Configuration Manual

MSc Research Project
Data Analytics

Nirmal Keecheril George Mathew
Student ID: x22245863

School of Computing
National College of Ireland

Supervisor: Qurrat Ul Ain

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Nirmal Keecheril George Mathew
Student ID:	x22245863
Programme:	Data Analytics
Year:	2023 - 2024
Module:	MSc Research Project
Supervisor:	Qurrat Ul Ain
Submission Due Date:	12/08/2024
Project Title:	Configuration Manual
Word Count:	654
Page Count:	10

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Nirmal Keecheril George Mathew
Date:	15th September 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Nirmal Keecheril George Mathew
x22245863

1 Introduction

Medical Visual Question Answering (mVQA) is a modern and promising area of the clever interface of artificial intelligence, computer vision, and NLP technologies in the healthcare field for enhancing diagnostics' accuracy and patient outcomes (Huang et al.; 2023). This paper discusses the adoption of the BLIP framework in mVQA tasks with the emphasis on the efficiency of the framework when it deals with noisy medical data and provides accurate, easily-interpreted diagnostic results. The performance analysis is made on BLIP with respect to different medical imaging modalities, with a basic understanding that it performs well more than the VLP type models. Hence, through the adoption of BLIP, the study aims at improving the accuracy of mVQA systems while aiming at having explainability of model features which is an important aspect in clinical practice since clinicians require to understand the decision-making process of AI to have confidence in the recommendations. During this, the research seeks to enhance the effectiveness of mVQA systems and hence positively influence the healthcare delivery systems.

This configuration manual gives the acts of how to set up, how to train and the methods of evaluating the BLIP model for Medical Visual Question Answering (mVQA) on PathVQA dataset. The guide covers two cases: evaluating the proposed method on the full PathVQA dataset and on a subset of the dataset.

2 System Requirements

To run this project, the following hardware and software requirements are necessary:

2.1 Hardware

- **Google Colab Pro** (or an equivalent environment with sufficient resources)
- **GPU**: L4 GPU (Total RAM: 53.0 GB)
- **System RAM**: 33.7 GB (required during processing)
- **Disk Space**: 137.9 GB (with a total of 201.2 GB available)

2.2 Software

- **Operating System**: Tested on macOS, but should be compatible with other systems that support Python.

- **Python Version**: 3.7 or higher
- **Required Libraries**:
  - `torch`
  - `transformers`
  - `datasets`
  - `PIL`
  - `matplotlib`
  - `pandas`
  - `numpy`
  - `tqdm`
  - `scikit-learn`

2.3 Additional Pre-trained Models

- **BLIP VQA Base Model**: `Salesforce/blip-vqa-base`, available from the Hugging Face model hub (**?**).

3 Code Components

3.1 Initial Setup and Imports

Begin by importing all the required libraries:

```
import requests
from PIL import Image
import torch
from transformers import BlipProcessor, BlipForQuestionAnswering,BlipImageProcessor
from transformers import AutoProcessor, DataCollatorWithPadding, AdamW, get_scheduler
from transformers import BlipConfig
from datasets import load_dataset
from torch.utils.data import DataLoader
from tqdm.notebook import tqdm
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display
from torch.utils.data import Dataset, DataLoader
```

3.2 Device Configuration

Determine whether to use a GPU or CPU for training:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

3.3 Data Loading

Load the PathVQA dataset:

```
dataset = load_dataset("flaviagiammarino/path-vqa")
```

3.4 Exploratory Data Analysis

Convert the dataset splits into pandas DataFrames for easier analysis and manipulation. This step involves transforming the raw dataset into a tabular format, enabling efficient data exploration and preparation.:

```
# Convert the dataset splits to pandas DataFrames
train_df = pd.DataFrame(dataset['train'])
```

Display basic information and statistics about the dataset:

```
# Display basic information
print("Train DataFrame Info:")
print(train_df.info())
```

```
# Summarize questions and answers length
train_df['question_length'] = train_df['question'].apply(len)
train_df['answer_length'] = train_df['answer'].apply(len)

print("\nQuestion Length Statistics:")
print(train_df['question_length'].describe())

print("\nAnswer Length Statistics:")
print(train_df['answer_length'].describe())
```

```
# Function to classify questions
def classify_question(question):
    question = question.lower()
    if question.startswith("is") or question.startswith("are") or question.startswith("do") or question.startswith("does")
    or question.startswith("did") or question.startswith("was") or question.startswith("were")
    or question.startswith("can") or question.startswith("could") or question.startswith("should") or question.startswith("would")
    or question.startswith("will") or question.startswith("won't") or question.startswith("shan't"):
        return "Yes/No"
    elif question.startswith("what"):
        return "What"
    elif question.startswith("where"):
        return "Where"
    elif question.startswith("how much") or question.startswith("how many"):
        return "How much/How many"
    elif question.startswith("how"):
        return "How"
    elif question.startswith("when"):
        return "When"
    elif question.startswith("whose"):
        return "Whose"
    else:
        return "Other"
```

```python
# Plot the distribution of question lengths
plt.figure(figsize=(10, 6))
sns.histplot(train_df['question_length'], bins=30, kde=True)
plt.title('Distribution of Question Lengths')
plt.xlabel('Question Length')
plt.ylabel('Frequency')
plt.show()

# Plot the distribution of answer lengths
plt.figure(figsize=(10, 6))
sns.histplot(train_df['answer_length'], bins=30, kde=True)
plt.title('Distribution of Answer Lengths')
plt.xlabel('Answer Length')
plt.ylabel('Frequency')
plt.show()

# Visualize a few sample images with questions and answers
num_samples = 5
sample_indices = train_df.sample(num_samples).index

for i in sample_indices:
    sample = dataset['train'][i]
    PIL_image = Image.fromarray(np.array(sample['image'])).convert('RGB')
    plt.imshow(PIL_image)
    plt.axis('off')
    plt.title(f"Question: {sample['question']}\nAnswer: {sample['answer']}")
    plt.show()
```

```python
# Print missing values
print("\nMissing Values in Train DataFrame:")
print(train_df.isnull().sum())
```

```python
# Classify each question and add as a new column
train_df['question_type'] = train_df['question'].apply(classify_question)

# Display the first few rows to verify
train_df.head()
```

```python
[ ] # Display the distribution of question types
print("\nQuestion Type Distribution:")
print(train_df['question_type'].value_counts())
```

3.5 Model Configuration

Configure the BLIP model:

```python
[ ] config = BlipConfig.from_pretrained("Salesforce/blip-vqa-base")
```

4 Case 1: Full PathVQA Dataset

4.1 Data Splitting and Selection

For this case, use the entire training and validation splits:

```
train_data = dataset['train']
val_data = dataset['validation']
```

4.2 Model Preparation and Data Handling

Create the VQADataset class to handle Visual Question Answering (VQA) tasks:

```
class VQADataset(torch.utils.data.Dataset):
    def __init__(self, data, segment, text_processor, image_processor):
        self.data = data
        self.questions = data['question']
        self.answers = data['answer']
        self.text_processor = text_processor
        self.image_processor = image_processor
        self.max_length = 32
        self.image_height = 128
        self.image_width = 128

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        # get image + text
        answers = self.answers[idx]
        questions = self.questions[idx]
        image = self.data[idx]['image'].convert('RGB')
        text = self.questions[idx]

        image_encoding = self.image_processor(image,
                                  do_resize=True,
                                  size=(self.image_height,self.image_width),
                                  return_tensors="pt")

        encoding = self.text_processor(
                                  None,
                                  text,
                                  padding="max_length",
                                  truncation=True,
                                  max_length = self.max_length,
                                  return_tensors="pt"
                                  )
        # # remove batch dimension
        for k,v in encoding.items():
            encoding[k] = v.squeeze()
        encoding["pixel_values"] = image_encoding["pixel_values"][0]
        # # add labels
        labels = self.text_processor.tokenizer.encode(
            answers,
            max_length= self.max_length,
            padding="max_length",
            truncation=True,
            return_tensors='pt'
        )[0]
        encoding["labels"] = labels

        return encoding
```

```
# Used to load the pre-trained processors for handling text and image data for the BLIP

text_processor = BlipProcessor.from_pretrained("Salesforce/blip-vqa-base")
image_processor = BlipImageProcessor.from_pretrained("Salesforce/blip-vqa-base")
```

4.3 Data Loading and Batching

Prepare data loaders for training and validation:

```
# Creating dataset instances....

train_vqa_dataset = VQADataset(data=train_data,
                    segment='train',
                    text_processor = text_processor,
                    image_processor = image_processor
                        )

val_vqa_dataset = VQADataset(data=train_data,
                    segment='validation',
                    text_processor = text_processor,
                    image_processor = image_processor
                        )
```

```
def collate_fn(batch):
    input_ids = [item['input_ids'] for item in batch]
    pixel_values = [item['pixel_values'] for item in batch]
    attention_mask = [item['attention_mask'] for item in batch]
    labels = [item['labels'] for item in batch]
    # create new batch
    batch = {}
    batch['input_ids'] = torch.stack(input_ids)
    batch['attention_mask'] = torch.stack(attention_mask)
    batch['pixel_values'] = torch.stack(pixel_values)
    batch['labels'] = torch.stack(labels)

    return batch

train_dataloader = DataLoader(train_vqa_dataset,
                              collate_fn=collate_fn,
                              batch_size=64,
                              shuffle=False)
val_dataloader = DataLoader(val_vqa_dataset,
                            collate_fn=collate_fn,
                            batch_size=64,
                            shuffle=False)
```

4.4 Data Verification

Verify that the data has been batched correctly:

```
batch = next(iter(train_dataloader))
for k,v in batch.items():
    print(k, v.shape)
```

4.5 Model Initialization

Initialize the BLIP model:

```
model = BlipForQuestionAnswering.from_pretrained("Salesforce/blip-vqa-base" )
model.to(device)
```

4.6 Model Training Preparation

Set up the optimizer and retrieve image normalization parameters:

```python
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)
image_mean = image_processor.image_mean
image_std = image_processor.image_std
```

4.7 Data Inspection and Visualization

Unnormalize and visualize an image from the batch:

```python
batch_idx = 1

unnormalized_image = (batch["pixel_values"][batch_idx].cpu().numpy() * np.array(image_std)[:, None, None]) + np.array(image_mean)[:, None, None]
unnormalized_image = np.moveaxis(unnormalized_image, 0, -1)
unnormalized_image = (unnormalized_image * 255).astype(np.uint8)

print("Question: ",text_processor.decode(batch["input_ids"][batch_idx]))
print("Answer: ",text_processor.decode(batch["labels"][batch_idx]))
plt.imshow(Image.fromarray(unnormalized_image))
```

4.8 Model Training

4.8.1 Training Function

Define the training function that handles both training and validation:

```python
[ ] model.train()
    for epoch in range(13):
        print(f"Epoch: {epoch}")
        total_loss = []
        for batch in tqdm(train_dataloader):
            # get the inputs;
            batch = {k:v.to(device) for k,v in batch.items()}

            # zero the parameter gradients
            optimizer.zero_grad()
            # forward + backward + optimize
            outputs = model(**batch)
            loss = outputs.loss
            total_loss.append(loss.item())
            loss.backward()
            optimizer.step()
        print("Loss:", sum(total_loss))
```

```python
def train(model, train_loader, val_loader, optimizer, epochs=3):
    train_losses = []
    val_losses = []
    model.train()
    for epoch in range(epochs):
        print(f"Epoch {epoch+1}/{epochs}")
        train_loss = 0.0
        for batch in tqdm(train_loader):
            inputs = {k: v.to(device) for k, v in batch.items()}
            optimizer.zero_grad()
            outputs = model(**inputs)
            loss = outputs.loss
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        avg_train_loss = train_loss / len(train_loader)
        train_losses.append(avg_train_loss)
        print(f"Training loss: {avg_train_loss:.4f}")

        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for batch in val_loader:
                inputs = {k: v.to(device) for k, v in batch.items()}
                outputs = model(**inputs)
                loss = outputs.loss
                val_loss += loss.item()

        avg_val_loss = val_loss / len(val_loader)
        val_losses.append(avg_val_loss)
        print(f"Validation loss: {avg_val_loss:.4f}")
        model.train()

    return train_losses, val_losses
```

4.8.2 Running the Training

Train the model and retrieve loss values:

```python
# Train the model and get the loss values
train_losses, val_losses = train(model, train_dataloader, val_dataloader, optimizer, epochs=13)
```

4.9 Evaluation

4.9.1 Updated Evaluation Function

Evaluate the model and compute precision, recall, F1 score, and other metrics:

```python
from sklearn.metrics import precision_score, recall_score, f1_score

def evaluate(model, dataloader, device):
    model.eval()  # Set the model to evaluation mode
    true_positives = 0
    false_positives = 0
    false_negatives = 0
    all_preds = []
    all_labels = []

    with torch.no_grad():  # Disable gradient computation for evaluation
        for batch in tqdm(dataloader):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            pixel_values = batch['pixel_values'].to(device)
            labels = batch['labels'].to(device)

            # Generate predictions
            outputs = model.generate(input_ids=input_ids, attention_mask=attention_mask, pixel_values=pixel_values)
            predictions = [text_processor.decode(output, skip_special_tokens=True) for output in outputs]

            # Decode the labels to text for comparison
            true_answers = [text_processor.decode(label, skip_special_tokens=True) for label in labels]

            # Collect all predictions and labels for metric calculation
            all_preds.extend(predictions)
            all_labels.extend(true_answers)

            for pred, true in zip(predictions, true_answers):
                if pred.strip().lower() == true.strip().lower():
                    true_positives += 1  # Count as true positive
                else:
                    false_negatives += 1  # Count as false negative
                    false_positives += 1  # For simplicity, consider any wrong prediction as false positive

    # Calculate precision, recall, and F1 score
    precision = precision_score(all_labels, all_preds, average='micro')
    recall = recall_score(all_labels, all_preds, average='micro')
    f1 = f1_score(all_labels, all_preds, average='micro')

    accuracy = true_positives / len(all_labels)

    return accuracy, precision, recall, f1, true_positives, false_negatives, false_positives
```

4.9.2 Evaluation Results and Metrics

Calculate and print the evaluation metrics:

```python
# Evaluate the model on the validation set
val_accuracy, val_precision, val_recall, val_f1, true_positives, false_negatives, false_positives = evaluate(model, val_dataloader, device)

# Print the evaluation results
print(f"Validation Accuracy: {val_accuracy:.4f}")
print(f"Precision: {val_precision:.4f}")
print(f"Recall: {val_recall:.4f}")
print(f"F1 Score: {val_f1:.4f}")
print(f"True Positives: {true_positives}")
print(f"False Negatives: {false_negatives}")
print(f"False Positives: {false_positives}")
```

Plot the training and validation losses:

```python
# Plot the losses
plt.figure()
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.show()
```

5 Case 2: Subset of PathVQA Dataset

Follow the same steps as in Case 1, but replace the data splitting and selection step with:

```
[ ]  train_data = dataset['train'].select(range(10000))
     val_data = dataset['validation'].select(range(1000))
```

6 How to Run the Code?

To execute the project, the following Jupyter Notebook files are provided in the 'code.zip' archive:

- `mVQA_case1-full_Code.ipynb`: This notebook handles the full PathVQA dataset and includes all the steps for training and evaluating the model on the complete dataset.

- `mVQA_case2-Subset_Code.ipynb`: This notebook processes a subset of the PathVQA dataset, demonstrating how to run the model on a smaller portion of the data for quicker evaluation.

Both notebooks include detailed instructions and are pre-configured to work with Google Colab or a similar environment with GPU support. Ensure that the required hardware and software dependencies are met before executing the code.

References

Huang, J., Chen, Y., Li, Y., Yang, Z., Gong, X., Wang, F. L., Xu, X. and Liu, W. (2023). Medical knowledge-based network for patient-oriented visual question answering, *Information Processing & Management* **60**(2): 103241.