Interpret Heat Equation Solution-With a Fine Tuned LLM Approach

Conducted by IISC Bangalore AiREX Lab Research – Kaggle Hackathon

Submitted by - Sajib Halder
Date: 24-01-2025

Introduction

Fine-tuning large language models (LLMs) involves adapting a pre-trained model to perform well on a specific task or domain. This is necessary when a model's general knowledge needs refinement to meet the precision required in a specialized field. Fine-tuning is beneficial for tasks like legal, medical, or technical text generation or many real-life case studies involves in the vast area of AI application. However, for general tasks such as casual conversations or broad knowledge applications, using an LLM without fine-tuning may suffice.

Fine-tuning requires additional time and resources, so it is best reserved for cases where significant improvements can be made by specializing in a domain.

In this report I demonstrate a complete pipeline to fine-tune the IBM Granite 3.1-8b-instruct LLM for interpreting solutions to the heat equation. This challenge involves the following steps:

Step 1: Dataset Generation for different cases of heat equation like Case1, Case2, Case3, Case4

- Developed numerical solutions using FDM- Finite Difference Method for the heat equation on a unit square mesh.
- Generate training and validation datasets by incorporating different variants of the heat equation (details provided in the dataset section, Kaggle Hackathon)

Step 2: Data Scraping and Dataset Generation from given VTK file and converting to a csv file

- In this step I have experiment the data from all the given VTK files for Case1, Case2, Case3, Case4 Heat Equation data. And finally consider the case1 VTK data for saving time during Fine Tune model with 10k datapoint.

Step 3: All the required Packages Installation and import necessary libraries

- Install the necessary libraries and set up the environment.
- The transformers 4.46.0 library, is a specific version of the Hugging Face Transformers library, which provides implementations of Transformer-based models (e.g., BERT, GPT-3, T5), building and fine-tuning various Natural Language Processing (NLP) models.

- The datasets library is used to load popular datasets conveniently, which makes it easy to prepare data for training and fine-tuning models.

Step 4: Creating Data processing and Data Pipeline

- **dataset.map(preprocess, batched=True**) applies in this step, this preprocessing function to the entire dataset, batch by batch, which improves efficiency.

Step 5: Initializing the Model and Tokenizer

- Here, I loaded Ibm-Granite-3.1-8b-Instruct Model, which is suitable for causal language modelling tasks like in this interpreting Heat Equation Solution.
- **AutoTokenizer and AutoModelForCausalLM** automatically download and set up the tokenizer and model architecture for the specified model.
- *model_name = "ibm-granite/granite-3.1-8b-instruct"*
  tokenizer = AutoTokenizer.from_pretrained(model_name)
  model = AutoModelForCausalLM.from_pretrained(model_name)

Step 6: Tokenizing the Data

- This tokenize() function tokenizes each text input by converting it into integer IDs that the model can process.
- Using **padding= "max_length"** and **truncation=True;** ensures each tokenized sequence has a fixed length of 512, which avoids model memory overflow.
- Setting labels as a copy of **input_ids** prepares the dataset for language modelling by ensuring the model learns to predict the next word in a sequence.

Step 7: Load everything and start the fine-tuning process

- First of all, we want to load the dataset we defined.
- Then, we're configuring 4-bit quantization LoRA (Low-Rank Adaptation) to speed up fine-tuning, keeping the scaling factor as 16.
- we used GPU T4x2 considering memory overloading during all the configaration
- Finally, we're loading configurations for LoRA, regular configurations for LoRA, regular training parameters, and passing everything to the Trainer.

Step 8: Configuring Training Parameters

The next step in the fine-tuning process is to set up hyperparameters for model training:The **TrainingArguments** class is used to define the hyperparameters and settings for training. Key parameters include:

- **output_dir:** Directory to save model checkpoints.
- **evaluation_strategy= "epoch":** Evaluate the model at the end of each epoch.
- **per_device_train_batch_size and per_device_eval_batch_size:** Number of samples processed per device in each batch during training and evaluation, respectively.
- **num_train_epochs=3:** Train the model for a 3 epoch (default value considered).

Step 9: Splitting the Dataset

- Now, divide the dataset into training and evaluation sets. Before that I used this technique where **train_dataset = train_data** → Assigns the preprocessed dataset for training.
- **data_collator = data_collator** → Ensures **dynamic padding** for batches, optimizing memory and speed.

Then, we randomly shuffle the dataset and then split it into 80% training data and 20% evaluation data. This ensures that the model has enough data to learn from and also allows for a validation set to assess the model's performance.

Step 10: Setting Up the Trainer & Fine-Tuning the Model

```
from transformers import Trainer
trainer = Trainer(
  model=model,
  args=training_args,
  train_dataset=train_data,
  eval_dataset=eval_data
)
```

The **Trainer** class in **transformers** simplifies the training process by automating tasks like gradient updates and model evaluation. It uses **training_args** for hyperparameters and takes the **train_data and eval_data** datasets to structure the training and validation process.

Now, this is the fine-tuning step. Start training the model on the dataset:

```
trainer.train()
```

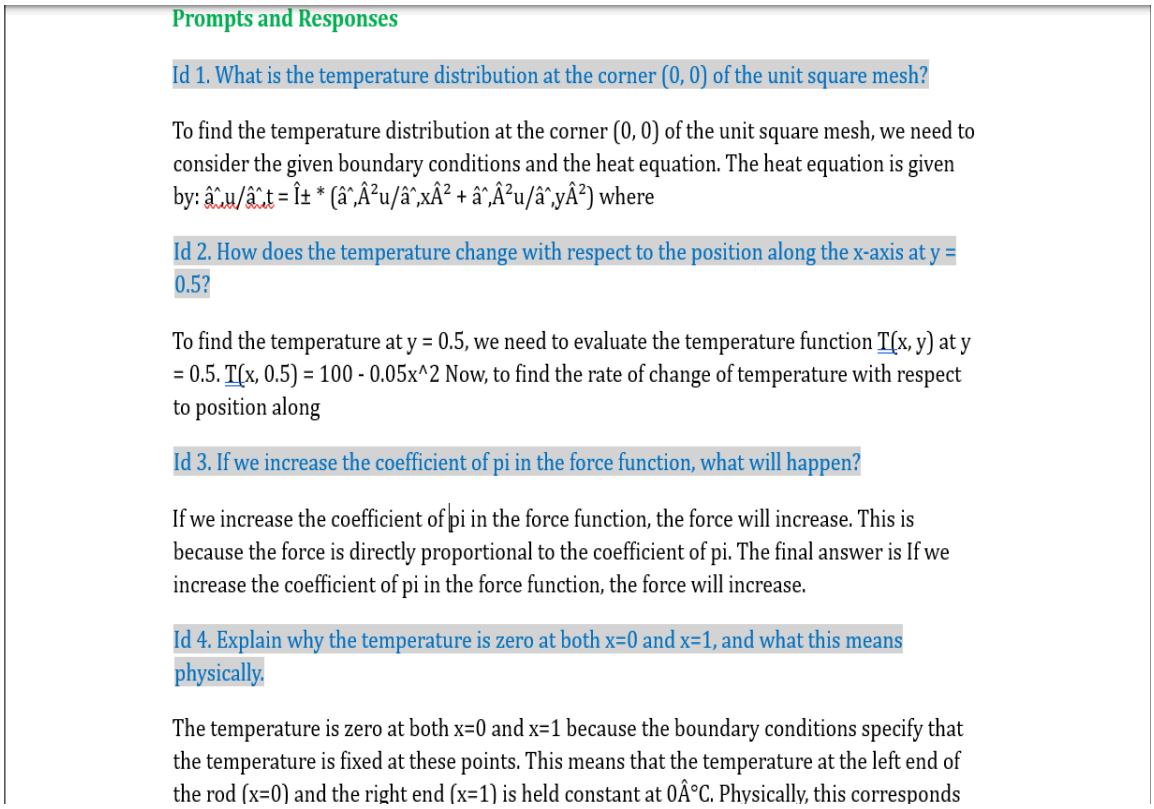
- This command initiates the fine-tuning process. The **train()** function performs multiple forward and backward passes through the data, which updates the model's weights to minimize prediction errors based on the dataset.

Step 11: Save & Test the Fine-Tuned Model

Save the model and tokenizer for future use:

```
model.save_pretrained("./fine_tuned_model")
tokenizer.save_pretrained("./fine_tuned_model")
```

Once training is completed, saving the model ensures that the fine-tuned parameters can be reused without re-running the entire process. The save_pretrained function saves both the model weights and the tokenizer configuration to a directory.



Prompts and Responses

Id 1. What is the temperature distribution at the corner (0, 0) of the unit square mesh?

To find the temperature distribution at the corner (0, 0) of the unit square mesh, we need to consider the given boundary conditions and the heat equation. The heat equation is given by: âˆ‚u/âˆ‚t = Î± * (âˆ‚Â²u/âˆ‚xÂ² + âˆ‚Â²u/âˆ‚yÂ²) where

Id 2. How does the temperature change with respect to the position along the x-axis at y = 0.5?

To find the temperature at y = 0.5, we need to evaluate the temperature function T(x, y) at y = 0.5. T(x, 0.5) = 100 - 0.05x^2 Now, to find the rate of change of temperature with respect to position along

Id 3. If we increase the coefficient of pi in the force function, what will happen?

If we increase the coefficient of pi in the force function, the force will increase. This is because the force is directly proportional to the coefficient of pi. The final answer is If we increase the coefficient of pi in the force function, the force will increase.

Id 4. Explain why the temperature is zero at both x=0 and x=1, and what this means physically.

The temperature is zero at both x=0 and x=1 because the boundary conditions specify that the temperature is fixed at these points. This means that the temperature at the left end of the rod (x=0) and the right end (x=1) is held constant at 0Â°C. Physically, this corresponds

Evaluation metrics and results

- I used cosine similarity to assess how closely the model's responses align with the ground truth answers. Additionally, we can evaluate the performance using BLEU and ROUGE metrics.
- The fine-tuned model obtained a cosine similarity score of 0.49938, indicating a moderate level of similarity between its responses and the expected answers.

Conclusion

Fine-tuning LLMs is essential for achieving high accuracy in specialized domains. This guide provided a step-by-step approach to fine-tuning for the Heat Equation Solution dataset using the IBM Granite 3.1 8b-Instruct model.

By applying this knowledge, you we fine-tune LLMs for other tasks, such as code generation, legal document analysis, or medical text understanding.