MedBLIP - Image Captioning for Radiology X-Ray Images

depi final project

name: dalia refaat

MedBLIP - Image Captioning for Radiology X-Ray Images

Overview:

This repository implements BLIP (Bootstrapping Language-Image Pre-training) to generate captions from Radiology X-ray images.

What is BLIP?
A powerful vision-language pre-training model designed for efficient handling of image-captioning tasks.
Focus of the Project:
The project specifically targets medical imaging, applying BLIP to the Radiology Objects in Context (ROCO) dataset.
Importance:
Enhances the accessibility of medical imaging by generating descriptive captions, aiding in better understanding and interpretation of X-ray images.

The BLIP (Bootstrapping Language-Image Pre-training) model comprises several components designed to enhance its performance in vision-language tasks. Here's a breakdown of the models and components within BLIP:

1. Vision Encoder:
Architecture: Typically based on Convolutional Neural Networks (CNNs) or Vision Transformers (ViTs).
Function: This component extracts features from the input images, providing a rich representation that the language model can utilize.
2. Language Decoder:
Architecture: Often implemented using Transformer architectures, similar to those used in models like GPT or BERT.
Function: This decoder takes the encoded visual features and generates text captions based on those features
3. Bootstrapping Mechanism:
Function: This unique aspect of BLIP allows it to iteratively improve its language-image representations. It leverages both image and text data during training to enhance the understanding of context and meaning.

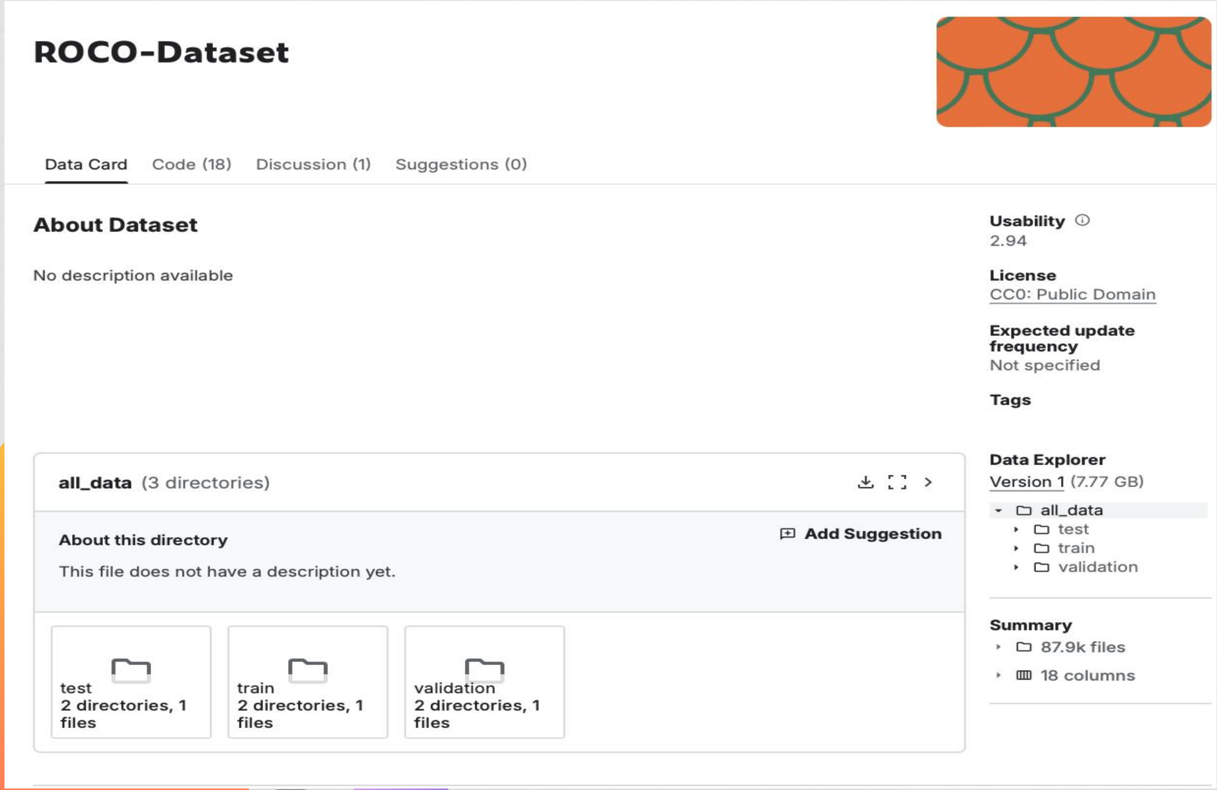
4. Cross-Modal Attention:
Function: The attention layers enable the model to focus on specific regions of an image while generating corresponding text. This is crucial for generating accurate and contextually relevant captions.

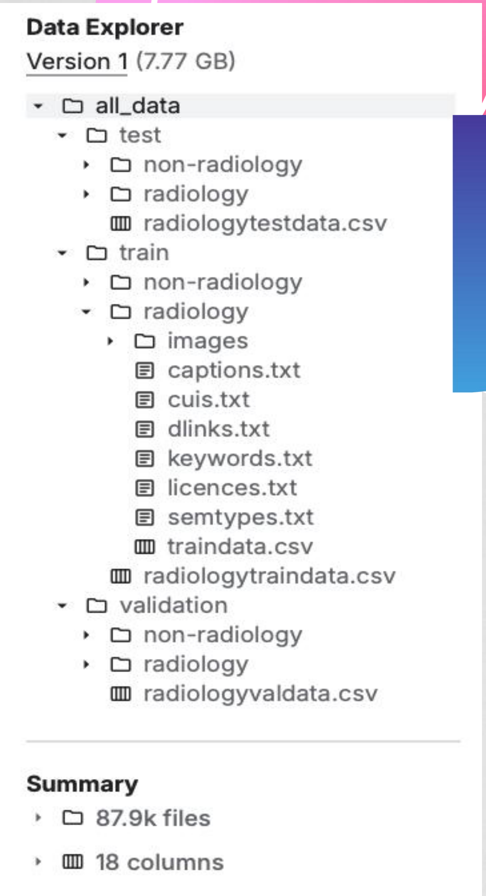
5. Multi-Task Learning:
Function: BLIP can be trained on various tasks, such as image captioning, visual question answering, and image-text retrieval, which enhances its versatility.

6. Pre-trained Checkpoints:
Function: BLIP often provides pre-trained models on large datasets, allowing users to fine-tune them on specific tasks or domains, such as medical imaging, without needing to train from scratch.

Summary of Models:
Vision Encoder: Extracts visual features (CNNs or ViTs).
Language Decoder: Generates textual captions (Transformers).
Bootstrapping Mechanism: Enhances representations iteratively.
Cross-Modal Attention: Links visual features with text.
Multi-Task Learning: Trained for various vision-language tasks.
Pre-trained Checkpoints: Ready for fine-tuning on specialized datasets.

About dataset:





checking for missing values:

Explanation:

- This function checks for missing or invalid image files in the DataFrame.
- It iterates through the image paths, verifying if each file exists and can be opened.
- If invalid files are found, those rows are removed from the DataFrame to ensure data integrity.

3. Checking for Missing or Invalid Files

python

Copy code

```python
def check_and_remove_missing_or_invalid_files(df, column_name):
    missing_or_invalid_files = []

    for idx, image_path in df[column_name].items():
        if not os.path.isfile(image_path):  # Check if file exists
            missing_or_invalid_files.append(idx)
        else:
            try:
                img = Image.open(image_path)
                img.verify()  # Verify that it is, indeed, an image
            except Exception as e:
                print(f"Error with file {image_path}: {e}")
                missing_or_invalid_files.append(idx)

    if missing_or_invalid_files:
        print(f"Removing {len(missing_or_invalid_files)} rows with missing or in
        df = df.drop(missing_or_invalid_files).reset_index(drop=True)
    else:
        print("All files are present and valid!")

    return df
```

Dataset Class creation:

Explanation:

- The ImageCaptioningDataset class inherits from Dataset, allowing for a customized dataset structure.
- The __init__ method initializes the dataset, processor, and image size.
- The __len__ method returns the total number of samples in the dataset.
- The __getitem__ method processes the image and caption, preparing them for input to the model.

4. Dataset Class Creation

python Copy code

```python
class ImageCaptioningDataset(Dataset):  # Custom Dataset class for image caption
    def __init__(self, dataset, processor, image_size=(224, 224)):
        self.dataset = dataset
        self.processor = processor
        self.image_size = image_size
        self.resize_transform = Resize(image_size)

    def __len__(self):
        return len(self.dataset)  # Returns the number of samples in the dataset

    def __getitem__(self, idx):
        item = self.dataset.iloc[idx]  # Get the sample at index idx
        img = Image.open(item['images'])  # Open the image file
        encoding = self.processor(images=img, text=item["caption"], padding="max
        # Process the image and text

        encoding = {k: v.squeeze() for k, v in encoding.items()}  # Adjust encod
        return encoding  # Return processed data
```

Checkpoint Management Functions:

Explanation:

- The load_checkpoint function checks if a checkpoint file exists.
- If it does, the model and optimizer states are restored, allowing training to resume from where it left off.
- If not, it initializes the training from the beginning, setting the starting epoch to zero.
- The save_checkpoint function creates a directory for saving checkpoints if it doesn't exist.
- It saves the current epoch, model state, optimizer state, and average loss to a file.
- This function ensures that progress can be restored later without losing any training data.

11. Checkpoint Management Functions

Load Checkpoint Function

python Copy code

```python
def load_checkpoint(model, optimizer, checkpoint_path):
    if os.path.exists(checkpoint_path):
        checkpoint = torch.load(checkpoint_path)  # Load the checkpoint file
        model.load_state_dict(checkpoint['model_state_dict'])  # Load the model
        optimizer.load_state_dict(checkpoint['optimizer_state_dict'])  # Load th
        start_epoch = checkpoint['epoch']  # Get the starting epoch from the che
        print(f"Loaded checkpoint from {checkpoint_path} at epoch {start_epoch}"
    else:
        start_epoch = 0  # If no checkpoint, start from epoch 0
        print(f"No checkpoint found at {checkpoint_path}. Starting from scratch.
    return start_epoch
```

Save Checkpoint Function

python Copy code

```python
def save_checkpoint(model, optimizer, epoch, avg_loss, checkpoint_path='checkpoi
    if not os.path.exists('checkpoints'):
        os.makedirs('checkpoints')  # Create checkpoints directory if it doesn't
    torch.save({
        'epoch': epoch,
        'model_state_dict': model.state_dict(),  # Save the model state
        'optimizer_state_dict': optimizer.state_dict(),  # Save the optimizer st
        'avg_loss': avg_loss,  # Save the average loss
    }, checkpoint_path)  # Save checkpoint file
    print(f"Checkpoint saved at epoch {epoch} with average loss: {avg_loss:.4f}"
```

Loss Calculation and Backpropagation:

Explanation:

- The loss calculated from the model's output is used for backpropagation.
- The optimizer clears previous gradients, computes the new gradients based on the current loss, and updates the model parameters accordingly.

9. Loss Calculation and Backpropagation

python   Copy code

```python
loss = outputs.loss  # Get loss

# Backpropagation
optimizer.zero_grad()  # Clear previous gradients
loss.backward()  # Backpropagate the loss
optimizer.step()  # Update model parameters
```

BLEU Score Calculation Function:

Explanation:

- The calculate_bleu_score function computes the BLEU score, a metric for evaluating the quality of generated text.
- It splits both reference captions and generated captions into words before calculating the score.
- This function can be used to assess the quality of captions generated by the model.

13. BLEU Score Calculation Function

python Copy code

```python
def calculate_bleu_score(references, hypothesis):
    references = [ref.split() for ref in references]  # Split reference sentenc
    hypothesis = hypothesis.split()  # Split hypothesis sentence into words
    score = sentence_bleu(references, hypothesis)  # Calculate BLEU score
    return score  # Return the BLEU score
```