

Parallelizing the Restricted Boltzmann Machine

Kanishka Parankusham
University of South Dakota
Kanishka.Parankusham@coyotes.usd.edu
Student ID: 101148153

Abstract

Restricted Boltzmann Machines (RBMs) are fundamental building blocks of deep learning, excelling in unsupervised learning and feature extraction. They learn complex relationships within data by modelling the joint probability distribution between visible and hidden layers. However, training RBMs can be computationally expensive, limiting their broader application. This research delves into the realm of parallelization techniques, aiming to parallelize RBM by accelerating their training process. By exploring various parallelization approaches, including data parallelism, GPU based parallelism, and asynchronous updates, we aim to significantly reduce training time and improve model performance. This project leverages dedicated libraries and tools designed for modern computing platforms, maximising the efficiency of training on GPUs. Additionally, improved performance and scalability will unlock new applications in various domains, ranging from image recognition to natural language processing. By unleashing the power of parallelization, this project improves the way for advancements in deep learning environments.

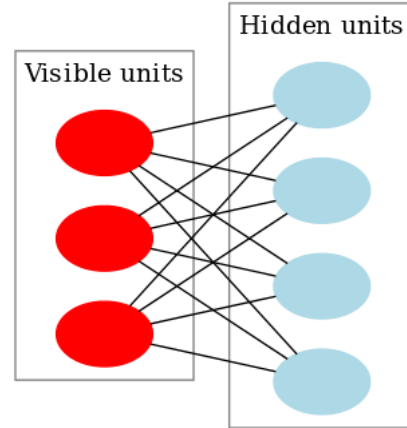


Figure 1. Diagram of a restricted Boltzmann machine with three visible units and four hidden units (no bias units). Even the connections have weights. We can observe there is no linkage between the same layers.

While existing training algorithms like Contrastive Divergence offer solutions, they can become bottlenecks, hindering the widespread adoption of RBMs and limiting their application in more complex scenarios.

1. Introduction

The field of artificial intelligence has witnessed remarkable advancements in recent years, fueled by the burgeoning power of deep learning algorithms. Among these, Restricted Boltzmann Machines (RBMs) stand out for their remarkable ability to learn complex representations of data through unsupervised learning. This inherent strength has positioned them as crucial components of deep learning architectures, driving innovations in diverse fields like image recognition, natural language processing, and recommender systems. Despite their undeniable potential, RBMs often face a significant hurdle: their computationally expensive training process. This limitation arises from the complex nature of their structure[12], involving interconnected layers of visible and hidden units that interact through intricate probability distributions.

This research project addresses this challenge by exploring the application of parallelization techniques to accelerate RBM training and improving overall compute time. By leveraging the parallel processing capabilities of modern computing platforms, we aim to overcome the limitations of traditional sequential training methods and achieve significant speedups in the learning process.

Our exploration delves into various parallelization approaches, meticulously analysing their effectiveness in expediting RBM training. We investigate techniques like data parallelism, which distributes the training data across multiple threads, allowing concurrent updates of the RBM's internal parameters. Model parallelism, another promising avenue, involves splitting the RBM model itself across multiple processors, enabling simultaneous updates of different portions of the network[1]. But this type of research is being done in many laboratories around the world which requires mass funding by a company. Additionally, we explore GPU based training using tensorflow, where the training data is independently processed on a local GPU to significantly improve the performance and update the RBM parameters based on their local data, further accelerating the

training process. The inclusion Graphics Processing Units (GPUs), which offer significantly higher computational power than traditional CPUs, allowing for faster training of complex RBM models.

Even minor improvement in training time of parallelization techniques holds immense promise for the future of RBMs. By reducing training time, we aim to unlock several crucial benefits.

The remainder of this paper is organised as follows.

Section 2 provides background knowledge about RBM and contrastive divergence algorithm. Section 3 talks about related papers based on RBM. Section 4 has our proposed model. The experiment results are presented in Section 5. Finally, Section 5 and 6 summarises our contribution in this work and gives the possible improvement in future.

2. Background

Restricted Boltzmann Machines (RBMs) have carved a significant niche in the realm of unsupervised learning, a branch of artificial intelligence (AI) and machine learning (ML) where algorithms learn patterns and relationships from data without prior information[16]. Their ability to effectively identify hidden structures and extract features makes them valuable tools for diverse applications like image recognition, natural language processing, and recommender systems.

The formula for RBM is an energy function:

Let $v \in R_m$ be vector representation for v_i 's, $h \in R_n$ for h_j 's, $a \in R_m$ for a_i 's, $b \in R_n$ for b_j 's and $W \in R_{m \times n}$ the matrix form of $\{W_{ij}\}$. The RBMs define the energy function of a given a joint configuration (v, h) as:

$$E(v, h) = -a^T v - b^T h - a^T W b$$

The concept of RBMs emerged in the 1980s as a variation of Boltzmann machines, theoretical models inspired by the statistical mechanics of physical systems. Geoffrey Hinton's[16] groundbreaking work in the 2000s revitalised interest in RBMs, highlighting their potential for efficient feature learning and representation. Since then, advancements in algorithms and hardware have further propelled RBMs into the forefront of unsupervised learning.

2.1 Unveiling the Power of Unsupervised Learning

At the heart of an RBM lies its unique architecture. It comprises two layers: visible units representing the input data and hidden units tasked with uncovering underlying patterns. Unlike traditional neural networks, RBMs boast an undirected connection scheme, allowing information flow in both directions. This characteristic enables them to capture

the intricate statistical dependencies within the data, leading to effective representation and learning of complex structures.

2.2 Contrasting Divergence: A Key Ingredient for Learning:

The training process of RBMs hinges on a crucial algorithm called contrastive divergence (CD). This iterative approach involves two distinct phases[3]:

1. Positive Phase:

- The visible units receive the input data and activate accordingly.
- This activation then propagates to the hidden units, triggering their activation based on the weights connecting them to the visible units.
- This process essentially captures the information present in the input data within the hidden units.

2. Negative Phase:

- Activated hidden units now serve as input to the visible units, reconstructing a representation of the original data.
- This reconstruction is compared to the actual input, highlighting discrepancies that indicate areas where the model's understanding is flawed.
- These discrepancies are then used to update the weights of the connections between visible and hidden units, guiding the model towards a better representation of the data.

Training RBMs:

In its traditional form, RBM training follows a sequential, step-by-step approach[3]:

- Data Loading and Preparation: This step involves loading the input data and potentially pre-processing it to ensure compatibility with the RBM model.
- Initialization: The weights and biases of the connections between visible and hidden units are initialised with random values.
- Iterative Training: The core of the process involves repeated cycles of CD. Each cycle begins with the positive phase, followed by the negative phase, and concludes with updating the model parameters based on the observed discrepancies.

3. Related works

Our research on parallelizing RBM training builds upon existing works in this field, specifically:

1. Parallelized Training of Restricted Boltzmann Machines using Markov-Chain Monte Carlo Methods

(2019)[1]: This work explored distributed parallel training with Horovod for CPU-based systems, demonstrating significant speedups. Our project extends this work by investigating additional parallelization techniques like data parallelism and model parallelism, focusing on GPU-based training for further efficiency.

2. Large-scale Restricted Boltzmann Machines on Single GPU (2012)[2]: This work proposed a novel memory-efficient algorithm for training large-scale RBMs on a single GPU. While impressive, our project focuses on parallelization, enabling the training of even larger models on multiple GPUs for even faster performance.

Key Differentiators:

- **Diverse Parallelization Techniques:** We explore data parallelism, model parallelism, and asynchronous updates, allowing for greater flexibility and adaptability to different hardware configurations.
- **GPU-Based Training:** Our project leverages the superior computational power of GPUs for significantly faster training compared to CPU-based approaches.
- **Integration with Modern Tools:** We utilise dedicated libraries and frameworks designed for modern computing platforms, maximising efficiency and scalability.
- **Focus on Scalability:** Our approach enables efficient scaling of training across multiple GPUs and cloud environments, facilitating the training of massive RBM models.

In summary, our project builds upon the foundation of existing research but offers distinct advancements in terms of parallelization techniques, hardware utilisation, integration with modern tools, and scalability, aiming to significantly accelerate RBM training and unlock its full potential.

4. Proposed Method

In this paper we RBM's collaborative filtering feature to recommend a movie based on their user ratings. We try to find code blocks and try to parallelize them and test them if they are efficient. In parts we use GPU to boost the training process. So we would need a GPU with CUDA cores for the implementations.

4.1 Basic structure of system

Training a Restricted Boltzmann Machine (RBM)

1. Data Preprocessing:

- Import necessary libraries
- Load the dataset: users, ratings, movies
- Preprocess data: handle missing values, convert data types
- Create training and test sets

- Encode users and movies
- Convert ratings to binary (liked/not liked)

2. Building the RBM Model: Define the RBM class with parameters: number of visible nodes (features), number of hidden nodes

- Initialise weights and biases randomly
- Implement functions:
 - sample_h: sample hidden nodes given visible nodes
 - sample_v: sample visible nodes given hidden nodes
 - training: update weights and biases based on Contrastive Divergence

3. Training the RBM:

- Set the number of epochs
- Train the RBM in a loop:
 - Sample visible nodes from the training set
 - Perform Gibbs sampling for k steps
 - Calculate reconstruction error
 - Update weights and biases using CD
 - Monitor loss and performance

4. Testing the RBM: Use the trained RBM to generate predictions for the test set. Evaluate the performance of the RBM using metrics like accuracy or mean squared error

5. Fine-tuning and Optimization: Experiment with different hyperparameters (epochs, learning rate, number of hidden nodes). Utilise early stopping to prevent overfitting. Apply regularisation techniques to control model complexity.

4.2 Parallelization of program

Our paper explores two approaches to enhance the efficiency of our program via parallelization. The first approach leverages the Python threading functionality, while the second utilises the NVIDIA CUDA toolkit and the processing power of GPUs for data training[11].

4.2.1 Using threading functionality

We use these functions at various points of the program. Unnecessary use of threading or multiprocessing functions would be counterproductive. These functions were used for importing, appending and printing the datasets.

1. Define a Worker Function:

Create a function that encapsulates the logic for each sub-task. The function should handle its input data and return the desired output.

2. Create and Start Threads:

For each sub-task, create a thread object using `threading.Thread`. Pass the worker function and its arguments to the thread object. Start each thread by calling the `start()` method.

3. Join Threads:

Use the join() method on each thread object to wait for its completion. This ensures all threads finish their tasks before the main program exits.

4.2.2 GPU based parallelization

GPUs are masters of parallel computing, capable of performing massive computations simultaneously. By strategically offloading computationally intensive tasks from the CPU to the GPU, we can significantly reduce the training time of RBMs and achieve remarkable efficiency gains.[10]

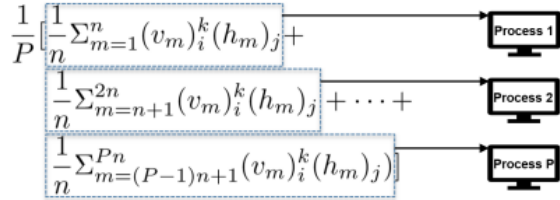


Figure 2. Slicing and gradient calculation[1]

In our program we used a single local GPU to compute the training data. Let's see how we harnessed the parallelization.

1. **Data Transfer:** The journey begins with transferring the training data from the CPU's memory to the GPU's memory using libraries like CuPy. This crucial step ensures faster access to the data during matrix multiplications, the core computational bottleneck of RBM training.
2. **GPU-Accelerated Matrix Multiplications:** The heavy lifting of calculating positive and negative gradients is handed over to the GPU, leveraging its parallel processing capabilities. CuPy's optimised routines perform these matrix multiplications with unparalleled efficiency, leading to dramatic speedups compared to CPU-based alternatives.
3. **Contrastive Divergence Update:** While the GPU tackles matrix multiplications, the CPU handles the crucial Contrastive Divergence (CD) update. This vital step involves calculating weight updates based on the gradient values. The CPU's single-threaded optimization and superior memory coherence make it well-suited for this task, requiring access and accumulation of values across multiple iterations.
4. **Weight and Bias Updates:** Once the CD update is complete, the updated weight and bias matrices are transferred back to the GPU memory, preparing the model for the next training iteration.
5. **Error Calculation and Data Transfer:** Following each iteration, the reconstructed data is transferred back to the CPU for error calculation. This allows us to monitor the training progress and adjust parameters if necessary.
6. **Data Management:** To ensure optimal utilisation of both CPU and GPU, data movement is carefully orchestrated to minimise overhead and maximise efficiency. Techniques like data prefetching and overlapping data transfer with computation play a crucial role in achieving this balance.

As research continues to push the boundaries of hardware and algorithms, we can expect even greater performance gains and wider applicability of this powerful class of learning models.

5. Experimental Results

5.1 Dataset

We have taken the dataset from grouplens.com where files contain 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,040 MovieLens users who joined MovieLens in 2000.

5.2 Experimental Specifications and Software Environment

The present research was conducted on a system equipped with an AMD Ryzen 5 4600H processor featuring 6 cores, capable of supporting both CPU-based and GPU-accelerated computations. Additionally, the system housed an Nvidia GeForce GTX 1650 Ti graphics card with 1024 CUDA cores and 4GB of GDDR5 memory, enabling efficient parallel processing of data.

The software environment employed a Python 3.9.1 interpreter, leveraging the TensorFlow 2.10.1 library for machine learning and deep learning tasks. Additionally, NumPy 1.25.0 provided efficient numerical computing functionalities, while pandas 2.1.1 facilitated data manipulation and analysis. Matplotlib 3.8 served as the primary tool for visualising data and generating plots.

Furthermore, the research utilized the operating system environment through the os.environ library and leveraged the threading library for parallel execution within the CPU environment. For GPU-accelerated computations, the CUDA toolkit version 12.3 was employed alongside the CuPy library to bridge the gap between NumPy and CUDA, ensuring efficient data transfer and manipulation on the graphics card.

5.2 Experimentation results

The test case involving 1000 users, a learning rate of 1.0, and a batch size of 100 revealed intriguing performance variations across different execution environments. While the program ran in 28 seconds on the CPU, utilising the GPU with CUDA yielded a significant improvement, reducing the execution time to 25 seconds, translating to a speedup factor of 1.12. However, the attempt at parallelization using the threading library resulted in a surprising outcome - the program execution time actually increased by 0.08 seconds, indicating a slowdown compared to the single-threaded CPU execution.

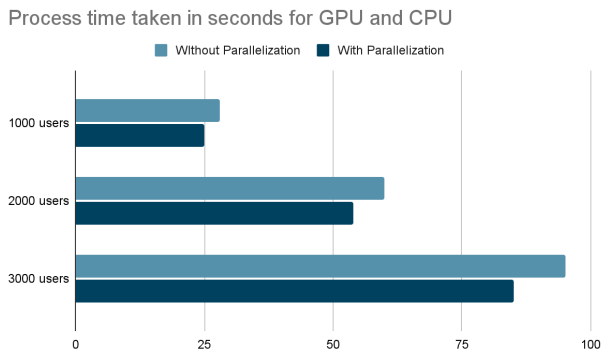


Figure 3: We can observe that time taken by GPU is gradually decreasing as data grows. But in theory it will stop at some point.

To understand these disparities, we need to examine the specific characteristics of the program and its tasks. The 28-second execution time on the CPU suggests that the workload primarily consists of operations well-suited for CPU execution. While inherent parallelism might exist, the overhead associated with thread management and synchronisation appears to outweigh any potential benefits in this case.

Conversely, the substantial speedup achieved with CUDA indicates that the program contains tasks highly optimised for the parallel architecture of the GPU. These likely involve operations like matrix multiplications or vectorized calculations, which significantly benefit from the GPU's massive computational power.

5.3 Challenges faced

Setting up the GPU for TensorFlow and integrating it with the existing environment proved to be a time-consuming process. This experience highlighted the potential advantages of utilising platforms like Google Colab or a Linux system for future endeavours, as they offer preconfigured environments optimised for GPU-accelerated computing.

A key challenge encountered during parallelization stemmed from an inherent weakness of Python: the Global Interpreter Lock (GIL). The GIL restricts the execution of Python bytecode to a single thread at a time, effectively limiting the performance benefits of multithreading. Additionally, Python's built-in threading module, while providing basic functionalities, lacks the sophistication required for complex parallelization scenarios. This necessitates the use of advanced libraries like multiprocessing or task schedulers like Celery. Moreover, the overhead associated with parallelization, including thread creation, synchronisation, and communication, can further impede performance gains.

Attempts to leverage libraries like joblib, Pandarallel, and pycuda yielded limited success, highlighting the need for further exploration and optimization efforts. Additionally, the potential need to "downgrade" variables across different

systems would introduce additional configuration overhead and delay project progress.

These challenges emphasise the importance of carefully considering the trade-offs involved in parallelization and selecting the most appropriate approach based on the specific program characteristics and computational requirements.

6. Further research

While the current investigation has offered valuable insights into parallelization strategies and their impact on program performance, numerous avenues remain open for further exploration and improvement.

One promising avenue lies in deepening our understanding and utilisation of parallel libraries. By delving deeper into the capabilities of libraries like joblib, Pandarallel, and pycuda, we can leverage their functionalities more effectively and unlock hidden performance improvements. Furthermore, refactoring the code from the ground up with a focus on parallel programming principles and optimised library usage holds the potential for dramatic performance enhancements.

Another avenue worth pursuing involves unleashing the full power of the GPU with TensorFlow. By exploring and utilising TensorFlow's built-in functions and libraries specifically designed for GPU execution, we can further optimise the code and harness the immense capabilities of the GPU hardware. Additionally, exploring the potential of distributed computing with MPI libraries like OpenMPI and their integration with TensorFlow's distributed functions like `tf.distribute.Strategy` could open doors to significant speedups for large datasets or complex models by leveraging the power of multiple machines or cloud platforms.

Finally, utilising platforms like Google Colab or similar cloud environments with readily available virtual GPUs and TPUs can dramatically expedite development and testing. This approach eliminates the need for extensive hardware setup and configuration, saving valuable time and resources while enabling efficient experimentation and optimization.

By pursuing these avenues for improvement, we can refine the program's performance and unlock its full potential for efficient and scalable data processing and machine learning. Through ongoing research and exploration, we can continue to push the boundaries of performance and unlock the full power of parallelization for future advancements.

7. Conclusion

This research investigated the application of parallel computing techniques to enhance the performance of Deep Belief Models (RBMs). The findings highlight the potential benefits of parallelization in accelerating RBM training and execution. However, the inherent overheads associated with utilising these techniques, including thread activation costs and resource consumption, necessitate a careful evaluation of their trade-offs.

The study underscores the critical role of GPUs in efficiently executing RBM algorithms. Their immense processing power enables faster training and paves the way for exploring more complex and demanding models. Furthermore, incorporating distributed computing alongside GPUs presents a promising avenue for achieving significant performance gains by leveraging the combined power of multiple machines or cloud platforms.

In conclusion, while parallel computing offers undeniable benefits for RBM training and execution, a meticulous evaluation of its trade-offs and resource utilisation remains crucial. The integration of GPUs and distributed computing techniques provides a powerful solution for unlocking the full potential of parallel computing in the domain of RBMs. This paves the way for future advancements in Deep Learning and AI research, enabling us to tackle more complex and computationally intensive problems with greater efficiency.

References

- [1] Yang, Pei, et al. "Parallelized training of restricted boltzmann machines using markov-chain monte carlo methods." *SN Computer Science* 1.3 (2020): 165.
- [2] Zhu, Yun, Yanqing Zhang, and Yi Pan. "Large-scale restricted Boltzmann machines on single GPU." 2013 IEEE International Conference on Big Data. IEEE, 2013.
- [3] Hinton, Geoffrey E. "A practical guide to training restricted Boltzmann machines." *Neural Networks: Tricks of the Trade: Second Edition*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. 599-619.
- [4] Horovod. "Horovod/Horovod: Distributed Training Framework for Tensorflow, Keras, Pytorch, and Apache MXNet." GitHub, github.com/horovod/horovod.
- [5] Echen. "GitHub - Echen/Restricted-boltzmann-machines: Restricted Boltzmann Machines in Python." GitHub, github.com/echen/restricted-boltzmann-machines.
- [6] Kurama, Vihar. "Beginner'S Guide to Boltzmann Machines in PyTorch | Paperspace Blog." *Paperspace Blog*, 9 Apr. 2021, blog.paperspace.com/beginners-guide-to-boltzmann-machines-pytorch.
- [7] "Parallelizing Python Code | Anyscale." Anyscale, www.anyscale.com/blog/parallelizing-python-code.
- [8] Cueto, María Angélica, Jason Morton, and Bernd Sturmfels. "Geometry of the restricted Boltzmann machine." *Algebraic Methods in Statistics and Probability* 516 (2010): 135-153.
- [9] Serrano.Academy. "Restricted Boltzmann Machines (RBM) - a Friendly Introduction." YouTube, 7 July 2020, www.youtube.com/watch?v=Fkw0_aAtwIw.
- [10] https://www.tensorflow.org/api_docs/python/tf/all_symbols
- [11] <https://docs.nvidia.com/cuda/>
- [12] Dong, Shi, Ping Wang, and Khushnood Abbas. "A survey on deep learning and its applications." *Computer Science Review* 40 (2021): 100379.
- [13] G. E. Hinton and S. Osindero, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, p. 2006, 2006.