CSE665: Large Language Models

Assignment 3

Fine Tuning Large Language Models

Purpose

The purpose of this code is to compare the accuracy of a pretrained and a fine-tuned language model on a classification task. We use the SNLI (Stanford Natural Language Inference) dataset to evaluate the models, comparing their predictions with true labels on a test dataset. Fine-tuning is performed using the QLoRA (Quantized Low-Rank Adaptation) technique.

Steps

1. Library Import and Model Setup : Necessary libraries (transformers, datasets, and peft) are imported to set up the models, data pipelines, and evaluation metrics.
Pretrained and fine-tuned models are loaded separately.
2. Dataset Preparation : The SNLI dataset is loaded and preprocessed by combining the premise and hypothesis fields for classification. A subset of this dataset is used as the test set for unbiased evaluation.
3. Evaluation Metric : The accuracy metric is loaded from the datasets library to compute the ratio of correct predictions to total predictions, which will be used to evaluate model performance.
4. Inference Pipeline Setup : Separate inference pipelines for text classification are created for the pretrained and fine-tuned models.
These pipelines streamline the prediction process by automatically handling tokenization, model prediction, and decoding.
5. Model Evaluation : The pretrained model is evaluated on the test set, and predictions are stored.
The fine-tuned model is then evaluated on the same test set, and predictions are similarly stored.
6. Accuracy Computation : The accuracy metric is computed for both models by comparing predictions with true labels from the test set.
Finally, accuracy results for both models are printed for comparison.

Components

1. Import Libraries and Load the Test Dataset

- **transformers.pipeline**: This function is a high-level API from the Hugging Face transformers library. It simplifies model inference by combining tokenization, model prediction, and decoding steps into a single function. It supports multiple tasks, like text-classification, summarization, etc.
- datasets.load_metric: This function loads evaluation metrics from the datasets library. Here, we use "accuracy" as our metric, which computes the ratio of correct predictions to total predictions.
  Test Dataset Preparation
- **test_data**: The code assumes test_data is a portion of the dataset kept separate for evaluation purposes. This data should not overlap with training or validation data to ensure unbiased evaluation. The test_data object is expected to contain:
- text field: Each example's combined text (e.g., a sentence or pair of sentences in the SNLI dataset).
- label field: The ground truth class label for each example.

2. Load the Pretrained and Fine-Tuned Models

AutoModelForCausalLM.from_pretrained: This function loads a model for causal language modeling from a specified checkpoint.

- **base_model**: This is the identifier for the pretrained model, which might be something like "microsoft/phi-2". This model has not been fine-tuned on the specific task dataset and will be used as a baseline for comparison.
- **new_model**: This is the identifier for the fine-tuned model (e.g., "phi-2-medquad"), which has undergone additional training on task-specific data (SNLI dataset).
- **device_map={"": 0}**: This parameter specifies which device the model will use. device_map={"": 0} sends the model to GPU device 0 if available; if running on CPU, you can set device_map={"": "cpu"}.

3. Initialize the Evaluation Metric
   - **load_metric("accuracy")**: This loads the accuracy metric, which calculates the fraction of correct predictions made by the model. For each prediction that matches the actual label, accuracy increases, providing a straightforward performance metric for classification tasks.

4. Set Up Inference Pipelines

pipeline("text-classification"): The pipeline function is used here to create pipelines for text classification. It takes in the model and tokenizer, and automatically applies them during inference.

- **model**: The specific model instance for each pipeline (either pretrained_model or fine_tuned_model).
- **tokenizer**: This is the tokenizer corresponding to the model. It handles text pre-processing, including tokenizing words and creating input IDs that the model can understand.
- **device=0**: This specifies that the pipelines should run on GPU device 0 if available. For CPU usage, you would set device=-1.

Each pipeline takes the input text, tokenizes it, runs it through the model, and then decodes the prediction. This setup allows easy access to both pretrained and fine-tuned model predictions.

5. Evaluate the Pretrained Model
   - **Loop Over Test Data**: This loop iterates over each example in the test dataset, obtaining a prediction from the pretrained model pipeline for each input.
   - **result = pretrained_pipeline(example["text"])**: Each input text is passed through the pretrained pipeline, which outputs a list of predictions with probabilities.
   - **Extracting label**: We assume the labels are in the format "LABEL_0", "LABEL_1", etc. The predicted_label.split("_")[1] extracts the numerical part of the label, converting it to an integer, and appends it to pretrained_predictions.

This process is repeated for each example in the test set, creating a list of predicted labels for the pretrained model.

6. Evaluate the Fine-Tuned Model
   - **Fine-Tuned Model Evaluation**: This loop mirrors the pretrained model evaluation loop, but it uses the fine-tuned model pipeline to get predictions. The outputs are appended to fine_tuned_predictions to compare accuracy with the pretrained model.

7. Compute and Compare Accuracies
   - **metric.compute()**: This function computes the accuracy for both models by comparing their predictions with the actual labels in test_data["label"].
   - **predictions**: A list of predicted labels generated by each model.
   - **references**: A list of true labels from the test dataset.
   - **Output**: The print statements display the computed accuracy for both the pretrained and fine-tuned models, allowing you to see if fine-tuning improved the model’s performance.

Summary

This code setup evaluates both pretrained and fine-tuned models on a common test dataset, calculating accuracy as the primary metric. By comparing accuracies, we can gauge the impact of fine-tuning on the model's performance.

This approach provides a generalizable framework to compare model versions and can be adapted to other tasks, datasets, and evaluation metrics by adjusting pipeline parameters and metric configurations.

Time Taken to Fine-Tune the Model Using QLoRA:

- The total training time for fine-tuning is listed as `train_runtime: 1852.7625` seconds (or about 30 minutes), found under `TrainOutput`.

Total Parameters in the Model and the Number of Parameters Fine-Tuned:

- The total parameters in the model are **1,521,392,640**. The code shows that `print_trainable_parameters(model)` returned `trainable params: 31457280 || all params: 1552849920 || trainable%: 2.025777223854318`, which may indicate that no additional parameters were fine-tuned in this configuration. This might be due to the specific setup or an incomplete configuration for LoRA in the model definition.

Resources Used (Hardware, Memory) During Fine-Tuning:

- Training configuration includes using `gradient_checkpointing`, batch size of `4`, gradient accumulation steps of `4`, and a learning rate of `1e-4`. The device is set to `cuda` if available, suggesting GPU use for fine-tuning.
- Quantization is enabled with 4-bit precision (`bnb_4bit_compute_dtype=torch.float16`), which reduces memory usage. The setup likely ran on a high-memory GPU, though the exact GPU type and memory specifics aren't provided.

Failure Cases of the Pretrained Model Corrected by the Fine-Tuned Model:

- No specific failure cases are mentioned in the notebook. For insights, you would typically run inference on both models and analyze instances where the fine-tuned model corrected errors made by the pretrained model. Potential explanations for successful corrections could include improvements in domain-specific knowledge or contextual understanding due to fine-tuning on task-specific data.

Accuracy Comparison Between Pretrained and Fine-Tuned Models

- The accuracy for pre-trained model came around 30% and for fine-tuned model, the accuracy is calculated around 75%.
- The large accuracy gain (from 30% to 75%) highlights the importance of fine-tuning pretrained language models on task-specific data. While pretrained models provide a solid linguistic foundation, they often require additional task-specific data to perform well on specialized tasks like SNLI.

- Fine-tuning enables the model to bridge the gap between general language understanding and specific inferential skills, improving its applicability to nuanced NLP tasks.
- This result confirms that pretrained models benefit greatly from adaptation when applied to tasks requiring specialized knowledge or reasoning abilities.