*Recurrent Neural Networks for Time Series Data*

An Exploration into the Efficacy of the Simple RNN with Gradient Clipping with Respect to Time Series Prediction

Vemuganti, Anand   
Department of Computer Science  
The University of Texas at Dallas

Richardson, USA  
axv190048@utdallas.edu

*Abstract*— Simple recurrent neural networks (RNNs) are an effective way to estimate a multivariate function with temporally sequential features. However, the simple recurrent neural network is prone to many issues such as the exploding and vanishing gradient.

Hyperparameter tuning of a simple recurrent neural network, along with the usage of gradient manipulation techniques and data that lacks noise, are essential to getting the most out of a recurrent neural network. Under certain circumstances, a recurrent neural network with a fewer number of hidden layers may be more effective at predicting short trends than its counterpart with more hidden layers.

# Introduction (*Heading 1*)

The artificial neural network has long been thought as a universal estimator for approximating a function. However, there are some capabilities which a traditional artificial neutral network (ANN) lacks that can be made up for by tweaking the design. The ANN is notorious for its tendency to be unable to properly estimate a function with sequential inputs (typically temporally sequential), and some adjustments must be made to the model to be able to estimate such a function with any degree of accuracy. This is because time series data, or data which is sequentially ordered in a temporal fashion, tends to follow a trend with respect to time. This trend would be unobserved given the anatomy of a traditional neural network, as data from the previous time step would be unable to be of use in the traditional model.

# Recurrent Neural Network

## Traditional ANN vs Recurrent Neural Networks: Anatomy

The anatomy of a traditional neural network is as follows:

Diagram

Description automatically generated

[4] Figure

In such a model, there exists an input layer, several hidden layers with varying sizes, and an output layer. The size of the input layer will be the number of features in the data, and the number of output nodes will equal the number of target variables. A recurrent neural network builds upon this anatomy, but with some changes to permit the network’s support for time steps:

A picture containing athletic game

Description automatically generated

[4] Figure

Firstly, all the layers of the artificial neural network are compressed such that each layer only has one node. This node will take a vector as an input (where the vector’s size is the number of features in the data). Next, all hidden layers of the artificial neural network are consolidated into one hidden layer with a recurrence, i.e., the previous iteration of the network’s output is fed back into the network as another input.

These changes make the schematic of a recurrent neural network simpler than that of a traditional artificial neural network. The weighting scheme is altered as well: rather than each layer of the neural network possessing its own weights matrix, there are only three matrices of weights for the entire network: the input weights, the weights for the recurrent output vectors, and the output weights. This simplifies the space cost of the recurrent neural network drastically, and the structure adds the capability of remembrance with respect to the temporal ordering of the data.

An alternate view of the recurrent neural network’s anatomy is shown below, when the recurrent network is unrolled through time:

A picture containing diagram

Description automatically generated

[4] Figure

where all the weights for the recurrences are identical, and all weights acting every input node are also identical.

## Recurrent Neural Network Training: Backpropagation

A popular way to train an artificial neural network is through gradient descent. This is derived from the partial derivative of the error function of the network with respect to the individual weights in the network, and each weight is updated such that the overall network’s error reduces. The training of a recurrent neural network works in much the same way: a forward pass through the network is made to obtain the error with respect to a training example, and the weights are updated during a backward pass through the network. The weight updates are as follows in the traditional artificial neural network:

If j is an output node:

If j is a hidden layer node:

For all wji and xji where wji represents the weight into node j from node i and xji represents the output of node i going into j:

Where is the activation function used at each node, is the summed input to node j, is the expected value of the output node j, and represents the output of node j, and is the learning rate hyperparameter.

The training for a recurrent neural network is identical to the above method, but with some added measures in place to rectify a potential side effect from the recurrence capability.

# The Exploding / Vanishing Gradient

## The Difficulty of Training Simple RNN

As noted by Sepp Hochreiter in a 1998 publication [5], the simple recurrent neural network is prone to exploding and vanishing gradients. That is, as gradients are propagated back through time in the backpropagation algorithm, they are applied to the same matrix repeatedly. If the gradient is sufficiently small, the gradient will tend to zero as it gets multiplied through the number of layers. However, if the gradient is sufficiently large, it will get extremely large in magnitude as it is propagated through multiple time steps. This causes problems for training simple recurrent neural networks, as the network will either train erratically and uncontrollably (leading to an integer overflow in the programming implementation), or the network will fail to train.

During the programming implementation of the simple recurrent neural network, exploding gradients were an issue. Two methods were used to address the exploding and vanishing gradient problem.

## Addressing the Vanishing Gradient: ReLU

To prevent the vanishing gradient issue that is notorious in simple recurrent neural networks, an activation function was chosen such that the gradient would not tend to zero with backpropagation through the network. The sigmoid and hyperbolic tangent (tanh) activation functions tend to exacerbate the vanishing gradient problem, so the ReLU (Rectified Linear Units) function was used instead. The function is defined as follows:

For the programming implementation of backpropagation training using the ReLU function, the derivative of the function needed to be computed, defined as *step*:

It is to be noted that:

As the Python implementation takes advantage of this fact for the sake of simplicity.

## Addressing the Exploding Gradient: Gradient Clipping

As previously posited, the exploding gradient problem became an issue with the usage of the ReLU function as the activation. The exploding gradient problem is typically solved using an alternative network structure, such as Long Short-Term Memory (LTSM) networks or Gated Recurrent Units (GRU). However, approaches to solve the exploding gradient issue with simple recurrent neural networks include hyperparameter tuning and gradient clipping.

With respect to hyperparameter tuning, the learning rate carries a large amount of weight. Ideally, to lower the probability of an exploding gradient, the learning rate is to be reduced. For model training in the Python implementation, the learning rate was tuned to a level as to avoid encountering an exploding gradient.

Gradient clipping is an approach to cap gradients at a specific magnitude, ensuring that the gradient for a single training example does not grow astronomically large. The Python implementation of the simple recurrent neural network uses two methods to implement gradient clipping.

All updates to vectors and matrices (i.e., the output weights vector and the input/recurrent weights matrices) are normalized before the gradients themselves are updated, using a normalization factor. In the Python implementation, this is a hyperparameter called *error\_norm*. The normalization factor dictates the maximum magnitude of the sum of all elements of the vector or matrix which is being normalized. Normalization in the Python implementation does not occur if the magnitude of the sum of weights in the update vector or matrix is larger than the normalization factor.

In addition to the vector and matrix updates being potentially normalized to *error\_norm*, individual weight updates are also clipped according to a second hyperparameter defined in the training of the model, called *clp\_factor.* This hyperparameter works on the updates to the gradients, but on individual elements of the update vectors/matrices rather than the entire weights vector or matrix. The magnitude of each weight update will be capped at the clp\_factor.

The aforementioned gradient clipping schemes, in combination with a highly-tuned learning rate and the application of the ReLU function, can prevent the model from experiencing the exploding or vanishing gradient problem. However, the downside to these measures is the cost to train the model, both in terms of raw units of time and in terms of the number of epochs required for the model to reach a state of sufficient accuracy.

# Python Implementation

Implementation of the simple recurrent neural network, including data processing for the dataset used, was done in a Python script. No libraries other than those for pre-processing and graphics generation were used for this implementation. The Python implementation makes use of a class called RNN to streamline the training and testing for each individual recurrent neural network.

## Dataset

The original dataset was found on the University of California at Irvine’s Machine Learning Repository. The dataset describes hourly Interstate 94 traffic volume on a stretch of road between Minneapolis and St. Paul. The data includes many features that would likely affect the flow of traffic. Such features likely include the time and the weather at that time, and the target variable would be the traffic volume at that time and with those weather conditions.

The data was preprocessed before being fed to the recurrent neural network so that errors would not occur, with the goal of allowing the network to estimate the function with the greatest possible accuracy. Text data was distilled into numerical data, and columns with a very low variance or those that were not useful were dropped (i.e., holidays, weather description). It was also found that there were missing times in the data, and that the data included ten discrete values for the weather which could be turned into separate binary attributes (as a single hour can have more than one type of weather). During the preprocessing stage, the missing times were filled in using the same details as the previous existing time step, and all duplicate time entries for separate weather patterns were condensed into a single time entry with the binary weather attributes.

## Training Method

During the training phase, it was essential to choose hyperparameters for the model which would prevent the exploding gradient problem from taking effect. Thus, the training was capped to a maximum of 500 epochs, the learning rate was set to 10-11, and the clipping factor and error normalization factor were set to 10-3 and 10-2, respectively. The goal of this hyperparameter selection was to minimize the exploding gradient problem, while still allowing the model to train and learn from all training data which it was presented.

The purpose of training the model in this Python implementation was to observe the correlation between the number of epochs and the model’s training and test error, as well as that between the size of the recurrent neural network and the error. It is important to note that the error metric for this simulation was the root mean squared error (RMSE), defined as:

Where is the expected output of the network for the ith example, is the actual output of the network for the ith example, and n is the number of examples in either the training or test set. Ideally, with more training, the RMSE will decrease.

Two identical networks (with identical starting weights matrices) were trained using nearly the same hyperparameters, although one of these networks had a size of 12 (this would be the number of times + 1 that recurrent outputs were fed back through the network) and the other network used a size of 6. Each of the networks was trained using 100, 200, 300, 400, and 500 epochs, and the training and test error was recorded for every combination of number of epochs and network size.

# Python Implementation: Results and analysis

Using a train-test split of 80% - 20%, the results of the experiment are as follows:

Network of size 12:

Table

Description automatically generatedChart, line chart

Description automatically generated

Network of size 6:

Table

Description automatically generatedChart, line chart

Description automatically generated

Note that the timeframe for the experiment was relatively short, which the seemingly high RMSE even with 500 epochs can be attributed to. Had the timeframe for the experiment been longer, more results would have been able to be obtained with a smaller learning rate, and further techniques to reduce the exploding gradient problem could have been implemented in the recurrent neural network.

The expected trend for epochs vs training error was that the training error would decrease with more epochs, and this is indeed the case for both network sizes tested. However, the test error started to increase for both network sizes after 200 epochs were passed.

This increasing test error with more epochs may be indicative of some overtraining, as an increasing test error is a key characteristic of an overtrained model. However, it is unlikely that the models have been overtrained given the magnitude of the error metric even for the largest number of epochs.

Given the nature of the data, there is likely a large percentage of data that can be described as “noise”: it is possible that a relatively large volume of traffic may not be attributable to the features of the model. There is a high degree of randomness in the amount of traffic in each hour: this is easily affected by outside events such as traffic accidents that the model does not consider.

It is also important to note that a larger network is not necessarily better than a smaller recurrent neural network at estimating a time-series function. The network of size 12 performed better than that of size 6 with respect to the test data only in cases where the number of epochs for which the model was trained was under 300. However, for over 300 epochs, the test RMSE for the smaller network was significantly less than that of the larger network.

This is likely attributable to the irrelevancy of data from twelve hours ago, as opposed to the relative relevancy of more recently obtained data. Although the model with a network size of 12 had access to the more recently obtained information, that from twelve hours ago likely serves only to add more noise into the network, which may distort the network’s ability to predict the traffic volume in a given hour.

Notably, the data in the chosen dataset was short-term (hourly) data, and the predictions for the traffic volume were also hourly. It may be the case that hourly data does not much depend on the previous twelve hours (which may make a smaller recurrent neural network a better choice), but in the long term this is not the case. It is possible that a larger recurrent neural network will be better at forecasting and predicting long-term data, such as data on the scale of months, years, or decades.

##### References

1. A. Géron, “Neural networks and deep learning,” *O'Reilly Online Learning*. [Online]. Available: https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html. [Accessed: 30-Apr-2023].
2. “How to make a table in python?,” *GeeksforGeeks*, 22-Feb-2022. [Online]. Available: https://www.geeksforgeeks.org/how-to-make-a-table-in-python/. [Accessed: 30-Apr-2023].
3. J. F. Shaikh, “Build a recurrent neural network from scratch in python - an essential read for data scientists,” *Analytics Vidhya*, 20-Jun-2022. [Online]. Available: https://www.analyticsvidhya.com/blog/2019/01/fundamentals-deep-learning-recurrent-neural-networks-scratch-python/. [Accessed: 30-Apr-2023].
4. N. McCullum, “The ultimate guide to recurrent neural networks in python,” *freeCodeCamp.org*, 28-Apr-2021. [Online]. Available: https://www.freecodecamp.org/news/the-ultimate-guide-to-recurrent-neural-networks-in-python/. [Accessed: 30-Apr-2023].
5. S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 06, no. 02, pp. 107–116, 1998.