Performance Benchmarking of Data Augmentation and GPU Count Variability for Deep Learning Tornado Predictions

Jonathan N. Basalyga[a], Carlos A. Barajas[a], Matthias K. Gobbert[a,*], Jianwu Wang[b]

[a] *Department of Mathematics and Statistics, University of Maryland, Baltimore County, USA*
[b] *Department of Information Systems, University of Maryland, Baltimore County, USA*

Abstract

Predicting violent storms and dangerous weather conditions with current models can take a long time due to the immense complexity associated with weather simulation. Machine learning has the potential to classify tornadic weather patterns much more rapidly, thus allowing for more timely alerts to the public. To deal with class imbalance challenges in machine learning, different data augmentation approaches have been proposed. In this work, we examine the wall time difference between live data augmentation methods versus the use of preaugmented data when they are used in a convolutional neural network based training for tornado prediction. We also compare CPU and GPU based training over varying sizes of augmented data sets. Additionally we examine what impact varying the number of GPUs used for training will produce given a convolutional neural network on wall time and accuracy. We conclude that using multiple GPUs to train a single network has no signficant advantage over using a single GPU. The number of GPUs used during training should be kept as small as possible for maximum search throughput as the native keras multi-GPU model provides little speedup with optimal learning parameters.

Keywords: deep learning, data augmentation, parallel performance,

*Corresponding author
Email address: `gobbert@umbc.edu` (Matthias K. Gobbert)

TensorFlow, Keras, GPUs

1. Introduction

Forecasting storm conditions using traditional, physics based weather models can pose difficulties in simulating particularly complicated phenomena. These models can be inaccurate due to necessary simplifications in physics or the presence of some uncertainty. These physically based models can also be computationally demanding and time consuming. In the cases where the use of accurate physics may be too slow or incomplete using machine learning to categorize atmospheric conditions can be beneficial [1]. Machine learning has been used to accurately forecast rain type [1, 2], clouds [2], hail [3], and to perform quality control to remove non-meteorological echos from radar signatures [4].

A forecaster must use care when using binary classifications of severe weather such as those which are provided in this paper. The case of a false alarm warning can be harmful to public perception of severe weather threats and has unnecessary costs. On the one hand, an increased false alarm rate will reduce the public's trust in the warning system [5]. On the other hand, a lack of warning in a severe weather situation can cause severe injury or death to members of the public. Minimizing both false alarms and missed alarms are key in weather forecasting and public warning systems.

With advances in deep learning technologies, it is possible to accurately and quickly determine whether or not application data is of a possibly severe weather condition like a tornado. Specifically one can use an supervised neural network such as a convolutional neural network (CNN) for these binary classification scenarios. However these CNNs must be heavily tuned and hardened to prevent false positives, or worse, false negatives from being produced. These CNNs require large amounts, hundreds of thousands and even millions, of data samples to learn from. Without an ample amount of data to learn from a CNN has no hope of achieving accurate predictions on anything except the original training data provided. Of the 183,723 storms in the data set used in this work only around 9,000 entries have conditions which lead to tornadic behavior in the future [6]. This imbalance of tornado versus no tornado results in a situation where a machine is very good at predicting no potential tornado but is very bad at predicting when there is a tornado imminent leading to false negatives.

It is for these reasons that there is a real motivation to acquire more data that would result in tornadic conditions however one cannot simply go outside hoping to collect storm data that result in these conditions. This heralds the need of synthetic data to bolster the amount of data used for training a neural network. Synthetic data must be generated such that it is indistinguishable from real data and can be used in conjunction with the natural data to train a neural network on a more balanced data set which produces fewer if any false negatives. To train and tune a neural network of this nature is very time consuming and resource intensive taking anywhere from several hours to several days given enough data. In order to quickly tune, train, and test the validity of a neural network with several different hyperparameter combinations, a variation of the parallel framework originally introduced in [7] to train many networks simultaneously with varying hyperparameter values in a high performance computing environment is used.

In addition to training multiple networks simultaneously, each on a different GPU, it is also possible to train a single network across multiple GPUs. We therefore explore the question of whether it is more efficient to train several networks simultaneously, or to parallelize the training of a single network. In addition we examine the effect that using multiple GPUs during training has on the network's accuracy. This discussion of accuracy constitutes an extension of the original special session paper [8].

This paper has several contributions. (1) Benchmarking of two data augmentation approaches and their effects to deep learning training times. Through the benchmarking, we examine their differences in terms of the effective use of resources. (2) Benchmarking of MPI-based parallel deep learning hyperparameter tuning. This is done with a custom framework that allows for in-depth examination of all possible hyperparameter configurations in an HPC environment.

(3) Benchmarking of CPU and GPU based parallel deep learning hyperparameter tuning. (4) Lastly, investigation of the effect of multiple GPUs on accuracy.

The remainder of this paper is organized as follows. Section 3 gives a basic introduction to convolution neural networks and the problem of data augmentation. Section 4 introduces the natural data used for training the neural networks and the preprocessing method used on the data prior to training. Section 5 discusses hyperparameters and their importance in training and the parallel framework used for hyperparameter tuning in a high performance computing environment. Section 6 presents the effect of various hyperparameter configurations on the wall time for training as well as on accuracy of the training. Lastly Section 7 collects the conclusions of this work.

2. Related Work

There are a plethora of papers and textbooks on deep learning and neural networks that go over methods for solving data imbalances. These texts, such as [9], [10], and [11] all talk about the importance of data augmentation to prevent bias, overfitting of the network, and more. Pundits and blogs may talk about the use of live augmentations as a cure all to an imbalanced data set because tools are readily available to do this task however there is little consideration for the possible performance benefits of using data that has been augmented apriori to run time. This work seeks to demonstrate that there is a clear difference in training time with regards to preaugmented data and live augmented data even in the case of an idle CPU during GPU training sessions rather than discuss the benefits of augmentation versus not.

There are several tools that exist for hyperparameter searching yet they do not solve all of the problems presented for tuning in our HPC environment or do not solve them adequately enough. Two mainstream frameworks are Talos and sklearn's GridCVSearch. Talos aims to the fix the clunky interface of sklearn by replacing the Keras fit method with a method that takes dictionary inputs and automatically searches over them during fitting. However both these frameworks are limited to a single node and as such would not automatically fully utilize a HPC system if given the resources to do so. The framework mentioned Section 5.2, from [7, 6], exists to solve that problem by creating an HPC based framework for hyperparamter searching. This framework has innate limitations like a lack of in-depth analytics on a hyperparameter by hyperparameter basis, lacks support for live data augmentation, and only has one type of parallel schema available. This work creates a parallel framework which solves all of the aforementioned problems.

There are a slew of technical reports and papers that talk about the importance of benchmarking and improving parallel timings such as [12], [13], and [14]. Texts which deal specifically with training neural networks even go so far as to mandate GPUs for training like in [9]. In the case where one may have access to many mid to high end GPUs, or may be considering a purchase of them, how many is too many? This work aims cover, in a high level manner, how use case is an important factor for the number of GPUs that should be used for optimal training times.

3. Deep Learning with Convolutional Neural Networks

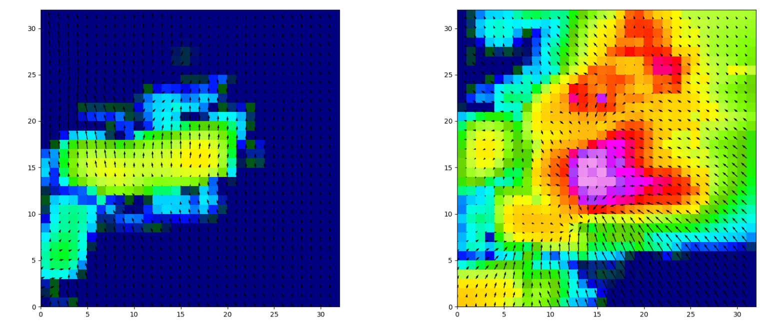
The general idea and information behind neural networks is that when given a set of inputs and known outputs we train a neural network to make predictions about future data inputs whose output is unknown. In order to gauge how accurate the network has become we provide data that was not in the learning data set and the CNN uses the knowledge gained from training to guess the outcome of data that it has not seen before [9]. We test against a testing set of data where our outputs are still known but the answers are not provided to the network. We then grade its accuracy based on the correctness of these predictions. A general neural network is made of three phases as seen in [10]. There is the input layer where the data is pushed into the network. Then there are some number of hidden layers which are responsible for digesting the input data and learning from it. Then finally the output layer whose output meaning is predetermined by the context of the problem. For example the output can be a binary classification of the input data, maybe even a new image entirely, but whatever output is produced, the network itself has no understanding of what the output truly means. In the context of tornado prediction consider a 32×32 grid of data points where each data point contains the composite reflectivity, 10 meter west-east wind component, and the 10 meter south-north wind component as the data used to predict future conditions. Then the mean future vertical wind velocity will serve as the indicator that a tornado will occur [7, 6]. A single input to the neural network would be a $32 \times 32 \times 3$ array with each variable in its own grid. This data would then be evaluated by the first hidden layer whose result would be pushed into the second hidden layer, and so on until the final result is put into the output layer. The output layer would contain an integer, specifically 0 or 1 in this case. A binary classifier in the context of mean future vertical wind velocity might seen nonsensical with regards to the question: what is the mean future vertical wind velocity given these input conditions? However the network is not attempting to, nor is it capable of, answering that question. With this binary classification the network provides an answer to: is the mean future vertical wind speed large enough to be considered tornadic? With regards to this question the network sensibly outputs either 0 for no or 1 for yes. These three weather conditions from a storm snapshot can be made into images as seen in Figure 1 which predicts if the winds result in a future tornado. With the lack of natural data available researchers must turn to synthetic data.

There are several methods to acquire synthetic data for fitting a CNN. The current method, outside of machine learning, is through storm simulation models. These are very computationally expensive often taking days for only a few hours of simulated data. On top of that there are variations between each of the models used to simulate these storms each with their own meaningful results and possible drawbacks. The computational expensive of these models and the time taken to generate the synthetic data is what gives machine learning an edge. If a storm can be predicted without the need for simulations, because the neural network takes raw satellite data and quickly produces a prediction, then solving the data imbalance for the initial training gives CNN's a clear advantage. Similarly, if we can train the CNN using quickly generated synthetic data we can forgo the need for these expensive simulations alltogether in the prediction process.

An alternative to simulated data would be using primitive duplication methods like data reflection and data rotation which can be used to fill out an existing data set rather than generating strictly new data. If the conditions present on the data grid can cause a tornado then simply reflecting the data grid over an axis results in a technically different storm that also results in a tornado. When only five percent of the data is storms that result in a tornado you would need to augment every entry in 19 unique ways to balance the data set to a perfect fifty-fifty balance of tornadic versus not tornadic.

4. Data

The data set used in this analysis was obtained from the Machine Learning in Python for Environmental Science Problems AMS Short Course, provided by David John Gagne from the National Center for Atmospheric Research [15]. Each file contains the reflectivity, 10 meter U and V components of the wind field, 2 meter temperature, and the maximum relative vorticity for a storm patch, as well as several other variables. These files are in the form of $32 \times 32 \times 3$ images describing the storm. We treat the underlying data as an image and push it through the CNN as if it were a normal RGB image. This allows our findings to generalize to other non-specialized CNNs. Figure 1 shows two examples image from one of these files. Storms are defined as having simulated radar reflectivity of 40 dBZ or greater as seen in Figure 1 (b). Reflectivity, in combination with the wind field, can be used to estimate the probability of specific low-level vorticity speeds. In the case of Figure 1 (a), the reflectivity and wind field were not sufficient enough to cause future low-level vorticity speeds. The dataset contains nearly 80,000 convective storm centroids across the central



(a) (b)

Figure 1: Sample images of radar reflectivity and wind field for a storm which (a) does not and (b) does produce future tornadic conditions.

United States.

We preprocessed the original NCAR storm data containing 183,723 distinct storms, each of which consists of $32 \times 32 \times 3$ grid points, and extracted composite reflectivity, 10 m west-east wind component in meters per second, and 10 m south-north wind component in meters per second at each grid point giving approximately 2 GB worth of data. We use the future vertical velocity as the output of the network. This gives us 3 layers of data per storm entry producing a total data size of $183{,}723 \times 32 \times 32 \times 3$ floats to feed into the neural network. We use 138,963 storms for training the model and 44,760 storms for testing the accuracy of the model. We track the total wall time for training and testing over both image sets.

5. Parallelism of Hyperparameter Tuning

5.1. Hyperparameters

As the popularity and depth of deep networks continues to grow, efficiency in tuning hyperparameters, which can increase total training time by many orders of magnitude, is also of great interest. Efficient parallelism of such tasks can produce increased accuracy, significant training time reduction and possible minimization of computational cost by cutting unneeded training.

We define hyperparameters as anything that can be set before model training begins. Such examples include, but are not limited to, number of epochs, number and size of layers, types of layers, types and degree of data augmentation, batch size, learning rates, optimizer functions, and metrics. The weights that are assigned to each node within a network would be considered a parameter, as opposed to a hyperparameter, since they are only learned through training. With so many hyperparameters to vary, and the near infinite amount of combinations and iterations of choices, hyperparameter tuning can be a daunting task. Many choices can be narrowed down by utilizing known working frameworks and model structures, however, there is still a very large area to explore even within known frameworks. This is compounded by the uniqueness of each dataset and the lack of a one-size-fits all framework that is inherent with machine learning.

Section 5.2 talks about the new MPI based framework which used the Dask framework in [7] as a baseline conceptually but many aspects, including how analytics are handled, have been improved or redesigned entirely.

5.2. MPI Framework for Parallelized Training

The Dask framework for hyperparamter tuning in an HPC environment from [7, 6] was used as a baseline for the new framework. We replace Dask with MPI by using the latest `mpi4py`. Dask had predetermined configurations for a SLURM based master-worker setup. With MPI we created two parallelism setups. The first is a typical master-worker configuration. The master-worker system allows one master process to distribute a specific combination of hyperparameters to each process. This allows for the most optimal load balancing

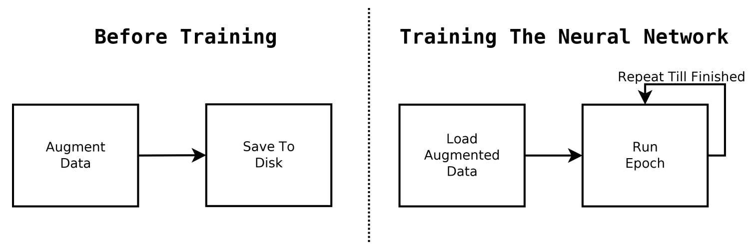


Figure 2: The preaugmented data is saved to disk before training begins. It is then loaded from disk to be used during training.

scheme at the cost of using one node for book keeping. The master node distributes a hyperparameter configuration to a worker node, waits for the work to finish, then collects all timing results and other metrics from the worker node and saves the results into a collection of JSON files.

The second parallelism configuration is the fully sychronized setup. We created a custom combination generator that takes in a dictionary full of all possible hyperparameters values and a process id and returns a dictionary that contains a specific combinations of hyperparameters. At a higher level this generator allows all combinations of hyperparameters to be indexed without actually being generated until they are needed by the workers. This generator also attempts to balance the loads by distributing the more theoretically intensive jobs evenly among all processes such that each process gets heavy and light work periodically throughout the training process.

By replacing Dask with these systems we have enabled a method which allows us to measure the effects of every single hyperparameter combination rather than just viewing things grouped by batch size. We now have the ability to group by any arbitrary hyperparameter and examine how each one plays a role in the training time and accuracy of the model. We also changed the base CNN used for testing to use multiple GPUs by using Keras' `multi_gpu_model` wrapper. TensorFlow will always allocate memory on all GPUs but may not bother to use the any additional GPUs provided. By using `multi_gpu_model` Keras duplicates the network on every GPU and trains each network with mini-batches of the original batch and then computes new weights based on the each of the mini-batches. In this way Keras does all high level management for multiple GPUs rather than TensorFlow.

6. Results

We use the framework detailed in Section 5.2 to investigate the effect of hyperparameters on *wall time*; to reflect that these are tests, relatively small numbers of epochs are used. Sections 6.1.1 and 6.1.2 take a close look at how each hyperparameter impacts training time of the neural network using both preaugmented data and live data augmentation, respectively. Then with the same framework we examine how varying the number of GPUs impacts wall time performance in Section 6.1.3, the central idea being that this helps determine an optimal hardware configuration for future training of similar networks with an immense data size. All forms of augmentation are done using Keras' datagen API with identical inputs. Any differences in accuracy are an artifact of seeding or data shuffling during training. With this in mind we present only wall times as a demonstration of how some hyperparamters can have a meaningful impact on wall time and thus should be tuned carefully, perhaps even last, to prevent cumbersome training times.

Extending the results presented originally in [8], the additional Section 6.2 investigates how batch size and GPU count affect *accuracy*; to ensure the networks are fully trained as well as to reflect real world usage patterns, in this section we use a much greater number of epochs than are used in the previous sections.

The numerical studies in this work use a distributed-memory cluster of compute nodes with large memory and connected by a high-performance InfiniBand network. The CPU nodes feature two multi-core CPUs, while the 2018 GPU node has four GPUs. The following specifies the details:

- **2018 CPU nodes**: 42 compute nodes, each with two 18-core Intel Xeon Gold 6140 Skylake CPUs (2.3 GHz clock speed, 24.75 MB L3 cache, 6 memory channels). Each node has 384 GB of memory (12×32 GB DDR4 at 2666 MT/s). The nodes are connected by a network of four 36-port EDR (Enhanced Data Rate) InfiniBand switches (100 Gb/s bandwidth, 90 ns latency).

- **2018 GPU node**: 1 GPU node containing four NVIDIA Tesla V100 GPUs connected by NVLink and two 18-core Intel Skylake CPUs. The node has 384 GB of memory (12×32 GB DDR4 at 2666 MT/s).

6.1. The Effect of Data Augmentation on Wall Time

6.1.1. Preaugmented Data

Each network was trained using a single node's total resources with the framework mentioned in Section 5.2 regardless of whether CPUs or GPUs were used during training. This section contains the wall time results for training all neural networks using data which has been preaugmented before training with primitive methods and saved to disk. This means that the network will not perform any live augmentation but rather read in the preaugmented data directly from disk. By timing in this way all the computational time will be tied directly to moving data and training the network. This is sketched in Figure 2. Additionally the words "data multiplier" refers to data that has been augmented enough that the total size of the data has increased multiplicatively by the multiplier. A data multiplier of 2 means that data has been augmented to be twice as large in size.

The results in Table 1 are made of up of the total times to train networks with various hyperparameter configurations using the 2018 CPU hardware. The timing in the upper left corner of the first subtable is the time taken to train a network on preaugmented data which has the same number of total records as the original nonaugmented data using a batch size of 128, 5 epochs, and a learning rate of 0.001. Similarly the bottom right entry of that same subtable is the time taken to train a network on preaugmented data which has four times as many entries as the original unaugmented dataset using a batch size 4096, 5 epochs, and a learning rate of 0.001.

The first subtable in Table 1 used 5 epochs and a learning rate of 0.001 for training all subconfigurations within the table. The first column of this subtable uses as many records as the original dataset but each network in the column used a different batch size for training. As the batch size increases the time taken to train the network decreases. However the time saved after each increase in batch size does not scale proportionally with the change in batch size. Now consider only the first row of the first subtable. All networks trained in this row use the same number of epochs, the same learning rate, and the same batch size of 128 but the total number of records increase multiplicatively with the column's associated multiplier. The first entry in the row uses the same number of entries as the original dataset but the second entry in that row uses twice as many entries and the last row uses four times as many entries. As the number of total entries used doubles the timings grow proportionally larger. With two times the amount of data used to train the network the network takes twice as long to train. Similarly using four times as much data results in the time taken to train being four times larger than the first entry in the row. The more data used the longer it takes to train. These changes in timings hold for all subtables in Table 1.

Examine the upper right entry in each of the subtables. Each of these entries were trained using the same learning rate, batch size, and dataset but with a varying number of epochs. The first subtable uses the least number of epochs and also has the fastest time among the three. The second subtable uses double the number of epochs as the first and also takes twice as long to train. Similarly the third subtable takes three times as long to train and uses three times as many epochs as the first subtable. An increase in the number of epochs means the data is passed that many more times to the network for training. It is sensible then that the time taken to train would increase linearly with the number of epochs used so long as all other hyperparamters are the same.

Table 2 contains the times taken to train networks with various hyperparameter configurations using the 2018 GPU hardware. All timing results draw the same conclusions as Table 1 except all timings for the GPUs are 10× faster

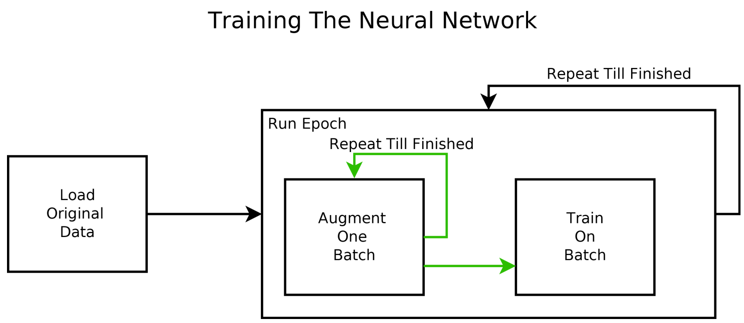


Figure 3: The original data is first loaded from disk. When an epoch starts the one batch of data is augmented and trained on. While the network trains on that batch another is augmented in parallel as indicated by the green arrow.

and in some instances even 12× faster. This massive increase in speedup is expected by researchers in the machine learning community and is a common theme seen when comparing CPU based training versus GPU based training. The process of training a convolutional neural network such as the one discussed in Section 1 uses many complex matrix operations in the process of computing weights for the hidden layers of the network. GPUs are specifically designed to do matrix operations of many flavors and it is accepted fact that they do these operations much faster than CPUs. Sensibly then, these specialized accelerators perform the training process considerably faster than a CPU. In the case of the 2018 GPUs there are four GPUs training the neural network at any one time as opposed to the two CPUs used to train the neural networks in the CPU tables.

Since there is no data augmentation happening during training, all the times listed are pure training times. The timings for the CPUs improve dramatically as the batch size is increased regardless of the number of epochs. The GPUs are so effective with regards to training that batch size plays a smaller role in the training time. GPUs are, in all regards, faster than CPUs for training.

6.1.2. Live Augmentation

This section contains the results that use live data augmentation during training. The original natural data is loaded, but while training the data is

Table 1: Wall time for batch size versus data multiplier grouped by epochs with learning rate 0.001 for the 2018 CPUs with preaugmented data in seconds.

5 Epochs	Data Multiplier		
Batch Size	1	2	4
128	195	369	737
256	124	253	484
512	95	194	384
1024	77	159	310
2048	64	125	251
4096	56	107	211
10 Epochs	Data Multiplier		
Batch Size	1	2	4
128	373	720	1494
256	238	486	962
512	189	382	763
1024	154	313	629
2048	123	240	506
4096	110	210	422
15 Epochs	Data Multiplier		
Batch Size	1	2	4
128	574	1120	2239
256	367	740	1408
512	284	558	1140
1024	233	468	929
2048	184	370	730
4096	158	308	649

Table 2: Wall time for batch size versus data multiplier grouped by epochs with learning rate 0.001 for the 2018 GPUs with preaugmented data in seconds.

5 Epochs	Data Multiplier		
Batch Size	1	2	4
128	20	36	72
256	12	24	47
512	11	18	38
1024	10	17	32
2048	10	16	30
4096	13	18	37
10 Epochs	Data Multiplier		
Batch Size	1	2	4
128	36	74	146
256	24	48	96
512	22	36	77
1024	19	32	62
2048	17	30	58
4096	20	36	67
15 Epochs	Data Multiplier		
Batch Size	1	2	4
128	56	110	223
256	37	72	144
512	32	55	109
1024	25	48	99
2048	25	48	88
4096	32	56	98

pushed through the primitive augmentation methods provided by Keras. The training times that are seen represent the wall time taken to move data, augment the data on-the-fly, and train the network. A high level view of this process can be seen in Figure 3. Keras' primitive augmentation supports parallel augmentation meaning that data is being augmented in parallel to the networks being trained. This parallel operation can be seen as the green arrows in Figure 3.

Live augmentation is typically done so that one does not need to preaugment gigabytes or even terabytes of unbalanced data. In some cases, you may even do live augmentation to turn small amounts of balanced or unbalanced data into larger amounts of balanced data so while the original dataset may fit into memory the larger augmented dataset might not. If your data is too large to fit into memory then preaugmented data would be I/O bound as it is read from disk rather than being CPU bound by being augmented on-the-fly.

Table 3 shows similar timing behaviors to Table 1 when examining how the data multiplier scales the timing results but a much stronger diminishing return when batch size is increased. In order to do live data augmentation Keras starts as many processes as there are cores on a node. The processes rotate, scale, and so on in parallel and send the data back to the main process. These processes are then cleaned up by the operating system forcing the main process to block during this time. This becomes a clear bottleneck as we can see that the timings for smaller batch sizes are much worse than the larger batch sizes. However the times approach the preaugmented timings as the overheard of process creation becomes a smaller player in the time it takes to augment the data. The less data that can be live augmented the less time the spawned processes work meaning they spend more time being created and cleaned up than they do actually generating new data.

The overhead is even more aparent when examining Table 4 compared to Table 2. The scaling in each individual row has the same behavior but all of the rows in Table 4 are much slower than expected. Subtable 3 is 2× to 3× slower than the preaugmented numbers in the same positions. This is clearly due to the CPU bounded operations that are inherent with live data augmentation.

Additionally if you examine the data mulitplier 4 column of subtable 3 the time savings as batch size increase disappears and makes way for varying wall times that are completely unrelated to the increase in batch size. Any savings that would normally be obtained from increasing batch size are lost due to the overhead of live augmentation.

The timings for primitive live augmentation methods using CPUs and GPUs are anywhere from a few minutes to a couple hours. The GPU training is so efficient the GPU spends most of its time waiting for the data to be augmented rather than training. In cases where you are doing CPU based training the processor is working hard to both train and augment the data in tandem and often does not have the spare resources to balance both tasks.

6.1.3. The Effect of GPU Count on Wall Time

This section contains the wall time results for varying the number of GPUs while training. The number of GPUs used during training can be treated as a hyperparameter, as it has an impact on both training time and prediction accuracy." If the impact of using more GPUs is negligible then all future hyperparamter sweeps should use the lowest number of GPUs possible. If luck would have it that the optimal number of GPUs can be evenly divided amongst the MPI processes during training, then result would be great boon for efficient training in the future. We use Keras' `mult_gpu_model` which will automatically force TensorFlow to use all available GPUs by duplicating the graph on each GPU and training each of these with mini-batches in a process we refer to as "forced" parallelism. Additionally it has already been show in Section 6.1.2 that live augmentation is far slower than preaugmented data thus for this section we only use preaugmented data to cut down the wall time as much as possible.

Table 5 contains the wall times for the numbers of GPUs versus data multiplier grouped by epochs on the 2018 GPUs with preaugmented data, forced parallelism, and a batch size of 32768. Consider the first row of 5 epoch table. For one GPU as the data multiplier increases the wall time increases proportionally. Now consider the data multiplier 1 column of the 5 epoch table. As

Table 3: Wall time for batch size versus data multiplier grouped by epochs with learning rate 0.001 for the 2018 CPUs with live augmented data in seconds.

5 Epochs	Data Multiplier		
Batch Size	1	2	4
128	2534	5052	9859
256	1324	2597	5174
512	723	1445	2897
1024	390	776	1527
2048	210	425	852
4096	154	302	527
10 Epochs	Data Multiplier		
Batch Size	1	2	4
128	5066	10122	19627
256	2626	5271	10322
512	1376	2766	5520
1024	762	1501	3026
2048	429	847	1735
4096	305	620	1636
15 Epochs	Data Multiplier		
Batch Size	1	2	4
128	7369	14779	30372
256	3893	7950	15476
512	2083	4161	8304
1024	1155	2327	4511
2048	631	1278	2555
4096	388	798	1689

Table 4: Wall time for batch size versus data multiplier grouped by epochs with learning rate 0.001 for the 2018 GPUs with live augmented data in seconds.

5 Epochs	Data Multiplier		
Batch Size	1	2	4
128	37	70	142
256	35	69	138
512	36	72	140
1024	37	72	142
2048	38	76	150
4096	44	83	163
10 Epochs	Data Multiplier		
Batch Size	1	2	4
128	73	146	285
256	71	143	286
512	69	141	278
1024	73	144	284
2048	77	150	295
4096	83	161	329
15 Epochs	Data Multiplier		
Batch Size	1	2	4
128	108	214	442
256	105	211	429
512	107	216	426
1024	109	217	432
2048	117	229	445
4096	126	245	502

the number of GPUs increases the time remains nearly identical despite the doubling, tripling, and quadrupling of the compute power being used during training. Even considering the entire 5 epoch subtable yields the same behavior: as the number of GPUs increase the wall time remains qualitatively the same. All other subtables exhibit the same behavior as the 5 epoch subtable. While the increase in epochs causes a general increase in the subtable timings, changing the number of GPUs does nothing to improve these timings. Conceptually the batch size of the table is 1/5 of all data with regards to a multiplier of 1. Multiple GPUs should have a real edge over a single GPU yet there this is not demonstrated. This is to say that the number of GPUs does nothing to improve wall time despite differences in data size.

Table 6 contains the wall times for the number of GPUs versus epochs grouped by data multiplier with preaugmented data, forced parallelism, and a batch size of 128. Consider the first row of the first subtable. For one GPU with a data multiplier of 1 and a varying number of epochs as the number of epochs increases the wall time increases proportionally. This proportional increase holds for all rows of the subtable and similarly this table wide behavior holds for the data multiplier 2 and 4 subtables. Examine the first column of the last subtable which is the 5 epoch column of data multiplier 4 table with a varying number of GPUs. As the number of GPUs increases the time also increases though the increase in time is steepest from one GPU to two GPUs. From there the time increase is 10 seconds per GPU additional GPU. As the number of epochs increases from 5 to 10 the increase from one GPU to two GPUs triples from around 20 seconds to approximately 60 seconds. Every additional GPU increases time by 20 seconds per GPU. As the number of epochs increases from 5 to 15 the increase from one GPU to two GPUs goes from around 20 seconds to approximately 90 seconds. Every additional GPU is around 30 seconds per GPU. At the smallest batch size the more GPUs used the slower the training time.

When even larger cases are run in isolation, this behavior is more easily observed with the tool `nvidia-smi`. With just one GPU and a batch size of

32,768 the GPU is entirely saturated for the majority of run-time with only occasional drops in GPU usage when the training rolls over to the next epoch. Similarly submitting a 4 GPU job with a batch size of 131,072, meaning each GPU gets as much data as the multiplier 1 case, results in maximum saturation as well. This is why timings at much larger batch sizes seem much closer in time as the GPUs spend around the same amount of time computing and idling. This would give the impression that it takes Keras more time to distribute the data to the GPUs than compute and finalize all other information associated with computation.

Table 5: Wall time for GPUs versus data multiplier grouped by epochs with batch size 32768, learning rate 0.001 for the 2018 GPUs with preaugmented data and forced parallelism in seconds.

5 Epochs	Data Multiplier		
GPUs	1	2	4
1	11	18	34
2	11	18	33
3	11	18	33
4	11	18	33
10 Epochs	**Data Multiplier**		
GPUs	1	2	4
1	17	29	58
2	16	30	59
3	16	30	57
4	18	31	60
15 Epochs	**Data Multiplier**		
GPUs	1	2	4
1	25	44	92
2	23	44	88
3	23	45	84
4	26	45	88

Table 6: Wall time for GPUs versus epochs grouped by data multiplier with batch size 128, learning rate 0.001 for the 2018 GPUs with preaugmented data and forced parallelism in seconds.

1 Data Multiplier	Epochs		
GPUs	5	10	15
1	20	38	61
2	27	51	77
3	31	55	83
4	41	61	92
2 Data Multiplier	**Epochs**		
GPUs	5	10	15
1	42	76	114
2	53	103	154
3	59	112	168
4	64	123	182
4 Data Multiplier	**Epochs**		
GPUs	5	10	15
1	85	157	229
2	106	215	311
3	116	231	340
4	125	247	368

6.2. The Effect of Batch Size and GPU Count on Accuracy

In this section we present accuracy results for varying batch size and the number of GPUs used during training. In order to ensure the network is fully trained, greater numbers of epochs are used (up to 1000) than in the previous sections. The data multiplier is kept to 1 so as not to artificially inflate run time.

Figure 4 shows training accuracy curves varied by number of GPUs for batch sizes 128, 4,096, and 32,768. Note that the sudden drops in accuracy (especially prominent in the batch size 4,096 plot) result from the use of dropout layers. In the batch size 128 plot accuracy plateaus after only a small number of epochs and the curves for each GPU count lie on top of each other, virtually indistinguishable. As batch size increases a tendency emerges for higher GPU counts to have a slightly higher accuracy for any given number of epochs. With a batch size of 32,768, throughout most of the time spent training the 4 GPU curve has an accuracy about 1% higher than the 1 GPU curve with the same batch size.

The training accuracy curves resulting from keeping GPU count fixed and varying batch size are shown in Figure 5. The 2 GPU plot on the left and the 4 GPU plot on the right are virtually identical, as would be expected from the results in Figure 4. For any fixed number of epochs increasing batch size decreases accuracy. Even after 1,000 epochs there is an approximately 10% difference in accuracy between the batch size 128 curve and the batch size 65,536 curve.

When using Keras' `mult_gpu_model` a copy of the network is sent to each GPU. For every batch, each copy of the network is trained on a smaller subset of the original batch, then the resulting weights are aggregated together and copied back to each network. This ensures that after every batch each copy of the network is identical, even though they have all been trained on different subsets of the original batch. The size of these subsets is equal to the total batch size divided by the number of GPUs used. Therefore, when comparing the training of two different networks, one might expect that when the respective batch sizes divided by the respective GPU counts equal some constant, their training curves will be more or less the same. Figure 6 does exactly this, varying both GPUs and batch size at the same time so that the batch size divided by GPU count is constant. We see that in fact the training curves are not the same. The effect of a smaller batch size outweighs the effect of a lower GPU count, and vice versa.

Table 7 contains the testing accuracies of the network, organized by batch size versus epochs, with 2 GPUs, data multiplier 1, learning rate 0.001, preaugmented data, and forced parallelism. We provide only the 2 GPU table since it allows us to provide data for the batch size 65,536 runs, and since other GPU counts result in similar accuracies. By considering just a single row of this table we see that the testing accuracies follow a similar trend to what is exhibited by the training accuracies in Figure 5, that is, accuracy decreases as batch size increases. Therefore, when using a larger batch size a network must be trained for a greater number of epochs to reach a similar accuracy as that reached by a network trained using a smaller batch size. By examining individual columns we see that testing accuracy plateaus between 92% and 93%. This would indicate that the network configuration which can reach a testing accuracy of around 93% in the least amount of time would be the optimal configuration.

The corresponding timings of each run of the network are presented in Table 8. Here we see that total training time increases linearly with epochs, but increases non-linearly with batch size. The speedup of training time decreases with each doubling of batch size until the speedup is negligible. We see that in the case of our test network that this point of negligible speedup is reached by a batch size of 4,096. This is in contrast to the effect that batch size has on accuracy, since it can be seen in Table 7 that accuracy continues to decrease across an entire row. As a result of these effects, neither maximizing nor minimzing any of these hyperparameters leads to optimal performance. This behavior can be observed when examining Table 8. The 256 batch size 100 epoch run and the 1024 batch size 200 epoch run both have an accuracy of approximately 93% and a run time of approximately 5 minutes. However the 128 batch size 100 epoch run has comparable accuracy but is double the run time at approximately 10 minutes. Additionally the 4096 batch size 400 epoch run has a 10 minute run

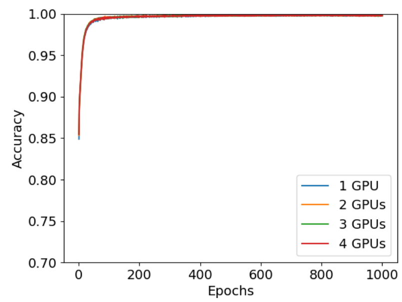
Table 7: Training accuracies for batch size versus epochs with 2 GPUs, data multiplier 1, learning rate 0.001 for the 2018 GPUs with preaugmented data and forced parallelism.

2 GPUs	Batch Size									
Epochs	128	256	512	1024	2048	4096	8192	16384	32768	65536
5	87.96	90.26	82.55	85.23	87.86	89.28	83.00	79.98	76.21	68.57
10	92.02	86.88	87.37	88.21	89.18	88.95	89.00	84.44	83.55	77.65
15	91.41	88.41	89.98	91.11	88.79	87.60	85.75	86.20	85.98	76.67
100	93.03	93.15	91.90	90.02	88.68	87.09	88.68	88.54	89.23	88.09
200	93.45	93.26	93.14	93.57	92.39	89.91	87.14	88.49	87.48	86.60
300	93.50	93.77	93.41	93.21	92.27	92.53	89.66	87.63	89.03	86.80
400	92.19	93.43	93.52	93.08	92.63	92.90	90.88	88.30	88.53	87.12
500	91.12	93.40	93.49	93.14	93.11	92.27	92.90	90.87	90.37	88.32
600	93.27	92.94	93.23	93.27	92.86	92.49	91.71	90.95	88.81	88.95
700	93.29	93.48	93.62	93.08	92.98	92.77	92.28	90.11	87.71	88.18
800	92.58	93.32	93.41	93.26	93.23	92.81	92.33	90.93	90.97	89.84
900	93.96	93.38	93.24	93.11	89.23	93.36	92.70	89.98	87.06	90.34
1000	92.12	92.98	93.23	93.48	92.92	93.34	92.37	91.29	89.05	87.21

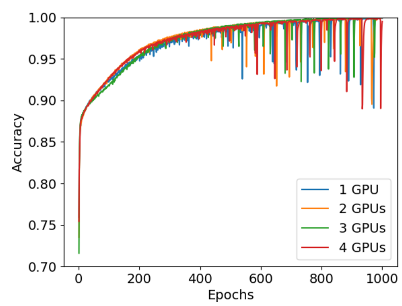
time for the same comparable accuracy.

Table 8: Timing for batch size versus epochs with 2 GPUs, data multiplier 1, learning rate 0.001 for the 2018 GPUs with preaugmented data and forced parallelism.

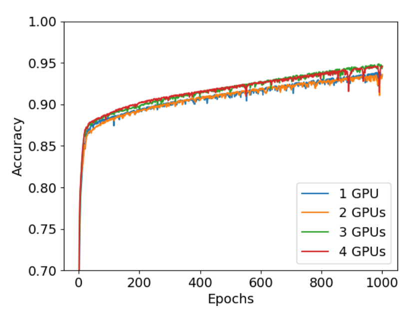
2 GPUs	Batch Size									
Epochs	128	256	512	1024	2048	4096	8192	16384	32768	65536
5	00:31	00:17	00:12	00:11	00:10	00:13	00:13	00:14	00:15	00:15
10	00:59	00:33	00:23	00:18	00:17	00:18	00:21	00:21	00:23	00:24
15	01:22	00:48	00:34	00:27	00:25	00:27	00:27	00:26	00:26	00:27
100	09:12	05:23	03:38	02:45	02:23	02:23	02:29	02:24	02:22	02:19
200	18:14	10:49	07:14	05:32	04:42	05:12	04:47	04:42	04:29	04:26
300	27:33	16:15	10:54	08:13	07:07	07:37	07:16	06:58	06:47	06:35
400	36:47	21:46	14:25	11:03	09:35	10:19	09:30	09:17	08:54	08:50
500	46:10	27:07	18:04	13:44	11:50	12:41	11:57	11:27	11:11	10:55
600	55:33	32:44	21:42	16:31	14:17	15:24	14:11	13:57	13:15	13:20
700	64:39	38:06	25:29	19:12	16:30	17:43	16:33	16:03	15:33	15:13
800	73:40	43:48	28:58	21:58	18:41	20:20	18:57	18:24	17:52	17:24
900	83:21	49:03	32:56	24:44	21:16	22:55	21:16	20:35	19:57	19:26
1000	92:02	54:55	36:29	27:40	23:40	25:22	23:58	22:48	22:42	21:39



(a) batch size 128

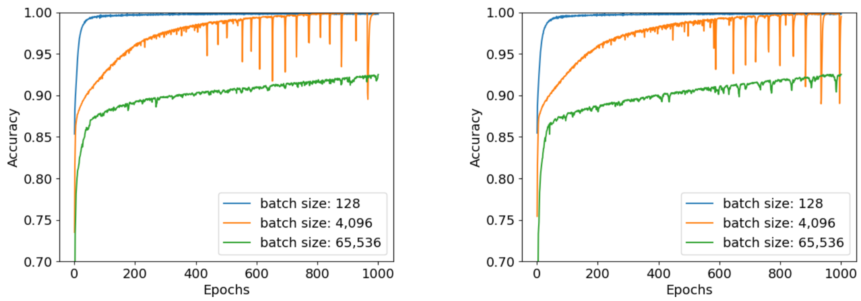


(b) batch size 4,096



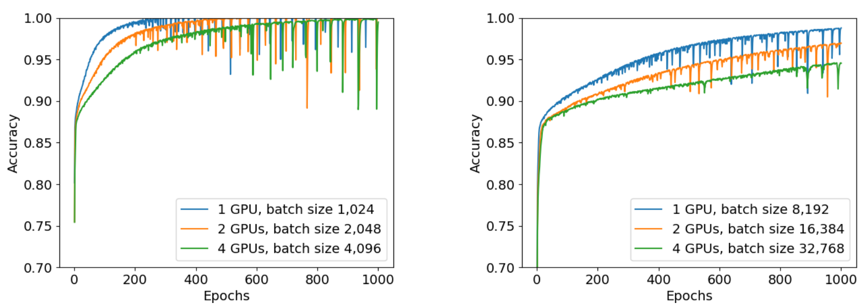
(c) batch size 32,768

Figure 4: Training accuracy curves for different batch sizes, varying GPU counts, with data multiplier 1 and learning rate 0.001 for the 2018 GPUs with preaugmented data and forced parallelism.



(a) 2 GPUs (b) 4 GPUs

Figure 5: Training accuracy curves for different GPU counts, varying batch sizes, with data multiplier 1 and learning rate 0.001 for the 2018 GPUs with preaugmented data and forced parallelism.



(a) batch size $\div$ GPU count $= 1,024$ (b) batch size $\div$ GPU count $= 8,192$

Figure 6: Training accuracy curves varied by both GPUs and batch size simultaneously, with data multiplier 1 and learning rate 0.001 for the 2018 GPUs with preaugmented data and forced parallelism.

7. Conclusions

There is not a lot of discussion on whether or not one should augment the data prior to experimentation. Careful consideration should be taken with regards to the time taken to train a network as can be seen in Section 6.1. The time difference between using preaugmented data versus the use of primitive live augmentation methods is substantial. If the disk space is available one should always opt for preaugmented data over primitive live method. This becomes especially important if one is looking to take advantage of accelerators like a GPU. The GPU training is so efficient the GPU spends most of its time waiting for the data to be augmented rather than training. In Section 6.1.1 the preaugmented data times were on the scale of minutes compared to the primitive live augmentation methods seen in Section 6.1.2 whose times were in hours. In cases where you are doing CPU based training the processor is working hard to both train and augment the data in tandem and often does not have the spare resources to balance both tasks. Preagumented data was clearly the better choice for both GPU and CPU training. Additionally, GPU training was so much faster than CPU training that even the GPUs in older CPU/GPU nodes (from 2013) were faster than the state-of-the-art CPUs from 2018 used in the studies here [6].

While the GPU training was clearly better than the CPU training, there are still more variables to tackle. The question, "do more GPUs equate to better performance time?," may seem obvious but the results in Section 6.1.3 beg to differ. Initially one might suspect that putting more computing power behind training will result in faster run times but this is not the case. At the smallest batch size, the more GPUs used, the slower the training time. The mini-batch system Keras uses does not cater toward pushing and pulling small amounts of data to the GPUs as the wall time is always worse as the number of GPUs increase for this batch size. Additionally the number of GPUs does nothing to improve wall time despite differences in data size. A single GPU still out performs all other counts of GPUs across the board. With just one GPU and a batch size of 32,768, the GPU is entirely saturated for the majority of run-time with only occasional drops in GPU usage when the training rolls over to the next epoch. Similarly submitting a 4 GPU job with a batch size of 131,072, meaning each GPU gets as much data as the multiplier 1 case, results in maximum saturation for very short bursts of a couple seconds. The original predictive model is computationally cheap to train and as such it is not unlikely that this leads to one GPU having the best performance times. Each additional GPU exhibits a near constant increase in time as it is only a small amount of overhead to micromanage additional GPUs. This is to say that training a more simple cheap network where one wants to train with as many hyperparameter combinations as possible should be done with only one high end GPU per process. With a node that has four GPUs you can train four networks per node rather than just one per node which dramatically increases throughput. For a sufficiently complex network it is still possible that multiple GPUs are more efficient as the extra computing power can be put to good use rather than left idling.

The tests in Section 6.2 show that the number of GPUs have no meaningful impact accuracy for small batch sizes. Yet when we increase batch size to be so large that the GPUs are fully saturated with full memory we see a large drop in accuracy on an epoch by epoch basis. This alludes to larger batch sizes being impractical for training unless one uses many more epochs to correct this large drop. As mentioned in Section 6.1.3 one must use larger batch sizes to see full computational saturation and huge boosts to speedup. If one were trying to see speedup while maintaining accuracy it would make sense to increase the number epochs to account for the accuracy lost due to batch size enlargement. However in many cases the speedup is completely lost by doing so. This further reinforces the arguement that the minimal number of GPUs necessary should be used in training a single network. This maximizes training throughput in regards to the number of networks trained at a time and the optimal speedup for the majority of training cases.

Acknowledgements

This work is supported by the grant "CyberTraining: DSE: Cross-Training of Researchers in Computing, Applied Mathematics and Atmospheric Sciences using Advanced Cyberinfrastructure Resources" from the National Science Foundation (grant no. OAC–1730250). Co-author Carlos Barajas additionally acknowledges support as HPCF RA. The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258, CNS–1228778, and OAC–1726023) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `hpcf.umbc.edu` for more information on HPCF and the projects using its resources.

References

[1] V. Nourani, S. Uzelaltinbulat, F. Sadikoglu, N. Behfar, Artificial intelligence based ensemble modeling for multi-station prediction of precipitation, Atmosphere 10 (2) (2019) 80–27.

[2] W. Ghada, N. Estrella, A. Menzel, Machine learning approach to classify rain type based on Thies disdrometers and cloud observations, Atmosphere 10 (251) (2019) 1–18.

[3] A. McGovern, K. L. Elmore, D. J. Gagne II, S. E. Haupt, C. D. Karstens, R. Lagerquist, T. Smith, J. K. Williams, Using artificial intelligence to improve real-time decision-making for high-impact weather, Bulletin of the American Meteorological Society 98 (10) (2017) 2073–2090.

[4] V. Lakshmanan, C. Karstens, J. Krause, K. Elmore, A. Ryzhkov, S. Berkseth, Which polarimetric variables are important for weather/no-weather discrimination?, Journal of Atmospheric and Oceanic Technology 32 (6) (2015) 1209–1223.

[5] L. R. Barnes, E. C. Gruntfest, M. H. Hayden, D. M. Schultz, C. Benight, False alarms and close calls: A conceptual model of warning accuracy, Weather and Forecasting 22 (5) (2007) 1140–1147.

[6] C. A. Barajas, *An Approach to Tuning Hyperparameters in Parallel: A Performance Study Using Climate Data*, M.S. Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County (2019).

[7] C. Becker, W. D. Mayfield, S. Y. Murphy, B. Wang, C. Barajas, M. K. Gobbert, An approach to tuning hyperparameters in parallel: A performance study using climate data, Tech. Rep. HPCF–2019–13, UMBC High Performance Computing Facility, University of Maryland, Baltimore County (2019).
URL `http://hpcf.umbc.edu`

[8] C. A. Barajas, M. K. Gobbert, J. Wang, Performance benchmarking of data augmentation and deep learning for tornado prediction, in: 2019 IEEE International Conference on Big Data (Big Data), IEEE, 2019, pp. 3607–3615.

[9] F. Chollet, Deep Learning with Python, Manning, 2018.

[10] D. Osinga, Deep Learning Cookbook, O'Reilly Media, 2018.

[11] F. H. K. dos Santos Tanaka, C. Aranha, Data augmentation using GANs, ArXiv abs/1904.09135.

[12] C. Barajas, P. Guo, L. Mukherjee, S. Hoban, J. Wang, D. Jin, A. Gangopadhyay, M. K. Gobbert, Benchmarking parallel implementations of k-means cloud type clustering from satellite data, in: C. Zheng, J. Zhan (Eds.), Benchmarking, Measuring, and Optimizing. Bench 2018, Vol. 11459 of Lecture Notes in Computer Science, Springer-Verlag, 2019, pp. 248–260.

[13] C. Barajas, M. K. Gobbert, G. C. Kroiz, B. E. Peercy, Challenges and opportunities for the simulation of calcium waves on modern multi-core

and many-core parallel computing platforms, Int. J. Numer. Meth. Biomed. Engng.doi:10.1002/cnm.3244.

[14] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, K. Keutzer, Imagenet training in minutes, in: Proceedings of the 47th International Conference on Parallel Processing, ACM, New York, NY, USA, 2018, pp. 1–10. doi:10.1145/3225058.3225069.
URL http://doi.acm.org/10.1145/3225058.3225069

[15] R. Lagerquist, D. J. Gagne II, Basic machine learning for predicting thunderstorm rotation: Python tutorial, https://github.com/djgagne/ams-ml-python-course/blob/master/module_2/ML_Short_Course_Module_2_Basic.ipynb (2019).