INTEL UNNATI INDUSTRIAL TRAINING PROGRAM - 2024

REPORT

In this report, I shall discuss the process and outcomes of the projects done during the Intel Unnati Industrial Training Program, 2024. The problem statement was - “Introduction to GenAI and Simple LLM Inference on CPU and finetuning of LLM Model to create a Custom Chatbot”

- **BUILDING CHATBOT ON SPR**

In the first session of the training program, we were given an introduction to LLMs and GenAI. Following this, the task given was to run the “build_chatbot_on_spr” notebook. There were different instances/variations of chatbots in the notebook, which I have segregated and run separately. The files can be found in the “simple-llm-inference” folder on GitHub. The outputs of these chatbots variation were observed as follows -

1. The model used for the chatbot was the Intel Neural Chat model.
2. On running the model , the average time taken for the model to load for the first time and provide a reply to the prompt was about 1 minute to 1 minute 30 seconds on average for all notebooks.
3. The model provides satisfactory results to prompts, the results of which can be seen in the uploaded notebooks on GitHub.

- **SINGLE NODE FINE TUNING ON SPR**

The aim of this example was to finetune the Llama model by Meta on different dataset for different downstream purposes such as text-generation, summarization and code generation. The Llama model is a LLM with a huge number of weights and parameters which make fine-tuning very cumbersome and resource-expensive. Thus, I replaced the Llama model with the lighter weight version, Tiny Llama.

Three separate datasets were to be used for fine tuning tasks - Alpaca dataset (text generation), CNN Dailymail (text summarization) and codealpaca dataset for code generation. However, the size of the datasets is moderately large, with the smallest one being over 51 thousand entries big. The training time for the Alpaca dataset (which is also the smallest one) is over 7 hours. Other datasets demand a training time of over 15 hours. Due to the same and the utilization of compute resources, the training had to be prematurely halted. However, over an increment of 500 steps, the training loss varies around a mean value of 1.5.

- **PROJECT**

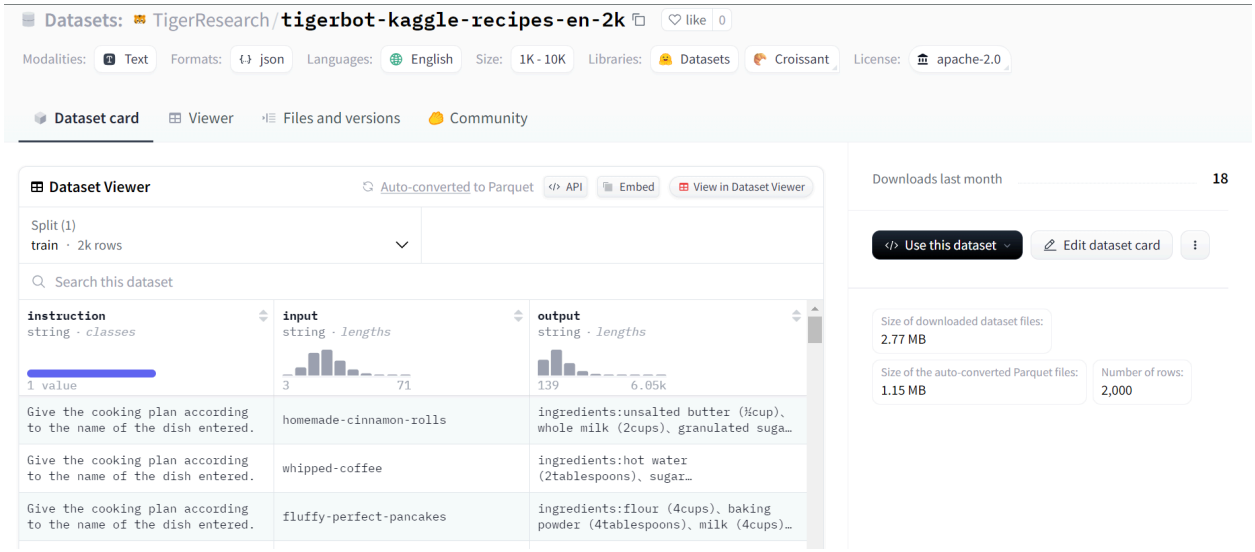
As a result of the above findings, I fine-tuned the Tiny Llama model for a more specific downstream task, which was to act as a recipe generator. The dataset used was found from

HuggingFace Hub, with only around 2000 entries. In the coming paragraphs, I have described the dataset, model, methodology and result.

Dataset

The dataset includes a collection of recipes, likely curated to serve as a resource for various machine learning tasks such as natural language processing (NLP), recommendation systems, and text generation. The dataset includes approximately 2,000 recipes. Each recipe contains various attributes such as the title, ingredients, instructions, and possibly other metadata like cooking time, difficulty level, and cuisine type. Overall, the structure of the dataset is very similar to the Alpaca dataset, thus allowing the use of existing code snippets for fine-tuning.

The dataset is derived from Kaggle's collection of publicly available recipe datasets, possibly cleaned and pre-processed by Tiger Research for better usability in machine learning tasks. A snapshot of the dataset is provided below -



Model

TinyLlama is a 1.1 billion parameter language model created by StatNLP Research Group through substantial pretraining on a trillion-token dataset.. Building on Llama 2's architecture and tokenizer, this model leverages open-source community developments, such as FlashAttention, to greatly improve computational efficiency while maintaining exceptional speed.

TinyLlama's architecture is similar to Llama 2, and it uses a Transformer decoder-only paradigm with particular optimizations.

1. Pre-training Data - Combines SlimPajama's natural language data with Starcoderdata's coding data.

2. Architecture - Uses a Transformer architecture with RoPE for positional embedding, RMSNorm for normalization, SwiGLU as the activation function, and grouped-query attention to improve efficiency.
3. Speed Optimizations - Includes FSDP for optimal multi-GPU utilization, Flash Attention for improved attention mechanisms, and xFormers for a larger memory footprint.
4. Training - For efficient pre-training, an autoregressive language modeling objective is used, together with precise optimizer settings and batch sizes.

The Tiny Llama Chat (v0.1) model was used for the purpose of fine tuning.

Methodology

The basic steps followed were -

1. Model Preparation - Choosing the pre-trained Tiny LLaMA model.
2. Data Preparation - Specifying the training and validation datasets.
3. Training Setup - Defining the training parameters including batch size, number of epochs, logging, and precision settings.
4. Fine-Tuning Configuration - Combining all configurations into a single object.
5. Model Fine-Tuning - Executing the fine-tuning process.

This process was done using the intel-extension-for-transformers Python module. Some of the important training arguments were set as follows -

1. output_dir='./tmp1' // Directory to store output files
2. do_train=True // Do training
3. do_eval=True // Do evaluation of model after training
4. num_train_epochs=3 // Number of passes through the dataset
5. overwrite_output_dir=True // If output folder exists, overwrite it.
6. bf16=True // Use bfloat16 precision

Results

The training loss was about 1.33. The evaluation metrics is as below -

```
***** eval metrics *****
  epoch                   =     2.9939
  eval_loss               =     1.2145
  eval_ppl                =     3.3685
  eval_runtime            = 0:00:01.88
  eval_samples            =         20
  eval_samples_per_second =     10.611
  eval_steps_per_second   =      2.653
```

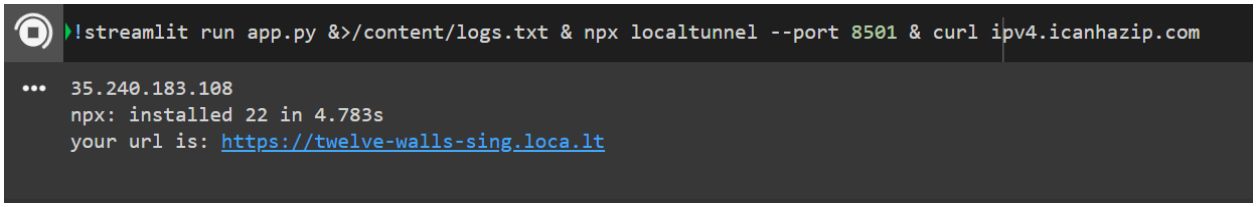
Testing

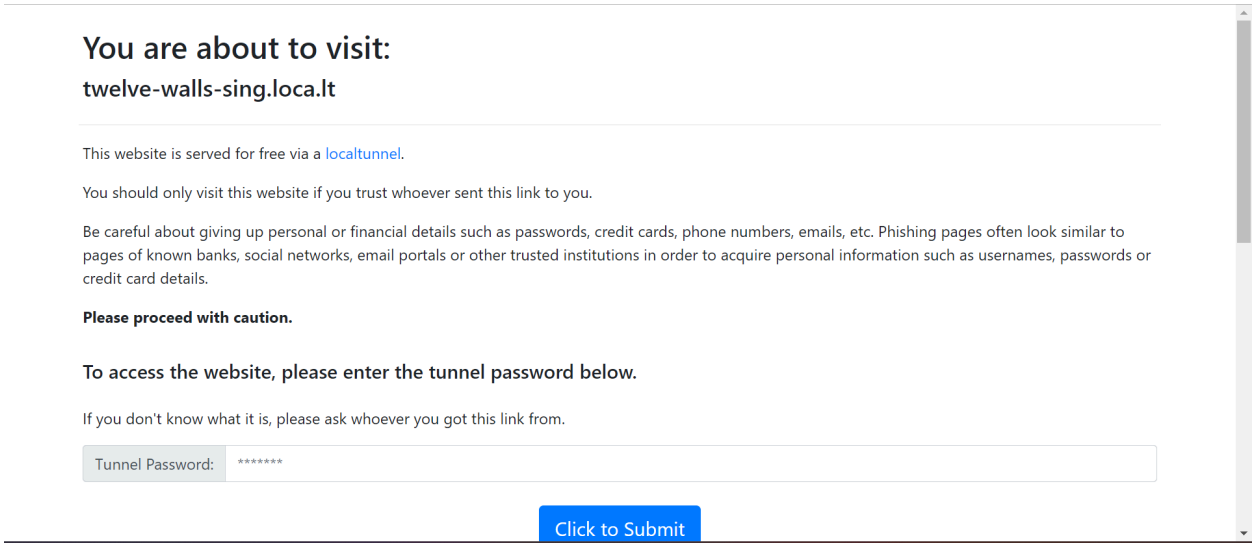
To test the model, the model was pushed onto the HuggingFace Hub after training. The model was then loaded in a separate notebook making use of Transformers and PEFT libraries. The model was allowed to generate a maximum of 500 new tokens in the response, and on running provided satisfactory results. The results of the same can be found in the testing-and-gui notebook on GitHub.

GUI

To allow for greater user interactivity, a simple Streamlit GUI was created where the user can enter the name of a dish they wish to get the recipe for and the response is displayed. To use the interface, the user simply has to run all the cells in the testing-and-gui notebook, on Google Colab, as the Streamlit interface is designed to be deployed from Colab.

The Streamlit app opens through a local tunnel, and to access it, a few steps have to be followed. After running all the cells, click on the link on the last cell. In case of error while running the last cell, the previous "`!npm install localtunnel`" cell has not been run properly. Run the previous cell properly and run the last cell. After clicking on the link, the below page will show up -





Here, for the tunnel password, enter in the 11-digit sequence given in the first output line of the first image above. In this case, it is **35.240.183.108.** Then press the ‘Click to Submit’ button.

It should lead to a UI that will take some time to load. Then, enter in the name of the dish whose recipe you need and press ‘Enter’. Wait for a while and the recipe will be displayed down below.

- **CONCLUSION**

Through this project I have tested out various chatbot versions on SPR through Intel Developer Cloud. I also was able to understand how LLMs work and fine-tune one such LLM to work as a recipe generator.

- **VIDEO LINK**

https://drive.google.com/drive/folders/1mS1PD9uvFVFpWDlimwSaylmMZ0xLGlk2?usp=drive_link