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

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 [6]. Machine learning has been used to accurately forecast rain type [2] [6], clouds [2], hail [5], 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 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. 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 the public. Minimizing both false alarms and missed alarms are key in weather forecasting and public warning systems.

Section 2 of this technical report discusses the data and types of augmentation used in this study. An atmospheric dataset is being used to train an algorithm for identifying vorticity within a storm system. This metric can be used to asses the severity of a storm. This section also describes data augmentation completed that was necessary to avoid any bias being introduced in the training process. Section 3 describes a method for tuning hyperparameters with parallelism over a distributed cluster using a integrated combination of popular Python modules - Dask, Scikit-learn and Keras. We then utilize this framework to conduct a performance study and present the results in section 4.

2 Data

2.1 Base Data

The dataset 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 [3]. 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 x 32 x 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.

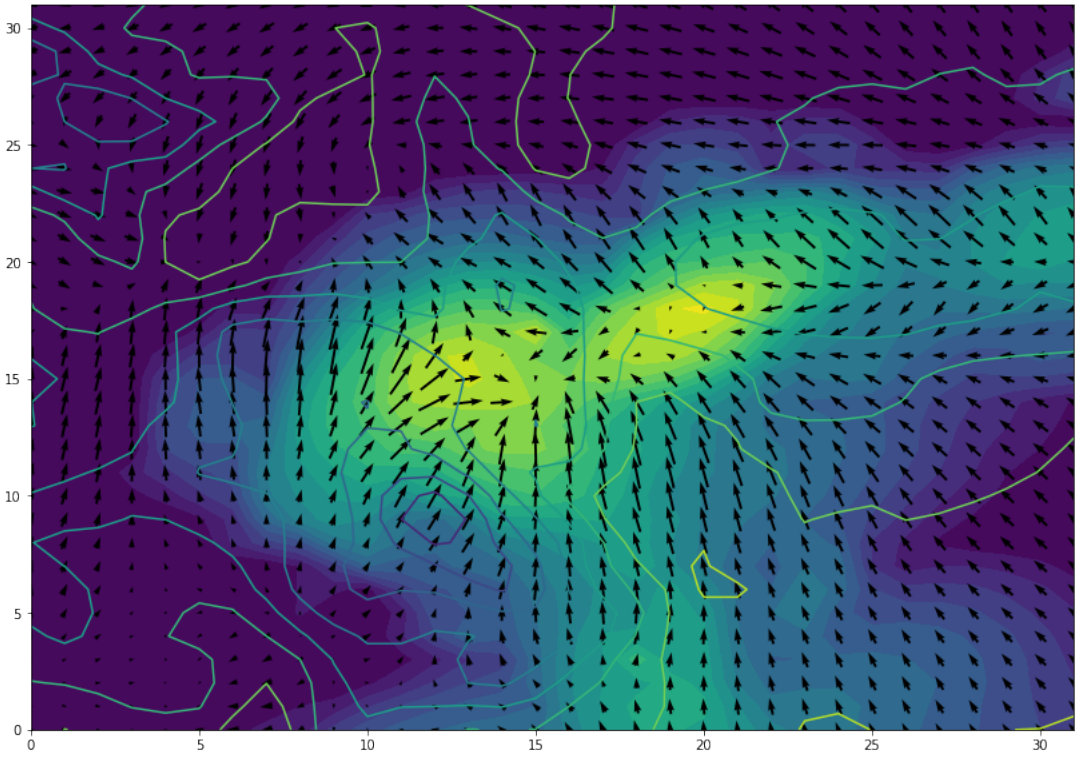


Figure 2.1: An example image of the radar reflectivity and wind field for a storm.

2.2 Data Augmentation

The data set comes from numerically simulated thunderstorms from the National Center for Atmospheric Research (NCAR) [3] 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 several options that we can tackle this: (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 [3]. 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 at most rank two array. 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 max 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 doesn't 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.

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

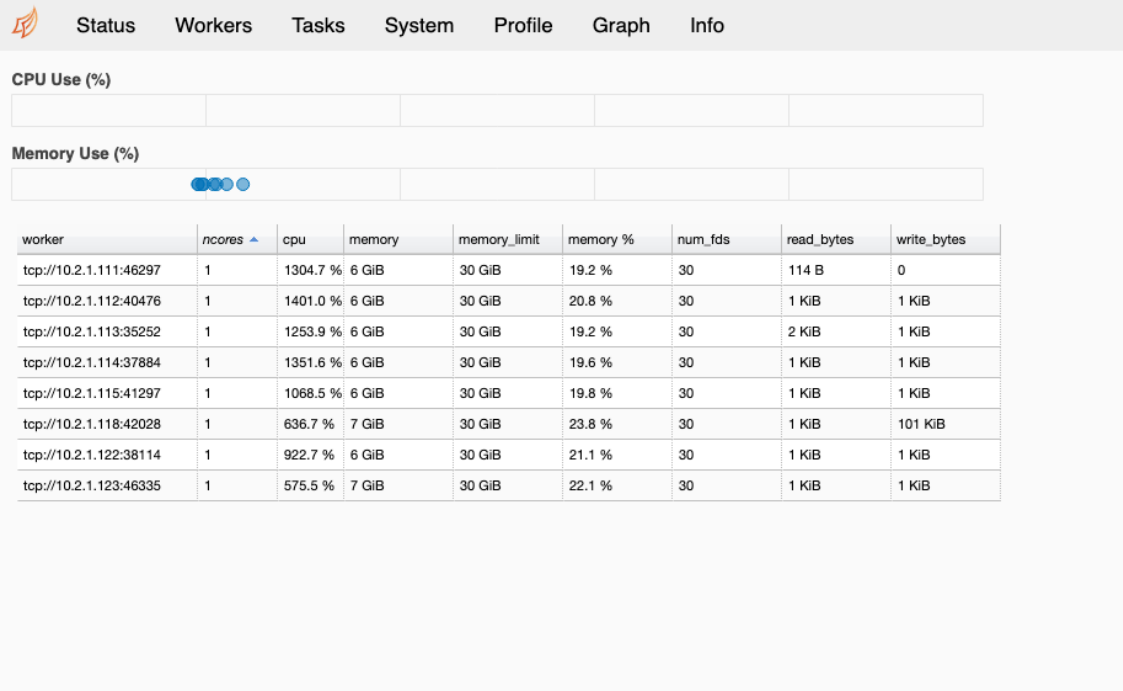
Dask is a parallel programming library that combines with the Numeric Python ecosystem to provide parallel arrays, dataframes, machine learning, and custom algorithms and is based on Python and the foundational C/Fortran stack. It can be implemented locally or on a distributed cluster and excels in large 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. This is identical to the common `map()` function in Python, but is worked on by multiple processes. 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 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 the Keras framework using Tensorflow as the backend. 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.

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 CPU's, 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

4 Performance Study

4.1 Hardware and Software

The University of Maryland, Baltimore County High Performance Computing Facility 'taki' cluster contains both a CPI cluster (170 nodes) and a GPU cluster (19 nodes). The `HPCF2018` division has 42 compute nodes, each with two Intel Skylake CPUs contatining 18 cores, and one GPU node with 4 NVIDIA Tesla V100 GPUs. The `HPCF2013` division has 49 compute nodes, each with 8-core Intel Ivy Bridge CPUs.

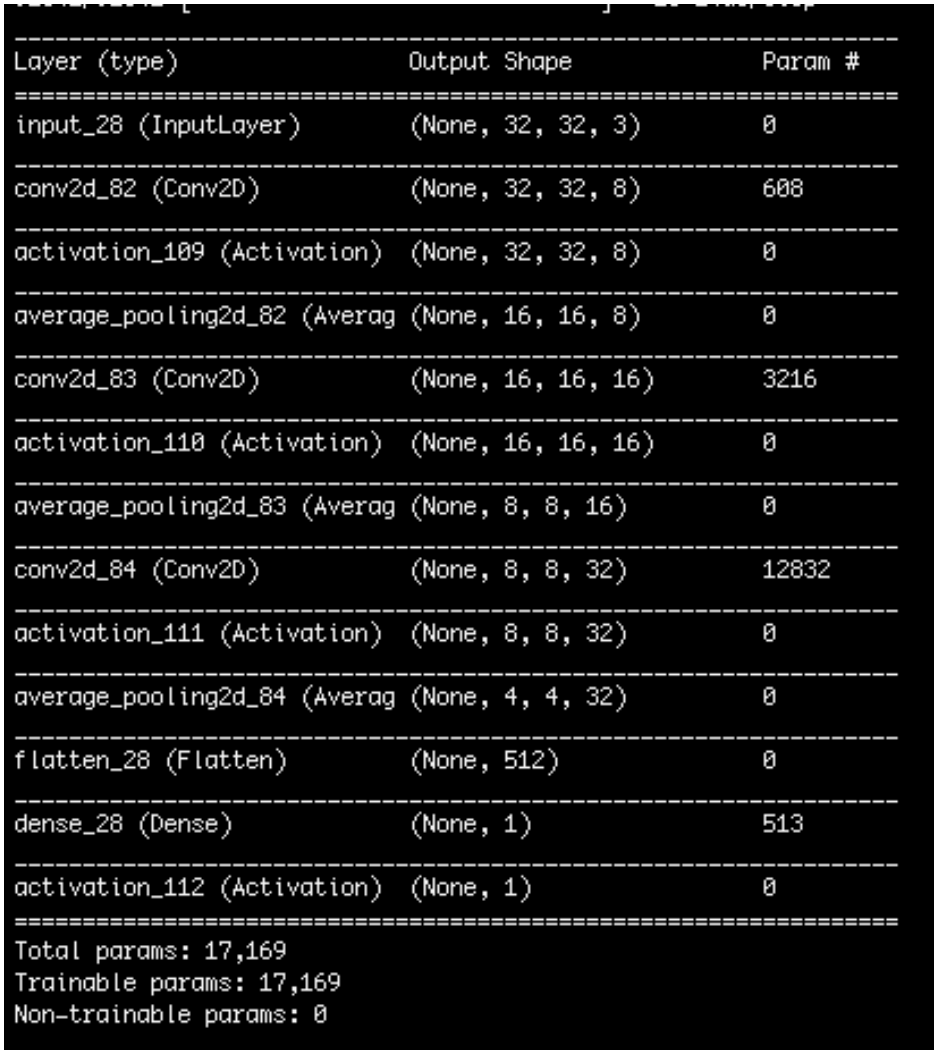
The software versions used in this study are as follows:

- `dask/0.16.0-intel-2017b-Python-3.6.3`
- `Keras/2.2.4-foss-2018b-Python-3.6.6`
- `Tensorflow/1.13.1`
- `scikit-learn/0.20.3`

4.2 Model Framework

We use the above framework to demonstrate a small performance. The model described here was based on the that used in the Machine Learning in Python for Environmental Science

Problems AMS Short Course [3]. The short course used a convolutional neural network architecture that is shown in figure 4.2. 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 (6), each consisting of a large suite of models from the scikit-learn CV gridsearch. 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 $*$ 3 epochs $*$ 3 learning rates $*$ 3 validation runs.



Layer (type)	Output Shape	Param #
input_28 (InputLayer)	(None, 32, 32, 3)	0
conv2d_82 (Conv2D)	(None, 32, 32, 8)	608
activation_109 (Activation)	(None, 32, 32, 8)	0
average_pooling2d_82 (Averag	(None, 16, 16, 8)	0
conv2d_83 (Conv2D)	(None, 16, 16, 16)	3216
activation_110 (Activation)	(None, 16, 16, 16)	0
average_pooling2d_83 (Averag	(None, 8, 8, 16)	0
conv2d_84 (Conv2D)	(None, 8, 8, 32)	12832
activation_111 (Activation)	(None, 8, 8, 32)	0
average_pooling2d_84 (Averag	(None, 4, 4, 32)	0
flatten_28 (Flatten)	(None, 512)	0
dense_28 (Dense)	(None, 1)	513
activation_112 (Activation)	(None, 1)	0

Total params: 17,169
Trainable params: 17,169
Non-trainable params: 0

Figure 4.1: Initial model structure, adapted from Machine Learning in Python for Environmental Science Problems AMS Short Course [3].

4.3 Results

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. Figures 4.2, 4.3, and 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.

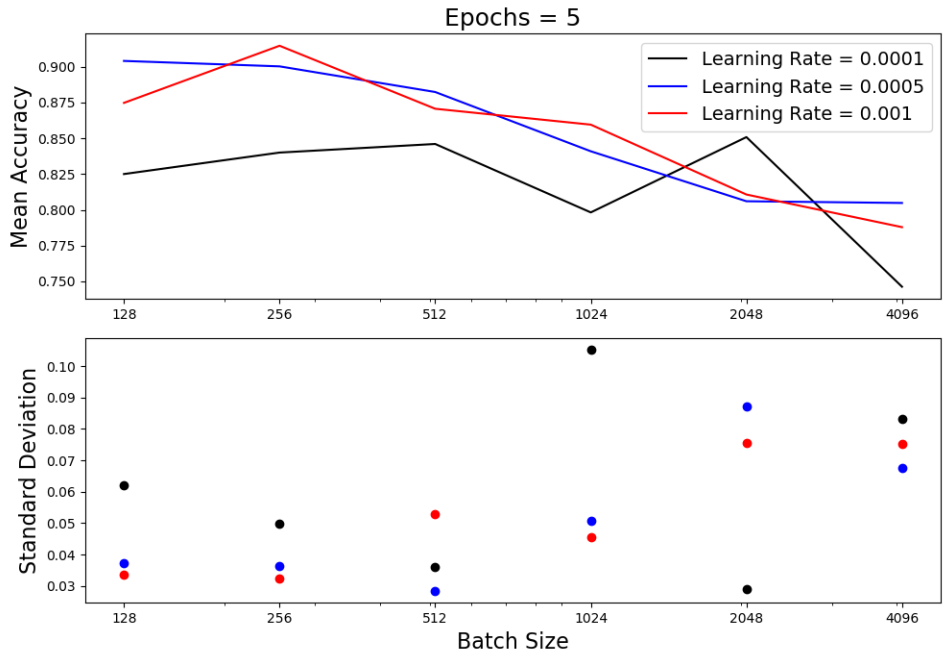


Figure 4.2: Accuracy results for a range of batch sizes and learning rates with 5 epochs (top) and the corresponding standard deviation for each set of results (bottom).

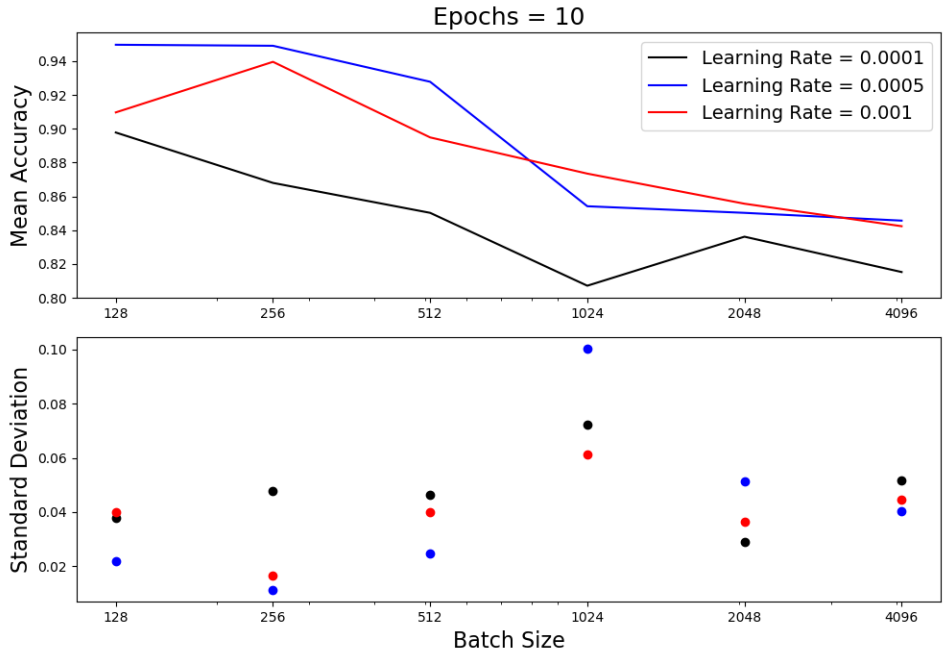


Figure 4.3: Accuracy results for a range of batch sizes and learning rates with 10 epochs (top) and the corresponding standard deviation for each set of results (bottom).

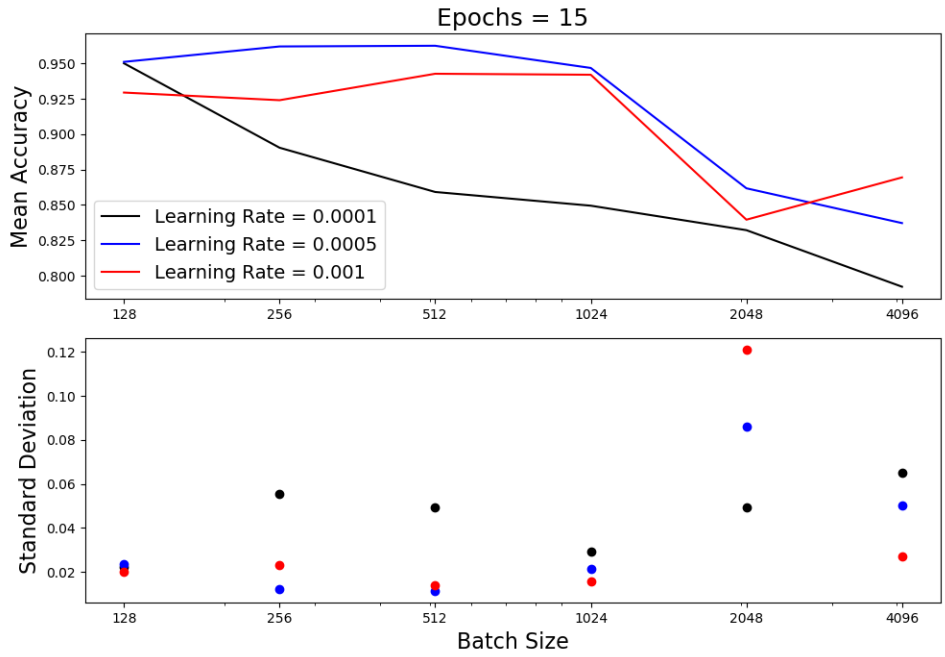


Figure 4.4: Accuracy results for a range of batch sizes and and learning rates with 10 epochs (top) and the corresponding standard deviation for each set of results (bottom).

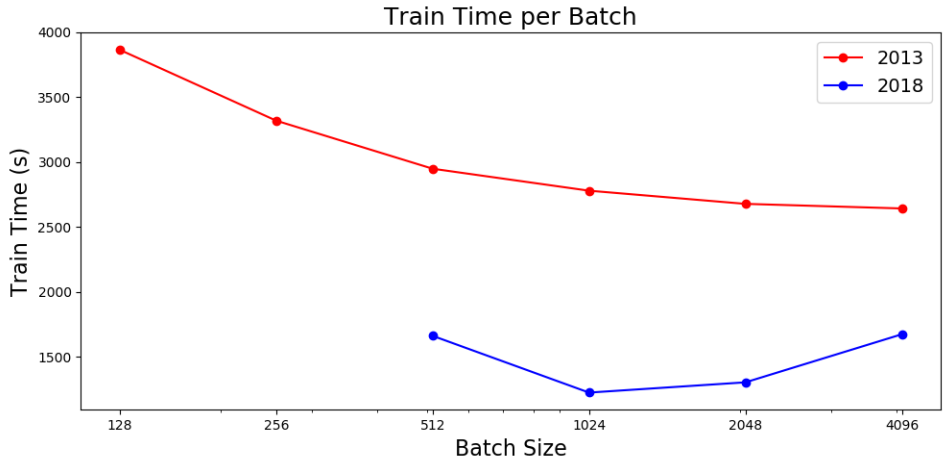


Figure 4.5: The training time for each batch size for runs on 2013 (red) and 2018 (blue) nodes.

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 CPU's 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.

5 Conclusions

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 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 'Taki' cluster at the University of Maryland at Baltimore county. Efficient parallelism is most comprehensively shown by the timing results in the 2013 and 2018 configurations, as the 2018 cluster has twice (32) as many available processors as the 2013 nodes (16).

5.1 Future Work

Possible future work includes parallelizing the hyperparameter tuning process using GPU enabled tensorflow on taki. While in test cases a GPU implementation of Tensorflow displayed a significant speedup, the capability was not available in time to include in this study. Furthermore, this study was fairly small in size and primarily used as a sample to test the framework. A logical next step would be to use this on a more robust study that may train on the order of days rather than hours.

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, October 2007.

[2] 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, May 2019.

[3] R. Lagerquist and 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.

[4] 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, June 2015.

[5] 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, October 2017.

[6] 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, February 2019.

Appendix I

Code used to produce presented results. Data was preprocessed seperately for more clarity.

```
from sklearn.model_selection import GridSearchCV
import time
import numpy as np
import pandas as pd
import xarray as xr
import matplotlib.pyplot as plt
from glob import glob
from os.path import join, expanduser
from sklearn.preprocessing import StandardScaler
from ipywidgets import interact
import ipywidgets as widgets
from keras.models import Model, save_model, load_model
from keras.layers import Dense, Activation, Conv2D, Input, AveragePooling2D,
    MaxPooling2D, Flatten, LeakyReLU, Dropout
from keras.layers import SpatialDropout2D
from keras.optimizers import SGD, Adam
from keras.regularizers import l2
import keras.backend as K
from scipy.ndimage import gaussian_filter
from sklearn.metrics import mean_squared_error, roc_auc_score
#from imblearn.over_sampling import RandomOverSampler
from keras.preprocessing.image import ImageDataGenerator
from dask_jobqueue import SLURMCluster
from dask.distributed import Client
from dask.distributed import progress
from keras.wrappers.scikit_learn import KerasClassifier, KerasRegressor

######################## INITIATE CLUSTER ###################################

cluster = SLURMCluster(cores=1, memory='32 GB',
    job_extra=['--exclusive','--qos=normal','--time=03:00:00'])
cluster.scale(6)

client = Client(cluster)

#############################################################################

start = time.time()
bs = [128,256,512,1024,2048,4096]
```

```
###################### RETRIEVE MODEL METRICS ##############################

def calc_verification_scores(test_labels,predictions):

    model_auc = roc_auc_score(test_labels, predictions)
    model_brier_score = mean_squared_error(test_labels, predictions)
    climo_brier_score = mean_squared_error(test_labels, np.ones(test_labels.size),
        * test_labels.sum() / test_labels.size)
    model_brier_skill_score = 1 - model_brier_score / climo_brier_score
    print(f"AUC: {model_auc:0.3f}")
    print(f"Brier Score: {model_brier_score:0.3f}")
    print(f"Brier Score (Climatology): {climo_brier_score:0.3f}")
    print(f"Brier Skill Score: {model_brier_skill_score:0.3f}")
    return model_auc
    #, model_brier_score, model_brier_skill_score

###################### BUILD MODEL STRUCTURE ##############################

def create_model(learning_rate='0.0001'):

    # Deep convolutional neural network
    num_conv_filters = 8
    filter_width = 5
    conv_activation = "relu"
    #learning_rate = 0.001
    # Input data in shape (instance, y, x, variable)
    conv_net_in = Input(shape=(32, 32, 3))  # train_norm_2d.shape[1:]
    # First 2D convolution Layer
    conv_net = Conv2D(num_conv_filters, (filter_width, filter_width),
        padding="same")(conv_net_in)
    conv_net = Activation(conv_activation)(conv_net)
    # Average pooling takes the mean in a 2x2 neighborhood to reduce the image size
    conv_net = AveragePooling2D()(conv_net)
    # Second set of convolution and pooling layers
    conv_net = Conv2D(num_conv_filters * 2, (filter_width, filter_width),
        padding="same")(conv_net)
    conv_net = Activation(conv_activation)(conv_net)
    conv_net = AveragePooling2D()(conv_net)
    # Third set of convolution and pooling layers
    conv_net = Conv2D(num_conv_filters * 4, (filter_width, filter_width),
        padding="same")(conv_net)
    conv_net = Activation(conv_activation)(conv_net)
    conv_net = AveragePooling2D()(conv_net)
```

```
    # Flatten the last convolutional layer into a long feature vector
    conv_net = Flatten()(conv_net)
    # Dense output layer, equivalent to a logistic regression on the last layer
    conv_net = Dense(1)(conv_net)
    conv_net = Activation("sigmoid")(conv_net)
    conv_model = Model(conv_net_in, conv_net)
    # Use the Adam optimizer with default parameters
    opt = Adam(lr=learning_rate)
    conv_model.compile(opt, "binary_crossentropy", metrics=['accuracy'])

    print(conv_model.summary())
    return conv_model

#################### LOAD DATA AND DISTRIBUTE MODEL ########################

def distribute_model(param):

    train_out = np.load('/home/cbecker/team3/research/preprocessedData/,
        train_out.npy')
    test_out = np.load('/home/cbecker/team3/research/preprocessedData/,
        test_out.npy')
    train_norm_2d = np.load('/home/cbecker/team3/research/preprocessedData/,
        train_norm_2d.npy')
    test_norm_2d = np.load('/home/cbecker/team3/research/preprocessedData/,
        test_norm_2d.npy')
    train_norm_2d_new = np.load('/home/cbecker/team3/research/preprocessedData/,
        train_norm_2d_new.npy')
    train_out_new = np.load('/home/cbecker/team3/research/preprocessedData/,
        train_out_new.npy')
    out_threshold = 0.005

    augmentation=False

    t0 = time.time()
    l = []

    model = KerasClassifier(build_fn=create_model, batch_size = param)

    if augmentation==True:
       datagen = ImageDataGenerator(
                 rotation_range=5,
                 width_shift_range=0,
                 height_shift_range=0,
```

```
            shear_range=0,
            zoom_range=0,
            horizontal_flip=True,
            fill_mode='nearest')

   #datagen.fit(train_norm_2d_new)
   print(train_norm_2d_new.shape)
   print(train_out_new.shape)
   print("Running augmented training now, with augmentation")
   history=model.fit_generator(datagen.flow(train_norm_2d_new, train_out_new,
    batch_size=param,shuffle=True),
        steps_per_epoch=len(train_norm_2d_new)/param,epochs=10)
   indices_test=np.where(test_out>out_threshold)[0]
   test_out_pos=test_out[indices_test]
   test_out_new=np.tile(test_out_pos,18)
   test_out_new=np.concatenate((test_out,test_out_new),axis=0)
   test_norm_2d_pos=test_norm_2d[indices_test,:,:,:]
   test_norm_2d_new=np.tile(test_norm_2d_pos,(18,1,1,1))
   test_norm_2d_new=np.concatenate((test_norm_2d,test_norm_2d_new),axis=0)
   t1 = time.time()
   return calc_verification_scores(model, test_norm_2d_new, test_out_new), t1-t0


else:
  print("Running regular training, no augmentation")
  #ros=RandomOverSampler(random_state=0)
  #train_norm_2d_resampled, train_out_resampled=ros.fit_resample(train_norm_2d,
    train_out)
  start_time = time.time()
  e = [5,10,15]
  learning_rate=[0.001, 0.005, 0.01]
  param_grid = dict(epochs=e,learning_rate=learning_rate)
  grid = GridSearchCV(estimator=model, param_grid = param_grid, cv = 3,
    n_jobs = -1)
  grid_result = grid.fit(train_norm_2d_new, train_out_new)
  means = grid_result.cv_results_['mean_test_score']
  stds = grid_result.cv_results_['std_test_score']
  params = grid_result.cv_results_['params']
  end_time = time.time()
  run_time = end_time - start_time
  #preds = model.predict(test_norm_2d)
  for mean, stdev, p in zip(means, stds, params):
    x = ("%f (%f) with: %r, time: %r and batch: %r" %,
```

```
            (mean, stdev, p, run_time, param))
        l.append(x)
    #calc_verification_scores(test_out,preds)
    return l


    #model.fit(train_norm_2d, train_out, batch_size=param, epochs=10)
    #model_scores =  calc_verification_scores(preds, test_norm_2d, test_out)
    #l.append(model_scores)
    #t1 = time.time()
    #return l, t1-t0



################### DISTRIBUTE MODELS AND GATHER RESULTS ##################

models = client.map(distribute_model, bs)
print(*(client.gather(models)), sep='\n')
end=time.time()
print('Total time used is ', end-start)
client.close()
```