BLIP-2 Models Benchmarking

Section 1: Introduction

The orignal paper of BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models has been implemented by Salesforce Lavis project. The Lavis project provides a documentation about how to use BLIP2 but it lacks a benchmarking section. It also doesn't explain how to work with these model when the compute power is relatively on the lower end. In this article we will explain how to circumvent some computation problem along with 3 (out of 7 models) model's benchmarking. The Lavis project provides a comprehensive guide of instructed vision-to-language generation in a Notebook-Demo.

The guide uses the default model `pretrain_flant5xxl`, which is the largest, around 48 GB. Without a doubt, this model performs the best, giving captions like:

Write a romantic message that goes along with this photo.

Love is like a sunset, it's hard to see it coming, but when it does, it's so beautiful.`

However, the model warns the user by stating,

Large RAM is required to load the larger models. Running on a GPU can optimize inference speed.

without providing any context or explanation of the terms "Large RAM," "GPU," and "CPU" usage.

Section 2: Loading BLIP2 Captioning models

The Lavis project provides 7 models for BLIP2 captioning. The models are listed below:

Model Name	Parameter Count	Estimated Size (in GB)
pretrain_flant5xxl	11 billion	~48.0 GB
pretrain_flant5xl	3 billion	~15.0 GB
caption_coco_flant5xl	3 billion	~4.0 GB
pretrain_opt2.7b	2.7 billion	~11.0 GB
pretrain_opt6.7b	6.7 billion	~26.0 GB
caption_coco_opt2.7b	2.7 billion	~11.0 GB
caption_coco_opt6.7b	6.7 billion	~26.0 GB

The Lavis project uses the pretrain_flant5xxl model to demonstrate the best-performing model. Following code snippet is suggested to use in order to load the pretrained model.

```
model, vis_processors, _ = load_model_and_preprocess(
    name="blip2_t5", model_type="pretrain_flant5xxl", is_eval=True, device=device
)
```

Unfortunately this code fails to load the model due to an error

```
OutOfMemoryError: CUDA out of memory. Tried to allocate 80.00 MiB (GPU 0; 8.00 GiB
total capacity; 22.51 GiB
already allocated; 0 bytes free; 22.68 GiB reserved in total by PyTorch) If reserved
memory is >> allocated memory
try setting max_split_size_mb to avoid fragmentation. See documentation for Memory
Management and
PYTORCH_CUDA_ALLOC_CONF
```

It is crucial to note that the error occurred with the below hardware specification:

Component	Specification
CPU	Intel® Core™ i7-7700 CPU @ 3.60GHz
GPU	NVIDIA GeForce GTX 1070
GPU Memory	8 GB (GDDR5)

We have circumvented this problem by implementing the following code:

```
import torch
import os
from torch.cuda.amp import autocast, GradScaler
from accelerate import Accelerator

torch.cuda.empty_cache()

os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'max_split_size_mb:128'

accelerator = Accelerator(cpu=True)

def clear_memory():
    torch.cuda.empty_cache()
    torch.cuda.synchronize()

clear_memory()

scaler = GradScaler()

try:
    with autocast():
        model, vis_processors, _ = load_model_and_preprocess(
            name="blip2_t5", model_type="pretrain_flant5xxl", is_eval=True,
device="cpu"  # Load on CPU first
        )

    if hasattr(model, 'gradient_checkpointing_enable'):
        model.gradient_checkpointing_enable()

    model = accelerator.prepare(model)

except RuntimeError as e:
    print(f"Error during model loading: {e}")
    clear_memory()
```

Code Explanation

1. `torch.cuda.empty_cache()`:
    - Clears the GPU memory cache to free up unused memory, helping to prevent out-of-memory errors.
2. `os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'max_split_size_mb:128'`:
    - Sets an environment variable to manage memory fragmentation by limiting the maximum size of memory allocation splits to 128 MB.
    - Memory fragmentation occurs when the available memory is broken into small, non-contiguous blocks over time, making it difficult to allocate large contiguous memory blocks

required by large models. By setting `max_split_size_mb`, we ensure that the memory allocator splits large allocations into smaller, more manageable chunks, reducing fragmentation and helping to make better use of the available memory. This setting helps avoid scenarios where large allocations fail due to insufficient contiguous free memory, even when there is enough total free memory.

3. `accelerator = Accelerator(cpu=True)`:
   - Initializes an `Accelerator` instance with CPU offloading enabled. This allows the model to offload some operations to the CPU, optimizing memory usage on the GPU.
4. `def clear_memory():`:
   - `clear_memory` to clear GPU memory and synchronize GPU operations.
   - `torch.cuda.empty_cache()`: Clears the GPU memory cache again to free up unused memory.
   - `torch.cuda.synchronize()`: Synchronizes the GPU operations, ensuring all pending operations are completed. This helps in accurately managing memory.
5. `clear_memory()`:
   - Calls the `clear_memory` function to clear memory before loading the model, ensuring there is enough free memory available.
6. `scaler = GradScaler()`:
   - Initializes a `GradScaler` instance for scaling gradients during mixed precision training, preventing underflow and maintaining training stability.
7. `with autocast():`:
   - Uses the `autocast` context manager to enable mixed precision for the operations within the block, reducing memory usage and improving performance.
8. `model, vis_processors, _ = load_model_and_preprocess(name="blip2_t5", model_type="pretrain_flant5xxl", is_eval=True, device="cpu")`:
   - Loads the model and preprocessing components. Initially loads the model on the CPU to avoid GPU memory issues during the loading process.
9. `if hasattr(model, 'gradient_checkpointing_enable'):`:
   - Checks if the model supports gradient checkpointing, a technique that saves memory during training by trading compute for memory.
10. `model = accelerator.prepare(model)`:
    - Prepares the model with the `Accelerator` instance, optimizing it for the available hardware and mixed precision.
11. `clear_memory()`:
    - Calls the `clear_memory` function again to free up memory in case of an error, attempting to mitigate memory issues.

This process of model loading is not only limited to the loading of the `pretrain_flant5xxl` model rather we have also used the same strategy to load even smaller models like `pretrain_flant5xl` and `caption_coco_flant5xl`

Section 3: Benchmarking

We will start the benchmarking comparison based on the following factors: * Model Size * Model Loading time * Model Inference timing and accuracy * Captioning * Non deterministic Necleus sampling Captioning

BLIP2 Model Versions	Input Image Dimension	Model Size	Model Loading Time	Model Saving Time	Inference Time	Caption	Non-Deterministic Nucleus Sampling
pretrain_flant5xxl	224 * 224	~48 GB	~18 Mins	~10 Mins	~8 Mins	a lion laying down in the grass	a lion laying down in the grass (same 3 captions in ~21 Minutes)

pretrain_flant5xl	224 * 224	~15.4 GB	`CPU: 3 Mins 52.1 Sec`
			
			`GPU: 1 Min 40 Sec`
				CPU: 1 Min 42.9 Sec	28 Sec	a lion laying down in the grass	a lion laying down in the grass
							
							(same 3 captions in 52 secs)

caption_coco_flant5xl	364 * 364	~4 GB	`CPU: 4 Mins 15.9 Sec`
			
			`GPU: 1 Min 48 Sec`
				CPU: 1 Min 48.9 Sec	43 Sec	a lion laying down in a grassy field	a lion laying down in a grassy field
							
							(same 3 captions in 1 Minute 16 secs)

Metric	pretrain_flant5xxl (CPU)	pretrain_flant5xxl (GPU)	pretrain_flant5xl (CPU)	pretrain_flant5xl (GPU)	caption_coco_flant5xl (CPU)	caption_coco_flant5xl (GPU)
Model Loading Time	N/A	~15-18 Mins	3 Mins 52.1 Sec	1 Min 40 Sec	4 Mins 15.9 Sec	1 Min 48 Sec
Model Saving Time	~10 Mins	N/A	1 Min 42.9 Sec	N/A	1 Min 48.9 Sec	N/A
Inference Time	~8 Mins	N/A	28 Sec	N/A	43 Sec	N/A