IIIT-Bangalore

Visual Recognition

AIM - 825

Professors:
Prof. Viswanath G
Prof. Sushree Behera

Submitted by:
Madhav Girdhar (IMT2022009)
Siddhesh Deshpande (IMT2022080)
Krish Patel (IMT2022097)

Dataset

Dataset Curation

- Amazon Berkely Objects (ABO) Dataset was used for the project.
- There were 147,702 product listings with multilingual metadata and 398,212 unique catalog images in the dataset.
- The small variant of the multilingual dataset with size 3GB was used.
- The product listings had different json files which contained the description of each product. There was also a metadata file in the images catalogue which contained the path of the images in the folder.
- We looped through each json file and perform a join operation on the json file and the images.csv file that had the metadata of the images in the images catalogue.The join was performed with the main_image_id from json object and image_id from the images.csv file.
- After this we got a csv file that has the images, their paths and description for each image.In terms of description of the product we decided to keep the bullet_points and item_name fields from the json files into our final metadata file.These fields will further be used for dataset creation.
- Now at the end of this process we have a file named merged_metadata_final.csv which contains image_id, height, width, path, bullet_point, item_name.
- The process explained above has been implemented in the metadata_creation.ipynb file.

Dataset Creation

- We used the Gemini 2.0 API for creating our VQA Dataset.
- **Prompt:** Analyze the given image and the description together. Generate three questions per image that can be answered using both. Then answer that question in just one word. Ensure diversity in question types and difficulty levels. Output format like given below:

  ```
  question1$one-word-answer1#question2$one-word-answer2#question3$one-word-answer3.
  ```

  Strict rules: no extra text, no greetings, no explanations, no angle brackets.
- This prompt was used to generate 3 question and answers pairs for each image. We decided to keep only 3 QA pairs per image because we wanted to keep quality of questions and answers good .

- We created around 102000 data points using this process. We randomly sampled images to create these datapoints , random sampling of the images from the final metadata file was essential because the data set was largely imbalanced towards mobile phone cases.
- After this we preprocessed the dataset created by dropping the data points where question or answer were absent then we dropped if any duplicate data points were present and split the dataset into train, test, val dataset. The train dataset has 67000 ,test dataset has 25000 and validation dataset has around 10000 data points.
- This entire process has been implemented in the preprocessing.ipynb and data_generation_1.ipynb.

Model Selection and Baseline Observations

In our experimentation, we explored multiple variants of the BLIP model family for Visual Question Answering (VQA) tasks, with and without LoRA-based fine-tuning. The models evaluated are:

1. BLIP-2 OPT (blip2-opt-2.7b)
2. BLIP-2 (blip2-flan-t5-xl)
3. BLIP VQA Base

The way how these model behavior change after fine tuning as well as the baseline evaluation results are written below.

BLIP-2 OPT (blip2-opt-2.7b):

1. Initially, the model responded with long sentence-form answers.
2. After applying LoRA fine-tuning, the behavior changed drastically: the model started producing short, repetitive phrases, leading to degraded accuracy.
3. This instability and inconsistency in response generation made it difficult to fine-tune the model effectively on our dataset.

BLIP-2 (blip2-flan-t5-xl):

1. This model generates mostly single-word answers by default(still depends on max_tokens specified we used 10), though it occasionally produces long descriptive sentences.
2. We hypothesized that fine-tuning would help regulate output length based on the question type.

3. However, due to the large number of parameters, fine-tuning on our dataset was computationally expensive and did not result in significant performance improvements.
4. The output distribution and patterns remained nearly the same as the baseline model.

BLIP VQA Base:

1. This model appears to be specifically optimized for VQA tasks, consistently producing concise, single-word answers.
2. It yielded higher BERT Score and Exact Match (EM) metrics compared to the other models.
3. Its smaller model size made it easier to fine-tune effectively.
4. Post-fine-tuning, we observed significant performance improvements, validating its suitability for VQA with limited data.

Evaluation Techniques:

- Exact Match (EM): Measured the number of predictions that exactly matched the ground truth.
- BERTScore: Evaluated semantic similarity between predicted and actual answers using pre-trained BERT embeddings.

In addition to BERT Score and Exact Match, we also explored:

- Average Cosine Similarity: Compared sentence embeddings of predictions vs. references using average and best cosine similarity.
- Best Cosine Match: In addition to the average, we identified high-confidence predictions by measuring the best cosine match—defined as prediction-reference pairs with a cosine similarity score 0.75.

These semantic evaluation metrics provided complementary insights to Exact Match and BERTScore, offering a deeper understanding of the model's performance beyond surface-level accuracy. These are the baseline scores for each of the models.

Results:

1. BLIP2-opt-2.7b:
   - Mean BERT Score F1: 0.8049
   - Mean BERT Score Precision: 0.7934
   - Mean BERT Score Recall: 0.8184

- Exact Match Accuracy: 0.02800
- Average Cosine Similarity Score: 0.4177
- Best Cosine Match: 0.1732

2. BLIP2-flan-t5-xl:
    - Mean BERT Score F1: 0.8925
    - Mean BERT Score Precision: 0.8851
    - Mean BERT Score Recall: 0.9022
    - Exact Match Accuracy: 0.19316
    - Average Cosine Similarity Score: 0.5201
    - Best Cosine Match: 0.2462

3. BLIP-VQA-BASE:
    - Mean BERT Score F1: 0.9278
    - Mean BERT Score Precision: 0.9353
    - Mean BERT Score Recall: 0.9228
    - Exact Match Accuracy: 0.23180
    - Average Cosine Similarity Score: 0.5433
    - Best Cosine Match: 0.2708

Optimization Attempts:

We explored techniques like Flash Attention to enhance training efficiency, but found it unsupported in BLIP-2 models. We also enabled use_cache, though it showed no significant impact on performance or speed. These efforts reflect our attempt to optimize the model performance process.

Conclusion:

Among all the variants tested, BLIP VQA Base emerged as the most effective model for our use case. It combines:

- Efficient fine-tuning due to its compact size,
- Stable and task-relevant answer formatting,
- Significant performance gains after training.

In contrast, the heavier models (BLIP-2 and BLIP-2 OPT) were not only difficult to fine-tune with our dataset but also exhibited limited or unstable improvements in answer quality.
Note: Inference_with_performance_metrics is the file that was used for inference on our test dataset and also for baseline of these models.

Fine Tuning With Lora

As per the discussion in above section we decided to move ahead with the blip-vqa-base model.This Model has 385 Million parameters.
Fine tuning the model means that we take a pre trained model and use a task specific dataset and train the model further on that dataset so that it performs good in this downstream task.
But often these pre trained models are large in size and we dont have that large of a dataset to be able to train such large models,also it will require large compute and memory resources.So here a technique called Lora(Low Rank Adaptation) comes to our rescue. So what Lora does is, instead of updating all the model's weights during fine-tuning, LoRA freezes the original weights and injects a small set of trainable parameters (low-rank matrices) into certain parts of the model — typically the attention or feedforward layers. These small matrices:

- Are much cheaper to train.
- Approximate the same effect as full fine-tuning.
- Can be added or removed easily, like plugins.

The Entire Fine Tuning process description for the project using lora is as follows:

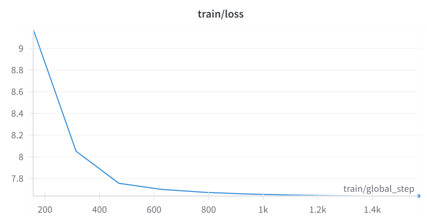
- Load the model and the processor using the BlipProcessor, BlipForQuestionAnswering from the transformers library.
- The class VQADataset is used to create the dataset into the torch.utils.data.Dataset format the processor is also passed to this class hence each data point is now a dictionary of 4 tuples that is input_ids, attention_mask, pixel_values and labels . here the first three are obtained through question and image and label is obtained through the answer or the response for that question.
- While creating dataset we pad the input_ids and attention_mask to 128 length tensors and labels to 32 length tensor.
- We define a data collator called collate_fn, this function stacks input_ids, attention_mask, pixel_values and labels into batch tensors, so that the model receives a single dictionary of batched inputs.
- We pad the labels with -100 as the padding value so that the model will ignore them while calculating the cross entropy loss.
- The Following are the LoraConfig parameters that are used while finetuning:
  - r(rank) = 16
  - lora_alpha=32
  - target_modules = ["query","value"]

  - lora_dropout=0.1
  - bias="none"

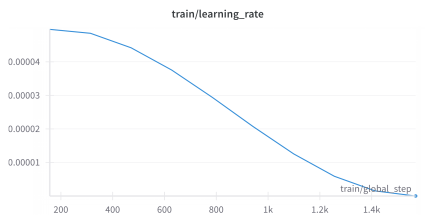
- We decided to go ahead with rank=16 because that is the good starting point and we continued with it as the results were observed were good and we keep lora_alpha=32 because it is usually recommended to keep a 1:2 ratio between rank and alpha.
- we then apply the loraconfigs to the model using the PEFT library get_peft_model.
- For the training purpose we use the hugging face trainer api. The details regarding its parameters and their values will be explained as we move ahead.

As mentioned in the project document we have created iterative baselines or followed an iterative method for fine tuning what i mean by this is that we started by finetuning with small portion of the train dataset and gradually move ahead to finetuning on the complete train dataset.

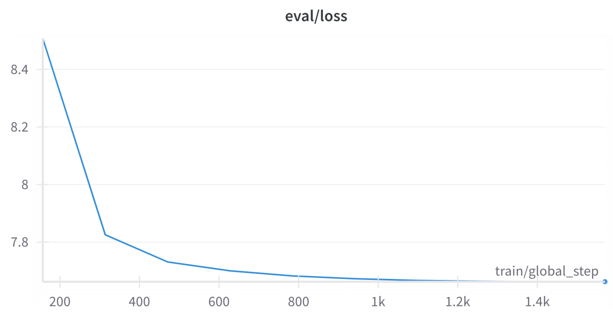
Fine Tuning with 10k datapoints



Training loss



Learning rate



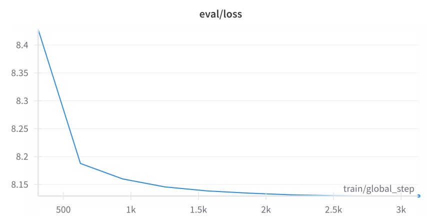
Validation loss

We downloaded the model weights and did inference on our test dataset and obtained following scores:

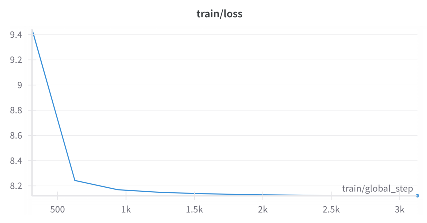
- Mean BERT Score F1: 0.9338
- Mean BERT Score Precision: 0.9424
- Mean BERT Score Recall: 0.9274
- Exact Match Accuracy: 0.41968
- Average Cosine Similarity Score: 0.6730
- Best Cosine Match: 0.45628

Initial Learning Rate - 5e-5

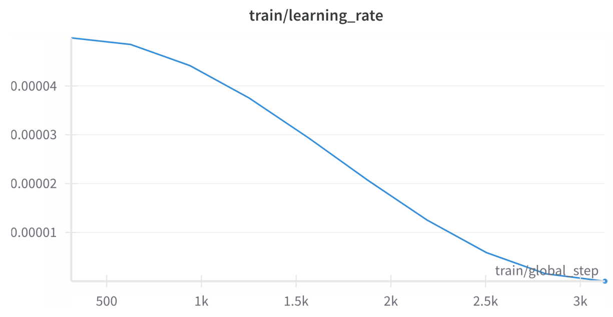
Fine Tuning with 20k datapoints



validation loss



training loss



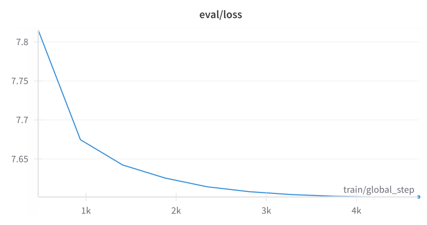
learning rate

We downloaded the model weights and did inference on our test dataset and obtained following scores:

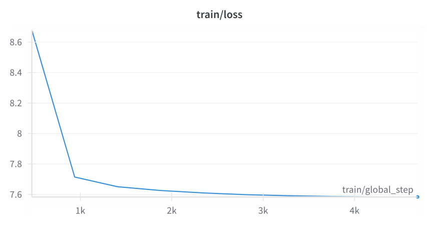
- Mean BERT Score F1: 0.9346
- Mean BERT Score Precision: 0.9429
- Mean BERT Score Recall: 0.9285
- Exact Match Accuracy: 0.44464
- Average Cosine Similarity Score: 0.6868
- Best Cosine Match:0.48244

Initial Learning Rate - 5e-5

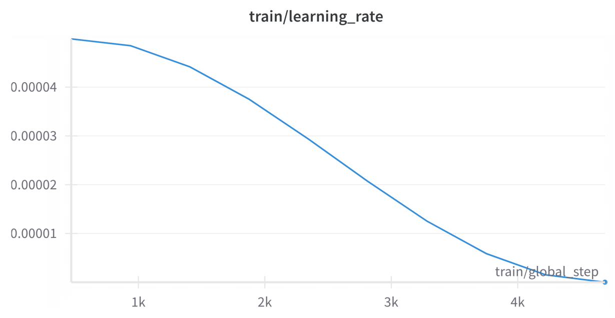
Fine Tuning with 30k datapoints



validation loss



training loss



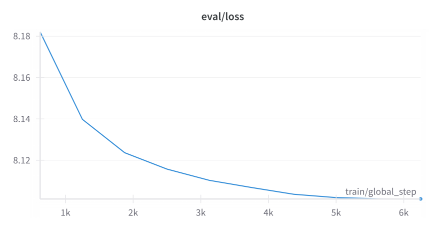
learning rate

We downloaded the model weights and did inference on our test dataset and obtained following scores:

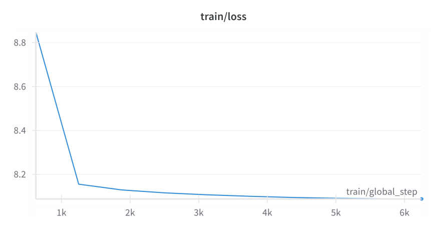
- Mean BERT Score F1: 0.9361
- Mean BERT Score Precision: 0.9438
- Mean BERT Score Recall: 0.9304
- Exact Match Accuracy: 0.47064
- Average Cosine Similarity Score: 0.6996
- Best Cosine Match:0.50728

Initial Learning Rate - 5e-5.

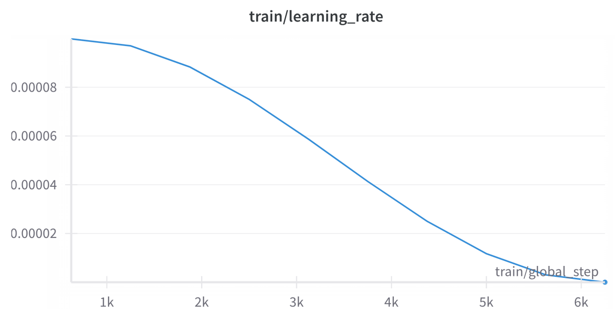
Fine Tuning with 40k datapoints



validation loss



training loss



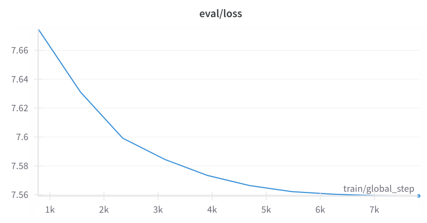
learning rate

We downloaded the model weights and did inference on our test dataset and obtained following scores:

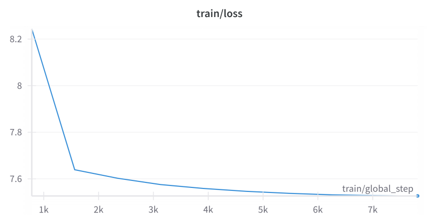
- Mean BERT Score F1: 0.9382
- Mean BERT Score Precision: 0.9455
- Mean BERT Score Recall: 0.9332
- Exact Match Accuracy: 0.51088
- Average Cosine Similarity Score: 0.7230
- Best Cosine Match:0.54948

Initial Learning rate - 1e-4.We have changed it into 1e-4 instead of 5e-5. Here we decided to go ahead with a higher learning rate because in previous stages we saw that by the last epoch the learning rate used to become negligible or very close to 0 it goes to a order of 1e-11 or 1e-12 hence it might happen that the weights are not getting updated significantly in the later epochs and here increasing the lr gave us good results while fine tuning on 40k datpoints or more.

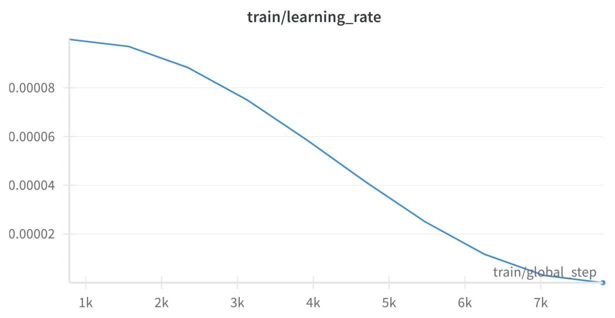
Fine Tuning with 50k datapoints



validation loss



training loss



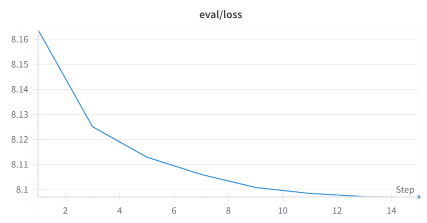
learning rate

We downloaded the model weights and did inference on our test dataset and obtained following scores:

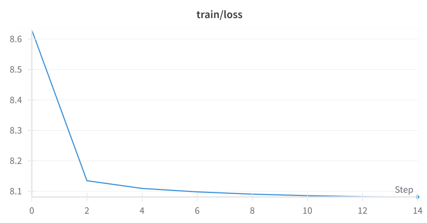
- Mean BERTScore F1: 0.9386
- Mean BERTScore Precision: 0.9453
- Mean BERTScore Recall: 0.9339
- Exact Match Accuracy: 0.52144
- Average Cosine Similarity Score: 0.7296
- Best Cosine Match: 0.56028

Initial Learning rate - 1e-4.

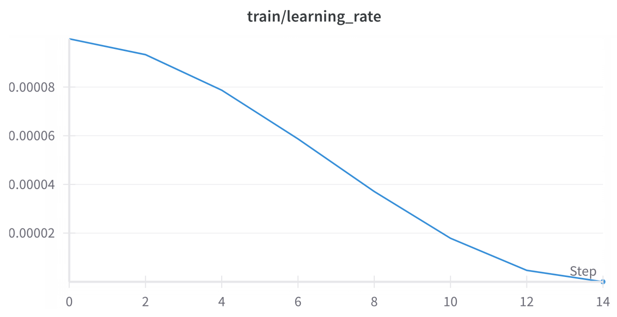
Fine Tuning With 67k datapoints



validation loss



training loss



learning rate

We downloaded the model weights and did inference on our test dataset and obtained following scores:

- Mean BERTScore F1: 0.9386
- Mean BERTScore Precision: 0.9456
- Mean BERTScore Recall: 0.9337
- Exact Match Accuracy: 0.51816
- Average Cosine Similarity Score: 0.7275
- Best Cosine Match: 0.5578

Initial Learning rate - 1e-4.
Now we will move on to explain the parameters of the Trainer API that we have used for the training/fine-tuning of the model.The Following arguements were given to the TrainingArguements function.

- eval_strategy - The evaluation strategy to adopt during training.Here we have used "epochs" which means evaluation is done at the end of each epoch.
- gradient_accumulation_steps - Number of updates steps to accumulate the gradients for, before performing a backward/update pass.We use the default value that is 1.
- warmup_ratio — Ratio of total training steps used for a linear warmup from 0 to learning_rate.We use a warmup ratio of 0.1 .
- fp16 - Whether to use 16-bit (mixed) precision training instead of 32-bit training.We use the value True for this to use 16-bit precision training to reduce memory consumption.
- label_names — The list of keys in dictionary of inputs that correspond to the labels.We use ["labels"] as the list of keys for this parameters because "labels" corresponds to the answers.
- per_device_train_batch_size — The batch size per device accelerator core/CPU for training.We use the batch size of 32 for training.
- per_device_eval_batch_size — The batch size per device accelerator core/CPU for evaluation.We use the batch size of 32 for evaluation.
- lr_scheduler_type - We use the cosine scheduler instead of default linear scheduler because the results seemed to be better for this scheduler.
- learning_rate - The learning rate that is used for the AdamW optimizer. we have specified what learning rate we have used in previous finetuning stages explaination along with reason. We used two different rates one is 5e-5 and other is 1e-4.

- load_best_model_at_end — Whether or not to load the best model found during training at the end of training. When this option is enabled, the best checkpoint will always be saved. We set this to True.
- save_strategy — The checkpoint save strategy to adopt during training. We used "epochs".
- metric_for_best_model — Used in conjunction with load_best_model_at_end to specify the metric to use to compare two different models.We used eval_loss.
- greater_is_better — Used in conjunction with load_best_model_at_end and metric_for_best_model to specify if better models should have a greater metric or not. We used False because metric_for_best_model is set to a value that ends in "loss".
- num_train_epochs - We used 8-10 epochs depeding upon the amount of data we were using while finetuning.During our final finetuning on entire training data we used 8 epochs.

Using this Trainer API allowed us to fit large batch sizes like 32 allowing us to run more epochs because it makes use of all available gpu's automatically.

Observation

We were able to observe that training on 50k gave better results as compared to training on 67k dataset this means that the 50k dataset is good enough to train the no of parameters injected with rank=16 in lora and hence we decided to keep that as our final weights for submission also we were able to run less epochs on 67k dataset i.e 8 epochs as compared to 10 epochs on the 50k dataset due to the kaggle runtime limits.Hence that could also be a potential reason for the results of 50k being better than the 67k dataset.

Additional Contribution

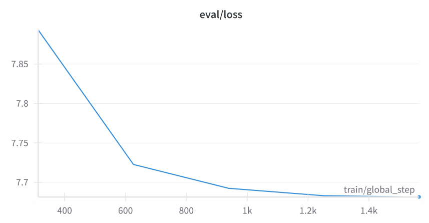
- As mentioned earlier we have tried out using KV Cache for faster inference but there was no significant improvement in the inference time it took 44 mins without the use of cache and 42 mins with the use of cache but this was also occassional improvement almost all runs took nearabout the same time close to 44 mins.
- We thought of using flash attention but realized that it is not supported by this model.
- We have also used QLora for finetuning as it reduces the model size significantly leading to less memory usage and still not compromising on the performance.

QLora

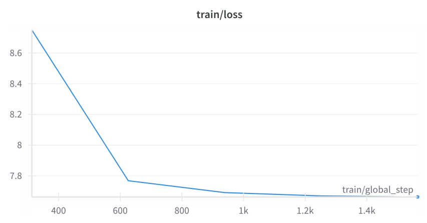
QLoRA, or Quantized Low-Rank Adaptation, is an advanced fine-tuning technique designed to optimize large language models (LLMs) for efficiency and performance. It combines two key strategies:

- **Quantization:** This process reduces the precision of the model's weights, typically to 4-bit representations, significantly decreasing memory usage without substantially impacting performance.
- **Low-Rank Adaptation (LoRA):** Instead of updating all model parameters during fine-tuning, LoRA introduces small, trainable adapter modules into the model. These adapters capture task-specific information, allowing for efficient fine-tuning with fewer parameters.

Here the qlora has been implemented in the qlora.ipynb file the implementation is very similar as compared to the usual implementation of lora except the fact that we need to load the model in 4bit using bitsandbytes the rest of the training code remains the same.For experimentation we finetuned the model only on a small train_dataset of 10k data points only and how the validation loss and training loss varies can be seen in the following graphs. The parameters of the



validation loss



training loss

trainer and loraConfigs are as follows:

- Learning Rate - 5e-5
- lr_scheduler - cosine
- epochs = 5
- lora_dropout=0.05

The rest of the parameters remain more or less the same wrt to the ones used for lora finetuning.The key obervations on using qlora for finetuning are as follows:

- We observed that the performance did not drop when compared to the 10k training done with lora.

- The memory usage was much lower even with the batch size of 32 the gpu consumption did not go above 5Gb it always stayed below that but this was not the case in lora the gpu usage was much higher there it went upto 16-18 Gb's.

Thus we can conclude that qlora can help achieve similar results without a very significant drop in performance and at the same time reduce memory usage drastically thus making fine tuning of large model's computationally cheaper as compared to the standard lora method.

Result on the 5 epoch run for qlora

- Mean BERTScore F1: 0.9328
- Mean BERTScore Precision: 0.9412
- Mean BERTScore Recall: 0.9266
- Exact Match Accuracy: 0.40544
- Average Cosine Similarity Score: 0.6654
- Best Cosine Match: 0.4422

We can compare the result of lora finetuning on this 10k dataset and that of this qlora run on same dataset and observe that the results are almost similar we can get those results easily or the similar ones if we execute 10 epochs here we only did 5 but the results of this clearly show that qlora can help acheive similar performance with less memory usage.