11/21/14
CS74: Machine Learning
Final Project Report

Location Independent Weather Forecasting

Andrew Pillsbury and Rebecca Leong

OVERVIEW:

Weather forecasting has traditionally been treated as a well-studied physics-based phenomenon for a specific location. Even so, weather still exhibits data patterns that can potentially be utilized as a basis for future prediction. By gathering data from a variety of different locations in the continental United States, we are attempting to create a location-neutral weather forecaster. For any given location, the model will predict 24 hours of future weather based on data from the past 32 hours in that location. This would offer significant benefits over current methods of weather forecasting that require region-specific models.

A Recurrent Neural Network with two hidden layers was used to model this data. The network was trained with a mixture of standard back-propagation and back-propagation through time (BPTT). Specific modifications were implemented to allow for continuous output values and accelerated convergence rates. We found that the model will extend weather trends, but does not detect changes in such trends.

DATA:

The data we are using is quality controlled local climatological data from the National Climatic Data Center (http://cdo.ncdc.noaa.gov/qclcd/QCLCD?prior=N). We have gathered hourly weather data from every day in 2013 from nine different cities, one for each climate region in the United States. These cities are: Seattle, WA, San Francisco, CA, Cortez, CO, Bismarck, ND, Dallas, TX, Atlanta, GA, Indianapolis, IN, Minneapolis, MN, Boston, MA. The weather stations record data roughly 40 times per day at varying intervals, so to normalize the time between recordings we only take the first reading from each four-hour period, which gives us 9 cities * 365 days/city * 24 hours/day * 1 reading/4 hours = 19,710 readings. Each reading has six features: visibility, temperature, dew point, wind speed, wind direction, and pressure. In order to standardize the different features, we have set the mean value of each feature to zero and divided that value by the standard deviation for that feature.

Figure 1: Data sources for our data set. 8760 Readings with 6 Features per reading in each location.

METHODS:

***Terminology*:**
stack (n) – a particular instance of the network in the time sequence
layer – a set of neurons in the network (i.e. hidden layer or input layer)
signal (s) – the value entering a neuron
output (y) – the value leaving a neuron
squashing function (f) – we used hyperbolic tangent to bound our neuron's between -1 and 1.

Throughout this project, we implemented a number of variations on a neural network in order to improve accuracy on prediction results and reduce training time. The learning objective for the network was to reduce the squared error for predicting (using all available features) the single next data point (for a specific feature) in a time sequence of that feature. The variations to our network are addressed here:

Simple Neural Network: Initially we attempted to model our data with a standard feed-forward, back-propagation neural network with two hidden layers (literature indicated that it's rarely beneficial to have more than two hidden layers). We removed the squashing function on the output node to allow the network to output continuous values. After doing some testing, the network architecture complexity appeared insufficient to model the time dependence of our data. The network at best predicted some average of the input data, rather than considering the data as sequences.

Recurrent Neural Network (MonteCarlo weight updates): To address the lack of consideration of time dependence, we implemented a recurrent feed-forward neural network (RNN) with a Monte Carlo update algorithm. In each epoch, the algorithm looped through every weight matrix. For each weight matrix, it updates it with a random weight matrix multiplied by a learning rate, and then checks to see whether this new matrix decreases the error of the network. If it does, the old matrix is replaced with the new matrix, otherwise the operation is repeated with a new random matrix. If after a set number of iterations a matrix that decreases the error has not been created, the learning rate for the weight matrix is reduced. This algorithm generally converged after roughly ten epochs, and was moderately successful on simple networks. As soon as the model complexity was increased by using more neurons in the layers and by using more stacks, however, the algorithm ceased to converge in a reasonable amount of time.

RNN (Back-propagation through time): To help ensure convergence at a minimum within a reasonable timeframe, we implemented the back-propagation through time (BPTT) algorithm for a single layer in the RNN (using the methods outlined by Jaeger). Literature noted that RNN's often could not account for and benefit from data trends that occur more than 10-20 stacks deep. The network initially expected 3 days of hourly input data (72 stacks), but this time interval was increased to 4 hours in order to capture more trends in the input data. Each weight matrix had a bias term that affected the feed-forward, but did not propagate error back. We modified the back-propagation method slightly so that the output weight matrix was updated with the deltas on the output layer rather than the deltas on the last hidden layer (see the update equations in Jaeger fig. 2.18). This allowed us to use the most informed value of error rather than propagating a value from the delta at a previous hidden layer. With the single layer RNN, the network still lacked sufficient complexity to model the non-linear trends in the data.

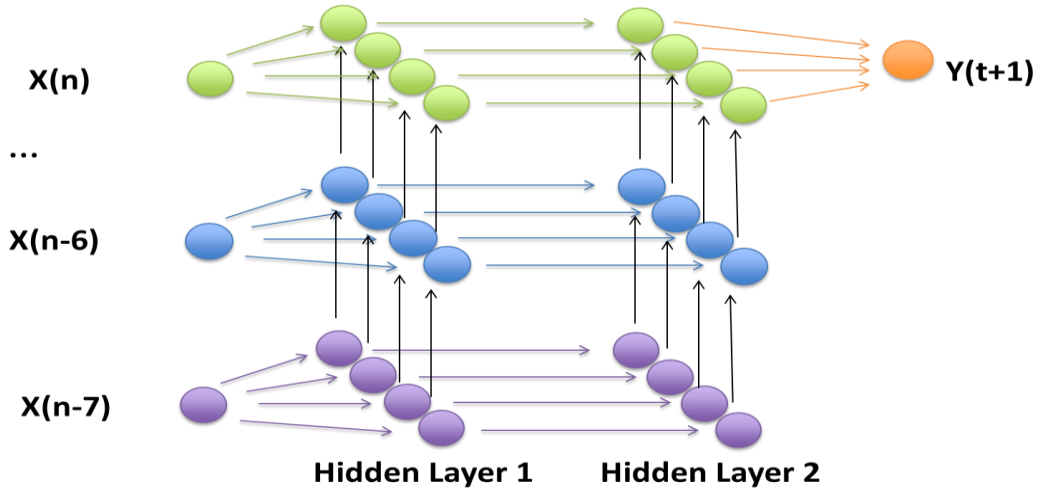


Figure 2: Final architecture for our RNN with two hidden layers, and 8 stacks and 30 neurons (as determined by cross-validation).

RNN (two hidden layers): To create a truly, non-linear system, a second hidden layer was added such that each hidden layer connected to the next layer of neurons as well as to the same layer one step in the future (see figure 2). There was minimal literature regarding such implementation of multiple hidden layer RNN's so we used a mixture of standard back-propagation and back-propagation-through-time. To update weights, the signals from the current stack had to be separated from signals coming from the previous stack:

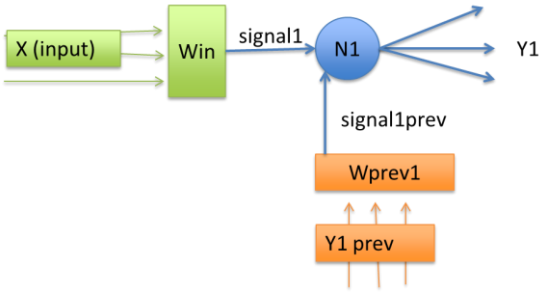


Figure 3: Example of inputs for a single neuron in the first hidden layer. The neurons incoming signal was a linear combination of both the inputs to current stack and the outputs from the previous stack.

For this sample neuron, the feed forward step would be computed as a combination of the signals from the same layer (either input or output from a hidden layer) and output values from the previous hidden layer. Similarly when back-propagating error, the *Wprev1* weight matrix is updated using only the *signal1prev* values and the delta on that neuron and the *Win* weight matrix is updated only using the *signal1* values and the delta. Cross-validation on this architecture showed that 8 network stacks and 30 neurons per layer was optimal (See Appendix A).

Feed-Forward Prediction Methods: A separate RNN was trained (on all features) for each output feature in order to account for varying trends in each data type. To do the actual prediction, the initial input data is fed through each network and then their outputs are used to create a single new data point. The initial data is shifted forward and then the new data point appended as the most recent sample. This allowed us to predict forward an arbitrary number of time units, but any error in prediction would propagate through into future predictions. This prediction method was used on both the MonteCarlo

and backpropagation trained networks. In addition, a third prediction method was derived by averaging the predications of the MonteCarlo and backpropagation networks.

RNN (continuous outputs): After running a few tests on a small data set, the network exhibited insufficient variation in output values. With only -1 and 1 (and the occasional value within the range) as inputs to the output neuron, the weight matrix could only compute a limited number of values with linear combinations. The squashing function was removed from the hidden layer prior to the output neuron. Thus for calculating the final output, no squashing function was applied to the hidden layer neurons. But in the feed-forward algorithm, the signal was squashed before entering the next stack. So all internal signals were bounded between -1 and 1, while all signals entering output nodes were unbounded, continuous values. This allowed for a truly continuous range of output values.

RNN (look-ahead back-propagation): Most literature implementations utilize the intermediate training values when training the RNN (i.e. output values from every stack in the network). However, our learning objective was targeted specifically at the final output value from the network and intermediate values were not of interest. So the back-propagation was modified to only consider the final output when determining update weights. This improved the algorithms ability to model data because it was no longer trying to optimize over all of the output values in its sequence and could focus on optimizing only one.

The usage of our network for predicting involves feeding predicted values back in as inputs, causing any errors to propagate into later predictions and thus the network was very sensitive to a single error made in the prediction sequence. This algorithm seemed to be sufficient at predicting initial values, but lost accuracy further into the prediction. In attempts to address this, we implemented a *look-ahead back-propagation* method to improve future predictions and create a more robust network. The look-ahead training method took our predicting algorithm into account when performing the training in order to optimize weights not just for a single prediction, but for our entire prediction process. For each desired output, additional stacks were predicted using the previous output as an input (for the respective feature) and then the outputs from these additional stacks were used to calculate updates to the weight matrices. The algorithm would theoretically optimize not only for the current output, but also for future outputs and be less sensitive to single inaccuracies in the prediction process.

Adaptive Learning Rate and Smoothing Convergence: After ensuring that the back-propagation methods worked correctly, we implemented a stochastic batch gradient descent to balance a trade-off between convergence speed and network accuracy. We split the data set into three portions (train, valid and test). To determine convergence, an average of the validation error was kept and the algorithm was terminated when the difference between the current iteration and a preset previous number of iterations was less than a set cut-off.

As the complexity of the network increased, it became increasingly unstable and very sensitive to learning rate. This led to either unsuccessful attempts at training or a very slow training process. To address this, we implemented an adaptive learning rate to help maintain stability while also keeping a reasonable convergence time. For each attempt to update the weights, the newly updated matrices were tested on the validation data. If the new matrices produced lower data, they were accepted for future iterations. If not the learning rate was decreased and a new weight matrix was computed using a smaller learning step. This continued until either an improved weight matrix was found or the learning rate fell below a certain value and no update was performed.

RESULTS:

To ensure that our network, particularly our back-propagation functions were working, we ran our data on a very small data set to ensure that we could precisely model it.

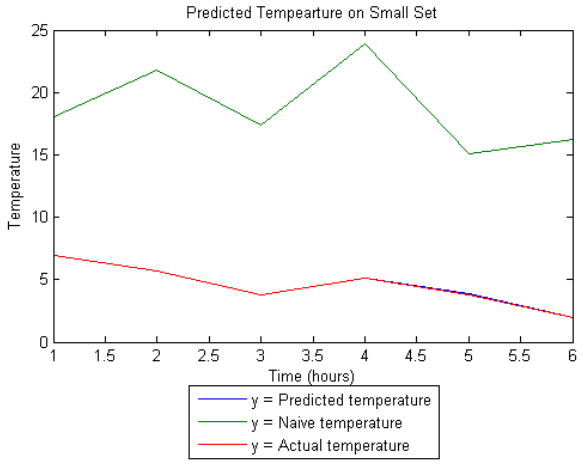


Figure 4: Network error after a training on a very small data set. The Naïve trend represents the prediction prior to any training.

Input data sequences were randomized and partitioned into a train (50%), validate (25%) and test (25%) set for experimentation. We performed a cross-validation and determined that 8 stacks and 30 neurons were optimal for accuracy. The network was then trained on data from all cities for each feature (Dew Point, Visibility, Temperature, Wind Speed, Wind Direction, Pressure) using the back-propagation and MonteCarlo algorithms. Outputs from the two algorithms were averaged to produce a third prediction method (combined). To test localtion-independence, data from Lebanon, NH was predicted on each of these networks (Graphs for the remaining output features as well as other test data can be found in Appendix A):

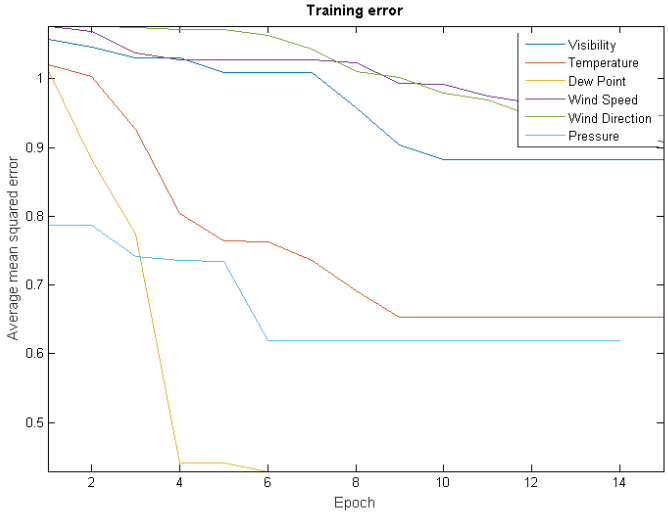


Figure 5: MonteCarlo training error for each feature

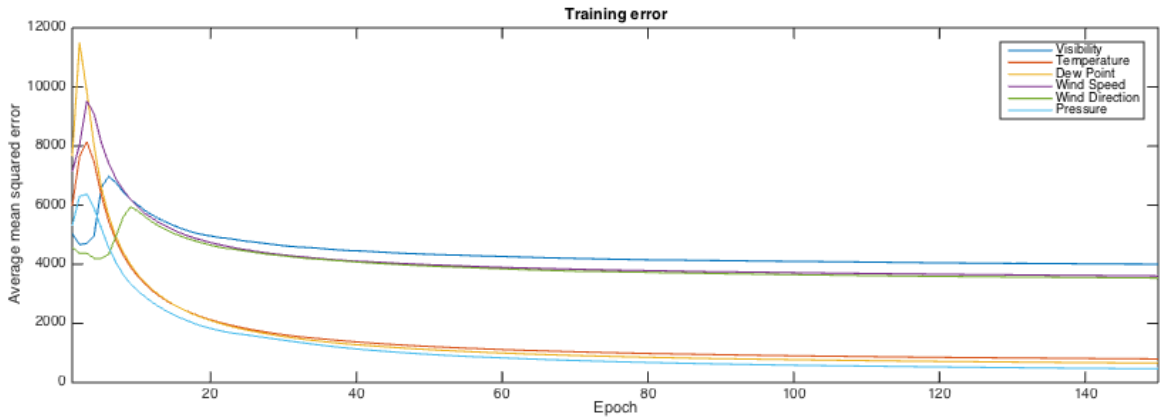


Figure 6: Training average mean squared error for each feature from training a network with backpropagation

The backpropagation method shows a much smoother error descent as it uses partial derivatives to calculate a gradient descent using a small learning step. The MonteCarlo errors appear more step-wise in nature as the algorithm will updates by randomly choosing a decreasing weights rather than adjusting a learning rate.

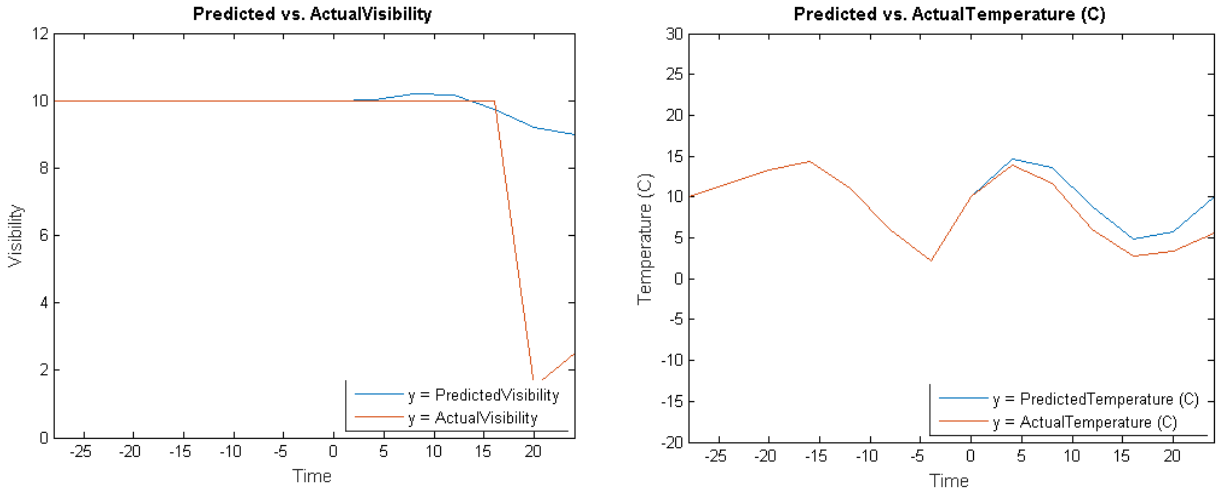


Figure 7: Visibility and temperature prediction (network trained with BPTT) for Lebanon, NH in October. Additional actual data is given as context

These two graphs characterize notable behaviors of the trained network. For the temperature graph, the network does a fair job in accurately predicting values and trends. But for the visibility, the network does not detect the steep decrease and instead continues to predict according to the initial trend in the input data. Overall, the network generally predicts values within the correct range and can predict trends present in the input, but often does not model changes in trends (i.e. from a wave to sharp points). This may indicate that the network lacks sufficient complexity and memory to account for multiple trends and detect indicators to switch between them.

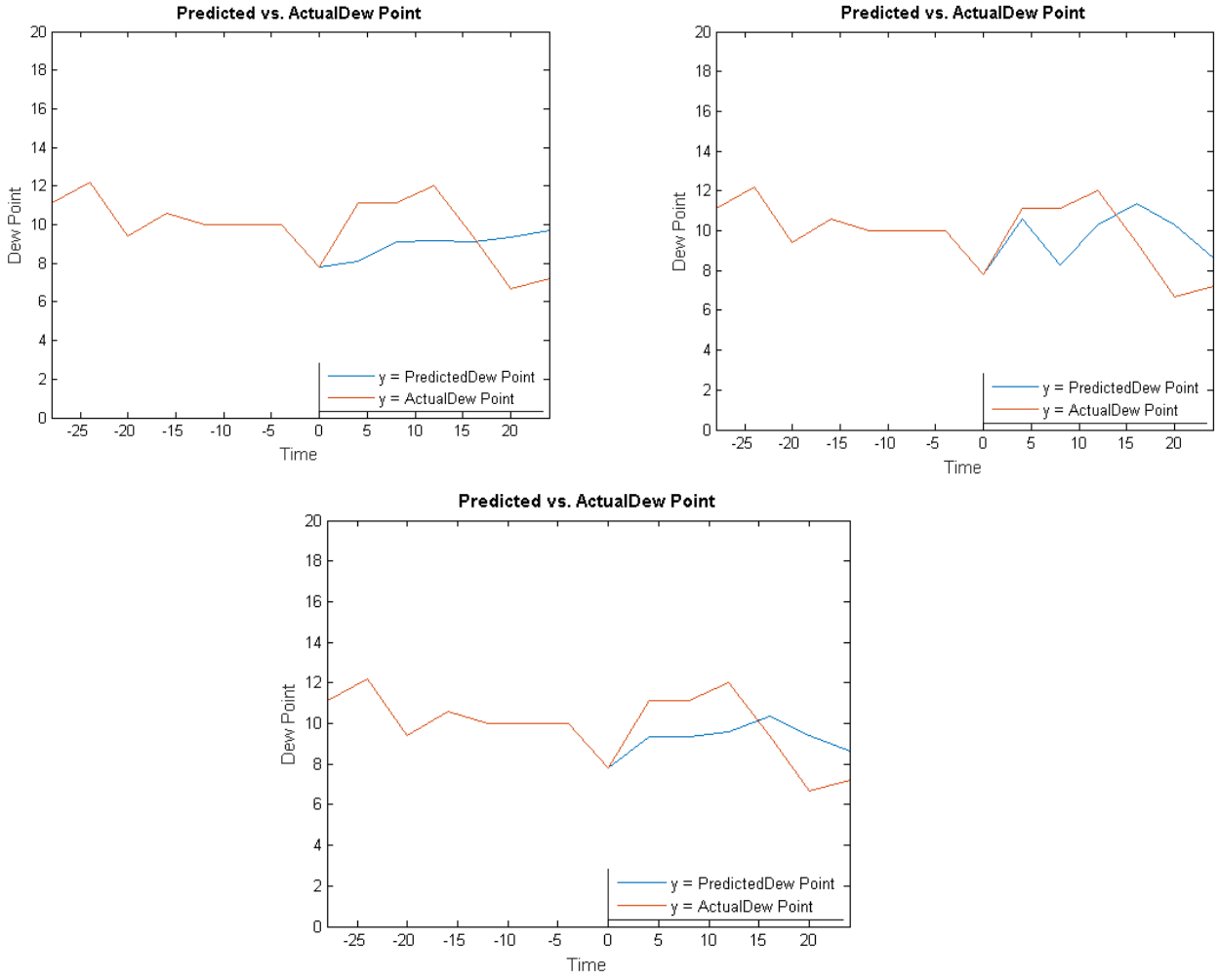


Figure 8: Top left: Dew point predictions by a network trained with backpropagation. Top Right: Dew point prediction by a network trained with MonteCarlo updates. Bottom Center: A combined method that averages the predictions from each network.

In general, the MonteCarlo trained network predicts jumpier data whereas the backpropagation produces a smoother prediction. The backpropagation attempts to optimize its predictions for a range of values by deterministically adjusting weights based on partial derivatives. The MonteCarlo algorithm, on the other hand, is non-deterministic and simply chooses a random weight that will produce good results for a particular pass of input data. Thus the MonteCarlo algorithm is more likely to train itself for a specific few number of trends while the backpropagation method will attempt to capture trends optimized for all the data it has seen. A combined method sometimes offers, at least visually, a good balance between these two characteristics to capture both the general trend and more idiosyncratic trend.

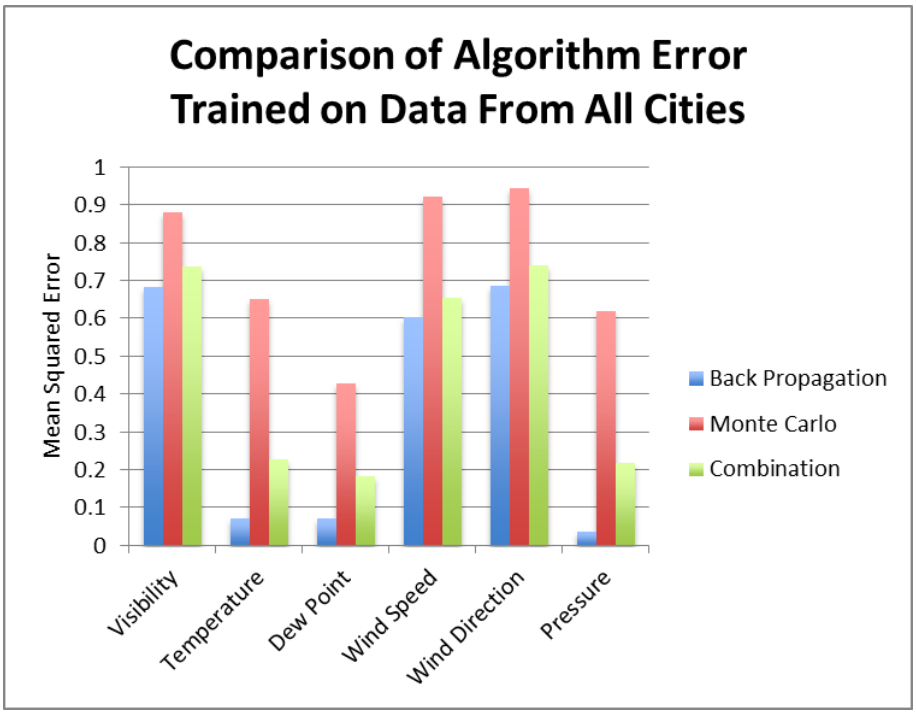


Figure 9: Mean square test error for each output feature for different training and prediction methods. The Combination prediction is an average of the values predicted by a back-propagation and MonteCarlo network.

The backpropagation trained network consistently performs better than a single MonteCarlo trained network. This can likely be attributed to the fact that the backpropagation method specifically optimizes for the learning objective, whereas the MonteCarlo method is non-deterministic and vulnerable to overtraining on a single data trend. In general, averaged over a test set, the combined error generally performs worse than the backpropagation, but better than the MonteCarlo algorithm. But this method of prediction (using outputs for networks trained under different algorithms) shows potential as a means to accurately model different trends in a data system.

CONCLUSION:

Throughout this project we implemented a number of variations to a basic neural network in order to optimize prediction accuracy and training speed for weather prediction. We started with an implementation of a single hidden layer neural network and RNN based on literature. Through experimentation with our data, we identified additional problems and implemented potential solutions. The following features were notable to our application: continuous outputs, propagative future prediction, and non-linear data. To address these issues we implemented a second hidden layer to our RNN, a look-ahead backpropagation method, and adaptive learning rate. The additional hidden layer and adaptive rate improved both the accuracy and training time of our network. But the look-ahead training method generally produced less accurate results. Under this training condition, the network was trying to optimize for some average of these values rather than focusing on optimizing the single desired value. Thus the network is best trained to produce accurate predictions for the next time-step rather than attempting to be robust to minimize error caused by a poor prediction.

Overall, the network was generally successful at predicting general trends present in the initial input sequence, but did a poor job of predicting new trends. A possible way to address this problem would be to increase the complexity of our model. This could be done by experimenting with additional hidden layers, features and a larger data set. We found that training on all the cities produced lower error than a single city's data, so adding more years of data from more cities may further increase the networks predictive power. Furthermore, a combined method with a MonteCarlo trained network, showed potential as a way to do this. Future work should be done to explore the potential of multiple networks as means to capture different trends.

REFERENCES:

Bontempi, Gianluca, Souhaib Ben Taieb, and Yann-Aël Le Borgne. "Machine learning strategies for time series forecasting." *Business Intelligence*. Springer Berlin Heidelberg, 2013. 62-77.

de Kock, Matthew. "Weather Forecasting Using Bayesian Networks." University of Cape Town. (2008).

Gupta, Madan M., Liang Jin, and Noriyasu Homma. *Static and dynamic neural networks from fundamentals to advanced theory*. New York: Wiley, 2003.

Haerter, Fabricio P., and Haroldo Fraga de Campos Velho. "New approach to applying neural network in nonlinear dynamic model." *Applied Mathematical Modelling* 32.12 (2008): 2621-2633.

Jaeger, Herbert. "A tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach." Fraunhofer Institute for Autonomous Intelligent Systems (AIS), 2013.

Lai, Loi Lei, et al. "Intelligent weather forecast." *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*. Vol. 7. IEEE, 2004.

Rasouli, Kabir, William W. Hsieh, and Alex J. Cannon. "Daily streamflow forecasting by machine learning methods with weather and climate inputs." *Journal of Hydrology* 414 (2012): 284-293.

Shrivastava, Gyanesh, et al. "Application of Artificial Neural Networks in Weather Forecasting: A Comprehensive Literature Review." *International Journal of Computer Applications* 51.18 (2012): 0975-8887.

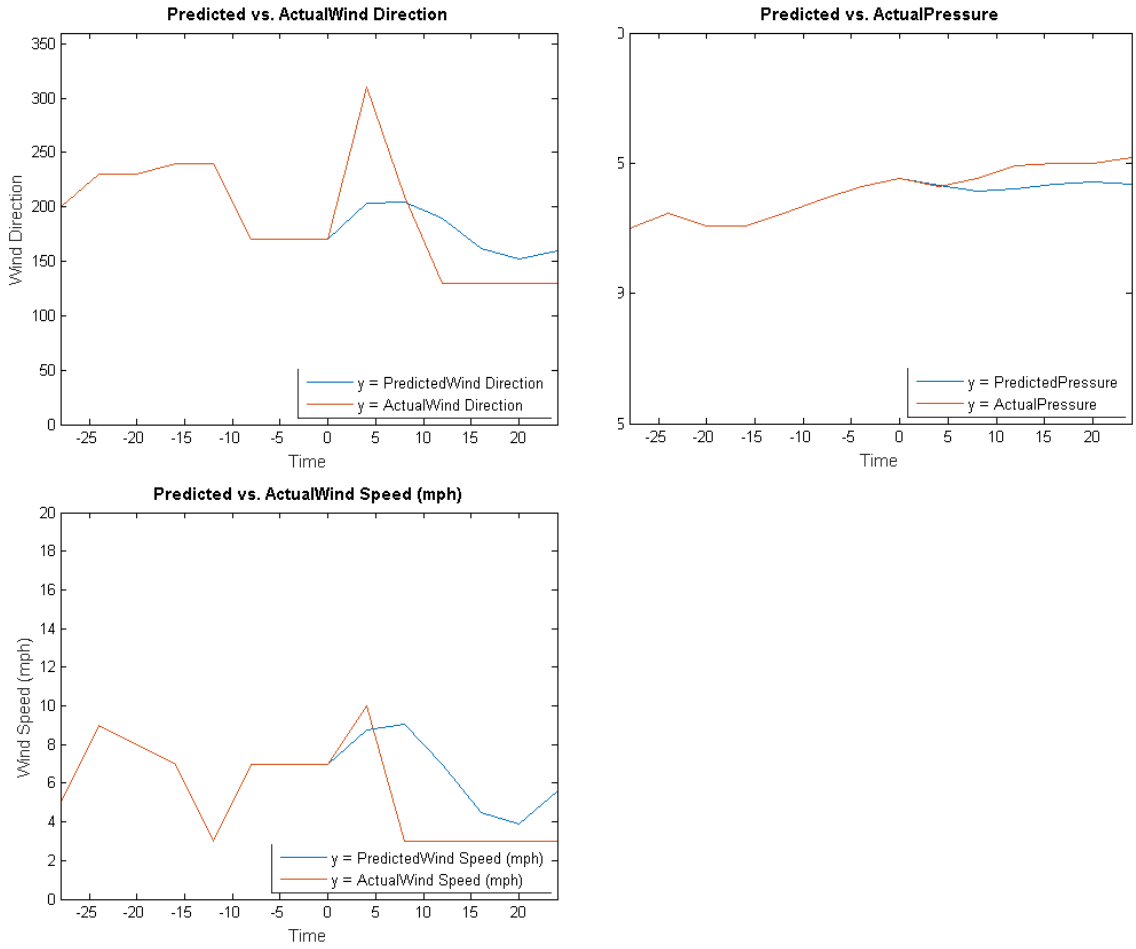
Sinha, N. K., M. M. Gupta, and D. H. Rao. "Dynamic neural networks: An overview." *Industrial Technology 2000. Proceedings of IEEE International Conference on*. Vol. 1. IEEE, 2000.

Soderland, Stephen. "Learning information extraction rules for semi-structured and free text." *Machine learning* 34.1-3 (1999): 233-272.

Williams, John K., et al. "A machine learning approach to finding weather regimes and skillful predictor combinations for short-term storm forecasting." *AMS 6th Conference on Artificial Intelligence Applications to Environmental Science and 13th Conference on Aviation, Range and Aerospace Meteorology*. 2008.

APPENDIX A:

Graphs of remaining features for Lebanon, NH:



Cross Validation Graphs:

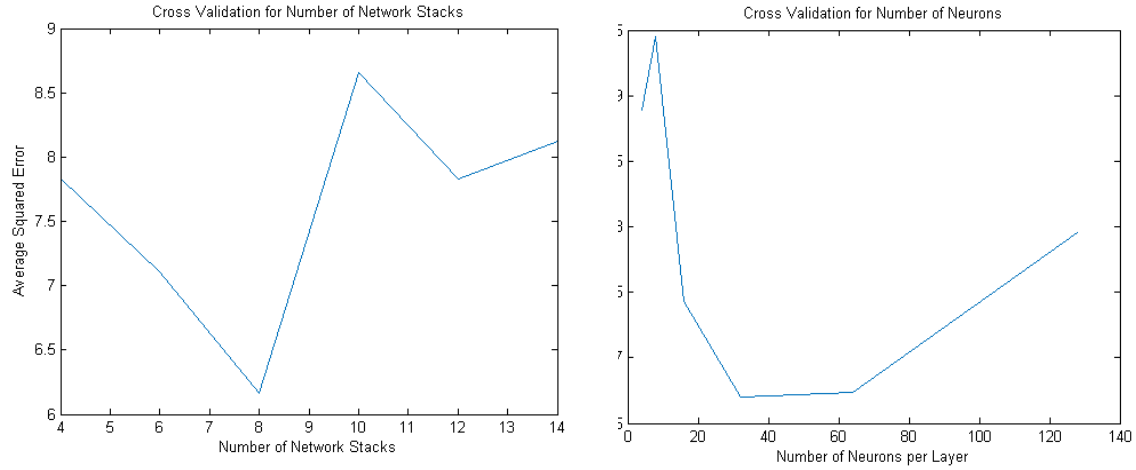


Figure 10: Cross validation to determine the number of network stacks and neurons per hidden layer. From these graphs we chose to use 8 network stacks with 30 neurons per hidden layer.