Report: Multimodal Visual Question Answering with Amazon Berkeley Objects Dataset

GitHub repo: https://github.com/subham-agarwal05/Multimodal-Visual-Question-Answering

Dataset: https://huggingface.co/datasets/subhamagarwal0512/VQA/tree/main

Team

Subham Agarwala: IMT2022110

Kaustubh Manda: IMT2022027

Ishan Jha: IMT2022562

Data curation

The goal of the data curation pipeline is to convert raw ABO (Amazon Berkeley Objects) metadata and images into a clean, filtered CSV of multimodal examples—with each row linking an image to one or more generated question–answer pairs. The pipeline comprises three main stages:

1. **Data Extraction**: Decompress raw JSON metadata archives.
2. **Data Filtering & Structuring**: Parse and filter metadata to extract relevant text fields, join with image availability, and output per-listing CSVs.
3. **Question Generation**: For each image–metadata pair, invoke a multimodal LLM API (e.g. Google Gemini) to generate diverse, single-word–answer questions.

---

1. Data Extraction and Curation

```
import gzip, shutil, os

source_dir = '/kaggle/input/abo-listings/'
dest_dir   = '/kaggle/working/abo-listings/listings/extracted_metadata'
os.makedirs(dest_dir, exist_ok=True)

for fname in os.listdir(source_dir):
    if fname.endswith('.json.gz'):
        gz_path = os.path.join(source_dir, fname)
        out_path = os.path.join(dest_dir, fname[:-3])  # strip ".gz"
        with gzip.open(gz_path, 'rb') as f_in, open(out_path, 'wb') as f_out:
            shutil.copyfileobj(f_in, f_out)
        print(f"Decompressed {fname} → {os.path.basename(out_path)}")
print("Extraction complete.")
```

- **Rationale**: ABO metadata comes as `.json.gz` files (~3 GB small variant). Decompression creates plain JSON for downstream parsing.

---

2. Data Filtering & Structuring

2.1. Parsing Metadata

Each JSON file contains a list of product records, with multilingual fields (e.g. `title`, `description`, `colour`, `material`). We:

1. **Define output schema**:

```
['main_image_id',
 'overall_description',
 'colour_description',
 'other_description',
 'material_description']
```

2. **Language filtering**:

```
def extract_values_by_language(entries, lang='en'):
    return [ rec['value']
             for rec in entries
             if rec.get('language') == lang ]
```

1. **Assemble rows** by joining all values for each field (semicolon-separated).

2.2. Joining with Image Metadata

- We load the ABO standalone image-metadata CSV (from `abo-images-small`) to map `main_image_id → file path`.
- We **filter out** any listings whose images are missing locally.
- We write, for each listing JSON file, a corresponding CSV (one row per listing) under:

```
/kaggle/working/abo-listings/listings/filtered_metadata/<original_json_name>.csv
```

Each row now contains exactly one image ID and its associated text descriptions.

---

3. Question Generation

5.1. Prompt Design

For each filtered row, we generate up to N question–answer pairs using a templated prompt structure. Example template for “colour” questions:

> Prompt (sent as part of a multimodal request):
>
> ```
> "You are given an image and a brief product description.\n"
> f"Use the product description context: {combined_description}\n"
> "Generate exactly 5 diverse, visually clear, and progressively challenging questic
> "Each question must be answerable by only looking at the image — do NOT rely
> "Ensure variation in the *type* of visual cues used: color, shape, count, spatial re
> "Ensure variation in *difficulty level*:\n"
> "- At least 2 simple questions (e.g., color, count)\n"
> ```

> "- At least 2 moderately difficult questions (e.g., spatial relations, comparisons)\r
> "- 1 challenging question requiring closer inspection or subtle visual reasoning (e
> "Do NOT ask about materials or properties that are not visually obvious (e.g., pla
> "Answers must be a single word — not all of them 'yes' or 'no'.\n"
> "Strictly use this format without extra text:\n"
> "Question 1: <question>\n"
> "Answer 1: <answer>\n"
> "Do not include any explanations or extra text."
>
> Response:
>
> • Q: What is the color of the bag?
>
> • A: __”

We dynamically assemble prompts according to which description fields are non–empty (overall, colour, material, etc.), aiming for **single-word answers**.

3.2. API Client & Rate Limiting

```
from google import genai

client = genai.Client(api_key="YOUR_API_KEY")
MAX_DAILY = 1500
DELAY_SECS = 60     # between batches

requests_made = load_progress(progress_file)
for idx, record in enumerate(reader):
    if idx < requests_made:  # skip already processed
        continue
    ...
    response = client.generate_image_question(
        image_path=img_path,
        prompt=constructed_prompt
    )
    write_to_csv(output_file, question, answer)
    save_progress(progress_file, idx + 1)
    time.sleep(DELAY_SECS)
```

- **Progress files** (e.g. `progress_<list_filename>.txt`) store the index of the last processed record, enabling safe restarts without duplication.
- We enforce a **delay** to avoid hitting API quotas and resume within daily limits.

3.3. Output

All generated QA pairs for a given listing are appended to:

```
generated_questionquestions_<list_json_name>_<question_set_number>.csv
```

Each CSV has columns:

```
listing_id, image_id, prompt_type, question, answer
```

4. Preprocessing & Quality Controls

- **Empty-field checks**: Skip question types when the corresponding metadata list is empty.
- **Deduplication**: Within each listing, ensure no identical Q–A pair is written twice.
- **Logging**: Print progress every 50 records and capture any API errors to a separate error log for later inspection.

NOTE: The questions with answer as yes/no were removed from the dataset before training, to finetune the model more robustly. Yes/No questions were also boosting the BERT score which prevented us from gauging the performance of the fine tuned model accurately.

Model Choice and Baseline Evaluation

▼ Model Choice

Before moving on to fine-tune a model, we carried out a baseline evaluation of some pre-trained models to set a benchmark for how the models were performing on our curated dataset. The models are as follows:

1. BLIP-VQA-Base:

▼ Evaluation Metrics

1. **Exact Match (EM) Accuracy:**
   - **Measures:** The percentage of predictions identical to reference answers.
   - **Formula:** EM = (Count of Exact Matches) / (Total Samples)
   - **Interpretation:** Higher score = more perfectly correct answers.
2. **BERTScore (F1):**
   - **Measures:** Semantic similarity using contextual embeddings.
   - **Formula:** F1_BERT = 2 * (P_BERT * R_BERT) / (P_BERT + R_BERT)
     - Where P_BERT (precision) and R_BERT (recall) are calculated based on cosine similarity of token embeddings between candidate and reference.
   - **Interpretation:** Higher F1 = better semantic overlap.
3. **BLEU Score (avg):**
   - **Measures:** N-gram precision with a brevity penalty.
   - **Formula:** BLEU = BP * exp(Σ_{n=1 to N} w_n * log p_n)
     - BP: Brevity Penalty (penalizes short candidates).
     - p_n: Modified n-gram precision.
     - w_n: Weights (typically uniform, e.g., 0.25 for N=4).
   - **Interpretation:** Higher score = better n-gram overlap with reference.
4. **ROUGE-1 F1 Score (avg):**
   - **Measures:** Overlap of unigrams (individual words), balancing precision and recall.

- **Formula:** ROUGE-1_F1 = 2 * (P_1 * R_1) / (P_1 + R_1)
  - P_1 = (Count of matching unigrams) / (Total unigrams in candidate)
  - R_1 = (Count of matching unigrams) / (Total unigrams in reference)
- **Interpretation:** Higher F1 = better individual word overlap.

5. **ROUGE-L F1 Score (avg):**
   - **Measures:** Longest Common Subsequence (LCS) of words, balancing precision and recall.
   - **Formula:** ROUGE-L_F1 = 2 * (P_LCS * R_LCS) / (P_LCS + R_LCS)
     - P_LCS = LCS(candidate, reference) / (Length of candidate)
     - R_LCS = LCS(candidate, reference) / (Length of reference)
   - **Interpretation:** Higher F1 = better structural similarity via shared word sequences.

▼ Performance

Model	Exact Match Accuracy	**BERTScore (F1)**	**BLEU Score (avg)**	**ROUGE-1 F1 Score (avg)**	**ROUGE-L F1 Score (avg)**
BLIP-VQA-Base	0.4661	0.7541	0.0854	0.4864	0.4864
BLIP 2	0.5444	0.8565	0.1521	-	-
Matcha Chart QA	0.3642	0.6431	0.01754	-	-

▼ Justification for finetuning BLIP-VQA-Base

While some other models perform better, their size is relatively large. To showcase the potential of LoRA we wanted to fine-tune a small model and see by how much we can improve the performance of the pretrained model. BLIP, being a familiar model, was the perfect choice with Total parameters: 384,672,572. Also, the particular variant of BLIP is designed specifically for VQA.

Finetuning

Low-Rank Adaptation and Quantisation

For fine-tuning, we will use the packages ` Peft ` and ` LoRA `. LoRA or Low Rank Adaptation is a method that involves the reduction of gradient change through approximating the matrix to lower rank factors. In this method, we prefer using a matrix of rank 16. We limit the use of the LoRA Adapter to only specific parts of the model, which can be found in the LoRA config sections for each fine-tuning notebook.

LoRA for Matcha-Chart-QA

```
# Lora Config for Matcha
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=[
        "attention.query",
        "attention.key",
        "attention.value",
        "mlp.wi_0",
        "mlp.wo",
```

```
        "mlp.DenseReluDense.wi_0",
        "mlp.DenseReluDense.wo"
    ],
    lora_dropout=0.15,
    bias="none",
    task_type="CAUSAL_LM"
)
model = get_peft_model(model, lora_config)
```

LoRA for BLIP-VQA

```
#LoRA Config for BLIP-VQA
TARGET_MODULES = ["q_proj",
        "k_proj",
        "v_proj",
        "mlp.fc1",
        "mlp.fc2",
        "output.dense"
]

R_LORA = 16
LORA_ALPHA = 32
LORA_DROPOUT = 0.05

peft_config = LoraConfig(
    r=R_LORA,
    lora_alpha=LORA_ALPHA,
    target_modules=TARGET_MODULES,
    lora_dropout=LORA_DROPOUT,
    bias="none"
)
```

LoRA for BLIP-2-OPT

```
# LoRA for BLIP-2-OPT
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=[
        "q_proj",
        "v_proj",
        "k_proj",
        "down_proj",
        "up_proj"
    ],
    lora_dropout=0.15,
    bias="none",
    task_type="CAUSAL_LM"
)
model = get_peft_model(model, lora_config)
```

We prefer using a smaller rank for BLIP-2-OPT due to the model size. Since this is a large 3B+ parameter model, we would like to reduce the update matrix size further and hence use rank=8 matrices. This is currently paired with Quantization methods(which decrease the model size by applying reduced precision) using the library BitsAndBytes. (We generally tested with both 8bit quantization and 4 bit quantization as well).

```
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype="float16",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4"
)
```

Data and PyTorch Strategy

Each question is processed by combining it with a prompt as follows. This ensures that the model always gives us an output consistent with the required solution (ONE Word answers).

```
def GeneratePrompt(question):
    return (
        "You are a helpful visual question answering assistant.\n"
        "Answer the question about the image using ONE word only.\n"
        f"Question: {question}\n"
        "Answer:"
    )
```

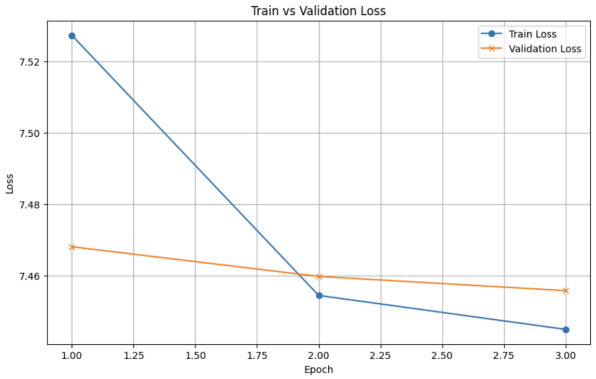
The actual training mechanism uses the class VQADataset, which helps in the model fine-tuning. This accesses the data using pandas. The data is present in the CSV file that has question, answer and full_image_path columns. While accessing the data point through the **__getitem__**() attribute, we index to that specific element and create the prompt corresponding to the question, open the image using the Image libraries open method and convert it to a standard RGB image. The question, answer and the image are processed using the corresponding processor and give the necessary encoded representations which can be used by the model while training.

We use both manual training loops by PyTorch and Trainer from the transformers library. This helps us efficiently train the model. The trainer class ensure that we are logging and saving the model at certain given time steps, which can be used.

Blip fine-tuning performance over epochs

During the fine-tuning process of the BLIP model on our custom Visual Question Answering (VQA) dataset, we observed the validation loss across multiple training epochs to monitor learning progress and prevent overfitting.

After the first epoch, the model showed a noticeable drop in validation loss, indicating that it was beginning to learn meaningful patterns from the data. By the end of the second epoch, the validation loss plateaued around **7.4**, and subsequent epochs yielded minimal to no improvements. In fact, continuing training beyond the second epoch not only failed to reduce the loss further but also showed signs of potential overfitting — with training loss decreasing while validation loss remained stagnant.



Given this observation, we decided to fine-tune our final model over **three epochs**. This allowed us to take advantage of the initial learning phase while avoiding unnecessary training that might degrade generalization performance. Our decision was also motivated by computational efficiency, as longer training did not yield measurable gains in validation performance.

Choice of LoRA Rank (r=16)

We began our fine-tuning experiments using a **LoRA rank (`r`) of 8**, which provided a good trade-off between training efficiency and model performance. Encouraged by the initial results, we subsequently increased the rank to **r=16** to explore whether a higher rank would enable the model to better capture task-specific representations.

The model fine-tuned with `r=16` consistently outperformed the lower-rank version in terms of validation loss and qualitative answer accuracy. However, after analyzing the improvements relative to the additional memory and compute costs, we opted **not to increase the rank further**. This decision was made to **keep the model lightweight** and **ensure faster inference**, especially considering resource-constrained environments.

Thus, we selected **r=16 as the final configuration**, balancing performance gains with practical limitations in training time and model size.

Choice of target modules for BLIP-VQA-Base

1. **q_proj, k_proj, v_proj:** Control **attention** – how the model weighs and relates different parts of the image and question. Finetuning these helps it focus on relevant information for your VQA task.
2. **mlp.fc1, mlp.fc2:** Form the **feed-forward networks** that process information after attention. Finetuning these allows the model to learn more complex, task-specific transformations and reasoning.
3. **output.dense:** Often part of the attention output or the final layer for **answer generation**. Finetuning this adapts how attention outputs are combined or how the final representation is mapped to an answer.

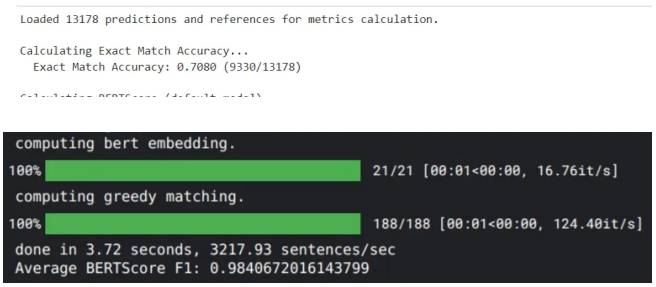
In short, finetuning these allows the model to **learn what to look at and how to reason about it** specifically for your VQA dataset, without retraining the entire model.

Results of Fine-Tuning and Post-Fine-Tuning Evaluations

We have carried out fine-tuning only on BLIP and Matcha. We were not able to extend our fine-tuning to BLIP2 due to computational restrictions. Fine-tuning seems to yield outstanding results in both model performances. BLIP, which only had a BERT Score of 0.751 initially, has managed to increase it's BERT

Score to 0.98406. We also tested our fine-tuned model on test data from other teams and saw similar scores.

```
Total samples evaluated: 131847
Exact Match Accuracy: 61456/131847 = 0.4661
BERTScore (F1): 0.7541
BLEU Score (avg): 0.0854
ROUGE-1 F1 Score (avg): 0.4864
ROUGE-L F1 Score (avg): 0.4864
```



For Matcha, we have seen similar increments. In the version we were initially using, a fine-tuned version of Matcha, fine-tuned (done by Google) on Chart-based data from various Data Visualisation sources, gives significant performance improvements after our custom fine-tuning when compared to its previous version.

```
Avg BERTScore F1: 0.8119
Avg BERTScore Precision: 0.8119
Avg BERTScore Recall: 0.8119
```

The table below shows the results combined:

Model	EMA(base)	EMA(fine-tuned)	BERT score (base)	BERT score (fine-tuned)
BLIP	0.4661	0.7080	0.7541	0.98406
Matcha	0.3642	0.3565	0.6431	0.8119

NOTE: The inference script uses the BLIP fine-tuned model.