Visual Recognition : Final Assignment

Subhajeet Lahiri
IMT2021022

Sai Madhavan G
IMT2021101

Aditya Garg
IMT2021545

***Abstract*—In this report, we delve into Visual Question Answering (VQA) using various image and text encoders. Specifically, we explore the performance of DINO as an image encoder, coupled with BERT as the text encoder, integrating them through a co-attention mechanism. Our investigation extends to fine-tuning these models and experimenting with the depth and architecture of the co-attention block we implemented. Additionally, we explore the efficacy of Lora fine-tuning in enhancing model performance. Our findings offer valuable insights into the interplay between different encoder architectures and co-attention mechanisms and the effectiveness of LoRA when used for fine-tuning.**

Link to the GitHub Repository.
Link to the Model Snapshots.

A) Sampling and Dataset creation

For this project, we utilized the Visual Question Answering (VQA) Dataset released by Georgia Tech, specifically focusing on the Balanced Real Images subset. Given the large size of the dataset, we were instructed to use only 25% of the training images to make the task more manageable.

Random Sampling

To achieve the 25% subset, we employed a random sampling method. This approach involves selecting images at random from the dataset, ensuring each image has an equal chance of being included. The code snippet below demonstrates our sampling process:

```python
def createRandomSample(p : float):
    sample = set()
    imgfiles = os.listdir('./../RawImages/images/')
    [1:]
    for img in imgfiles:
        if np.random.choice([0, 1], p = [1 - p, p]):
            sample.add(FiletoId(img))
    return sample
sample = createRandomSample(0.25)
```

This method was chosen due to its simplicity and effectiveness. Random sampling ensures that the subset is representative of the entire dataset, capturing a diverse range of images and questions without introducing any bias. This randomness is crucial in avoiding overfitting and ensuring that the model generalizes well to unseen data.

By leveraging this approach, we maintain the integrity and diversity of the original dataset, facilitating robust training and evaluation of our VQA model.

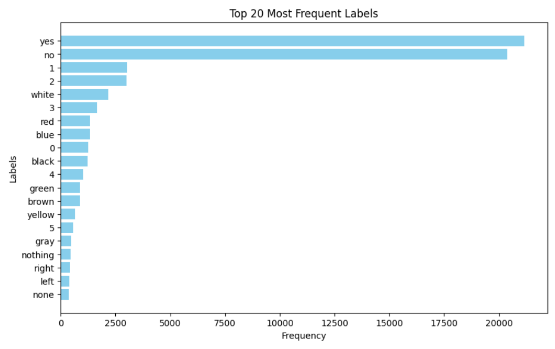
Hugging face dataset

To facilitate easy access and management of our sampled dataset, we created a Hugging Face dataset. This approach allows us to leverage the powerful data handling and machine learning tools provided by the Hugging Face ecosystem, streamlining the data pre-processing phase and enhancing our model training and evaluation process. The final dataset is structured to contain pairs of questions and images as input, with the corresponding answers as labels. Each entry in the dataset includes:

- **Image**: The image file associated with the question.
- **Question**: The text of the question related to the image.
- **Answer**: The textual answer to the question.

We have the following observations about the data :

- The answer text is generally very short and there are 9214 distinct answers in the sampled dataset with around 110k samples. This, coupled with the required reporting metric being accuracy and f1 score leads us to training a classifier on the answer space.
- Within the dataset, there are some super classes such as 'yes' and 'no', which constitute almost half of the labels. The figure below demonstrates the stated fact :



DataLoader and Collator

The DataLoader and collator play crucial roles in the preprocessing pipeline, ensuring that raw samples from the dataset are transformed into a format suitable for model training and evaluation.

Collator: The collator is responsible for applying the necessary **non-learnable pre-processing** steps to each batch of samples. The key steps involved in this process are:

1) **Image Processing**:
   - The images are processed using `AutoImageProcessor.from_pretrained(DINO)`. This step involves resizing, normalizing, and possibly augmenting the raw image data to match the input requirements of the DINO model. The processor returns the pixel values as tensors, ready for model consumption.
2) **Question Tokenization**:
   - The questions are tokenized using `AutoTokenizer.from_pretrained(BERT)`. This step converts the raw text questions into a sequence of token IDs that the model can understand. The tokenizer handles padding, truncation, and attention mask creation to ensure that all tokenized questions in a batch have the same length, facilitating efficient processing by the model.
3) **Label Encoding**:
   - The answers, which are in textual form, are transformed into numerical labels using a `LabelEncoder`. The encoder has been pre-fitted to the entire answer space, mapping each unique answer to a corresponding integer label.

DataLoader: The DataLoader acts as a wrapper around the dataset, managing the batching and indexing of samples. It operates in two modes: training and validation. In training mode, the DataLoader uses the first 100,000 samples from the dataset and in validation, it uses the last 9000.
The DataLoader has methods that make it compatible with the `enumerate` function, allowing for easy iteration over batches of data.

B) Training Environment

The models for the Visual Question Answering (VQA) task were trained in a standardized environment to ensure consistency and reproducibility across experiments. The training environment, unless specified otherwise, is as follows:

- **Platform**: Kaggle
- **Accelerator**: Nvidia Tesla P100
- **RAM**: 16 GB
- **Library**: PyTorch

The VQA problem is treated as a classification task, where the goal is to predict the correct answer given an image and a corresponding question. The models are trained using the `torch.nn.CrossEntropyLoss` criterion, which computes the cross-entropy loss between the predicted and target labels.

For optimization, the Adam optimizer from `torch.optim` is utilized. The initial learning rate for the optimizer is typically set in the range of $1e^{-3}$ to $1e^{-4}$, and it is decayed by a factor ranging from 0.1 to 0.7 every 1 to 2 epochs. This decay strategy helps in fine-tuning the learning rate during training and improving convergence.

While reporting the performance metrics, `sklearn` library's built-in functions are utilized. Additionally, the training time is recorded using the clock embedded in the Kaggle notebook environment.

The standardized training environment ensures consistency and comparability of results across different experiments and models.

Note : Since the models are put together using DINO and BERT, there are several instances in the report where we refer to them by the name Dinobert.

C) Base Model - DINO + Bert with a single co-attention layer

We employed a base model for our initial experiments consisting of a pre-trained image encoder and a pre-trained text encoder. Specifically, we used **facebook/dinov2-base** for the image encoder and **FacebookAI/roberta-base** for the text encoder. The model architecture is demonstrated by Fig. 1.

Encoders

The ‘facebook/dinov2-base‘ model is based on the Vision Transformer (ViT) architecture, which processes images by dividing them into patches and using transformer layers to capture spatial relationships and features. The ‘FacebookAI/roberta-base‘ model is a variant of the BERT architecture designed for robust text understanding, which uses transformer layers to capture the contextual relationships between words in a sentence.

Combining Encoder Outputs

To combine the outputs of these models, we implemented a co-attention mechanism using ‘nn.MultiheadedAttention‘ with a single attention head. This allows the model to focus on relevant parts of both the image and the text simultaneously. The process is as follows:

- We extract (Sequence length, Embedding dimension) shaped vectors from both the text and the image encoders. The sequence length for the image encoder is 257 (An image is worth 16 * 16 words) and that of the text encoder depends on the length of the question and is typically around 8 - 10. The embedding dimension is 768 for both the encoders.

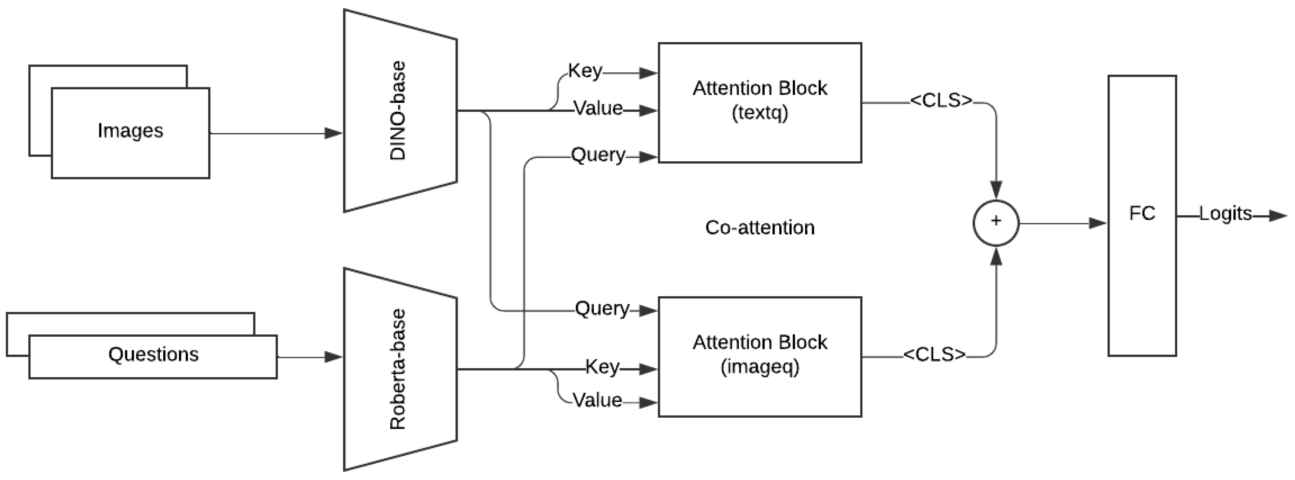


Fig. 1. Dinobert base model architecture

- We apply co-attention between the image and the text encoders. Specifically, values and keys from the text encoder are used to answer queries from the image encoder and vice-versa.
- We combine the two co-attention outputs by extracting the attended [CLS] token vectors, and then adding them to get a single [CLS] token vector of dimension (batch_size, 768). It now contains information from both the image and the text.
- We pass this vector through a fully connected (FC) layer to obtain a 9214-dimensional vector representing the logits.

Training and Performance

The model was trained for 5 epochs, and not further as the loss plateaued. Despite/due to the unsophisticated architecture, the base model achieved an accuracy of 18.61%. The performance metrics are summarized in the table below:

Metrics	Values
Accuracy (Validation)	18.61%
Accuracy (Training)	19.33%
Precision (Validation)	0.07
Recall (Validation)	0.19
f1-score (Validation)	0.09
Time taken per epoch	1 hr 07 min
Total time taken	5 hr 33 min

TABLE I
PERFORMANCE METRICS OF THE BASE MODEL

A significant issue observed was that the model predominantly predicted only 4 classes: ”yes”, ”no”, ”snowboarding” and ”skiing” - two of them being one of ”super” classes pointed out above.

Label	Predictions
Yes	1548
No	7755
Snowboarding	178
Skiing	4

TABLE II
DISTRIBUTION OF PREDICTIONS FOR DIFFERENT CLASSES

D) MODEL OPTIMIZATION 1: LIGHTER MODEL WITHOUT CO-ATTENTION

We introduced a few optimizations to our model to address the disparity between the number of samples and the model size. The model architecture is demonstrated in Fig. 2.

Model Size Reduction: Firstly, we down-scaled the pretrained encoders from their base variants to their smaller counterparts. Specifically, we replaced the original models with `smallbenchnlp/roberta-small` for BERT and `facebook/dinov2-small` for ViT. The rationale behind this step is that the original models had over 400 million parameters, whereas our dataset consisted of only 100k samples. By reducing the model size to 135 million parameters, we aimed to better match the model complexity with the available data.

Removal of Co-Attention Mechanism: Secondly, we experimented with removing the co-attention mechanism. Instead, we opted for a simpler approach of concatenating the output of the encoders and passing them through fully connected (FC) layers directly. This decision was based on the hypothesis that the complexity introduced by the co-attention mechanism may not be necessary for our task, and a simpler architecture could suffice. But it was necessitated by the fact that the embedding dimension of the text encoder and the image encoder are not the same - thereby making co-attention unfeasible without another linear transform.

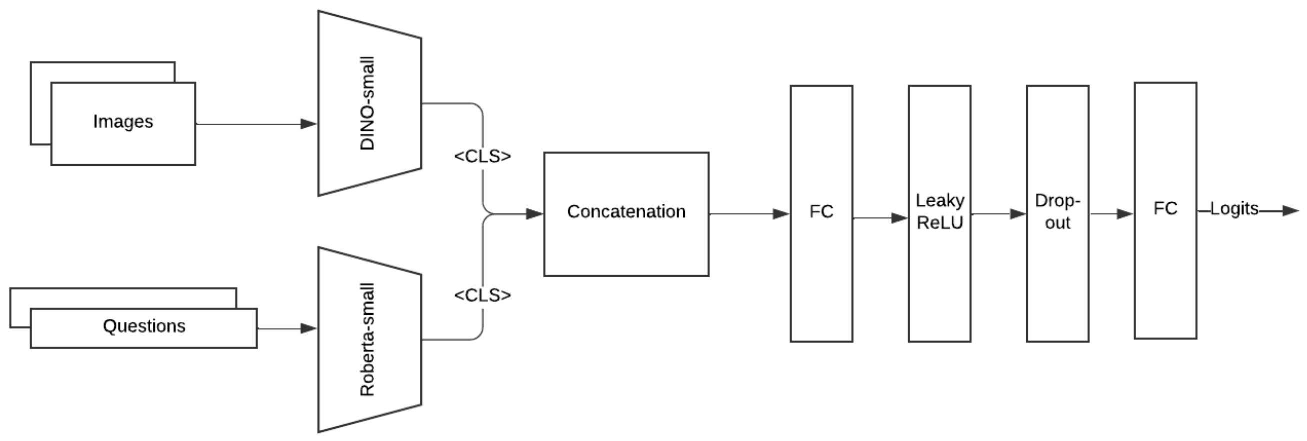


Fig. 2. Dinobert small variant architecture

Stronger Classifier: Instead of having just a single FC layer, the classifier now has a Leaky ReLU non-linearity along with a dropout layer sandwiched between two linear layers.

Model	Embedding dimension
roberta-small	256
dino-small	384

TABLE III
EMBEDDING DIMENSIONS FOR SMALL MODEL VARIANTS

Training Details and Performance

We trained the optimized model for 10 epochs with an initial learning rate of 5e-4. We decayed the learning rate by a factor of 0.2 every 2 epochs. The decrease in model complexity led to a significant reduction in training time per epoch, from 1 hour 7 minutes to 27 minutes.

The resulting model exhibited the following metrics:

Metrics	Values
Accuracy (Validation)	19.14%
Accuracy (Training)	20.33%
Precision (Validation)	0.07
Recall (Validation)	0.19
f1-score (Validation)	0.10
Time taken per epoch	27min
Total time taken	4 hr 31 min

TABLE IV
PERFORMANCE METRICS OF THE OPTIMIZED MODEL

Observations

- Despite the optimizations, the model did not achieve a significant improvement in accuracy. This can largely be attributed to the model's increasing bias towards the "super" classes ("yes" and "no") as training progressed. Initially, the untrained model exhibited predictions across over 1,000 classes on the validation data. However, towards the end of training, it predominantly focused on predicting the "yes" and the "no" classes.
- Accuracy scores of around 21% can be obtained if the model just learns that "yes" is the most frequent class and predicts it for every sample.

Label	Predictions
Yes	6714
No	2771

TABLE V
DISTRIBUTION OF PREDICTIONS FOR "YES" AND "NO" CLASSES

E) MODEL OPTIMIZATION 2: BROADER MODEL WITH SKIP CONNECTIONS

After analyzing the issues faced by the smaller model, we incorporated several changes to develop an advanced version of Dinobert. The architecture is depicted in Fig. 3. The modifications are as follows:

Broader Model

In both the base and the small variants, the model becomes too narrow when only the `<CLS>` tokens are added together, creating a bottleneck. To address this, we added width to the model:

- Instead of using a single attention head, we now use 8 attention heads in parallel.
- The `<CLS>` tokens from the encoders are concatenated instead of added, and the resulting vector is passed directly to the fully connected (FC) layers.

Skip Connections

To provide a gradient highway to the encoders and allow the attention heads to better model unity, we introduced a skip connection:

- The encoder `<CLS>` token is added to the `<CLS>` token corresponding to the output of the attention heads. This

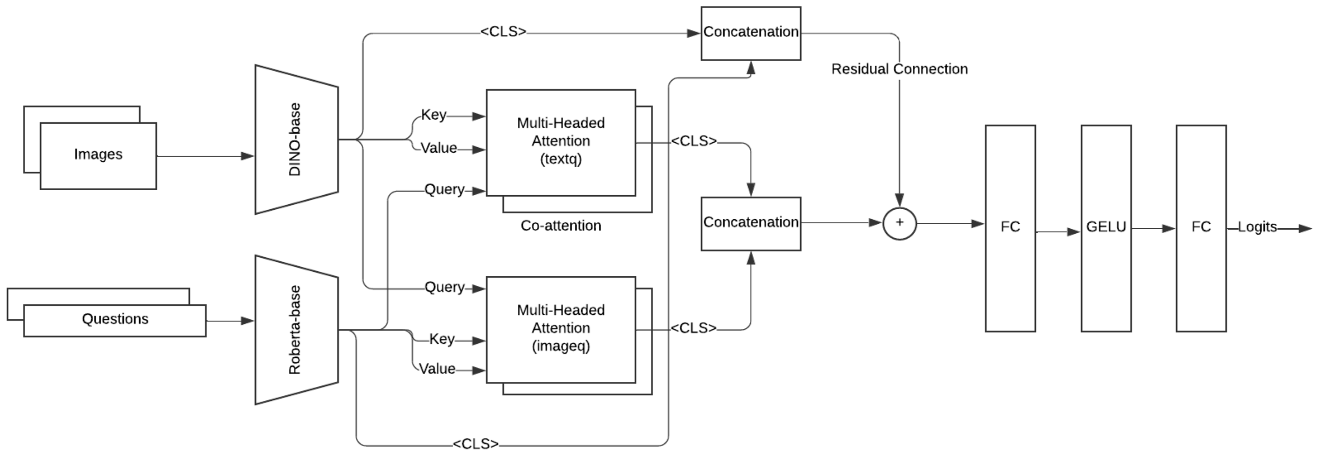


Fig. 3. Dinobert advanced variant architecture

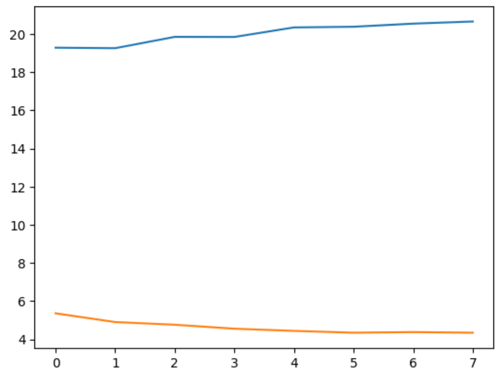


Fig. 4. Accuracy, loss curve - X-axis: epochs, Y-axis: the blue line denotes percentage accuracy while the yellow line denotes loss

skip connection helps maintain gradient flow and enhances the model's ability to capture important features.

Training Details and Performance

The advanced DINO-BERT model was trained for 8 epochs using the same standardized environment and optimization techniques as described previously. Fig. 4 shows the accuracy-loss curve while training. The resulting model exhibited the following metrics:

Metrics	Values
Accuracy (Validation)	20.19%
Accuracy (Training)	20.65%
Precision (Validation)	0.09
Recall (Validation)	0.20
f1-score (Validation)	0.11
Time taken per epoch	1 hr 17 min
Total time taken	10 hr 15 min

TABLE VI
PERFORMANCE METRICS OF DINOBERT ADVANCED

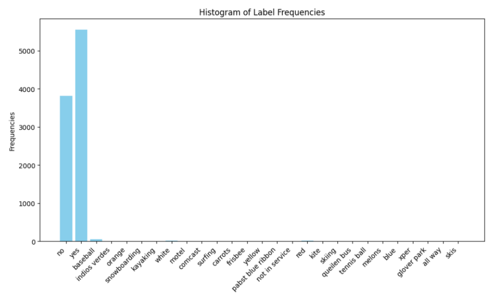


Fig. 5. Predicted class label distribution

Observations

- This model definitely performs better than the previous ones with improved metrics throughout. We will therefore be restricting ourselves to this one for further experiments.
- The output also represents the data better than other models as it doesn't limit itself to one-two classes (Fig. 5).

F) Training Optimization 1: Cross entropy with weights

To address the extreme class imbalance in the dataset, we introduced a training optimization by assigning weights to the classes in the cross-entropy loss function. These weights were set to be inversely proportional to the class frequencies, calculated as follows:

$$\text{class weight} = \frac{\text{total number of samples}}{\text{class frequency}}$$

This weighting strategy was implemented to ensure that the model paid more attention to the less frequent classes during training. We trained models similar to both the Dinobert small and Dinobert advanced versions using this modified criterion. The models were trained under the same conditions as previously described, with the addition of class weights in the cross-entropy loss function. Despite this adjustment, the results were abysmal, with the accuracy scores being close to zero. The introduction of class weights may have caused the model to overcompensate for the less frequent classes, leading to instability during training.

G) Training Optimization 2: LoRA + Data Augmentation

In this section, we experimented with two training optimizations: data augmentation and Low-Rank Adaptation (LoRA) fine-tuning.

Data Augmentation

To enhance the diversity of the training dataset, we applied a series of transformations to the images. This resulted in a new dataset with double the number of samples as the original. The question and answer pairs were kept the same, while the images underwent the following transformations:

```
    augpipe = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.ColorJitter(brightness=0.2,
    contrast=0.2, saturation=0.2, hue=0.2),
        transforms.RandomResizedCrop(size=(224, 224)
    , scale=(0.8, 1.0)),
        transforms.ToTensor()
    ])
```

These transformations are designed to improve the model's generalization by introducing variability in the training images, thereby helping the model to learn more robust features.

Low-Rank Adaptation (LoRA)

We also applied Low-Rank Adaptation (LoRA) to the `dinobert_adv` model. Using a utility script, we identified all linear layers and included them in the `target_modules` for LoRA fine-tuning.

```
    target_modules = [
        "dense", "word_embeddings", "query", "key",
    "value",
        "position_embeddings", "out_proj"
    ]
```

This approach significantly reduced the number of trainable parameters from 100% to 1.46%, resulting in a trainable model size of 4.74 million parameters. This reduction is more suitable given the size of our training sample.

Training and Performance Metrics

We trained the model using the above optimizations for 4 epochs and evaluated its performance. The training and evaluation were conducted under the same conditions as previously described.

Metrics	Values
Accuracy (Validation)	20.50%
Accuracy (Training)	20.85%
Precision (Validation)	0.08
Recall (Validation)	0.21
f1-score (Validation)	0.11
Time taken per epoch	52 min
Total time taken	3 hr 28 min

TABLE VII
PERFORMANCE METRICS OF DINOBERT ADVANCED WITH LORA

Observations

1) The predicted speed-up during training using LoRA was very clearly apparent. The time per epoch was reduced to 52 min from 77 min earlier.
2) We still could not see a marked improvement in the model's performance.

H) Training Optimization 3: LoRA with frozen weights

In the following experiments, we tried to understand what would be the behaviour if we froze the weights of both the encoders and trained the classifier layer alone, as a precursor to the LoRA optimization. The architecture for this experiment is identical to that given in Fig 2, except that the classifier is changed such that there is a singular fully connected layer instead of two.

The motivation for this experiment was that experiments with LoRA, with a randomly initialized fully connected classifier was not performing satisfactorily. So, we hypothesized that training the classifier layer for a bit before we apply LoRA on the encoders could actually aid the LoRA optimization later. We chose to freeze the weights of the encoders as a measure for optimization.

Once the final layer converged, i.e., its performance plateaued, we applied LoRA-based fine-tuning on the key and value matrices of all self-attention blocks in both the encoders. This resulted in the number of trainable parameters of the model at a mere 12.57% of all the parameters of the model. In spite of this reduction in number of parameters and while using the smaller model variant, we obtained the best performance in this model out of all our experiments.

Training Details and Performance: We document the results in the tables below. Please note that the hardware used in the below experiments is different and inferior to the hardware used in other experiments.

Metrics	Values
Accuracy (Validation)	21.59%
Accuracy (Training)	20.32%
Precision (Validation)	0.47
Recall (Validation)	0.22
f1-score (Validation)	0.22
Time taken per epoch	16 min
Total time taken	2 hr 39 min

TABLE VIII
PERFORMANCE METRICS OF DINOBERT SMALL FROZEN EXPERIMENT BEFORE LORA

Metrics	**Values**
Accuracy (Validation)	35.46%
Accuracy (Training)	33.31%
Precision (Validation)	0.54
Recall (Validation)	0.35
f1-score (Validation)	0.36
Time taken per epoch	24 min
Total time taken	3 hr 54 min

TABLE IX
PERFORMANCE METRICS OF DINOBERT SMALL FROZEN EXPERIMENT AFTER LORA

J) CONCLUSION

Using the combination of separate classifier training and LoRA-based fine-tuning on the encoder, we achieved a massive jump in accuracy - from 20.4% on the largest variant to 35.46% on the smallest variant - while navigating the challenges of an imbalanced dataset, incorrect gradient directions due to small batch sizes and an environment which constrained the number of experiments that could be performed.