Data-optimized frameworks for tiny ml applications

Abstract

Datasets have grown tremendously and need a resource rich computer to train deep learning algorithms. Owing to the size of datasets, they can only sometimes be stored on local storage due to memory constraints. A standard solution to overcome this challenge is storing it on a remote distributed file system. Distributed storage involves distributing the data across multiple nodes in a cluster, which can reduce the I/O load on individual nodes and improve the overall throughput.

The project tries to solve this problem by introducing a data-parallel model (large batch training) for training deep learning jobs over various resource-constraint edge devices. The training process is carried out parallel to the streaming process to overlay the CPU and IO workloads for faster training. The project plans to develop an optimized data storage and transfer framework that will provide APIs to the IoT devices for facilitating data movement from the IoT devices to the edge and vice-versa. We attempt to implement a multi-tiered cache mechanism by having a cache in the local storage and main memory of the compute nodes to optimize the overall training time and the idle time of the CPU.

Acknowledgment

I would like to convey my appreciation to all those who have contributed to the completion of this thesis.

Firstly, I am grateful to my supervisor Dr. Arnab Paul, for his guidance, encouragement, and unwavering support throughout the research process. The valuable guidance provided by his perceptive comments and helpful suggestions was instrumental in shaping the course of this project.

I am grateful to my colleagues and friends, who provided me with moral support and encouragement during the entire process. Their feedback and suggestions helped me refine my ideas and sharpen my arguments.

Finally, I would like to express my heartfelt appreciation to my family for their love, support, and understanding throughout my academic journey.

Introduction

Over the last few years, deep learning has experienced a significant rise in popularity and adoption across a wide range of industries and applications. One of the main drivers of the rise of deep learning has been the rapid advancements in hardware technology, such as the development of powerful graphics processing units (GPUs) and the increasing availability of cloud computing resources. These advancements have made it possible to train and run complex deep learning models faster and at a lower cost, making deep learning more accessible to businesses and individuals.

1.Problem with large datasets

At the same time, the growth of big data has provided deep learning algorithms with more data to train on, enabling them to make more accurate predictions and decisions. The increasing availability of data from sources such as social media, sensors, and the Internet of Things has further fueled the growth of deep learning applications.
With this increase in dataset sizes, a new problem arises. In most cases, the entire dataset cannot be stored on the local storage of the device the model is being trained on as the dataset size is larger than the storage capacity of the secondary storage device. Thus, they have to be stored on a remote system. The training process is carried out by breaking down the entire dataset into chunks that are small enough to fit into the main memory of the compute nodes. Each chunk is transmitted over the network individually and treated as a mini-batch by the training process.

As accelerators like GPUs, TPUs, etc., keep improving year after year, there is a lack of development on the I/O infrastructure of deep learning frameworks to support the highly developed accelerators. Transmitting such huge datasets can be very demanding on the network backbone and lead to huge latencies. This results in the CPU waiting idly to receive the next mini-batch to continue training the model on the next mini-batch.

2.Introduction of a layer of edge devices

This thesis proposes a framework that consists of 3 layers of devices- The innermost layer made of IoT devices which sits closest to the end users for sampling data through its sensors. The middle layer comprising edge devices and the outer layer which consists of the remote distributed file system tasked with archiving the dataset. The transmission of data between these layers is handled by apache kafka. The brokers and the clients are suitably configured to be optimized for the task.

Several approaches have been proposed where the IoT sensors transmit the sampled data directly to the cloud for analysis or storage. Both the training and the inference processes are executed on the cloud.
This method eliminates the need for a layer of edge devices but consequently introduces a greater latency for the inference step. Another problem is that this solution might not work in areas with a weak network connection or a network with a small bandwidth. Transmitting all the sampled data to the cloud drastically increases the workload on the communication channel. This problem is solved by introducing a layer of edge devices. This layer sits closer to the IoT devices in terms of the amount of time needed to transmit data between these layers. By doing so, we are essentially pushing the computational resources closer to the end user, thereby mitigating the workload of the network. It also reduces the latency of the inference step.

3. Heterogeneity in devices

Heterogeneity in IoT and edge devices (due to hardware, communication protocols, operating systems, data formats, data sampling rate, etc.) presents significant challenges for designing and deploying deep learning applications. This heterogeneity needs to be considered to ensure that their applications are compatible with the various devices and protocols that may be in use. The project plans to propose a framework with which it is possible to leverage the power of IoT to collect and exchange data and enable new applications and services.

Summary: In Chapter 2, we discuss and learn about all the essential technologies and fields of study that function as prerequisites to understanding this thesis. In Chapter 3, we talk through a brief overview of the relevant research papers, which assist in developing a deeper understanding of the project. Chapter 4 dives into the system developed during the thesis term. Chapter 5 gives details about the experiments conducted and their findings. Finally, in Chapter 6, we discuss the conclusion and possibilities of future work.

Background

This section lays out the context for our work. We first review the basic concepts used to dive deep into each section.

1.Deep Learning
Deep learning is a category of machine learning that is based on a connection of artificial neural networks. It involves using multiple layers of nodes (neurons) connected to each other in a hierarchical manner to learn and represent complex patterns in data.

In deep learning classification problems, the neural network is trained on a large amount of labeled data to learn the most relevant features for making accurate predictions. The network then uses these learned features to predict new, unlabeled data.

Deep learning can be applied to various fields, such as speech recognition, computer vision, natural language processing, and game playing. Some examples include image recognition, object detection, data analysis, language translation, speech recognition, and recommendation systems.

Some key terminologies used in deep learning are:

Deep neural network (DNN): It is a neural network with multiple hidden layers, which allows it to learn more complex and abstract features from the input data.

Backpropagation: A technique used to adjust the weights of the connections between neurons in a neural network during the training process. It is responsible for minimizing the error between the actual output and the predicted output.

Convolutional neural network (CNN): It is a type of neural network that is specifically designed for image and video analysis, where the input data is treated as a grid of pixels.

Epoch: It refers to one complete training pass through the entire training dataset during the training process. During each epoch, the neural network is fed with all the training data, and the weights of the neural network are modified based on the error between the actual and the predicted outcomes.

Mini-batch: It refers to a subset of the training dataset that is used to modify the weights of the neural network during each epoch for a smaller loss. Instead of using the entire dataset right away, the training data is broken down into tinier batches or subsets, which are processed sequentially by the network. The weights are updated after each batch is processed, rather than after each individual example.

Batch Size: It is an integer equal to the number of data samples that propagate forward through the neural network before updating the model parameters using backpropagation.

2. Large batch training

This thesis aims at creating a framework that aligns with the large batch training paradigm.

Although every edge device possesses its own set of local data, it can be viewed as a component of the overall batch. Each device conducts a forward and backward pass on its data and produces a single local gradient. These gradients are then transmitted to a parameter server, where they are combined to create a global update to the model. Alternatively, a distinct mechanism can be employed to update the global model without relying on a parameter server.

It should be noted that large batch training and federated learning are not interchangeable terms. In federated learning, each edge device trains independently on its local data for multiple iterations in each round without communicating with other devices. The model-update is only sent to the server after these iterations for aggregation with updates from other devices to enhance the shared model. In a federated learning approach, the data remains private to the device from which it was sampled. On the other hand, in large batch training the entire dataset is public to all edge devices and is stored on some globally accessible server.

3. Kafka

We utilize Apache Kafka in the data pipeline to stream data between the distributed file system, the IoT devices, and the edge devices.

Kafka is an open-source, distributed streaming platform for real-time data streaming and processing. It is developed to manage high volumes of data from multiple origins in real time, making it an excellent solution for modern data-driven applications. Kafka also provides data retention, fault tolerance, and stream processing features. Data retention allows messages to be stored for a configurable amount of time, while fault tolerance ensures that messages are not lost in the event of node failure. Stream processing allows data to be processed in real-time, enabling applications to respond to data as it is generated.

Fundamentally, Kafka is a publish-subscribe messaging system that allows messages to be sent between different applications or components in a distributed system. In Kafka, data is organized into topics; producers can write messages about one or more topics while consumers can read messages from one or more topics. A Kafka system has three categories of nodes- brokers, consumers, and producers.

A broker is a server that can save event streams from various sources. A Kafka cluster is a set of brokers synchronized by a zookeeper instance. Every broker in a cluster also acts as a bootstrap server, meaning that if we establish a connection with one broker in the cluster, we can connect to all of them. The Kafka cluster reliably handles and stores streams of events in topic-based buckets. Topics are the primary data organization unit in Kafka, and they are similar to log files in a folder structure, where the events are the files. Producers are the clients that produce events to Kafka, and they indicate the topics they will write to, as well as the method of partitioning events within a topic. Consumers are the clients that read events from Kafka topics, and they can subscribe to one or more topics.

4. MQTT protocol

We had initially planned to use MQTT for transferring data between different layers but consequently decided to replace it with Kafka.

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol used in constrained and unreliable network environments.
MQTT is a publish-subscribe messaging protocol widely used in IoT applications and its alternative CoAP (Constrained Application Protocol). In MQTT, data is organized into topics, and publishers can write messages to one or more topics, while subscribers can receive messages from one or more topics. This allows for the decoupling of producers and consumers, enabling efficient and scalable communication between devices.
One of the critical features of MQTT is its lightweight design, which allows it to be used in low-power, low-bandwidth devices such as sensors and mobile devices. MQTT uses minimal network bandwidth, and its protocol headers are designed to be small and efficient. Additionally, MQTT supports quality of Service (QoS) levels and message persistence, allowing reliable communication in unreliable network environments.

MQTT is widely used in the Internet of Things (IoT) and other distributed systems that require efficient and reliable messaging. MQTT is supported by a wide range of programming languages and platforms, and many open-source implementations are available. MQTT's lightweight design and support for constrained networks make it an ideal choice for IoT devices, where resource constraints are a common challenge.

Literature Review

Typically, system optimizations focus on creating a system that can address the I/O bottleneck through various means, such as implementing caching mechanisms, predictive prefetching, and hardware configurations that are optimized for this purpose. On the other hand, software-based solutions aim to produce middleware or libraries that can be utilized by deep learning applications to decrease training time. During the conducted literature review, the most noteworthy techniques were cache-based prefetching methods, which are the most sought-after optimization methods for data optimization. Such techniques predictively prefetch the required data for the deep learning job ahead of time to minimize the request's waiting period.
In this chapter, we review some papers that attempt to solve the problems mentioned above.

1. Caching

-HVAC

The progress in High Performance Computing (HPC) has resulted in a significant portion of large-scale Deep Learning (DL) applications' training time being devoted to input/output (I/O) operations on a parallel storage system. Researchers have attempted to enhance this process by using methods such as prefetching and caching. However, when applied to HPC supercomputers for large-scale DL training applications, these solutions present difficulties.

These difficulties include poor performance or failures on a large scale, a lack of generality and portability in design, complex deployment methodologies, and being limited to a specific dataset or application. To overcome these challenges, the authors of this paper suggest a solution known as the High-Velocity AI Cache (HVAC). HVAC is a distributed read-cache layer that employs node-local or near node-local storage technology to speed up read I/O operations by combining node-local or near node-local storage, eliminating metadata lookups and file locking while preserving the application code's portability.

The papers contributions in this paper include designing and implementing a scalable and read-only cache system (HVAC) for HPC computers, utilizing distributed hashing to find the memory addresses of data points in the cache to improve random read performance, and ensuring that DL applications or parallel file systems don't need to adapt to the use of HVAC due to the use of POSIX standards. HVAC sets itself apart from other caching strategies (LROC/ILM and LPCC) as it does not explicitly support writes. This mitigates a significant overhead in locking, metadata checks and data search.

2. Distributed machine learning with edge devices

Distributed machine learning in edge devices refers to using machine learning algorithms that are distributed across a network of edge devices, such as smartphones, IoT devices, and sensors. This approach's main goal is to leverage these devices' computing power to perform machine learning tasks locally without the need to transfer large amounts of data to a centralized server or cloud.

A use case of distributed machine learning in edge devices could be in the field of healthcare, where wearable devices such as smartwatches and fitness trackers can be used to collect data about a patient's health in real time. By using distributed machine learning algorithms, the data can be analyzed and processed locally on edge devices, enabling faster and more accurate insights. For example, a smartwatch equipped with sensors to monitor heart rate, blood pressure, and oxygen levels can use distributed machine learning algorithms to detect patterns in the data that may indicate an impending health issue, such as a heart attack.

Another use case is in the field of autonomous vehicles, where edge devices such as cameras, lidar, and radar sensors generate vast amounts of data that need to be processed in real-time. Distributed machine learning algorithms can be used to analyze the data locally on edge devices, enabling faster and more accurate decision-making by the vehicle's control system.

Overall, distributed machine learning in edge devices has the potential to improve efficiency, reduce latency, reduce power consumption, and enable real-time decision-making in a variety of applications, making it an important area of research and development in the field of machine learning.

3. Importance sampling
-SHADE

This paper \cite{shadecache} proposes a novel data sampling algorithm involving importance sampling and a caching mechanism that runs on top of this. It adopts a novel rank-based approach that captures the relative importance of samples across mini-batches. SHADE dynamically updates these values throughout the training process. It aims to improve the cache hit ratio and a faster convergence time with a marginal impact on accuracy.
The key highlights of this paper are.

- Establishment of a rank-based importance estimation of data samples to better the training performance.
- A priority-based sampling policy to utilize the inherent data locality of samples.
- An improved caching mechanism that encompasses rank-based importance values and the priority-based sampling policy to enhance the I/O efficiency for deep learning applications.

Their evaluation illustrates that SHADE enhances the read-hit ratio of a small memory cache (of just 10% of the working sample space of the dataset) by up to 4.5× compared to traditional, non-DLTaware caching policies, thereby improving the DLT performance.
The mechanism enhances accuracy convergence by a factor of 3.3 and the training throughput by a factor of 2.7 compared to the baseline LRU caching policy.

-iCACHE
The purpose of this paper is to introduce a new technique called iCACHE that accelerates DNN training jobs that are limited by input/output speed. iCACHE has been designed to fetch only the necessary parts of samples rather than the entire dataset to minimize training time while maintaining accuracy at an acceptable level. The study's main contributions are as follows:

- Proposing the integration of importance sampling into the caching system.
- Developing a cache replacement algorithm based on importance values, a dynamic packing method, and a multi-job handling mechanism.
- Categorizing data items based on their importance values into H-samples (high importance) and L-samples (low importance), and computing them with different probabilities during importance sampling.

The experimental results demonstrate that iCACHE has little impact on training accuracy and can speed up DNN training times by up to 2.0x compared to conventional caching systems used in the industry.
.

System Design

The system consists of 3 layers:

The first layer consists of IoT devices that sample data in their locality and transmit it to an edge device configured to capture data from it.

The middle layer consists of edge devices responsible for training the model on the data received from the IoT layer and archiving the collected data to the third layer.
Edge devices refer to the computing devices deployed at the network edge, closer to the end-users or data sources. These devices typically have limited computational and storage capabilities and are used for processing data in real time, often in a distributed environment. It is also responsible for processing inferences using the ml model and archiving all received data to the remote distributed file system.

The final layer consists of a remote distributed file system server responsible for archiving the entire dataset and transmitting it to the edge devices efficiently.

Each edge device is connected to multiple IoT devices, and the connection is formed such that each IoT device falls under the supervision of a single edge device.

It was initially proposed to use the MQTT protocol for data transmission but decided not to continue with it. The reason is that there was no in-built support for online training using MQTT in TensorFlow/PyTorch.
Kafka solved this problem as Tensorflow had an in-build API for using Kafka for streaming data and doing online training.
Since Kafka and MQTT are both built on top of TCP and a publisher-subscriber model, they are easily interchangeable in our use case.

1. setup

The experimental setup consists of a server and a raspberry pi 3B+. The server stores the dataset and the message publisher as well. The raspberry pi acts as the message subscriber.

2. data transmission pipeline

While working with MQTT, the clients (publisher and subscriber) were operated using the paho-mqtt python library, and Mosquitto was used as the broker.
Each packet transmitted using the MQTT protocol can have a payload of up to 256MB. Thus, datasets larger than 256MB need to first be broken down into chunks small enough to fit inside a packet. To do this, multiple data samples are first sequentially loaded and combined until they reach the upper limit of 256MB. Once no more data samples can be added to the current chunk or if all the data samples have been consumed, the message is sent to the broker.
The subscriber, on the other hand, polls messages from the broker. Each message is fetched and appended to the dataset in the local storage. Once the entire dataset has been stored on the subscriber, the model training can then begin.
Tensorflow requires the entire dataset to be loaded to the main memory to begin training. Suppose the dataset is such that it can fit in the local storage but not in the main memory; then,

training the model using conventional means is not possible. In such cases, we would have to use a data loader. A data loader prefetches a part of the data into memory beforehand so that the training process is not blocked during the data loading process. On top of this, data loaders have inbuilt support for batching, shuffling, Etc.

One of the disadvantages of using a data set loader is that it requires the entire dataset to be present in a location in the secondary storage.
In cases where the dataset size is larger than the size of the local storage, we are left with no other option but to resort to online training. Since Tensorflow does not have any in-built support for online training with MQTT, we switched to Kafka.

The same data pipeline is used when working with Kafka. The difference is that Kafka replaces MQTT. Also, the process of serialization and deserialization of data changes depending on the technology used. One advantage of making the transition is that Kafka clients can be configured with a finer granularity to suit our specific needs.

The datasets used are- The mnist Fashion dataset and Birds classification dataset.
Both of these are image datasets and are stored in the form of jpeg images.
Kafka requires the data to be transmitted in the form of a byte array or a string. To convert the image set to byte arrays, they are first loaded using the PIL Python library. They are then converted to a 2D numpy array, flattened to a single dimension, and then converted to a byte array. The labels corresponding to the images are encoded as utf-8 strings.
For each data sample, the image and the label are sent to the Kafka broker in the form of a key-value pair.
The consumer of the messages has to first deserialize the fetched data before feeding it to the model. The deserialization process is as follows. The incoming data is first decoded to uint8 using the tf.io library. As we are using the TensorFlow API for Kafka, the incoming data is received in the form of tensors. After decoding the data, it is then reshaped back into its original shape. It is then converted to a float and finally normalized between the range 0-1 by dividing all the intensity values by 255. Normalizing an image dataset between 0 and 1 (or between -1 and 1) is a common preprocessing step in machine learning tasks involving image data. This is done for several reasons: numerical stability, improved convergence, and better generalization. The key or the label is simply converted from a string to an integer.

Experiments

The experiments are conducted on the MNIST fashion dataset. It contains 60000 training images and 10000 testing images. Each image has a size of 28x28 pixels. It is roughly 70 MB in size. A simple Convolutional Neural Network (CNN) is used as the model.

The system resource utilization metrics (CPU utilization , Disk read and write, Memory utilization and network monitoring) are tracked and stored to analyze the performance of the scripts. The time period of sampling is one second for all resources.

For the experiment, the processing of streaming data from the kafka broker and the training of the model is executed in parallel. The resource monitoring scripts execute in the background as long as the training process does not terminate.

Explanation of the diagram: The CPU utilization is quantified in terms of the joint workload on all the cores. Since the model training was executed on a raspberry pi having four cores, the CPU utilization can range from 0 - 400%. The utilization ranges roughly from 200 to 300 but drops to 0 when the CPU is waiting idly for the next mini-batch to be fetched from the Kafka broker. The memory utilization increases and consumes about 250 MB when loading the TensorFlow library. It then increases steadily as the mini-batches are continuously polled from the broker. Since the Kafka consumer doesn't flush out the mini-batches which have already been used for training in previously completed epochs, the mini-batches keep on consuming space on the RAM. Consequently, memory utilization increases slowly but steadily during the entire process. There is no activity on the disk read graph, as nothing is read from the disk during the process's lifetime. The disk write shows spikes at the end of every epoch. This is because the model weights are stored on disk after executing each epoch. The weights are stored locally so that another process can load them for inferencing incoming data in parallel. Data is received from the network in chunks, as indicated by the spikes on the relevant graph. The graph pertaining to the rate of data received shows spikes as the Kafka consumer needs to connect to the broker to preserve its state of liveliness and also update the offset parameter. The offset parameter determines the index of the last retrieved message from the broker.

Summary

In this thesis, we attempted to develop a framework for training massive datasets (in the range of TBs/PBs) on resource constrained devices, where the training dataset is larger than the memory capacity of the local storage using a data-parallel model. Parallelizing the training job and managing incoming streaming data adds another layer of optimization. It speeds up the overall training process by improving CPU utilization and throughput.
Moreover, having a multi-tiered cache can further optimize the training time by reducing the overhead required for waiting to load the next mini-batch into the RAM.

Future work

The work on the multi-tiered cache needs to be completed and will need further fine-tuning. We need a way to use Kafkas streaming services on devices with small RAM sizes.

The data-parallel model needs to be tested on multiple edge devices using Kafka. Kafka is built for scalability and can handle multiple consumers and consumer groups. This will facilitate scaling the model training step to multiple devices simultaneously.

Handling heterogeneity in the edge and IoT devices must be ensured to create a framework that can work consistently and independently of the underlying hardware. Since Kafka can only stream data using a finite kind of data type- byte arrays or string- the IoT and edge applications must be developed using libraries that can handle sending, receiving, and processing data in these formats.

Traditionally in deep learning jobs, all the data points are used for training exactly once in a single epoch. Since we are not storing the dataset on edge devices, the entire dataset needs to be transmitted over the network in every single epoch. This puts a lot of pressure on the network. There is a need for optimization here, but since each minibatch is sampled from the dataset randomly, there is no pattern or data locality which can be exploited to create a cache. Deep learning jobs are especially not cache friendly. Also, DLT systems use SGD with a random sampling policy because it treats all data points with equal importance.
Recent findings indicate that this may not hold. Samples contribute unequally to the final accuracy of the model. Some samples may generate little to no impact on the model accuracy and thus can be ignored during training. This finding creates an opportunity to optimize the IO bottleneck. This problem can be solved by having a cache at the computational nodes, which can work atop an importance-based sampling caching mechanism.
Importance sampling is the process of figuring out which samples have relatively more impact on the loss function from the sample space. By prioritizing samples with relatively higher importance, a DL job can improve training time and testing accuracy.

We plan to make use of a remote distributed file system for storing the dataset as it can range from terabytes to petabytes. The modifications to the system design are highlighted in the diagram below.