

Practical Lab File for

Agentic AI

**Sharda School of Engineering
and Technology**

Name – Mohd Mujeeb

Sys ID- 2023248197

Semester/Year- 6th sem, 3rd Year

**Faculty-In-Charge/Submitted To
Mr. Ayush Singh**



BLIP Image Captioning Fine-Tuning Documentation

Overview

This notebook demonstrates how to fine-tune the BLIP (Bootstrapping Language-Image Pre-training) model for image captioning on a custom dataset. BLIP is a vision-language model developed by Salesforce that can generate natural language descriptions of images.

What This Notebook Does

1. Sets up the environment with necessary libraries
 2. Loads a custom football-themed image captioning dataset
 3. Prepares the data for training using PyTorch
 4. Fine-tunes a pre-trained BLIP model on the custom dataset
 5. Generates captions for images using the fine-tuned model
 6. Demonstrates how to load and use models from Hugging Face Hub
-

Table of Contents

1. Environment Setup
 2. Dataset Loading
 3. PyTorch Dataset Creation
 4. Model and Processor Loading
 5. Training Configuration
 6. Training Loop
 7. Inference
 8. Loading Pre-trained Models from Hub
 9. Batch Inference and Visualization
-

Environment Setup

Installation Commands

```
!pip install git+https://github.com/huggingface/transformers.git@main  
!pip install -q datasets
```

Purpose: These commands install the required libraries:

- **transformers:** Hugging Face's library for state-of-the-art NLP and vision-language models (installed from the main branch to get the latest features)
- **datasets:** Hugging Face's library for easy access to datasets and efficient data processing

Why install from GitHub: Installing transformers from the main branch ensures you have the latest features and bug fixes that might not be in the PyPI release yet.

Dataset Loading

Code

```
from datasets import load_dataset  
  
dataset = load_dataset("ybelkada/football-dataset", split="train")
```

What This Does:

- Loads a custom football-themed image captioning dataset from Hugging Face Hub
- The dataset contains images of football/soccer scenes with corresponding text captions
- Uses only the "train" split for this demonstration

Dataset Exploration

```
# View a caption dataset[0]["text"]  
  
# View the corresponding image  
dataset[0]["image"]
```

Dataset Structure:

- Each example contains two fields:
- "image": A PIL Image object
- "text": A string caption describing the image

Example Output: A caption might be something like "a player kicking a football on the field" with the corresponding image showing that scene.

PyTorch Dataset Creation

Custom Dataset Class

```
from torch.utils.data import Dataset, DataLoader class

ImageCaptioningDataset(Dataset):
    def __init__(self, dataset, processor):
        self.dataset = dataset
        self.processor = processor

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

    def __getitem__(self, idx):
        item = self.dataset[idx]
        encoding = self.processor(images=item["image"], text=item["text"],
                                  padding="max_length", return_tensors="pt")
        # remove batch dimension
        encoding = {k:v.squeeze() for k,v in encoding.items()} return encoding
```

Class Breakdown

Purpose: This custom PyTorch Dataset class wraps the Hugging Face dataset and handles preprocessing for BLIP.

Key Components:

1. __init__ method:

- Stores the dataset and processor for later use
- The processor handles both image and text preprocessing

2. __len__ method:

- Returns the total number of examples in the dataset
- Required by PyTorch's Dataset interface

3. `__getitem__` method:

- Retrieves a single example at index `idx`
- Processes both the image and text using the BLIP processor
- **Processing steps:**
 - Converts the image to the format expected by BLIP (normalized tensors)
 - Tokenizes the text caption
 - Applies padding to ensure consistent sequence length
 - Returns PyTorch tensors
 - `.squeeze()`: Removes the batch dimension added by `return_tensors="pt"` since the DataLoader will add it back

Output Format:

The encoding dictionary contains:

- `input_ids`: Tokenized text caption
 - `pixel_values`: Preprocessed image tensor
 - `attention_mask`: Mask indicating which tokens are padding
-

Model and Processor Loading

Code

```
from transformers import AutoProcessor, BlipForConditionalGeneration  
  
processor = AutoProcessor.from_pretrained("Salesforce/blip-image-captioning-base")  
model = BlipForConditionalGeneration.from_pretrained("Salesforce/blip-image-captioning-base")
```

What This Does:

1. **AutoProcessor**: Loads the BLIP processor which handles:

- Image preprocessing (resizing, normalization)
- Text tokenization
- Creating proper input formats for the model

2. **BlipForConditionalGeneration**: Loads the pre-trained BLIP model

- "base" variant is a medium-sized model balancing performance and speed
- The model is already trained on large-scale image-caption pairs
- Fine-tuning will adapt it to the specific domain (football images)

DataLoader Creation

```
train_dataset = ImageCaptioningDataset(dataset, processor) train_dataloader = DataLoader(train_dataset,  
shuffle=True, batch_size=2)
```

Parameters:

- **shuffle=True**: Randomizes the order of examples each epoch (important for training)
 - **batch_size=2**: Processes 2 examples at a time (small batch size, likely due to GPU memory constraints)
-

Training Configuration

Optimizer Setup

```
import torch  
  
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)
```

Optimizer Choice: AdamW (Adam with Weight Decay)

- Popular choice for fine-tuning transformer models
- Includes weight decay for regularization
- **lr=5e-5(0.00005)**: A typical learning rate for fine-tuning pre-trained models

Device Configuration

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

Purpose:

- Automatically detects if a GPU (CUDA) is available
- Moves the model to GPU for faster training if available
- Falls back to CPU if no GPU is present

Training Mode

```
model.train()
```

What This Does:

- Sets the model to training mode
 - Enables dropout and batch normalization layers to update during training
 - Essential before starting the training loop
-

Training Loop

Complete Training Code

```
for epoch in range(50): print("Epoch:", epoch)
    for idx, batch in enumerate(train_dataloader): input_ids =
        batch.pop("input_ids").to(device) pixel_values =
        batch.pop("pixel_values").to(device)

        outputs = model(input_ids=input_ids,
                        pixel_values=pixel_values, labels=input_ids)

        loss = outputs.loss print("Loss:", loss.item())

        loss.backward()

        optimizer.step() optimizer.zero_grad()
```

Training Loop Breakdown

Outer Loop - Epochs:

- `range(50)`: Trains for 50 complete passes through the dataset
- Each epoch processes all training examples once

Inner Loop - Batches:

1. Data Preparation:

```
`python
input_ids = batch.pop("input_ids").to(device)
pixel_values = batch.pop("pixel_values").to(device)
```

- Extracts inputs from the batch
- Moves tensors to GPU/CPU
- `.pop()` removes the item from the dictionary

2. Forward Pass:

```
'python  
outputs = model(input_ids=input_ids,  
pixel_values=pixel_values, labels=input_ids)
```

- Passes images and text through the model
- `labels=input_ids`: Uses the same text as labels (teacher forcing)
- The model learns to predict the caption given the image

3. Loss Calculation:

```
'python  
loss = outputs.loss  
  
'  
  
• Automatically calculated by the model (cross-entropy loss)  
• Measures how different the predicted caption is from the true caption
```

4. Backward Pass:

```
'python loss.backward()  
  
'  
  
• Computes gradients for all model parameters  
• Gradients show how to adjust weights to reduce loss
```

5. Parameter Update:

```
'python optimizer.step()  
optimizer.zero_grad()  
  
'  
  
• optimizer.step(): Updates model weights using computed gradients  
• optimizer.zero_grad(): Resets gradients to zero for the next iteration
```

Training Monitoring:

- Prints the current epoch number
 - Prints loss for each batch to track training progress
 - Loss should generally decrease over time if training is working
-

Inference

Single Image Captioning

```
# Load image
example = dataset[0] image =
example["image"]

# Prepare image for the model
inputs = processor(images=image, return_tensors="pt").to(device) pixel_values = inputs.pixel_values

# Generate caption
generated_ids = model.generate(pixel_values=pixel_values, max_length=50) generated_caption =
processor.batch_decode(generated_ids, skip_special_tokens=True)[0] print(generated_caption)
```

Inference Breakdown

1. Load and Display Image:

```
`python
example = dataset[0] image =
example["image"]`
```

- Retrieves the first example from the dataset
- Extracts the PIL Image object

2. Preprocess Image:

```
`python
inputs = processor(images=image, return_tensors="pt").to(device) pixel_values =
inputs.pixel_values`
```

- Converts image to model-compatible format
- `return_tensors="pt"`: Returns PyTorch tensors

- Moves to GPU/CPU for processing

3. Generate Caption:

```
`python
generated_ids = model.generate(pixel_values=pixel_values, max_length=50)
`
```

- **Autoregressive generation:** Generates caption one token at a time
- **max_length=50:** Limits caption to 50 tokens maximum
- Returns token IDs (numerical representation)

4. Decode to Text:

```
`python
generated_caption = processor.batch_decode(generated_ids, skip_special_tokens=True)[0]
`
```

- Converts token IDs back to human-readable text
 - **skip_special_tokens=True:** Removes special tokens like [PAD], [SEP], etc.
 - **[0]:** Gets the first (and only) caption from the batch
-

Loading Pre-trained Models from Hub

Code

```
from transformers import BlipForConditionalGeneration, AutoProcessor
model =
    BlipForConditionalGeneration.from_pretrained(
        "ybelkada/blip-image-captioning-base-football-finetuned"
    ).to(device)

processor = AutoProcessor.from_pretrained(
    "ybelkada/blip-image-captioning-base-football-finetuned"
)
```

Purpose:

- Demonstrates how to load a model that has already been fine-tuned and uploaded to Hugging Face Hub
- Useful for:
 - Sharing models with others
 - Resuming work later
 - Using models trained by others

Model ID: "ybelkada/blip-image-captioning-base-football-finetuned"

- Format: `username/model-name`
 - This is a pre-trained version of the same model shown in the training section
 - Trained on the same football dataset
-

Batch Inference and Visualization

Visualization Code

```
from matplotlib import pyplot as plt fig =  
plt.figure(figsize=(18, 14))  
  
# Process and visualize multiple images for i, example in  
enumerate(dataset):  
    image = example["image"]  
  
    # Prepare image  
    inputs = processor(images=image, return_tensors="pt").to(device) pixel_values = inputs.pixel_values  
  
    # Generate caption  
    generated_ids = model.generate(pixel_values=pixel_values, max_length=50) generated_caption =  
    processor.batch_decode(generated_ids, skip_special_tokens=True)[0]  
  
    # Plot image with caption  
    fig.add_subplot(2, 3, i+1) plt.imshow(image)  
    plt.axis("off")  
    plt.title(f"Generated caption: {generated_caption}")
```

Visualization Breakdown

1. Figure Creation:

```
`python
```

```
fig = plt.figure(figsize=(18, 14))
```

```
`
```

- Creates a large figure (18x14 inches)
- Provides space for multiple images

2. Loop Through Dataset:

```
`python
```

```
for i, example in enumerate(dataset):
```

- Iterates through all examples in the dataset
- `enumerate()` provides both index and example

3. Image Processing:

- Same preprocessing and generation steps as single image inference
- Repeated for each image in the dataset

4. Subplot Creation:

```
`python fig.add_subplot(2,  
3, i+1)
```

- Creates a 2x3 grid (2 rows, 3 columns)
- `i+1`: Subplot indices start at 1, not 0
- Can display up to 6 images

5. Display Settings:

```
`python  
plt.imshow(image)  
plt.axis("off")  
plt.title(f"Generated caption: {generated_caption}")
```

- Shows the image
- Hides axis labels for cleaner display
- Displays the generated caption as the title

Result: A grid showing multiple football images with their generated captions, making it easy to evaluate model performance visually.

Key Concepts and Best Practices

1. Transfer Learning

- The notebook uses **transfer learning**: starting with a pre-trained model and adapting it to a specific domain

- Pre-trained BLIP already understands general image-text relationships
- Fine-tuning teaches it football-specific vocabulary and contexts

2. Teacher Forcing

- During training, `labels=input_ids` implements teacher forcing
- The model sees the correct previous tokens when predicting the next token
- Speeds up training but means the model learns with perfect context

3. Batch Size Considerations

- `batch_size=2` is very small
- Likely due to GPU memory constraints
- Smaller batches mean:
 - Less memory usage
 - More gradient updates
 - Potentially noisier training
 - Slower training overall

4. Learning Rate

- `lr=5e-5` is a typical fine-tuning learning rate
- Much smaller than training from scratch
- Prevents catastrophic forgetting of pre-trained knowledge

5. Evaluation Considerations

- The notebook doesn't include a validation set
- In practice, you should:
 - Split data into train/validation/test sets
 - Monitor validation loss during training
 - Use metrics like BLEU, METEOR, or CIDEr for caption quality

Potential Improvements

1. **Add Validation:** Split dataset and monitor validation loss to prevent overfitting

- 2. Learning Rate Scheduling:** Decrease learning rate over time for better convergence
 - 3. Early Stopping:** Stop training when validation loss stops improving
 - 4. Data Augmentation:** Apply image transformations to increase dataset variety
 - 5. Gradient Accumulation:** Simulate larger batch sizes with limited GPU memory
 - 6. Mixed Precision Training:** Use FP16 to speed up training and reduce memory usage
 - 7. Checkpointing:** Save model periodically to resume training if interrupted
 - 8. Quantitative Evaluation:** Add BLEU/METEOR scores to measure caption quality
-

Common Issues and Solutions

Issue 1: Out of Memory (OOM) Errors

Solution:

- Reduce batch size to 1
- Use gradient accumulation
- Enable mixed precision training
- Use a smaller model variant

Issue 2: Model Not Improving

Solution:

- Check if loss is decreasing
- Try different learning rates
- Ensure data is properly preprocessed
- Verify labels are correct

Issue 3: Poor Caption Quality

Solution:

- Train for more epochs
- Use a larger dataset
- Try a larger BLIP model (e.g., "large" instead of "base")

- Check if dataset quality is good

Issue 4: Slow Training

Solution:

- Use GPU if available
 - Increase batch size if memory allows
 - Use mixed precision training
 - Reduce dataset size for experimentation
-

Requirements Summary

Hardware

- **GPU:** Recommended (NVIDIA with CUDA support)
- **RAM:** At least 8GB
- **Storage:** A few GB for models and datasets

Software

- **Python:** 3.7+
 - **PyTorch:** Latest version (installed with transformers)
 - **transformers:** Latest from GitHub main branch
 - **datasets:** Latest version
 - **matplotlib:** For visualization
-

Conclusion

This notebook provides a complete pipeline for fine-tuning BLIP on a custom image captioning dataset. The process involves:

1. Loading a pre-trained vision-language model
2. Preparing a custom dataset
3. Fine-tuning with appropriate hyperparameters

4. Generating captions for new images
5. Sharing models via Hugging Face Hub

The approach demonstrated here can be adapted to any image captioning task by simply changing the dataset while keeping the overall structure intact.