**Harnessing true potential of Multi-Threading in Machine Learning Algorithms**

Raunak Nandkumar More

*Department of Computer Science*

*University of South Dakota*

**ABSTRACT:**

This study delves into the impact of tiny work units called threads on program performance. It explores effective thread utilization, enabling multiple programs to run concurrently within a single thread. Parallel computing is highlighted for its role in simultaneous program execution, crucial for tasks like visual rendering, multitasking, and managing diverse input/output operations. The study addresses seamless collaboration between threads through synchronization, preventing errors, conflicts, or stalls.

The study highlights the role of a thread as the smallest worker in a program, working collaboratively with other threads to achieve a common goal. The concept of "speedup" is introduced, measuring the acceleration of a program when employing multiple threads compared to a single thread. While more threads can enhance speed, particularly with extensive datasets, caution is emphasized due to challenges such as data sharing, increased communication workload, and the need for optimal program design to accommodate multiple threads. The study further explores challenges associated with large datasets and the potential for acceleration using multiple threads. Key challenges include managing data distribution, ensuring efficient communication between threads, and considering the design aspects of programs dealing with substantial datasets.

We used these tools on three types of machine learning programs: logistic regression, MiniBatchKMeans, and Convolutional Neural Networks (CNN). We also use clever methods with logistic regression and MiniBatchKMeans and use both multiprocessing and multithreading to prepare data for CNN. The results help us manage many threads better, making them work well for big sets of data and improving how fast the program runs.

**INTRODUCTION:**

In the dynamic world of computer science, our focus lies on the powerful concept of multi-threading—a technique that allows a program to tackle multiple tasks at the same time, with a thread being the smallest unit of execution. This case study delves into the background and context of multi-threading, aiming to understand its role in improving how programs run, especially when dealing with substantial amounts of data.

**MOTIVATION:**

Our main goal is to unravel the impact of multi-threading on program performance, measured through a key metric called "speedup." This study seeks to explore the challenges associated with multi-threading, including data distribution, communication overhead, and the careful design of parallel algorithms. Additionally, our objectives extend to providing valuable insights into the efficient management of multiple threads, particularly when dealing with large datasets.

*Speedup* = *T*serial / ​*T*parallel

Where, Tparallel = execution time of the serial (single-threaded version)

Tserial = execution time of the parallel (multiple threaded version)

**SCOPE AND LIMITATIONS:**

The scope of this study encompasses the exploration of multi-threading multi-threading as a tool to enhance the performance of program execution, with a specific emphasis on large datasets. Challenges such as distributing data, managing communication overhead, and designing effective parallel algorithms are part of our investigation. However, it's important to acknowledge the automatic nature of multi-threading algorithms, limiting our control over data distribution and communication overhead. Despite these limitations, the study aspires to offer practical insights into managing multiple threads effectively for optimal performance and speedup.

Furthermore, we examine the practical application of threading libraries in Python. This involves parallelizing operations in three machine learning algorithms—Logistic Regression, Kmeans Clustering, and Convolutional Neural Networks (CNN). Ensemble methods with logistic regression and Kmeans Clustering, as well as the utilization of multiprocessing and multithreading for data preprocessing in CNN, provide additional dimensions to our exploration.

**METHODOLOGY:**

We will discuss methodologies for all 3 algorithms used i.e. Logistic Regression, MiniBatchKMeans, Convolutional Neural Networks.

**Logistic Regression:**

In the pursuit of enhancing the efficiency of logistic regression execution, our research drew inspiration from Ensemble techniques, leading to a significant reduction in execution time. This achievement is attributed to a carefully crafted methodology that encompasses various elements, including research design, data collection, and data analysis techniques.

**Methodology Overview:**

Our approach involved the creation of two synthetic datasets, one with 1000 data points and another with a substantial 10 million data points. To ensure the replicability of results, a consistent seed value was employed. The experimentation included varying the thread count from 1 to 24, with the logistic regression model set to run a maximum of 1000 iterations.

Drawing from Ensemble methodology, we divided the datasets into subsets corresponding to the number of threads designated for use. Subsequently, we aggregated the models, combining and averaging their coefficients and intercepts to construct a final model.

**Code Discussion:**

import threading

import numpy as np

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import time

import matplotlib.pyplot as plt

def train\_logistic\_regression(X\_train, y\_train, X\_test, y\_test, num\_threads):

start\_time = time.time()

models = []

threads = []

for \_ in range(num\_threads):

model = LogisticRegression(max\_iter=1000)

models.append(model)

# Split the data into equal parts for each thread

X\_train\_splits = np.array\_split(X\_train, num\_threads)

y\_train\_splits = np.array\_split(y\_train, num\_threads)

def train\_model\_thread(model, X, y):

model.fit(X, y)

# Create and start threads

for i in range(num\_threads):

thread = threading.Thread(target=train\_model\_thread, args=(models[i], X\_train\_splits[i], y\_train\_splits[i]))

threads.append(thread)

thread.start()

# Wait for all threads to finish

for thread in threads:

thread.join()

# Combine models and average coefficients

final\_model = LogisticRegression(max\_iter=1000)

final\_model.coef\_ = np.mean([model.coef\_ for model in models], axis=0)

final\_model.intercept\_ = np.mean([model.intercept\_ for model in models], axis=0)

# Fit the final model on the entire training set

final\_model.fit(X\_train, y\_train)

end\_time = time.time()

# Make predictions on the test set

y\_pred = final\_model.predict(X\_test)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

return accuracy, end\_time - start\_time

# Generate synthetic data for testing

np.random.seed(42)

X = np.random.rand(10000000, 10)

y = np.random.randint(2, size=10000000)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Define the range of thread counts

thread\_counts = [1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]

#thread\_counts = [1, 2, 4, 8, 12, 16, 32]

# Measure accuracy and time for different thread counts

accuracies = []

execution\_times = []

for num\_threads in thread\_counts:

accuracy, exec\_time = train\_logistic\_regression(X\_train, y\_train, X\_test, y\_test, num\_threads)

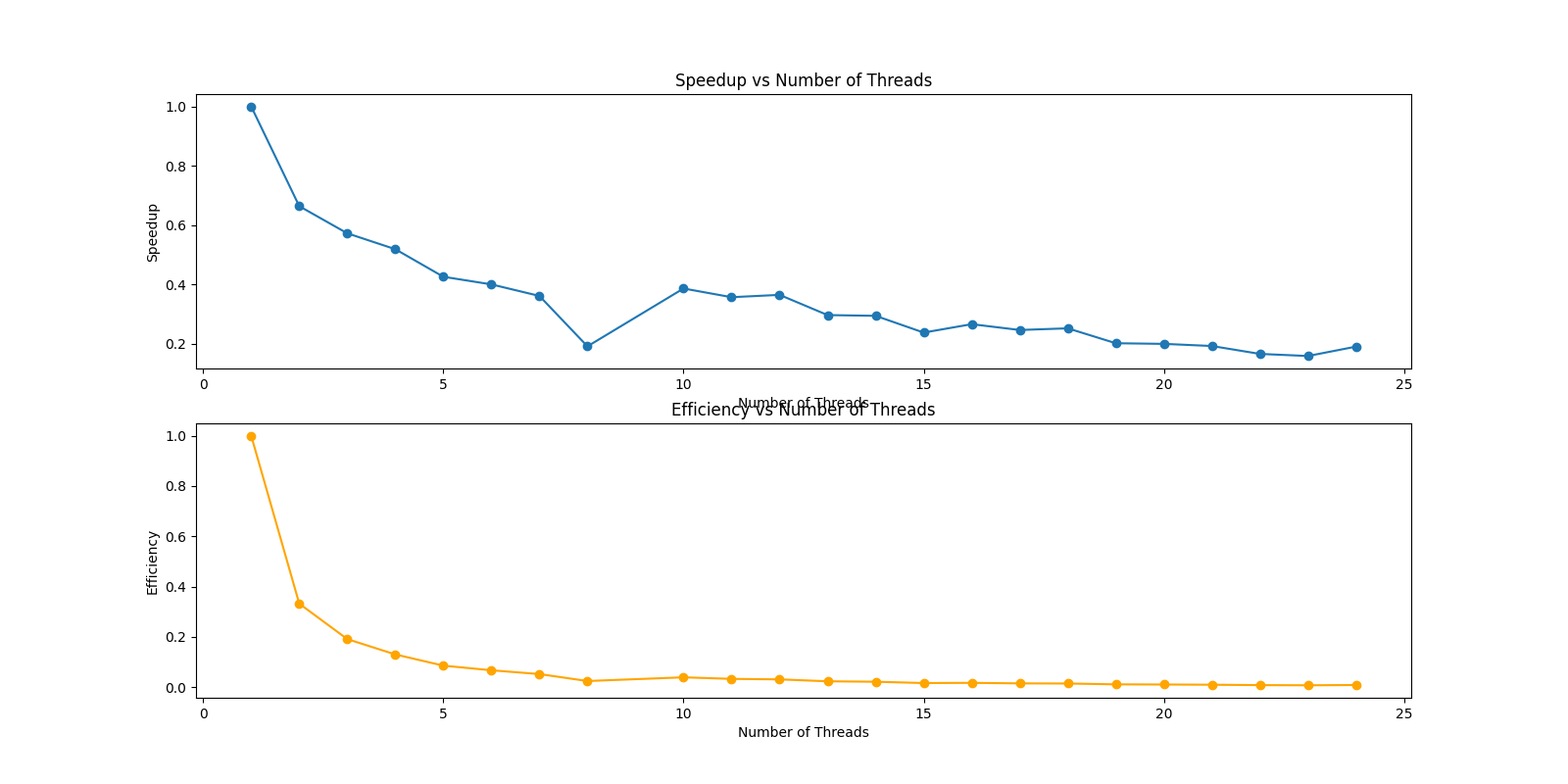
accuracies.append(accuracy)

execution\_times.append(exec\_time)

As seen in the code, after importing necessary libraries, we call the train\_logistic\_regression function that takes training and testing data along with the number of threads as input.Then Initializes a list of logistic regression models and a list of threads.Divides the training data into equal parts based on the number of threads.Defines a function train\_model\_thread to fit the logistic regression model on a subset of the data.Creates threads, each responsible for training a model on a subset of the data.Waits for all threads to finish.Combines the trained models by averaging their coefficients to create the final model.Fits the final model on the entire training set.Measures the time taken for training and evaluates the accuracy on the test set.

**Results and Analysis:**

Our focus extended to plotting Speedup and Efficiency against Thread count for both scenarios—1000 data points and 10 million data points.



When executing the model with 1000 data points, we observed a decrease in Speedup with an increased thread count. This phenomenon can be attributed to the overhead associated with thread creation, along with the additional burden of model creation and combination surpassing the training workload.

A graph of a graph with blue and orange lines

Description automatically generated with medium confidence

The results were markedly different when dealing with 10 million data points. Here, an increase in Speedup was observed with an elevated thread count, aligning with the theoretically expected outcome. This divergence highlights the nuanced impact of dataset size on the effectiveness of parallel processing, providing valuable insights for optimizing logistic regression models at scale.

**MiniBatchKMeans:**

The utilization of MiniBatchKMeans, employing mini-batches or subsets of randomly sampled input data, offers an effective means of parallelizing the algorithm. This distinctive characteristic lays the foundation for efficient parallel processing. In our study, we applied this approach to two synthetic datasets, one comprising 10,000 data points and the other an extensive 100 million data points. Maintaining consistency, we utilized the same seed value for result reproducibility.

**Methodology Overview:**

With thread counts ranging from 1 to 24 and a fixed number of clusters set at 3, our methodology involved addressing the challenge of non-thread-safe operations when multiple threads called the partial\_fit method on a shared kmeans object. To mitigate this issue, we implemented thread locks, ensuring that only one thread could update the kmeans object at any given time. The data splitting methods remained consistent with those previously discussed.

Unlike the logistic regression scenario, the Kmeans algorithm, working iteratively to refine clusters, did not require result aggregation. Consequently, our evaluation focused on plotting Speedup and Efficiency against Thread count for both scenarios.

**Code Discussion:**

import numpy as np

from sklearn.cluster import MiniBatchKMeans

import threading

import time

import matplotlib.pyplot as plt

# Generate synthetic data

np.random.seed(42)

X = np.random.rand(100000000, 2)

# Number of clusters

n\_clusters = 3

#Number of threads for parallel processing

#num\_threads\_list = [1, 2, 4, 8, 12, 16, 24, 28, 32]

num\_threads\_list = [1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]

# Function for K-Means clustering with multithreading

def kmeans\_thread(X\_thread, kmeans, lock):

with lock:

kmeans.partial\_fit(X\_thread)

# Function to measure execution time with multithreading

def measure\_time(X, num\_threads):

kmeans = MiniBatchKMeans(n\_clusters=n\_clusters, n\_init=3)

lock = threading.Lock()

X\_splits = np.array\_split(X, num\_threads)

threads = []

start\_time = time.time()

for i in range(num\_threads):

thread = threading.Thread(target=kmeans\_thread, args=(X\_splits[i], kmeans, lock))

threads.append(thread)

for thread in threads:

thread.start()

for thread in threads:

thread.join()

end\_time = time.time()

return end\_time - start\_time

# Benchmark with different numbers of threads

execution\_times = []

for num\_threads in num\_threads\_list:

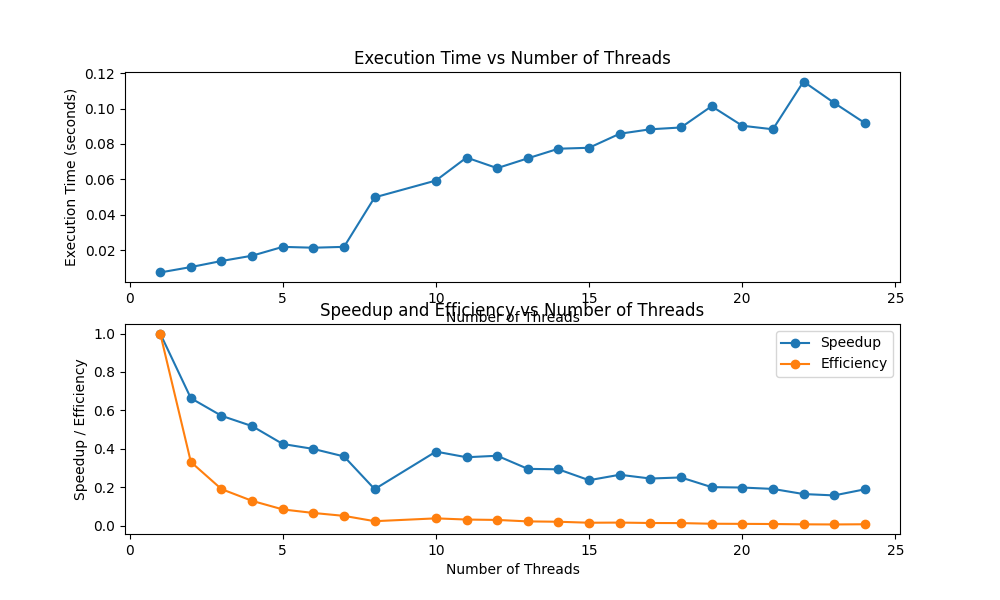
execution\_time = measure\_time(X, num\_threads)

execution\_times.append(execution\_time)

This code performs K-Means clustering on a large synthetic dataset using Mini-Batch K-Means algorithm with multithreading. It generates a synthetic dataset X with 100 million data points and once with 10000 points(changed manually), defines the number of clusters n\_clusters as 3, and tests the algorithm with various numbers of threads specified in the num\_threads\_list. The kmeans\_thread function is responsible for updating the Mini-Batch K-Means model in a thread-safe manner using a lock. The measure\_time function measures the execution time of the clustering operation with the specified number of threads. The script then iterates over the list of thread counts, records the execution times, and stores them in the execution\_times list. This code aims to analyze the impact of multithreading on the performance of Mini-Batch K-Means clustering for different thread counts.

**Results and Analysis:**

Upon executing the algorithm with 10,000 data points, we observed a decrease in Speedup with an increased thread count, mirroring the trend seen in the logistic regression scenario.



Strikingly, even with 100 million data points, Speedup did not exhibit significant improvement with an increased thread count. This was attributed to the introduction of thread locks, a necessary measure to ensure thread safety by preventing race conditions. However, the trade-off was evident, as other threads were compelled to wait for the lock to be released, counteracting potential gains from parallelization.

A graph of a number of threads

Description automatically generated

This insight underscores the delicate balance between parallelism and synchronization in optimizing MiniBatchKMeans algorithmic performance.

**Convolutional Neural Networks:**

In our exploration of Convolutional Neural Networks (CNN), our focus was on introducing multi-processing to enhance the efficiency of the data preprocessing phase within the algorithm. The methodological approach involved utilizing the fashion\_mnist dataset, comprising 60,000 images, with a variable thread count ranging from 1 to 24.

**Methodology Overview:**

For multi-threaded data augmentation, we employed the ImageDataGenerator, facilitating parallel batch creation by controlling the number of threads through the 'workers' parameter. The initial execution of CNN model training without multiprocessing for data preprocessing yielded unfavorable results, particularly as the thread count increased. Keras, in its default configuration, utilizes multiple threads to run the data generator concurrently, aiming to expedite data loading with lower overhead compared to processes.

**Code Discussion:**

import tensorflow as tf

from tensorflow.keras import layers, models, datasets

import multiprocessing

import time

import os

import matplotlib.pyplot as plt

# Function to train the model

def train\_model(num\_threads):

# Load the Fashion MNIST dataset

(train\_images, train\_labels), (test\_images, test\_labels) = datasets.fashion\_mnist.load\_data()

# Preprocess the data

train\_images = train\_images.reshape((60000, 28, 28, 1)).astype('float32') / 255

test\_images = test\_images.reshape((10000, 28, 28, 1)).astype('float32') / 255

train\_labels = tf.keras.utils.to\_categorical(train\_labels)

test\_labels = tf.keras.utils.to\_categorical(test\_labels)

# Build a simple convolutional neural network

model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(10, activation='softmax'))

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Create a data generator with multi-threading

train\_data\_gen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255)

test\_data\_gen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255)

train\_generator = train\_data\_gen.flow(train\_images, train\_labels, batch\_size=128, shuffle=True)

test\_generator = test\_data\_gen.flow(test\_images, test\_labels, batch\_size=128, shuffle=False)

# Record start time

start\_time = time.time()

# Train the model using fit\_generator with multi-threading

#model.fit\_generator(train\_generator,

#steps\_per\_epoch=len(train\_images) // 128,

#epochs=5,

#workers=num\_threads,

#use\_multiprocessing=True,

#validation\_data=test\_generator,

#validation\_steps=len(test\_images) // 128)

model.fit(train\_generator,

steps\_per\_epoch=len(train\_images) // 128,

epochs=5,

workers=num\_threads,

use\_multiprocessing=True,

validation\_data=test\_generator,

validation\_steps=len(test\_images) // 128)

# Record end time

end\_time = time.time()

# Return execution time

return end\_time - start\_time

# Define the range of thread counts

#thread\_counts = [1, 2, 4, 8, 12, 16, 32]

thread\_counts = [1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 24]

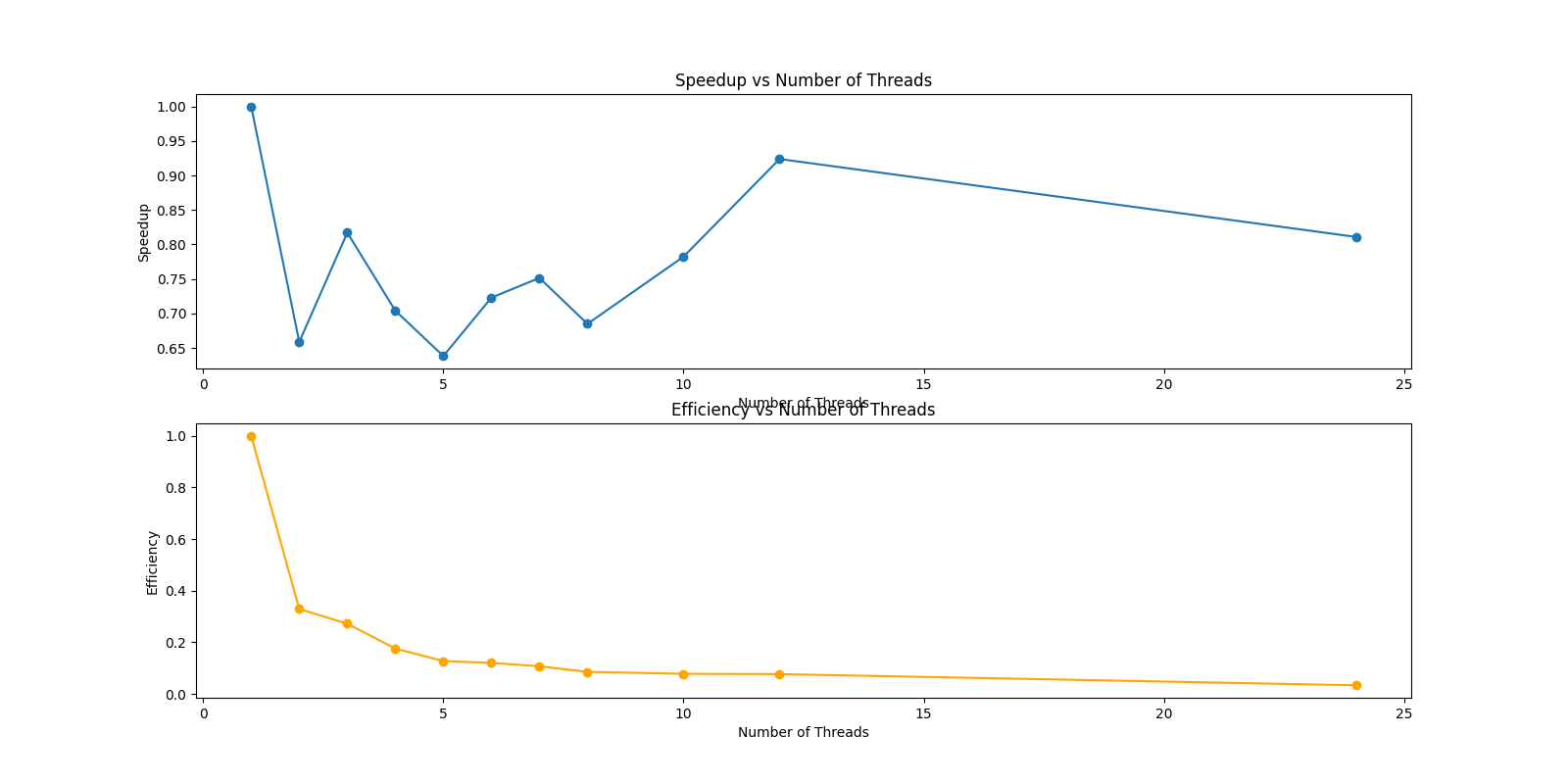
# Measure execution times for different thread counts

execution\_times = [train\_model(num\_threads) for num\_threads in thread\_counts]

This code trains a convolutional neural network (CNN) on the Fashion MNIST dataset, exploring the impact of multithreading on training performance for different thread counts. The `train\_model` function defines and compiles a simple CNN model and then uses the `fit` method to train the model on the Fashion MNIST dataset. The training is done using a data generator (`ImageDataGenerator`) with multi-threading enabled and other time with it disabled where the code starts working on multithreading instead, and the number of threads is controlled by the `num\_threads` parameter. We also measure the execution time of the training process for various thread counts specified in the `thread\_counts` list. The execution times for each thread count are stored in the `execution\_times` list, allowing analysis of the training performance under different parallelization settings.

**Results and Analysis:**

The inherent requirement for the data generator to be thread-safe presented a potential bottleneck. Thread safety is crucial to handling concurrent access to shared resources without introducing errors or inconsistencies, and it involves the use of thread synchronization. This aspect, in turn, could contribute to a decrease in speedup.



To address these challenges, we introduced multi-processing for data preprocessing, observing notable improvements with an increasing number of workers up to a certain threshold. Beyond this point, the overhead of inter-process communication began to offset the speedup gains. The inclusion of more inter-process communication mechanisms became necessary to facilitate seamless data exchange between processes.

A graph with blue lines and orange lines

Description automatically generated

Distinct from previous experiments involving threads, here, we opted for processes. Processes, unlike threads, do not necessitate thread safety, thereby altering the dynamics of concurrent execution. This experiment sheds light on the nuanced considerations and trade-offs associated with incorporating multi- processing into the data preprocessing phase of CNN, providing valuable insights for optimizing the training pipeline.

**CONCLUSION:**

In our comprehensive exploration of optimization techniques across logistic regression, MiniBatchKMeans, and Convolutional Neural Networks (CNN) work faster, we aimed to use parallel processing to make the algorithms more efficient. Each experiment came with its own set of challenges and lessons, giving us a better understanding of how parallelization, the details of each algorithm, and the characteristics of the data interact.

For logistic regression, inspired by Ensemble techniques, we successfully cut down the time it takes to run. However, we found that increasing the thread count doesn't always lead to better performance, especially when dealing with different dataset sizes. This highlighted the need to carefully consider both the structure of the algorithm and the size of the data when using parallelization.

MiniBatchKMeans, which processes mini-batches of data, showed us a trade-off between making things thread-safe and getting the benefits of parallelization. Introducing thread locks to keep shared objects safe ended up causing some slowdowns, emphasizing the tricky balance needed when optimizing algorithms that continuously refine clusters.

And For CNN, our attempt to use multi-processing for data preprocessing brought up some interesting points. While using multiple threads at first didn't give us great results due to safety concerns, shifting to multi-processing did improve things up to a certain point. This made us realize the importance of carefully handling communication between processes, which is a unique factor when using processes instead of threads.

In summary, the increase in the number of datasets generally enhances performance, yet a substantial rise in the number of threads can present challenges and potentially lead to decreased performance, as previously discussed. Our comprehensive exploration of optimization techniques across logistic regression, MiniBatchKMeans, and Convolutional Neural Networks (CNN) underscores the necessity to customize our parallelization approach based on the unique characteristics of each algorithm and dataset. The delicate balance between reaping the benefits of parallelization and navigating the associated challenges, such as ensuring safety and managing communication, demands thoughtful consideration. As we continue to navigate the evolving landscape of parallel computing, the insights gleaned from our experiments provide a robust foundation for making informed decisions in the ongoing quest to optimize various machine learning algorithms.

**BIBLIOGRAPHY:**

1. Prabhakaran, Selva. [“Parallel Processing in Python – A Practical Guide with Examples.” Machine Learning Plus](https://www.machinelearningplus.com/python/parallel-processing-python/)
2. [“Parallelising Python with Threading and Multiprocessing.” QuantStart](https://www.quantstart.com/articles/Parallelising-Python-with-Threading-and-Multiprocessing/)
3. [“A Simple Guide to Leveraging Parallelization for Machine Learning Tasks.” Oracle Blogs](https://blogs.oracle.com/ai-and-datascience/post/a-simple-guide-to-leveraging-parallelization-for-machine-learning-tasks)
4. [“The best Python libraries for parallel processing.” InfoWorld](https://www.infoworld.com/article/3542595/the-best-python-libraries-for-parallel-processing.html)
5. [“multiprocessing — Process-based parallelism — Python 3.12.0 documentation.” Python.org](https://docs.python.org/3/library/multiprocessing.html)
6. Junga, Yong Gyu, Min Soo Kanga, and Jun Heob. [“Clustering performance comparison using K-means and expectation maximization algorithms.” Biotechnology & Biotechnological Equipment](https://www.tandfonline.com/doi/pdf/10.1080/13102818.2014.949045)
7. Pai, Nanda Kishor M. [“Building sharp regression models with K-Means Clustering + SVR.” Paperspace Blog](https://blog.paperspace.com/svr-kmeans-clustering-for-regression/)
8. “Classification using K-Means Clustering vs. [Multiclass Logisitic Regression (Iris data).” Azure AI Gallery](https://gallery.azure.ai/Experiment/Classification-using-K-Means-Clustering-vs-Multiclass-Logisitic-Regression-Iris-data)
9. [“How to use KMeans clustering to improve the accuracy of a logistic regression model.” Stack Overflow](https://stackoverflow.com/questions/75250553/how-to-use-kmeans-clustering-to-improve-the-accuracy-of-a-logistic-regression-mo)
10. [“How does KMeans and Logistic Regression interact with MNIST dataset in pipeline.” Stack Overflow](https://stackoverflow.com/questions/70979417/how-does-kmeans-and-logistic-regression-interact-with-mnist-dataset-in-pipeline)
11. [“How can take advantage of multiprocessing and multithreading in Deep learning using Keras?” Stack Overflow](https://stackoverflow.com/questions/56344611/how-can-take-advantage-of-multiprocessing-and-multithreading-in-deep-learning-us)
12. Wong, Kay Jan. “Multithreading vs. [Multiprocessing Explained.” Built In](https://builtin.com/data-science/multithreading-multiprocessing)
13. [“Multiprocessing vs. Threading in Python: What Every Data Scientist Needs to Know.” FloydHub Blog](https://blog.floydhub.com/multiprocessing-vs-threading-in-python-what-every-data-scientist-needs-to-know/)
14. [“Multithreading or multiprocessing.” Stack Overflow](https://stackoverflow.com/questions/671607/multithreading-or-multiprocessing)
15. [“MultiProcessing in Python to Speed up your Data Science.” Medium](https://medium.com/@rviswa00/multi-processing-in-python-to-speed-up-your-data-science-8a810267b07)