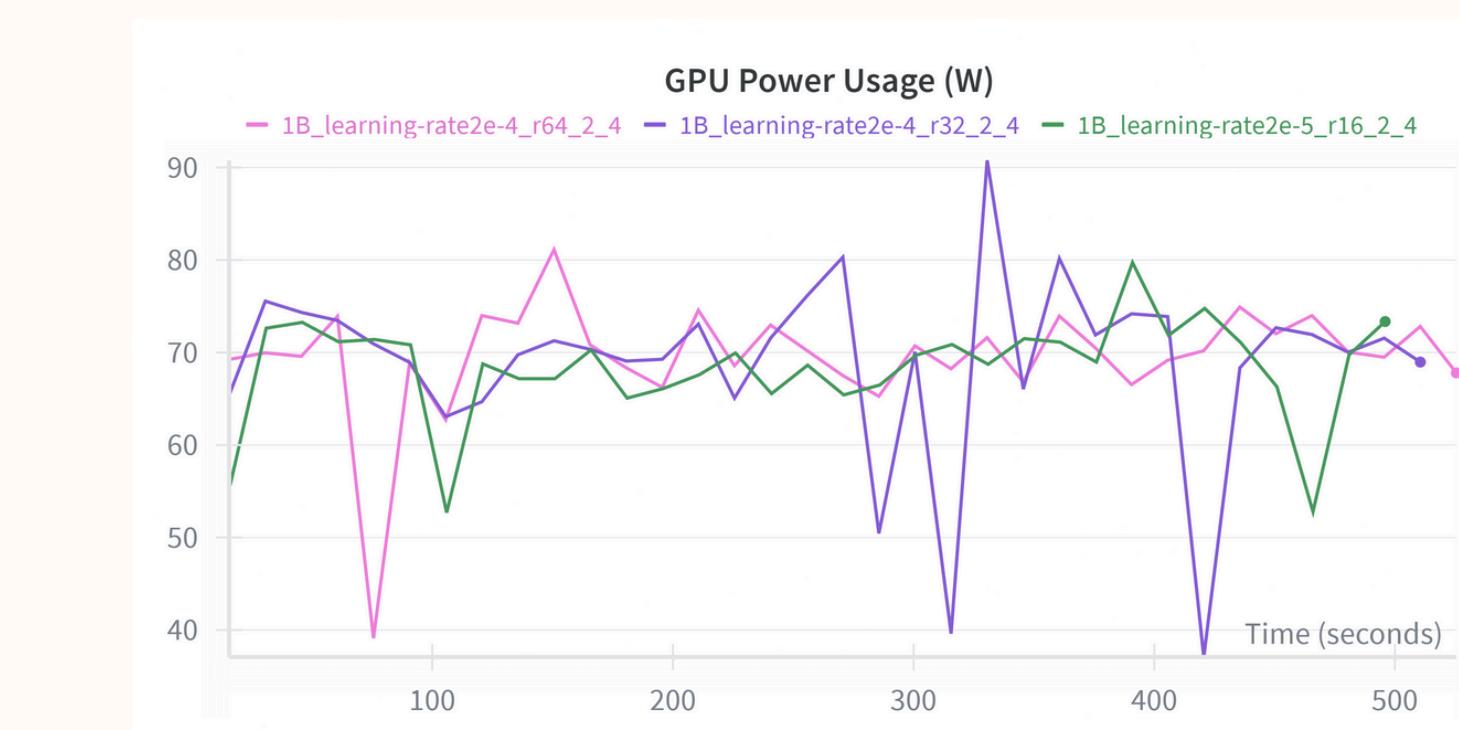
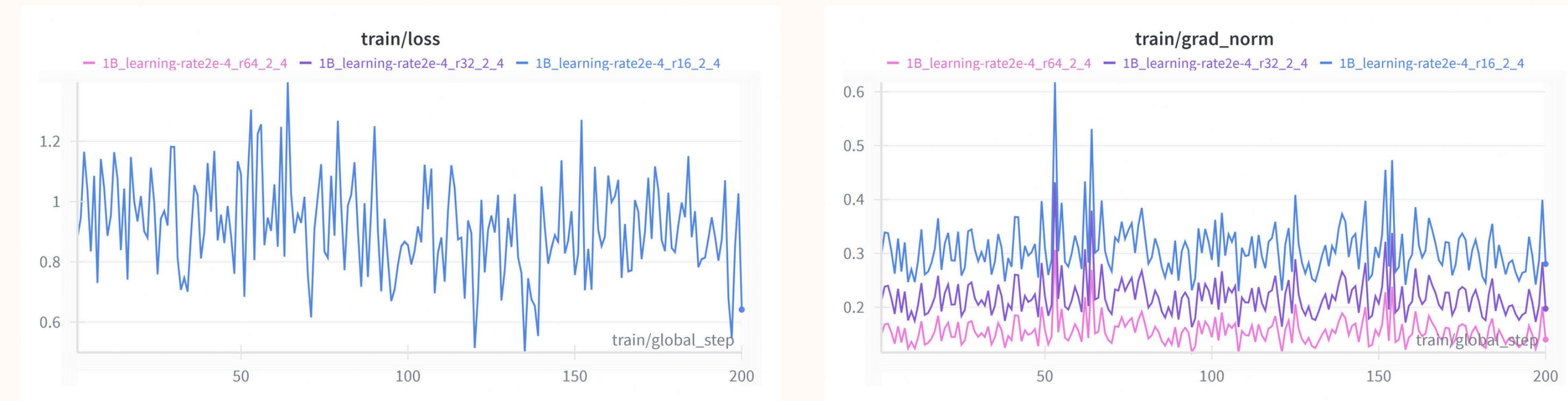
## Improvement Result: Tuning Hyperparameters (Learning Rate)

- Base Case: Llama 3.2 1B, per\_device\_train\_batch\_size = 2, gradient\_accumulation\_steps = 4, r = 16
- Learning Rate: varying 2e-3, 2e-4, 2e-5



## Improvement Result: Tuning Hyperparameters (Rank of LoRA)

- Base Case: Llama 3.2 1B, per\_device\_train\_batch\_size = 2, gradient\_accumulation\_steps = 4, r = 16
- Rank: r = 16, 32, 64



---

## Improvement Result: Tuning Hyperparameters (per\_device\_train\_batch\_size, gradient\_accumulation\_steps)

- Base Case: Llama 3.2 1B, per\_device\_train\_batch\_size = 2, gradient\_accumulation\_steps = 4, r = 16
- per\_device\_train\_batch\_size = 4, gradient\_accumulation\_steps = 8



---

## Improvement Result: Changing the model

- Base Case: Llama 3.2 1B, per\_device\_train\_batch\_size = 2, gradient\_accumulation\_steps = 4, r = 16
- Llama 3.2 3B



## Evaluation on a rephrased input of the FineTome dataset :

 Conversational View

Explain how a ternary operator functions within a recursive method. Illustrate your explanation with a suitable programming example.

A ternary operator is a conditional operator that provides a shorter syntax for an if-else condition. It has three parts: a condition/expression to evaluate, a result for when the condition is true, and a result for when the condition is false.

The syntax for a ternary operator is: condition ? result\_if\_true : result\_if\_false;

In recursive methods, the ternary operator can play a vital role to set a base case or termination condition for the recursive calls.

For instance, we can depict a recursive method for calculating the factorial of a number using a ternary operator in Python as follows:

```
python
def factorial(n):
    return 1 if n == 0 or n == 1 else n * factorial(n - 1)
```

Copy

In this example, the provided function `factorial` calculates the factorial of a number `n`. Here, the ternary operator is used as the base case for the recursion. When `n` equals `0` or `1`, the function stops calling itself and returns `1`.

If the condition `n == 0 or n == 1` is not met (that is, `n` is neither `0` nor `1`), then the function executes `n * factorial(n - 1)`. This is a recursive call to the same function `factorial`, with an argument reduced by `1`. This continues until `n` reaches `0` or `1`, and then all nested calls will return their results, giving us the factorial of the original number `n`.

This is an effective example of how a ternary operator can function within a recursive function - to lead to a base case or termination condition.

# Evaluation on a rephrased input of the FineTome dataset : Base Llama3 1B

## Example:

We want to calculate the sum of all elements in a list of integers. We can use recursion to solve this problem. We'll use the ternary operator to simplify the recursive calls.

```
def recursive_sum(lst):
    if len(lst) == 0:
        return 0
    else:
        return lst[0] + recursive_sum(lst[1:])
```

## Explanation:

When we use a ternary operator inside a recursive function, it works as follows:

- If the current recursive call returns a value, we discard it and continue with the next recursive call.
- If the current recursive call returns `None` or an empty value, we discard it and use the value from the previous recursive call as the new value.

In our example, the ternary operator allows us to simplify the recursive calls as follows:

```
def recursive_sum(lst):
    if len(lst) == 0:
        return 0
    else:
        first = lst[0]
        second = recursive_sum(lst[1:])
        return first + second if second else first
```

## Example Usage:

```
numbers = [1, 2, 3, 4, 5]
result = recursive_sum(numbers)
print(result) # Output: 15
```

In this example, the recursive function `recursive_sum` is called with the list `numbers` as its argument. The function first checks if the list is empty, in which case it returns 0. Otherwise, it recursively calls itself with the first element of the list (`lst[0]`) and the remaining elements (`lst[1:]`). The ternary operator allows the function to use the first element of the list as the new value in the recursive call, while also returning the sum of the remaining elements. The final result is the sum of all elements in the list.

# Evaluation on a rephrased input of the FineTome dataset : Llama3 1B with finetuning

A ternary operator is a concise way to express a conditional statement. When used inside a recursive function, it allows the function to make decisions based on previous computations. Here's how it works:

1. The function takes three parameters: the current value, the condition, and the value to return if the condition is true.
2. The function checks the condition. If it is true, it returns the value to be returned.
3. If the condition is false, the function recursively calls itself with the current value, the condition, and the value to return if the condition is true.
4. The recursive call returns the result of the call, which is then returned to the previous call.

Let's consider an example:

```
def recursive_function(n, condition, value_to_return):
    if condition:
        return value_to_return
    else:
        return recursive_function(n, condition, value_to_return)

result = recursive_function(5, True, 10)
print(result) # Output: 10
```

In this example, the `recursive_function` takes three parameters: `n` (the current value), `condition` (the condition to check), and `value_to_return` (the value to return if the condition is true). The function checks the condition and returns the value to be returned if the condition is true. If the condition is false, the function recursively calls itself with the current value, the condition, and the value to return if the condition is true. The recursive call returns the result of the call, which is then returned to the previous call.

In the example provided, the function is called with `n` equal to 5, `condition` set to `True`, and `value_to_return` set to 10. The function checks the condition and returns the value to be returned if the condition is true. In this case, the condition is true, so the function returns the value 10. The returned value is then printed to the console.

# Evaluation on a rephrased input of the FineTome dataset : Base Gemma 270m

## Explanation:

A ternary operator (also known as a conditional operator) is a concise and efficient way to evaluate expressions that can be expressed as a series of binary operations. It's designed to be concise and readable, particularly when dealing with simple expressions. It's particularly useful for situations where you need to perform a series of operations, like comparing values, or performing calculations.

## How it Works:

The ternary operator evaluates a single expression, and it compares the result with the result of the corresponding operation. If the result is *greater than* the result of the operation, the expression is considered equal to the operation, and the operator is executed. Otherwise, the expression is considered *less than* the result of the operation.

## Example:

Let's consider a simple ternary operator that evaluates the expression `2 + 3 * 4`.

1. **Initialization:** The ternary operator is initialized with `0` as the left-hand operand.
2. **Evaluation:** The ternary operator evaluates `2 + 3 * 4`.
3. **Comparison:** The result of the ternary operator is compared to the result of the operation `4`.
4. **Result:** If the result of the ternary operator is greater than 4, the expression is considered equal to the operation. Otherwise, it is considered *less than* the result of the operation.

## Code Example:

```
def ternary_operator_expression(expression):
    """
    Evaluates a ternary operator expression.

    Args:
        expression: A string representing the ternary operator expression.

    Returns:
        The result of the ternary operator expression.
    """

    try:
        result = eval(expression)
        return result
    except Exception as e:
        print(f"Error evaluating ternary operator expression: {e}")
        return 0 # Return 0 if there's an error
    else:
```

# Evaluation on a rephrased input of the FineTome dataset : Base Gemma 270m with finetuning

A ternary operator is a function that takes two input values and returns a value that can be expressed as a ternary sum. It can be used inside a recursive function to evaluate the expression entered into the ternary operator and return the result.

Here's an example in code:

```
def ternary_sum(a, b):
    return ternary_sum_recursive(a, b)

def ternary_sum_recursive(a, b):
    if b == 0:
        return a + b
    elif b == 1:
        return a + a + b
    else:
        return ternary_sum_recursive(a + b, b)

a = 10
b = 15
result = ternary_sum_recursive(a, b)
print(result)
```

In this code, the ternary operator `ternary_sum_recursive` is defined to evaluate the ternary sum of the input `a` and `b`. The ternary sum is defined as `a + b`. The function `ternary_sum_recursive` is declared as `a` and `b`, and its body is declared as `if b == 1: return a + a + b; else: return ternary_sum_recursive(a + b, b);` Where `a + b` represents the ternary sum of the two input values.

The ternary operator uses ternary operators to evaluate the ternary sum of the input values. The ternary sum of the input values is expressed as a ternary sum, which is the sum of the ternary sum of the input values. This function can be used inside a recursive function to evaluate the expression entered into the ternary operator and return the result.

Here is an example of how the ternary operator can be used inside a recursive function:

```
def ternary_sum_recursive(a, b):
    if b == 0:
        return a + b
    elif b == 1:
        return a + a + b
    else:
        return ternary_sum_recursive(a + b, b)
```

In this code, the ternary operator `ternary_sum_recursive` is defined to evaluate the ternary sum of the input values. The ternary sum is defined as `a + b`. The function `ternary_sum_recursive` is declared as `a` and `b`, and its body is declared as 'if b == 1

# Hyperparameter Tuning with Optuna

```
● ○ ●  
  
def objective(trial):  
    # Hyperparameters to tune  
    learning_rate = trial.suggest_float("learning_rate", 1e-6, 1e-4, log=True)  
    warmup_steps = trial.suggest_int("warmup_steps", 5, 50)  
    weight_decay = trial.suggest_categorical("weight_decay", [0.0, 0.001, 0.01])  
    batch_size = trial.suggest_categorical("batch_size", [1, 2, 4])  
    grad_acc = trial.suggest_categorical("grad_acc", [1, 2, 4])  
  
    # Load model here  
  
    # Trainer config with sampled params  
    training_args = SFTConfig(  
        dataset_text_field = dataset_text_field,  
        per_device_train_batch_size = batch_size,  
        gradient_accumulation_steps = grad_acc,  
        learning_rate = learning_rate,  
        warmup_steps = warmup_steps,  
        weight_decay = weight_decay,  
  
        optim = "adamw_8bit",  
        max_steps = 50,          # Hyperband will prune bad configs early  
        logging_steps = 10,  
        report_to = "none",  
        output_dir=f"outputs_trial_{trial.number}",  
  
        save_strategy = "no",    # do NOT save checkpoints during tuning  
    )
```

```
● ○ ●  
  
study = optuna.create_study(  
    direction="minimize",  
    pruner=HyperbandPruner(),  
)  
  
study.optimize(  
    objective,  
    n_trials=10,  
    timeout=None,  
)
```

---

## Conclusion

- Fine-tuning with Unslot on a Colab T4 GPU is feasible but comes with several constraints.
- 1-hour session limits and daily usage caps on free Colab slow down training progress.
- Training already takes a long time; these constraints make it even longer and harder to iterate.
- The fine-tuned model actually performed worse, likely due to overfitting on the small new dataset.
- A learning rate too high (or other hyperparameter choices) may have contributed to the performance drop.