Multimodal Visual Question Answering with Amazon-Berkeley Objects Dataset

Bhavya Kapadia

May 2025

Contents

1 Introduction

This project focuses on multimodal visual question answering using the Amazon-Berkeley Objects Dataset. The primary objectives include:

- Processing and preparing the data
- Curating a suitable dataset
- Establishing a baseline model
- Fine-tuning the model using the LoRA approach
- Evaluating the performance of the fine-tuned model

This report presents a comprehensive overview of the methodologies employed, the key observations made, and the resulting analyses.

2 Team

- Siddeshwar Kagatikar (IMT2022026)
- Bhavya Kapadia (IMT2022095)
- Dikshant Mahawar (IMT2022549)

Project Directory Structure

The overall structure of the project directory is organized as follows:

```
DataCuration
   Input-Data
   Output-Data
      data.json
      data_curation.py
      train_test.py
images
inference-setup
   data
   sample-submission
      IMT20220...
   requirements.txt
   run_inference_for_all_public.py
README.md
blip-base-model.ipynb
lora-final-model.ipynb
```

3 Dataset Curation

This section outlines the step-by-step pipeline used to curate a high-quality Visual Question Answering (VQA) dataset from the Amazon Berkeley Objects (ABO) dataset. Raw product listings and images were transformed into structured Q&A pairs suitable for training vision-language models such as BLIP-2 and CLIP.

The curation process was implemented across two Python scripts: *data_curation.py* and *train_test.py*. We used the Gemini API to automatically generate five diverse QA pairs per image, grounded in both the image content and associated metadata. This ensured rich and varied annotations for downstream training and evaluation tasks.

3.1 Extract and Inspect Image Data

The image data preparation begins with the `abo-images-small.tar` archive provided in the ABO dataset. This archive is extracted into a structured directory hierarchy located at `extracted-folder/images/small/`, where images are organized by prefix-based subfolders (e.g., `00/000a12ff.jpg`).

The corresponding image metadata is provided in the `images.csv.gz` file, which includes key fields such as `image_id`, `width`, `height`, and relative image paths. As a validation step, all listed image paths are programmatically verified to ensure they point to existing image files in the filesystem.

3.2 Preprocessing Product Listings and Descriptions

The product-level metadata was originally stored in compressed JSON files (e.g., `listings_*.json.gz`), with each line representing an individual product listing in JSON format. These files were first converted from line-delimited JSON into a valid JSON array and subsequently loaded into Pandas DataFrames for further processing.

For each product listing:

- The primary image was identified via the `main_image_id` field and matched with its path from `images.csv.gz`.
- English-language bullet points (tagged with `language: en_IN`) were extracted and concatenated into a coherent product description.

Each dataset contained the following fields: `image_id`, `image_path`, `product_description`, and other relevant attributes.

3.3 Methodology for QA Pair Generation

The VQA dataset was generated using Google's Gemini 2.0 Flash model through an automated process consisting of the following key steps:

- **Data Loading:** Metadata and image paths were loaded from CSV and JSON files. Each product entry was linked to a corresponding image via its `image_id`.
- **Prompt Construction:** For each product, a prompt was constructed to instruct Gemini to generate five VQA questions, each with a single-word answer. The prompt included both the product description and a directive to ensure coverage of diverse visual aspects.

  ```
  "Generate 5 Visual Question Answering (VQA) questions based on an image.  Each
  question should have a one-word answer.  Cover diverse aspects from the image
  and metadata.  Don't just include binary (yes/no) questions."
  ```

- **API Interaction:** The image (as binary), product metadata (as JSON), and constructed prompt were sent to the Gemini API using the `generate_content` function. The Gemini model returned structured Q&A pairs in response.
- **Output Parsing:** Responses were parsed using regular expressions to extract well-formed question–answer pairs. Only entries with valid, meaningful answers were retained.
- **Result Storage:** For each processed product, the Q&A pairs were saved in a structured JSON format:

  ```
  {
    "item_id": "...",
    "image_id": "...",
    "questions": [
      {"question": "...?", "answer": "..."},
      ...
    ]
  }
  ```

All Q&A data was continuously saved into `output_1.json` to safeguard progress during execution. Already processed entries were tracked to avoid duplication.

3.4 Rate Limiting and Fault Tolerance:

A 6-second delay was added between consecutive API calls to comply with Gemini's rate limits. The script was fault-tolerant—skipping already processed entries and allowing safe resumption after any error.

3.5 Final Dataset Compilation

The file `data.json`, which stored intermediate outputs in the format:

```
full/image/path.jpg#What color is the object?#Red
```

was parsed and converted into a final CSV file. Image paths were trimmed to their last two directory levels (e.g., `55/551934fc.jpg`) for compactness.

The data is saved into 2 new files train_dataset.csv and test_dataset.csv The final structured output was saved as `final_vqa_dataset.csv` with the following columns:

- `image_path`
- `question`
- `answer`

3.6 QA Category Pie Chart

Figure 6 presents a pie chart illustrating the distribution of question–answer (QA) categories within our dataset. Each segment represents a different type of question generated based on the image content and associated metadata, providing insight into the diversity and balance of the dataset.

3.7 Question Type Distribution Bar Chart

Figure 2 shows a bar chart representing the frequency of different types of questions generated in the dataset. Each bar corresponds to a distinct question type, helping to visualize the relative occurrence and balance among the various categories. This analysis provides insight into the diversity of questions and highlights potential biases in the automatic generation process.

4 Baseline Model

4.1 Model Selection

- **Model Overview:** The `Salesforce/blip-vqa-base` model is a vision-language model designed for Visual Question Answering (VQA) tasks. Based on the BLIP (Bootstrapped Language-Image Pretraining) framework, it fuses visual and textual information to generate answers from images and related questions. The `BlipProcessor` handles input preprocessing, while the `BlipForQuestionAnswering` class performs inference. It is optimized for high speed and low latency, making it ideal for real-time applications such as image analysis, AI assistants, and accessibility solutions.
- **BLIP-VQA Components:** The model comprises two main components—`BlipProcessor`, which preprocesses the image and question inputs, and `BlipForQuestionAnswering`, which performs the multimodal reasoning and generates answers based on the fused visual-textual representation.
- **Comparison with Other Models:** Compared to earlier models like `VilBERT`, which are heavier and slower, `blip-vqa-base` offers greater efficiency and lighter deployment. While `BLIP-2` provides enhanced zero-shot capabilities through a more complex and larger architecture, `blip-vqa-base` achieves a practical balance between performance, size, and speed, making it a strong candidate for real-world VQA systems.

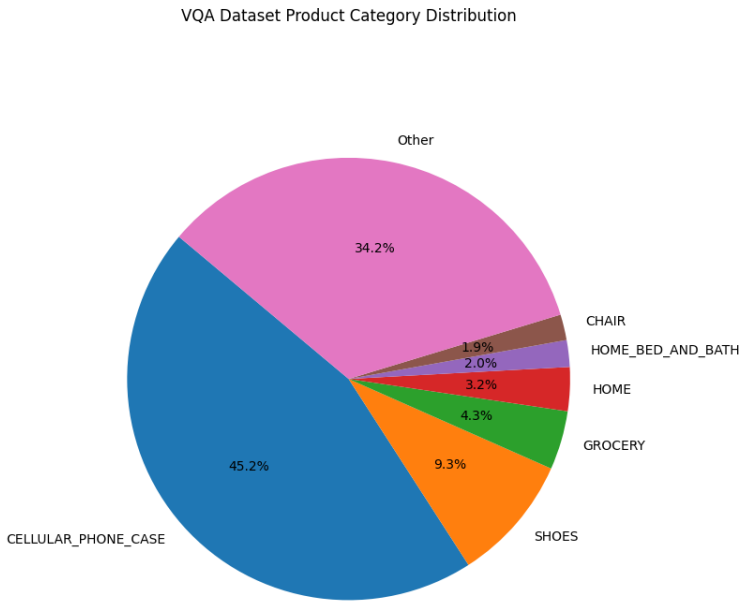

Figure 1: Distribution of QA categories in the dataset.

4.2 Inference

The `blip-base-model.ipynb` notebook was used to obtain the baseline accuracy on 20% of the `data.json` file, which was later converted to `data.csv`. The notebook was executed on Kaggle using a **P100 GPU** accelerator. Subsequently, `data.csv` was split into `train.csv` and `test.csv` using the `train_test.py` script located in the `DataCuration` folder on GitHub. Batch-wise testing was implemented using a custom `VQADataset` class to preprocess the test data and a PyTorch `DataLoader` with a batch size of 8 for efficient loading and batching. A custom `collate_fn` was used to handle variable-length inputs during batching. The steps to run the notebook are as follows:

1. **Setup To Be Done On Kaggle**
   - Upload `test.csv` as an Input dataset and name it `finale-dataset`.
   - Enable Internet access in the notebook settings.
   - Select `P100 GPU` as the hardware accelerator.
   - The images from the folder `abo-images-small/images_small` should be uploaded to Kaggle as a dataset named `vr-dataset`.
2. **Running the Notebook**
   - Click `Run All` to execute all cells in the notebook.
   - Evaluation metrics are displayed and predictions are saved as `predictions.csv`.

5 Fine-Tuning using LoRA

To improve the performance of the baseline BLIP-VQA-Base model on our specific Visual Question Answering (VQA) task, we applied Low-Rank Adaptation (LoRA) for fine-tuning. LoRA enables efficient adaptation

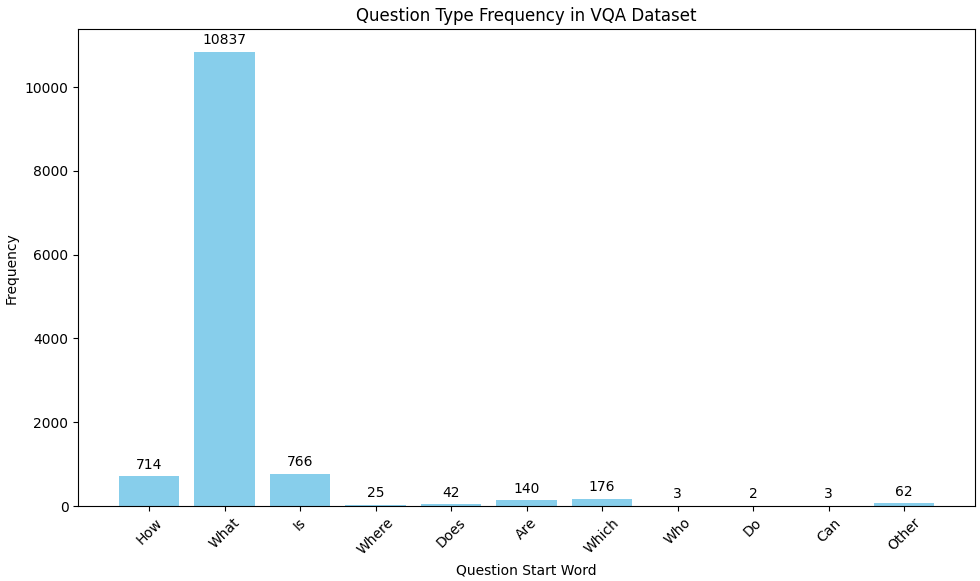

Figure 2: Distribution of question types across the dataset.

of large pre-trained models by introducing and training a small number of additional parameters, significantly reducing computational and memory overhead. The fine-tuning was performed in the lora-final-model.ipynb notebook using a P100 GPU on Kaggle.

Key Aspects of the Fine-Tuning Process

Model and Processor Setup

- **Base Model:** Used the `Salesforce/blip-vqa-base` model as the foundation for fine-tuning on our VQA task.
- **Processor Setup:** Employed the associated `BlipProcessor` to handle both image and text preprocessing for multimodal input preparation.
- **Hardware Acceleration:** Deployed the model on a CUDA-enabled **P100 GPU** (via Kaggle) to accelerate training and optimize performance.
- **Forward Pass Modification:** Patched the model's `forward()` method to remove `inputs_embeds` and `decoder_inputs_embeds`, ensuring compatibility with LoRA-based fine-tuning.

Dataset and Input Preparation

The final VQA dataset (`final_df`) was prepared using a custom PyTorch Dataset class named BlipVQA-Dataset, which handled both image and text preprocessing. Key steps included:

- **Image Processing:** Each image was loaded and converted from BGR to RGB format to maintain consistency with the model's expected input.
- **Text-Image Pairing:** Each sample combined an image with a corresponding question and ground-truth answer from the dataset.
- **Multimodal Preprocessing:** The `BlipProcessor` was used to tokenize and format the inputs, preparing both the image and the question for the model.

- **Label Encoding:** Answers were tokenized separately to serve as the target output (labels), with appropriate padding and truncation applied.
- **Batch Preparation:** A custom `collate_fn` function was implemented to batch samples and ensure label sequences were padded correctly. Padding tokens were masked with -100 to exclude them from the loss computation during training.

LoRA Configuration via PEFT

To make fine-tuning more efficient and lightweight, we integrated Low-Rank Adaptation (LoRA) using the `peft` library. Configuration included:

- **Rank (r):** 16
- **Alpha:** 32
- **Dropout:** 0.1
- **Target Modules:** Limited to attention-related layers (`query`, `value`)
- **Bias:** Set to `none`
- **Task Type:** Sequence-to-sequence language modeling (`SEQ_2_SEQ_LM`)

Only the LoRA adapter parameters were made trainable, drastically reducing memory usage and training time.

Training Strategy and Hyperparameters

Training was performed using a combination of techniques to maximize GPU efficiency:

- **Epochs:** 2
- **Learning Rate:** 5×10^{-5}
- **Scheduler:** Cosine decay with 100 warm-up steps
- **Batch Size:** 8
- **Optimizer:** AdamW
- **Precision:** Enabled FP16 using `torch.cuda.amp` for mixed-precision training
- **Memory Optimization:** Used gradient checkpointing and dynamic loss scaling (`GradScaler`) allowing us to train with larger effective batch sizes.

Loss was monitored per step and averaged across epochs. Training loss plateaued around 600–800 iterations, indicating convergence for the given dataset and setup. Though in total it took `7500 steps per epoch`.

Output

- The fine-tuned LoRA adapter weights were saved locally using `model.save_pretrained()`. These adapters capture only the LoRA-specific modifications and can be merged or reused later.
- For final inference, the base BLIP model was loaded and augmented with the saved LoRA adapter via `PeftModel.from_pretrained()`, enabling modular and scalable deployment.
- The final fine-tuned adapter is publicly available on the Hugging Face Hub at: `bk45/blip-vqa-finetuned`

The processor configuration was saved together with the adapter.

6 Evaluation Metrics

The performance of both the **baseline BLIP-VQA-Base model** and our **LoRA fine-tuned variant** was evaluated using a range of metrics tailored for the **Visual Question Answering (VQA)** task, particularly focusing on the challenge of **single-word answers**.

Initially, we relied on `BERTScore` as the primary evaluation metric due to its ability to capture **semantic similarity**. While the baseline model achieved high `BERTScore` values (around **0.95**), its **Exact Match Accuracy** was relatively low (**0.1725**), indicating that high semantic similarity did not always correspond to **correct or precise answers**.

This revealed a limitation of using `BERTScore` alone, as it sometimes failed to fully reflect answer correctness or subtle distinctions important in VQA.

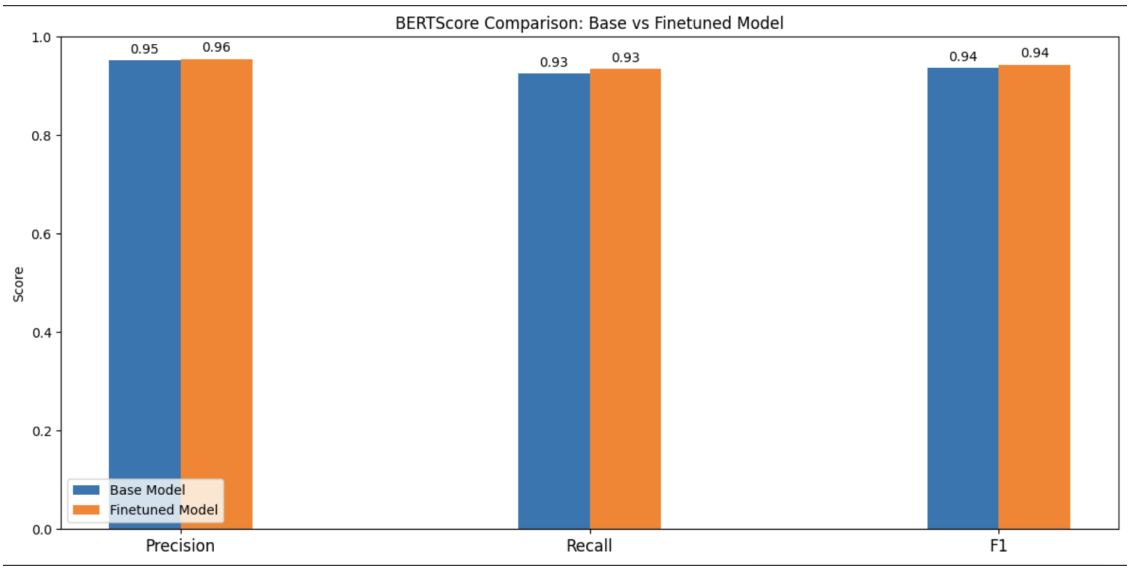

Figure 3: Comparison between Base and Finetuned Model Bert Score Metrics

To gain a more comprehensive understanding of model performance, we incorporated additional metrics:

- **Exact Match Accuracy**: Measures the strict correctness of the model's predictions.
- **METEOR Score**: Considers precision, recall, and synonymy to better capture partial matches.
- **SBERT Similarity**: Measures sentence-level semantic similarity beyond token-level matches.
- **WUPS (Word-Utilized Precision Score)**: Evaluated at two thresholds—`0.0` (lenient) and `0.9` (strict)—to assess similarity based on an ontology.

The **LoRA fine-tuned model** showed consistent improvements across all these metrics:

- **Exact Match Accuracy**: Improved from **0.1725** to **0.2864**.
- **METEOR Score**: Rose from **0.0994** to **0.1496**.
- **SBERT Similarity**: Increased from **0.4584** to **0.5630**.

These gains demonstrate that **LoRA fine-tuning** effectively enhanced the model's ability to generate **more accurate and semantically relevant answers**.

Terminology

- **Exact Match Accuracy:** Measures the percentage of model predictions that exactly match the ground truth answers. Useful for strict evaluation, especially with short or single-word answers.
- **BERTScore (Precision, Recall, F1):** Uses contextual embeddings from a pre-trained BERT model to compare the semantic similarity between predicted and reference answers.
  - **Precision:** How much of the predicted answer is semantically relevant to the ground truth.
  - **Recall:** How much of the ground truth is captured by the prediction.
  - **F1:** Harmonic mean of precision and recall.

  Effective for measuring semantic similarity but may overestimate correctness in vague predictions.
- **WUPS (Wu-Palmer Similarity) @0.0 and @0.9:** Calculates similarity based on a lexical database (WordNet).
  - **@0.0 (Lenient):** Accepts loosely related terms as correct.
  - **@0.9 (Strict):** Only highly similar or synonymous words are accepted.

  Useful for balancing lexical correctness with semantic leniency.
- **Final Weighted WUPS Score:** A weighted average of WUPS@0.0 and WUPS@0.9 that offers a balanced measure of lexical similarity.

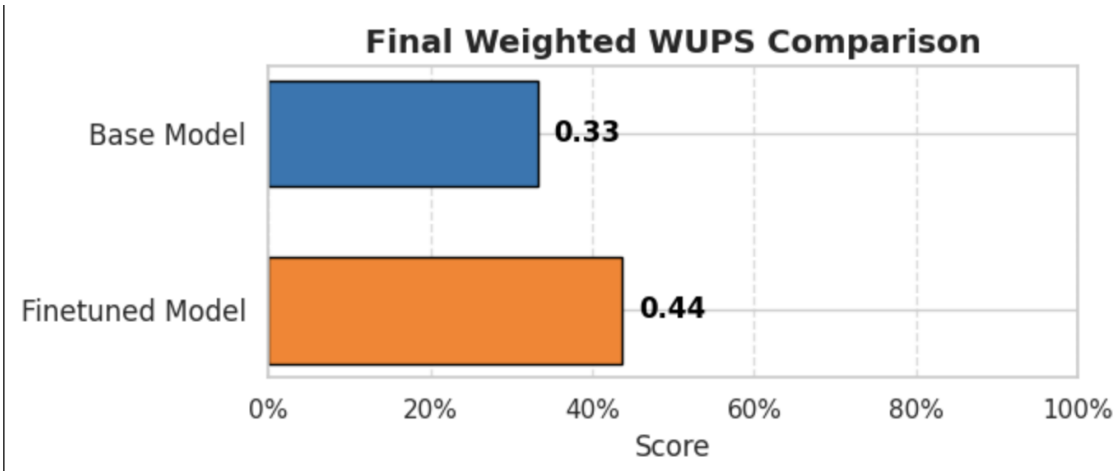

Figure 4: Comparison between Base and Finetuned Model Final Weighted WUPS Score Metrics.

- **SBERT Similarity (Sentence-BERT):** Computes cosine similarity between sentence embeddings of the predicted and true answers using Sentence-BERT. Provides a robust measure of higher-level semantic similarity.
- **METEOR Score:** Based on precision, recall, and alignment of word stems, synonyms, and paraphrases. Originally designed for machine translation but effective for short and phrase-based VQA responses.

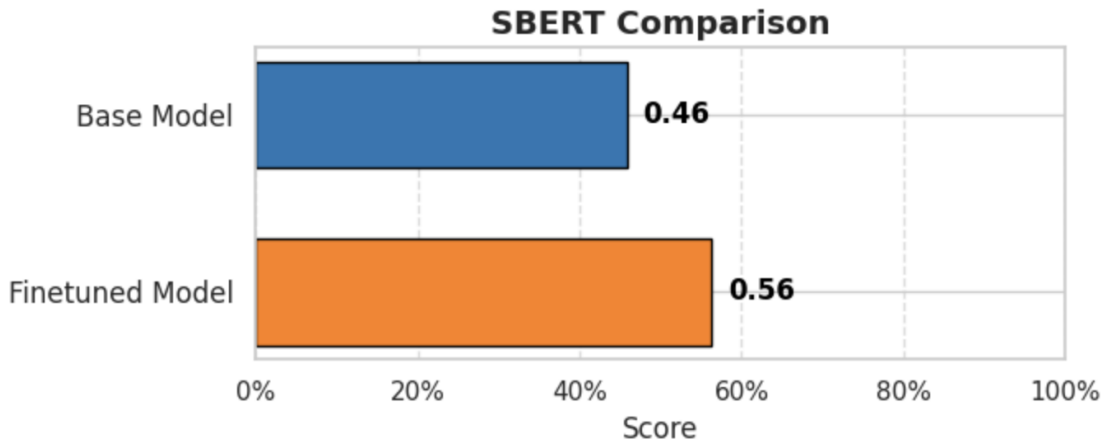

Figure 5: Comparison between Base and Finetuned Model SBert Score Metrics.

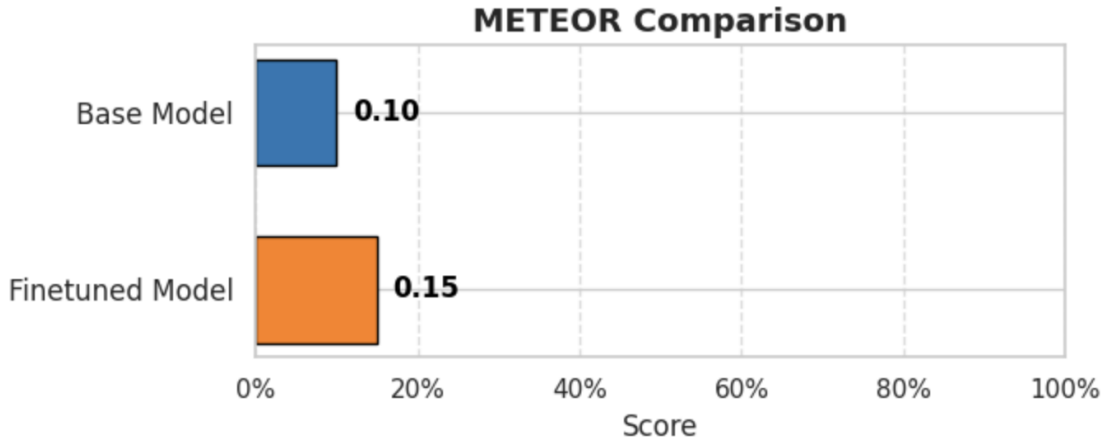

Figure 6: Comparison between Base and Finetuned Model Average METEOR Score Metrics.

7 Observations and Analysis

The fine-tuning process with LoRA on the BLIP-VQA-Base model demonstrated notable improvements across all evaluation metrics. Most significantly, the Exact Match Accuracy increased from approximately 17.25% for the baseline model to 28.64% for the LoRA fine-tuned version, indicating a marked enhancement in the model's ability to produce precise answers.

The baseline BLIP-VQA-Base model, while capable of understanding and responding to visual questions, showed limitations in consistently generating exact matches with the ground truth answers. This led to relatively modest performance in metrics such as
`Exact Match and WUPS scores`. Additionally, the baseline often produced answers that, although valid in context, lacked semantic closeness to the ground truth, as reflected in lower SBERT similarity and METEOR scores.

`After fine-tuning with LoRA:`

- The model significantly improved its capacity to generate the specific single-word answers expected by the dataset, reducing the occurrence of irrelevant or generic tokens.
- Improvements in METEOR and SBERT similarity scores indicate that even when the model did not produce exact matches, its answers were semantically closer and more relevant to the ground truth.

Table 1: Baseline vs. LoRA Fine-Tuned Model

No.	Metric	Baseline (BLIP-VQA-Base)	LoRA Fine-Tuned Model
1	Exact Match Accuracy	0.1725	0.2864
2	BERTScore (Precision)	0.9517	0.9552
3	BERTScore (Recall)	0.9251	0.9344
4	BERTScore (F1)	0.9371	0.9438
5	WUPS @ 0.0 (Lenient)	0.4538	0.5673
6	WUPS @ 0.9 (Strict)	0.2116	0.3064
7	Final Weighted WUPS	0.3327	0.4369
8	SBERT Similarity	0.4584	0.5630
9	Average METEOR Score	0.0994	0.1496

- The LoRA fine-tuning effectively adapted the pretrained model to the specific downstream task with minimal computational cost compared to full fine-tuning, focusing on salient visual features to produce concise answers..

Overall, the fine-tuning process successfully addressed the unique requirements of the VQA task with single-word answers, resulting in a model that is substantially more accurate and reliable for this dataset.

8 Running Inference Script

The `inference.py` script is designed to perform Visual Question Answering using the fine-tuned `BLIP-VQA-Base` model. It loads the pretrained `BLIP-VQA-Base` architecture, applies the trained LoRA adapters from the Hugging Face Hub, and generates answers for questions provided in the `train_dataset.csv` file.

8.1 Key Features of the Inference Script

1. **Model and Adapter Loading:**
   - The script loads the base `Salesforce/blip-vqa-base` model.
   - It then loads and applies the fine-tuned LoRA adapter weights from the specified Hugging Face Hub repository (`bk45/blip-vqa-finetuned`). These weights are integrated into the base model using `peft.PeftModel`, enabling efficient and accurate inference without altering the original model architecture.
   - The model is transferred to the appropriate device (GPU or CPU) and set to evaluation mode.
2. **Processor and Input Formatting:**
   - The `BlipProcessor` from the base model is used to prepare the inputs.
   - Each image-question pair is processed using this processor to ensure compatibility with the model, including padding, truncation, and maximum token length handling.
3. **Data Flow and Batching:**
   - A custom PyTorch `Dataset` class is defined to load image paths and corresponding questions from a given DataFrame.
   - Images are retrieved from a specified directory and converted to RGB format.
   - A `DataLoader` is used to efficiently batch the dataset, with support for parallel loading using multiple workers.
4. **Answer Generation:**
   - The model uses the `generate()` function to produce token IDs corresponding to the predicted answers.

- These token IDs are decoded using the processor's tokenizer with `skip_special_tokens=True` to yield clean textual outputs.

5. **Post-processing and Output:**
   - The decoded answers are stored in a list and appended as a new column, `predicted_answer`, to the original DataFrame.
   - The updated DataFrame now contains the model's predictions aligned with the original questions for further evaluation or manual inspection.
6. **Inference Optimization:**
   - **Flash Attention 2:** If a CUDA-enabled device is available, Flash Attention 2 is automatically enabled. This significantly improves the speed and efficiency of the attention mechanism while reducing memory consumption during inference.
   - **KV Caching:** The model leverages key-value (KV) caching during generation (with `use_cache=True`), allowing it to reuse previously computed attention states. This greatly accelerates the generation of subsequent tokens in the answer.

8.2 Setup and Usage Instructions

1. **Prerequisites:**
   - Make sure Python version 3.9 or later (≥ 3.9) is installed.
   - Create a new environment using `conda`.
   - Install the required Python packages. First, navigate to the `inference-setup` folder on GitHub and run the command below.
   - Repeat the same process inside the `sample_submission/IMT2022026_095_549` directory.
   - Run the following command to install dependencies:

     ```
     pip install -r requirements.txt
     ```

2. **Preparation of Data:**
   - **Image Directory:** A folder that holds all the images mentioned in the CSV file.
   - **CSV File:** A CSV input file (e.g., metadata.csv) that includes at minimum the columns `image_name` and `question`.
3. **Run the Script:** Execute the script from your terminal after having installed all the dependencies from requirements.txt inside `sample_submission/IMT2022026_095_549`.

   ```
   python inference.py
   --image_dir /path/to/your/image_folder
   --csv_path /path/to/your/input.csv
   ```

4. **Output:**
   - A visual progress bar is displayed during the inference process.
   - After completion, a CSV file is produced (default: `results.csv`, or as specified by the `--output_csv_path` argument), containing the original input data along with an added column for the model's predicted answers.