An Approach to Tuning Hyperparameters in Parallel: A Performance Study Using Climate Data

CyberTraining: Big Data + High-Performance Computing + Atmospheric Sciences

Charlie Becker[1], Will D. Mayfield[2], Sarah Y. Murphy[3], Bin Wang[4],
Research assistant: Carlos Barajas[5], Faculty mentor: Matthias K. Gobbert[5]

[1]Department of Geosciences, Boise State University
[2]Department of Mathematics, Oregon State University
[3]Department of Civil and Environmental Engineering, Washington State University
[4]Department of Mathematics, Hood College
[5]Department of Mathematics and Statistics, UMBC

Abstract

The ability to predict violent storms and bad weather conditions with current models can be difficult due to the immense complexity associated with weather simulation. For example when predicting a tornado caution must be used when attempting to quickly classify a weather pattern as tornadic or not tornadic. Thus one can use machine learning to quickly classify these weather patterns but great care must be taken to obtain the maximal amount of accuracy while maintaining prediction wall time. We then create a general framework for determining hyperparameters with tensorflow and keras and use it for training a convolutional neural network that specializes in classifying storms as tornadic or not tornadic based on important factors like vorticity. We demonstrate our framework's ability to determine optimal hyperparameter values for batch size, epochs, and learning rate by examining accuracy and training time with regards to a small amount of application data. In the context of training time we leverage both CPUs and GPUs and found the performance of GPUs to be vastly superior in time taken to train the various networks than CPUs.

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 incomplete understanding. These physically based models can also be computationally demanding and time consuming. For situations where the exact physics are not of importance, using machine learning to categorize atmospheric conditions can be beneficial [7]. Machine learning has been used to accurately forecast rain type [3, 7], clouds [3], hail [6], and to perform quality control to remove non-meteorological echos from radar signatures [5].

A forecaster must use care when using binary classifications of severe weather such as those we are providing in this report. 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 [1]. 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.

We look to commonly used machine learning techniques such as convolutional neural networks and deep neural networks specifically. Both can be used to accurately and quickly determine whether or not application data shows a possibly severe whether condition like a tornado. However these networks must be heavily tuned and hardened to prevent false positives or worse false negatives from being produced. It is here that we focus our efforts by designing a general hyperparameter tuning framework with Python, Dask, Keras,Scikit-learn, and Tensorflow which allows for easy expansion of hyperparameter sets and automated parameter combination testing using a distribution high performance computing cluster. This enables us to test many hyperparameter combinations for the network while evaluating the accuracy of tornado prediction for each possible combination and how long it takes for the network to both train and perform a prediction based on application data. Additionally we implement a system to take small amounts of training data and turn them into large amounts of training data via augmentation allowing small sample sets of data to train a network as accurately as a larger distinct data set or even help remove bias that can be introduced by an imbalanced data set.

Section 1 gives a brief overview of the application, purpose, and general ideas behind the work done. Section 2 of this technical report discusses the atmospheric data set used to train an algorithm for identifying vorticity within a storm system. are possible. Section 2 also describes the data augmentation completed that was necessary to avoid any bias being introduced in the training process. Section 3 describes our method for tuning hyperparameters using the various mentioned python libraries and how we utilize this framework. Section 4 conducts a performance study for CPUs and GPUs and present the results and comparisons thereof. Lastly Section 5 discusses our conclusions and possibilities of future work that can be done on this project.

2 Data

2.1 Base 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 [4]. 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 $32 \times 32 \times 4$ "images" describing the storm. Figure 2.1 shows an example image from one of these files. Storms are defined as having simulated radar reflectivity of 40 dBZ or greater. Reflectivity, in combination with the wind field, can be used to estimate the probability of specific low-level vorticity speeds. The dataset contains nearly 80,000 convective storm centroids across the central United States.

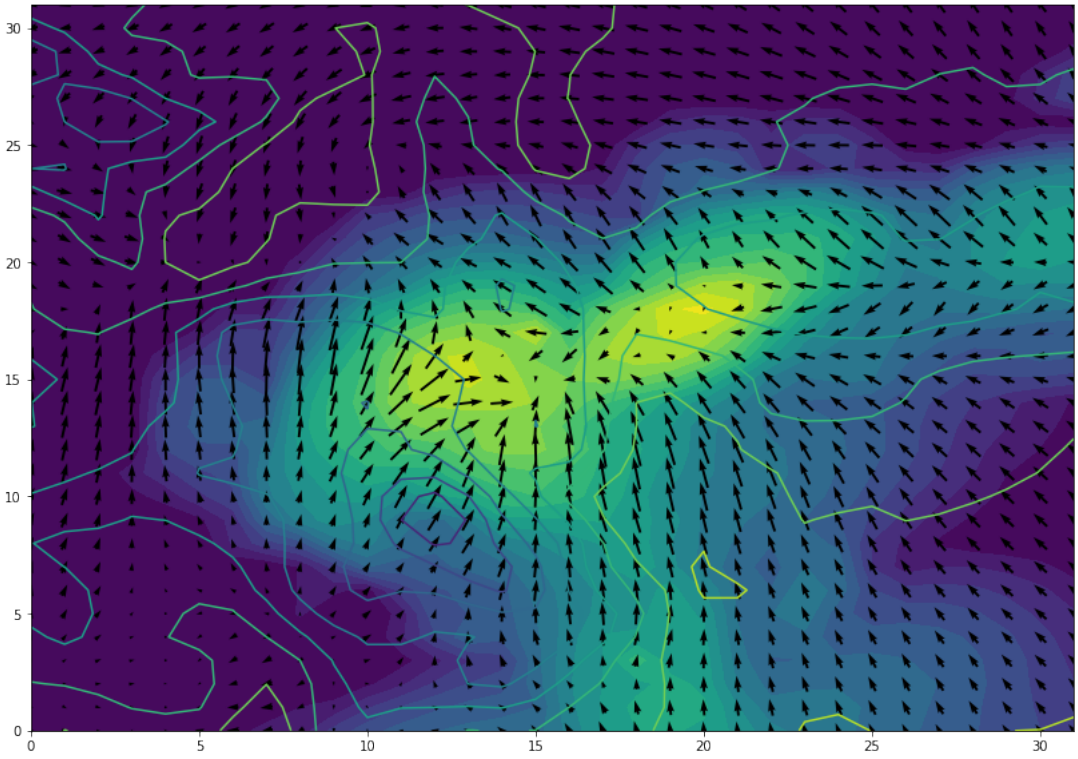


Figure 2.1: An example image of the radar reflectivity and wind field for a storm.

2.2 Data Augmentation

The data set comes from numerically simulated thunderstorms from the National Center for Atmospheric Research (NCAR) [4] convection-allowing ensemble. Within both the training and test data set, approximately 5% of examples has a maximum vorticity larger than the threshold value of 0.005, which indicates strong rotation. In the algorithm, the strong rotation case is classified as tornadic. Because of the large ratio between the majority (non-tornadic) and minority (tornadic) classes, which is 19:1, the data set is unbalanced. The problem of unbalanced data for classification problem is that this model will be trained in a way that is biased toward the over-represented data. If the validation and test samples are biased in the same fashion, this will compromise the validation results of the deep learning model. Especially for climate problems such as tornado prediction, false negatives (non-tornado cases) may have life/death consequences for the general public should this prediction model be adopted in practice. Therefore, it is critical to address this issue and to understand its implications with the performance metrics of the deep learning methods.

There are three options that we can use to solve these problems: (i) we can undersample the majority class by deleting some samples at random in this class; (ii) we can oversample the minority class by duplicating some samples at random; (iii) we can generate synthetic data to adjust the ratio of majority and minority classes. In this project, we approach this issue using the last two methods.

The current parallelization framework is built upon the deep learning architectures distributed by the AMS short course on machine learning using Python [4]. For the deep neural network, the input data is a one-dimensional array. We used the `RandomOverSampler` class from `imblearn.over_sampling` to perform random over-sampling of the minority class by picking samples at random with replacement. One of the parameters that we can tune is the desired ratio of the number of samples in the majority class over the number of samples in the minority class after resampling.

The limitation of this method is that it works with a two dimensional array at most. For a two-dimensional convolutional neural network, the inputs are the tensor image data. To use this method, we have to cast the images into the required format. Instead, we resort to a new approach using the `ImageDataGenerator` class from `Keras`. We can use this class to generate batches of tensor image data with real-time data augmentation while training the network. The data will be looped over in batches. With the different parameters provided by this class, new synthetic samples can be generated by translations, rotations, zooming, shearing, and horizontal/vertical flips. Since the labels of the data set are generated by the maximum vorticity value of the image sample, we want to perturb the images with care so that the labels are not accidentally changed. For this purpose, we carefully choose to apply the rotations and flips to the original samples to generate the new ones.

Note that the `ImageDataGenerator` class can shuffle and enlarge the data set on the fly, but it does not automatically handle class imbalances of the data set. Therefore, we pre-process the data before feeding it to the `ImageDataGenerator`. Specifically, since the majority over minority ratio is 19, we duplicate the minority class 18 times, and append to the original data set to make the entire data set largely balanced. We can then shuffle the data, and use `ImageDataGenerator` to slightly vary the images randomly, using rotations and reflections. Instead of augmenting the data in real time while doing the training, we can also complete the augmentation at the preprocessing stage before the data is fed into the model via `skimage.transform.rotate`. We notice that it is also possible to explore alternative metrics more suitable for unbalanced dataset.

3 Parallelism of Hyperparameter Tuning

3.1 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. Below, we propose a method for the parallelism of hyperparameter tuning using the dask cluster distribution combined with a cross-validated hyperparameter grid search, implemented with Keras and a Tensorflow backend.

3.2 Framework

Dask (ver1.2.2) is a parallel programming library that combines with the Numeric Python ecosystem to provide parallel arrays, dataframes, machine learning, and custom algorithms and is based on Python and the foundational C/Fortran stack. It can be implemented locally or on a distributed cluster and excels in large memory jobs. 'Dask Distributed' is the specific branch of Dask that allows for integration into a high performance computing cluster. There are many ways to distribute work across a cluster, however, we will focus on the common method of applying a function across a list of iterables, with each iteration being distributed to an open process on the cluster. It even goes as far as to create and submit automatically generated SLURM files to the scheduler for the user. Additionally one can also feed custom SBATCH arguments like `--cpus-per-task=1` to Dask and it will append them to the SBATCH collection during the creation of the worker nodes. This usability and conceptual operational is identical to `multiprocessing.Pool.map` but the processes can live on one or several nodes rather than being confined to a single machine. The primary limitation to this method is that we are limited to a single iterable and thus could only tune one hyperparameter at a time using this method alone. It is here that we combine this approach with a cross validated grid search when fitting each model.

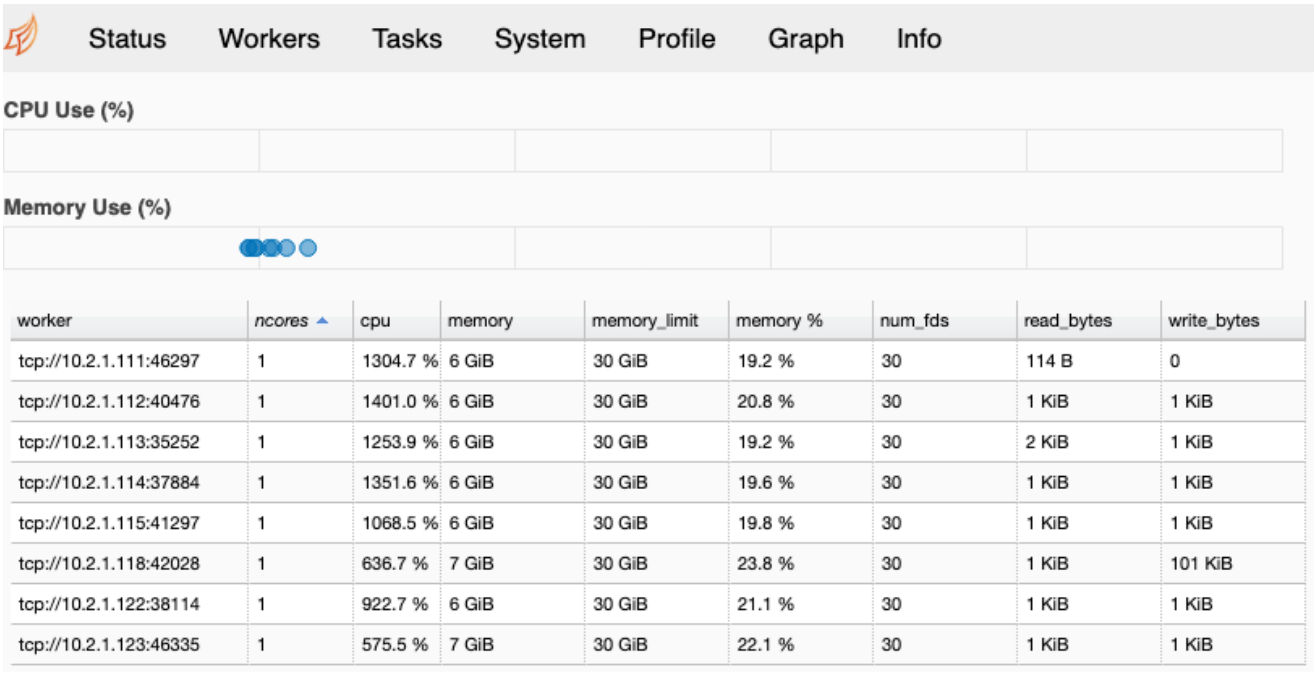
Python's well-known scikit-learn (ver0.20.3) machine learning module has many built-in features to help train and tune models appropriately, however, many of them are not optimized for deep networks and big data. We chose to implement the GridSearchCV method which produces a hyperparameter grid using stratified K-Fold cross validation, known as brute force training. We combine these two methods, and thus create a nested grid of hyperparameters with the outer iterable being distributed over the cluster. To link the two methods and build the models themselves, we utilize Keras (ver2.2.4) with Tensorflow (ver1.13.2) as the backend for CPU testing. To integrate Keras models into scikit-learn you must wrap them in the KerasClassifier or KerasRegression class, as a function, which acts as a model constructor. This enables the ability to utilize scikit-learns grid search, however one more step is needed to distribute them along the cluster.

Due to technical limitations we cannot use TensorFlow 1.13 for GPU computations. The current version of the kernel is not new enough to use the newest TensorFlow. Instead we opt for TensorFlow 1.12 with GPU capabilities which we hope has comparable speeds.

Dask is used to first initiate workers, with the option of static or dynamic scaling, and

then waits for processes to be received. We then build a train function to train the grid search models. Lastly, we utilize the `client.map()` method to distribute the train function across our processes, iterating over our outer parameter we pass. To further simplify the process, Dask easily integrates with many popular cluster schedulers such as SLURM and PBS.

One additional feature of Dask Distributed is the Dask Dashboard, which can be used to stream the performance of the cluster to view memory consumption, CPU usage, time spent per process, among other features. One must use port forwarding to enable this; Dask workers utilize a default port of :8787 for this purpose. Figure 3.1 shows a sample screenshot of the dashboard. The figure represents the 'workers' tab which in this example, is streaming each dask worker as an entire compute node. In our SLURM configuration file, we set '`--tasks-per-node=1`' as each task is a separate grid search function and is noted as such in the '`ncores`' sub-column. Thus, setting the tasks to 1, allows the CPU usage to exceed 100%, with there being 100% available per CPU within the node. The example shown here is from a study (similar to the one in the following section) using compute nodes with 16 CPUs, and thus a maximum CPU Usage, sub-column `cpu`, of 1600%. We note some of the workers consistently utilizing over 1300% ($> 81.25\%$ total) processing power per node, and other workers utilizing much less (but still much greater than 100%), the lowest in this example at 575.5% (36% total). This is directly related to the batch size, which in this example, is the parameter we varied per node. Too small of a batch size prevents optimal computing efficiency. Optimizing total CPU usage is slightly out of scope for this project, but it is important to see how different configurations utilize available CPU power. Many other metrics can be explored through the dashboard such as memory and CPU usage though time (System tab) and time spent per specific task (Tasks tab).



worker	ncores	cpu	memory	memory_limit	memory %	num_fds	read_bytes	write_bytes
tcp://10.2.1.111:46297	1	1304.7 %	6 GiB	30 GiB	19.2 %	30	114 B	0
tcp://10.2.1.112:40476	1	1401.0 %	6 GiB	30 GiB	20.8 %	30	1 KiB	1 KiB
tcp://10.2.1.113:35252	1	1253.9 %	6 GiB	30 GiB	19.2 %	30	2 KiB	1 KiB
tcp://10.2.1.114:37884	1	1351.6 %	6 GiB	30 GiB	19.6 %	30	1 KiB	1 KiB
tcp://10.2.1.115:41297	1	1068.5 %	6 GiB	30 GiB	19.8 %	30	1 KiB	1 KiB
tcp://10.2.1.118:42028	1	636.7 %	7 GiB	30 GiB	23.8 %	30	1 KiB	101 KiB
tcp://10.2.1.122:38114	1	922.7 %	6 GiB	30 GiB	21.1 %	30	1 KiB	1 KiB
tcp://10.2.1.123:46335	1	575.5 %	7 GiB	30 GiB	22.1 %	30	1 KiB	1 KiB

Figure 3.1: An example image from the Dask Distributed dashboard.

3.3 Hardware Description

The University of Maryland, Baltimore County High Performance Computing Facility 'taki' cluster is a heterogeneous cluster with equipment acquired in 2009, 2013, and 2018. It contains over 170 CPU nodes, a GPU cluster containing 20 hybrid CPU/GPU nodes. We only used the 2018 and 2013 portions of the cluster. Currently the 2018 portion of taki is comprised of 42 compute and 2 develop nodes, each with two Intel Skylake CPUs each with 18 cores and 384 GB of memory. There is 1 GPU node containing 4 NVIDIA Tesla V100 GPUs connected by NVLink and two 18 core Intel Skylake CPUs. The 2013 portion of the cluster contains 49 compute, 2 develop, and 1 interactive nodes, each containing two 8-core Intel Ivy Bridge CPUs and 64 GB of memory. Additionally there are 18 CPU/GPU nodes, each is a hybrid node with two 8-core Intel Ivy Bridge CPUs and 2 NVIDIA K20m GPUs.

4 Results

We use the framework detailed in Section 3 to demonstrate a small time performance study. The model described here was based on the that used in the Machine Learning in Python for Environmental Science Problems AMS Short Course [4]. The short course used a convolutional neural network architecture that is shown in Figure 4.1. We choose our hyperparameter grid to consist of epochs, learning rate (using the 'Adam' optimization function) and batch size with values of [5,10,15], [0.001, 0.005, 0.01], and [128, 256, 512, 1024, 2048, 4096], respectively. Batch size was the 'outer' parameter that was changed and distributed over multiple nodes, each consisting of a large suite of models from the scikit-learn CV grid-search. Each suite of models was cross validated with a stratified 3-fold validation, making for a total of 162 models trained in the study: 6 batch sizes $\times$ 3 epochs $\times$ 3 learning rates $\times$ 3 validation runs. Fully reproducible Python code using this specific framework can be found on GitHub [2].

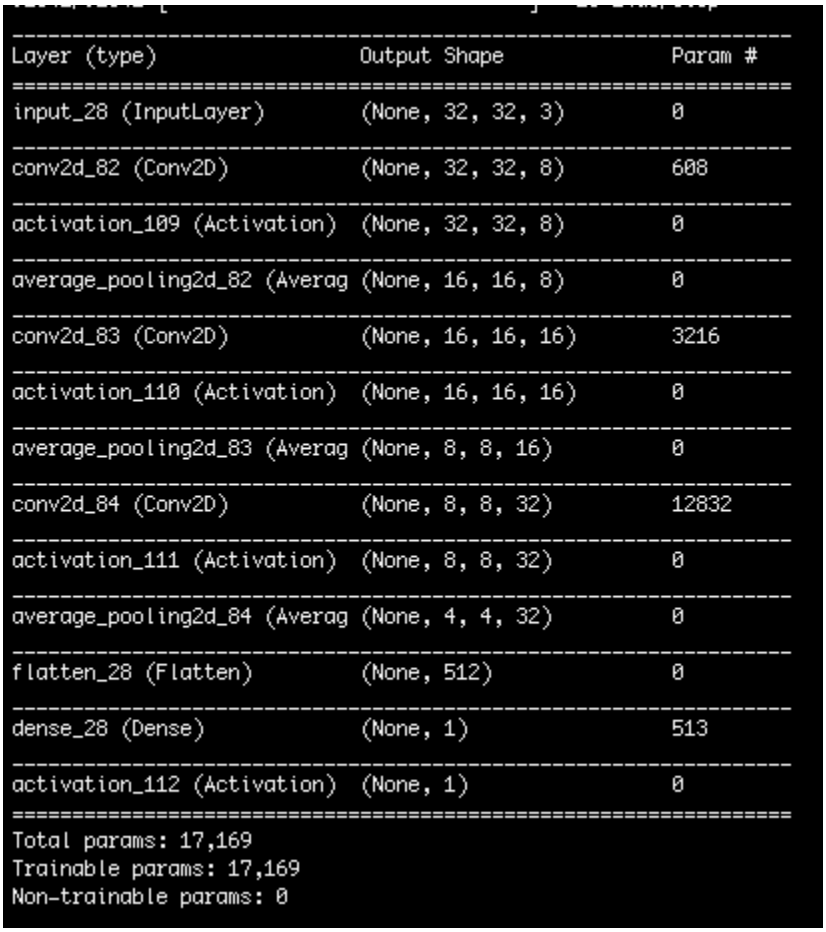


Figure 4.1: Initial model structure, adapted from Machine Learning in Python for Environmental Science Problems AMS Short Course [4].

A script and description of how to download the dataset can be found at https://github.com/djgagne/ams-ml-python-course/blob/master/download_data.py[1] [4]. We preprocessed the original NCAR storm data containing $183,723$ distinct storms, each of which consists of 32×32 grid points, and extracted composite reflectivity, 10m west-east wind component in meters per second, and 10m south-north wind component in meters per second at each grid point giving approximately 2GB 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 without data augmentation and $44,760$ storms for testing the accuracy of the model. We track the total wall time for both training and test over all images in the sets.

4.1 Hyperparameter Tuning

The display of hyperparameter tuning poses a challenge, as it is often a large n-dimensional grid. We chose a relatively small grid to optimize over for two primary reasons: (a) It will be easier to visualize the data, and (b) we are less concerned with finding the actual best model here and are more concerned with demonstrating the approach.

For all epoch numbers, we provide the mean accuracies and standard deviations for each model configuration. Accuracy universally trends downward with increasing batch size, which is expected as there are less weights to learn. Figure 4.2, Figure 4.3, and Figure 4.4 provide the mean accuracies and corresponding standard deviations for each batch size for 5, 10, and 15 epochs, respectively, with each learning rate separated by color. Mean accuracy and standard deviation used in the figures were calculated from three stratified, cross validated runs. In general, the standard deviation did not show any trends with increasing batch size, accuracy, or learning rate. However, the distribution of the standard deviations can provide insight on how many K-folds to use for cross validation. This is a demonstration using 3-fold cross validation, and the spread of standard deviation here probably lends itself to a higher K-fold scheme.

The larger learning rate of 0.01 was, overall, significantly less accurate than the lower learning rates (0.001 and 0.005) in all three scenarios. Learning rates of 0.001 and 0.005 had similar accuracy results, though Figure 4.4 shows better performance from a learning rate of 0.005. Overall combined accuracy increases for all three learning rates as epoch size increases, indicating the model may be slightly underfit and training with more epochs could be beneficial. These plots also provide insight on the most efficient batch size, without losing significant accuracy, which is clearly demonstrated in figure 4.4 as a batch size of 512. Thus, we can conclude that the optimal configuration (within our chosen grid space) is a batch size of 512, a learning rate of 0.005 and total epochs of 15. In practice, a larger grid space would be ideal to reduce noise and more clearly identify trend and ensure your model is neither underfit or overfit.

[1] Dataset can be downloaded from https://storage.googleapis.com/track_data_ncar_ams_3km_csv_small/track_data_ncar_ams_3km_csv_small.tar.gz.

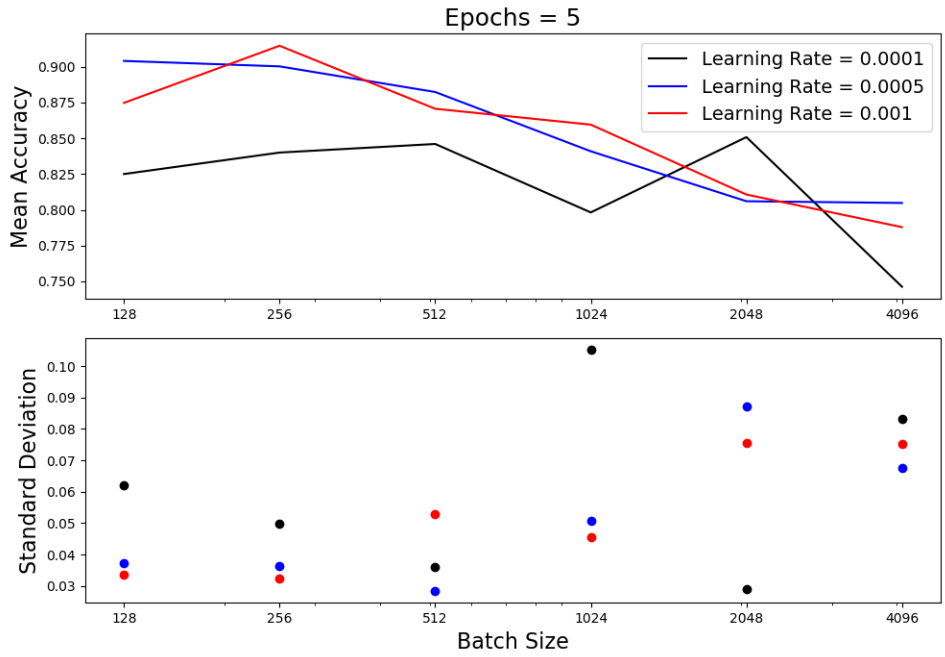


Figure 4.2: Accuracy results for a range of batch sizes and learning rates with 5 epochs (top) and the corresponding standard deviation for each set of results (bottom).

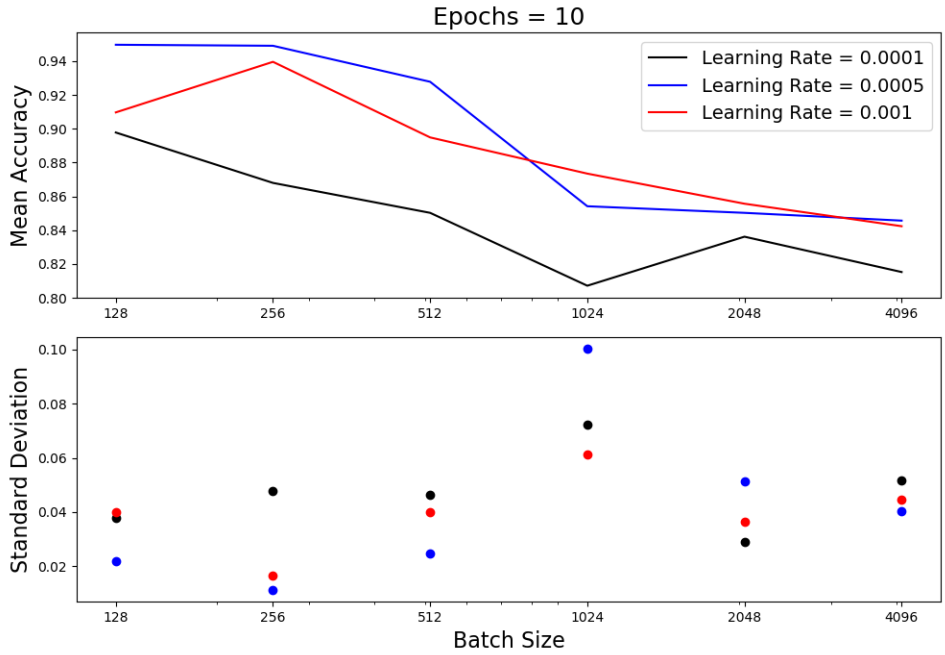


Figure 4.3: Accuracy results for a range of batch sizes and learning rates with 10 epochs (top) and the corresponding standard deviation for each set of results (bottom).

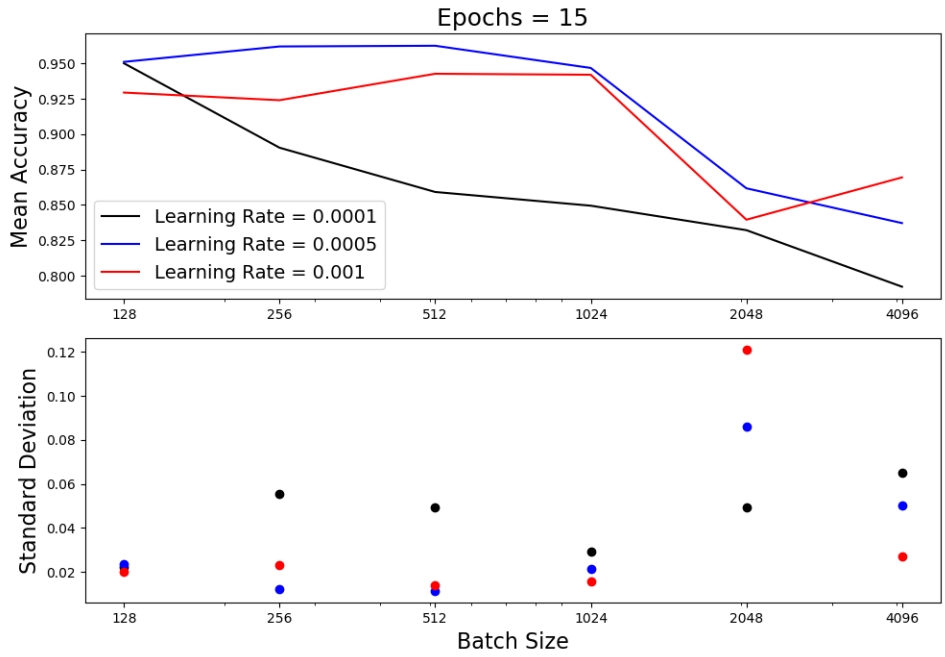


Figure 4.4: Accuracy results for a range of batch sizes and and learning rates with 10 epochs (top) and the corresponding standard deviation for each set of results (bottom).

4.2 CPU Performance

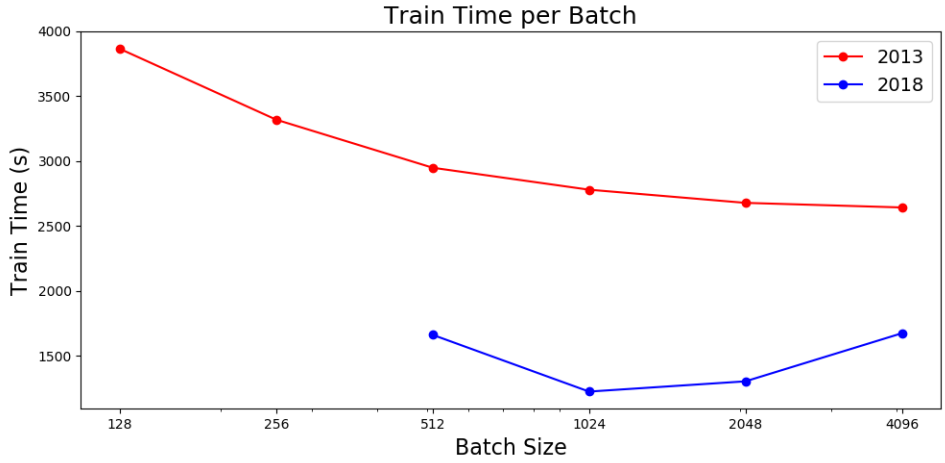


Figure 4.5: The training time for each batch size for runs on 2013 (red) and 2018 (blue) nodes. Each data point represents the total time it takes to train all combinations of hyper-parameters with the given batch size.

We ran the same study on both the 2013 and 2018 hardware and timed the results for each model grid distributed to each node (by batch size) and can be seen in Figure 4.5.

2013 hardware shows an exponential decrease with increasing batch size, which is generally expected, as less computation is necessary for increased batches. The same timing pattern is not found in the 2018 nodes, with increasing times as the batch size increases in later iterations. The primary reason for this is still uncertain to us at this time. However, the main take away from the figure, is that overall timing on the 2018 hardawre is significantly reduced across all batch sizes compared to the 2013 hardware, due to the effective parallelization utilizing 2018's 32 available CPUs vs 2013's 16. Note that each time displayed in this figure are representative of the entire run for each batch size and are not an average as the accuracy measurements are.

4.3 GPU Performance

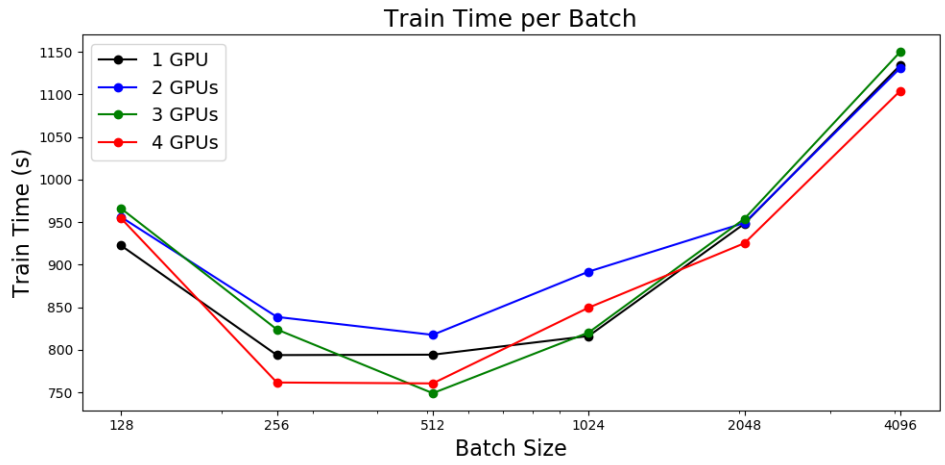


Figure 4.6: The training time for each batch size using a varying number of GPUs. Each data point represents the total time it takes to train all combinations of hyperparameters with the given batch size.

Figure 4.6 is the performance time that occurs when using one to several GPUs for trainings with a varying number of batch sizes. This chart is arranged and is considered to be similar in thought to Figure 4.5. The most notable aspect of this plot is that despite the variation in batch size and total GPUs used for computation there is a clear U bend in each of the lines. Each GPU line bottoms out at some time associated with one particular batch size and then the times become slower as the batch size increases. It is more important to see that even though this general trend is followed by all lines do not share the same optimal batch size. For example 4 GPUs have 256 as the optimal batch size but 1 GPU has an optimal batch size of 512. Additionally 2 GPUs and 3 GPUS have the same optimal batch size of 512 but 3 GPUs performs much better at this optimal point. There is also no "best" line. Each arrangement of GPUs performs worse than another at least once. At a batch size of 512 3 GPUs actually out perform 4 GPUs. At a batch size of 4096 and 128 2 GPUs out perform 3 GPUs. The optimal number of GPUs depends largely batch size and overall datasize. With

a smaller dataset like this we cannot be completely sure that this behavior holds for larger datasets. The GPUs are clearly powerful and a superior way to train even if the number that should be used undetermined.

5 Conclusions and Future Work

In this paper, we propose a general framework for augmentation and a parallel tuning scheme for hyperparameters using a combination of the popular Python utilities Dask, Scikit-learn, Keras, and for data agumentation using the RandomOverSampler package and Keras' ImageDataGenerator. We implemented a convolutional neural network trained by the augmented and balanced data and demonstrated results over a variety of configurations, showing trends related to the tuned hyperparameters. We then tested this approach by conducting a sample performance study on the 2013 and 2018 nodes of the HPCF cluster at the University of Maryland, Baltimore County.

We also compared the performance of two CPU architectures and the NVIDIA V100. The 2018 nodes vastly out performed the 2013 nodes. This is not a very surprising as the 2018 nodes have double the number of cores per node even if the 2018 nodes are slightly slower. While the iteration size changed the 2018 did not show any meaningful speedup and actually had a slow down after a certain point. The reason is not particularly clear. One could argue that even if the batch size is smaller or larger the same amount of data is still being put into the CNN. For smaller batch sizes the weights are updated more frequently but there is a less data to perform computation on at a time. For larger batch sizes the weights are updated less frequently but there is more data to perform computation on. We see the same behavior when comparing the performance trend of the GPUs to the CPUs. There is an optimal point with the best performance where after training becomes much slower. This optimal point is not the same for all numbers of GPUs. It is possible that with a larger dataset, achieved through augmentation or a new problem, could yield more concrete behavior and trends. The only conclusive finding in the performance study is that regardless of how many GPUs are used the GPUs always out perform the CPUs are the same batch size despite core count.

Possible future work includes doing indepth hyperparameter tuning. We only mutate four hyperparameters when we could be changing several other hyperparameters and we could also test more variability in our current hyperparameter set. Additionally this study was fairly small in size and primarily used as a sample to test the framework. To address the problematic smallness of our dataset we would want to use the data augmentation feature we implemented in order to balloon the size of our data significantly. Doing so would allow us to better tune our hyperparameters and see longer less volatile performance times which grants us more substantial results.

Acknowledgments

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). The hardware in the UMBC High Performance Computing Facility (HPCF) 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. Co-author Carlos Barajas was supported as HPCF RAs.

References

[1] Lindsey R. Barnes, Eve C. Gruntfest, Mary H. Hayden, David M. Schultz, and Charles Benight. False Alarms and Close Calls: A Conceptual Model of Warning Accuracy. *Weather and Forecasting*, 22(5):1140–1147, 2007.

[2] Charlie Becker, Will D. Mayfield, Sarah Y. Murphy, and Bin Wang. https://github.com/big-data-lab-umbc/cybertraining/tree/master/year-2-projects/team-3. Source Code.

[3] Wael Ghada, Nichole Estrella, and Annette Menzel. Machine Learning Approach to Classify Rain Type Based on Thies Disdrometers and Cloud Observations. *Atmosphere*, 10(251):1–18, 2019.

[4] R. Lagerquist and D. J. Gagne II. Basic machine learning for predicting thunderstorm rotation: Python tutorial. https://github.com/djgagne/ams-ml-python-course/, 2019.

[5] Valliappa Lakshmanan, Christopher Karstens, John Krause, Kim Elmore, Alexander Ryzhkov, and Samantha Berkseth. Which Polarimetric Variables Are Important for Weather/No-Weather Discrimination? *Journal of Atmospheric and Oceanic Technology*, 32(6):1209–1223, 2015.

[6] Amy McGovern, Kimberly L. Elmore, David John Gagne II, Sue Ellen Haupt, Christopher D. Karstens, Ryan Lagerquist, Travis Smith, and John K. Williams. Using Artificial Intelligence to Improve Real-Time Decision-Making for High-Impact Weather. *Bulletin of the American Meteorological Society*, 98(10):2073–2090, 2017.

[7] Vahid Nourani, Selin Uzelaltinbulat, Fahreddin Sadikoglu, and Nazanin Behfar. Artificial Intelligence Based Ensemble Modeling for Multi-Station Prediction of Precipitation. *Atmosphere*, 10(2):80–27, 2019.