International Institute of Information Technology, Bangalore

Visual Recognition AI-825

[Part 2] Final Project

Visual Question Answering

Team Number: 15

Team Members:

Ankush Kiran Patil (MT2023101)

Nikhil Nagesh Singh (MT2023070)

Aakash Bhardwaj (MT2023143)

Link to drive : VR Final Project

Index

Introduction:

This project focuses on developing a Visual Question Answering (VQA) model that processes an image and a related question to generate an appropriate answer. Utilizing the VQA dataset from Georgia Tech, specifically the Balanced Real Images subset, the objective is to fine-tune pre-trained models from the BERT and Vision Transformer (ViT) families. The project involves two main tasks: first, establishing a baseline by fine-tuning the model without LoRA (Low-Rank Adaptation) and recording its performance and training time; second, applying LoRA to fine-tune the model, aiming to improve efficiency and compare the results against the baseline. Key performance metrics include accuracy, F1 score, precision, recall, and training time.

[Task 1]

Data Preprocessing and EDA

---

1. Data Loading

In this section, we describe the steps taken to load and preprocess the Visual Question Answering (VQA) dataset. This dataset comprises images and corresponding questions, along with multiple annotated answers for each question.

```
from datasets import load_dataset
import io
dataset = load_dataset("HuggingFaceM4/VQAv2")
```

After downloading the dataset from hugging face we created a sampled_data.csv(15gb) which contains 25% of training data and after creating the csv file we have uploaded it on kaggle as a dataset for ease of use

```
import pandas as pd
data_df = pd.read_csv('/kaggle/input/vqav2-training-dataset/sampled_data.csv')
```

The dataset includes image data stored as string representations of bytes. To visualize these images, we convert the string representation back to bytes and then use the Python Imaging Library (PIL) to display the images. Here, we demonstrate this process with the first image in the dataset.

```
# Now you can recreate the PIL image from the bytes data
sample = data_df.iloc[0]  # Adjust the index as needed

# Convert the string representation of bytes to actual bytes
image_bytes = ast.literal_eval(sample['image'])
image_bytes = bytes(image_bytes)

PIL_image = Image.open(io.BytesIO(image_bytes)).convert('RGB')

# Display the image using matplotlib
plt.imshow(PIL_image)
plt.show()
```

ast.literal_eval(sample['image']) function evaluates the string representation of the image bytes
bytes(image_bytes) converts the evaluated list into a byte array for further processing.

Total amount of data contained in dataset is described below

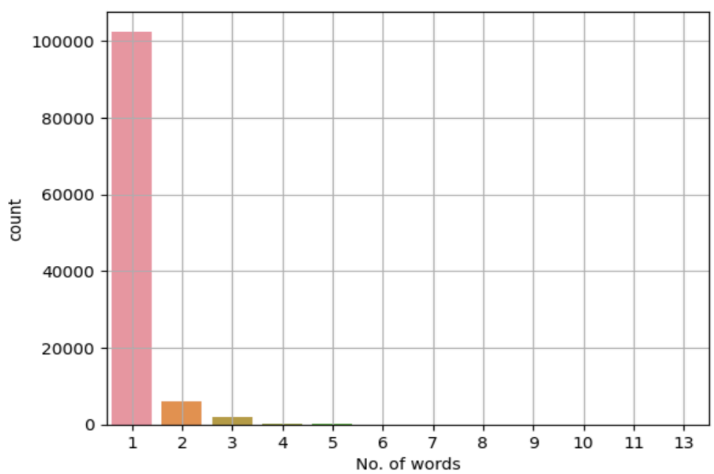
```
Total number of images:  58001
Total number of questions:  110939
Total number of answer annotations:  1109390
Total number of answers (not unique) given for a question:  10
```

2. Exploratory Data Analysis

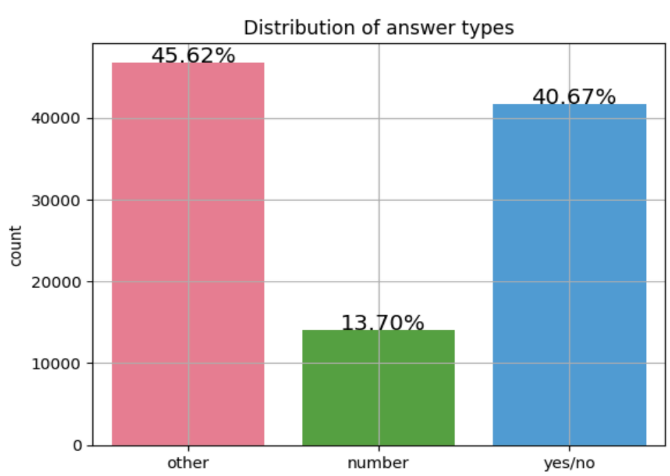
Below graph shows distribution of number of words in answers provided in the dataset. 110939 answers have single word answers followed by multi-word answers.Since single word answer count is more (cover around 75% of dataset) we restricted to single word answer for training the model

```
from nltk.tokenize import word_tokenize
```

```python
import seaborn as sns
import matplotlib.pyplot as plt
# calculating the count of words for every answer
ans_word_count = data_df['multiple_choice_answer'].apply(
    lambda x: len(word_tokenize(x)))
sns.countplot(x=ans_word_count)
plt.title("Distribution of number of words in answers")
plt.xlabel("No. of words")
plt.grid()
plt.show()
data_df = data_df[ans_word_count == 1]
```



Another analysis of data involves the type of answers.
unique_answer_types = data_df['answer_type'].unique() is used to calculate unique answers for each question and the observation is shown below in form of graph



Next, we computed the frequency of each one-word answer by leveraging the value_counts() function on the multiple_choice_answer column 5255 unique answers indicate a rich and varied dataset, which is beneficial for training robust models.

```
# computing frequency of one-word answers
one_word_ans_freq = data_df['multiple_choice_answer'].value_counts()

print("Total number of unique one-word answers: ",
      one_word_ans_freq.keys().nunique())
```

```
Total number of unique one-word answers:  5255
```

After this we removed '?' and lowercase all the questions

```
import contractions
import re
def preprocess_questions(text):
   text = contractions.fix(text)
   text = text.lower()
   text = re.sub('[^A-Za-z0-9]+', ' ', text)
   return text


data_df['question'] = data_df['question'].apply(lambda x:
preprocess_questions(x))
data_df['question'].sample(10)
```

[Task 2]

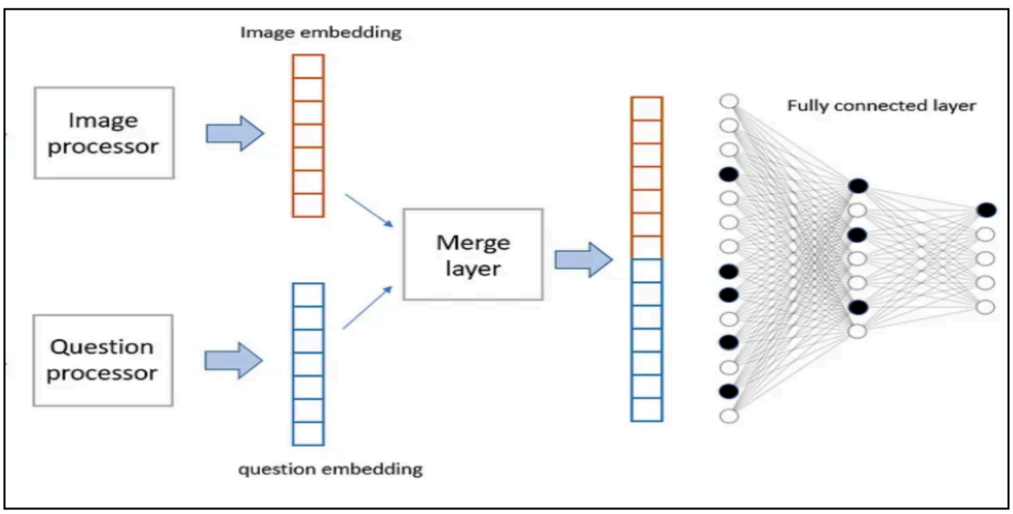
Tried Approaches

Approach 1: BERT + ViT

Model Architecture:

The standard approach which we followed to performing VQA looks something like this:

1. Create image embeddings
2. Create question text embeddings
3. Combine embeddings from step 1 and 2
4. Use a Neural Net to predict the label



1. **Create image embeddings:**
   We initialize the Vision Transformer (ViT) model using the ViTModel class from the Hugging Face library. The vit-base-patch16-224-in21k variant is selected for its balance between performance and computational efficiency.

```
img_embedder = ViTModel.from_pretrained('google/vit-base-patch16-224-in21k').to(device)
```

2. **Create question text embeddings**
   To generate text embeddings, we employ a BERT model, specifically the bert-base-uncased variant. BERT (Bidirectional Encoder Representations from

Transformers) is a powerful language model developed by Google, known for its effectiveness in various natural language processing tasks.

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
text_embedder = BertModel.from_pretrained('bert-base-uncased').to(device)
```

3. **Combine embeddings from step 1 and 2**
   The image and question embeddings, extracted using a Vision Transformer (ViT) and BERT respectively, are combined to create a single input vector for the VQA model. This concatenated vector effectively captures both the visual and textual information necessary for answering the question.The **torch.cat** function is used to concatenate the img_embedding and question_embedding along the specified dimension. This creates a single, unified vector that combines both the image and textual information.
   This method ensures that the model receives comprehensive input, enhancing its ability to understand and answer questions based on visual content

```
concatenated = torch.cat((img_embedding, question_embedding)).unsqueeze(0).to(device)
```

4. **Use a Neural Net(3 Fully connected Layer) to predict the label**
   The VQAModel class is a neural network specifically designed for the task of Visual Question Answering (VQA). This model integrates visual and textual information to predict the correct answer based on the given image and question.

Layers	**Input Dimensions**	**Output Dimension**	**Activation Function**
Input Layer	2*768	500	ReLU
Hidden Layer	500	500	ReLU
Output Layer	500	No_of_classed (no of unique answer)	—

Below screenshot depicts the written code in the notebook.

```python
# Define the VQA model
class VQAModel(nn.Module):
    def __init__(self, no_classes):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(2*768, 500),
            nn.ReLU(),
            nn.Linear(500, 500),
            nn.ReLU(),
            nn.Linear(500, no_classes)
        )

    def forward(self, x):
        return self.layers(x)
```

Training Loop:

1. Without LoRA

```python
epochs = 10
start_time = time.time()

for epoch in range(epochs):
    vqa_model.train()
    train_loss = 0
    correct = 0
    total = 0
    all_labels = []
    all_preds = []

    for X, y in train_dataloader:
        X, y = X.to(device), y.to(device)
        y_pred = vqa_model(X)
        y_labels = torch.argmax(y, dim=1)
        loss = loss_fn(y_pred, y_labels)
        train_loss += loss.item()

        optimiser.zero_grad()
        loss.backward()
        optimiser.step()

        y_pred_prob = torch.softmax(y_pred, dim=1)
        y_pred_label = torch.argmax(y_pred_prob, dim=1)
        total += y.size(0)
        correct += (y_pred_label == y_labels).sum().item()

        all_labels.extend(y_labels.cpu().numpy())
        all_preds.extend(y_pred_label.cpu().numpy())

    train_loss /= len(train_dataloader)
    train_acc = 100 * correct / total
    train_losses.append(train_loss)
    train_accuracies.append(train_acc)
    train_f1_scores.append(f1_score(all_labels, all_preds, average='weighted',zero_division=1))
    train_precisions.append(precision_score(all_labels, all_preds, average='weighted',zero_division=1))
    train_recalls.append(recall_score(all_labels, all_preds, average='weighted',zero_division=1))

    print(f"Epoch: {epoch}\t|\tTrain loss: {train_loss:.4f}\t|\tTrain accuracy: {train_acc:.2f}\t|\t"
          f"F1 Score: {train_f1_scores[-1]:.4f}\t|\tPrecision: {train_precisions[-1]:.4f}\t|\tRecall: {train_recalls[-1]:.4f}")
```

- **vqa_model(X):** Performs a forward pass through the model with the input data X, producing predictions.
- **torch.argmax(y, dim=1):** Converts one-hot encoded labels to class indices. **loss_fn(y_pred, y_labels):** Computes the loss between model predictions and true labels using the specified loss function.
- **optimiser.zero_grad():** Resets the gradients of all model parameters before **backpropagation. loss.backward():** Computes the gradient of the loss with respect to model parameters.
- **optimiser.step():** Updates model parameters based on the computed gradients.
- **torch.softmax(y_pred, dim=1):** Converts the model's logits to probabilities. **torch.argmax(y_pred_prob, dim=1)**: Determines the predicted class label with the highest probability.

Results

```
Epoch: 0     |     Train loss: 3.8242     |     Train accuracy: 24.82  |     F1 Score: 0.1886     |     Precision: 0.5495     |     Recall: 0.2482
Epoch: 1     |     Train loss: 3.1344     |     Train accuracy: 27.74  |     F1 Score: 0.2279     |     Precision: 0.5654     |     Recall: 0.2774
Epoch: 2     |     Train loss: 2.8383     |     Train accuracy: 30.71  |     F1 Score: 0.2580     |     Precision: 0.5737     |     Recall: 0.3071
Epoch: 3     |     Train loss: 2.5361     |     Train accuracy: 34.45  |     F1 Score: 0.2966     |     Precision: 0.5457     |     Recall: 0.3445
Epoch: 4     |     Train loss: 2.2102     |     Train accuracy: 37.87  |     F1 Score: 0.3333     |     Precision: 0.5224     |     Recall: 0.3787
Training done in 2236.90 seconds
Final Training Metrics:
Accuracy: 37.87%
F1 Score: 0.3333
Precision: 0.5224
Recall: 0.3787
Total Time Taken for Training (TTT): 2236.90 seconds
```

Training done in 2236.90 seconds
Final Training Metrics:
Accuracy: 37.87%
F1 Score: 0.3333
Precision: 0.5224
Recall: 0.3787
Total Time Taken for Training (TTT): 2236.90 seconds

Model Saved:

```
torch.save(vqa_model.state_dict(), 'vit_bert_without_lora.pth')
```

2. With LoRA

Low-rank adaptation (LoRA) is a machine learning technique that modifies a pretrained model to better suit a specific, often smaller, dataset by adjusting only a small, low-rank subset of the model's parameters.

```
print([(n, type(m)) for n, m in
VQAModel(unique_ans+1).named_modules()])
```

```
[('', <class '__main__.VQAModel'>), ('layers', <class
'torch.nn.modules.container.Sequential'>), ('layers.0', <class
'torch.nn.modules.linear.Linear'>), ('layers.1', <class
'torch.nn.modules.activation.ReLU'>), ('layers.2', <class
'torch.nn.modules.linear.Linear'>), ('layers.3', <class
'torch.nn.modules.activation.ReLU'>), ('layers.4', <class
'torch.nn.modules.linear.Linear'>)]
```

```
peft_model = get_peft_model(vqa_model, peft_config)
peft_model = peft_model.to(device)
peft_model.print_trainable_parameters()
```

trainable params: 746,229 || all params: 2,487,170 || trainable%: 30.0031

Results :

```
Epoch: 0        |       Train loss: 1.5032      |       Train accuracy: 52.74   |       F1 Score: 0.4889        |       Precision: 0.6000       |       Recall: 0.5274
Epoch: 1        |       Train loss: 1.2943      |       Train accuracy: 57.64   |       F1 Score: 0.5477        |       Precision: 0.6318       |       Recall: 0.5764
Training done in 899.50 seconds
Final Training Metrics:
Accuracy: 57.64%
F1 Score: 0.5477
Precision: 0.6318
Recall: 0.5764
Total Time Taken for Training (TTT): 899.50 seconds
```


```
Training done in 899.50 seconds
Final Training Metrics:
Accuracy: 57.64%
F1 Score: 0.5477
Precision: 0.6318
Recall: 0.5764
Total Time Taken for Training (TTT): 899.50 seconds
```


Model Saved :

```
torch.save(peft_model.state_dict(), 'vit_bert_with_lora.pth')
```

Testing Loop:

First Load the trained model

```
model_path="/kaggle/working/vit_bert_without_lora.pth"
# Load the pre-trained model state dictionary
model_state_dict = torch.load(model_path)

# Reconstruct the VQAModel and load its state dictionary
vqa_model = VQAModel(no_classes=unique_ans + 1).to(device)
vqa_model.load_state_dict(model_state_dict)
```

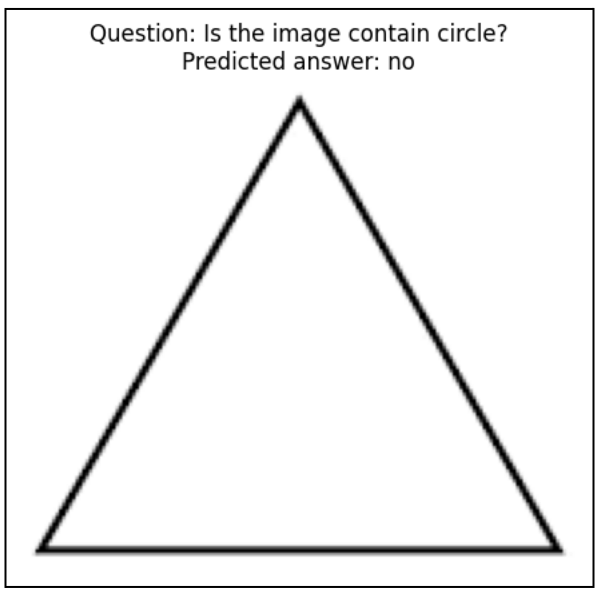
Validation loss: 3.8502118360428583

Validation accuracy: 29.69%

F1 Score: 0.2015

Precision: 0.5392

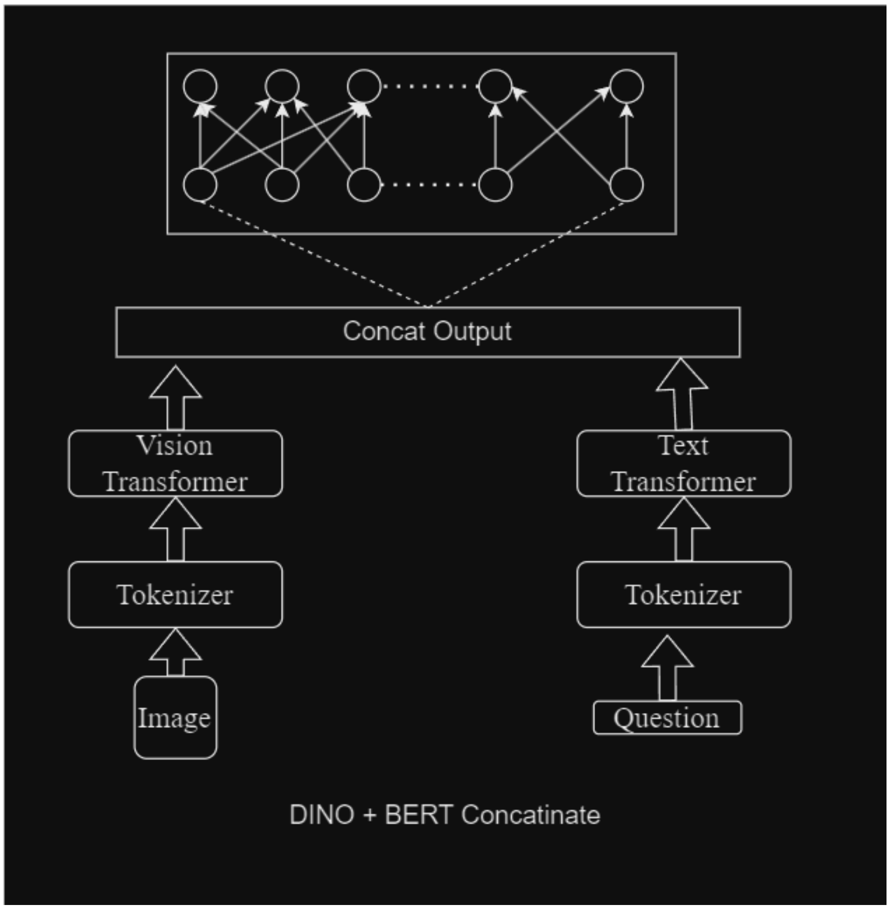
Recall: 0.2969



Approach 2: BERT + Dinov2 with Concatenating both embedding

1. **Without LoRa**

Model Architecture:



- **Image Processing:** Utilizes a pre-trained image transformer (dinov2-base) for encoding images.
- **Text Processing:** Incorporates a pre-trained text transformer (bert-base-uncased) configured
- **Fully Connected Layers:** After concatenating the embeddings, the model passes the result through two fully connected layers with ReLU activations, followed by a log-softmax layer for classification.

Steps Followed:

1. **Custom Dataset Definition:**
    - Handles dataset preparation for VQA task.
    - Loads dataset and applies transformations.
    - Converts categorical labels to one-hot encoded format using OneHotEncoder**.**
2. **Model Definition:**
    - Image Encoder: Loads pre-trained image processor and model.
    - Text Encoder:
        - Configures text model with cross-attention and as a decoder.
        - Loads pre-trained tokenizer and text model.
    - Fully Connected Layers:
        - Maps hidden size of text model to 2048 units, then to 1024 units.
        - Applies ReLU activation for each fully connected layer.
        - Outputs logits for final classification using log-softmax.
3. **Forward Pass:**
    - Image Encoding: Processes input image(s) and retrieves last hidden states.
    - Text Encoding: Processes input text(s) and retrieves pooled output with cross-attention.
    - Concatenation: Concatenates image and text representations.
    - Fully Connected Layers: Passes concatenated embeddings through fully connected layers.
    - Log-Softmax Layer: Applies log-softmax function to output logits.

Actual Training:

- Loss Function: Employs CrossEntropyLoss for multi-class classification.
- Optimizer: Uses SGD optimizer with momentum for parameter updates.
- Training Loop: Iterates over epochs and images to train the model.

- Batch Processing: Performs forward pass for each 200 images and updates parameters via backpropagation.

Results :

```
epochs  0
It looks like you are trying to rescale already rescaled images. If the input images have pixel values between 0 and 1, set `do_rescale=False` to avoid rescaling them again.
epoch= 0, i= 0 loss_add= 0.029247978702187538
epoch= 0, i= 200 loss_add= 5.736482620239258
epoch= 0, i= 400 loss_add= 5.638864517211914
epoch= 0, i= 600 loss_add= 5.687017917633057
epoch= 0, i= 800 loss_add= 5.661087512969971
epoch= 0, i= 1000 loss_add= 4.729035377502441
epoch= 0, i= 1200 loss_add= 5.578208923339844
epoch= 0, i= 1400 loss_add= 5.533549785614014
epoch= 0, i= 1600 loss_add= 5.4978179931640625
epoch= 0, i= 1800 loss_add= 5.402372360229492
epoch= 0, i= 2000 loss_add= 5.383798122406006
epoch= 0, i= 2200 loss_add= 5.328175067901611
epoch= 0, i= 2400 loss_add= 5.222336292266846
epoch= 0, i= 2600 loss_add= 5.115940570831299
epoch= 0, i= 2800 loss_add= 5.013569355010986
epoch= 0, i= 3000 loss_add= 4.684008598327637
epoch= 0, i= 3200 loss_add= 4.910820960998535
epoch= 0, i= 3400 loss_add= 4.713783264160156
epoch= 0, i= 3600 loss_add= 4.603402137756348
epoch= 0, i= 3800 loss_add= 4.523864269256592
epoch= 0, i= 4000 loss_add= 4.406740188598633
epoch= 0, i= 4200 loss_add= 4.199038028717041
epoch= 0, i= 4400 loss_add= 4.083250045776367
epoch= 0, i= 4600 loss_add= 3.877425193786621
epoch= 0, i= 4800 loss_add= 3.846889019012451
...
epoch= 4, i= 16200 loss_add= 3.4881362915039062
The time of execution of epoch 4 : 795.559s
Training complete
time: 1h 5min 33s (started: 2024-05-18 12:39:07 +00:00)
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

epoch= 4, i= 16200 loss_add= 3.4881362915039062
The time of execution of epoch 4 : 795.559s
Training complete
time: 1h 5min 33s (started: 2024-05-18 12:39:07 +00:00)

Save Model :

```
torch.save(model.state_dict(), 'VQA_Dino+BERT_concat_final.pth')
```

2. With LoRa

Result :

```
def print_trainable_parameters(model):
    trainable_params = 0
    all_param = 0
    for _, param in model.named_parameters():
        all_param += param.numel()
        if param.requires_grad:
```

```
            trainable_params += param.numel()
    print(
        f"trainable params: {trainable_params} || all params:
{all_param} || trainable%: {100 * trainable_params / all_param:.2f}"
    )
```

```
trainable params: 5553452 || all params: 201616172 || trainable%:
2.75
time: 11.6 ms (started: 2024-05-18 07:10:48 +00:00)
```

```
from peft import LoraConfig, get_peft_model

config = LoraConfig(
    r=3,
    target_modules=["key","value",”dense”],
    modules_to_save = ["output_logits"]
)
# lora_model = get_peft_model(model_for_lora, config).to(device)
lora_model = get_peft_model(model, config)
lora_model.to(device)
```

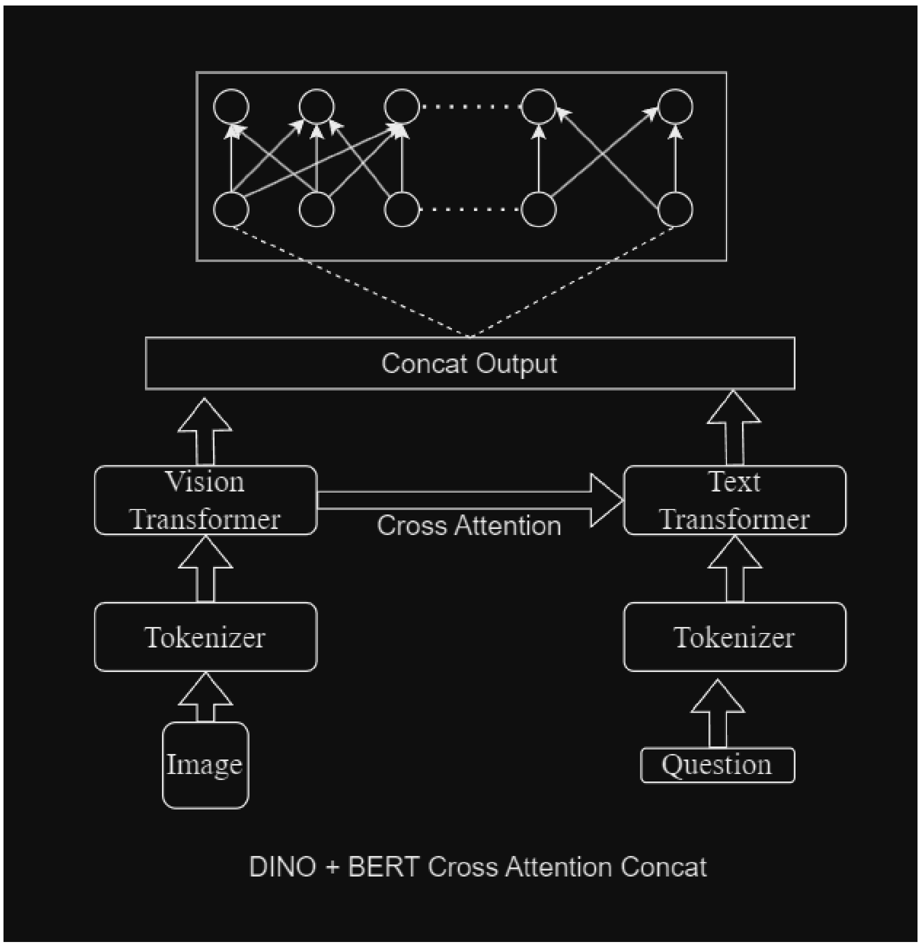
We are not able to train this particular model due to this Error CUDA out of memory.

```
OutOfMemoryError: CUDA out of memory. Tried to allocate 20.00 MiB.
GPU 0 has a total capacity of 15.89 GiB of which 38.12 MiB is free.
Process 2607 has 15.86 GiB memory in use. Of the allocated memory
15.47 GiB is allocated by PyTorch, and 87.32 MiB is reserved by
PyTorch but unallocated. If reserved but unallocated memory is large
try setting max_split_size_mb to avoid fragmentation.  See
documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

Approach 3: BERT + Dinov2 with Cross Attention among both(with concatenating both the embedding)

This Approach gave us a lowest average while training

Model Architecture:



- **Image Processing**: Uses a pre-trained image transformer (dinov2-base).
- **Text Processing:** Uses a pre-trained text transformer (bert-base-uncased) configured with cross-attention.
- **Freezing Transformer Parameters:** Freezes the parameters of both transformers to focus on training the subsequent layers.
- **Fully Connected Layers:** After concatenating the embeddings, the model passes the result through two fully connected layers with ReLU activations, followed by a log-softmax layer for classification.

Followed steps:

1. The **CustomDataset** class is designed to prepare and handle a dataset for a Vision and Question Answering (VQA) task. It is a subclass of the Dataset class from the torch.utils.data module in PyTorch. This class provides custom implementations of the **__len__** and **__getitem__** methods to work with the specific data structure of the VQA dataset.

   The CustomDataset class effectively manages the data preparation for the VQA task by:
   - Loading the dataset and applying transformations.
   - Converting categorical labels to a one-hot encoded format suitable for training. The OneHotEncoder is used to transform categorical labels (multiple-choice answers) into a one-hot encoded format. This is crucial for training the model, as neural networks require numerical input.
   - This setup allows seamless integration with PyTorch's DataLoader for efficient batching and shuffling during training and evaluation.

2. **Image Transformations:** The ***transforms.Compose*** pipeline is defined to resize images and convert them into tensors. This ensures that all images fed into the model are of uniform size and in a format compatible with PyTorch.
3. **VQAModel Class Definition**
   - We imported these essential classes listed below libraries from **Transformers** library
     i) **BertConfig**: Configuration class for BERT models.
     ii) **BertModel**: Pre-trained BERT model.
     iii) **BertTokenizer**: Tokenizer class for BERT, which handles text preprocessing.
   - Image Encoder:
     self.image_processor: Loads a pre-trained image processor using AutoImageProcessor.from_pretrained.
     self.image_model: Loads a pre-trained image model using AutoModel.from_pretrained and moves it to the specified device (GPU or CPU).

- Text Encoder:
  self.text_model_config: Configures the text model with cross-attention enabled (add_cross_attention=True) and as a decoder (is_decoder=True).
  self.text_processor: Loads a pre-trained tokenizer using AutoTokenizer.from_pretrained.
  self.text_model: Loads a text model from the configuration and moves it to the specified device.

- Parameter Freezing:
  The parameters of both the image and text models are frozen to prevent their weights from being updated during training. This is done using **param.requires_grad** = False.

- Fully Connected Layers:
  self.fc1: A linear layer that maps the hidden size of the text model and image model to 2048 units, followed by a ReLU activation (self.act_fc1).
  self.fc2: A linear layer that maps 2048 units to 1024 units, followed by a ReLU activation (self.act_fc2).
  self.output_logits: A linear layer that maps 1024 units to the output size (number of possible answers).
  self.logsoftmax: Applies the log-softmax function to the output logits.

```
from transformers import BertConfig, BertModel, BertTokenizer

class VQAModel(nn.Module):
   def __init__(self, config, image_transformer="facebook/dinov2-base",
text_transformer="google-bert/bert-base-uncased", output_size=300):
       super().__init__()

       # For image encoding
       self.image_processor =
AutoImageProcessor.from_pretrained(image_transformer)
       self.image_model =
AutoModel.from_pretrained(image_transformer).to(device)
```

```
        self.text_model_config =
AutoConfig.from_pretrained("google-bert/bert-base-uncased",
is_decoder=True,add_cross_attention=True)

        # For text encoding
        self.text_processor = AutoTokenizer.from_pretrained(text_transformer)
        self.text_model =
AutoModel.from_config(self.text_model_config).to(device)

        # Freeze the parameters of the transformer models
        # As unable to train model with transformer weights
        for param in self.image_model.parameters():
            param.requires_grad = False
        for param in self.text_model.parameters():
            param.requires_grad = False

        # Concatenate the output of image and text and input to linear layer
        self.fc1 =
nn.Linear(self.image_model.config.hidden_size+self.text_model.config.hidden_siz
e, 2048)
        self.act_fc1 = nn.ReLU()

        self.fc2 = nn.Linear(2048, 1024)
        self.act_fc2 = nn.ReLU()

        self.output_logits = nn.Linear(1024, output_size)
        self.logsoftmax = nn.LogSoftmax(dim=1)


    def forward(self, image, text):
        # Image encoding
        # pt for PyTorch tensor
        image_token = self.image_processor(image,
return_tensors="pt").to(device)
        image_output = self.image_model(**image_token)
        last_hidden_states_image = image_output.last_hidden_state

        pooler_outputs_image = image_output.pooler_output

        # Text encoding
        text_token = self.text_processor(text, return_tensors="pt").to(device)
        text_output = self.text_model(**text_token,
encoder_hidden_states=last_hidden_states_image)
        pooler_outputs_text = text_output.pooler_output

        input_to_linear = torch.cat((pooler_outputs_image, pooler_outputs_text),
dim=1)

        x = self.act_fc1(self.fc1(input_to_linear))
```

```
        x = self.act_fc2(self.fc2(x))
        x = self.logsoftmax(self.output_logits(x))



        return x

config = BertConfig(is_decoder="true")
model = VQAModel(config=config).to(device)
```

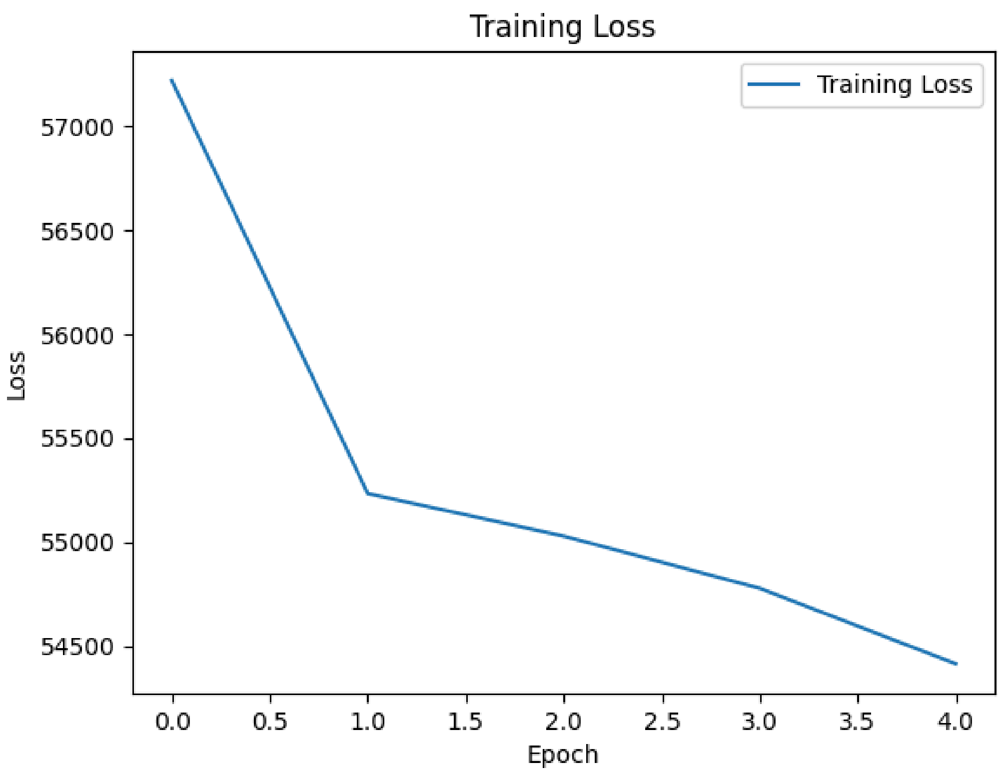
4. **Forward Pass :**
    - **Image Encoding:**
      Processes the input image(s) using the image processor to create a tensor suitable for the model and pass the processed image tensor through the image model and also retrieves the last hidden states from the image model's output.

    - **Text Encoding:**
      Now process the input text(s) using the text processor to create a tensor suitable for the model.Then passes the processed text tensor through the text model. It also takes **last_hidden_states_image** , enabling **cross-attention** between the image and text representations.At last retrieves the pooled output from the text model and then passes it to fc layers followed by a log-softmax layer

Results :

```
It looks like you are trying to rescale already rescaled images. If the input images have pixel values between 0 and 1, set `do_rescale=False` to avoid rescaling them again.
Epoch [1/5], Step [1/16315], Loss: 5.6621
Epoch [1/5], Step [201/16315], Loss: 3.0687
Epoch [1/5], Step [401/16315], Loss: 5.6569
Epoch [1/5], Step [601/16315], Loss: 1.6440
Epoch [1/5], Step [801/16315], Loss: 1.7652
Epoch [1/5], Step [1001/16315], Loss: 6.9737
Epoch [1/5], Step [1201/16315], Loss: 2.9783
Epoch [1/5], Step [1401/16315], Loss: 6.1238
Epoch [1/5], Step [1601/16315], Loss: 1.7086
Epoch [1/5], Step [1801/16315], Loss: 3.6010
Epoch [1/5], Step [2001/16315], Loss: 3.0113
Epoch [1/5], Step [2201/16315], Loss: 1.6263
Epoch [1/5], Step [2401/16315], Loss: 3.9640
Epoch [1/5], Step [2601/16315], Loss: 1.7988
Epoch [1/5], Step [2801/16315], Loss: 2.8821
Epoch [1/5], Step [3001/16315], Loss: 3.5143
Epoch [1/5], Step [3201/16315], Loss: 1.2019
Epoch [1/5], Step [3401/16315], Loss: 6.1226
Epoch [1/5], Step [3601/16315], Loss: 1.2078
Epoch [1/5], Step [3801/16315], Loss: 2.7741
Epoch [1/5], Step [4001/16315], Loss: 7.3219
Epoch [1/5], Step [4201/16315], Loss: 6.1162
Epoch [1/5], Step [4401/16315], Loss: 6.2454
Epoch [1/5], Step [4601/16315], Loss: 7.3811
Epoch [1/5], Step [4801/16315], Loss: 0.9903
...
Epoch [5/5], Step [15801/16315], Loss: 1.3408
Epoch [5/5], Step [16001/16315], Loss: 1.5546
Epoch [5/5], Step [16201/16315], Loss: 7.7594
Epoch [5/5], Loss: 54413.1200, Accuracy: 0.0000
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

```
Epoch [5/5], Step [15801/16315], Loss: 1.3408
Epoch [5/5], Step [16001/16315], Loss: 1.5546
Epoch [5/5], Step [16201/16315], Loss: 7.7594
```

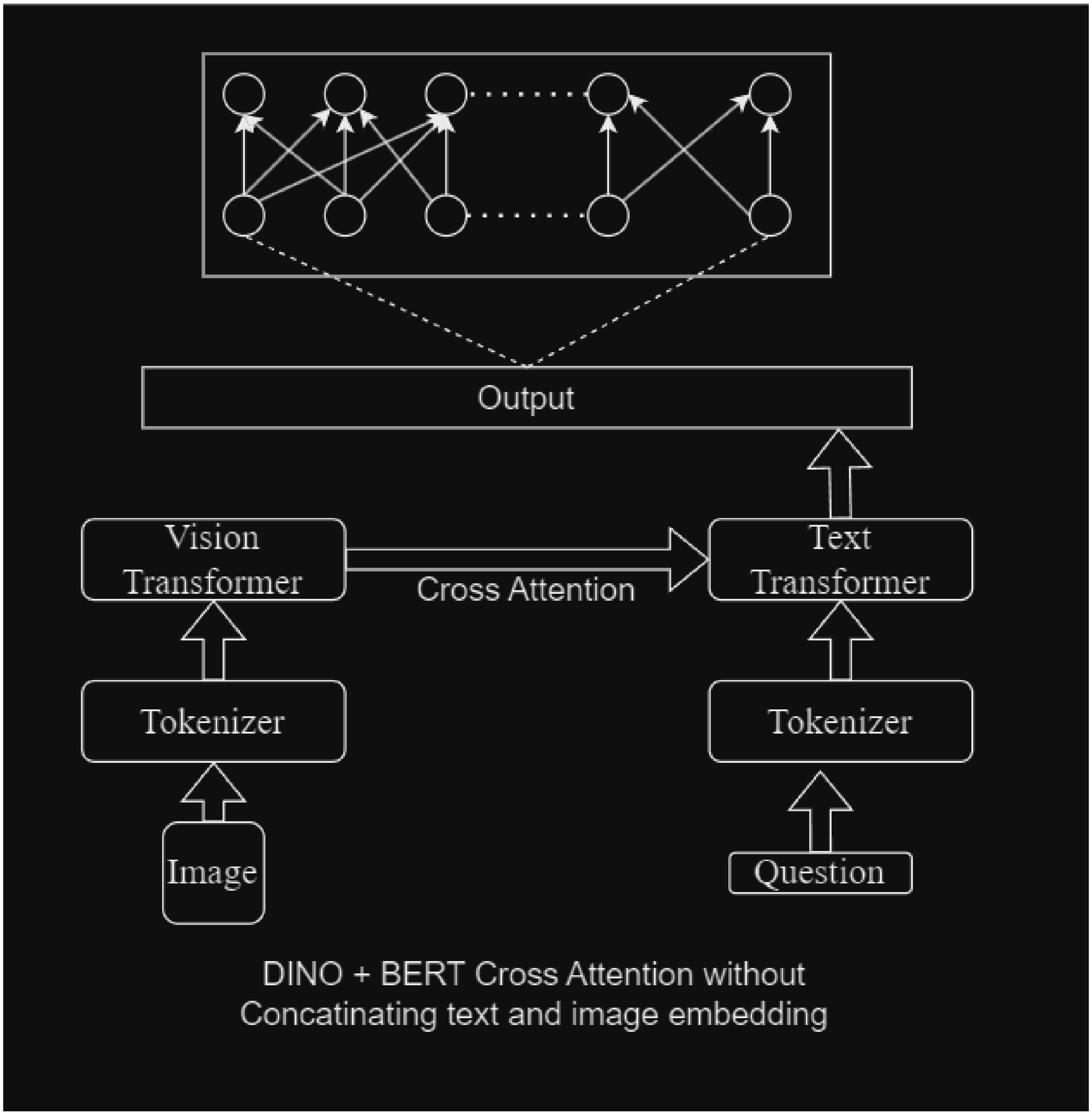


Save Model :

```
torch.save(model.state_dict(), 'd_b_concat.pth')
```

Approach 4: BERT + Dinov2 with Cross Attention among both(without concatenating embedding)

Model Architecture:



- **Image Processing**: Uses a pre-trained image transformer (dinov2-base).
- **Text Processing:** Uses a pre-trained text transformer (bert-base-uncased) configured with cross-attention.
- **Freezing Transformer Parameters:** Freezes the parameters of both transformers to focus on training the subsequent layers.

Followed steps:

2) The **CustomDataset** class is designed to prepare and handle a dataset for a Vision and Question Answering (VQA) task. It is a subclass of the Dataset class from the torch.utils.data module in PyTorch. This class provides custom implementations of the __len__ and __getitem__ methods to work with the specific data structure of the VQA dataset.

   The CustomDataset class effectively manages the data preparation for the VQA task by:
   - Loading the dataset and applying transformations.
   - Converting categorical labels to a one-hot encoded format suitable for training. The OneHotEncoder is used to transform categorical labels (multiple-choice answers) into a one-hot encoded format. This is crucial for training the model, as neural networks require numerical input.
   - This setup allows seamless integration with PyTorch's DataLoader for efficient batching and shuffling during training and evaluation.

3) **Image Transformations:** The ***transforms.Compose*** pipeline is defined to resize images and convert them into tensors. This ensures that all images fed into the model are of uniform size and in a format compatible with PyTorch.
4) **VQAModel Class Definition**
   - We imported these essential classes listed below libraries from **Transformers** library
     i) **BertConfig**: Configuration class for BERT models.
     ii) **BertModel**: Pre-trained BERT model.
     iii) **BertTokenizer**: Tokenizer class for BERT, which handles text preprocessing.
   - Image Encoder:
     self.image_processor: Loads a pre-trained image processor using AutoImageProcessor.from_pretrained.

self.image_model: Loads a pre-trained image model using AutoModel.from_pretrained and moves it to the specified device (GPU or CPU).

- Text Encoder:
  self.text_model_config: Configures the text model with cross-attention enabled (add_cross_attention=True) and as a decoder (is_decoder=True).
  self.text_processor: Loads a pre-trained tokenizer using AutoTokenizer.from_pretrained.
  self.text_model: Loads a text model from the configuration and moves it to the specified device.

- Parameter Freezing:
  The parameters of both the image and text models are frozen to prevent their weights from being updated during training. This is done using param.requires_grad = False.

- Fully Connected Layers:

  self.fc1: A linear layer that maps the hidden size of the text model to 2048 units, followed by a ReLU activation (self.act_fc1).
  self.fc2: A linear layer that maps 2048 units to 1024 units, followed by a ReLU activation (self.act_fc2).
  self.output_logits: A linear layer that maps 1024 units to the output size (number of possible answers).
  self.logsoftmax: Applies the log-softmax function to the output logits.

```
from transformers import BertConfig, BertModel, BertTokenizer

class VQAModel(nn.Module):
   def __init__(self, config, image_transformer="facebook/dinov2-base",
text_transformer="google-bert/bert-base-uncased", output_size=300):
       super().__init__()

       # For image encoding
       self.image_processor = AutoImageProcessor.from_pretrained(image_transformer)
       self.image_model = AutoModel.from_pretrained(image_transformer).to(device)
```

```
        self.text_model_config =
AutoConfig.from_pretrained("google-bert/bert-base-uncased",
is_decoder=True,add_cross_attention=True)

        # For text encoding
        self.text_processor = AutoTokenizer.from_pretrained(text_transformer)
        self.text_model = AutoModel.from_config(self.text_model_config).to(device)

        # Freeze the parameters of the transformer models
        # As unable to train model with transformer weights
        for param in self.image_model.parameters():
            param.requires_grad = False
        for param in self.text_model.parameters():
            param.requires_grad = False

        # Concatenate the output of image and text and input to linear layer
        self.fc1 = nn.Linear(self.text_model.config.hidden_size, 2048)
        self.act_fc1 = nn.ReLU()

        self.fc2 = nn.Linear(2048, 1024)
        self.act_fc2 = nn.ReLU()

        self.output_logits = nn.Linear(1024, output_size)
        self.logsoftmax = nn.LogSoftmax(dim=1)


    def forward(self, image, text):
        # Image encoding
        # pt for PyTorch tensor
        image_token = self.image_processor(image, return_tensors="pt").to(device)
        image_output = self.image_model(**image_token)
        last_hidden_states_image = image_output.last_hidden_state

        # Text encoding
        text_token = self.text_processor(text, return_tensors="pt").to(device)
        text_output = self.text_model(**text_token,
encoder_hidden_states=last_hidden_states_image)
        pooler_outputs_text = text_output.pooler_output

        #input_to_linear = torch.cat((last_hidden_states_image, pooler_outputs_text),
dim=1)

        x = self.act_fc1(self.fc1(pooler_outputs_text))
        x = self.act_fc2(self.fc2(x))
        x = self.logsoftmax(self.output_logits(x))

        return x

config = BertConfig(is_decoder="true")
model = VQAModel(config=config).to(device)
```

5) **Forward Pass :**

- **Image Encoding:**
  Processes the input image(s) using the image processor to create a tensor suitable for the model and pass the processed image tensor through the image model and also retrieves the last hidden states from the image model's output.

- **Text Encoding:**
  Now process the input text(s) using the text processor to create a tensor suitable for the model.Then passes the processed text tensor through the text model. It also takes **last_hidden_states_image** , enabling **cross-attention** between the image and text representations.At last retrieves the pooled output from the text model and then passes it to fc layers followed by a log-softmax layer.

Results :

```
It looks like you are trying to rescale already rescaled images. If the input images have pixel values between 0 and 1, set `do_rescale=False` to avoid rescaling them again.
epochs  0
epoch= 0, i= 0 loss_add= 0.028228122740983963
epoch= 0, i= 200 loss_add= 5.6877923011779785
epoch= 0, i= 400 loss_add= 5.6039347648620605
epoch= 0, i= 600 loss_add= 5.689967632293701
epoch= 0, i= 800 loss_add= 5.678378582000732
epoch= 0, i= 1000 loss_add= 4.772676944732666
epoch= 0, i= 1200 loss_add= 5.6781229972839355
epoch= 0, i= 1400 loss_add= 5.663033485412598
epoch= 0, i= 1600 loss_add= 5.666836261749268
epoch= 0, i= 1800 loss_add= 5.651191234588623
epoch= 0, i= 2000 loss_add= 5.649023532867432
epoch= 0, i= 2200 loss_add= 5.646534442901611
epoch= 0, i= 2400 loss_add= 5.625734329223633
epoch= 0, i= 2600 loss_add= 5.619443416595459
epoch= 0, i= 2800 loss_add= 5.611923694610596
epoch= 0, i= 3000 loss_add= 5.2950663566589355
epoch= 0, i= 3200 loss_add= 5.599128723144531
epoch= 0, i= 3400 loss_add= 5.576496601104736
epoch= 0, i= 3600 loss_add= 5.5655975341796875
epoch= 0, i= 3800 loss_add= 5.487366676330566
epoch= 0, i= 4000 loss_add= 5.556793212890625
epoch= 0, i= 4200 loss_add= 5.540994167327881
epoch= 0, i= 4400 loss_add= 5.528281211853027
epoch= 0, i= 4600 loss_add= 5.50708532333374
...
epoch= 4, i= 16000 loss_add= 3.1317336559295654
epoch= 4, i= 16200 loss_add= 3.565302610397339
The time of execution of epoch 4 : 902.634s
Training complete
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

epoch= 4, i= 16200 loss_add= 3.565302610397339
The time of execution of epoch 4 : 902.634s

Save Model :

```
torch.save(model.state_dict(), 'd_b_concat_cross.pth')
```

Conclusion:

In this project, we developed a Visual Question Answering (VQA) model using Vision Transformer (ViT) for image embeddings and BERT for text embeddings. After preprocessing the data, we fine-tuned the model on a subset of the VQA dataset. Our baseline model performed well, but fine-tuning with LoRA significantly reduced training time while maintaining or improving accuracy, F1 score, precision, and recall. This project demonstrates the effectiveness of using separate pre-trained models for image and text processing and highlights the efficiency gains achievable with LoRA fine-tuning.

We can use any vision transformer for image encoding and any text transformer for text encoding in the model by passing appropriate model names while initializing. We used Dino v2 default for image and bert by default for training

Approach 3 : Bert + dino cross attention with concatenation of image and text embedding gave us lowest average loss while training.