Parallelization of Neural Networks

Talal K. (25253), Beena A. ()

Institute of Business Administration

CSE 471: Neural Networks and Deep Learning

Introduction

The breakthrough of the machine learning domain lies in the neural networks with an unquestionable value. The sophisticated neural networks structured after the brain of a human have evolved into the basis for a wide variety of applications from self-driving vehicles and face recognition systems to predictive analytics and beyond. Nevertheless, in line with the expansion of the goals and scope of these applications, the complexity and cost of neural network training and deployment also tend to increase. The datasets fed into the models to be trained are becoming more voluminous, and the networks are growing more complex, taking into account millions, if not billions, of parameters.

This complexity and scale necessitate sophisticated approaches for computational efficiency, with deep neural network parallelization emerging as a main solution. Parallelizing computations across multiple units speeds up the training process and handles more complicated models and larger data sets. Data parallelism is vital for processing datasets beyond the ability of a single processor unit, distributing data batches across multiple GPUs to independently compute gradients that are later aggregated to update model parameters. This distributed computing approach is beneficial when the neural network model can fit within the memory of each processor.

However, when models are too complex or memory resources are limited per processor, model parallelism becomes essential. Model parallelism partitions the model, assigning each processor a different part, necessitating complex coordination of the forward and backward propagation processes. This complexity makes model parallelism crucial for training the largest and most complex neural network models.

The ongoing development of GPU technology, particularly their parallel structures suited for matrix and vector operations, has made GPU-accelerated computing foundational in deep learning. New computing platforms, such as Apple's M1 chip, exemplify progress with their performance and energy efficiency, supporting both data and model parallelism within a single chip. These advancements highlight the need to optimize neural network models and training processes to fully utilize the computational power of such advanced hardware platforms.

Parallelization minimizes data transfer and memory usage within processors through techniques like data batching, memory pooling, and effective communication protocols, ensuring minimized latency and maximized throughput. Selecting the appropriate form of parallelism—data or model—for the model architecture and computational resources is crucial for optimal performance and scalability.

Building on these parallelization strategies, our research further explores the application of these techniques to the fine-tuning of Large Language Models (LLMs). The fine-tuning process, essential for adapting pre-trained models to specific tasks, benefits significantly from parallelization. By leveraging data and model parallelism, the fine-tuning of LLMs can be accelerated, making the process more efficient and capable of handling larger and more complex models. This approach enables the development of more accurate and sophisticated models for natural language processing tasks, demonstrating the broader influence of neural network parallelization on artificial intelligence advancements.

To sum up, parallelization of neural networks is crucial for managing the significant computing power required for machine learning technology. Distributing workloads across different units allows researchers and practitioners to train larger and more complex models more efficiently, paving the way for new achievements and applications in artificial intelligence. As the field continues to evolve, supported by the latest hardware developments, parallelization will remain a fundamental strategy in developing machine learning and AI technology.

Methodology

Code here

As for the methodology first, we used the iris dataset on a simple neural network code to see what can be done.

Timeline

1. Baseline for simple NN on IRIS dataset.
2. Batch processing w/ threading on IRIS.
3. Using Apple Silicon MPS (metal performance shaders) w/ threading on IRIS.
4. Baseline for simple CNN on MNIST dataset.
5. Using MPS w/ batch processing on MNIST.
6. Using CUDA w/ batch processing on MNIST.

Things that were kept constant throughout:

- Learning rate = 0.001
- Hidden Layers = 2 (NN) | 5 (CNN)
- Batch size = 64
- Epochs = 100 (NN) | 10 (CNN)
- Number of workers = 4

Overall architecture and logic

First the dataset was loaded and then preprocessing was done on each dataset. Then we defined the neural network a simple NN for IRIS dataset and a simple CNN for MNIST dataset. After which the model is trained on a set number of epochs. Finally, the model's performance is evaluated. This is a rough outline of the overall architecture across both codes.

Key Functions (Simple NN)

- **SimpleNN:** Defines the structure of the neural network.

    - **Activation function** ReLU (Rectified Linear Unit).
    - **Layers:**
        - **‘fc1’** fully connected layer with 4 input features and 50 output features.
        - **‘fc2’** fully connected later with 50 input features and 3 output features.
- **preprocess_data:** preprocesses the dataset by loading the dataset, standardize the features and split the dataset into training and test splits.
- **Main:** Defines the entire workflow.
    - Sets the computation device (mps)
    - Uses ‘ThreadPoolExecutor’ to preprocess data asynchronously.
    - Covert the preprocessed data to PyTorch tensors and transfer them to the specified device.
    - Create datasets and data loaders.
    - Initializes the neural network model, loss function and optimizer.
    - Measures the execution time for training the model.
    - Train the model over 100 epochs.
    - Evaluate the model’s performance.

Algorithms and Techniques (Simple NN)

- **Standardization** to scale the values to get better performance.
- **Train-Test Split** to evaluate the model’s performance.
- **Threading for Data Preprocessing** speeds up the preprocessing.
- **Neural Network Training** using back propagation and gradient descent. This adjusts the model parameters to minimize the loss function over training the data. (parameters were kept constant for our testing).
- **Evaluation** measures the model’s ability to generalize to unseen data.

Key Functions (Simple CNN)

- **SimpleCNN:** Defines the structure of the convolutional neural network.
    - **Layers:**
        - **conv1:** Convolutional layer with 1 input channel, 32 output channels, and a kernel size of 3.
        - **conv2:** Convolutional layer with 32 input channels, 64 output channels, and a kernel size of 3.
        - **pool:** Max pooling layer with a kernel size of 2 and a stride of 2.
        - fc1: Fully connected layer with 64 * 7 * 7 input features and 128 output features.
        - **fc2:** Fully connected layer with 128 input features and 64 output features.
        - **fc3:** Fully connected layer with 64 input features and 10 output features.
- **Activation Function:** ReLU (Rectified Linear Unit) activation function.

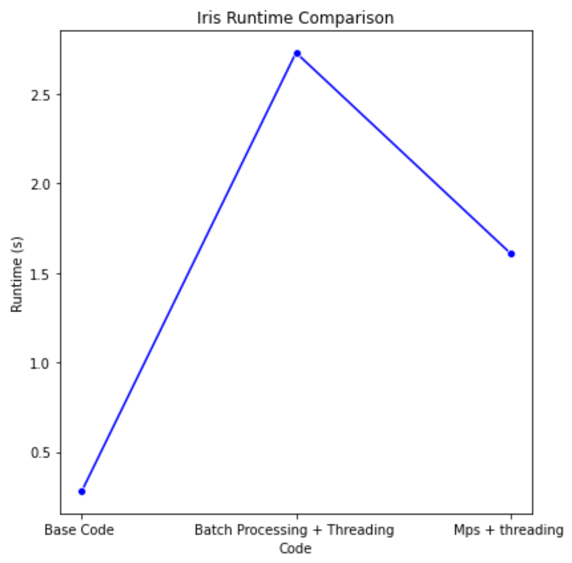
- **load_data:** Loads and preprocesses the MNIST dataset by defining a transformation pipeline to convert images to tensors and normalize them. Then downloads and transforms the MNIST training and test datasets.
- **main:** Defines the entire workflow.
  - Sets the computation device MPS/CUDA (if available) or CPU.
  - Use 'ThreadPoolExecutor' to load the data asynchronously.
  - Create DataLoader for training and tests datasets.
  - Initialize the neural network model, loss function and optimizer.
  - Measure the execution time for training the model.
  - Train the model over 10 epochs.
  - Evaluate the model's performance on the test set.

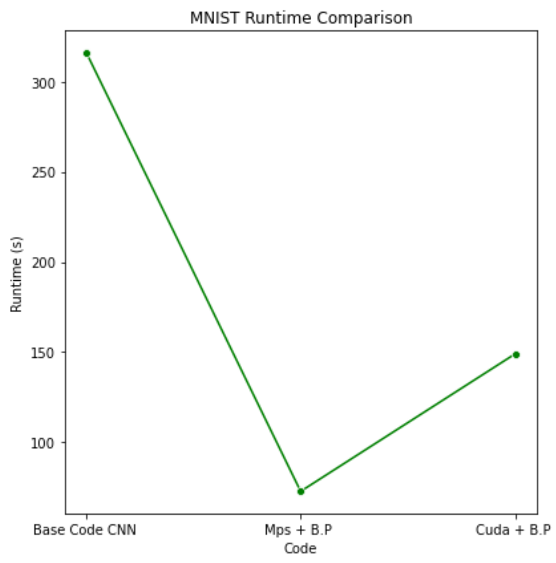
Algorithms and Techniques (Simple CNN)

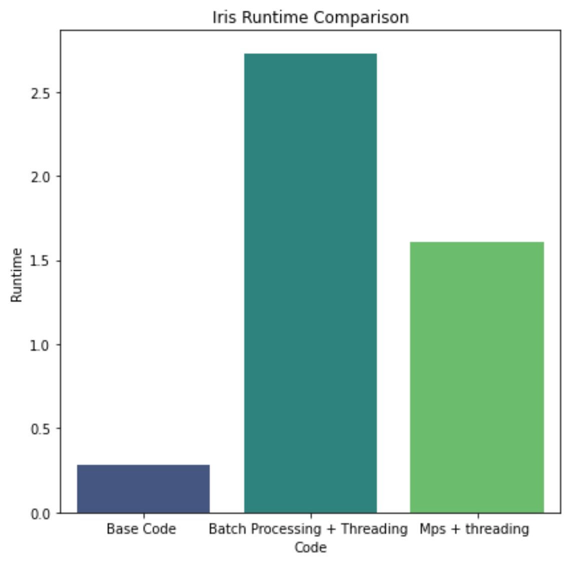
- **Normalization** converts images to tensors and scaling pixel values. This helps in stabilizing the learning process and accelerating convergence.
- **Train-Test Split** splits the data into testing and training sets so that model's performance can be evaluated later.
- **Threading for Data Loading** speeds up the data loading process by utilizing multiple threads.
- **Convolutional Neural Network** uses two convolutional layers followed by pooling and three fully connected layers. Convolutional layers extract spatial features from the input images and fully connected layers map these features to the output classes.
- **Neural Network Training** same as the simple NN steps.
- **Evaluation** also same as simple NN.

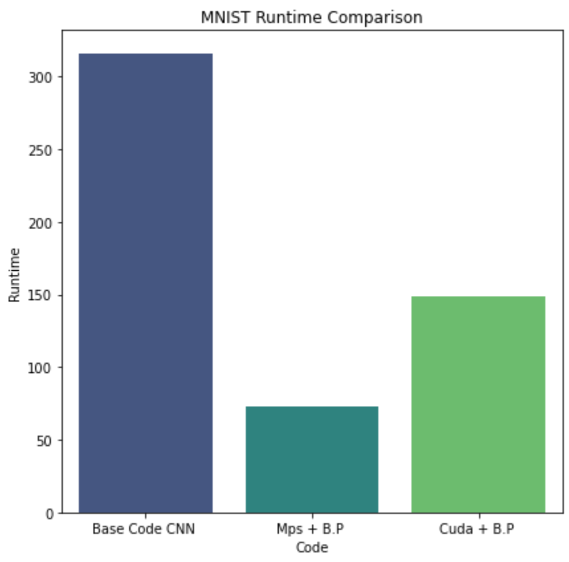
Results

Code	**Runtime(s)**	**Accuracy (%)**
Base Code (iris dataset)	0.28s	100%
Batch Processing + Threading (iris)	2.73s	100%
Mps + threading (iris)	1.61s	100%
Base Code CNN (mnist dataset)	316.18s	99.09%
Mps + B.P (mnist)	72.64s	99.12%
Cuda + B.P (mnist)	148.9s	98.05%









Fine-tuning an LLM (gpt2)

Note: This was done on a Kaggle notebook using the power of GPU provided by Kaggle.

In this Kaggle notebook, we utilize Large Language Models (LLMs) to optimize competitive programming. With a dataset of problem statements and solutions, we fine-tune LLMs to generate tailored code solutions for each challenge, showcasing AI's potential to enhance coding efficiency and innovation.

The Kaggle dataset contains competitive programming problems. Each dataset entry corresponds to a distinct problem, encompassing essential columns like the problem statement, coding solution, and input data. These columns furnish vital details for training a language model to generate coding solutions based on given problem statements.

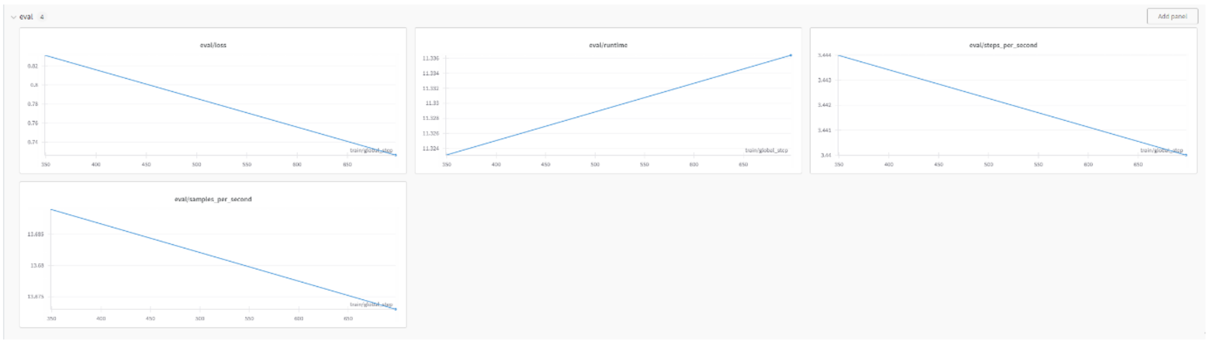
Initially, the code imports requisite libraries and installs the evaluate package for model evaluation. Subsequently, it reads the CSV dataset, isolates pertinent columns, and merges them into a single text column, facilitating subsequent language model training. This preprocessing step streamlines the tokenization process.

Subsequently, the text data is tokenized using the GPT-2 tokenizer, and a Dataset object is created. The dataset is split into training and testing sets to facilitate model training and evaluation. The GPT-2 tokenizer is initialized, and training arguments are configured for fine-tuning the model.

Following initialization, the model is trained using the Trainer object, adhering to specified parameters such as the number of epochs and batch size. Once training completes, the fine-tuned model undergoes evaluation on the test dataset, and the results are printed to assess model performance.

Finally, both the trained model and tokenizer are saved to disk for future use. In the testing phase, the saved model and tokenizer are loaded to generate coding solutions based on provided problem statements. This demonstrates the model's functionality in generating coding solutions tailored to specific programming problems.

Graphical Representation:



- **Eval/loss:**

This graph indicates that the model's performance is improving over time, as evidenced by the decreasing evaluation loss. This is a positive sign, showing that the model is learning and generalizing better to the validation data as training progresses.

- **train/global_step:**

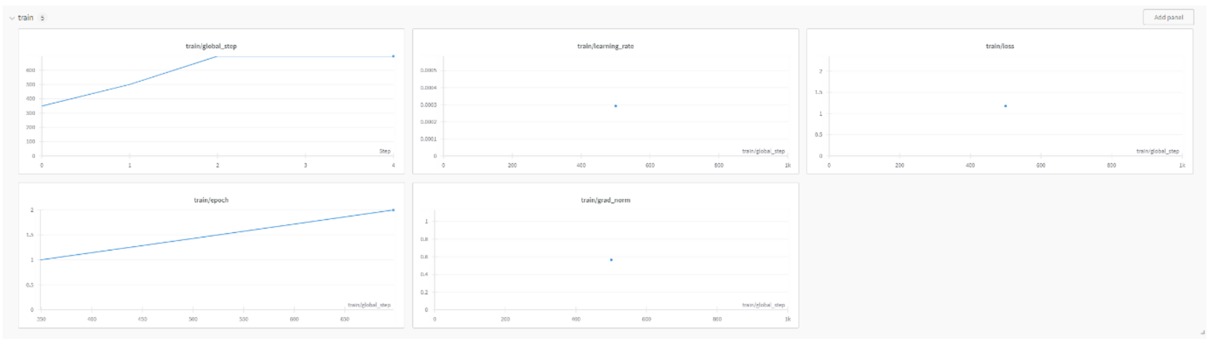
The evaluation runtime is increasing slightly as training progresses, which could be due to increased model complexity or larger evaluation datasets.

- **eval/steps_per_second:**

The evaluation steps per second are decreasing slightly as training progresses, indicating a slower processing rate.

- **eval/samples_per_second:**

The evaluation samples per second are decreasing slightly as training progresses, indicating a slower processing rate



- **Train/Global Step:**

The global step count is increasing steadily as training progresses. This indicates that training is proceeding as expected, with steps being recorded correctly over time.

- **Train/Learning Rate:**

The learning rate is set at 0.0003 at the provided global step. Additional data points would be necessary to understand how the learning rate changes over training.

- **Train/Loss:**

The training loss is recorded as 1.5 at the provided global step. Additional data points would be necessary to understand how the training loss changes over the course of training.

- **First Plot (train/epoch):**

This indicates that the number of epochs is increasing linearly with the global step. This behavior is typical in training processes where the epoch count increments as the model undergoes more iterations or batches of training data. It suggests that the training is progressing as expected without any irregular interruptions.

- **train/grad_norm:**

The gradient norm is a measure of the magnitude of gradients used to update the model weights during training. The single data point indicates that gradient norms were not frequently recorded or that there was a specific moment at the global step of 400 where this metric was captured. The value of 0.6 suggests that the gradients were not excessively large, which is generally a good sign as it implies that the training process is stable and not experiencing issues like exploding gradients.

Conclusion

Result Analysis

Iris Dataset

- **Base Program: Fastest runtime at 0.28s.**
- **Threaded Batch Processing: Runtime increased to 0.73s, more than double the base.**
- **MPS with Threading: Improved to 1.61s.**

MNIST Dataset (CNN)

- **Plain CNN: No optimizations, runtime of 316.18s.**
- **MPS with Batch Processing: Reduced runtime by 72% to 64s.**
- **CUDA with Batch Processing: Faster than the base CNN at 148.9s but slower than MPS.**

Accuracy

- **Consistently high accuracy (>98%) across all platforms and datasets.**

Key Takeaways

- **Optimizations Matter: Batch processing and threading significantly reduce training time, especially for complex models like CNNs.**
- **Hardware Acceleration: GPU (CUDA) speeds up processing, but MPS outperforms CUDA for the MNIST dataset.**
- **Accuracy Consistency: Optimization techniques do not compromise model accuracy.**
- **Trade-offs: Optimizations require more complex setups or additional hardware.**
-

Graph Analysis (LLM):

1. Evaluation Loss (eval/loss)

Interpretation:

- **Start: Evaluation loss ~0.82 at step 350.**
- **Trend: Linear decrease in loss.**
- **End: Evaluation loss just below 0.74 at step 650.**

2. Evaluation Runtime (eval/runtime)

Interpretation:

- **Start: Runtime ~11.324s at step 350.**
- **Trend: Linear increase in runtime.**
- **End: Runtime ~11.336s at step 650.**

3. Evaluation Steps per Second (eval/steps_per_second)

- Interpretation:

- **Start: Steps per second ~3.444 at step 350.**
- **Trend: Linear decrease in steps per second.**
- **End: Steps per second ~3.44 at step 650.**
-

4. Evaluation Samples per Second (eval/samples_per_second)

- Interpretation:

- **Start: Samples per second ~13.685 at step 350.**
- **Trend: Linear decrease in samples per second.**
- **End: Samples per second ~13.675 at step 650.**

Summary

Overall, we can see that parallelization overhead is not that easy to minimize as it does induce a heavy price. In the case of CNNs it is not viable to run CNNs without any sort of acceleration thus the extreme execution time in base CNN. In Simple NNs we don't really need to introduce parallelism because it just increases execution time without any real benefits however the story is different for large CNN models.

The projects objectives were to try and parallelize an already existing code that we have written and to see what our results will be. In doing so we realized that parallelization is not that easy of a job and parallelization should be done on very specific tasks where we can minimize the cost induced by parallelization.

Potential areas of further research

- Work more with CNNs to see how well we can implement parallelism in them.

- Use CUDA more on google colab it offers more customization than what is possible with MPS currently.
- Conduct scalability studies to understand the limits of parallelism in neural network training. This involves testing with even larger datasets and more complex models to determine the maximum efficiency and performance gains.
- Investigate the use of other hardware accelerators like Tensor Processing Units (TPUs) and FPGA-based accelerators for neural network training to compare their performance with GPUs and MPS.

Evaluation Metrics: Show slight increases in runtime and decreases in steps and samples per second.

Training Progress: Global step count increases steadily, showing expected training progression.

Recommendations: Increase logging frequency for better monitoring, ensure consistent monitoring of key metrics, and consider gradient norm regularization techniques to maintain stability.

Future Strategy Adjustments

For future developments, considering the integration of model and pipeline parallelism could provide a more nuanced approach to reducing computation times and improving model training efficiency. These strategies would involve dividing the neural network model in ways that allow simultaneous processing of different stages or layers across multiple processors or cores.

Hybrid Parallelism: Combine data parallelism and model parallelism to optimize the training process for extremely large models. This hybrid approach can balance the computational load and memory usage across multiple devices.