Parallel Adversary Generation for Training Robust Neural Networks

Neehal Tumma[1] and Rajat Mittal[1]

[1]Harvard College

Abstract

In recent years, convolutional neural networks (CNNs) have become increasingly popular as the solution for image classification tasks. Although earlier studies showed CNNs are very versatile and accurate, more recent research has found that they are vulnerable to specially designed attacks known as adversarial images. To make networks robust to these attacks, it is necessary to train the CNN with adversarial data. In this paper, we present the first known methodology to generate adversarial examples (using the projected gradient descent method) in parallel. We employ an MPI framework and explore potential applications of OpenMP in order to parallelize this adversary generation algorithm. We achieve a significant speedup over the sequential implementation with little to no loss in model performance.

1 Background and Significance

Computer vision and image recognition have historically been one of the hardest real-world problems for computers to solve. However, advances in machine learning have made image recognition a far more tractable problem via the advent of convolutional neural networks (CNNs) [3]. Modern applications of CNNs include detecting medical anomalies in medical images, recognizing road signs for autonomous vehicles, and facial recognition. Many of these applications require a very high level of robustness, as failures could have large scale negative impacts.

It turns out that images can be engineered to fool the CNN. These are called adversarial examples. Some of the initial research done by Szegedy et al. [7] found that they could force the network to misclassify an image by adding a very slight noise that was engineered to maximize the network's error. The image with the noise looks almost identical to the image without the noise, so the noise does not interfere with human recognition of the image. Hence, CNNs have lots of room for improvement before they can automate tasks in safety-critical applications. Research by Szegedy et al. [7] has also found that if we include adversarial examples in the training set and teach the network to classify those correctly, it makes the network resistant to these attacks .

One computationally inexpensive way to generate the adversaries is the fast gradient sign method [1]. The fast gradient sign method works by adding a vector with elements that are equal to the sign of the elements of the gradient of the loss function with respect to the input image. Further research was done to create even stronger adversaries by running an iterated version of this algorithm [4].

Although research has been done into making a neural network resistant to adversarial attacks, training with adversaries has its own set of issues. It turns out that generating the adversaries while doing model training is far more computationally expensive than the actual model training. This poses a serious barrier to the widespread use of robust neural networks. Over time, there has been research done into making adversarial training more efficient and effective. Fast gradient sign method can be accelerated using techniques including mixed-precision training [5] and cyclic learning rates [6]. But, to our knowledge, there hasn't been any published research done on accelerating the adversary generation using parallel computing.

2 Scientific Goals and Objectives

Adversarial Model Training

Training robust models requires exposing the model to adversaries during training. The current state-of-the-art algorithm for training robust models, researched by Madry et al. [4], applies projected gradient descent (PGD) to input images before the model is trained on them. PGD (eq. 1) iteratively modifies the input image k times to fool the model, and is a variant of the fast gradient sign method [1].

$$x_{adv}^{(i)} = x_{adv}^{(i-1)} - \epsilon \nabla_{x_{adv}^{(i-1)}} J(x_{adv}^{(i-1)}, y_{target}) \quad \text{for } i = 1, \ldots, k \quad \text{s.t. } x_{adv}^{(0)} = x \tag{1}$$

This computation is expensive because it requires performing backpropagation k times, where k is desired to be large since more PGD iterations corresponds to stronger adversaries, which translates to a more robust model. Since the adversary generation is dependent on the specific model weights, adversaries cannot be precomputed for a dataset. Instead, adversaries are computed immediately prior to being used to performing a weight update (fig. 1). Since adversary generation requires k backpropagation steps, and the weight update is also a backpropagation step, adversary generation takes approximately k times as long as the weight update.

As we stand today, training large scale machine learning models is already very expensive, and thus training *robust* machine learning models is nearly infeasible due to the k times slowdown. Thus, our goal is to make training robust neural networks more practical for researchers working in HPC environments. We propose and show significant performance improvements of a novel parallel approach to robust model training using MPI and OpenMP. By completing this study, we'll also be able to observe how well this task scales with more cores. This would necessitate the need for compute hours on a HPC architecture with many cores. As adversary generation is a very intensive task to add into ML model training, an effective parallel implementation could have major implications for a variety of real-world applications. We hope this research can eventually stem into a library for other researchers.

3 Algorithms and Code Parallelization

Parallelization with MPI

Our primary parallelization strategy is to perform adversary generation in parallel with weight updates via MPI processes (fig. 1). Neural networks are typically trained with minibatch gradient descent, where the entire dataset is divided into batches.

Consider a simple example where we are training a robust model with $k = 1$ adversary steps. Since a single iteration of PGD and a single weight update both take approximately the same amount of time, if we use 2 processes, then one process can be performing the weight update on one minibatch B_1 while the other process is generating adversaries on another minibatch B_2. After the adversary-generating process is finished, it can perform an asynchronous send of adversarial B_2 to the weight-updating process, and then begin generating adversaries for B_3. Meanwhile, the weight-updating process will have finished updating the model weights based on B_1, so it can post a receive to the other process and receive the adversarial B_2. This process can continue indefinitely, and we can effectively hide all the latencies associated with generating adversaries by doing them simultaneously with weight training.

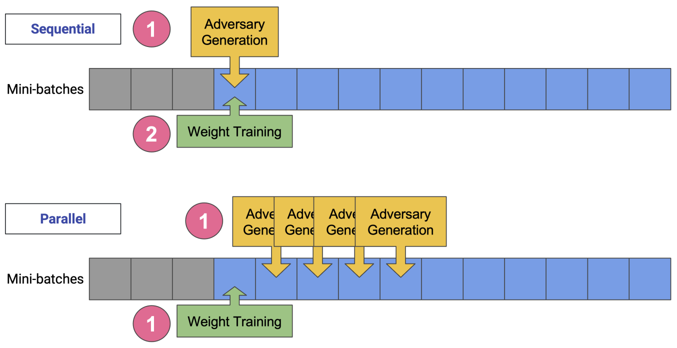


Figure 1: Weight Updates

Note that this code is no longer the exact same as the sequential computation. Adversaries sent over to the weight-updating process have now been generated on an older, stale copy of the weights that no longer exist. We define the *staleness* of an adversary as the number of batches that are used to update the weights after the oldest batch used in any step to generate the adversary (appendix fig. 3). That is, if a step of the PGD occured with model weights that have been updated 5 times since then, then the staleness of that adversary is 5. In general, we have reason to believe that for small staleness values, the parallel algorithm will not suffer in performance compared to the sequential algorithm. However, if many of the adversaries have a high staleness, then they likely do not effectively attack the current model weights. Thus, we limit the staleness of adversaries by only letting them start generating a batch of adversaries when the previous batch has been received by the weight-updating process. This is naturally implemented with a `MPI_Ssend` when sending over a batch of adversaries.

Generalization to n Processes

With n processes, we dedicate $n-1$ processes to adversary generation, and a single process (rank 0) to updating the weights. PyTorch provides a `torch::data::samplers::DistributedRandomSampler` class that is used for distributed-training jobs, so that the B_{total} minibatches are randomly shuffled after each epoch and each process loads an equal proportion of the total number of minibatches. It is important to note that since $n-1$ processes are splitting the entire dataset, each adversary generating process generates adversaries for roughly $\frac{B_{total}}{n-1}$ batches. If the number of adversary steps is $k = n-1$, each adversary-generating process is performing $\frac{B_{total}}{n-1} \times (n-1) = B_{total}$ backpropagation steps, which is the same number of backpropagation steps as the weight-updating process. Thus, to minimize load imbalance, it is optimal to choose $k = n-1$ for our algorithm.

Communication and Compute-Transfer Overlap

There are two types of communication occurring in this framework: adversary batches are being sent to the weight-updating process and fresh model weights are being sent to the adversary generating process. With n processes, there are $n-1$ adversary generating processes all competing to send adversaries (and the corresponding targets, see Equation 1) to the weight-updating process. Thus, we need to employ some notion of fairness (to avoid issues with load imbalance). An adversary-generating process cannot be allowed to send the second of two batches of adversaries if after sending the first batch, another process has posted a send of adversaries. Normally, this would be handled with a `MPI_Waitsome`, however, since the adversary-generating processes is sending both adversaries *and* targets, we have to implement a custom solution. Thus, we loop through each of the possible adversary-generating processes (ranks 1 to n) and use `MPI_Iprobe` to check whether they have posted a send of adversaries and targets. If we encounter one that has, say process with rank p, we immediately receive that batch of adversaries, and then continue the loop probing from the next process with rank $p + 1$ (or 1 if $p = n$).

The second type of communication of the model weights naturally allows for compute-transfer overlap. After every weight update, the weight-updating process sends a new copy of the weights to each adversary-generating process. Meanwhile, the adversary generating process constantly has an asynchronous receive posted, (`MPI_Irecv`) which allows for the desired compute-transfer overlap on the adversary-generating end. That is, after every t adversary steps (where t is a hyperparameter called `kCheckReceiveInterval` in the code), the adversary checks if its previous asynchronous receive it posted has completed. If so, the adversary checks if any additional new model(s) have been sent via `MPI_Iprobe` and if so, stores the latest one in its copy of the model weights. Right after, it posts a new asynchronous receive and then continues adversary generation. Thus, the communication related to receiving model weights is completely hidden and done simultaneously with adversary generation. This is absolutely necessary for us to achieve any reasonable speedup with this algorithm.

Parallelizing with OpenMP

We implemented a second form of parallelization that employs OpenMP. To see why this was reasonable, let us revisit the computation necessary to generate adversaries. Recall that in projected gradient descent, we need to compute

$$\frac{\partial L}{\partial I_v} = \frac{\partial L}{\partial x_L}\frac{\partial x_L}{\partial x_{L-1}}\cdots\frac{\partial x_2}{\partial x_1}\frac{\partial x_1}{\partial I_v}$$

which is effectively a chain matrix multiplication. Note that if we use PyTorch to compute this gradient, it computes this matrix product sequentially since it performs backpropagation which is inherently a sequential process. Hence, we aimed to employ OpenMP under a shared memory framework to compute this chained matrix multiplication per adversary generation MPI process.

Computing the Gradient Manually

In order to parallelize the chain matrix multiplication detailed above, we needed to compute the gradient manually. While the gradient with respect to the linear layers is trivial, the gradient with respect to the convolutional layers is not since convolutional is naturally represented as a sliding window as opposed to a matrix multiplication. One approach is to represent the convolution as a matrix multiplication which is done by generating a doubly blocked Toeplitz matrix using the weights in the convolution layer. However, after implementing and even parallelizing the Toeplitz computation, we found that it was a computationally expensive operation and that the PyTorch backward function was substantially outperforming our manual computation. This indicates PyTorch likely has an optimized function to perform backpropagation on convolutional layers that we are unable to leverage since we cannot access PyTorch's gradient matrices which is one of the reasons we failed to find use for our OpenMP implementation (detailed below) in performing this gradient computation.

OpenMP for Chain Tensor Multiplication

The need for a fast algorithm to multiply potentially large chains of tensors lead us to an OpenMP implementation for tensor multiplication. Torch does not support functionality for chain tensor multiplication - it is up to the user to implement such an algorithm. As such, our goal was to attempt to beat a naive implementation of chain tensor multiplication that that simply iterated through the array of `torch::Tensor` and multiply using `torch::matmul` in index order.

When considering algorithms to implement, we decided that it would be interesting to test out two implementations and see if one was better than the other. The first algorithm we implemented was a simple inter-op parallel algorithm splitting up the array into contiguous pieces. The second algorithm we implemented was a more complex algorithm that allowed threads to visit a ”queue” of jobs that were available, where jobs referred to pairwise matrix multiplications that were available. To ensure that there are no data races, and that code is as parallel as possible, the granularity of synchronism is on the level of each tensor - that is, there is array of locks that correspond to each matrix within the chain. This ensures that there are no conflicts between threads when searching and updating through possible jobs. Both parallel algorithms and the sequential algorithm are outlined in pseudocode below (see Appendix: **Algorithm 1** for sequential, **Algorithm 2** and **Algorithm 3** for parallel).

One thing to note is that the implementation of `matmul` in the torch library is already multithreaded. As such, we wish to compare our intra-op parallelism using only a single thread to the sequential implementation using the same number of threads. This is allows for fair comparison by keeping the number of resources constant across implementations.

Problems with Backpropogation Parallelization

After implementing the OpenMP code to parallelize backpropogation, we ran into some issues. The first problem is with regard to the manual computation of the gradient which is detailed in the previous sections, in which computing Toeplitz matrices became computationally expensive. The second problem and main issue with the approach is with regard to the computational inexpensiveness which is inherent to backpropagation.

To expand on the second more, see that for modern CNNs, we take matrix products from the *back to the front*. Due to the first matrix being the matrix represnting the impact of the final layer weights to a scalar loss, we start by multiplying a $1 \times N^{\ell}$ matrix, where N^{ℓ} is the final layer of weights. We then see that sequential matrix multiplication has the edge here, as multiplying a one dimensional matrix takes much less computation than multiplying larger matrix sizes, which then becomes the bottleneck.

For example, consider the example where we have matrix sizes of 1×80, 80×5000, 5000×1440, and 1440×768. In the sequential case, since the row dimension is always 1, we perform four SGEMV kernel operations. Note that in the parallel case, however, we would split up the matrix calculations into 1×80, 80×5000, and 5000×1440, 1440×768. The first computation requires SGEMV kernel, while the second requires a SGEMM kernel, the latter of which is vastly more expensive computationally. Thus, it is not reasonable to parallelize in this case.

Though we could not parallelize in this particular case, we derived results that were desirable for other general cases. These results will be noted in in our benchmarks.

Validation/Verification

The sequential version of our code (with PyTorch intra-op parallelism enabled) is the same (translated to C++) as that implemented by Madry et al. [4] in their state-of-the-art work. Thus, our performance comparisons in all future sections are directly comparable to their work.

4 Performance Benchmarks and Scaling Analysis

Roofline Analysis

Flops

The two primary kernels we employed in this project were a vector-matrix product for the linear layers and a convolution for convolution layers in our neural network. First, we compute the number of flops for a vector-matrix project for a linear layer. Let us denote the length of the input vector by N and let us denote the size of the weight matrix we are multiplying it with by $N \times M$. Then, the number of flops of the matrix-vector product is $M(2N-1) = 2MN - M$. Next, let us compute the number of flops for a convolution with no padding and a stride of 1. Let us denote the dimension of our input tensor by $X \times X$, the number of input channels by C_i and the number of output channels by C_o. We will denote the dimension of our square kernel by K. Then, the total number of flops in the convolution is equal to $C_i C_o (2K^2 - 1)(X - K + 1)^2$.

Our model can be expressed as follows (see appendix fig. 2):

$$\text{softmax}(\text{linear2}(\text{relu}(\text{linear1}(\text{flatten}(\text{relu}(\text{conv2}(\text{relu}(\text{conv1}(\text{x})))))))))$$

where the dimensions of the weights in each of the layers is as follows:

$$\text{conv1} = (10, 1, 5, 5), \text{conv2} = (20, 10, 5, 5), \text{linear1} = (8000, 50), \text{linear1} = (50, 10)$$

Using these expressions, we can calculate the number of flops required for a forward pass through our model. The grand total flops is 5009460 for a forward pass and 10018920 for a backward pass. The individual linear and convolutional layers are also calculated. Please see appendix for full calculations.

Memory Accesses

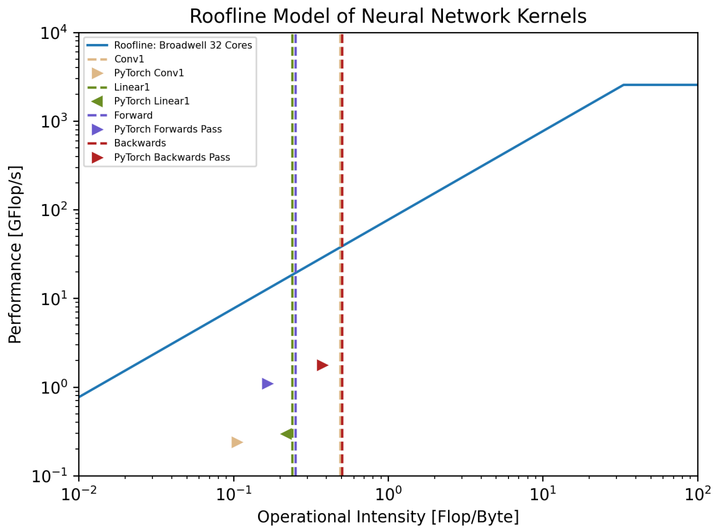
For memory accesses, we do an approach, without the assumptions that everything fits in the cache perfectly. From this, we derive that for forwards and backwards passes, we would need 5196750 memory operations.

To check our arithmetic, we measured the number of load and store operations with PAPI. For CONV1 and Linear1, we saw on the order of magnitude 600,000 and 200,000 load/store operations, respectively. Noting that Pytorch vectorizes code with SSE, multiplying these operations by 4 for the 4 floats loaded into the XMM registers at a time yields results on the same order of magnitude as our analysis, and nearly exactly the results we expect analytically.

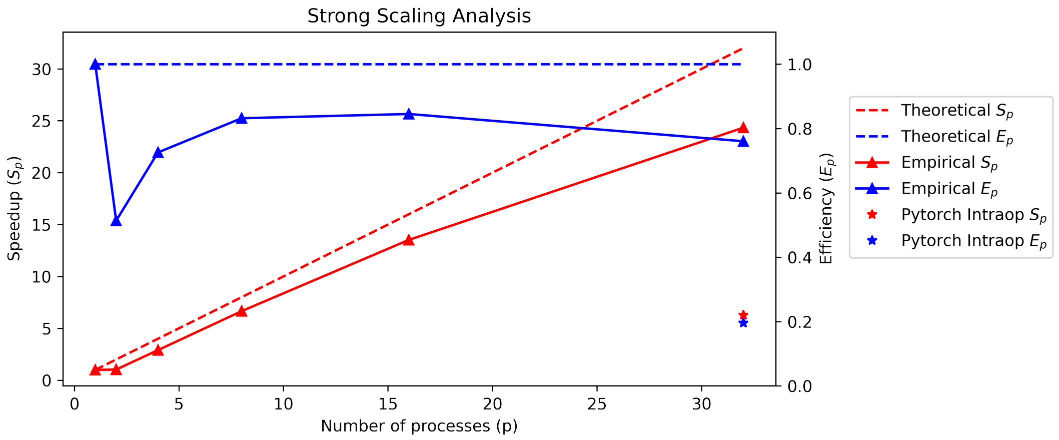
Operational Intensities

Using the calculations above, we can compute the operation intensities for our compute kernels. For the sake of brevity, we do kernel analysis on one linear, one convolution, one forward pass, and one backwards pass. For the LINEAR1, we see a compute of around $2.4e-01$. For CONV1, we see a compute of around $4.89150e-01$. For forward pass, we see a compute of around $2.509900e-01$, and for backwards passes, we see a compute of around $5.0131e-01$. These make sense - though the operational intensity of the CONV1 layer is higher, because the compute kernel of forward and backwards are dominated by a large linear problem size, the operational intensity becomes biased towards linear.

The roofline model below shows the performance of the forward, backwards, linear, and convolutional kernels. Note that all the kernels are memory bound due to the small sizes of the problems and the analysis for our current problem size. Though we show the performance here, note that our objective is not to enhance the performance of all the kernels - rather, our goal was to optimize backwards. It is also crucial to note that Pytorch does not meet the roofline, even with all of their optimizations, meaning that it is still possible to optimize further.



Strong Scaling Analysis



To formulate our strong scaling analysis, we first note the two modes of parallelization present in this problem: parallelization across MPI processes for adversary generation and the parallelization across cores by PyTorch when PyTorch performs computations. We will first address the theoretical and empirical S_p and E_p and then move on to explaining the PyTorch Intraop S_p and E_p shown in the graph above.

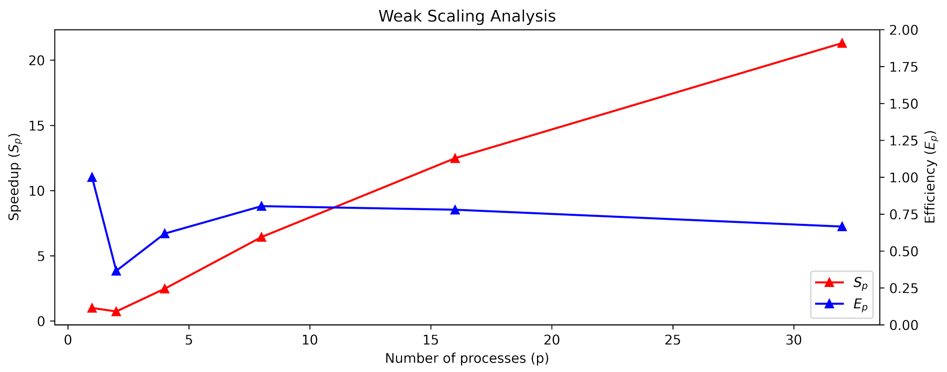
For this strong scaling analysis, we fix the number of adversary generation steps (which is the problem size) at 32. We then note that the serial fraction of our code is $f = 0$ because both weight training and adversary generation algorithms can be split up onto individual MPI processes and parallelized. Using the equation $S_p = \frac{1}{f+\frac{1-f}{p}}$, we graphed the theoretical speedup under the strong scaling model as a function of the number of processes p. This serves as an upper bound for our empirical speedup. Note that since the serial fraction of the code is 0, we observe a theoretical efficiency of 1 for all values p.

For the empirical speedup, we conducted experiments timing the execution time of the parallel algorithm for a fixed number of adversary generation steps 32 and a varied number of processes $p = 1, 2, 4, 8, 16, 32$. We also fixed the number of cores available to our algorithm and PyTorch to be equal to p. Namely, T_p represents the execution time of our algorithm in which both the weight updates/adversary generation and PyTorch were limited to using p cores. Since the serial fraction of our code is 0, the execution time for the sequential baseline T_s is equal to T_1. So, our sequential baseline is the amount of time is takes to run a weight update and an adversary generation process that uses 32 steps on a single process that uses a single core. Hence, the empirical speedup we compute for varying values of p is $\frac{T_1}{T_p}$ which is shown on the graph above. We note a significant speedup as the number of processes grows, achieving results slightly below the theoretical upper bound.

Next, we address the two stars in the graph which represent the PyTorch intraop speedup and efficiency. The PyTorch intraop speedup is equal to $\frac{T_1}{P}$ where P denotes the execution time of the sequential algorithm in which the weight update and single adversary generation both run on one core but PyTorch is given access to all 32 cores to parallelize its computations. We note that the speedup obtained here is significantly below that of our parallel algorithm when using all 32 cores (denoted by S_{32}) for both adversary generation/weight updates and PyTorch computation. This demonstrates that most of the speedup that we obtain is from the parallelization across MPI processes as opposed to the parallelization employed by PyTorch.

We also briefly note that the results in our presentation were not accurate since we computed the speedup S_p using $\frac{P}{T_p}$ instead of $\frac{T_1}{T_p}$ (since we were not aware that PyTorch was using all 32 cores). The results in the graph reflect this correction.

Weak Scaling Analysis



To conduct our weak scaling analysis, we devised our experiments such that the work executed per process remained fixed. Let us define a unit of work as either a single weight update or a single step within adversary generation. Then, our sequential baseline (whose execution time we denote with $T_s = T_1$), will be defined as the amount of time is takes to run a single weight update (on the entire dataset) on a single process that has a single core (note that this is different that the sequential baseline presented in our strong scaling analysis). So, we can define T_p (for $p = 1, 2, 4, 8, 16, 32$) as the execution time for an algorithm that uses p processes in which 1 of the processes is performing a single weight update (on the entire dataset) and the other $p - 1$ processes are performing $p - 1$ adversary steps on $\frac{N}{p-1}$ of data points where N denotes the total size of the dataset. This ensures the amount of work done per process is fixed for varying values of p. Using this formulation, we can define $S_p = \frac{pT_1}{T_p}, E_p = \frac{S_p}{p}$. These expressions are graphed as a function of p as shown above. We note that we achieve a speedup that is consistent with an increase in the number of processes and have an efficiency that hovers close to 0.75. We attribute the slight decrease in speedup from $p = 1$ to $p = 2$ processes to communication overhead associated with MPI.

Analysis of Parallel Matrix Multiplication

As discussed earlier, the parallel matrix algorithms were not employed in backpropogation. However, upon testing in environments where the first matrix is not a vector, we see that there is a place for these algorithms. We tested in three environments, the first two of which are homogeneous matrix size environments, and the third of which is heterogeneous matrix size environments:

1. Multiplying 100 1000×1000 matrices together
2. Multiplying $50,000$ 100×100 matrices together
3. Multiplying 100 random sized matrices together (of size 1-1000)

We average all of these metrics over 100 trials. In addition, for the sequential algorithm, we allowed Pytorch to use all of the cores for intra-op parallelism of matrix multiplication, while for the parallel algorithms, we restricted Pytorch to only use one thread for matrix multiplication whilst using all the cores for inter-op parallelism. The plots are shown in appendix 4).

Here, we see that our inter-op parallelism is **faster** than PyTorch's implementation. We also see some interesting comparisons between the simple contiguous array allocating parallel algorithm (Algorithm 2), and the more complex queue based parallel algorithm (Algorithm 3). When we have homogeneous sized matrices, Algorithm 2 outperforms both algorithm 1 and algorithm 3 significantly. This is due to the fact that Algorithm 2 essentially performs the same work as Algorithm 1, but reduces all of the matrices into a significantly smaller number of matrices to deal with, while Algorithm 3 must spend more time with overhead in locking and unlocking.

When looking at heterogeneous matrix sizes, however, we see that Algorithm 3 clearly beats all the other algorithms out. This is due to algorithm 3 allowing for increased amount of parallelism throughout the entire chain product - while algorithm 2 must wait at the implicit barrier after `pragma omp parallel`, algorithm 3 has no such restrictions, and threads can move on immediately after they have finished their matrix multiplications. Something interesting to note here is that algorithm 2 is outperformed by algorithm 1. This is exactly in line with our intuition - the probability that the threads in the simple parallel algorithm start with a larger matrix size is amplified due to the number of threads taking on such tasks, meaning that we have threads that will be a bottleneck. Sequentially, since we only have one thread, the probability of getting higher values is lower, meaning that even though we do not have the inter-op parallelism, without an expected larger matrix to start at, we see on average a lower time.

5 Resource Justification

We used an AWS Elastic Compute Cloud server for many of our experiments, so we will outline the specifications of it here. The specs for the `m4.16xlarge` instance we used would be 32 physical cores (64 vCPUs) and 256 GiB of RAM. The processor type is Intel Xeon E5-2686v4 (Broadwell), which has a base clock of 2.3 GHz and turbo speed of up to 3.0 GHz. We decided to use AWS because of reliability issues on the FASRC cluster. We chose the AWS `m4.16xlarge` instance because it was a similar configuration to what is available on the FASRC cluster. The cluster's Broadwell nodes are Intel Xeon E5-2683v4 which have a base clock of 2.1 GHz and a turbo speed of up to 3.0 GHz. So, we would expect our AWS server to be similar to or slightly faster than the cluster.

When thinking about possible tasks, there are three important ones that have wide use cases within the deep learning community:

1. Train a simple neural network with adversarial examples with MNIST
2. Perform a hyperparameter search over 100 neural networks with MNIST
3. Perform a hyperparameter search over 100 neural networks with ImageNet

The resources needed are computed as such: in our case, the representative benchmark is 32 cores (what we benchmark with). The corresponding wall time is 50s, which is then equivalent to ~ 0.44 core hours, as a result of the following product:

$$0.44 \text{ core hours} = 32 \text{ cores per node} \times 1\text{node} \times \frac{50s}{3600\frac{s}{\text{hour}}}$$

Our final task is to compute the best neural network, which may or may not involve a hyperparameter search over the learning rate, weight decay, etc. Each of these simulations then require a full training loop, which requires around 100 epochs. In addition, we might wish to classify larger, more interesting datasets than just MNIST (such as CIFAR-10 or ImageNet). These, in general, will change the wall time by a constant factor, which we also factor into our analysis. For ImageNet, because the dataset is 20 times larger, we need to multiply our wall time by 20, resulting in 8.9 core hours.

Table 1.

	Task 1	Task 2	Task 3
Simulations per task	1	100	100
Epochs per simulation	100	100	100
Node hours per iteration	0.44 s	0.44 s	8.9 s
Total core hours	44	4400	89000

Table 1: Justification of the resource request

Table 1 shows the total node hours necessary for all three of these tasks. A total of 93444 core hours would be necessary, most of which is due to Task 3 being much more computationally expensive than the other two.

References

[1] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. 2014. DOI: 10.48550/ARXIV.1412.6572. URL: https://arxiv.org/abs/1412.6572.

[2] Marius Hobbhahn and Jaime Sevilla. *What's the backward-forward flop ratio for neural networks?* Dec. 2021. URL: https://www.alignmentforum.org/posts/fnjKpBoWJXcSDwhZk/what-s-the-backward-forward-flop-ratio-for-neural-networks.

[3] Yann LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[4] Aleksander Madry et al. *Towards Deep Learning Models Resistant to Adversarial Attacks*. 2017. DOI: 10.48550/ARXIV.1706.06083. URL: https://arxiv.org/abs/1706.06083.

[5] Paulius Micikevicius et al. *Mixed Precision Training*. 2017. DOI: 10.48550/ARXIV.1710.03740. URL: https://arxiv.org/abs/1710.03740.

[6] Leslie N. Smith and Nicholay Topin. *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates*. 2017. DOI: 10.48550/ARXIV.1708.07120. URL: https://arxiv.org/abs/1708.07120.

[7] Christian Szegedy et al. *Intriguing properties of neural networks*. 2013. DOI: 10.48550/ARXIV.1312.6199. URL: https://arxiv.org/abs/1312.6199.

Appendix

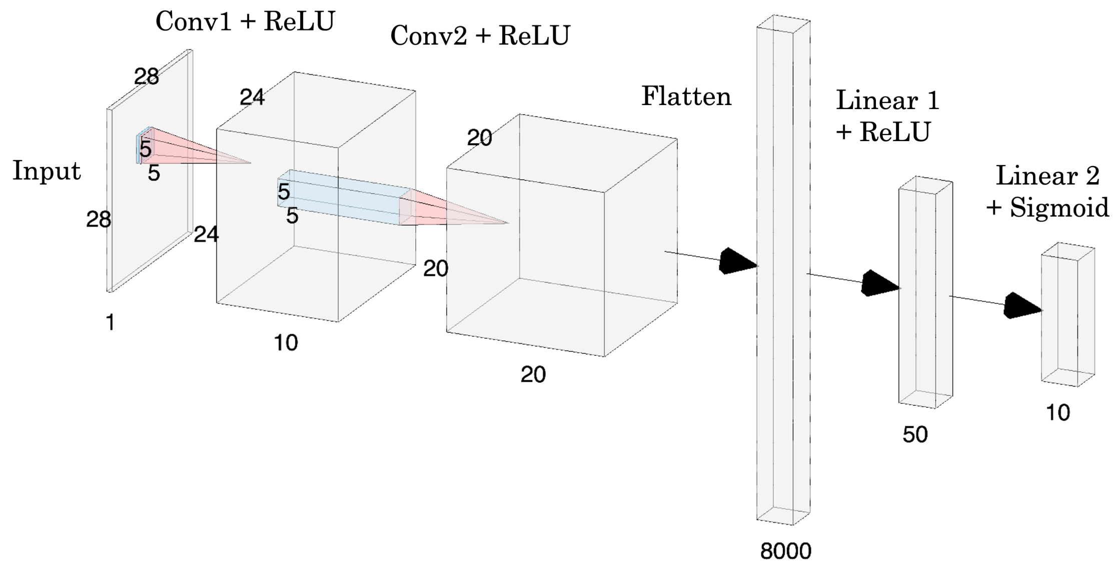


Figure 2: CNN Architecture

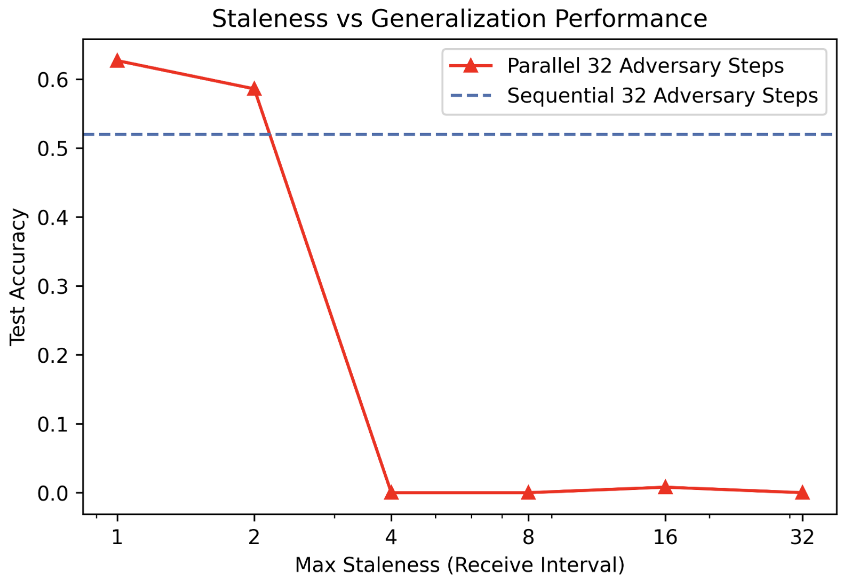


Figure 3: Staleness has a large impact on accuracy. Setting the staleness to be 1 yields the best accuracies and does not take longer than other staleness values.

Roofline Flop/Memory Calculations

The size of our input image x is $(1, 28, 28)$. So, the number of flops for conv1 is 282240. Next, we perform an element wise ReLU on the output (which is of shape $(10, 24, 24)$). A single ReLU operation takes one flop (a comparison which we don't count and a multiplication). Hence, the number of flops for the ReLU is 5760. The next operation we perform is conv2 which takes in an input of size $(10, 24, 24)$. The number of flops for this portion of the forward pass is 3920000. Then, the tensor is flattened and we now execute a vector matrix product with linear1. This operation takes 792000 flops. Next, we have another ReLU that takes as input a vector of length 50 which also takes 50 flops. Then, we take this vector and multiply it by linear2 which takes 990 flops and produces an output vector of size 10. The final computation performed by the model is a softmax which can be denoted by the following function:

$$\sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

where $z \in \mathbb{R}^k$ for $i = 1 \dots K$. To measure the flops here, we will have to make an approximation as to the degree of a mini-max polynomial used by PyTorch to approximate e^x. Let us assume that they employ a polynomial of degree 10. Then, for a given entry in the softmax, assuming we use repeated squaring to compute the powers in the polynomial, we would execute $0+1+2+3+3+4+4+5+4+5+5+10 = 46$ flops where the first 11 terms in the summation denote the number of multiplications done for each term in the 10-degree polynomial and the last term denotes the number of additions to evaluate the sum. Since we do this for each element of the output, this yields 460 flops for the softmax function. Hence, our grand total number of flops for the forward pass of the model is

$$282240 + 5760 + 3920000 + 8000 + 799950 + 50 + 950 + 460 = 5009460$$

We cite the following profiling study on the number of flops done in the backward pass of a neural network [2]. This study states that the ratio of flops between the forward and backward passes for a given network is approximately 1 : 1 for the first layer and 1 : 2 for the rest of the layers. Hence, using these ratios and our computations from above, we have the following number of flops for the backward pass in our network:

$$1 * (282240 + 5760) + 2 * (3920000 + 8000 + 799900 + 50 + 950 + 460) = 9730920$$

To calculate the memory bandwidth needed, we start with linear. See that for linear, we are doing exactly the SGEMV kernel. This results in $M(2N + 1)$ memory operations for linear passes. For convolutional layers, we note that we need to access the kernel matrix, as well as a submatrix the size of the kernel matrix for each cell of the resulting convolution. For our purposes, assume that cache block size ie 64 bytes - then, it is not the case that all the data can fit in the cache in one iteration. We then need $C_O * (X - K + 1)^2 * (2 * K^2 + 1)$ loads for convolution. For non linearities such as ReLU and Softmax, we assume memory operations that of the object the functions are to be applied, meaning for a matrix of size NM, we have memory operations of that of $2NM$ (loads and stores).

For example, the memory operations for CONV1 with this model would be measured as $10 * 1 * 24^2 * (2 * 5^2 + 1) = 293760$

The number of memory operations of LINEAR1 is $50(2 * 8000 + 2) = 800100$ The output from this convolution is size $(20, 20, 20)$ which means that the ReLU operation executes 8000 flops with memory operations 16000.

Continuing on with this analysis similarly on CONV2, LINEAR2, and non-linearities results in the grand total of memory operations is

$$293760 + 5760 + 4080000 + 16000 + 800100 + 100 + 1010 + 20 = 5196750$$

The memory operations for the backwards pass is a round of reversed operations of the forwards, meaning that we still have the same number of memory operations.

OpenMP Chain Matrix Multiplication Algorithms

Algorithm 1 Sequential

```
Require: $T, n$ {T is the tensor array, n is size of T}
  $t \to T[0]$
  for int i = 1; i < n; ++i do
    $t \leftarrow$ torch::matmul($T[i]$)
  end for
  return t_arr[0]
```

Algorithm 2 Simple Parallel Algorithm

```
Require: $T, n$ {T is the tensor array, n is size of T}
  $nThreads \leftarrow$ omp_get_max_threads()
  $bsz, rem = n/nThreads, n\%nThreads$
  torch::Tensor $t\_arr[nThreads]$
  #pragma omp parallel{
  $tid \leftarrow$ omp_get_thread_num()
  $start, end \leftarrow tid * bsz, (tid + 1) * bsz$
  if $tid < rem$ then
    if $tid = 0$ then
      $end \leftarrow end + 1$
    else
      $start \leftarrow start + tid$
      $end \leftarrow end + tid + 1$
    end if
  else
    $end \leftarrow end + rem$
    $start \leftarrow start + rem$
  end if{resolving load imbalance}
  tensor t = T[start]
  for int i = start+1; i < end-start; ++i do
    $t \leftarrow$ torch::matmul($T[i]$)
  end for
  } {END parallel}
  reduce t_arr
  return t_arr[0]
```

Algorithm 3 Queued Parallel Algorithm

```
Require: $T, n$ {T is the tensor array, n is size of T}
  $Q, N, L, Q\_l$ {Q is task queue, N is pointers of next array, L is omp_lock array, Q_l is lock for task queue}
  $f = F$ {f is a boolean flag to tell threads when computation is done}
  $Q \leftarrow$ all even tasks (even indices)
  $N \leftarrow$ index +1, or -1 if last tensor
  $L \leftarrow$ omp_lock_init()
  while $f = F$ do
    lock Q
    if Q empty then
      unset lock
      continue
    end if
    pop index h from Q
    if L[h] locked then
      unset lock
    else
      if $L[h] > -1$ then
        if L[h] locked then
          Q.push(h)
          unset L[h], Q_l
        else
          $Q.push(h)$
          unset $Q\_l$
          torch::matmul($T[h], T[L[h])$
          $L[h] = L[L[h]$
          $L[L[h]] = -1${update pointers to next task}
        end if
      else
        $f = T$
        unset $L[h], Q\_l$
      end if
    end if
  end while
  return T[0]
```

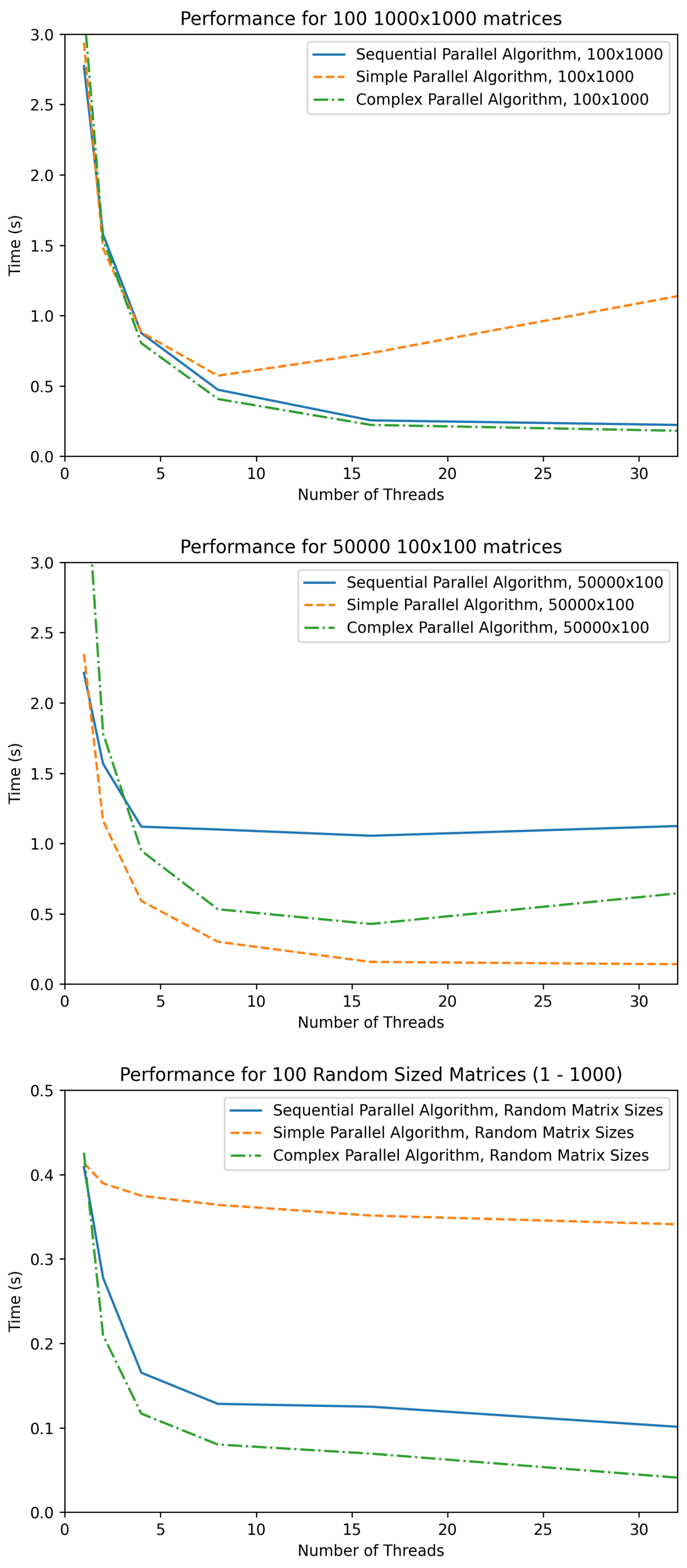


Figure 4: Parallel Matrix Multiplication Performance