TECHNIQUES TO ACCELERATE THE NEURAL NETWORKS*

SHENG HU , HONG JIN , WILLIAM JONES , PATRICK RENAUD , AND BIYAO WANG

Abstract. With the help of GPUs, this project aims to accelerate the training process of neural networks by implementing a light-weighted package based on [15] and to investigate the improvement of its performance with the lottery ticket hypothesis and other commonly-used techniques. The tiled matrix multiplication and GPU-based activation functions are faster than the codes that do the computation on the CPU for large enough matrices. At the same time, the lottery ticket hypothesis succeeded in preserving the accuracy of our implementation while maintaining a reasonable speed.

1. Introduction. Neural networks have become a powerful tool for solving a wide range of complex problems in various fields, such as character recognition [3, 18], stock prediction [4, 22], and medical diagnosis from CT scans [24, 23]. While neural networks produce highly accurate results, the training process can be time-consuming, especially with large datasets. For example, VGG and ResNet are computationally expensive and slow to train [11]. To address this issue, researchers have devised several techniques for accelerating the training process and optimizing the neural network's architecture.

In this project, we investigate various strategies for accelerating a neural network's training process while maintaining the model's accuracy. Specifically, we investigate the benefits and trade-offs of hardware acceleration and neural network complexity reduction and compare their effectiveness in improving training efficiency. Hardware acceleration can involve using specialised hardware, such as graphics processing units (GPUs), to speed up the computation of the neural network. On the other hand, neural network complexity reduction involves simplifying the neural network's architecture by reducing the number of layers and parameters. By evaluating these methods, our goal is to find the most promising approaches for accelerating neural network training and provide insights for future research in this field.

For more information, please visit our GitHub page.

2. Fundamentals of the Neural Networks. A neural network is an architecture for building machine learning models that consists of multiple layers of interconnected nodes, called neurons, that can learn to recognise complex patterns and

*Supervisor: James Chok and Benedict Leimkuhler

relationships in data. In this section, we explore some fundamental concepts of neural networks, including matrix multiplication and activation functions. Additionally, there are some advanced techniques like batch normalization and weight decay. By understanding these concepts and techniques, we can develop more efficient and effective neural networks for various applications.

2.1. Matrix Multiplication. Matrix multiplication plays an integral part in Neural Networks as it is required for some of their key components, such as back-propagation and forward pass. We will look at forward pass as an example of matrix multiplication within a Neural Network below.

Forward pass: A method where starting with an input $\{\mathbf{x_i}\}$ we compute and store the output of each layer $\mathbf{z}^k(x_i)$ in the NN as well as the derivative of the output of each layer w.r.t its inputs $\frac{\partial \mathbf{z}^{k+1}}{\partial \mathbf{z^k}}$. As an example here is what the output from a hidden layer would be using a forward pass:

$$z_j(\mathbf{x}_i) = \sigma(\alpha_{j,0} + \Sigma_{n=1}^{D} \alpha_{j,n} x_{i,n}) \tag{2.1}$$

where $\mathbf{z}_j(\mathbf{x}_i)$ represents the out from the hidden layer at neuron j, σ represents an activation function, in this case, a sigmoid function, $\alpha_{j,0}$ is the bias from the neuron in the hidden layer, $\alpha_{j,n}$ is a weight associated with the neuron and D is the number of features within our input x_i. From the equation, we can see that the sum $(\Sigma_{n=1}^{D} \alpha_{j,n} x_{i,n})$ is equivalent to a matrix multiplication.

Matrix multiplication is something that can easily be parallelised and this can help us speed up our Neural Network. This will be further investigated in Section 3.3.

2.2. Activation functions. The activation function is one of the essential parts of the neural network. Let $z = Wx + b$ be the weighted outputs of a layer in the neural network, computed following the procedures in Subsection 2.1. The activation function transforms the weighted outputs $z \in \mathbb{R}^n$ to be non-linear so that we can fit the model to data following a more complicated distribution or with a fixed range.

Two commonly used activation functions in neural networks are softmax and ReLU. The softmax function is defined as:

$$\sigma(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}. \tag{2.2}$$

On the other hand, ReLU is defined as:

$$\phi(z_i) = \max(0, z_i). \tag{2.3}$$

ReLU is generally used for efficiently introducing the non-linearity in the hidden layer by taking advantage of its more straightforward equation. Furthermore, it has been proven to perform better than other activation functions, such as sigmoid, following from [9]. Besides, the simple equation of ReLU can avoid issues caused by data with an enormous numerical value and let the computation of the weighted output after applying an activation function becomes simpler and more efficient. Therefore we chose ReLU as the activation function being implemented and examined in Section 3.4.

2.3. Tricks for the Neural Network. Apart from having a faster running speed, we must ensure that the updated neural network has acceptable accuracy compared to the initial model. Moreover, we need to ensure that the neural network does not overfit the training data, which means that the trained model performs well on unseen data. By employing the following techniques, we can ensure that the updated neural network not only be trained faster but also maintains an acceptable level of accuracy compared to the initial model.

Batch Norm: One widely used technique [13] for speeding up the training of neural networks while improving their performance is batch normalization. Batch normalization enables the use of higher learning rates and reduces the need for careful initialization, and it functions as a regulariser, potentially removing the need for Dropout. The application of batch normalization to a state-of-the-art image classification model attains equivalent accuracy with fewer training steps and outperforms the original model by a significant margin [14]. It normalises the input data of each layer in a neural network by scaling and shifting it, making it have zero mean and one variance.

Suppose that our set contains L data points, $\{x^k\}_{k=1,\dots,L}$, with $x^k \in \mathbb{R}^N$ for all k. During training, we divide the data into batches $\{x^k\}_{k\in K}$ for $K \subset \{1,\dots,L\}$. Let $K^t \subset \{1,\dots,L\}$ be the t-th batch used during training. To perform the scaling and shifting, we compute the element-wise mean $\mu_i = \frac{1}{|K^t|}\sum_{k\in K^t} x_i^k$, and the element-wise variance $\sigma_i^2 = \frac{1}{|K^t|}\sum_{k\in K^t}\left(x_i^k - \mu_i\right)^2$ over the batch. Next, along with a small positive constant $\epsilon \ll 1$ to ensure no division by zero, we normalise x^k:

$$x_{normalise} = \frac{x_k^t - \mu}{\sqrt{\sigma^2 + \epsilon}}. \tag{2.4}$$

Finally, we apply two parameters, a scaling factor $\Gamma \in \mathbb{R}^N$ and a shifting factor

$\beta \in \mathbb{R}^N$ to the normalised input:

$$x_{out} = \Gamma x_{normalise} + \beta. \tag{2.5}$$

Weight Decay: Weight decay (L2 regularisation) is a regularisation technique used to prevent overfitting during the iterative pruning and retraining process.

Suppose we train a neural network with a weight matrix $W \in \mathbb{R}^{n \times n}$, where n is the number of input units. During training, we use the cross entropy [10] to define the loss function as

$$C\left(\hat{y}, y\right) = -\sum_{i=1}^{n} y_i \log f(\hat{y}_i), \tag{2.6}$$

which measures the discrepancy between the predicted output $\hat{y}$ and the actual output y. Weight decay involves adding an additional term to the loss function being optimised during training. This is accomplished by adding the L2 norm of the weight matrix, $||W||_2^2$, multiplied by a hyperparameter, typically denoted by λ. Thus, the new loss function with weight decay, denoted by $C_{wd}(\hat{y}, y)$, is given by: [21]

$$C_{wd}(\hat{y}, y) = C(\hat{y}, y) + \lambda ||W||_2^2, \tag{2.7}$$

By penalizing large weights, weight decay encourages the network to learn more generalizable features that are more likely to be retained during pruning. [17]

Pruning: Instead of preventing overfitting by penalizing the complexity, pruning removes connections between neurons to avoid overfitting. Unlike weight decay, the pruning method can omit some connections between neurons during the training, using techniques such as masks, which reduces the time required to fit the model. After the network has been trained, pruning can identify these unnecessary connections and set their weights to zero, thus removing them. The pruned network can then be retained using the remaining connections to improve its performance. Furthermore, weight decay can help to improve the performance of the pruned subnetworks and increases their ability to generalise to new data.

However, the result of applying a training mask could be full of uncertainty. If the accuracy is not ideal, we might consider other methods and retrain the model with the original weight, which takes quite a long time. Therefore a new technique that can prune during the process would be beneficial for improving the training time for large enough neural networks. Section 4 will further explain this topic.

The following section will explore methods for speeding neural networks by utilizing GPUs and implementing pruning techniques.

3. Neural Network on GPUs. In this section, we reveal more details about the construction of neural networks based on GPUs and explain the rationale for the acceleration of the training process using GPUs, in addition to the experiments on the time taken for the activation functions in practice.

3.1. GPUs vs CPUs. A Graphics Processing Unit (GPU) is a specialised processor in a computer that is usually used to render images and visuals. However, they also have other applications in computing, for example, to speed up some processes [20, 12, 1]. One of the main reasons for this growing interest is due to how GPUs differ from Computational Processing Units (CPUs) in terms of their structure. Most CPUs comprise of a small number of highly sophisticated cores, usually 1-8, that can be used on certain tasks. In comparison, GPUs usually have between 400-4000 cores. These cores, however, are much less sophisticated when compared to those in GPUs. However, what they lack in speed and accuracy, they can make up in sheer numbers.

3.2. GPU memory hierarchy. One of the most important features when it comes to GPUs is their memory structure. GPUs are highly potent due to their ability to run thousands of different tasks at the same time. This is known as parallelism. This can be done due to the sheer number of cores within a GPU, as well as its distribution of memory.

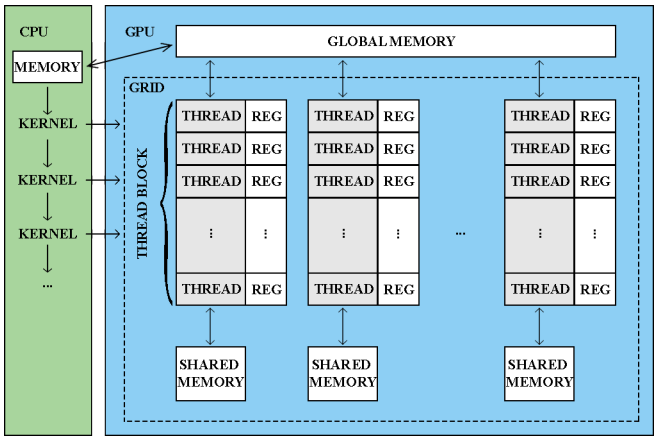


Fig. 3.1: GPU memory structure [19]

Figure 3.1 provides us with a brief overview of the structure of a GPU. There are three different types of memory in a GPU: registers, shared memory, and global memory.

Each thread within a thread block has its own private memory known as a register. Only the thread associated with the given register can access the data stored within. When a GPU is given a task the threads can reach into global memory in order to receive their assignments, which they can then in turn store within their own register.

The next step up from this is shared memory. Each thread block has its own shared memory. Unlike the registers, all the threads within the thread block can access the shared memory. This is very beneficial because it allows for threads within a given thread block to communicate amongst each other without having to access the global memory. Moreove, it means that each thread block can operate individually without being bottlenecked by the global memory, which allows for multiple tasks to be accomplished at the same time without having to communicate with one another.

Finally there is global memory. All threads and thread blocks have access to the global memory. Global memory can be one of the biggest bottlenecks when it comes to computation as it can only manage a certain amount of requests at once. The biggest benefit to the structure found within a GPU is the fact that the threads are able to operate and communicate without needing to access the global memory at all times.

Now that we have understood how the structure of a GPU may be beneficial, we next want to understand under what circumstances we would want to use a GPU over a CPU. Thanks to the property of parallelism, GPUs work well when it comes to element-wise operations such as summing two arrays together. This is due to the fact that each element can be assigned to a separate thread blocks, and the threads can perform the different operations independently. In comparison, a CPU only has a small number of cores and usually have to wait for one operation to finish before starting the next to start.

On the other hand, if you wanted to sum all the elements of an array together, it might be more beneficial to use a CPU. This type of operation requires previous information, so threads will have to wait for one another to finish their own operations. Since CPUs have more sophisticated cores they can perform this type of operation much faster.

Another thing to consider is what precision we are working with. GPUs in general are not as sophisticated as CPUs. This means that as the precision increases, GPUs efficiency will typically drop off. Figure 3.2 which shows how newly released GPUs and CPUs perform at both single and double precision.

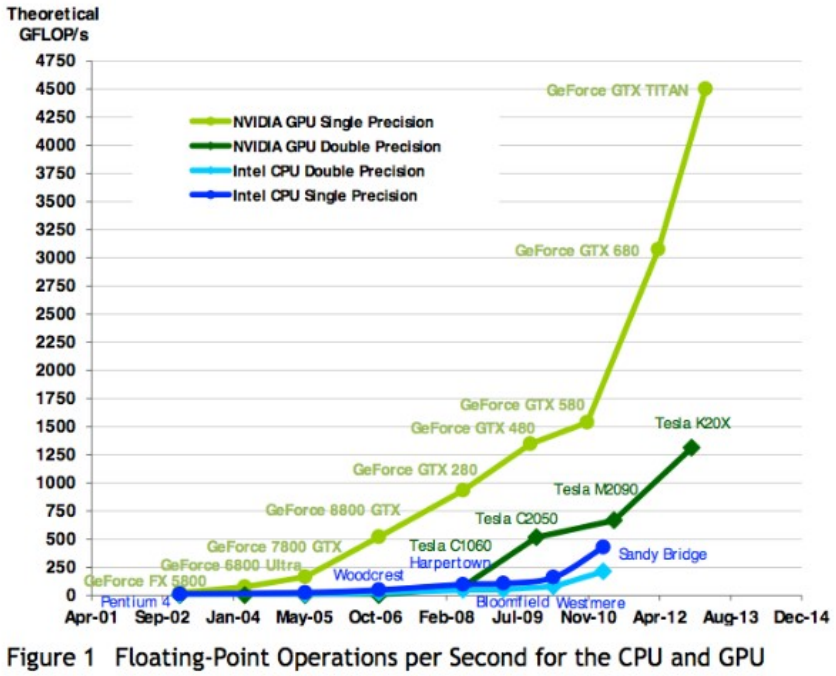


Fig. 3.2: GPU vs CPU performance [20]

As we can see from Figure 3.2, GPUs perform drastically better than CPUs do at single precision. However, we notice that as soon as it comes to higher precision the GPUs performance drops off significantly. In comparison, while the CPU may not perform as well as the GPU at single precision their performance actually increases as the precision increases.

3.3. Matrix Multiplication. Matrix multiplication is a fundamental operation in neural networks (NNs), forming the backbone of many NN architectures. The primary two operations involved are forward passing, where network weights are applied to input data, and backpropagation where weights are updated through gradient descent. This motivates us to implement matrix multiplication efficiently on a GPU.

For this section, consider the matrices $A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}, C \in \mathbb{R}^{m\times n}$, and the

matrix multiplication $AB = C$, where

$$c_{i,j} = \sum_{p=1}^{k} a_{i,p} \cdot b_{p,j}, \quad \text{for } 1 \leq i \leq m, 1 \leq j \leq n. \tag{3.1}$$

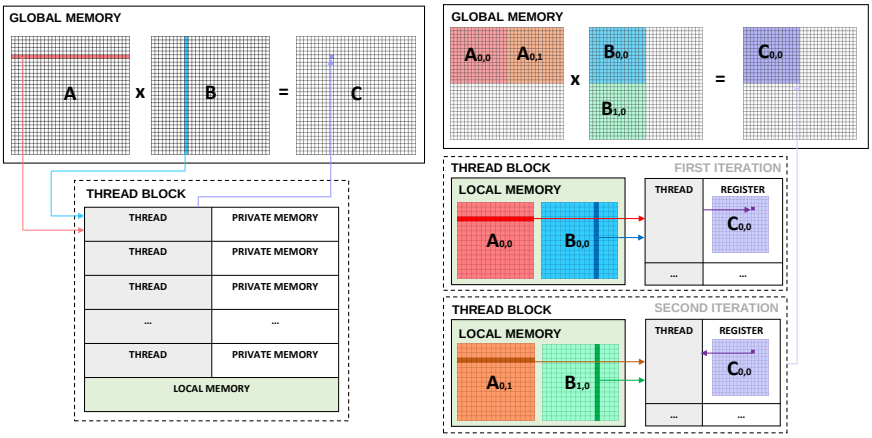
A basic CPU implementation of this sequentially loops over the m rows of A, and n columns of B, and calculates the dot product, all on a single thread. This is worst case $m \times k \times n$ multiplications and $(k-1) \times m \times n$ additions, resulting in $2\,m\,k\,n - m\,n$ floating-point operations (flops) all done one after the other. Improvements can be made by utilising the parallelism of a CPU and optimising cache memory but these are limited and considered on a GPU also. We aim to improve on this using a GPU and the methods described below.

3.3.1. Naive MatMUl. A naive GPU implementation of (3.1) has each thread calculate one element of matrix C, with all threads calculating all elements of C in parallel (Figure 3.3a). Each thread loads, from global memory, an i-th row of A and j-th column of B, and calculates the term $a_{i,p} \cdot b_{p,j}$. The thread repeats this for all $p = 1, ..., k$ multiplications, summing the total and returning the corresponding element $c_{i,j}$ back to global memory. In short, each thread calculates the sum in (3.1) for a row-column pair, which is worst case k multiplications and $k - 1$ additions, a total of $2\,k-1 \in o(k^2)$ flops done sequentially. The method does $2\,m\,k\,n - m\,n \in o(k^4)$ total flops but since the threads are working in parallel, the total time taken will be the time taken for the worst case thread, at $2k - 1$ flops. In contrast, the sequential nature of the CPU requires us to wait for $2\,m\,k\,n - m\,n$ flops to be performed

The primary issue faced by this method is that each thread has to access global memory for every multiplication, which is k loads from global memory for each thread and $m\,k\,n \in o(k^4)$ global memory loads in total. So even though we only have to wait for each thread to perform $2k-1$ flops, we will need to allocate extra time to account for this memory access. This will likely bottleneck performance due to the bandwidth of global memory access.

3.3.2. Tiling. Here we describe a 'tiling' method [25, 16] for MatMul, which better utilises the memory hierarchy.

Without loss of generality, let $A \in \mathbb{R}^{16m \times 16k}, B \in \mathbb{R}^{16k \times 16n}$ and $C \in \mathbb{R}^{16n \times 16m}$ If A, B and C do have this structure we can pad the matrices with zeros to fit. Then define $A^{i,p}, B^{p,j}, C^{i,j} \in \mathbb{R}^{16 \times 16}$ for $i = 1, \ldots n, p = 1, \ldots k$ and $j = 1, \ldots, m$ where



(a) Naive method

(b) Tiling method, a thread block can compute $C_{0,0}$ in two iterations: $C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0}$.

Fig. 3.3: GPU matrix multiplication methods

$$C^{i,j} = \sum_{p=1}^{k} A^{i,p} B^{p,j}, \quad \text{for} 1 \leq i \leq m, 1 \leq j \leq n \tag{3.2}$$

One thread block (workgroup) computes one tile, $C^{i,j}$, of matrix C, by computing 3.2 over multiple iterations (for each p in 3.2). One thread in the thread block computes one element, $C^{i,j}_{l,m}$, of the tile similar to the Naive method. Over $p = 1, \ldots k$, each thread block loads one tile of A, $A^{i,p}$, and one tile of B, $B^{p,j}$, from global memory into local memory, calculates that part of the elements of $C^{i,j}$, and stores the temporal result in the register. Once all iterations are complete, the thread block returns $C^{i,j}$ to global memory.

Figure 3.3b demonstrates how this can be done simply on a 32×32 matrix. The computational cost for each thread remains the same at $o(k^2)$ flops but by using a tile size of 16 we massively reduce the amount of global memory access.

Using a tile width of 16 is a common optimisation technique as it leverages the parallel processing capabilities of GPUs [16]. For a tile width of 16 and 4 bytes per word (element), each thread block uses $2 \times 256 \times 4$ B $= 2$ KB of shared memory, so for 16KB shared memory there can be a possible 8 thread blocks executing. However it is important to note optimal block size depends on matrix size and GPU architecture,

so some fine-tuning may be required for optimality. For simplicity, we will pad our matrices to be divisible by 16×16 blocks.

3.3.3. Memory coalescing and avoiding memory bank conflicts. Before implementing this in PyOpenCL, we must ensure that we are accessing memory efficiently as bad memory access patterns will harm performance.

Memory coalescing is a technique used to optimize the way data is loaded from global or shared memory, which combines multiple memory requests from adjacent threads into a single memory transaction. This allows for data to be retrieved in a contiguous, aligned manner, reducing the number of memory transactions and improving memory access efficiency.

In context, this means to load a row 0 of A we want thread t_i to load element $a_{0,i}$ and thread t_j to load element $a_{0,j}$. PyOpenCL is based on C, which is row-major so A and B are stored as a list of rows. This means elements of A can be accessed efficiently, as shown in Figure 3.4. On the other hand, to load a column 0 of B we want thread t_i to load element $b_{i,0}$ and thread t_j to load element $b_{j,0}$, but these elements are not stored next to each other, which is not efficient. To fix this, we can store the transpose of B, which gives B the same memory access patterns as A.

It is also important to consider memory bank conflicts. Memory bank conflicts occur when multiple memory requests are made to the same memory bank (some memory storage in the hierarchy) at the same time. In a conflict, the processor has to prioritize which memory request to execute first, leading to delays in processing other memory requests. This can result in reduced performance and slower execution of programs. The key to avoiding memory bank conflicts is to distribute memory accesses evenly across all available memory banks.

3.3.4. Results. We implement the methods described above and compare the runtimes to NumPy and Tensorflow on square matrices of increasing dimension and matrix-vector multiplication where the matrix is square. The GPU used in these tests is a Tesla T4, and the CPU is an Intel Xeon 2.20GHz.

Figure 3.5a shows that the CPU outperforms the naive method and is massively outperformed by TensorFlow. We suspect that this is due to the large volume of global memory accesses that the naive method has to do.

Figure 3.5a demonstrates that for large dimension matrices, the tiling method on a GPU outperforms the CPU and naive method. TensorFlow is still quicker than

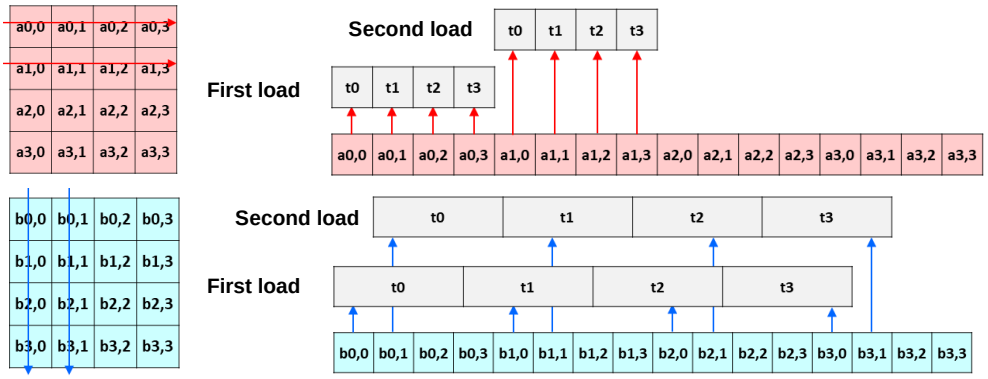
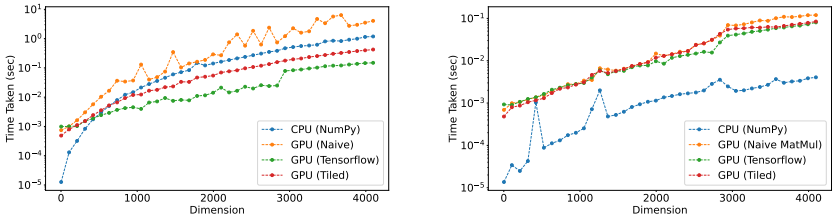


Fig. 3.4: Memory access for A and B



(a) Matrix multiplication runtimes on square matrices of increasing dimension

(b) Matrix-vector multiplication runtimes of increasing dimension

Fig. 3.5: MatMul runtimes on a CPU and GPU of tiling and naive GPU methods, compared to NumPy and TensorFlow MatMul

the tiling method, likely due to having more sophisticated algorithms with respect to memory access, but it is built into a large Python package whereas our method is cheap and small which can be advantageous.

For smaller dimensions, the CPU is quicker than Tensor flow and our methods but we are less interested in this as neural nets often have lots of data points, layers and nodes, and hence deal with large matrices. Therefore, we will aim to use the 'tiled' method described here to speed up neural network algorithms that rely on matrix multiplication.

Interestingly, Figure 3.5b shows that the CPU is still by far the best choice for matrix-vector multiplication as it also outperforms TensorFlow by quite a margin. The tiling method is still worse than TensorFlow but better than naive, which is to

be expected.

3.4. Activation functions. As described in Subsection 2.2, the activation functions characterise the non-linearity of the predictive values by applying a nonlinear mathematical equation to each neuron. Since neurons are stored as entries of a matrix, activation functions can be applied to each neuron in parallel, indicating that the training process can again be accelerated by rewriting the activation functions running on GPUs. In this section, we conduct experiments using Tesla T4 as GPU and Intel Xeon 2.20GHz as the CPU, to keep consistency with Section 3.3.

During the project, we implemented two commonly-used activation functions, namely ReLU and softmax, as defined in Section 2.2. Since the ReLU function is applied element-wisely, it can be parallelised easily, and the computational cost can be only one floating-point operator for each entry of the weighted sum when we parallelised the computation fully. Such a feature allows us to accelerate the computation of the output layer when the weighted sum passes through ReLU. However, the softmax function of a given weighted sum $z \in \mathbb{R}^n$, as clarified in Equation 2.2, could have a poor performance when z has some extremely large entries z_k. The massive z_k would result in the value computed exceeding the limit of the given data type, i.e., overflow, when computing $\exp(z_k)$. In terms of keeping the numerical stability of the softmax, we consider shifting z by its maximum in our code, hence Equation 2.2 becomes:

$$\sigma(z_i) = \frac{\exp(z_i - z_{\max})}{\sum_{j=1}^{n} \exp(z_j - z_{\max})}. \tag{3.3}$$

After shifting, the largest $\exp(z_i)$ is set to zero, which reduces the probability of overflow occurring. This implementation could however require a larger amount of computational cost, due to the search for the maximum in each row.

To examine the improvement of the activation functions in the aspect of the training time, we compare the performance of our implementation of ReLU and one of the most popular packages for neural networks. We conduct the experiments by measuring the average running time for every execution of the activation function ReLU using both TensorFlow and our implementation, running on CPUs and GPUs, respectively. The average values are taken from 100 iterations in 10 separated runs to make the result less biased. The test input is set to a vector with different numbers of elements, from 500000 to 10^7, in increments of 500000. Since the ReLU function is

applied element-wisely, the time taken for a matrix output and a vector output should be the same.

In Figure 3.6, the time taken is log-scaling to make subtle changes easier to spot. Compared to CPU-base functions in the other packages, our implementation of the activation functions ReLU, as we can see from Figure 3.6, starts to achieve a better performance when the number of elements exceeds roughly 8×10^{-6} for the vector input z. In contrast, the GPU-based activation functions on PyTorch are still way faster than our implementation.

However, considering the size of our package, our model still has the advantage of being able to run on devices with limited storage space, such as mobile phones, as long as they have a GPU. The difference in dimensions of which our implementation begins to be faster also indicates that our model might be more efficient on the square matrix. This is realistic since it could be hard to assign the processing units efficiently for the non-square matrix due to the difference in the number of rows and columns.

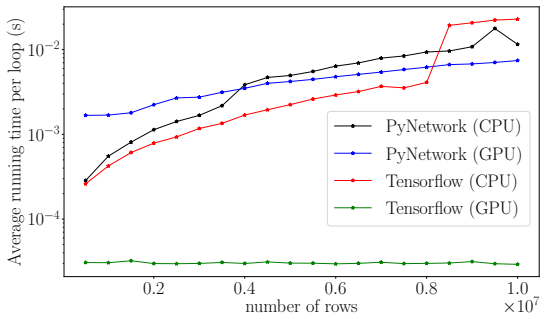


Fig. 3.6: Running time comparison for ReLU in different packages

Implementation with Naive Matrix Multiplication: Due to the time limit and implementation difficulty, we have only implemented the model with naive matrix multiplication. However, given the running time comparison for the tiled matrix multiplication and that for the activation functions shown in Figure 3.5a and 3.6, we can still be confident with the comparativeness of our model concerning the training time when using the faster matrix multiplication, which can be accelerated more if we have a neural network with a simpler architecture.

4. Pruning and Lottery Ticket Hypothesis. A deep neural network always contains many components and a large number of parameters. Training and testing

an entire neural network are costly. Therefore, pruning techniques have become very important. [2] Pruning is a technique that can be used to remove unnecessary parameters from an existing neural network. It aims to reduce the computational cost and increase the network's efficiency without compromising accuracy.

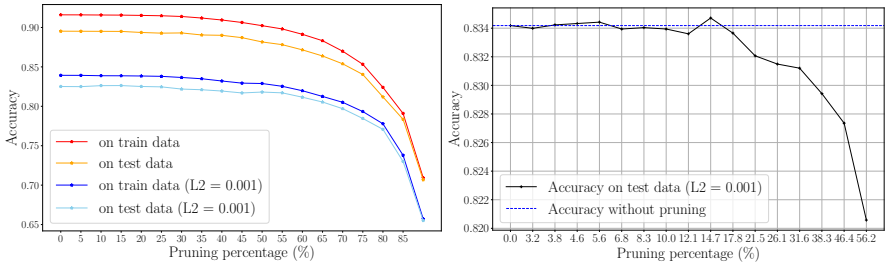
The principle that pruning can reduce the computational cost is to make the matrices sparse. For instance, if we want to remove some unnecessary weight parameters, we can create a mask on the weight matrix to force some elements to zero. These zero elements do not acquire extra memory. In addition, when the computer implements the matrix multiplications, these zero elements can be ignored to reduce the operation costs. In Lee's article [5], he and his research group used the Viterbi algorithm to propose a new class of sparse matrix representation to achieve a high-performance index decoding process with low energy.

In this section, we apply our code to explore pruning methods and the famed lottery ticket hypothesis on GPUs. We use a data set of handwritten character digits called the Extended MNIST [8] derived from the NIST Special Database 19. Compared with the MNIST data set, the EMNIST data set has a larger size and can constitute a more complicated and exciting task.

4.1. Traditional pruning technique. Traditionally, we can prune the neural network after the network is fully trained. Using a simple model containing three dense layers, we have explored the traditional pruning method in this subsection. Figure 4.1a indicates the accuracy of data sets against different pruning percentages. It shows a general declining trend with different changes when the pruning percentage falls in different ranges. When the pruning percentage $p\%$ is less than 40%, the accuracy of the data sets has no significant change. When the pruning percentage is between 40% to 75%, the accuracy starts to decline gradually. The decline accelerates significantly when $p\%$ is greater than 80%. However, the accuracy stays around 80% even if we remove 80% of parameters from the neural network.

By adding the weight decay (L2 norm), we notice that although the accuracy is not better than the accuracy without adding the weight decay, the difference between the training data accuracy and testing data accuracy becomes smaller. This indicates that adding an appropriate weight decay can prevent overfitting the neural network model. Despite the difference, the overall change of the accuracy with weight decay alongside the increase of pruning percentage demonstrates the same trend.

As a matter of fact, the accuracy might even increase slightly sometimes when the pruning percentage is small (less than 20%). Moreover, it can be higher than the initial accuracy in this case. This increase could result from the overfit improvement by removing a small number of parameters from the network. We indicate this fact in Figure 4.1b, in which we use the log scale for the pruning percentage.



(a) Accuracy with/without weight decay (b) Effects of pruning a small portion

Fig. 4.1: Accuracy after applying different pruning percentages (traditional pruning)

Although our experiment above shows that with the traditional pruning method, almost half of the parameters could be pruned while not compromising the accuracy of the network, there is still room for improving the method. For instance, complete training of the network is required by this method which usually takes much time if the size of the network is large. Additionally, the required complete training of the network also makes subsequent fine-tuning difficult.

4.2. Lottery Ticket Hypothesis. In 2019, Frankle and Carbin [7] raised the lottery ticket hypothesis, which is a randomly initialised, dense neural network containing a subnetwork (the winning ticket). When trained in isolation, this subnetwork can match the original network's test accuracy after training for at most the same number of iterations.

The lottery ticket hypothesis [7] suggests that for a dense feed-forward neural network $f(x;\theta)$ with initial parameters $\theta = \theta_0 \sim \mathcal{D}_\theta$, there exists a mask $m \in \{0,1\}^{|\theta|}$ such that training $f(x; m \odot \theta)$ using stochastic gradient descent (SGD) on the same training set results in reaching the minimum validation loss l at iteration j with commensurate training time, and a test accuracy a with commensurate accuracy, where $\|m\|_0 \ll |\theta|$ fewer parameters.

The hypothesis is verified by the authors, who also provide clear steps for finding

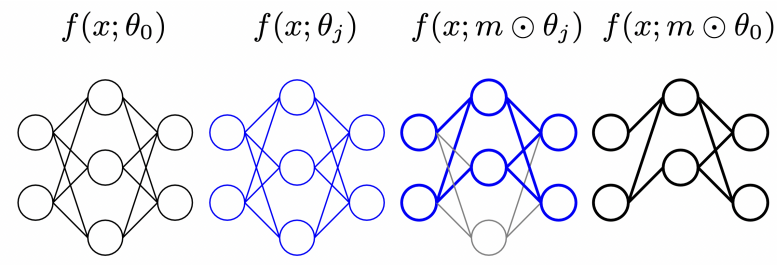


Fig. 4.2: The steps for finding the winning ticket

the winning tickets (Figure 4.2 is the visualization):

- Initializing a neural network randomly: $f(x;\theta_0)$, where $\theta_0 \sim \mathcal{D}_\theta$.
- Training the network with j iterations: $f(x;\theta_j)$.
- Pruning $p\%$ of the parameters and creating a mask $m \in \{0,1\}^{|\theta|}$: $f(x;m\odot\theta_j)$.
- Resetting the survival parameters to their initial parameters. The winning ticket is: $f(x;m\odot\theta_0)$.

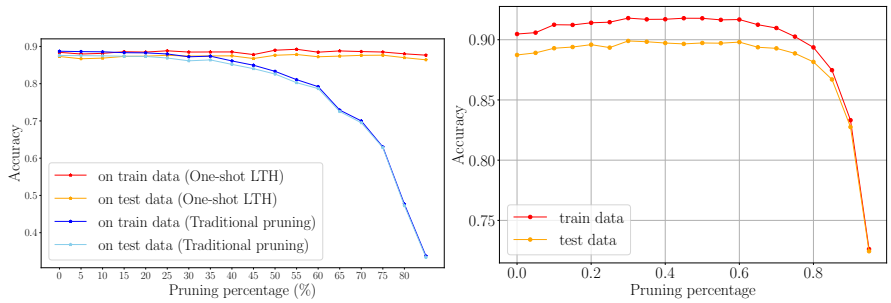
These four steps indicate a one-shot pruning approach, which the network only trained once. Frankle and Carbin [7] focus on iterative pruning, in which the network is repeatedly trained, pruned, and reset over n rounds with pruning rate $p^{\frac{1}{n}}\%$.

This advancement of neural network training makes an essential contribution to the field and is therefore widely celebrated and receives significant scholarly attention. Building on these findings, we could prune neural networks before the completion of their training. What's more exciting is that their iterative method finds winning tickets that allow 80-90% of pruning percentage, which will drastically reduce computational cost and improve the efficiency of using the network.

4.3. Numerical results. To showcase how powerful the pruning method developed by the authors, we conduct a one-shot pruning experiment and compare the performance with the traditional pruning method in this subsection. Figure 4.3a compares the accuracy of the one-shot pruning method with that of the traditional one. It indicates that the one-shot pruning method results in a much higher accuracy of the neural network. Although the gap between the results of these two methods is small when the pruning percentage is lower than 40%, it widens rapidly as more parameters are pruned. In general, the one-shot method is much more effective than the traditional method in maintaining the neural network's accuracy level when pruning

it.

Figure 4.3b indicates the change of the network's accuracy as the pruning percentage increases with the one-shot method. It shows that when a low percentage of parameters is pruned, the accuracy increases as the pruning percentage increases. When 30% of parameters in the neural network are pruned, the accuracy of the network reaches its peak, which is significantly higher than the initial accuracy. Even when the pruning percentage goes around 70%, the accuracies of both the train data and test data accuracies are higher than or equivalent to the initial accuracy without pruning any parameter. Therefore, we could identify a winning ticket that can remove 75% of parameters without reducing the initial accuracy. When the pruning percentage increases to more than 75%, the accuracies of both data decrease below the initial accuracy.



(a) Comparison with traditional pruning (b) Performance of the one-shot pruning

Fig. 4.3: Accuracy after applying different pruning percentages (one-shot pruning)

With multiple experiments of the one-shot method, we found winning tickets 75%-85% smaller than the original network. When we adopt an iterative method, better-winning tickets with smaller sizes can be found, as shown in [7], which implies that an even higher percentage of parameters could be pruned with this method while maintaining an accuracy level that matches the accuracy of the original network.

5. Conclusion & Discussion. GPUs (Graphics Processing Units) are specialised hardware designed for highly parallel processing and can be used for accelerating certain computations, such as training neural networks. Due to the property of parallelism and multiple cores, GPUs can significantly speed up the training process in neural networks compared to CPUs.

Matrix multiplication is a fundamental operation we use on a GPU that can benefit from parallel processing. GPUs are designed to handle massive amounts of data and simultaneously perform parallel computations on multiple data elements, making them highly efficient for tasks involving large-scale matrix operations. One approach to optimising matrix multiplication on GPUs is tiling, which helps reduce memory access overhead and improve performance. Model techniques, such as pruning, weight decay, mask, and lottery ticket hypothesis, can help reduce the size of the model and reduce the computational requirements of training neural networks, making them more suitable for GPU acceleration.

In some cases, CPUs may outperform GPUs when implementing the PyOpenCL code. Since the PyOpenCL code needs to be optimised properly, it may not fully leverage the parallel processing capabilities of the GPU. On the other hand, some techniques, like memory allocation, can add overhead to the implementation process. If the overhead outweighs the potential performance gains, then using a CPU is much faster. Additionally, the operation of the sparse matrix in Matrix Multiplication can perform less well in practice. There is much research on this area. [6] Consequently, all zero elements still take computing power even though they are pruned. We aim to conduct more experiments for the optimisation in the future.

REFERENCES

[1] F. Bernal, J. Morón-Vidal, and J. A. Acebrón, *A hybrid probabilistic domain decomposition algorithm suited for very large-scale elliptic pdes*, 2023, https://arxiv.org/abs/2301.05780.

[2] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, *What is the state of neural network pruning?*, arXiv:2003.03033 [cs, stat], (2020), https://arxiv.org/abs/2003.03033.

[3] J. Blue, G. Candela, P. Grother, R. Chellappa, and C. Wilson, *Evaluation of pattern classifiers for fingerprint and ocr applications*, Pattern Recognition, 27 (1994), pp. 485–501, https://doi.org/https://doi.org/10.1016/0031-3203(94)90031-0, https://www.sciencedirect.com/science/article/pii/0031320394900310.

[4] P. Chhajer, M. Shah, and A. Kshirsagar, *The applications of artificial neural networks, support vector machines, and long–short term memory for stock market prediction*, Decision Analytics Journal, 2 (2022), pp. 100015–.

[5] D. Lee, D. Ahn, T. Kim, P. I. Chuang, and J. J. Kim, *Viterbi-based pruning for sparse matrix with fixed and high index compression ratio*, International Conference on Learning Representations, (2018), https://openreview.net/forum?id=S1D8MPxA-.

[6] S. Dalton, L. Olson, and N. Bell, *Optimizing sparse matrix—matrix multiplication for the gpu*, ACM Transactions on Mathematical Software, 41 (2015), pp. 1–20, https://doi.org/10.1145/2699470.

[7] J. Frankle and M. Carbin, *The lottery ticket hypothesis: Finding sparse, trainable neural networks*, 2019, https://arxiv.org/abs/1803.03635.

[8] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, *Emnist: Extending mnist to handwritten letters*, International joint conference on neural networks (IJCNN), IEEE, (2017), pp. 2921–2926, https://doi.org/10.48550/arXiv.1702.05373.

[9] X. Glorot, A. Bordes, and Y. Bengio, *Deep sparse rectifier neural networks*, Journal of machine learning research, 15 (2011), pp. 315–323.

[10] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, Springer, second ed., 2009.

[11] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in 2016 IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR), vol. 2016- of IEEE Conference on Computer Vision and Pattern Recognition, NEW YORK, 2016, IEEE, IEEE, pp. 770–778.

[12] intel, *What is a gpu?*, https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html.

[13] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, (2015).

[14] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015, https://arxiv.org/abs/1502.03167.

[15] James Chok, *Pynetwork package*, 2023, https://github.com/infamoussoap/PyNetwork (accessed 2023/03/06).

[16] D. Kirk, *Matirx multiply (memory and data locality)*, https://www.cs.ucr.edu/~amazl001/

teaching/cs147/S21/slides/09-Matrix_Multiplication.pdf.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, Communications of the ACM, 60 (2012), pp. 84–90, https://doi.org/10.1145/3065386.

[18] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, Nature, 521 (2015), pp. 436–444, https://doi.org/10.1038/nature14539.

[19] A. Li, R. Maunder, B. Al-Hashimi, and L. Hanzo, *Implementation of a fully-parallel turbo decoder on a general-purpose graphics processing unit*, IEEE Access, 4 (2016), pp. 1–1, https://doi.org/10.1109/ACCESS.2016.2586309.

[20] C. Navarro, N. Hitschfeld, and L. Mateu, *A survey on parallel computing and its applications in data-parallel problems using gpu architectures*, Communications in Computational Physics, 15 (2013), pp. 285–329, https://doi.org/10.4208/cicp.110113.010813a.

[21] G. B. Orr and K.-R. Müller, *Neural Networks: Tricks of the Trade*, Springer, 07 2003.

[22] A. M. Rather, A. Agarwal, and V. Sastry, *Recurrent neural network and a hybrid model for prediction of stock returns*, Expert Systems with Applications, 42 (2015), pp. 3234–3241, https://doi.org/10.1016/j.eswa.2014.12.003.

[23] H.-C. Shin, H. R. Roth, M. Gao, L. Lu, Z. Xu, I. Nogues, J. Yao, D. Mollura, and R. M. Summers, *Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning*, IEEE Transactions on Medical Imaging, 35 (2016), pp. 1285–1298, https://doi.org/10.1109/tmi.2016.2528162.

[24] K. Steimle, M. Mogensen, D. Karbing, J. Bernardino de la Serna, and S. Andreassen, *A model of ventilation of the healthy human lung*, Computer Methods and Programs in Biomedicine, 101 (2011), pp. 144–155, https://doi.org/10.1016/j.cmpb.2010.06.017.

[25] L. Waeijen, *Matrix multiplication cuda*, https://ecatue.gitlab.io/gpu2018/pages/Cookbook/matrix_multiplication_cuda.html.