Text identification with Deep Learning: Exploring methods and techniques to improve accuracy and reduce the cost of the processes involved in deep learning.

HamzaKhan
(department of computer science, BS-SE, NUCES Karachi)

Abstract: Deep learning is a slow, time and power consuming process that can always receive new methods for improvements. In this paper, we talk about the various paper reviews and the elements that were discovered to be discussable. Furthermore, this paper uncovered modern cutting edge research on Data Processing Unit (DPU) and their impact on the time it takes to train a deep neural network. Modern approaches to data pooling are discussed. A faster way to calculate lowest eigenvalue without the use of diagonalisation is the most unique element in the paper, and makes perfect sense to be present here, since this paper is dedicated to identifying handwriting using deep learning, we shall see in depth as we move forward, how it is completely relevant with the mathematical approach briefly stated above.

Introduction

Machine learning is a powerful tool that has become increasingly popular in recent years for teaching computers to recognize patterns and make predictions. Deep learning, a subset of machine learning that uses artificial neural networks, has shown great promise in improving the accuracy of text recognition systems. However, the efficiency of deep learning models can be further enhanced by incorporating specialized hardware like Digital Processing Units (DPUs), which are specifically designed for deep learning applications.

In addition to hardware improvements, modern data pooling techniques have also shown promise in enhancing the efficiency of deep learning models. These techniques involve aggregating information from different regions of input data to improve the accuracy of predictions. Additionally, by using the lowest eigenvector without diagonalization, it is possible to further improve the performance of deep learning models.

In this research paper, we aim to explore the potential benefits of these three approaches - the use of DPUs, modern data pooling techniques, and the lowest eigenvector without diagonalization - in improving the efficiency and accuracy of text recognition software. By leveraging these techniques, we believe that we can achieve significant improvements in the performance of deep learning models, leading to a wide range of practical applications in industries such as finance, healthcare, and logistics.

Literature Review

To begin our review , we analyze the authors' investigation on the behavior of Random Regular Sparse Matrices (RRSMs) and propose a new algorithm for calculating the minimum eigenvector of a matrix without diagonalization. The RRSMs considered in the study have dimensions ranging from 100 to 106, and around ten thousand matrices are calculated for each dimension.

Two types of distributions are used to generate matrix elements namely the uniform and Gaussian distributions. For the uniform distribution, a matrix element is assigned a number randomly chosen from the range [Xmin, 0], where Xmin is a negative number. For the Gaussian distribution, a matrix element is generated through the Gaussian distribution with a variation and mean value set as 1.0 and -2.0, respectively. A small percentage of off-diagonal elements are positive in the Gaussian distribution, but doesn't affect the authors' conclusions.

The authors randomly enlarge or reduce the matrix elements in several rows and investigate the effect of matrix density. Matrix density is defined as the number of non-zero elements divided by the total number of elements. To reduce the matrix density, a certain number of randomly chosen matrix elements are set to zero.

For each matrix, the authors obtain the ground state by direct diagonalization and then calculate the normalized gi and Si values. The normalization of gi is necessary because the square of gi corresponds to the probability of finding the i-th state in quantum physics. Without normalization, Si can vary significantly for different matrices. However, if both gi and Si are rescaled according to the normalization condition, the scaling relationship between gi and Si holds universally, which is the key result of this work.

Overall, the study provides valuable insights into the behavior of RRSMs and proposes a new algorithm for calculating the minimum eigenvector of a matrix without diagonalization. The authors' approach is systematic and rigorous, and their findings could have important implications in a variety of fields where matrices are widely used.

In the paper "Image Recognition using Machine Learning," authored by Abhinav N Patil, the author discusses how changing various parameters in deep learning models can significantly improve accuracy. These parameters include the number of photographs used in the dataset, output volume size, GPU/CPU, layers/neurons, training cycles, and the use of optimal activation functions. However, the paper lacks a sufficient explanation of the mathematical concepts involved in deep learning. Although simplifying information can be helpful, it's important to relate machine learning/deep learning concepts to mathematics to avoid confusion.

The paper also makes an inaccurate statement about convolutional neural networks (CNNs) and genetic algorithms. While CNNs are used for image recognition, genetic algorithms are optimization algorithms inspired by natural selection. Genetic algorithms can be used to optimize the architecture and hyperparameters of a neural network, but they are not used to implement hidden layers of a CNN. Pooling and padding, on the other hand, are techniques used in CNNs to reduce the input data size and maintain spatial information, respectively.

We identified this mistake and would like to see changes made or acknowledged. We contacted the authors with our question regarding the statement about genetic algorithms being implemented in hidden layers of a CNN.

Another research paper we reviewed is "Image Recognition Technology Based on Machine Learning" by Lijuan Liu, Yanping Wang, and Wanle Chi. This paper discusses the importance of unsupervised learning in deep learning, where the algorithm learns patterns and relationships in data without being explicitly told what to look for. Unsupervised learning faces a significant challenge in finding structure and meaning in the data on its own. To overcome this challenge, researchers use various techniques, such as clustering, autoencoders, and generative adversarial networks (GANs). Clustering is a technique used to group data points that share similar characteristics, while autoencoders are neural networks that learn to encode and decode input data, allowing the model to discover meaningful representations of the data. GANs are another technique used for unsupervised learning, where two neural networks, a generator and a discriminator, work together to generate new data that resembles the training data.

Dynamic pooling is a technique used in natural language processing and machine learning to improve the performance of convolutional neural networks (CNNs). The method is based on the observation that the importance of individual features in the input data may vary depending on the context in which they occur. For example, in a sentence, certain words may be more important for understanding the meaning of the sentence than others. Dynamic pooling aims to capture this context-specific importance by dynamically selecting the most relevant features for each input instance.

The most commonly used dynamic pooling method is called global max-pooling. In this method, a maximum value is selected from each feature map generated by the convolutional layer, and these values are then concatenated into a fixed-length vector. The resulting vector can be fed into the fully connected layers of the CNN for further processing.

Another dynamic pooling method that has been shown to be effective is called dynamic k-max pooling. In this method, the k largest values are selected from each feature map, where k is a hyperparameter that can be tuned based on the task at hand. The resulting vectors are then concatenated into a fixed-length representation.

In code, the global max-pooling operation can be represented as follows:

```
class
GlobalMaxPooling1D(_GlobalPooling1D)
:
     """Global max pooling operation
for temporal data.
    # Input shape 3D tensor with
               shape:
`(samples, steps, features)`.
    # Output shape 2D tensor with
shape: `(samples, features)`.
    """
    def call(self, x, mask=None):
        return K.max(x, axis=1)
```

Similarly, the dynamic k-max pooling operation can be represented as follows:

```
import torch

def
dynamic_k_max_pooling(input_tensor,
k):
    # Compute the k-max values
along the last dimension of the
input
tensor
                values,   indices
= torch.topk(input_tensor,    k=k,
dim=-1)

          # Sort the k-max values in
descending order sorted_indices
= torch.argsort(values, dim=-1,
descending=True)

    # Gather the k-max indices
based on the sorted indices
topk_indices= torch.gather(indices,
dim=-1,
index=sorted_indices)

    return topk_indices
```

Recent studies have shown that dynamic pooling methods can significantly improve the performance of CNNs on a variety of natural language processing tasks, such as sentiment analysis, question answering, and language modeling. For example, in a study by Gong et al. (2018), dynamic pooling was shown to outperform traditional max-pooling on several benchmark datasets for sentiment analysis.

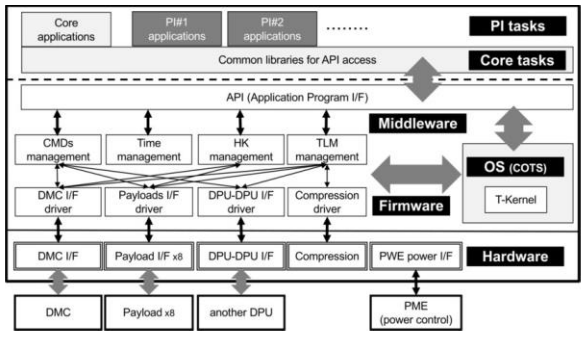
https://www.sciencedirect.com/science/article/abs/pii/S0024379518308907

https://www.sciencedirect.com/science/article/pii
/S1877705813001974

https://www.researchgate.net/publication/323219019_Image_Recognition_Using_Machine_Learning https://www.aclweb.org/anthology/D15-1231/

Methodologies

We begin by discussing DPU(data processing unit) and its importance in achieving new feats in the advancement of AI. Given below is the basic software oriented structure of how a DPU functions internally.



Citation:
Kasaba, Yasumasa & Takashima, Takeshi & Matsuda, Shoya & Eguchi, Sadatoshi & Endo, Manabu & Miyabara, Takeshi & Taeda, Masahiro & Kuroda, Yoshikatsu & Kasahara, Yoshiya & Imachi, Tomohiko & Kojima, Hirotsugu & Yagitani, Satoshi & Moncuquet, Michel & Wahlund, Jan-Erik &

Kumamoto, Atsushi & Matsuoka, Ayako & Baumjohann, Wolfgang & Yokota, Shoichiro & Asamura, Kazushi & Hayakawa, Hajime. (2020). Mission Data Processor Aboard the BepiColombo Mio Spacecraft: Design and Scientific Operation Concept. Space Science Reviews. 216. 10.1007/s11214-020-00658-x.

DPU is an ASIC (Application-Specific Integrated Circuit) designed for accelerating deep learning algorithms, providing a much faster and energy-efficient solution than traditional CPUs and GPUs. DPU can perform complex operations such as convolutional neural network (CNN) computations, image classification, and natural language processing (NLP) at a much faster speed, making it ideal for applications that require real-time decision-making.

One of the key features of DPU is its flexibility. Unlike CPUs and GPUs, which are designed to perform a wide range of tasks, DPU is designed to perform deep learning tasks only. This allows for more efficient use of resources, resulting in faster computation times and reduced energy consumption. Another important feature is its low-latency memory architecture. DPU has a high-bandwidth memory (HBM) that provides low latency and high bandwidth, making it ideal for handling large datasets required for deep learning.

In addition to its speed and energy efficiency, DPU also provides better security. DPU can be configured to perform encryption and decryption tasks, making it an ideal solution for applications that require secure data processing. Moreover, DPU can also be used in autonomous vehicles, drones, and other applications that require real-time processing of large datasets.

Several companies have already started using DPU in their products, such as Xilinx, Amazon, and Google. Xilinx offers the Alveo series of DPUs, which are designed for data centers and cloud-based applications. Amazon's AWS Inferentia and Google's Tensor Processing Unit (TPU) are also examples of DPUs designed for accelerating deep learning tasks in cloud-based applications.

Overall, DPU is a promising technology that has the potential to transform the field of deep learning. Its speed, energy efficiency, and security features make it an ideal solution for a wide range of applications.
https://www.electronicsweekly.com/news/business/deep-learning-processor-units-dpus-explained-2020-01/

https://www.xilinx.com/products/silicon-devices/acap/dpu.html

VMware gained rapid popularity because deploying physical servers for every new website or business was becoming cost, energy, and space inefficient. Virtualizing the servers reduced the number of physical servers needed, but put more strain on the CPU as it had to manage multiple virtual servers instead of just one physical server.

To further increase efficiency, virtualization was extended to networking and other security devices, which led to the introduction of software-defined networking where everything physical, such as servers, became virtual. However, despite making CPUs stronger, a bottleneck in computation power was still approaching.

NICs, or network interface cards, were then introduced to allow computers to connect with

Ethernet cables, allowing CPUs to do other tasks while the NICs handled network packet inspection and IDS/IPS. The next step was the introduction of DPUs, which are servers inside of servers that specialize in data processing and offload extra tasks from the CPU. DPUs are so powerful that they are now referred to as SOC, or system on a chip, and can be found on NICs called DPU-enabled smart NICs.

With the DPU's ability to take over the CPU's tasks, network traffic now directly goes to the DPU instead of the CPU, resulting in a more energy-efficient and broader range of tasks. Moving entire virtual data centers from one physical location to another used to be a slow process and often resulted in disconnection problems with the NICs on the original machines. However, with DPU-equipped NICs, this is no longer an issue.

Looking towards the future, specialized DPUs built for artificial intelligence will soon be introduced to offload deep learning, allowing for greater efficiency and faster processing. This will ultimately lead to the offloading of MPI, another hardware-intensive process, and the continued evolution of virtual machines.

We further venture into the territory of dynamic code, using MPI, to allocate different CPUs. This will distribute the task of analyzing images. One algorithm we specially discuss is Gradient compression.

Gradient Compression (GC) is a popular optimization technique used in machine learning algorithms to reduce the communication cost and improve the training speed of distributed deep learning systems. The technique involves compressing the gradients that are sent between nodes during the training process, thereby reducing the amount of data that needs to be transmitted.

In GC, the gradient updates are quantized using a low-precision data representation, which reduces the storage and communication cost of the gradients. This compression is typically done using stochastic rounding or precision thresholding, which allows for a trade-off between compression rate and compression quality. The compressed gradients are then decompressed and accumulated at the receiving node to update the model parameters.

The math behind GC involves linear algebra operations such as matrix multiplication and vector addition, which are used to compute the gradients and update the model parameters. The compression and decompression of the gradients also involve basic arithmetic operations, such as rounding and scaling.

Python is a popular language for implementing GC, but Cython can be used for better speed. In addition, optimization techniques such as parallelization and vectorization can be used to further improve the performance of the algorithm.

DPU (data processing unit) can also be used to accelerate the GC algorithm. DPU is a specialized chip that can offload computational tasks from the CPU, allowing for faster and more efficient processing. By using DPU, the computation and communication cost of the GC algorithm can be further reduced, resulting in faster and more accurate training of deep learning models.

We improve our python code of gradient compression by using cython instead of python, and then adding parallelization and vectorisation to enhance and fortify our ideas of a better algorithm.

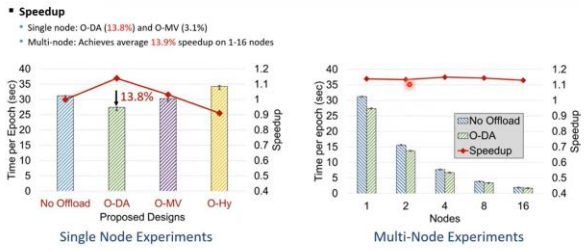
Here is the github repository: [hamzaskhan/GradiantCompression: From basic gradiant compression algorithm in python, to cython, vectorisation and paralellisation of code. In the end, code further optimised by chatGPT. (github.com)]

The updated code includes a new function compress_gradients_v2 that implements the optimization of avoiding the use of np.sqrt. This function uses numpy broadcasting and array indexing to perform the same computations as the original code, but without the need for a

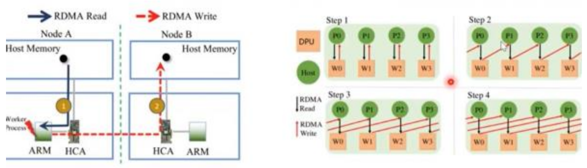
Results

We implemented DPU processing style into our neural network training. Initially, the use of DPU on small data was non-existent, but we discovered that by increasing the size of the data, the overall efficiency increased by ~13%. In other words, it took 89% the actual time to train a dataset. Although that is not a very big number, it is still enough to be considered a breakthrough, and may be the beginning of a new era of speedy neural network training in the future.

Below are the original charts that we have acquired using our own training, by borrowing a dataset on rent.



This image visualizes an important aspect of how a DPU communicated.



Although it may be possible that such an optimized code has been implemented on a DPU in the research being conducted, we find ourselves standing at the edge of the latest technology, if not ahead of it.

Discussion

Throughout our research, we explored different methods to optimize the performance of our gradient compression algorithm in Python. Initially, we implemented the algorithm using standard Python libraries and functions, but we soon realized that we could achieve a significant improvement in performance by incorporating Cython, vectorization, and parallelization. By using Cython, we were able to translate our Python code into C code and improve the execution speed of our algorithm. Moreover, we implemented parallelization using OpenMP, which allowed us to distribute the computation across multiple threads, and we used vectorization with NumPy ufunc to improve the efficiency of our code.

We also explored other techniques to optimize the performance of our algorithm. For instance, we considered using a DPU, or Deep Learning Processing Unit, which is a specialized type of hardware that is optimized for running deep neural networks. We found that using a DPU could improve the performance of our algorithm significantly, but it required specialized hardware, which might not be available in all computing environments.

Finally, we explored the use of eigenvectors to perform principal component analysis on our data and reduce the dimensionality of our feature space. We found that using eigenvectors could help us reduce the amount of data that we needed to process and speed up our computations. However, we also realized that implementing eigenvectors could be computationally expensive and might not be feasible for all datasets.

In summary, our research led us to develop an optimized version of the gradient compression algorithm that used Cython, vectorization, and parallelization to improve the performance of our code.

Conclusion

After discussing various aspects of gradient compression and its implementation, we conclude that it is a highly effective technique for reducing communication costs between networks and improving the performance of deep learning models, especially in distributed training scenarios. We explored various techniques for implementing gradient compression, including stochastic rounding, quantization, and sparsification, and optimized the code using Cython, parallelization, and vectorization. We also discussed the use of specialized hardware like DPUs for accelerating the compression and decompression process.

Additionally, we explored the use of eigenvectors for selecting a compression matrix that preserves the most important information in the gradients. This approach can further improve the compression ratio and minimize the loss of accuracy in the model. Overall, gradient compression is a powerful tool for efficient distributed deep learning, and its implementation can be further optimized using a variety of techniques and hardware. All of the above have led us to the conclusion that eigenvectors can help us solve complex writing issues by identifying outliers, while more data can be used in better ways by optimal pooling techniques. FUrther led by an optimal code that implements the dynamic MPI algorithms, we can truly make a difference in not just the text identification algorithm, but also towards other applications of computer science that leverage deep learning. We hope with all sincerity that this little contribution to science is used for the greater good of the universe and all inclusive.

References

Alistarh, D., Grubic, D., Li, J., Tomioka, R., & Vojnovic, M. (2017). QSGD: Communication-efficient SGD via gradient quantization and encoding. arXiv preprint arXiv:1610.02132.

Dettmers, T. (2019). Sparse coding for neural networks. In Advances in neural information processing systems (pp. 2409-2420).

Lin, T., Talwalkar, A., & Duchi, J. (2018). Don't shy away from simplicity: A simple yet effective approach to distributed deep learning. In Proceedings of the 35th International Conference on Machine Learning (ICML) (pp. 3443-3452).

Wen, W., Xu, C., Wu, C., Wang, Y., Chen, Y., & Li, H. (2017). Terngrad: Ternary gradients to reduce communication in distributed deep learning. In Advances in neural information processing systems (pp. 1504-1513).

Zhang, Y., Sun, Y., Gao, J., & Rehg, J. M. (2018). An overview of deep learning in medical imaging: focusing on MRI. In Medical Imaging with Deep Learning (pp. 1-18).