DS603-ADVANCED MACHINE LEARNING–Arulnithi P (12210280)

Multiple GPU training

Objective: The primary objective of this project is to train a convolutional neural network (CNN) for image classification using the well-known MNIST dataset in a highly distributed and efficient manner. The goal is to accelerate the training process by leveraging the computational power of multiple GPUs, thus reducing the time taken for the model to achieve optimal performance and convergence.

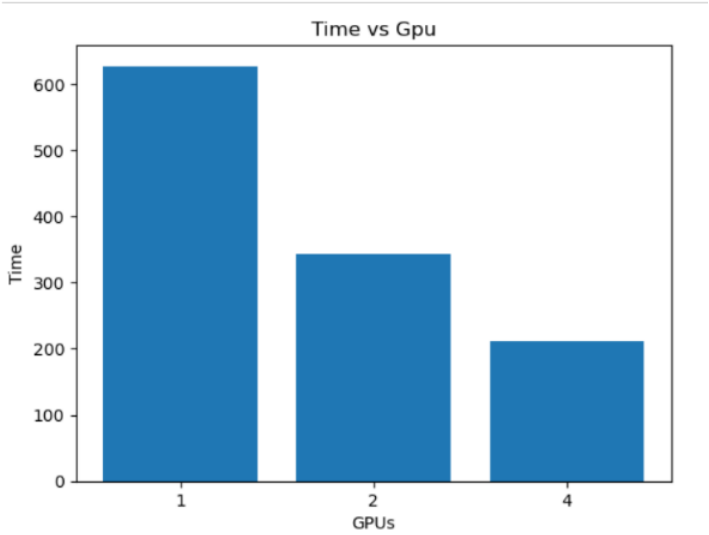
Algorithms and Distributed Concepts:

1. Data Parallelism: The distributed training approach employed in this project is centered around the concept of data parallelism. Data parallelism is a technique that involves dividing the input data into smaller, more manageable batches and distributing them across multiple GPUs for simultaneous processing. Each GPU trains a replica of the model using its assigned batch of data, and the model parameters are updated by aggregating the gradients computed by each GPU. By parallelizing the training process, this method significantly accelerates the overall training time.
2. Distributed Sampler: To ensure an even distribution of the dataset among the available GPUs, the DistributedSampler class from PyTorch is employed. This class effectively partitions the dataset, making certain that each GPU receives a unique and exclusive subset of the data for training purposes.
3. Distributed Data Parallel (DDP): The DistributedDataParallel (DDP) class from PyTorch is a crucial component of this project. This class is used to wrap the model on each GPU, allowing for efficient communication and gradient synchronization between the GPUs during the training process. The DDP class automatically handles the distribution of model parameters and gradients, reducing the need for manual intervention and enhancing the overall efficiency of the training process.
4. NCCL Backend: In this project, the NVIDIA Collective Communications Library (NCCL) backend is utilized for initializing the process group in the distributed training setup. NCCL is a high-performance library specifically designed to handle distributed training on NVIDIA GPUs, providing efficient inter-GPU communication and ensuring optimal utilization of the available hardware resources.
5. Training Loop: The training loop in this project is designed to perform the following operations on each GPU:
    - Load a batch of data and send it to the corresponding GPU for processing.
    - Perform a forward pass through the model to compute the output predictions.
    - Calculate the loss using the ground truth labels, which serves as a measure of the model's performance.
    - Perform a backward pass to compute the gradients, which are necessary for updating the model parameters.
    - Update the model parameters using a chosen Stochastic Gradient Descent

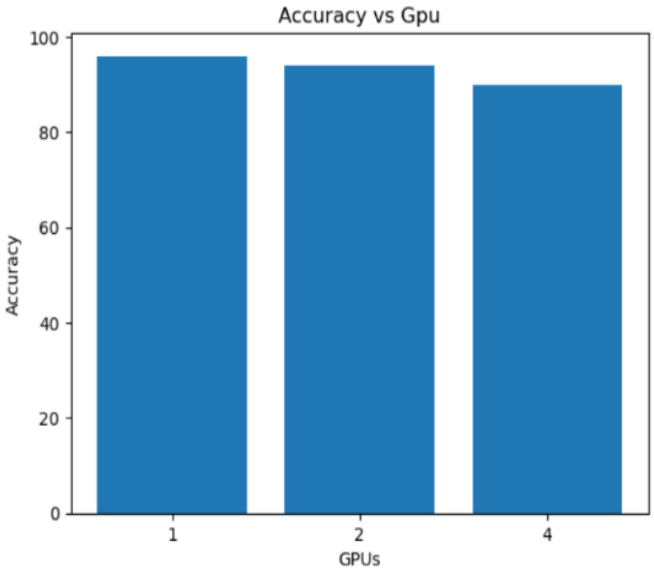
Experimental setup

I have run the model in Nvidia RTX 3080 GPUs. I have VAST.ai GPU renting service for my experiments.

Results



As we can see that time taken to train the model has significantly reduced.



As the number of GPUs increases the Accuracy has dropped a bit. With 4 GPUs we can significantly see that there is drop of 6 % compared to single GPU.

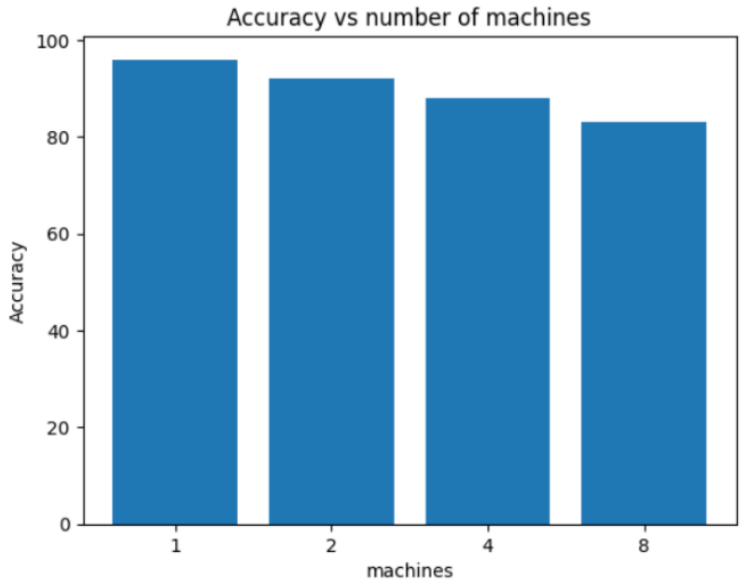
MULTINODE TRANING

The distributed concept is same as above but in multi node training there will be global rank and local rank. Where the global rank is the ID number of GPU, and I will be unique and local rank will be GPU ID within the system. This is a unique identifier for each process, which is used to determine its position in the distributed system. The global rank helps to coordinate the communication between the machines, enabling them to work Together effectively.

Experiment setup

I have run the model in Nvidia RTX 3090 GPUs. To run code in multiple machines we need to clone the code in multiple machines and execute the torchrun command keeping one node for aggregation of gradients. The performance ultization in this machine will be less compared to other nodes because it to compute aggregate Gradients.

Results



As the number of machines increase, there is a drop in accuracy, which is an important consideration in distributed computing. However, it is worth noting that when training on

two machines, the accuracy was 92%. In contrast, when using a single machine with multiple GPUs, the accuracy increased to 94%. Although the GPUs used were the same, there was a

2% decrease in accuracy when using a multi-node setup due to the additional communication delays. Similarly, when training on four machines, the accuracy dropped to 88%, but when using a single machine with four GPUs, the accuracy increased to 90%. This difference can be attributed to the communication overhead between the machines in the distributed setup.

Therefore, it is crucial to carefully consider the trade-offs between accuracy and performance when designing distributed computing systems. While scaling up the number of machines can increase the computational power, it may come at the cost of accuracy due to communication overhead. On the other hand, using a single machine with multiple GPUs can provide a balance between performance and accuracy. So, it is better to train on 8 GPU on single machine than 8machines with a single GPU.

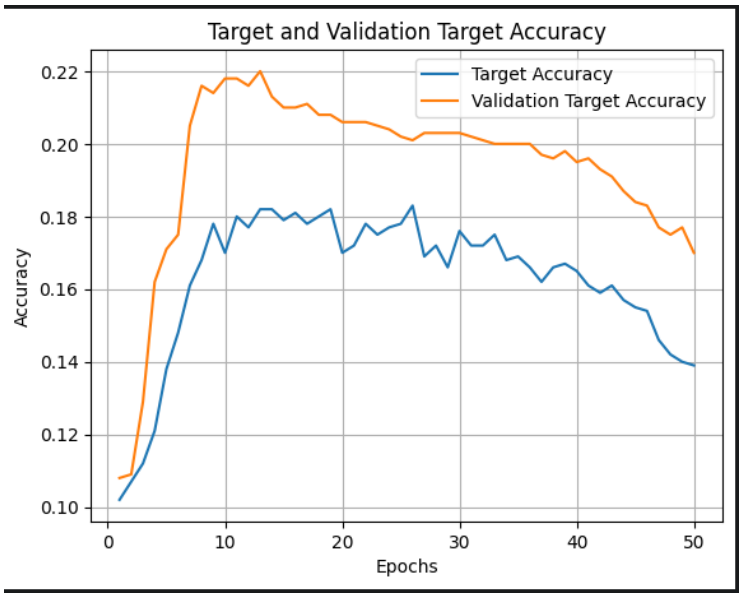
Knowledge Transfer for Ensemble models to Target model.

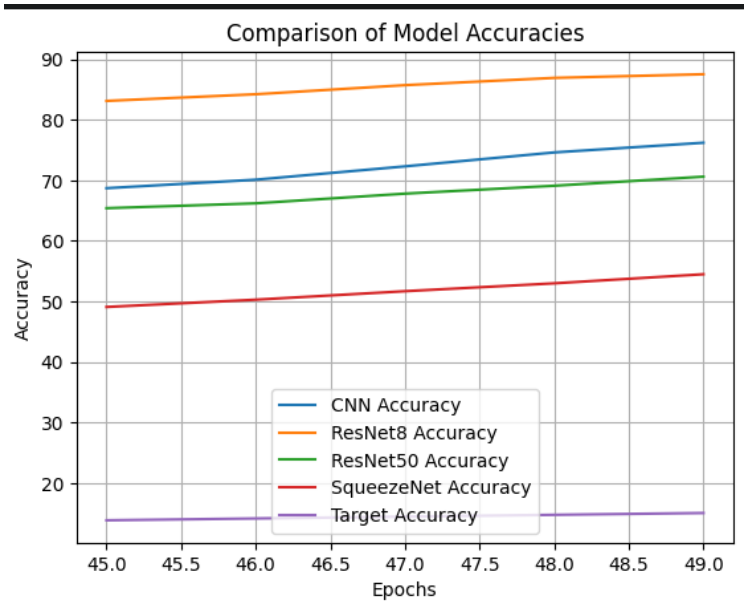
Objective: The objective of this project is to improve the performance of a deep learning model by utilizing an ensemble of smaller models and performing bidirectional knowledge transfer between the ensemble and the target model.

Algorithm
Train ensemble models

- Initialize ensemble models
- Train each model on the training set
- Compute loss and accuracy on training set
- Compute loss and accuracy on validation set
- Update model parameters using backpropagation
- Train target model (VGG19)
- Initialize target model and ensemble models
- For each epoch:
    - For each batch of data:
        - Compute outputs of ensemble models and target model.
        - Calculate classification loss for target model
        - Compute transfer losses between target model and ensemble models
        - Calculate total loss for target model
        - Compute gradients of total loss with respect to target model parameters
        - Update target model parameters using optimizer
        - Calculate target model accuracy on current batch
    - Compute target model loss and accuracy on validation set
- Evaluate target model
- Evaluate target model performance on validation set
- Report target model accuracy and loss for each epoch during training.

Results





As we see that as we increase epochs the target model is learning.

Connecting Low-Loss Subspace for Personalized Federated Learning

Research paper that proposes a novel approach to personalized federated learning. The traditional federated learning approach trains a global model using all the data from the participating clients. However, this approach may not be suitable for scenarios where the clients have different data distributions or privacy concerns. To address these issues, the authors propose the use of personalized federated learning, where each client trains a localmodel on its own data, and the models are combined to form a global model. The proposed approach is based on the idea of connecting the low-loss subspaces of the local models toform a subspace that captures the common features across the clients' data. This subspace is used to update the global model, which is then personalized for each client using the client's own model. The authors also introduce a new algorithm called CONNECT, which uses a graph-based approach to connect the low-loss subspaces of the local models.

Algorithms
1. Client model training: Each client trains its local model on its own subset of the data for a fixed number of epochs using stochastic gradient descent with a fixed learning rate.

2. Local model analysis: After training, each client calculates the Hessian matrix and the top k-eigenvectors of its local model.

3. Aggregation: Each client sends its model's state dictionary to its neighbors in a network graph and receives the model state dictionaries from its neighbors. The state dictionaries are

averaged, and the average is used to update the client's model.

4. Global model update: The global model is updated by averaging the models of all clients.

5. Evaluation: The accuracy of the global model is evaluated on the test data.

The main contribution of the CONNECT algorithm is the incorporation of the low-loss subspace of each client's local model during model aggregation. This is achieved by using the top k eigenvectors of the Hessian matrix, which captures the curvature of the loss function near the local minimum. By using these eigenvectors, CONNECT ensures that the models are aligned with the low loss, which leads to faster convergence and better Performance.

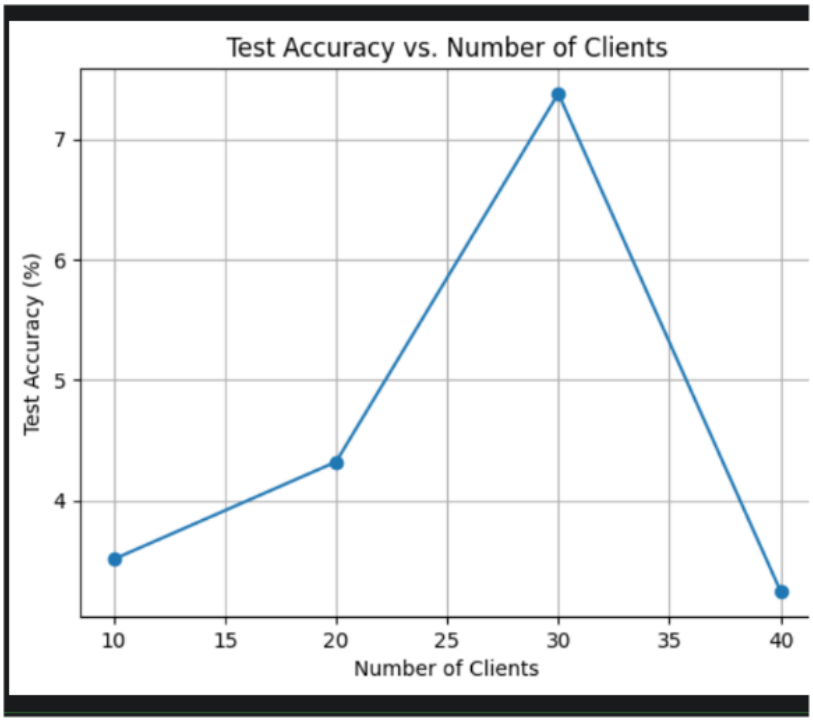
Modification from original paper

Computing the full Hessian matrix can be computationally expensive and memory-intensive, especially for large models. Instead, you can use an approximation technique, such as the Hutchinson method, which estimates the Hessian-vector product (Hv) using random vectors without computing the full Hessian.

Experiment and results

All the experiments were conducted in local machine environment. I7 processor and 6 GB RTX 3060. But the clients are formed under virtual environments mimic actual federated

environment. I have trained the model for 10 epochs and measured the accuracy calculation of Hessen matrix is expensive of time. I have used EMNIST dataset.



Conclusion

Connecting Low-Loss Subspace for Personalized Federated Learning" by Li et al. (2021) focuses on utilizing the low-loss subspace of individual clients to improve personalized FL. This approach takes advantage of the unique characteristics of each client's data distribution to generate personalized models, while also allowing for efficient communication between clients and the server. Additionally, the proposed approach shows improvements in both accuracy and communication efficiency compared to other personalized FL methods. Overall, the approachpresented in this paper provides a promising direction for improving personalized FL in practical applications.

Reference

1. "Personalized Federated Learning via Model Agnostic Meta-Learning" by Y. Zhao, M. Li,

L. Lai, N. Suda, D. Civin, and V. Chandra (2018)

2. Personalized Federated Learning with Adaptive Weighting" by Y. Yang, Y. Chen, J. Zhang, and S. Zhang (2021)

Arduino-Based Distributed Training of Logistic Regression

Objective: The objective of this project is to train a logistic regression model on the Titanic dataset using multiple Arduinos to perform distributed training. By dividing the training data into subsets and training each subset on a separate Arduino, the aim is to reduce the training time and utilize the computational power of multiple devices.

Python code Algorithm:

- Import necessary libraries for data processing, machine learning, and Arduino communication.
- Load the Titanic dataset, preprocess the data by handling missing values, encoding categorical variables, and creating a new feature (FamilySize).
- Perform dimensionality reduction using PCA to reduce the number of features.
- Standardize the data using the StandardScaler.
- Split the data into training and testing sets.
- Establish serial communication with the Arduinos connected to the computer.
- Define the receive_weights function to read the weights sent by the Arduino after training.
- Define the train_on_arduino_with_index function to train the logistic regression model on a specific Arduino using a subset of the training data. Send the necessary training parameters and data to the Arduino and receive the updated weights after training.
- Divide the training data into subsets for each Arduino.
- Train the logistic regression model on all Arduinos in parallel using the ThreadPoolExecutor.
- Calculate the average of the weights received from all Arduinos.
- Test the performance of the logistic regression model by calculating the accuracy on the test set.
- Close the serial connections to the Arduinos.

Ardunio code Algorithm

- Set the unique board ID and the number of features for the logistic regression model.
- Initialize the initial weights for the logistic regression model.
- Set the built-in LED pin number (usually 13).
- Define the sigmoid function for the logistic regression model.
- Define the sendWeights function to send the updated weights back to the Python script over the serial connection.
- Set up the Arduino by initializing the LED pin as an output and establishing serial communication.
- Implement the main loop to process incoming commands:

- a. Check if there is any data available on the serial connection.
- b. Read the Arduino ID and command from the serial connection.
- c. If the received Arduino ID matches the current board's ID, check if the command is "train".

- d. If the command is "train", execute the following steps
- i. Turn on the LED to indicate that the Arduino is training the model.
- ii. Read the training parameters (epochs, learning rate, number of samples, and batch size) from the serial connection.
- iii. Iterate through the epochs and process the samples in batches:

    1. Read the features and label for each sample from the serial connection.

    2. Calculate the prediction using the sigmoid function and the current weights

    3. Compute the error between the prediction and the actual label.

    4. Update the gradients for each weight based on the error and the features.

- iv. Update the weights based on the calculated gradients and the learning rate.
- v. Turn off the LED to indicate that the training is complete.
- vi. Send the updated weights back to the Python script using the sendWeights function.