

Parallel Hyperparameter Tuning of Accuracy for Deep Learning based Tornado Predictions

Senior Thesis, Spring 2020, Mentor: Matthias K. Gobbert

Jonathan N. Basalyga

Department of Mathematics and Statistics, University of Maryland, Baltimore County

Abstract

Predicting violent storms and dangerous weather conditions with current physics based weather models can take a long time due to the immense complexity associated with numerical simulations. Machine learning has the potential to classify tornadic weather patterns much more rapidly, thus allowing for more timely alerts to the public. In this work, we examine what impact varying the batch size and the number of GPUs a convolutional neural network is trained on has on the network's accuracy at classifying storm data. We conclude that using multiple GPUs to train a single network has no significant advantage over using a single GPU. Therefore, multiple GPUs should instead be used to maximize search throughput by using each of them simultaneously for single GPU runs or to solve larger problems by pooling their memory.

Key words. Deep learning, parallel performance, TensorFlow, Keras, GPU programming.

1 Introduction

Forecasting storm conditions using traditional, physics based weather models can pose difficulties in simulating particularly complicated phenomena. These physically based weather models are computationally demanding and time consuming, typically hours for every run that starts with the current meteorological conditions. 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 [8].

A forecaster must use care when using binary classifications of severe weather such as those which are provided in this work. 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 [3]. 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 a supervised neural network such as a convolutional neural network (CNN) for these binary classification scenarios.

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. It should be understood that this computation is only done once for training; the resulting network can then be used in mere minutes to process particular data of the current meteorological conditions. In order to quickly tune, train, and test the validity of a neural network with several different hyperparameter combinations, a parallel framework originally introduced in [5] to train many networks simultaneously with varying hyperparameter values in a high performance computing environment is used. This framework was used to explore data augmentation's effect on wall time in [1, 2]. This thesis presents material covered in [4]. We use the above mentioned framework to investigate how batch size and GPU count affect accuracy.

The remainder of this thesis is organized as follows. Section 2 introduces the natural data used for training the neural networks. Section 3 gives a basic introduction to convolutional neural networks. Section 4 discusses hyperparameters and their importance in training and the parallel framework used for hyperparameter tuning in a high performance computing environment. Section 5 presents the effect of various hyperparameter configurations on the neural network’s accuracy at classifying storm data. Lastly Section 6 collects the conclusions of this work.

2 Data

The data set used in this analysis is included in the Machine Learning in Python for Environmental Science Problems AMS Short Course, provided by David John Gagne from the National Center for Atmospheric Research [7]. 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 2.1 shows two example images from one of these files. Storms are defined as having simulated radar reflectivity of 40 dBZ or greater as seen in Figure 2.1 (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 2.1 (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 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.

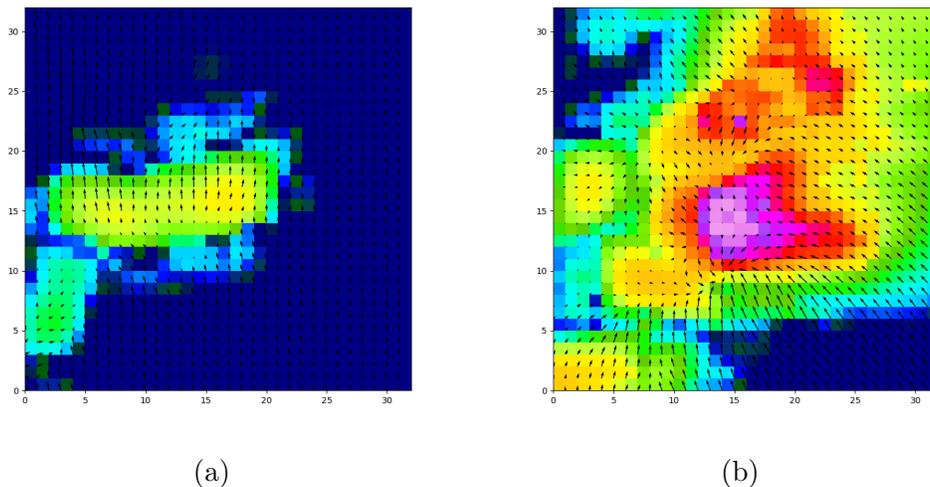


Figure 2.1: Sample images of radar reflectivity and wind field for a storm which (a) does not and (b) does produce future tornadic conditions.

3 Deep Learning with Convolutional Neural Networks

The general idea 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 outputs are unknown. In order to gauge how accurate the network has become we provide data that was not in the training data set and the CNN uses the knowledge gained from training to guess the outcome of data that it has not seen before [6]. 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 [9]. 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. Finally there is 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, or 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 $32 \times 32 \times 3$ 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 [4]. 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 seem 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.

4 Parallelism of Hyperparameter Tuning

4.1 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, 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 number 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 to machine learning.

4.2 MPI Framework for Parallelized Training

The Dask framework for hyperparameter tuning in an HPC environment from [1, 5] was used as a baseline for the new framework [2, 4]. 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 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 synchronized 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.

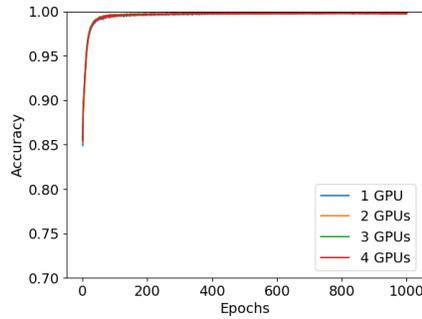
By replacing Dask with these systems we have enabled a method which allows us to measure the effects of every single hyperparameter combination rather than just viewing things grouped by batch size. We now have the ability to group by any arbitrary hyperparameter and examine how each one plays a role 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.

5 Results

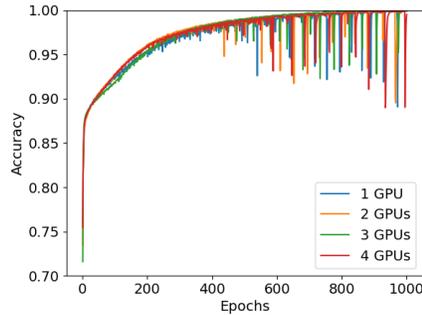
We use the framework detailed in Section 4 to train a series of convolutional neural networks with a learning rate of 0.001 using Keras' `multi_gpu_model` to classify storm data as tornadic or non-tornadic. The number of GPUs used to train a single network is varied from 1 to 4, while batch size is varied from 128 to 65,536 by powers of 2, except in the 1 GPU case, which, due to memory limitations, has a maximum batch size of 32,768. We investigate the effect that batch size and GPU count have on *accuracy*. To ensure the networks are fully trained, as well as to reflect real world usage patterns, we use a large number of epochs, as many as 1,000. Additional tests investigating timings, as well as the effect of data augmentation, can be found in [4].

The numerical studies in this work use one GPU node from 2018 containing four NVIDIA Tesla V100 GPUs connected by NVLink. The node has 384 GB of memory (12×32 GB DDR4 at 2666 MT/s) connected through two 18-core Intel Xeon Gold 6140 Skylake CPUs (2.3 GHz clock speed, 24.75 MB L3 cache, 6 memory channels).

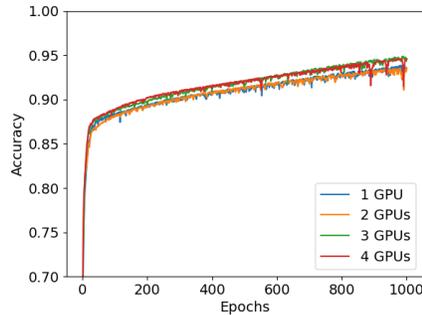
Figure 5.1 shows training accuracy vs. epochs varied by number of GPUs for batch sizes 128, 4,096, and 32,768. Note that the sudden drops in accuracy (especially prominent in the batch size 4,096 plot) result from the use of dropout layers. In the batch size 128 plot accuracy plateaus after only a small 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



(a) Batch size 128



(b) Batch size 4,096



(c) Batch size 32,768

Figure 5.1: Training accuracy vs. epochs for different batch sizes, varying 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 vs. epochs resulting from keeping GPU count fixed and varying batch size are shown in Figure 5.2. 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 5.1. 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

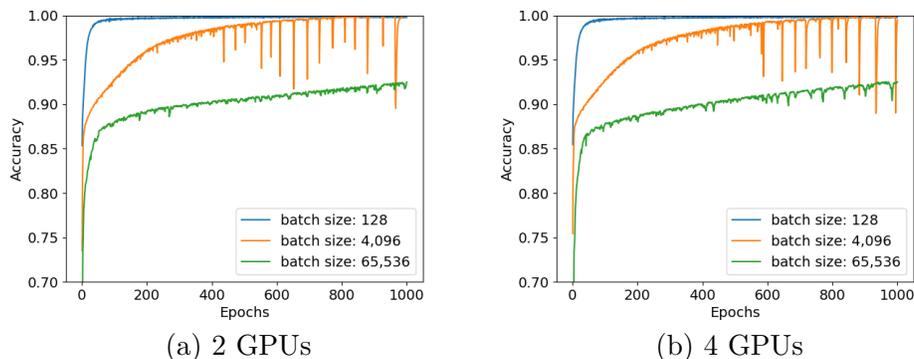


Figure 5.2: Training accuracy vs. epochs for different GPU counts, varying batch sizes.

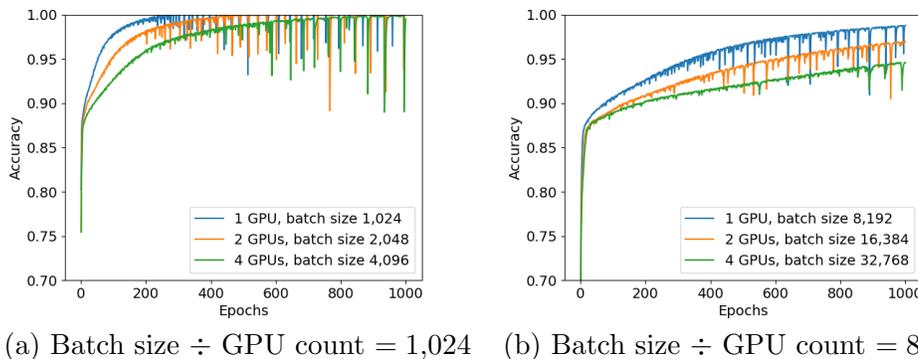


Figure 5.3: Training accuracy vs. epochs varied by both GPUs and batch size simultaneously.

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 5.3 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 5.1 contains the testing accuracies in percent of the network, organized by batch size versus epochs, with 2 GPUs. 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 5.2, 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 in minutes and seconds of each run of the network are presented in Table 5.2. 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

Table 5.1: Testing accuracies (as percentages) for batch size versus epochs with 2 GPUs.

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	

Table 5.2: Timings (in minutes:seconds format) for batch size versus epochs with 2 GPUs.

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	

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 5.1 that accuracy continues to decrease across an entire row. As a result of these effects, neither maximizing nor minimizing any of these hyperparameters leads to optimal performance. This behavior can be observed when examining Table 5.2. 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 time for the same comparable accuracy.

6 Conclusions

The tests in Section 5 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 [4], 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 argument in [4] 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.

Acknowledgments

We thank Carlos Barajas for providing the code used to run the studies done in this work. He also provided a great deal of guidance and patiently explained the trickier aspects of neural networks. We also thank Dr. Matthias K. Gobbert for his mentoring and support over the course of this work. 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 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] Carlos 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.
- [2] Carlos A. Barajas, Matthias K. Gobbert, and Jianwu Wang. Performance benchmarking of data augmentation and deep learning for tornado prediction. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3607–3615. IEEE, 2019.
- [3] Lindsey R. Barnes, Eve C. Grunfest, 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.
- [4] Jonathan N. Basalyga, Carlos A. Barajas, Matthias K. Gobbert, and Jianwu Wang. Performance benchmarking of parallel hyperparameter tuning for deep learning based tornado predictions. *Big Data Research*, submitted (2020).
- [5] Charlie Becker, Will D. Mayfield, Sarah Y. Murphy, Bin Wang, Carlos Barajas, and Matthias K. Gobbert. An approach to tuning hyperparameters in parallel: A performance study using

climate data. Technical Report HPCF–2019–13, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2019.

- [6] François Chollet. *Deep Learning with Python*. Manning, 2018.
- [7] 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.
- [8] 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.
- [9] Douwe Osinga. *Deep Learning Cookbook*. O’Reilly Media, 2018.