Deliverables

Notebook link:

https://drive.google.com/file/d/1NELvqA4y-gLJd_RVLrg4-FlaHcW53TI4/view?usp=sharing

Sample data link (snapshot at the end):

https://drive.google.com/file/d/1tK6Pr54fYAVlqt7TPjUD9s-Gdkr46wg5/view?usp=sharing

Assignment Report

Dataset Sources and Total Size

The raw dataset for this assignment was a subset of the **wikimedia/wikipedia** dataset, specifically the **20231101.en** configuration. This public dataset, hosted on the Hugging Face Hub, was chosen as the primary source due to its encyclopedic nature and its suitability for pre-training large language models.

To avoid downloading the entire corpus (which is tens of gigabytes), a memory-efficient approach using Hugging Face's **streaming mode** was employed.

```
dataset_name = "wikimedia/wikipedia"
config_name = "20231101.en"

# Load the dataset in streaming mode, this avoids downloading the whole file
print(f"Loading dataset in streaming mode: {dataset_name} with config {config_name}")
streaming_dataset = load_dataset(dataset_name, config_name, split="train", streaming=True)
```

To get the required **1GB** of raw text, we collected and processed approximately **150,000 articles** into an in-memory `Dataset` object.

```
# Collect articles until we have roughly 1GB data
num_articles_to_collect = 150000
subset_data = []

print(f"Collecting approximately {num_articles_to_collect} articles...")
for i, article in enumerate(streaming_dataset):
    if i >= num_articles_to_collect:
        break
    subset_data.append(article)

print(f"Finished collecting {len(subset_data)} articles.")
```

Cleaning Strategies and Reasoning

The preprocessing pipeline was designed to transform the raw text into a high-quality, normalized corpus for pre-training.

Two custom functions were written.

1. clean_and_normalize(). It leverages regular expression to remove HTML tags and Wikipedia internal links, converts all letters to lower case, removes irrelevant symbols and white spaces.
2. filter_short_documents(). It removes all documents shorter than 50 words.

Than we apply map() function on the raw dataset, and filter() function on the cleaned and normalized dataset, as this separation would maximize efficiency of those two functions.

```python
def clean_and_normalize(examples):
    cleaned_texts = []

    for text in examples['text']:
        # Step 1: Optional - Remove HTML tags and markdown
        text = re.sub(r'<[^>]+>', '', text)  # Remove HTML tags
        text = re.sub(r'\[\[[^\]]+\]\]', '', text) # Remove wiki-internal links

        # Step 2: Lowercase the entire text
        text = text.lower()

        # Step 3: Strip irrelevant symbols and normalize
        text = re.sub(r'[^a-z0-9\s.,?!;:\'-]', '', text)

        # Step 4: Remove extra whitespace
        text = re.sub(r'\s+', ' ', text).strip()

        cleaned_texts.append(text)

    examples['text'] = cleaned_texts
    return examples

def filter_short_documents(examples):
    return [len(text.split()) >= 50 for text in examples['text']]
```

Tokenization Choices

We chose to use a **WordPiece** tokenizer, specifically the **`bert-base-uncased`** tokenizer from Hugging Face. This is a standard choice for transformer-based models and is well-suited for English text.

A key decision was the **chunking strategy** to handle the very long documents found in the Wikipedia dataset. We set a **`max_length` of 512** tokens, which is a common block size for models like BERT. To avoid data loss and preserve context, we used an **overlap `stride` of 128 tokens** between consecutive chunks. This ensured that no information was lost at the boundaries of the chunks.

```
from transformers import AutoTokenizer

model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

✓ 2.6s

```
# Tokenize the dataset and handle long sequences

def tokenize_function(examples):
    tokenized_chunks = tokenizer(
        examples['text'],
        truncation=True,
        return_overflowing_tokens=True,
        max_length=512,  # Set your desired chunk size (e.g., 512 for BERT)
        stride=128  # Use a stride to create overlapping chunks and preserve context
    )
    return tokenized_chunks

# Apply the chunking function to the dataset
chunked_dataset = filtered_dataset.map(
    tokenize_function,
    batched=True,
    remove_columns=filtered_dataset.column_names # Remove the old columns to prevent errors
)
```

Data Loader Implementation Details

The data loader was implemented using the standard PyTorch `DataLoader` and was designed to work efficiently with the tokenized, streaming data.

- **Data Structure:** The tokenized data was maintained in an efficient, in-memory Hugging Face `Dataset` object.
- **Batching and Padding:** We used a **`DataCollatorWithPadding`** to dynamically pad each batch to the length of the longest sequence within that batch. This is more memory-efficient than padding all sequences to the maximum sequence length (512) and significantly speeds up training.
- **Iterable Streaming:** The `DataLoader` was built on a streaming dataset, meaning it processed data on-the-fly without needing to load the entire tokenized corpus into memory, which was a critical detail given the dataset's size.

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

class WikipediaDataset(Dataset):
    def __init__(self, dataset, tokenizer):
        self.dataset = dataset
        self.tokenizer = tokenizer

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

    def __getitem__(self, idx):
        # Extract input_ids and attention_mask from the dataset
        item = self.dataset[idx]
        # Convert lists to PyTorch tensors
        return {
            'input_ids': torch.tensor(item['input_ids'], dtype=torch.long),
            'attention_mask': torch.tensor(item['attention_mask'], dtype=torch.long)
        }

# Create an instance of the custom dataset
# Pass the tokenizer to the dataset
pytorch_dataset = WikipediaDataset(chunked_dataset, tokenizer)
```

Verifying saved file

We load the sample_processed_data.py file and check the first batch. It should have two keys “input_ids” and “attention_mask”, and their values should be vectors. The snapshot confirms it’s correct.

Check file

```python
import torch

# Load the saved file
sampled_batches = torch.load("sample_processed_data.pt", weights_only=False)

print(f"Loaded {len(sampled_batches)} batches from file.")

# Check structure of the first batch
first_batch = sampled_batches[0]
print("Type of batch:", type(first_batch))
print("Batch content:", first_batch)
```

[4] Python

```
Loaded 5 batches from file.
Type of batch: <class 'transformers.tokenization_utils_base.BatchEncoding'>
Batch content: {'input_ids': tensor([[  101,  1996,  2208,  ...,     0,     0,     0],
        [  101, 20756,  2022,  ...,     0,     0,     0],
        [  101, 18461,  2011,  ...,  1010,  5893,   102],
        ...,
        [  101,  2146,  1011,  ..., 15428,  2000,   102],
        [  101,  2444,  1999,  ...,     0,     0,     0],
        [  101, 19614,  1997,  ...,     0,     0,     0]]), 'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 1, 1, 1],
        ...,
        [1, 1, 1,  ..., 1, 1, 1],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0]])}
```

Challenges Encountered

- **Memory Overflow on Colab:** The most significant challenge was the sheer size of the raw Wikipedia data. This was mitigated by using Hugging Face's `streaming=True` feature, which allowed us to process a large dataset iteratively without requiring a massive amount of RAM or disk space. Also, we shrank to 150 documents instead of the full 1GB raw data, so that we could get the code correct.
- **Data shape expected by from_list() method.** This method expects a list of dictionaries, but when we retrieved 'text' key from Wikipedia, it's a list of strings.

Reflections on Preprocessing Impact

The preprocessing pipeline is a foundational step that directly impacts the quality and efficiency of model pre-training. By carefully cleaning and normalizing the data, we provide the model with a consistent, high-quality signal, reducing the noise it must learn to ignore. The chunking and padding strategies, while complex to implement, were essential for making the training process tractable and memory-efficient. Ultimately, a well-preprocessed dataset is the bedrock upon which a robust and performant foundation model is built.