Pipelined Training for Convolutional Neural Networks

Jingxing Fang [*1], Emil Gilkes [*1], and Albert Ge [*1]

[1]Harvard University

May 6, 2022

Abstract

Convolutional Neural Networks (CNNs) have enhanced the effectiveness of multispectral remote sensing applications. Understanding our world is not stationary, dataset shift can negatively impact the sustained utility of a trained model. As a result, there is significant need to efficiently retrain models to incorporate data about our everchanging environment. This is an especially important issue when it comes to remote sensing for high-risk applications involving natural disasters, threat assessment, and land development projects. High performance computing (HPC) is proven to be capable of increasing the performance of matrix-matrix operations, which are the foundation of training neural networks. We propose a distributed training system for a well-established model architecture used in satellite image classification. We have demonstrated promising results using two Broadwell nodes on the FAS cluster, achieving 1.3x speedup over sequential training. By partitioning the model layers across these nodes, and by leveraging pipeline parallelism, we were able to reduce the time required to train the model without sacrificing model performance, when compared to the conventional training methods.

1 Background and Significance

Deep Neural Networks (DNNs) are quickly becoming ubiquitous in academia and in industry. Cutting-edge research has demonstrated that DNNs can achieve impressive levels of accuracy on complex tasks, such as computer vision. However, immense resources are required to train these state-of-the-art (SOTA) models. For example, the best models on ImageNet (a 160GB dataset that is often used as a benchmark for SOTA image models) achieve over 90% accuracy, but they contain billions of parameters and can take weeks to train [9]. Training a DNN consists of two stages: 1) the forward pass, in which the input features are passed through the model and into the objective function to calculate the loss, and 2) the backward pass, in which the model weights are updated using the gradients of the weights with respect to the loss.

Since the breakthrough of AlexNet [4] in 2012, CNNs have been the dominating model architecture for computer vision. As a result, CNNs have become popular for multispectral remote sensing. In a 2015 study [5], researchers from Louisiana State University and the NASA Ames Research Center released DeepSat, a dataset containing a subset of images from the National Agriculture Imagery Program. As part of the study, the researchers manually labelled uniform image patches belonging to a particular land-cover class. There is great demand for up-to-date land cover data for any sort of sustainable improvement programs since it serves as one of the real input information criteria. In 2018, researchers presented results from applying their Modified AlexNet architecture [8] to classify the six land-cover classes in the DeepSat SAT-6 dataset. The model architecture contains more than 600,000 parameters.

The negative impact of dataset shift on the sustained performance of ML models is well documented, so it can be extremely valuable to efficiently retrain models. Understanding that there is great demand for up-to-date land cover data, we have designed a parallel training system for Modified AlexNet. Our system aims to leverage high performance computing (HPC) to enable efficient retraining of the model, therefore reducing the consequences of dataset shift.

[*]Equal Contribution

2 Scientific Goals and Objectives

Remote sensing is a widely used tool in emergency response and disaster recovery. Satellites equipped with camera sensors can detect and analyze threats and incidents, such as fire hazards, flood warnings, landslide zones. Furthermore, these devices can be used to monitor remote locations, which can often be difficult to get to by foot. On-ground resources (like high-performance clusters) can be utilized to train these algorithms where there are more computational resources.

Our primary objective is to leverage the resources of HPC to speedup the training time of deep learning tasks. In particular, we explore a general form of instruction-level parallelism, known as pipelining, to utilize several compute instances and collectively train a model. By increasing the throughput of training, the model should be able to train on a higher number of images, and in theory, converge faster as well.

3 Algorithms and Code Parallelization

For our dataset, we reference DeepSat (SAT-6) Airborne dataset to train on remote sensing data, licensed under Creative Commons (CC0). These images are extracted from the National Agricultural Imagery Program (NAIP) dataset. This dataset contains 400,000 28x28 training images and 100,000 test images. Each image contains 4 channels, which correspond to red, green, blue, and Near Infrared (NIR) data. The dataset identifies six prediction classes, which represent the six broad land covers: barren land, trees, grassland, roads, buildings and water bodies. Further information about methodology of data collection can be found on Kaggle [1].

We are a user of the Pytorch version 1.4 library, for its ease in developing and training deep learning architectures. Pytorch 1.4 is necessary for the execution of our specific application, due to a gradient computation check that was added in version 1.5 and later. We specifically will adjust the weights needed for gradient computation, and this is discussed in the following section on weight stashing.

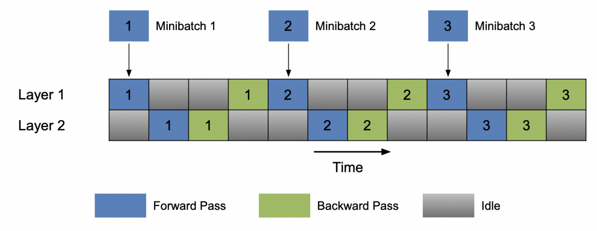
While the code is written in Python, Pytorch sits atop a substantial C++ codebase providing foundational data structures and functionality such as tensors and automatic differentiation [3]. We contribute to the development of distributed machine learning algorithms.

To communicate between compute instances, we make use of gloo [2]. Gloo is a collective communications library. It comes with a number of collective algorithms useful for machine learning applications. We elect to use gloo over MPI because of its tight integration with the Pytorch ecosystem, and is optimized particularly for communicating Tensors (the canonical Pytorch datatype) between ranks. Preliminary tests have shown that point-to-point communication is faster using the gloo backend, over MPI. Furthemore, its communication interface is very similar to MPI, which allow for ease of development.

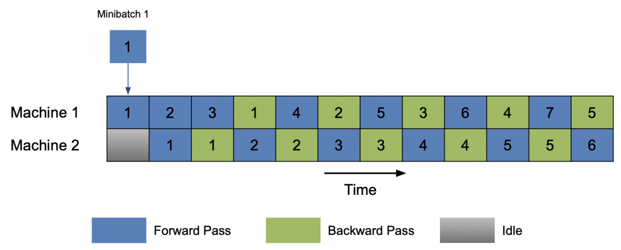
Pipeline Design

Our pipeline design takes inspiration from the PipeDream research published in 2018 [6]. In order to make use of pipelining, we first partition our model into stages. Each compute node in our training framework is responsible for exactly one stage. When training starts, the first node will process the first minibatch and produce an intermediate feature map. This map is passed to the subsequent stage, and the process repeats until all stages have computed their forward passes. At the final stage, the training loss is computed by comparing the model's predictions against the targets. During backpropagation, each stage will pass its gradients with respect to its input to each preceding stage, until all stages have computed the backward pass. Figure 1a shows a simplified version of our partitioned model across time, by sequentially sending updates between the layers.

Our pipelined approach takes advantage of idle compute time in each stage during the calculation of forward and backward passes, as depicted in Figure 1b. Each stage is able to compute different minibatches simultaneously. For example, as soon as the first machine completes minibatch 1, it can schedule the forward pass on minibatch 2, while machine 2 receives and computes its forward pass on minibatch 1. The communication overhead between stages is partially hidden by the overlap of computation. As a



(a) Sequential Execution



(b) Pipeline Scheduling

result, there is very little per-stage idle time, and each machine can execute forward or backward work as they become available.

Weight Stashing

One issue that pipelining creates is that weights are now out of sync, and backward passes no longer are guaranteed to be calculated using the same weights. In order to maintain partial consistency of model parameters, we reference PipeDream's custom optimizer function, known in the literature as weight stashing [6]. Weight stashing maintains multiple versions of model weights at each stage, but not across stages. This ensures that within a stage, the same version of model weights are used for the forward and backward passes of a minibatch. Figure 1b illustrates that machine 1 computes minibatch 4's forward pass, after the backward pass from minibatch 1 completed and had its weights updated. Later down the pipeline, when machine 1 computes the backward pass for minibatch 4, the gradient is computed using the same weights (i.e., after minibatch 1 was completed). This necessitates Pytorch 1.4, as we restore the weights to a previous version during the computation of a different minibatch. This gradient is then applied to the most recent version of the weights.

Because weight stashing is an optimization technique, it is not the primary focus of parallelization in our project. Nevertheless, we provide a couple of high-level justifications for its use. One is that, due to minibatch stochasticity, each gradient step is already sufficiently randomized, and with a large enough batch size, the model's loss will still converge. The second assumption is that, with the use of momentum-based optimizers like Adam [7], the gradient step is a moving average of all previous steps - thus, as long as the optimizer is moving in the correct general direction, any deviations in the gradient step will average out with enough iterations.

Communication

Finally, we detail our inter-stage communication layer, which also takes inspiration from PipeDream. Each stage is equipped with four queues, which act as buffers to send and receive tensors. The four queues (send-forward, send-backward, receive-forward, receive-backward) are asynchronous; that is, they are each controlled by a separate thread which removes and adds items to the buffer as they become available, while the main process performs its forward or backward pass. These queues are generalized versions of nonblocking operations, as they allow multiple tensors to be queued together, and allow for more flexibility when training multi-stage pipelines (at least 4 stages).

Because tensor shapes can be variable (for example, the shape of a feature map is different from a gradient map), we defined a simple two-step communication protocol for sending and receiving tensors. A stage must send, in separate messages, the tensor shape and the actual tensor itself. This way, the receiving stage will know how large of a tensor to allocate before receiving the actual tensor.

Validation, Verification

We validate the numerical consistency of our parallel design by comparing the accuracy achieved using the parallel training system (98.76%) with the accuracy achieved using the conventional sequential training (98.99%). As can be seen in later sections, both training methodologies achieve the same accuracy.

Moreover, since we adopt a model architecture first implemented by researchers on the same dataset, we can verify that the accuracy of our trained model convergence of our model is comparable to their results[8]. Although the researchers in the were able to reach almost 100% accuracy, their benchmarks were used to predict only four out of size land-cover classes, whereas our model predicts on all six classes.

4 Performance Benchmarks and Scaling Analysis

Sequential Baseline

We trained the modified AlexNet model with a batch size of 128 for 2 epochs on the DeepSat training set. There are 2532 minibatches going through the neural network per epoch. We used the Adam optimizer with a learning rate of 0.001. As shown in Fig 2, we achieved a train accuracy of 96.78% after 2 epochs, and the final test accuracy was 98.99%. We used the Intel Xeon E5-2683v4 architecture (Broadwell node) with 2 cores for the training, and it took a total of 3400 seconds to train the model sequentially for 2 epochs.

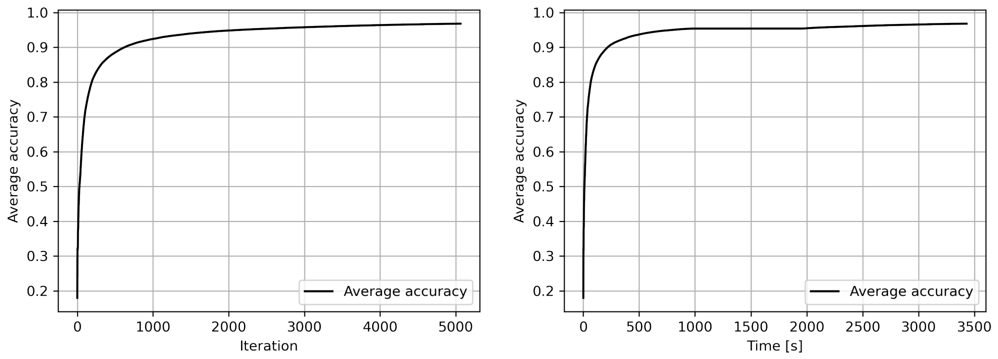


Figure 2: Modified AlexNet sequential training result.

Sequential Profiling

In the forward pass, each layer in our Modified AlexNet takes an input and calculates the output in sequence. In the backward pass, each layer calculates the gradients of its output with respect to its input and weights in reverse order. The optimizer then updates all the model parameters based on the gradients calculated in the backward pass. We measured the calculation time of both the forward and backward passes for each layer of our neural network as shown in Table 1. The forward pass took 137.24 ms in total, of which the fourth Conv2d layer is the most time-consuming layer. The second MaxPool2d layer also took up a lot of forward computation time because there are a lot of comparison operations in this layer. The backward pass took 205.43 ms in total, of which the fourth Conv2d layer is the most time-consuming part, taking up 55.11 ms, on average. This profiling identifies the fourth Conv2d layer as the bottleneck in our algorithm.

Node Level Performance

We perform a roofline analysis on the fourth convolutional layer of the Modified AlexNet. According to the architecture parameters (Table 2) and using $p = 32$ bit floating point precision, the nominal peak arithmetic performance is

$$\pi = f \times n_c \times \frac{w_s}{p} \times \phi = 134.4\text{Gflop/s}$$

During the forward pass of the fourth Conv2d layer, an input tensor X of shape $[B, S_i, C_i] = [128, 12, 12, 96]$ is convolved with 64 $S_k = 3 \times 3$ kernels K to calculate an output tensor Y of shape $[B, S_o, C_o] = [128, 12, 12, 64]$, where B is the batch size, S_i, S_o and S_k are the input feature map shape, output feature

Table 1: Forward and backward calculation time of each layer in Modified AlexNet.

Layer No.	Layer name	Forward time (ms)	Backward time (ms)	Layer No.	Layer name	Forward time (ms)	Backward time (ms)
0	Conv2d+ReLU	1.740	5.889	8	Conv2d+ReLU	16.418	29.770
1	BatchNorm2d	2.135	4.045	9	MaxPool2d	2.081	0.654
2	MaxPool2d	9.606	1.649	10	Flatten	0.036	0.013
3	Conv2d	20.263	36.960	11	Dropout	2.410	0.236
4	BatchNorm2d	10.355	17.948	12	Linear+ReLU	0.916	1.614
5	MaxPool2d	23.306	3.494	13	Dropout	0.781	0.059
6	Conv2d+ReLU	22.732	46.944	14	Linear+ReLU	0.160	0.300
7	Conv2d+ReLU	24.248	55.594	15	Linear	0.051	0.262

Table 2: Intel Xeon E5-2683v4 architecture parameters

Base clock rate f	2.10×10^9 cycle/s
Number of cores n_c	16
SIMD vector width w_s	256 bit
Total flop per cycle ϕ	4 Flop/cycle
Peak memory bandwidth β	76.8 GB/s

map shape, and kernel shape, respectively, and C_i and C_o are the numbers of input channels and output channels, respectively. Because each element in the output tensor is the sum of the element-wise production of a kernel and an area in the input tensor with the same shape of the kernel, the total flops of forward calculations in the fourth Conv2d layer is

$$B \times S_o \times C_o \times S_k \times C_i \times (1_{add} + 1_{mul})$$

We load the input tensor and kernels, and write the output tensor, so the total memory accesses are

$$B \times (S_o \times C_o + S_i \times C_i) + S_k \times C_i \times C_o$$

During the backward pass of the fourth convolutional layer, we calculate the partial derivative of the loss function L with respect to the input tensor and the layer parameters according to the chain rule

$$\frac{\partial L}{\partial K_{i,j}} = \sum_{m,l \in S_o} \frac{\partial L}{\partial Y_{m,l}} \frac{\partial Y_{m,l}}{\partial X_{i,j}}, \ \frac{\partial L}{\partial X_{i,j}} = \sum_{m,l \in S_o} \frac{\partial L}{\partial Y_{m,l}} \frac{\partial Y_{m,l}}{\partial X_{i,j}},$$

for which we need to read in $\frac{\partial L}{\partial Y}$, X, and K, and write $\frac{\partial L}{\partial K_{i,j}}$ and $\frac{\partial L}{\partial X_{i,j}}$. Therefore, the total number of flops is

$$B(1_{add} + 1_{mul})(S_o \times S_k \times C_i \times C_o + S_k \times C_i \times C_o \times S_i),$$

and the number of memory accesses is

$$B(S_o \times C_o + S_i \times C_i + C_o \times S_k \times C_i + S_o \times C_o + C_o \times S_k \times C_i + S_i \times C_i).$$

With single precision $p = 4$ and the forward and backward calculation time of the fourth Conv2d layer measured in the previous section, we can calculate the operational intensity and performance of this layer (Table 3). From the roofline plot (Figure 3), it can be seen that both forward and backward kernels are compute-bound, and they are quite close to the peak performance. This might be because PyTorch already makes use of vectorization and fused pointwise operations. We can further improve node level performance by making use of thread-level parallelism.

Table 3: The operational intensity and performance of the forward pass and the backward pass of the fourth Conv2d layer.

	Forward pass	Backward pass
Operational Intensity (Flop/Byte)	169.62	50.82
Performance (GFlops)	85.85	73.97

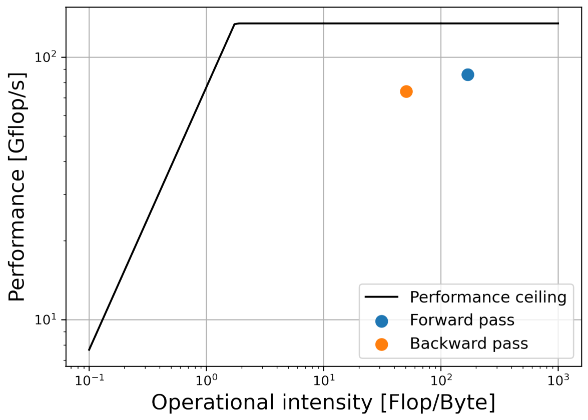


Figure 3: Roofline analysis of the fourth Conv2d layer

Parallel Profiling

.1 Expected Performance

As described in the previous section, we want to partition the Modified AlexNet into stages and assign each stage to a machine. We want the forward and backward execution times of each stage to be as similar as possible to reduce load imbalance on each machine. Therefore, we propose two ways of splitting the layers of Modified AlexNet in Table 1, 2-stage model and 4-stage model, as shown in Table 4 and measured the forward and backward execution times of each stage in each model with a batch size of 128. Theoretically, the total forward time and the total backward time should be the same for the 2-stage model and the 4-stage model, and it can be seen that they are the same within the measurement errors.

With the scheduling mechanism described in the previous section (Figure 1b), at two consecutive time blocks each stage executes one forward pass and one backward pass, and the length of two consecutive time blocks is the average execution time of one iteration (step) for pipelined training. Therefore, theoretically the length of the two time block is determined by the sum of the maximum execution times among the forward and backward passes of all stages. Assuming there is no overhead from communication or any other causes, for the 2-stage model, the theoretical average step time of pipelined training is

$$\max\{\text{Stage0}_f, \text{Stage1}_b\} + \max\{\text{Stage0}_b, \text{Stage1}_f\} = 137.79 + 112.41 = 250.20\text{ms},$$

and the theoretical ideal speed-up with 2-stage pipelining is

$$S_2 = \frac{T_s}{T_2} = (149.32 + 250.20)\text{ms}/250.20\text{ms} = 1.597$$

For the 4-stage model, the theoretical average step time of pipelined training is

$$\max\{\text{Stage0}_f, \text{Stage1}_b, \text{Stage2}_f, \text{Stage3}_b\}+\max\{\text{Stage0}_b, \text{Stage1}_f, \text{Stage2}_b, \text{Stage3}_f\} = 88.94+59.87 = 148.81\text{ms},$$

and the theoretical speed-up with 4-stage pipelining is

$$S_4 = \frac{T_s}{T_4} = (164.89 + 269.37)\text{ms}/148.81\text{ms} = 2.918$$

Table 4: Two splits Modified AlexNet and stage-level forward and backward execution times. The stage-level execution time measurements are averaged over 200 minibatches (iterations) each of size 128. The standard deviation of measurements are denoted in the brackets.

Split		Stage 0	Stage1	Stage 2	Stage 3	
2-stage	Layers	layers 0-6	layers 7-15	\	\	Total
	Forward time (ms)	95.42 (10.4)	53.90 (5.0)	\	\	149.32 (11.5)
	Backward time (ms)	137.79 (15.7)	112.41 (10.8)	\	\	250.20 (19.0)
4-stage	Layers	layers 0-4	layers 5-6	layer 7	layers 8-15	Total
	Forward time (ms)	50.43 (3.2)	58.05 (3.2)	29.50 (2.0)	26.91 (2.0)	164.89 (5.3)
	Backward time (ms)	88.94 (2.8)	58.41 (2.0)	62.15 (1.8)	59.87 (1.9)	269.37 (4.3)

Table 5: Profiling for the 2-stage parallel training with pipelining. The results are averaged over 200 measurements, and the standard deviations are denoted in the brackets.

	Stage 0	Stage 1
Forward time (ms)	116.58 (7.2)	55.68 (6.5)
Backward time (ms)	156.13 (6.4)	113.16 (7.4)
Forward send time (ms)	65.32 (26.5)	\
Backward send time (ms)	\	55.63 (25.9)
Average step time (ms)	283.98 (33.2)	

Note that even without any overhead, we are not able to reach 100% parallel efficiency with pipeline parallelism because of the inevitable load imbalance between stages. For the 2-stage pipelined training, we are not able to hide the communication overhead of Stage 1 sending gradients back to Stage 0, so the actual speedup should be smaller than our theoretical analysis.

.2 Parallel Profiling

We implemented the pipelined training algorithm described in the previous section with the 2-stage and 4-stage model, and measured the average step time of each model as well as the forward and backward communication time of each stage. Because we used synchronous point-to-point send and receive, we only measure the send times for communication times. To implement the distributed training pipeline, there is some extra overhead to each stage than the sequential training code, so we also measured the forward and backward execution time of each stage during the parallel training to check if the overheads are negligible. All the measurements are averaged over 200 minibatches, each with 128 training samples. We chose to profile the parallel algorithm on the 200 minibatch subset because for profiling we are not concerned about the result of the model and only care about the time used to calculate one batch of data with the model, so we do not need to use the entire DeepSat dataset. By averaging over the 200 minibatches, we can obtain relatively accuracy time measurements, while at the same time avoid spending too much runtime on profiling.

Table 6: Profiling for the 4-stage parallel training with pipelining. The standard deviation of measurements are denoted in the brackets.

	Stage 0	Stage 1	Stage 2	Stage 3
Forward time (ms)	65.62 (6.2)	51.41 (5.7)	21.16 (8.3)	33.36 (11.6)
Backward time (ms)	83.83 (4.4)	70.71 (5.7)	70.68 (4.3)	64.49 (12.3)
Forward send time (ms)	214.21 (8.5)	63.08 (9.5)	33.61 (11.9)	\
Backward send time (ms)	\	277.36 (8.1)	81.87 (17.8)	34.16 (19.8)
Average step time (ms)		829.63 (78.7)		

Table 7: Weak scaling analysis.

p	W_p (epochs)	T_p (s)	E_p
1	25	1963.31	1
2	50	2705.74	0.726
4	100	12070.75	0.041

For the 2-stage parallel training (Table 5), it can be seen that the forward and backward execution times of each stage do not deviate much from the sequential execution time. The communication time is less than the execution time of each stage. We are not able to hide the communication overhead of Stage 1 sending gradients backward to Stage 0. The average step time 283.98 ms $\simeq$ 156.13 ms (Stage 0 backward time) + 113.16 ms (Stage 1 backward time) + 55.63 ms (Backward send time) is within the error range, which is consistent with our theoretical analysis in the previous subsection.

For the 4-stage parallel training (Table 6), the forward and backward execution times of each stage also do not deviate much from the sequential execution time. However, the communication times of Stage 0 and Stage 1 are significantly larger than the computation time of each layer, which makes the actual average step time significantly larger than the theoretical average step time calculated in the previous subsection. Therefore, we are not able to see any speed-up with 4-stage parallel training.

.3 Weak Scaling Analysis

For weak scaling analysis, we set W_s = "training for 25 epochs with 200 minibatches of size 128", so W_2 = "training for 50 epochs with 200 minibatches of size 128" and W_4 = training for 100 epochs with 200 minibatches of size 128 and measured the total training time (Table 7). As shown in Figure 4, the weak efficiency is good with $p = 2$, but dropped drastically as p increases to 4. Since our parallel strategy split the Modified AlexNet based on layers and we try to balance the execution time of each stage during the split, we are not able to further divide our model and have higher p for weak scaling analysis.

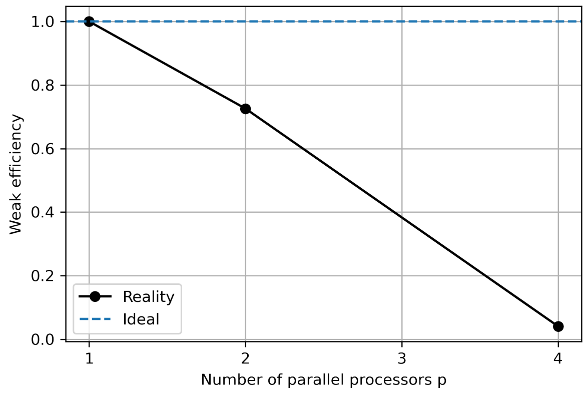


Figure 4: Weak scaling analysis.

.4 Parallel Results

To verify that our pipeline training strategy does not affect the convergence of the convolutional neural network, we trained the 2-stage Modified AlexNet with pipelining on the whole training set for 4 epochs and evaluated the model on the test set. The training finished in 2702.77 s. Compared with the sequential training (Figure 5), our pipelined parallel training algorithm is able to achieve the same level of training accuracy within shorter amount of time. The final parallel training accuracy after 4 epochs was 96.77% and the test accuracy is 98.06%. Compared to the sequential final training accuracy 96.78% and test accuracy 98.99%, we can conclude that our pipeline parallelism algorithm does not significantly hurt the convergence of the model, and that we are able to achieve faster training speed with our pipeline parallelism algorithm.

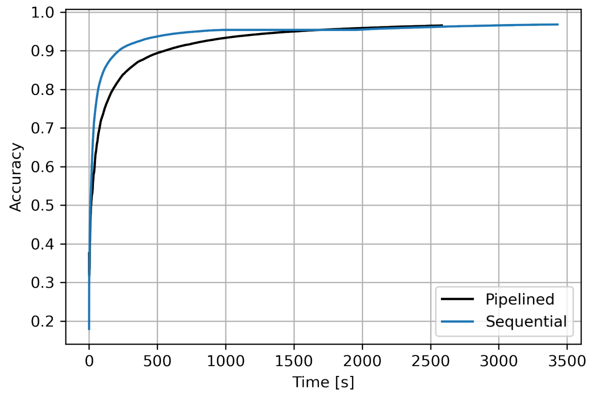


Figure 5: 2-stage pipelined training result.

Summary of a Typical Job

Table 8 presents the parameters of a typical job submitted in the development of our project. In general, two nodes were requested with 32 GB of memory per node. 32 GB of memory was necessary in order for there to be enough space available to store the entire dataset in memory, in addition to stashing the weights of previous iterations for future use by the optimizer. Weight stashing significantly increased the amount of memory needed for a job. As shown in the table, the typical wall time was about three-quarters of an hour, and we used Pandas to read and write CSV files because it can easily handle pytorch datatypes.

	2-stage model
Typical wall clock time (hours)	0.75
Typical job size (nodes)	2
Memory per node (GB)	32
Maximum number of input files in a job	1
Maximum number of output files in a job	1
Library used for I/O	Pandas to handle CSV files

Table 8: Workflow parameters of the two test cases used during project development.

5 Resource Justification

The number of node hours for the two stage benchmark can be calculated as:

$$2 \text{ nodes} \times \frac{0.255}{3600\frac{s}{\text{hour}}} = 0.00014167 \text{ node hours}$$

Table 9 assumes that a dataset of the same size as DeepSat (324,000 images) is trained using the same batch size (128) for four epochs, meaning there will be 10125 iterations in total. In reality, higher resolution images might be used in the future, meaning that the complexity of our model would likely also increase. This would require us to partition layers among more nodes and train the model for more epochs to reach convergence. Therefore, the required node hours would likely increase in the future. Lastly, model might need to be retrained multiple times per year, therefore increasing the total annual node hours needed.

	2-stage model
Simulations per task	1
Iterations per simulation	10125
node hours per iteration	0.00014167
Total node hours	1.43

Table 9: Justification of the resource request

References

[1] Deepsat (sat-6) airborne dataset. https://www.kaggle.com/datasets/crawford/deepsat-sat6?select=X_test_sat6.csv. Accessed: 2022-05-03.

[2] Gloo. https://github.com/facebookincubator/gloo. Accessed: 2022-05-03.

[3] Pytorch c++ frontend. https://pytorch.org/tutorials/advanced/cpp_frontend.html#writing-the-training-loop. Accessed: 2022-05-03.

[4] I. S. Alex Krizhevsky and G. E. Hinton. Imagenet classification with deep convolutional neural networks, 2012. Accessed: 2022-05-03.

[5] S. Basu, S. Ganguly, S. Mukhopadhyay, R. DiBiano, M. Karki, and R. Nemani. Deepsat - a learning framework for satellite imagery, 2015. Accessed: 2022-05-03.

[6] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, and P. B. Gibbons. Pipedream: Fast and efficient pipeline parallel DNN training, 2018. Accessed: 2022-05-03.

[7] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014.

[8] A. Unnikrishnan, S. V, and S. K P. Deep alexnet with reduced number of trainable parameters for satellite image classification, 2018. Accessed: 2022-05-03.

[9] P. with Code. Image classification on imagenet. https://paperswithcode.com/sota/image-classification-on-imagenet. Accessed: 2022-05-03.