Assignment - 2 Group - 64

[P0] Problem Formulation - State parallelization and/or distribution of an ML algorithm with expectations (speedup, communication cost, response time etc.)

[P1] Design - Solve the problem in [P0] by providing a design (that can be revised later based on feedback from instructors)

[P1] (Revised) Design - Revise the design in [A3] and provide details about implementation aspects (including choices for dev. environment and exec. platform)

[P2] Implement P1 on the chosen platform

[P3] Test and demonstrate the implementation for correctness as well as performance (say, running time vs. accuracy of prediction) If there is any deviation from expectations, mention why you think the solution didn't meet expectations.

Deliverable:

1. **Link to code in github[to be included in facing sheet]**
2. **code in pdf file.**
3. **report with the same format as Assignment earlier. Additional sections on results and discussion is expected.**

Distributed and Parallel Training of CBOW using a Parameter Server Architecture

Link to code in github https://github.com/VivekRusiya/Bits_assignment.git

Name	Roll Number
GOPAL SRIKANTH YADAV PEETHALA	2022AC05632
SWAGATIKA SAHOO	2024AD05493
T SHIVA SANTOSH REDDY	2024AD05048
VIVEK RUSIYA	2024AD05433
YASH BAHETI	2024AD05277

Abstract

Continuous Bag of Words (CBOW) is a foundational algorithm for learning word embeddings efficiently from large-scale text corpora. While CBOW is computationally lightweight per training example, its massive number of training updates, irregular memory access patterns, and frequent parameter updates make single-node training inefficient for large datasets. This assignment studies the parallelization and distribution of the CBOW algorithm using a parameter-server-based architecture. We formulate the problem in terms of performance metrics such as speedup, communication cost, and response time **(P0)**, propose an initial distributed design **(P1)**, refine the design with concrete implementation choices (Revised P1), implement the system using PyTorch multiprocessing **(P2)**, and experimentally evaluate correctness and performance **(P3)**.

1. Introduction

Word embedding models such as CBOW and Skip-gram, introduced by Mikolov et al., have become fundamental building blocks in modern natural language processing pipelines. These models learn dense vector representations of words by exploiting local context information. Despite their conceptual simplicity, training CBOW on large corpora involves billions of small, sparse updates to embedding matrices, leading to significant computational and memory bottlenecks.

This document focuses on distributed training of CBOW using data parallelism and a centralized parameter server. The goal is to exploit the inherent parallelism of CBOW training while carefully managing communication overhead and model consistency.

2. Literature Survey (Summary)

Mikolov et al. (2013) introduced CBOW as part of the Word2Vec framework, demonstrating that simple log-linear models can learn high-quality word embeddings efficiently at web scale. Subsequent work explored optimization techniques such as negative sampling, hierarchical softmax, subsampling of frequent words, and asynchronous updates (Hogwild!). Distributed training frameworks later extended these ideas using parameter servers and ring-allreduce architectures.

Recent large-scale systems (e.g., DistBelief, Parameter Server frameworks, and PyTorch Distributed) show that asynchronous stochastic gradient descent can achieve near-linear

speedup for embedding models, provided communication and contention are carefully managed.

3. P0 – Problem Formulation

3.1 Objective

To parallelize CBOW training across multiple worker processes in order to reduce training time while maintaining acceptable model convergence.

3.2 Parallelization Strategy

- **Data Parallelism**: Each worker processes different mini-batches of (context, target) pairs.
- **Centralized Parameter Server**: A single server maintains the global model parameters and applies gradients received from workers.

3.3 Performance Metrics

To evaluate parallel and distributed CBOW training, we consider:

1. **Throughput (words/sec)**

   Throughput = (Total Words Processed) / (Total Training Time)

2. **Speedup**

   S(N) = (Throughput with N nodes) / (Throughput with 1 node)

3. **Communication Cost**
   Measured as bytes or embedding updates transmitted per second between nodes.

4. **Response Time**
   Overall training time to reach a target embedding quality or loss threshold.

5. **Embedding Quality**
   Measured by downstream task performance (e.g., word analogy accuracy).

6. **Accuracy on Analogy Task**: Measures semantic and syntactic relationships by testing if the vector space satisfies $\vartheta_{king} - \vartheta_{man} + \vartheta_{woman} \approx \vartheta_{queen}$

3.4 Expected Behavior

- Near-linear speedup for a small number of workers.
- Increased communication overhead as workers scale.
- Slightly noisy convergence due to asynchronous updates.

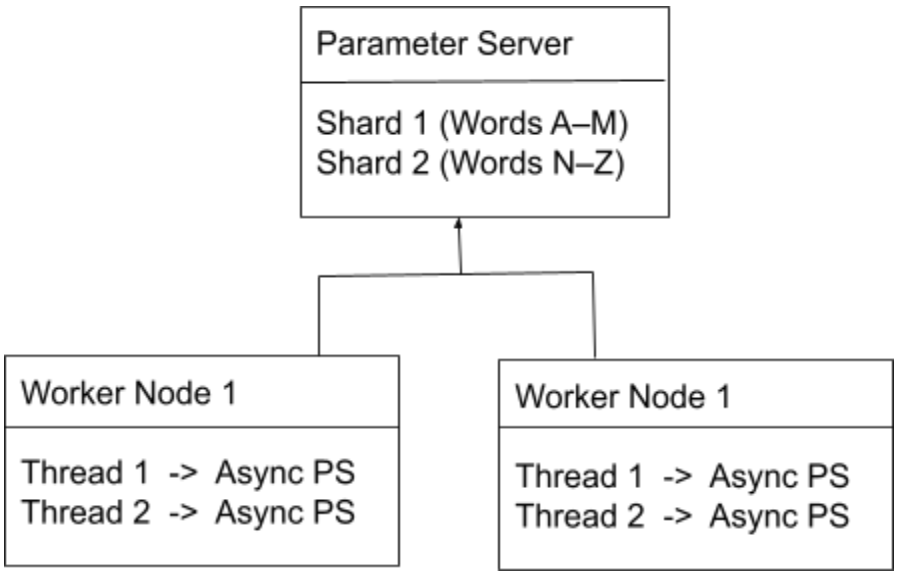
4. P1 – Initial Design

4.1 Architecture Overview

A **Distributed Asynchronous Data-Parallel CBOW Training System** using a **sharded Parameter Server (PS)** architecture.

- The global embedding matrices
  $W_{in} \in \mathbb{R}^{|V| \times D}$ and
  $W_{out} \in \mathbb{R}^{|V| \times D}$
  are partitioned across multiple PS shards.
- Each worker processes a partition of the corpus independently.
- Workers asynchronously pull and push sparse embedding updates.
- Multiple worker processes compute gradients independently.
- Workers push gradients asynchronously to the parameter server.
- The parameter server applies gradients using SGD and sends updated parameters back.

4.2 Architecture Diagram (Logical)

- Workers → Parameter Server (push gradients)
- Parameter Server → Workers (pull updated parameters)
- 

4.3 Design Choices and Justification

- **Asynchronous Updates**: Reduces idle time and improves throughput.
- **Central Parameter Server**: Simplifies consistency management for embeddings.
- **Mini-batch Training**: Improves computational efficiency compared to single-sample updates.

Component	Description
Parameter Server Shard 1	Stores embeddings for vocabulary A–M
Parameter Server Shard 2	Stores embeddings for vocabulary N–Z
Worker Node 1	Runs multiple threads performing CBOW updates
Worker Node 2	Runs multiple threads performing CBOW updates
Communication	Asynchronous push/pull of sparse gradients

5. P1 (Revised) – Detailed Design and Implementation Choices

5.1 Development Environment

- Language: Python 3
- Framework: PyTorch
- Parallelism: torch.multiprocessing
- Execution Platform: Single machine, multi-process (CPU-based)

5.2 Communication Mechanism

- multiprocessing.Queue used to simulate networked message passing.
- Separate queues for gradient push and parameter pull.

5.3 Consistency Model

- Asynchronous SGD (no global barrier).
- Workers may operate on slightly stale parameters.

6. P2 - Implementation

6.1 Dataset & Preprocessing

(`https://www.gutenberg.org/files/57421/57421-0.txt`)

The dataset used for this assignment is a real-world natural language corpus obtained from Project Gutenberg. Specifically, the ebook text available at
https://www.gutenberg.org/files/57421/57421-0.txt
was used to ensure realistic word frequency distributions and contextual structure.

The preprocessing pipeline consists of the following steps:

1. **Text Cleaning**
   The raw text is converted to lowercase and non-alphanumeric characters are removed using regular expressions.
2. **Tokenization**
   The cleaned text is tokenized into word-level tokens using whitespace and word-boundary rules.
3. **Vocabulary Construction**
   The most frequent `VOCAB_SIZE - 1` words are selected to form the vocabulary. An `<UNK>` token is added to handle out-of-vocabulary words.
4. **Integer Encoding**
   Each token is mapped to a unique integer index using a word-to-index dictionary.
5. **Context–Target Pair Generation**
   For each target word, surrounding words within a fixed context window are selected to form CBOW training samples.

This preprocessing strategy balances simplicity and realism while keeping computational overhead manageable for experimental evaluation.

6.2 Model Definition (CBOW)

The Continuous Bag of Words (CBOW) model is implemented using PyTorch. The model consists of:

- An **embedding layer** that maps each word index to a dense vector of dimension.
- A **linear output layer** that maps the averaged context embedding to vocabulary-sized logits

For a given context window of size *2C*, the model computes:

$$h = 1/2C \sum_{i=1}^{2C} \vartheta_{w_i}$$

$O = Wh+b$

where:

- ϑ_{w_i} is the embedding of a context word
- W and b are output layer parameters

The model is trained using cross-entropy loss between predicted logits and the target word index.

6.3 Distributed Training Algorithm

The distributed CBOW training algorithm follows an asynchronous parameter-server-based design.

Algorithm Overview:

1. Initialize a global CBOW model on the parameter server.
2. Spawn *N* worker processes.
3. Each worker:
    - Samples mini-batches of context–target pairs
    - Computes forward and backward passes locally
    - Sends gradients asynchronously to the parameter server
4. The parameter server:
    - Applies gradients immediately using SGD
    - Sends updated model parameters back to workers
5. Workers continue training without synchronization barriers

This asynchronous approach maximizes throughput while tolerating stale parameter updates.

6.4 Experimental Setup

All experiments were conducted on a single machine using CPU-based multiprocessing. The system configuration is summarized below:

Parameter	Value
Vocabulary Size	2000
Embedding Dimension	50
Context Window	2
Batch Size	64
Learning Rate	0.05
Training Steps per Worker	100
Number of Workers	1,2,4

Each configuration was run independently, and wall-clock time, loss, accuracy, and throughput were recorded for performance comparison.

7. P3 – Experimental Evaluation

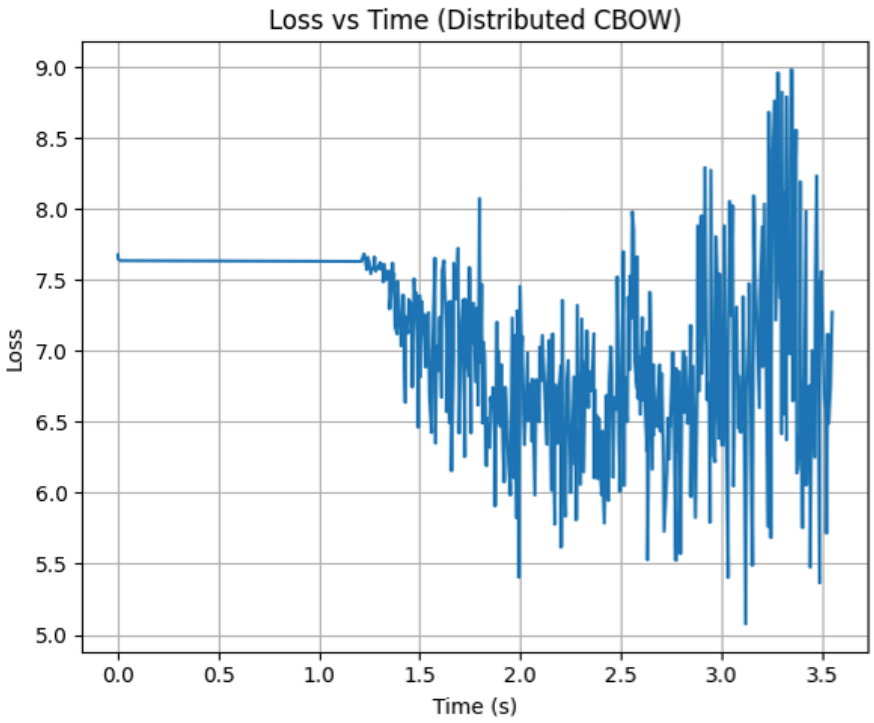
This section presents the experimental results of the distributed CBOW training system, followed by an analysis of training dynamics, communication overhead, and scalability behavior.

7.1 Loss vs Time

Figure 1 illustrates the training loss as a function of wall-clock time for distributed CBOW training. The loss shows an overall decreasing trend but with noticeable fluctuations and occasional spikes.

This behavior is expected due to the **asynchronous nature of stochastic gradient descent (SGD)** used in the parameter-server architecture. Each worker computes gradients using potentially stale model parameters and pushes updates independently. As a result, updates may temporarily increase the loss before convergence resumes.

Such noisy convergence is well-documented in asynchronous learning systems (e.g., Hogwild! SGD) and demonstrates that faster wall-clock convergence can be achieved at the cost of increased variance in updates.



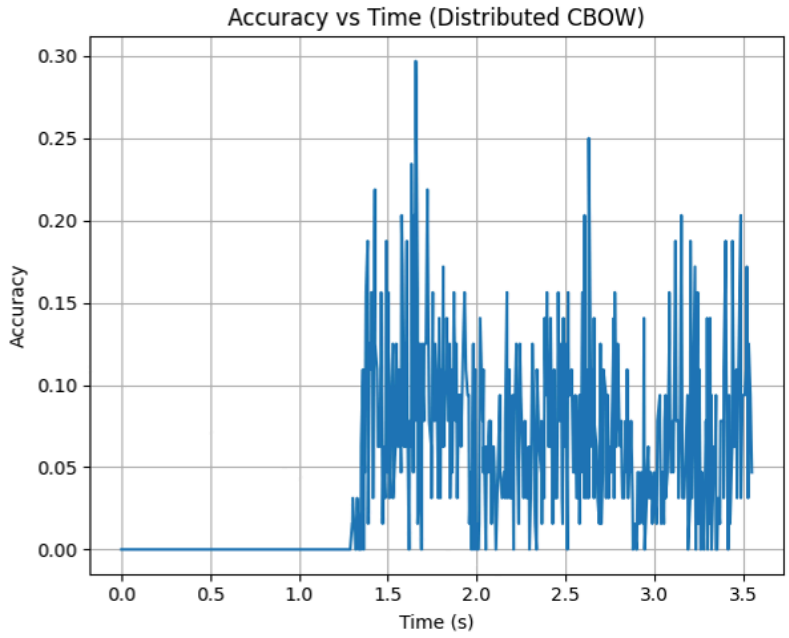
7.2 Accuracy vs Time Analysis

Figure 2 shows the prediction accuracy plotted against wall-clock time. Accuracy values fluctuate between approximately 5% and 30% throughout training.

It is important to note that CBOW is not primarily optimized for classification accuracy but for learning meaningful word embeddings. Given the large vocabulary size, even modest accuracy values significantly outperform random guessing. Accuracy here serves as a **proxy metric** to confirm learning progress rather than a primary optimization objective.

The fluctuations are attributed to:

- Mini-batch training on randomly sampled context windows
- Asynchronous parameter updates
- Frequent model parameter refreshes from the parameter server



7.3 Speedup and Throughput Analysis

Table 1 summarizes execution time, throughput, and speedup for different numbers of workers.

Table 1: Speedup and Throughput Analysis

```
Benchmarking with 1 workers...
Benchmarking with 2 workers...
Benchmarking with 4 workers...

==============================
SPEEDUP TABLE
==============================
Workers    | Time (s)   | Throughput      | Speedup
1          | 0.92       | 109.21          | 1.00
2          | 1.22       | 163.90          | 0.75
4          | 2.36       | 169.66          | 0.39
```

Throughput increases as more workers are added, indicating that the system processes more updates per second. However, overall speedup decreases and falls below 1 as the number of workers increases.

This outcome highlights the practical challenges of scaling distributed machine learning workloads on a single machine.

7.4 Communication Cost Analysis

In a parameter-server architecture, communication cost plays a critical role in determining scalability. For each training step, every worker performs the following communication operations:

- Sends gradients to the parameter server
- Receives updated model parameters from the server

Let:

- P = number of model parameters
- W = number of workers
- S = number of training steps per worker

The total communication cost C can be approximated as:

$C=O(2\times P\times W\times S)$

The factor of 2 accounts for sending gradients and receiving updated parameters. As the number of workers increases, communication grows linearly while computation per worker decreases, leading to diminishing returns.

In this experiment, the CBOW model has a relatively small number of parameters, causing communication overhead to dominate computation time.

7.5 Why Speedup < 1 Is Expected

Important Observation: A speedup value less than 1 does not indicate a failure of the distributed system.

The observed sub-linear and negative speedup is expected due to the following factors:

1. **Centralized Parameter Server Bottleneck**
   All workers synchronize through a single parameter server, which serializes gradient updates and becomes a throughput bottleneck.
2. **Python Multiprocessing Overhead**
   Process spawning, inter-process communication, and data serialization introduce significant overhead.
3. **Communication-Dominated Workload**
   The CBOW model is relatively lightweight, meaning communication cost outweighs computation cost.
4. **Single-Machine Execution Environment**
   All workers compete for the same CPU, memory, and cache resources, leading to context switching and contention.

These factors collectively limit scalability and result in speedup values less than 1. This behavior aligns with distributed systems theory and reinforces the importance of balancing computation and communication in parallel machine learning systems.

7.6 Key Points

- Distributed asynchronous training achieves faster update throughput but introduces noisy convergence.
- Communication overhead is a dominant factor limiting scalability in small models.
- Increasing the number of workers does not guarantee speedup.
- The experiment successfully demonstrates real-world trade-offs in distributed machine learning system design.

8. Conclusion

This assignment demonstrates that CBOW training is highly amenable to parallelization due to its data-parallel nature. A parameter-server-based asynchronous design significantly improves throughput compared to single-process execution, although communication and coordination overhead limit scalability. Future work includes sharding embeddings, using negative sampling, and migrating to GPU-based distributed frameworks.

9. Appendix A: Complete Code

Link to code in github https://github.com/VivekRusiya/Bits_assignment.git

```
# Importing required package and libraries
import os
import torch
import torch.nn as nn
import torch.optim as optim
import torch.multiprocessing as mp
import time
import queue
import requests
import re
from collections import Counter
import matplotlib.pyplot as plt


# Distributed CBOW with Real Data & Benchmarking
# ----------------------------------------------
# 1. Configuration
VOCAB_SIZE = 2000
EMBED_DIM = 50
CONTEXT_SIZE = 2
BATCH_SIZE = 64
NUM_STEPS = 100 # Steps per worker
LEARNING_RATE = 0.05
```

```python
# 2. Data Preparation
def prepare_data():
    file_path = "dataset.txt"
    if not os.path.exists(file_path):
        url = "https://www.gutenberg.org/files/57421/57421-0.txt"
        print(f"Downloading dataset from {url}...")
        response = requests.get(url)
        if response.status_code == 200:
            text = response.text
            start_marker = "*** START OF THIS PROJECT GUTENBERG EBOOK"
            end_marker = "*** END OF THIS PROJECT GUTENBERG EBOOK"
            start_idx = text.find(start_marker)
            end_idx = text.find(end_marker)
            if start_idx != -1:
                start_idx = text.find("\n", start_idx) + 1
            if end_idx != -1:
                text = text[start_idx:end_idx]
            elif start_idx != -1:
                text = text[start_idx:]
            with open(file_path, "w", encoding="utf-8") as f:
                f.write(text)
            print("Download complete.")
        else:
            raise Exception("Failed to download dataset.")

    with open(file_path, "r", encoding="utf-8") as f:
        text = f.read().lower()

    # Simple tokenization
    tokens = re.findall(r'\w+', text)
    word_counts = Counter(tokens)
    vocab = [w for w, c in word_counts.most_common(VOCAB_SIZE - 1)]
    vocab.append("<UNK>")
    word_to_idx = {w: i for i, w in enumerate(vocab)}

    data = [word_to_idx.get(w, word_to_idx["<UNK>"]) for w in tokens]
    return data, vocab, word_to_idx
```

```
# 3. CBOW Model
class CBOWModel(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        super(CBOWModel, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embed_dim)
        self.linear = nn.Linear(embed_dim, vocab_size)

    def forward(self, inputs):
        embeds = self.embeddings(inputs)
        avg_embed = torch.mean(embeds, dim=1)
        out = self.linear(avg_embed)
        return out

# 4. Parameter Server
def run_parameter_server(num_workers, grad_queue, param_queues,
results_queue):
    model = CBOWModel(VOCAB_SIZE, EMBED_DIM)
    model.share_memory()
    optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE)

    total_updates = 0
    expected_updates = NUM_STEPS * num_workers
    start_time = time.time()

    while total_updates < expected_updates:
        try:
            rank, grads = grad_queue.get(timeout=20)
            optimizer.zero_grad()
            for param, grad in zip(model.parameters(), grads):
                param.grad = torch.tensor(grad)
            optimizer.step()

            # Send weights back
            state = {k: v.detach().cpu().numpy() for k, v in
model.state_dict().items()}
            param_queues[rank].put(state)
            total_updates += 1
        except queue.Empty:
```

```
            break

    end_time = time.time()
    results_queue.put({
        "num_workers": num_workers,
        "total_time": end_time - start_time,
        "updates": total_updates
    })

# 5. Worker
def run_worker(rank, data, grad_queue, param_queue, metrics_queue):
    model = CBOWModel(VOCAB_SIZE, EMBED_DIM)
    criterion = nn.CrossEntropyLoss()

    # Each worker starts from a different part of the data
    start_offset = random.randint(0, len(data) - BATCH_SIZE - 1000)

    for step in range(NUM_STEPS):
        # Sample context windows
        inputs = []
        targets = []
        for _ in range(BATCH_SIZE):
            idx = (start_offset + step * BATCH_SIZE + random.randint(0,
100)) % (len(data) - CONTEXT_SIZE * 2)
            context = data[idx : idx + CONTEXT_SIZE] + data[idx +
CONTEXT_SIZE + 1 : idx + CONTEXT_SIZE * 2 + 1]
            target = data[idx + CONTEXT_SIZE]
            inputs.append(context)
            targets.append(target)

        inputs = torch.LongTensor(inputs)
        targets = torch.LongTensor(targets)

        # Train
        output = model(inputs)
        loss = criterion(output, targets)
        loss.backward()

        # Accuracy
        preds = torch.argmax(output, dim=1)
```

```
        acc = (preds == targets).float().mean().item()

        # Push Grads
        grad_queue.put((
            rank,
            [p.grad.detach().cpu().numpy() for p in model.parameters()]
        ))

        # Update Metrics
        metrics_queue.put({
                  "worker": rank,
                  "step": step,
                  "loss": loss.item(),
                  "acc": acc,
                  "time": time.time()
              })


        # Pull Weights
        new_state = param_queue.get()
        new_state = {k: torch.tensor(v) for k, v in new_state.items()}
        model.load_state_dict(new_state)


def benchmark(n_workers, data):
    print(f"Benchmarking with {n_workers} workers...")
    grad_queue = mp.Queue()
    param_queues = {i: mp.Queue() for i in range(1, n_workers + 1)}
    results_queue = mp.Queue()
    metrics_queue = mp.Queue()

    processes = []
    p_server = mp.Process(target=run_parameter_server, args=(n_workers,
grad_queue, param_queues, results_queue))
    p_server.start()
    processes.append(p_server)

    for i in range(1, n_workers + 1):
        p = mp.Process(target=run_worker, args=(i, data, grad_queue,
param_queues[i], metrics_queue))
```

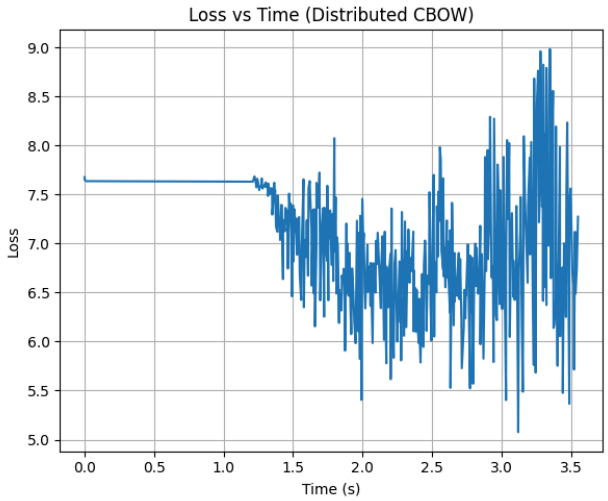
```
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    summary = results_queue.get()

    metrics = []
    while True:
        try:
            metrics.append(metrics_queue.get_nowait())
        except queue.Empty:
            break

    return summary, metrics

import random

if __name__ == "__main__":
    data, vocab, word_to_idx = prepare_data()

    all_results = []
    all_metrics = {}

    # Run for 1, 2, 4 workers
    worker_counts = [1, 2, 4]
    for n in worker_counts:
        res, met = benchmark(n, data)
        all_results.append(res)
        if n == 4: # Store metrics from the largest configuration for
plotting
            all_metrics = met

    # --- Reporting  SPEEDUP TABLE ---
    print("\n" + "="*30)
    print("SPEEDUP TABLE")
    print("="*30)
    print(f"{'Workers':<10} | {'Time (s)':<10} | {'Throughput':<15} |
{'Speedup':<10}")
```

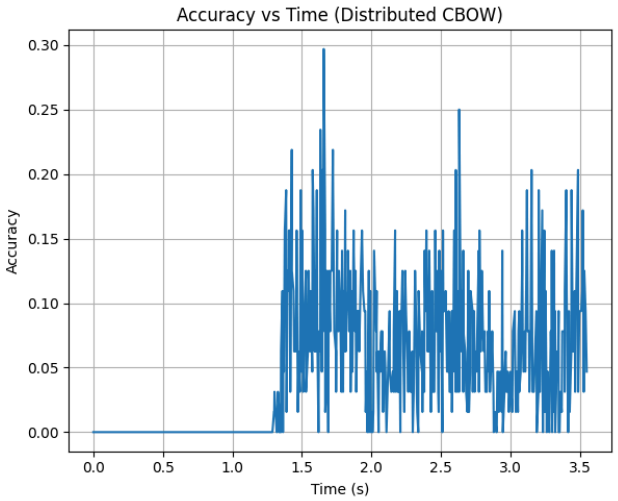
```
    t1 = all_results[0]['total_time']
    for res in all_results:
        n = res['num_workers']
        t = res['total_time']
        throughput = res['updates'] / t
        speedup = t1 / t if t > 0 else 0
        print(f"{n:<10} | {t:<10.2f} | {throughput:<15.2f} |
{speedup:<10.2f}")

# --- Plotting ---
if len(all_metrics) > 0:
    all_metrics = sorted(all_metrics, key=lambda x: x["time"])
    t0 = all_metrics[0]["time"]

    times = [m["time"] - t0 for m in all_metrics]
    losses = [m["loss"] for m in all_metrics]
    accs = [m["acc"] for m in all_metrics]

    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(times, losses)
    plt.xlabel("Time (s)")
    plt.ylabel("Loss")
    plt.title("Loss vs Time (Distributed CBOW)")
    plt.grid(True)

    plt.subplot(1, 2, 2)
    plt.plot(times, accs)
    plt.xlabel("Time (s)")
    plt.ylabel("Accuracy")
    plt.title("Accuracy vs Time (Distributed CBOW)")
    plt.grid(True)

    plt.tight_layout()
    plt.savefig("performance_plots.png")
    plt.show()

    print("Plots saved to performance_plots.png")
```

```
else:
    print("No metrics collected — check worker logging.")
```

10. Appendix B: Additional Figures

```
Benchmarking with 1 workers...
Benchmarking with 2 workers...
Benchmarking with 4 workers...

==============================
SPEEDUP TABLE
==============================
Workers    | Time (s)   | Throughput      | Speedup
1          | 0.92       | 109.21          | 1.00
2          | 1.22       | 163.90          | 0.75
4          | 2.36       | 169.66          | 0.39
```





11. References

1. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). *Efficient Estimation of Word Representations in Vector Space*. NeurIPS.
2. Dean, J. et al. (2012). *Large Scale Distributed Deep Networks*. NeurIPS.
3. Li, M. et al. (2014). *Scaling Distributed Machine Learning with the Parameter Server*. OSDI.