# Evaluation Results: Framework and Issues with Fine-Tuning

In this section, I will walk through the evaluation framework I intended to use, the challenges faced during the fine-tuning process, and an explanation of why the fine-tuning didn't work due to memory constraints (RAM issues). Additionally, I'll suggest possible solutions for tackling these challenges in the future.

## 1. Intended Evaluation Framework

The goal of the evaluation framework was to assess the performance of the fine-tuned **Qwen 2.5 3B-Instruct** model on unseen AI research questions. I implemented the following:

**Evaluation Pipeline:**

- **Metrics:** Accuracy, F1-score, and loss were planned to be used for evaluation. Accuracy measures the percentage of correct answers, F1-score captures the balance between precision and recall, and the loss indicates how well the model's predicted probability distributions match the actual labels.
- **Data Split:** The test set consisted of 10% of the dataset that was held out during training. This ensures that the evaluation is on unseen data and provides an indication of how the model would perform on real-world queries.
- **Batch Processing:** The evaluation was meant to be done in batches to reduce memory consumption and avoid RAM overload during the process.

**Example Code for Evaluation:**

```
from sklearn.metrics import accuracy_score, f1_score
import torch
from torch.utils.data import DataLoader
from transformers import Trainer, TrainingArguments

# Generate predictions on the test set
def evaluate_model(model, dataset):
```

```
    # Set the model in evaluation mode
    model.eval()

    # Prepare the data loader
    dataloader = DataLoader(dataset, batch_size=4)

    # Store predictions and true labels
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for batch in dataloader:
            # Forward pass
            outputs = model(**batch)
            predictions = torch.argmax(outputs.logits, dim=-1)
            labels = batch['labels']

            # Store predictions and labels
            all_preds.extend(predictions.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    # Calculate accuracy and F1-score
    accuracy = accuracy_score(all_labels, all_preds)
    f1 = f1_score(all_labels, all_preds, average='weighted')

    print(f"Accuracy: {accuracy:.4f}")
    print(f"F1-Score: {f1:.4f}")
    return accuracy, f1

# Example usage for evaluation
evaluate_model(model, test_dataset)
```

This framework was intended to provide insights into the model's performance by comparing its predictions to the ground truth and calculating standard metrics such as accuracy and F1-score.

## 2. Fine-Tuning Issue: RAM Crashes

**Problem:**

During the fine-tuning process, I encountered issues related to **out-of-memory (OOM) errors**, which caused the training process to crash. These issues were specifically due to memory constraints while training the **Qwen 2.5 3B-Instruct** model, which has 3 billion parameters.

- **Cause of the Issue:**
  - **Large Model Size**: The **Qwen 2.5 3B** model has 3 billion parameters, and fine-tuning this model requires a significant amount of GPU memory. During the backpropagation and weight update steps, the model's memory footprint increases, leading to **Out of Memory (OOM)** errors, particularly when using large batch sizes or training with multiple layers of the model being updated simultaneously.
  - **Limited GPU/CPU Memory**: The system running the training process had **limited GPU memory** (e.g., 16GB or less), and the batch size and model's memory demand exceeded the available resources. This is especially true for training large models like **Qwen 2.5 3B**.

**Symptoms:**

- The training process started successfully but failed midway with memory errors, usually within the first few epochs.
- The error messages pointed to exceeding available GPU memory (usually `CUDA out of memory` errors).

## 3. Proposed Solutions and Recommendations

Here are the steps I would recommend to mitigate memory issues and optimize the fine-tuning process:

**1. Reducing the Batch Size:**

- One immediate solution to reduce memory consumption is to lower the **batch size**. Smaller batch sizes consume less memory and may prevent OOM errors, although

this could slightly slow down training. For instance, reducing the batch size to 2 or 1 per device may help fit the model into memory.

- **Adjust the batch size dynamically** based on available memory.

```
# Decrease batch size to 2 or 1
train_dataset.set_format(type='torch', columns=['input_ids',
'attention_mask', 'labels'])
```

**2. Gradient Accumulation:**

- **Gradient accumulation** allows the model to perform multiple forward passes with small batch sizes, accumulate gradients, and then update weights only after a larger effective batch size. This helps simulate a larger batch size while keeping memory usage low.

```
# Update the training arguments to use gradient accumulation
training_args = TrainingArguments(
    gradient_accumulation_steps=8,  # Accumulate gradients over 8
steps
    per_device_train_batch_size=2,  # Smaller batch size for
individual steps
)
```

**3. Use of Mixed Precision Training:**

- **Mixed Precision Training** (using FP16 or 16-bit precision instead of FP32) reduces memory usage and speeds up training by utilizing the tensor cores of modern GPUs.

```
# Use mixed precision for memory efficiency
training_args.fp16 = True
```

**4. Use Model Parallelism:**

- If you are working with multiple GPUs, you can use **model parallelism** to split the model across GPUs and reduce the memory load on each device. This can be helpful

for training large models like Qwen 2.5 3B, which can't fit entirely in the memory of a single GPU.

**5. Checkpointing and Restarting:**

- **Checkpointing** allows you to save the model's progress and resume training from the last saved point if an OOM error occurs. This helps prevent starting from scratch if the training process is interrupted.

```
# Save model checkpoints every few steps
training_args.save_steps = 500
trainer.save_model("checkpoint_dir")
```

**6. Consider LoRA (Low-Rank Adaptation):**

- **LoRA** reduces the number of parameters to be fine-tuned by introducing low-rank layers instead of fine-tuning the entire model. This is memory-efficient and reduces the computational burden.

**7. Gradient Checkpointing:**

- **Gradient checkpointing** saves memory by only storing a subset of the intermediate activations in the computation graph and recomputing them during the backward pass. This reduces memory usage at the cost of additional computation.

```
from transformers import Trainer, TrainingArguments

training_args.gradient_checkpointing = True  # Enable gradient
checkpointing
```

## 4. Moving Forward

Since the training did not succeed due to memory issues, the immediate next step would be to:

- **Implement gradient accumulation** and try **smaller batch sizes**.

- Enable **mixed precision** training to reduce memory load.
- If possible, try **gradient checkpointing** to further reduce memory usage.

Additionally, using a smaller model or employing techniques like **model pruning** could help alleviate memory constraints without significantly compromising performance.

Once the fine-tuning process is successful, I would resume the **evaluation framework** outlined in Section 5.1 to assess the model's performance on the test set.