# 1.Despite spending so much time on this script it failed due to lack of proper compute understanding.

## Code

"""

NOTE:

1.⚖️ Scale Jump

From 100M → 5.3B tokens.

That's ~53X more data.

"""

import json

import os

from azure.storage.blob import BlobServiceClient

from datasets import load\_dataset

from tqdm import tqdm

from dotenv import load\_dotenv

load\_dotenv()

# Config

# Note: C4 is a good choice for GPT-2 replication. Alternatives: OpenWebText, The Pile, RedPajama

# Our 117M parameter model needs ~2.3B tokens (Chinchilla optimal), 20GB provides ~5B tokens

DATASET\_NAME = "allenai/c4"

DATASET\_CONFIG = "en"

SPLIT = "train"

TARGET\_SIZE\_GB = 20

BLOB\_NAME = "c4\_20gb.jsonl"

# Quality filter settings

# Note: In production, add extensive filtering for PII, violence, sexual content, toxicity

MIN\_TEXT\_LENGTH = 10

MAX\_TEXT\_LENGTH = 100000

# Azure connection

conn\_str = os.getenv("AZURE\_STORAGE\_CONNECTION\_STRING")

container\_name = os.getenv("CONTAINER\_NAME")

class DatasetProcessor:

    def \_\_init\_\_(self, conn\_str, container\_name, target\_size\_gb, blob\_name):

        self.conn\_str = conn\_str

        self.container\_name = container\_name

        self.target\_size\_gb = target\_size\_gb

        self.blob\_name = blob\_name

        self.limit\_bytes = target\_size\_gb \* (1024\*\*3)

    def is\_good\_text(self, text):

        """Filter for text quality"""

        if not text or not text.strip():

            return False

        if len(text) < MIN\_TEXT\_LENGTH:

            return False

        if len(text) > MAX\_TEXT\_LENGTH:

            return False

        return True

    def data\_generator(self, dataset):

        """Generator that yields filtered data as bytes"""

        self.bytes\_written = 0

        self.docs\_processed = 0

        self.docs\_accepted = 0

        pbar = tqdm(

            desc="Processing",

            unit="docs",

            postfix={'GB': '0.00', 'accepted': '0%'}

        )

        try:

            for sample in dataset:

                self.docs\_processed += 1

                text = sample.get("text", "").strip()

                if not self.is\_good\_text(text):

                    pbar.update(1)

                    continue

                self.docs\_accepted += 1

                line = json.dumps(sample, ensure\_ascii=False) + '\n'

                line\_bytes = line.encode('utf-8')

                self.bytes\_written += len(line\_bytes)

                if self.docs\_accepted % 100 == 0:

                    accept\_rate = (self.docs\_accepted / self.docs\_processed) \* 100

                    pbar.set\_postfix({

                        'GB': f'{self.bytes\_written/1024\*\*3:.2f}',

                        'accepted': f'{accept\_rate:.1f}%'

                    })

                pbar.update(1)

                yield line\_bytes

                if self.bytes\_written >= self.limit\_bytes:

                    print(f"\nReached {self.target\_size\_gb}GB with {self.docs\_accepted:,} documents")

                    break

        finally:

            pbar.close()

    def process\_and\_upload(self, dataset\_name, dataset\_config, split):

        """Process dataset and stream directly to Azure Blob"""

        try:

            print(f"Loading data from {dataset\_name} until {self.target\_size\_gb}GB...")

            dataset = load\_dataset(dataset\_name, dataset\_config, split=split, streaming=True)

        except Exception as e:

            print(f"Error loading dataset: {e}")

            return False

        try:

            print("Streaming JSONL directly to Azure Blob...")

            blob\_client = BlobServiceClient.from\_connection\_string(self.conn\_str).get\_blob\_client(

                container=self.container\_name,

                blob=self.blob\_name

            )

        except Exception as e:

            print(f"Error connecting to Azure: {e}")

            return False

        try:

            # Single upload using generator - streams data without loading into memory

            print("Starting upload to Azure...")

            blob\_client.upload\_blob(self.data\_generator(dataset), overwrite=True)

            print("Upload completed!")

            print(f"Successfully uploaded to {self.blob\_name}")

            print(f"Final stats: {self.docs\_accepted:,} documents accepted from {self.docs\_processed:,} processed")

            print(f"Total size: {self.bytes\_written/1024\*\*3:.2f}GB")

            return True

        except Exception as e:

            print(f"Error during upload: {e}")

            return False

# Usage

processor = DatasetProcessor(conn\_str, container\_name, TARGET\_SIZE\_GB, BLOB\_NAME)

success = processor.process\_and\_upload(DATASET\_NAME, DATASET\_CONFIG, SPLIT)

if not success:

    print("Processing failed!")

## Bottlenecks

1. **CPU bottleneck — Tokenization + JSON decoding**

For every line of the JSONL file, Python must:

1. Parse JSON (json.loads(line))
2. Extract "text"
3. Tokenize it using GPT-2’s BPE tokenizer
4. Convert the list of ints to np.int32
5. **Network I/O bottleneck — Azure streaming**

You’re reading data from Azure Blob (download\_blob().chunks())  
and simultaneously writing tokenized data back to Azure (upload\_blob(generator)).

3. **Streaming back-pressure — Generator + Azure SDK**

**What happens:**  
You gave upload\_blob() a generator that yields token bytes.  
Internally, Azure buffers chunks before sending; if that buffer fills faster than it can upload, it stops pulling new data from your generator.

So thinking from first principles I think we need to ship fast I have disk space in d drive lets use it ?

So we will migrate the entire operation to run locallly.

Change 1-

1.We will use fast tokenizer which is written in rust.

2.

"""

Local tokenizer - downloads JSONL from Azure blob, tokenizes locally, saves train.bin/val.bin

CHANGES FROM v0 (Original Azure streaming tokenizer):

Performance Optimizations:

1. Fast Tokenizer: Switched from GPT2Tokenizer to GPT2TokenizerFast

   - 5-10x faster tokenization using Rust backend instead of pure Python

2. Batch Processing: Tokenizes 500 documents at once instead of one-by-one

   - Reduces Python ↔ Rust boundary crossings from millions to thousands

   - Enables internal parallelization in Rust code

   - Expected speedup: 5-10x (total ~1-1.5 hours vs 8-10 hours for 20GB)

3. Single-Pass Processing: Reads input file once instead of twice

   - v0 streamed from Azure twice (once for train, once for val)

   - v1 downloads once, processes once, writes to both files simultaneously

   - 50% reduction in I/O and processing time

Architecture Changes:

4. Download-Then-Process: Downloads complete file locally before tokenizing

   - v0: Streamed from Azure during tokenization (network-dependent)

   - v1: Download first (resumable), then process locally (network-independent)

   - Eliminates network interruptions during tokenization

5. Resume-Safe Downloads: Partial downloads are preserved and resumed

   - Checks existing file size and resumes from last byte

   - Uses Azure range requests (offset + length parameters)

   - Prevents restarting 20GB downloads from scratch

6. Simplified Architecture: Removed streaming complexity

   - No GeneratorStream wrapper class needed

   - No retry logic during tokenization (only during download)

   - Cleaner, more maintainable code

Trade-offs:

- Requires local disk space (~20GB for input + output files)

- Download step adds upfront time (but only once, and resumable)

- Better for unstable networks, worse if disk space is limited

"""