VR Mini Project 2 Report

Siddharth Reddy Maramreddy
IMT2022031
IIIT - Bangalore
Bengaluru, India
Siddharth.Maramreddy@iiitb.ac.in

Sai Venkata Sohith Gutta
IMT2022042
IIIT - Bangalore
Bengaluru, India
Sohith.Gutta@iiitb.ac.in

Margasahayam Venkatesh Chirag
IMT2022583
IIIT - Bangalore
Bengaluru, India
MV.Chirag@iiitb.ac.in

I. Introduction

This project focuses on building a curated Visual Question Answering (VQA) dataset using the Amazon-Berkeley Objects (ABO) dataset. The goal is to develop an effective VQA pipeline by leveraging pretrained vision-language models, followed by fine-tuning them on our custom dataset to optimize performance. This approach enables the models to understand object-specific visual questions better and produce more accurate answers in real-world scenarios.

II. Data Curation

We first loaded all the JSON files containing metadata, parsed all the listings, which were in the form of JSON objects, into Python dictionaries. We shuffled the dictionaries because we were planning to select only a subset of the images, and we did not want the dataset to have a large number of similar products.

Since we had already planned to select a small number of images (20k), we decided to take only the main images from the listings to create the dataset. Each JSON object in the file contained information about one Amazon product listing. Each product listing could have one main image at max, but can have multiple 'other images'. Since there were nearly 147000 listings and we only needed 20k images, we only used the main images from the listings.

To connect the metadata to images, we first converted the CSV which which contained the image ID and the image path, into a Pandas Dataframe. As we mentioned previously, most of the listings contained the `main_image_id` attribute, which contained the image ID of the corresponding image. So for a given listing, we get its `main_image_id` and search for the image ID in the Dataframe to obtain the image path.

The listing contained so many attributes, and we found that some of them were not useful for this project. They contained data about the product which could not be deduced from the image or some redundant information. Some of such attributes are `item_weight`, `color_code`, `marketplace`, `model_name`, `model_number`, etc. So, we have removed all the unnecessary attributes and used only a few selected attributes. The selected attributes are as follows

- `bullet_point`
- `color`
- `product_type`
- `fabric_type`
- `finish_type`
- `item_keywords`
- `item_name`
- `item_shape`
- `material`
- `pattern`
- `product_description`
- `style`

We parsed all the data from these attributes based on the specification given in the Readme file. The final metadata which we pass to the model is a string in which the data corresponding to each attribute is separated by a newline character. If an attribute has multiple data values, they are separated by commas.

Some of the listings were in different languages. Since we were supposed use only English, we checked all the attributes first and used the listing only if it had data in English for at least five attributes. We had checked to ensure there were enough listings which satisfy the above criteria. We found that around 98000 images satisfied out criteria.

Regarding the prompt for the model, we have experimented with multiple prompts. Some of the prompts are given below

- **Prompt 1:** *Using the image and its metadata, generate unambiguous questions with one-word answers which should be unambiguously answerable from the image. Since I need to automate parsing these question and answers, please provide them in CSV format: question, answer. Please do not generate anything else other than question and answers as it makes it difficult to write an automated parser.*
- **Prompt 2:** *I am preparing a dataset to train a Visual Question Answering (VQA) model. I have a set of images and corresponding metadata from*

Amazon product listings (Amazon Berkeley Object Dataset). Using the image and its metadata, generate unambiguous questions with one-word answers which should be unambiguously answerable by 'SOLELY' looking at the image. Since I need to automate parsing these question and answers, please provide them in CSV format: question, answer. Please do not generate anything else other than question and answers as it makes it difficult to write an automated parser.

- **Prompt 3:** *I am preparing a dataset to train a Visual Question Answering (VQA) model. I have a set of images and corresponding metadata from Amazon product listings (Amazon Berkeley Object Dataset). Using the image generate questions with single unambiguous one-word answers in English which should be answerable by S̈OLELYÏooking at the image without providing any other data.Use its metadata to verify the answers. Each question has to be independent of the others. Since I need to automate parsing these question and answers, please provide them in CSV format: question, answer. Please do not generate anything else other than question and answers as it makes it difficult to write an automated parser.*

We had used multiple other prompts which we did not keep a record of. The last prompt gave the best results. That was the one we used to generate our datasets.

We tried using multiple different ways to generate datasets. First, we tried using the Google Developers API. We used **gemini-2.0-flash**, which is the latest model and works with multimodal data. The API had rate limits which were 30 requests per minute and 1500 requests per day for this model. There was another model called **gemma3** which had a higher rate limit (nearly 15000 request per day). Although the model supports images, it does not support images through the Google Developers API. This approach was very slow and took a lot of time to even generate questions and answers for 1000 images.

We explored alternate ways in which we could use **gemma3** and found out about the Google Cloud API in which **gemma3** supports images. But we also found out that it has the same rate limits for all models and that it is not a free API. But it gives $300 free credits. So, we decided to use the best model **gemini-2.0-flash** instead of **gemma3** for this API. From our analysis we also found out that the credits are more than sufficient to get q&a for all 147742 images.

First we used the Google Cloud API to generate a dataset for 6k images. It still took around 3 to 4 hours to generate q&a for all the images. It had around 20k q&a.

While testing the models with our prompts, we found out the model sometimes gives question which cannot be answered from the images. To reduce such questions and also for repeatability, we used the following parameters for the model

- `temperature`: 0.0
- `top_k`: 1
- `top_p`: 1.0
- `seed`: 42
- `max_output_tokens` = 256

We have set a limit for output tokens to prevent the model from generating too many question.

The model usually gave output according to the format we asked in the prompt. Since we asked the model to give q&a in CSV format, it was easy to parse. We parsed all the q&a and stored them in a CSV file containing the following columns: `image_path`, `question` and `answer`.

We have also tried to run Ollama locally, but the MacBook did not have the required capacity. It restarted as soon as we tried to run it. Since the other model was giving good results, we did not experiment with this much.

After the fine-tuning gave decent result on out 6k image dataset, we decided to generate a dataset with 20k images. We also found out that the Google Cloud API support multiple threads, we used 8 threads to speed up the process. It took only 2 hours to generate a dataset with almost 90k question and answers.

Since there was a repetition of images in some of the listings and we had set the parameters such that it generates same questions everytime, we made sure to use one image only once by keeping track of the images already used.

III. Baseline Evaluation

A. Pretrained Models Overview

For our pretrained models evaluation we employed four state-of-the-art vision-language models:

- **BLIP** (`Salesforce/blip-vqa-base`): Pretrained on both captioning and VQA data, it excels at pinpointing visual cues for concise answers.
- **LLaVA** (`unsloth/llava-1.5-7b-hf-bnb-4bit`): A 7B-parameter LLM pretrained on text and minimal multimodal data, but *not* VQA-fine-tuned. Produces fluent but often verbose responses in zero-shot.
- **Qwen** (`Qwen/Qwen2.5-VL-7B-Instruct`): A 7B-parameter instruction-tuned language model. While pretrained on a mix of text and image-text data, it lacks task-specific VQA fine-tuning. In zero-shot, it generates contextually appropriate but predominantly multi-word or sentence-level answers, leading to low exact-match scores despite strong semantic relevance.

- **VilBERT** (`dandelin/vilt-b32-finetuned-vqa`): Pretrained on large image-text corpora and fine-tuned on VQAv2.0. Strong token-region alignment yields robust single-word attribute extraction.

B. Detailed Prediction Analysis of Pretrained Models

a) BLIP (`Salesforce/blip-vqa-base`):

- **Answer Form:**
  - Single-word in ∼93.5% of cases.
  - Multi-word qualifiers (e.g. “dark green”, “a shade of red”) in ∼6.5%. This was made single-word by performing post-processing, which has been explained in the next section.
- **Common Single-Word Errors:**
  1) “yes” (over-affirming binary questions)
  2) “white” / “black” (contrast confusion)
  3) “blue” (color conflation)
  4) “plastic” (material label on colour/pattern queries)
- **Commentary:** When BLIP errs on single-token questions, it defaults to high-frequency labels (e.g. “yes”, “white”) or confuses attribute categories. Its multi-word qualifiers show nuance but penalize exact-match metrics, which was the reasoning behind performing post-processing to convert multi-word qualifiers into single-word.

b) LLaVA (`unsloth/llava-1.5-7b-hf-bnb-4bit`):

- **Answer Form:**
  - Single-word in ∼68.4% of cases.
  - Multi-word answers (e.g., “dark blue color”, “most likely red”) in ∼31.6% of cases. These were semantically relevant but penalized under strict lexical metrics.
- **Common Single-Word Errors:**
  1) “yes” / “no” (over/under generalization)
  2) “red” / “blue” (training frequency bias)
  3) “fabric” (category drift from attribute)
  4) “shiny” (descriptive terms not aligned with label set)
- **Commentary:** LLaVA’s instruction-tuned nature helps it generate semantically grounded responses, but it often includes qualifiers or slight rephrasings that diverge from the ground truth lexicon (e.g., “probably green” vs. “green”). This results in lower exact-match and F1 scores despite high ROUGE-L and BERTScore. Its flexibility is both a strength and a liability in strict evaluation settings, motivating format-constrained fine-tuning or post-processing for improved performance.

c) Qwen (`Qwen/Qwen2.5-VL-7B-Instruct`):

- **Answer Form:**
  - Predominantly multi-word or full-sentence outputs (e.g. “The color is green”).
  - Single-word matches are extremely rare (<5%).
- **Common Failure Modes:**
  - Generates plausible but off-annotation text with correct semantics, e.g. “it appears blue” when the GT is “blue”.
  - High BERTScore indicates semantic relevance, but low lexical overlap produces near-zero Exact Match and F1.
- **Commentary:** Qwen lacks VQA supervision, so it relies on its text-instruction training. It often offers well-formed sentences that semantically relate to the image, yet misalign with the dataset’s concise label format.

d) VilBERT
: *(`dandelin/vilt-b32-finetuned-vqa`)*

- **Answer Form:**
  - Single-word in ∼96.9% of cases.
  - Multi-word in ∼3.1%. This was made single-word by performing post-processing, which has been explained in the next section.
- **Error Patterns:**
  - Occasional color swaps (e.g. “white” vs. “gray”).
  - Rarely drifts into non-attribute tokens.
- **Commentary:** VilBERT yields a disciplined prediction distribution-very few hedges or wildcards and consistently single-token outputs.

e) Cross-Model Patterns:

- **Binary Questions (Yes/No):** BLIP and VilBERT both show balanced handling of yes/no questions. LLaVA tends to be concise but occasionally adds qualifiers like “probably yes,” which reduce exact-match. Qwen often responds with elaborative phrases such as “Yes, it appears to be,” which degrade token-level metrics despite semantic relevance.
- **Color Confusions:** BLIP and VilBERT frequently confuse similar high-frequency colors under low-light or ambiguous image conditions (e.g., red vs. pink). BLIP adds qualifiers (“light red”), while LLaVA is relatively accurate but still shows some drift toward commonly seen training-set color names. Qwen produces abstract or overly descriptive terms (e.g., “a bluish tone”) that fail exact matching.
- **Attribute Category Errors:** BLIP occasionally answers with material or general appearance rather

than the specific queried attribute. LLaVA, although instruction-tuned, sometimes returns synonyms or rephrasings that are semantically valid but lexically incorrect (e.g., "nylon" vs. "fabric"). Qwen often includes multiple attributes in one answer, further increasing deviation.

- **Answer Lengths:** BLIP and VilBERT mostly generate single-token answers, aligning well with the dataset. LLaVA outputs short phrases with moderate consistency. Qwen almost exclusively outputs multi-word or sentence-level answers, severely impacting exact-match and F1 but maintaining reasonable semantic similarity. This issue was addressed by performing post-processing.

C. *Quantitative Comparison of Pretrained Models*

Among the pretrained models evaluated in the zero-shot setting, BLIP demonstrates the strongest overall performance. It achieves the highest exact match accuracy, indicating that it is better aligned with the single-word answer format of the curated dataset. In addition to lexical accuracy, BLIP also scores well on token-level F1, ROUGE-L, and semantic metrics such as BERTScore and Semantic Cosine Similarity, reflecting both its precise token alignment and strong semantic grounding.

LLaVA, on the other hand, performs significantly worse in terms of exact match accuracy, indicating its difficulty in matching the exact lexical form of the ground truth answers. However, it shows relatively higher BERTScore and semantic similarity, suggesting that while its answers may not match word-for-word, they often capture the intended meaning.

Qwen also exhibits a similar pattern, with extremely low exact match and token-level F1 scores. Like LLaVA, it frequently generates full-sentence outputs even when a single-word answer is expected, which heavily penalizes it under strict lexical metrics. Nonetheless, its semantic scores remain competitive, reflecting a reasonable grasp of visual and contextual semantics despite poor format alignment.

VilBERT shows better performance than both LLaVA and Qwen on this metric. Its training on VQAv2 and architecture that aligns visual and textual streams helps it produce concise, well-targeted responses.

In summary, BLIP provides the most balanced and reliable zero-shot performance across both exact lexical and semantic metrics. VilBERT is more format-compliant than LLaVA and Qwen. LLaVA and Qwen, while semantically capable, suffer under strict evaluation due to their verbosity and lack of task-specific fine-tuning. This reinforces the importance of both domain-relevant pretraining for robust VQA performance and post-processing.

IV. LoRA for Parameter-Efficient Fine-Tuning

To efficiently fine-tune large vision-language models under computational constraints, we applied **LoRA (Low-Rank Adaptation)**. Our primary goal was to minimize the number of trainable parameters while preserving model performance. We prioritized smaller models and used additional optimization techniques like quantization and mixed precision training.

Among the models tested-*BLIP, BLIP-2, LLaVA, Qwen2.5, GIT, InstructBLIP, MiniGPT, VilBERT*-the best performing in terms of fine-tuned accuracy were BLIP, LLaVA, and Qwen2.5.

V. Fine-Tuning and Evaluation of BLIP

Base Model: `Salesforce/blip-vqa-base` with **395 million parameters**.

A. *LoRA Configuration*

```
config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    target_modules="all-linear",
    task_type="QUESTION_ANS"
)
```

For efficient fine-tuning, Low-Rank Adaptation (LoRA) was employed with carefully selected hyper-parameters. A rank of $r = 16$ was chosen to balance parameter efficiency and expressive power, enabling the model to capture sufficient task-specific information without significantly increasing the parameter count. The scaling factor $\alpha = 32$ was used to appropriately amplify the LoRA updates. A dropout rate of 0.05 was applied to the LoRA layers to introduce regularization and reduce overfitting. Bias parameters were kept frozen (`bias = "none"`) to maintain stability and prevent unnecessary updates. The adaptation was applied to all linear layers (`target_modules = "all-linear"`) to maximize the learning capacity across the model. The task type was set to `"QUESTION_ANS"` to ensure compatibility with the question-answering format of the dataset.

Parameter Efficiency:

- **Trainable Parameters:** 10,552,256
- **Total Parameters:** 395,224,828
- **Trainable %:** 2.6699%

B. *Training Setup*

The fine-tuning process followed an iterative approach using two datasets:

- **Dataset-6k:** Served as the initial dataset for model selection and early validation. It helped guide configuration tuning and checkpoint selection.
- **Dataset-20k:** Used in the subsequent stage for final fine-tuning and comprehensive evaluation based on insights from the initial stage.

We used the following dataset splits:

- 80% for training
- 7.5% for validation
- 12.5% for testing

C. Training Arguments

```
training_args = TrainingArguments(
    output_dir="./blip-vqa-lora-final-2",
    report_to="wandb",
    run_name="blip-vqa-final-finetune",
    per_device_train_batch_size=8,
    gradient_accumulation_steps=2,
    num_train_epochs=3,
    learning_rate=2e-5,
    weight_decay=0.01,
    fp16=True,
    logging_dir="./logs",
    logging_strategy="steps",
    logging_steps=1000,
    evaluation_strategy="steps",
    eval_steps=1000,
    save_strategy="steps",
    save_steps=1000,
    save_total_limit=2,
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    greater_is_better=False,
    label_names=["labels"]
)
```

D. Key Training Arguments and Justification

The training was configured with a per-device batch size of 8 and gradient accumulation steps set to 2, resulting in an effective batch size of 16. This setup balances GPU memory constraints with gradient stability.A learning rate of 2×10^{-5} was chosen in the later stages of fine-tuning (from epoch 2 onward) to ensure stable optimization; initially, a higher learning rate of 2×10^{-4} was used during epoch 1 to accelerate convergence. To regularize the model and prevent overfitting, a small weight decay of 0.01 was applied. Mixed precision training was enabled (fp16 = True) to reduce memory usage and speed up computation. The model was trained for 3 epochs, with evaluation, logging, and checkpointing occurring every 1000 steps to enable consistent monitoring. Only the two most recent checkpoints were retained (save total limit = 2) to conserve disk space. The best model was automatically restored at the end based on the lowest validation loss (load best model at end = True), using eval loss as the selection metric. Logging was integrated with Weights & Biases (report to = wandb) for better experiment tracking and visualization.

E. Optimization Techniques

We employed several optimization strategies to fit the training within hardware and time limits:

1) **8-bit Quantization:** Enabled to reduce memory usage and allow training on consumer GPUs.

$$\text{load\_in\_8bit} = \text{True} \tag{1}$$

2) **Mixed Precision Training:** Used half-precision for faster training and lower memory usage.

$$\text{fp16} = \text{True} \tag{2}$$

3) **Gradient Accumulation:** Effective batch size of 16 was achieved without memory overflow by:

$$\text{per\_device\_train\_batch\_size} = 8 \tag{3}$$

$$\text{gradient\_accumulation\_steps} = 2 \tag{4}$$

4) **Device Map Auto-Sharding:** Distributed layers automatically across two T4 GPUs.

$$\text{device\_map} = \text{"auto"} \tag{5}$$

5) **Manual Learning Rate Adjustment:** Learning rate was increased mid-training for faster convergence.

$$\text{Initial learning rate} = 2 \times 10^{-5} \tag{6}$$

$$\text{Updated learning rate (Epoch 1)} = 2 \times 10^{-4} \tag{7}$$

F. Challenges Faced

One of the primary challenges during training was the 12-hour time limit imposed by Kaggle GPU sessions. This constraint made it difficult to complete training in a single uninterrupted run, especially on the larger Dataset-20k. To overcome this, we employed checkpointing and resumed training from the latest saved checkpoint in subsequent sessions. This allowed us to accumulate training progress across multiple sessions without starting over.

G. Post-Processing

To ensure consistent and fair evaluation, we applied a simple but effective post-processing step to model predictions. Specifically, we extracted only the first alphanumeric word from each predicted answer, after converting it to lowercase and stripping any extra whitespace or punctuation. This standardization helped normalize answers such as `"Bird flying"` or `"a dolphin"` to simply `"bird"` or `"dolphin"`, making evaluation more robust to minor variations in phrasing.

H. Results

For the fine-tuned BLIP model, we evaluated its performance using a diverse set of metrics to capture various aspects of response quality. The results are summarized below:

- **Exact Match:** 0.7615
- **Token-level F1:** 0.7615
- **ROUGE-L:** 0.7884
- **BERTScore (F1):** 0.9866
- **BARTScore:** -3.1493
- **Semantic Cosine Similarity:** 0.8817

These metrics collectively indicate that the fine-tuned model performs well in both exactness and semantic similarity, with particularly strong scores in BERTScore and ROUGE-L. The slightly negative BARTScore is expected due to the scoring scale and model specifics.

I. Graphs from Weights & Biases (WandB)

The discontinuities observed at 3k and 8k steps in the graphs are due to training restarts from checkpoints.

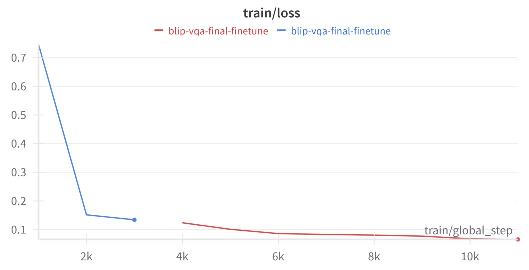

Fig. 1: Training loss over steps from WandB. The dips and fluctuations reflect learning progress and checkpoint restarts.

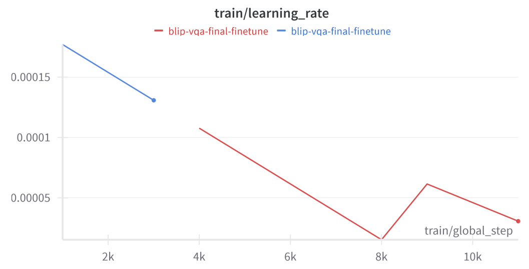

Fig. 2: Learning rate schedule over steps from WandB. A significant decay in the learning rate is observed across training.

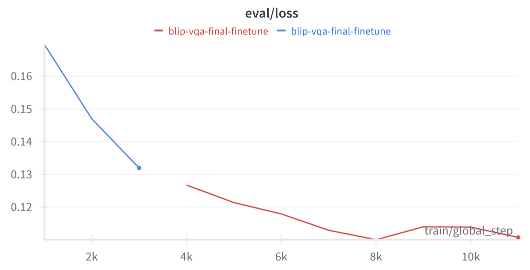

Fig. 3: Validation loss over steps from WandB. The upward trend in later stages indicates potential overfitting.

Based on the training and validation loss curves, signs of overfitting begin to appear after approximately 11,000 steps. Thus, the model checkpoint at this point is selected as the optimal stopping point for evaluation and deployment.

J. Interesting Observations

During our fine-tuning experiments, we explored different LoRA target modules. Initially, we applied LoRA only to attention-specific layers, such as `q_proj`, `k_proj`, and `v_proj`, with the expectation that adapting these components would effectively capture task-specific signal. However, the resulting improvement was minimal. Validation accuracy plateaued early and did not show consistent gains compared to baseline.

We hypothesize this is because vision-language models like BLIP rely heavily on cross-modal interactions that are distributed across multiple MLP and projection layers-not just self-attention. Restricting adaptation to attention heads may have limited the model's ability to update higher-level reasoning components.

As a result, we opted to use target_modules as "all-linear", enabling LoRA to adapt all linear layers in the model. This setting provided a wider adaptation surface and led to consistent performance gains. This conclusion aligns with recent findings that LoRA applied broadly can improve generalization in multi-modal settings without significantly increasing trainable parameters.

VI. Fine-Tuning and Evaluation of LLaVA-1.5 via Unsloth

Base Model: unsloth/llava-1.5-7b-hf-bnb-4bit with **7 billion parameters**.

A. Model Selection

LLaVA-1.5 was selected for fine-tuning not due to its raw pretrained benchmark scores, but because of its qualitative performance-producing fluent full-sentence answers-and the clear potential for improvement through task-specific fine-tuning. Despite its verbosity, we found LLaVA well-aligned with our VQA use case after instruction tuning.

The model was loaded using Unsloth's accelerated 'FastLanguageModel' API with 4-bit quantization enabled. This drastically reduced memory footprint and enabled dynamic GPU utilization on Kaggle's dual NVIDIA T4 (32 GB) setup via 'device_map="auto"'.

B. LoRA Configuration

```
lora_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    target_modules="all-linear",
    task_type="CAUSAL_LM"
)
```

We applied Low-Rank Adaptation (LoRA) to enable parameter-efficient fine-tuning. A rank of $r = 16$ balanced learning capacity with memory constraints. The scaling factor $\alpha = 32$ controlled the amplitude of the adapted updates. A dropout of 0.05 helped mitigate overfitting. Bias parameters were frozen, and LoRA was applied broadly to all linear layers using `"all-linear"` target modules, aligning with the model's transformer architecture.

Parameter Efficiency:

- **Trainable Parameters:** 47,846,400
- **Total Parameters:** 7,000,000,000
- **Trainable %:** 0.68%

C. Training Setup

An iterative training strategy was adopted:

- **Dataset-6k:** Used for early fine-tuning and validation of architecture choices.
- **Dataset-20k:** Used for final tuning. Despite limited steps, results from 6k justified scaling.

Dataset splits were:

- 80% training
- 10% validation
- 10% testing

D. Training Arguments

```
training_args = TrainingArguments(
    output_dir="/kaggle/working/" +
               "llava-fast-ft-final",
    report_to="wandb",
    run_name="llava-vqa-finetune-final",
    learning_rate=2e-4,
    per_device_train_batch_size=8,
    gradient_accumulation_steps=1,
    weight_decay=0.01,
    logging_dir=log_dir,
    logging_strategy="steps",
    logging_steps=200,
    eval_strategy="steps",
    eval_steps=200,
    save_strategy="steps",
    save_steps=200,
    save_total_limit=2,
    max_steps=1500,
    gradient_checkpointing=True,
    remove_unused_columns=False,
    fp16=True,
    bf16=False,
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    greater_is_better=False,
    label_names=["labels"]
)
```

We used a high learning rate of 2×10^{-4} to enable faster convergence within the limited training window of 1500 steps. The per-device batch size was set to 8, and with no gradient accumulation, this allowed for quicker iterations and reduced memory overhead.

Weight decay was set to 0.01 to apply L2 regularization and mitigate overfitting, especially during rapid training. Mixed precision training (fp16=True) helped lower memory usage and speed up computation, which was necessary for running a 7B model with LoRA adapters on a 32GB GPU.

To further reduce memory usage, gradient checkpointing was enabled, allowing intermediate activations to be recomputed during the backward pass instead of being stored.

Logging, evaluation, and saving were all configured to occur every 200 steps, ensuring consistent monitoring and recovery options in the case of unexpected interruptions (e.g., time limits on Kaggle sessions). We retained only the last two checkpoints (save_total_limit=2) to manage limited disk space.

We disabled removal of unused columns to ensure that all necessary inputs, including visual tokens, were preserved through the data pipeline. The training process was set to load the best-performing model checkpoint based on the lowest evaluation loss (metric_for_best_model="eval_loss", greater_is_better=False). Finally, we explicitly specified label_names=["labels"] to ensure compatibility with the trainer's label handling logic for supervised fine-tuning.

E. Optimization Techniques

1) **4-Bit Quantization:** We employed 4-bit quantization via the Unsloth framework to significantly reduce the memory footprint of the model. This allowed us to fine-tune a 7B parameter vision-language model on commodity GPUs while maintaining acceptable accuracy.

2) **Device Map Auto-Sharding:** To make use of both available NVIDIA T4 GPUs on Kaggle, we enabled automatic device mapping. This technique splits the model layers across multiple GPUs based on available memory, without requiring manual intervention. It was crucial for parallelizing the training process given the model's large size.
3) **Flash Attention and Key-Value Caching:** Unsloth internally leverages Flash Attention, an optimized attention mechanism that reduces memory usage and accelerates matrix operations during training. Additionally, the use of key-value caching improved decoding speed during both training and validation.
4) **Gradient Checkpointing:** To manage limited GPU memory, we used gradient checkpointing. This technique avoids storing all intermediate activations in memory by recomputing them during the backward pass. It enabled us to train deeper models on smaller hardware with minimal performance loss.

F. *Challenges Faced*

A significant bottleneck during training was Kaggle's 12-hour GPU session limit, which made it infeasible to complete even a single full epoch on the larger datasets. For example, Dataset-20k required over 4000 steps per epoch, yet we could only complete 1500 steps (one-third of an epoch) in a single session, consuming nearly the entire 12-hour window. This constraint directly impacted performance gains: while Dataset-6k reached a more complete pass through the data, Dataset-20k suffered from undertraining despite having richer supervision.

Training speed was another limiting factor. The large model size, combined with dual-GPU communication overhead, resulted in slow iteration throughput-averaging around 3 minutes per 10 training steps. Additionally, validation checkpoints triggered every 200 steps added up to 30 minutes of overhead per evaluation. These cumulative delays not only reduced training efficiency but also introduced considerable downtime when resuming from checkpoints.

Consequently, the limited number of optimization steps applied to Dataset-20k failed to fully leverage its increased sample size. This explains why the improvements from Dataset-6k to Dataset-20k were minimal-an undertrained model, even on a larger dataset, simply lacked enough iterations to generalize better.

G. *Post-Processing*

To improve the robustness and fairness of our evaluation metrics, we applied a normalization function to all predicted answers. This post-processing step removed common formatting artifacts and helped standardize predictions for better comparison against ground truth labels.

Specifically, we stripped common prefixes (e.g., "Answer:"), removed leading non-alphabetic characters, converted all text to lowercase, and filtered out punctuation. Only the first word of the cleaned result was retained as the final prediction. For example, answers like "A dolphin" or "dolphin!" were both normalized to "dolphin". This reduced evaluation noise and accounted for minor formatting differences in model outputs.

H. *Results*

For the fine-tuned LLaVA model, we evaluated its performance using a comprehensive set of metrics to assess both lexical accuracy and semantic alignment. The evaluation results are summarized below:

- **Exact Match:** 0.5894
- **Token-level F1:** 0.5905
- **ROUGE-L:** 0.6258
- **BERTScore (F1):** 0.9581
- **BARTScore:** -3.9560
- **Semantic Cosine Similarity:** 0.7794

These metrics indicate that the fine-tuned LLaVA model achieves strong alignment with ground truth answers, particularly in terms of semantic similarity. The high BERTScore and ROUGE-L values suggest robust language modeling and paraphrastic accuracy, while the exact match and token-level F1 scores confirm consistent surface-level correctness. The slightly negative BARTScore is expected due to the nature of the scoring scale and generation length dynamics.

I. *Graphs from Weights & Biases (WandB)*

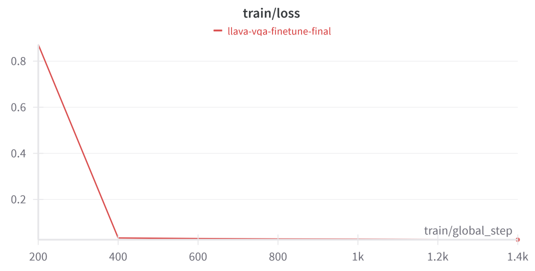

Fig. 4: Training loss over steps recorded via WandB. The dips and fluctuations reflect the model's learning dynamics.

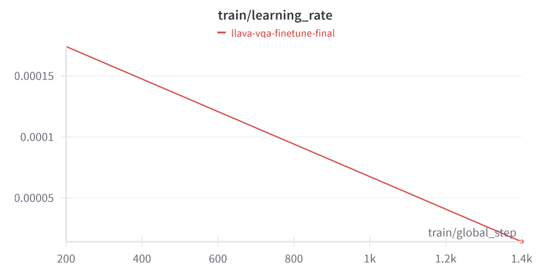

Fig. 5: Learning rate schedule over training steps. A noticeable decay trend is visible as training progresses.

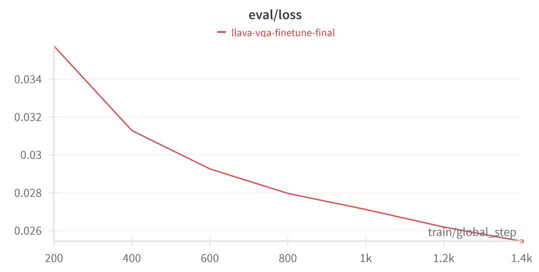

Fig. 6: Validation loss across steps. Due to limited runtime, training did not reach a stage of overfitting.

Although the training and validation loss curves indicate gradual learning, we were unable to observe overfitting within the allowed GPU time. The model exhibited slow convergence, and training sessions had to be truncated before completing even one full epoch. Consequently, we could not identify the optimal stopping point for maximum generalization performance.

J. Interesting Observations

One notable observation was that LLaVA's training and validation losses decreased significantly faster than BLIP's during the initial phases of training. Upon manual inspection, it was evident that LLaVA quickly learned to respond with concise, often one-word answers, aligning with the expected format of the dataset.

However, despite its much larger size (7 billion parameters compared to BLIP's 400 million), LLaVA did not outperform BLIP in terms of evaluation metrics. This suggests that model size alone does not guarantee better performance-especially when training time is limited and the dataset favors more structured or pattern-based answers.

VII. Fine-Tuning and Evaluation of Qwen2.5 via Transformers

Base Model: Qwen/Qwen2_5-VL-7B-Instruct with **7 billion parameters**.

A. Model Selection

Qwen2.5-VL-7B-Instruct was selected due to its promising generation quality and potential to improve through fine-tuning. While it initially generated full-sentence outputs, its closeness to the correct answer made it ideal for concise visual question answering (VQA).

Although we initially explored Unsloth, internal compatibility errors forced us to use Hugging Face Transformers directly. We leveraged the `Qwen/Qwen2.5-VL-7B-Instruct` model, dynamically allocating GPUs via `device_map="auto"` to fully utilize Kaggle's dual T4 (32 GB) GPUs. The model was loaded in 8-bit precision for memory efficiency. Fine-tuning with the smaller Qwen2.5-VL-3B-Instruct variant yielded subpar results.

B. LoRA Configuration

```
lora_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    target_modules="all-linear",
    task_type="CAUSAL_LM"
)
```

We applied Low-Rank Adaptation (LoRA) to all linear layers for expressive and flexible fine-tuning. With $r = 16$ and $\alpha = 32$, this configuration provided a balance between capacity and regularization. Dropout of 0.05 reduced overfitting risk. The 'all-linear' targeting ensured wide coverage of trainable modules across Qwen's transformer blocks.

Parameter Efficiency:

- **Trainable Parameters:** 51,824,640
- **Total Parameters:** 8,343,991,296
- **Trainable %:** 0.6211%

C. Training Setup

Training followed an iterative curriculum:

- **Dataset-6k:** Initial fine-tuning for pipeline testing and early loss reduction.
- **Dataset-20k:** Full training to maximize generalization and performance.

Dataset Splits:

- 80% training
- 10% validation
- 10% testing

D. Training Arguments

```
o_dir = /kaggle/working/
        qwen2vl-lora-kaggle-7b-final

training_args = TrainingArguments(
    output_dir=o_dir,
    report_to="wandb",
    run_name="qwen-7B-lora-final
                -finetune",
    learning_rate=2e-4,
    per_device_train_batch_size=8,
    gradient_accumulation_steps=1,
    weight_decay=0.01,
    logging_dir=log_dir,
    logging_strategy="steps",
    logging_steps=100,
    eval_strategy="steps",
    eval_steps=100,
    save_strategy="steps",
    save_steps=100,
    save_total_limit=2,
    max_steps=5000,
    gradient_checkpointing=True,
    remove_unused_columns=False,
    fp16=True,
    bf16=False,
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    greater_is_better=False,
    label_names=["labels"]
)
```

A high learning rate of 2×10^{-4} was used to accelerate convergence during fine-tuning, and mixed precision training was enabled via the fp16=True flag to reduce memory usage and speed up computation on compatible GPUs. The model was trained for a fixed upper bound of 5000 steps, as set by the max_steps parameter, which helped fit within Kaggle's time-limited environment.

Frequent evaluation and checkpointing were configured by saving, logging, and validating every 100 steps, which provided granular insights into training progress and helped prevent overfitting. The TrainingArguments also included real-time tracking via the report_to="wandb" integration, enabling robust visualization and monitoring of key metrics.

Additional configurations such as gradient_checkpointing=True allowed trading compute for memory, which was essential for fitting the 7B parameter model within dual T4 GPUs. Moreover, load_best_model_at_end=True ensured that the final model used for inference corresponded to the checkpoint with the lowest validation loss, preserving model quality.

E. Optimization Techniques

1) **8-Bit Quantization:** The model was loaded in 8-bit precision, significantly reducing memory footprint and enabling the 7B parameter model to fit within the memory limits of dual T4 GPUs available on Kaggle.
2) **Auto Device Mapping:** The device_map="auto" setting automatically partitioned the model layers across available GPUs, ensuring optimal hardware utilization without manual layer assignment.
3) **Gradient Checkpointing:** This feature was enabled to recompute intermediate activations during the backward pass, which helped reduce peak memory usage during training at the cost of additional computation.
4) **Selective LoRA Adaptation:** LoRA was applied selectively to all linear layers. This allowed effective parameter-efficient fine-tuning while keeping GPU memory usage in check.
5) **Flash Attention (Not Used):** Flash Attention, an efficient attention mechanism that speeds up training and reduces memory usage, was not used because it requires Ampere or newer GPU architectures, whereas Kaggle kernels are based on NVIDIA T4 (Volta architecture).
6) **KV Cache:** Key-value caching, which stores previously computed attention keys and values during decoding, is essential for speeding up inference in autoregressive generation.

F. Challenges Faced

The most significant challenge encountered during training was Kaggle's 12-hour runtime limit per session. To address this, we employed the resume_from_checkpoint parameter to continue training seamlessly across multiple sessions. Specifically, training on the larger Dataset-20k required nearly 24 hours for approximately 3 epochs, which had to be split across two consecutive 12-hour Kaggle sessions. This approach allowed us to persist training progress despite session interruptions.

However, despite the increased volume of training data, performance improvements from Dataset-20k were relatively modest. One major bottleneck was computational efficiency: every 10 training steps took roughly 30 seconds, while each validation phase required approximately 27 minutes. These time constraints significantly limited the total number of epochs we could feasibly train within Kaggle's resource limits.

G. Post-Processing

To reduce noise and improve evaluation consistency, predicted outputs were normalized through a lightweight post-processing pipeline. The procedure involved lower-casing the predicted string, removing all non-alphabetic characters, eliminating duplicate tokens while preserving order, and finally selecting the first valid alphabetic token. If no valid token remained, the output was replaced with `"Unknown"`. This normalization step ensured fairer comparison during evaluation by mitigating artifacts such as extra punctuation, repeated words, or empty responses.

H. Results

After fine-tuning the Qwen2.5-VL-7B-Instruct model on Dataset-20k for approximately 3 epochs, the model achieved the following performance metrics on the validation set:

- **Exact Match (EM):** 0.4720
- **Token-level F1:** 0.4720
- **ROUGE-L:** 0.5365
- **BERTScore (F1):** 0.9296
- **BARTScore:** -3.4406
- **Semantic Cosine Similarity:** 0.7947

These results indicate that while exact string matches remained under 50%, the model demonstrated strong semantic understanding, as reflected in the high BERTScore and cosine similarity metrics. The gap between token-level and semantic metrics suggests that many predictions were paraphrased or closely related to the ground truth, even when not matching exactly.

I. Graphs from Weights & Biases (WandB)

The training and validation curves appear somewhat discontinuous due to multiple resumed checkpoints. This was necessary because Kaggle imposes a 12-hour time limit per session, requiring the training process to be split and resumed across several intervals.

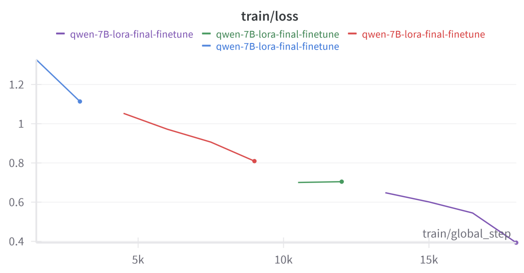

Fig. 7: Training loss curve showing a gradual downward trend across steps.

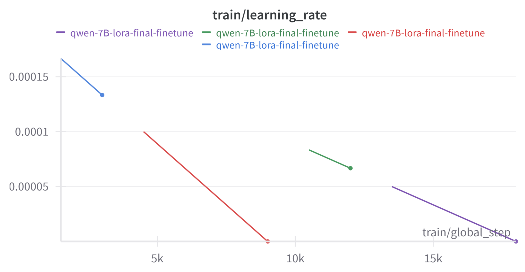

Fig. 8: Learning rate curve showing the scheduled decay over time.

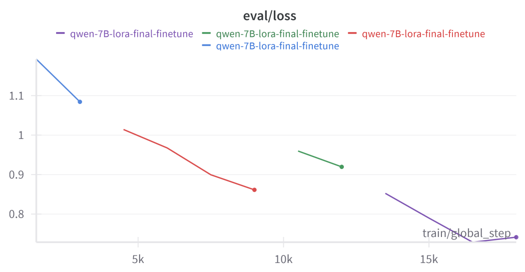

Fig. 9: Validation loss initially decreases, but later shows an upward trend.

From the training and validation loss graphs, we observe that while the model steadily improved during the early phases, the validation loss began to rise around step 18,000. This indicates the onset of overfitting, where the model continues to minimize training loss but generalizes poorly to unseen data.

J. Interesting Observations

We also experimented with the 3 billion parameter variant of the model. Interestingly, it required significantly more epochs to learn the fundamental task of predicting a single-word answer, compared to the 7 billion parameter version. This suggests that larger models may have an advantage in converging faster on tasks that require strong instruction-following behavior.

Moreover, although the 7B Qwen model was trained for more steps than both LLaVA and, to some extent, BLIP, it still did not match their performance. This highlights that model size and training duration alone do not guarantee superior results-architecture, pretraining objectives, and alignment strategies also play critical roles.

VIII. Other Models

We experimented with several alternative vision-language models, including BLIP-2, GIT, InstructBLIP, MiniGPT, and ViLBERT. However, each presented significant limitations within the constraints of our setup and the specific requirements of our task.

BLIP-2, while a well-regarded model for vision-language tasks, performed poorly on our one-word answer dataset. It achieved less than 1% Exact Match accuracy, likely because it is pre-trained for descriptive and multi-token outputs rather than short, precise answers. Its generation style did not align well with our strict evaluation format.

InstructBLIP, although designed for instruction-following tasks, was not implemented in our code due to integration complexity and time constraints. Based on observed outputs from available resources and prior studies, it tended to generate verbose, sentence-style answers instead of the concise, one-word responses our task required. This stylistic mismatch would have likely led to poor performance on strict evaluation metrics like Exact Match and token-level F1, making it an unsuitable fit for our fine-tuning goals without extensive prompt and output control.

GIT and **MiniGPT** presented practical challenges during integration. While we implemented a working codebase for GIT, we encountered limited compatibility with Hugging Face's PEFT library-particularly for applying LoRA fine-tuning. GIT's tokenizer and model structure were not designed for chat-style instruction tuning, which required extensive custom adaptation. In the case of MiniGPT, we did not proceed to code implementation, as its architecture and fine-tuning workflow were not well-aligned with Hugging Face tooling and lacked sufficient documentation for streamlined integration. Both models would have required significant engineering overhead to support our generative, instruction-based task setup within Kaggle's constrained environment.

ViLBERT showed strong quantitative performance on classification-based VQA benchmarks. However, we excluded it from training because it is not a generative model. ViLBERT treats VQA as a multi-class classification task, outputting one of several predefined labels. This design does not align with our goal of training autoregressive models capable of producing open-text answers.

We also attempted to run **Phi-3**, a lightweight model promising strong performance in text generation. Unfortunately, Phi-3 requires Ampere or newer GPU architectures to support certain operations (like Flash Attention and advanced fused kernels). Since Kaggle currently only provides T4 GPUs (which use the older Turing architecture), we were unable to run Phi-3 in this environment.

Due to these practical limitations-ranging from architectural incompatibility and toolchain issues to hardware restrictions-we ultimately focused our efforts on Qwen-VL-7B, which offered the best balance between capability, compatibility with LoRA, and alignment with our one-word generative answer format.

IX. Authors' Contributions

1) **Siddharth Reddy Maramreddy (IMT2022031)**: Curated VQA datasets by testing different prompts, filtering metadata, using different models and APIs.

2) **Margasahayam Venkatesh Chirag (IMT2022583)**: Investigated and evaluated pretrained models on the VQA dataset. Explored and implemented evaluation metrics to systematically compare the performance of pretrained versus fine-tuned models.

3) **Sai Venkata Sohith Gutta (IMT2022042)**: Led the complete implementation of parameter-efficient fine-tuning using LoRA across multiple vision-language models. Designed and executed the training pipeline, including model selection, configuration tuning, checkpointing, and optimization for resource-constrained environments.

Performance Comparison and Model Releases

Table 1. Comparison of Pretrained and Fine-tuned Models Across Evaluation Metrics

Model	Exact Match	Token-Level F1	ROUGE-L	BERTScore (F1)	BARTScore	Semantic Cosine Sim.
BLIP (pretrained)	0.4090	0.4090	0.4241	0.9719	-4.7987	0.6891
BLIP (fine-tuned)	0.7615	0.7615	0.7884	0.9866	-3.1493	0.8817
LLaVA (pretrained)	0.1144	0.1737	0.2260	0.8588	-5.9035	0.4583
LLaVA (fine-tuned)	0.5894	0.6258	0.9581	0.9296	-3.9560	0.7794
Qwen (pretrained)	0.0300	0.0300	0.0300	0.9655	-6.4838	0.3178
Qwen (fine-tuned)	0.4720	0.4720	0.5365	0.9296	-3.4406	0.7947

Table 2. Fine-tuned VQA Models Published on Hugging Face

Model Name	Hugging Face Repository
BLIP (with LoRA)	sohith18/blip-lora-vqa
LLaVA-1.5 7B (with LoRA)	sohith18/llava-lora-vqa-7b
Qwen2-VL 7B (with LoRA)	sohith18/qwen2vl-lora-vqa-7b
Qwen2-VL 3B (with LoRA)	sohith18/qwen2vl-lora-vqa-3b