**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 72 hours in that location. This will offer significant benefits over current methods of weather forecasting that require region-specific models.

**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.

**Algorithms and Results**:

*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

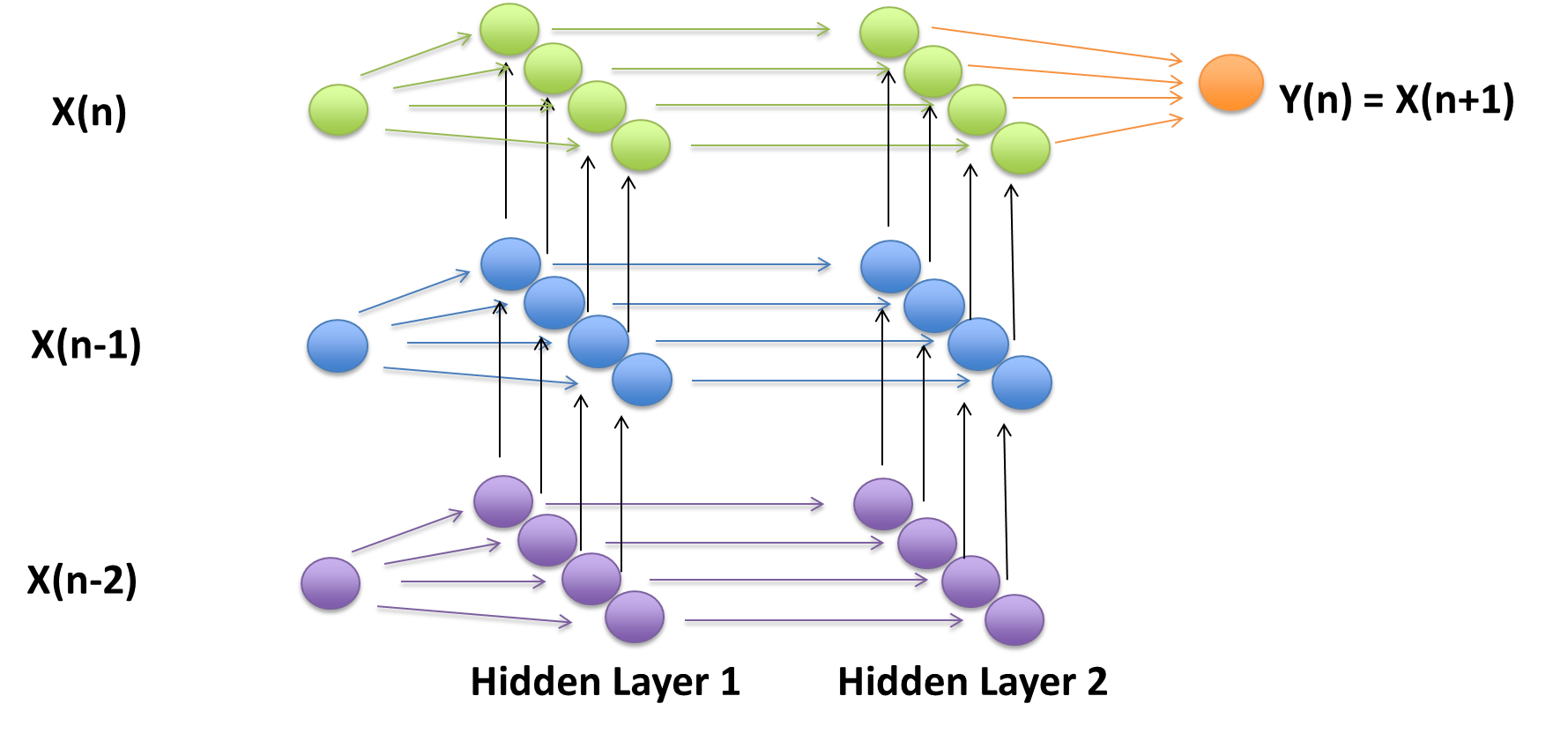
output (y) – the value leaving a neuron

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 attempts to attain better results. We started 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 on this initial network we found that the architecture was insufficient to model the time dependence of our data. The network appeared to at best predict some average of the input data, rather than considering sequences.

Initially, we implemented a recurrent feed-forward network 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.

Next we implemented a simple recurrent neural network with a single hidden layer (using the methods outlined here [CITATION]). For our network, we would need a stack for each input time (3 days by 24 hours = 72 stacks). However, networks typically see little benefit from more than 10 to 20 stacks. To accommodate this, we decided to reduce our input to sample every 4 hours to allow a greater sampling time. 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. This seemed to make more intuitive sense that we would use the most direct form of error rather than propagating it from the previous hidden layer. With the recurrent network, our network still seemed to lack sufficient complexity.



To create a truly, non-linear system, we added a second hidden layer such that each hidden layer connected to the next layer of neurons as well as to the same layer one step in the future. There was minimal literature online about the actual implementation of multiple hidden layers 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.

Our general feed-forward algorithm is as follows (shown for the first hidden layer as an example):

( +

Our general back-propagation algorithm is as follows:

**[Fill in equations here]**

After running a few tests on a small data set, the network seemed to produce insufficient variation in output values, so we removed the squashing function for the hidden layer prior to output. 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 layer. 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. 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.

Most literature implementations utilize the intermediate training values when training the RNN. However, we were only interested in the final output value and the network would not be expected to predict the intermediate outputs correctly. So we modified the backpropagation 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 model for predicting involves feeding predicted values back in as inputs, which meant that any errors would be propagated into later predictions. The algorithm seemed to be sufficient at predicting initial values, but was losing accuracy further into the prediction. We implemented a look-ahead back-propagation method to address this. The algorithm would predict an additional 2-4 time periods beyond what the output in question and then use those values also in the backpropagation. We were attempting to get the algorithm to optimize not only for the current output, but also for future outputs.

As we increased the complexity of the network, it became increasingly unstable and very sensitive to learning rate. To address this, we implemented an adaptive learning rate to help maintain stability while also keeping a reasonable convergence time. 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). We determined convergence by keeping an average of the validation error and stopping when the difference between the current iteration and a preset previous number of iterations was less than a set cut-off.

**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 model it.

[Insert overtain figs]

We trained a separate network using all features to produce weight matrices for each output category. To do the actual prediction, the initial input data is fed through each network and then their outputs are used to create a 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.

[Add error graphs and prediction graphs]

**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 our context. We started with an implementation of a single hidden layer neural network and RNN based on literature. Through experimentation with our model we identified additional problems and implemented what we believed to be the best solution. The following features were distinctive 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 lookahead training and backpropagation method, and adaptive learning rate. We used cross validation to optimize the value of hyperparameters (number of neurons and stacks). Overall, the majority of these modifications improved the algorithm.

[How did the network do??]

[Improvements]

[Open sourcing code?]

**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)

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.

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.

Gupta, Madan M., Liang Jin, and Noriyasu Homma. *Static and dynamic neural networks from*

*fundamentals to advanced theory*. New York: Wiley, 2003.