Solar Weather Lab: Predicting Sunspot Activity

Calvin Stahoviak

Feburary 2025

University: University of New Mexico
Course: ST, Advanced Machine Learning
Instructor: Professor Trilce Estrada
Student: Calvin Stahoviak
Assignment: Lab 1, Sequence Learning

1 Introduction

The Sun experiences a period of changing electromagnetic properties, surface activity, and emissions called a solar cycle every 11 years. One of the behaviors monitored during each cycle is the appearance of sunspots – dark areas on the sun's surface that are caused by strong magnetic fields. They are cooler than the surrounding areas, which makes them appear darker on the surface. Sunspot activity can be monitored by counting the total number of sunspots that appear each month. The National Oceanic and Atmospheric Administration (NOAA) is an American scientific and regulatory agency charged with a number of environmental research initiatives. One of these tasks involves recording sunspot activity, which has publicly available data going back to the year 1749.

Using this data, sequence learning models are trained and evaluated in depth. A Recurrent neural network (RNN), long-short term memory (LSTM), gated recurrent unit (GRU), and MAMBA architecture are selected to be compared. The paper is presented in the following order: In section 2 the data pre-processing steps are described. In section 3 the underlying architectures are described. Finally, in section 4 we evaluate and discuss performance differences.

2 Data Pre-Processing

Raw data is collected directly from the NOAA. It includes two features, "time-tag" and "ssn", which is the month and year in which sunspots were counted and the sunspot number respectively. Prior to training our models, a number of data pre-processing steps are taken to shape and clean the dataset.

First and foremost, the raw JSON-formatted data is saved as a pandas DataFrame and dtypes are edited accordingly. Next, the data is normalized using a **MinMaxScaler**. This method scales a feature, x, to a given range, typically $[0, 1]$. The transformation is defined as:

$$x_{\text{scaled}} = a + \frac{(x - x_{\text{min}})(b - a)}{x_{\text{max}} - x_{\text{min}}}.$$

where x is an original feature value, x_{min} is the minimum value in the feature, x_{max} is the maximum value in the feature, and $[a, b]$ is the range of scaling. In our case, the default range of $[0, 1]$ is used. Finally, a dataset is generated taking an arbitrary window of size 48 at each time step. This effectively generates a dataset of 3266 sequences, each 48 time steps long. There is reason to speculate that a window size of 132 would perform better. This correlates to 11 years of data, which is one solar cycle as described in the introduction. It's important to note that all of the series in the dataset must be the same length in order for sequence learning models to function properly. The train and test set is created by taking a 80/20 split of the full dataset.

3 Model Architectures

For educational purposes, the different kinds of architectures are described below. Sequential learning has a major advantage over traditional neural networks in that it is able to learn temporal relationships in the data. A neural network only feeds information in one direction, from input to output. It also makes a critical assumption that the inputs are independent of each other. For sequential data, this is obviously not the case.

After the architectures have been defined, they are trained on the previously generated dataset. All models are trained using the *Adam* optimizer and **mean squared error** (MSE) loss, which can be formally defined:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 ,$$

where y_i is the true target value, $\hat{y}_i$ is the corresponding predicted value, and N is the total number of samples. The use of both of these techniques consistently across all three models ensures a fair comparison in section 4 and section 5.

3.1 Recurrent Neural Network

An RNN is a type of neural network whose architecture has been modified to better predict sequential data. It is one of the simplest forms of sequential learning and consequently suffers from several limitations.

Starting with the traditional neural network architecture, in which information only flows from previous layers to the next, the RNN introduces a new kind of connection from the output of previous inputs. This is visualized in Figure 1, which shows the inputs, outputs, and flow within an RNN cell. This RNN cell effectively replaces the neuron of a traditional neural network. Before the activation function, the input x_t is linearly combined with the input of the previous input, sometimes referred to as the previous time step, h_{t-1}. The output of the activation function is then used to predict on that instance of the sequence, and also stacked onto the next input value, after which the cycle continues.

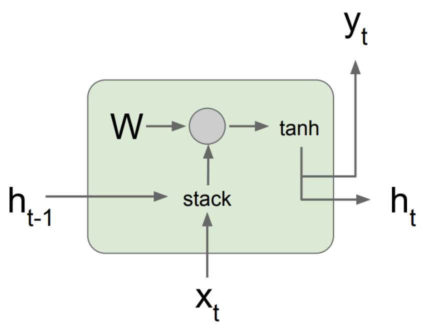
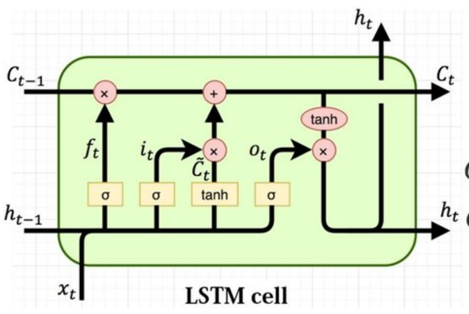


Figure 1: A visualization of the RNN cell where x_t is the current input, h_{t-1} is the previous output, W weights, h_t is the output to be used in the next time step, and y_t is the prediction.

The RNN architecture commonly suffers from vanishing gradients or exploding gradients. Which is when, during backpropagation, the derivative becomes either too small or too large. Another major limitation is that an RNN will tend to forget old information. Therefore, temporally distant relationships in sequential data aren't realized very effectively.

3.2 Long-Short Term Memory

The LSTM architecture improves the RNN architecture in several ways, specifically addressing its greatest limitation, its inability to realize long-term relationships in the data. A visual of the LSTM cell is shown in Figure 2. Key components are the addition of a *cell state*, which is used to capture long-term dependencies, and also *gates*, which are used to carefully update the cell state and form and output. An LSTM cell is made up of a forget gate, an input gate, and an output gate. The forget gate, f_t, outputs a value in the range $[0, 1]$. This value is then multiplied with the previous cell state, effectively forgetting features that are close to 0 in the cell state. The input gate, $i_t \times \tilde{C}_t$, is responsible for updating the cell state with the current input. Finally, the output gate, o_t, is responsible for formulating an output for the current instance of the sequence and further updating the cell state.



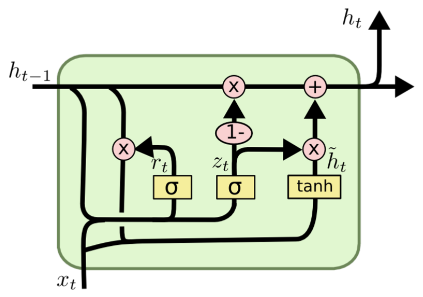
$$i_t = \sigma\big(x_t U^i + h_{t-1} W^i\big)$$
$$f_t = \sigma\big(x_t U^f + h_{t-1} W^f\big)$$
$$o_t = \sigma\big(x_t U^o + h_{t-1} W^o\big)$$
$$\tilde{C}_t = \tanh\big(x_t U^g + h_{t-1} W^g\big)$$
$$C_t = \sigma\big(f_t * C_{t-1} + i_t * \tilde{C}_t\big)$$
$$h_t = \tanh(C_t) * o_t$$

Figure 2: A visualization of the LSTM cell where x_t is the current input, h_{t-1} is the previous output, h_t is the output to be used in the next time step, and C_t is the cell state at each time step. Internally, x_t and h_{t-1} are used to compute C_t and h_t through several carefully structured *gates*.

Although a major improvement on RNNs, the LSTM also has its own limitations. LSTMs introduce several more sets of parameters, dramatically increasing training time. Memory usage also increases in the need to track the cell state over the entire sequence.

3.3 Gated Recurrent Unit

The GRU architecture further improves on the LSTM architecture by addressing its unique limitations. Specifically, GRUs reduces the amount of gates, therefore reducing the amount of parameters and memory needed. It does this by consolidating some of the gates used in LSTMs and combing the cell state with h_t, the output. In a GRU, the typical gates are consolidated into what is called the update gate and the reset gate. The update gate, z_t, is responsible for capturing long-term dependencies and updating h_{t-1}, which acts as the cell state in this case. The reset gate, r_t, is responsible for capturing short-term dependencies and editing h_{t-1} accordingly. Finally the activation of the entire unit, h_t is a summation of h_{t-1} and the output of activation gate, $\tilde{h}_t$. A much more clear visual of this is shown in Figure 3.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 3: A visualization of the GRU cell where x_t is the current input, h_{t-1} is the previous output,h_t is the output to be used in the next time step, z_t is the update gate, and r_t is the reset gate. Internally, the update and reset gates alter h_{t-1} until it is ready to be passed on.

Although an improvement from LSTMs in terms of parameter and memory size, GRUs suffer from their own unique limitations. GRUs are not as expressive since they utilize fewer gates. They also suffer similarly suffer from vanishing and exploding gradients. There also exists many variants of GRUs that seek to find a balance between benefits and limitations.

3.4 MAMBA

The Multi-Head Attention Memory Based Architecture (MAMBA) is dramatically different from the previous architectures discussed. Unlike how GRUs can be thought of as the successor of LSTMs, and LSTMs the successor or RNNs, MAMBA is more easily compared to state-space models (SSM) and transformers. The mamba architecture is made of up MAMBA blocks, which themselves consist of multi-head attention modules and SSM modules. The typical MAMBA block is visualized in Figure 4.

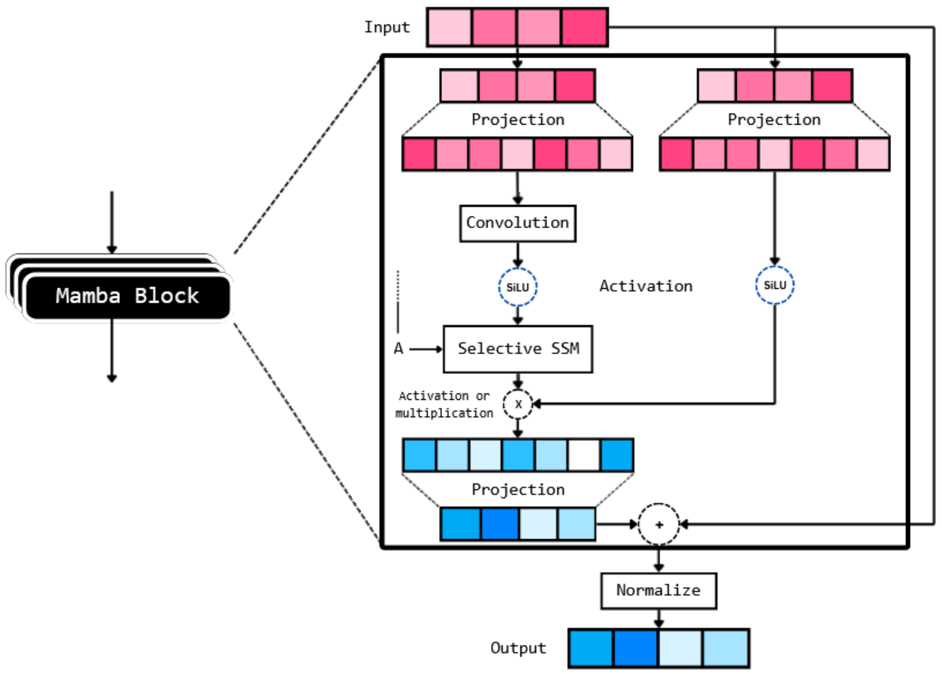


Figure 4: A visualization of a MAMBA block. In each block, an input is is utilized at several points, projected into a high space, ran through convulsions and a selective SSM, eventually being projected back into its original dimension. Several MAMBA blocks form the complete architecture.

MAMBA is only made possible because it was released with CUDA enhanced computation. Its complexity naturally makes complexity much longer.

4 Results

All models are trained on the same dataset for exactly 30 epochs and evaluated below. A manual hyperparameter search is done by changing the following: number of layers, size of the hidden dimension, and learning rate. The results of each of the 28 different configurations is shown in detail in Table 1. The MAMBA API that was used did not have the option to configure the size of the hidden dimension so this was skipped in those cases. The set of RMSE values for each model is also show in box plot form in Figure 9. We can see RNN, LSTM, and GRU all have a similar average and an almost identical optimal RMSE while MAMBA very precisely remains in close range for all of its configurations.

The true values of the test set are shown in Figure 6 alongside the predicted values for the LSTM, RNN, GRU, and MAMBA. We notice a similar behavior in the first three models, the LSTM, RNN, and GRU, in which they all seem to predict relatively close to the true value with the exception of extreme peaks and valleys. These three models also seem to perform very close to each other as well. MAMBA, on the other hand, seems to predict almost exactly despite having a similar RMSE (more thoughts on this in section 5)

This visual similarity in predictions is further corroborated by the scores presented in Figure 7. We see a best root mean squared error (RMSE) of 0.0571 for LSTM, 0.0574 for RNN, 0.0585 for GRU, and 0.0633 for MAMBA. Although there is only a difference 0.0062 in RMSE between the highest (MAMBA) and lowest (RNN) performing models, that amounts to about a 10.299% difference. We see the best performance come from the RNN model with only a slight advantage.

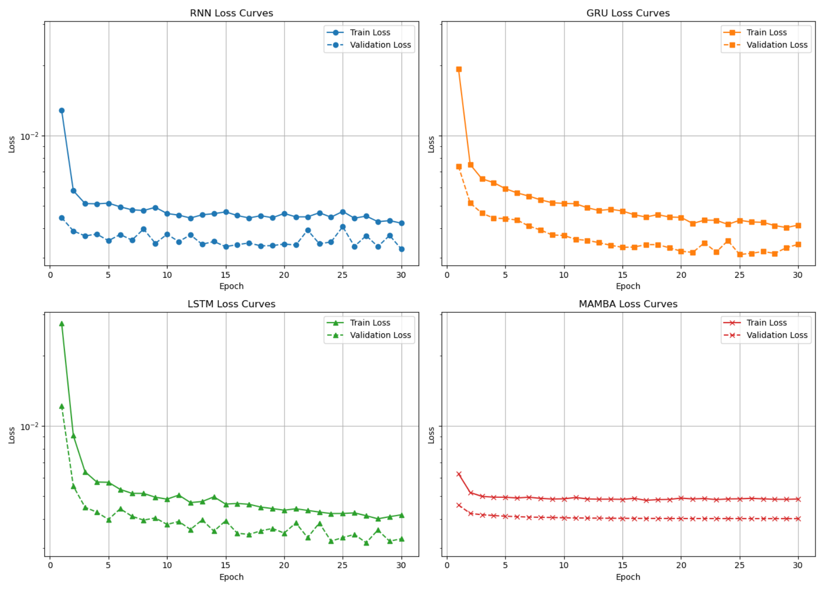


Figure 5: The training loss and test loss curves are plotted at every epoch for all four architectures, RNN, LSTM, GRU, and MAMBA.

5 Discussion

The behavior and training patterns for the RNN, LSTM, and GRU models are all very similar. We see a sharp decline in both training and validation loss early on, followed by a plateau after about 10 epochs. This implies that we are not overfitting and likely hitting the maximum performance for our dataset and architecture. The closeness of RMSE scores collected from our hyperparameter search also suggests that these architectures are likely incapable of improving any further on this dataset. Although LSTMs and GRUs are commonly viewed as successors to RNNs, we observe our RNN architecture with the highest performance here. This suggests that the dataset is too limited or the signal coming from our singular feature, sunspot number, is not complex enough for LSTMs or GRUs to take full advantage of their more complicated architecture. The average time to train and inference time on the test set, visualized in Figure 8, leads us to a similar conclusion. For example, we see GRU with the longest training time, opposite of what is expected from an architecture with less parameters than LSTM and MAMBA. I believe if we had used a more complicated dataset then there would a clear difference in performance between these models.

On the other hand MAMBA exhibits several interesting behaviors which I ultimately chalk-up to unknowns coming from the specific MAMBA API that was used. In Figure 6 we notice MAMBA tracking the true values almost exactly but with a delay. This figure suggests that MAMBA is simply returning the last value of the input window as its prediction with out learning anything novel. However, the loss curves seems to show that MAMBA is learning very slightly. Possibly, MAMBAS exceptional architecture is allowing it to learn the datasets signal within only a few epochs. MAMBA, also had a significantly higher inference time and training time, which are suspect of an error in training. It is hard not to speculate in this situation without a more complete knowledge of this specific MAMBA implementation. It is very possible that I misused this API.

6 Conclusion

In conclusion, the RNN, LSTM, and GRU architectures all have very similar optimal performances. Our best performance, by a very slight margin, comes from the RNN architecture. MAMBA exhibited very interesting results and was not entirely conclusive, something that may have been error in the API or misuse of the API by my hand. Overall I would be very interested in seeing results from a more complex dataset. This code is publicly available at `github.com/cjstahoviak/Solar-Weather-Lab-Predicting-Sunspot-Activity`.

A Supplementary Material

This section provides additional figures and tables.

A.1 Additional Figures

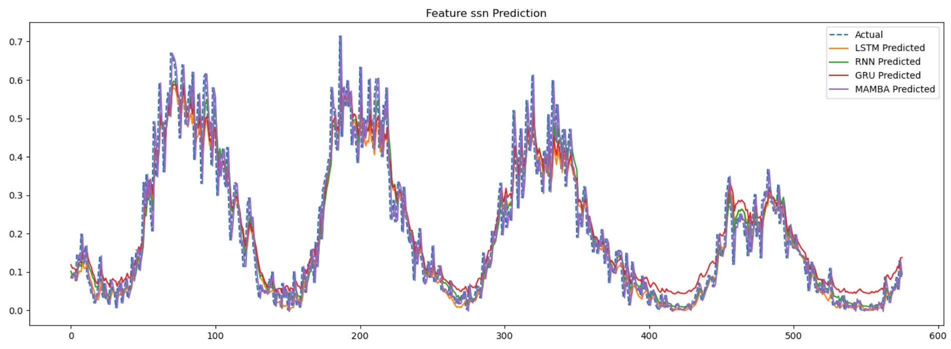


Figure 6: Feature prediction on the test set for each architecture. The true values are show in dotted blue, while predicted values are color coded.

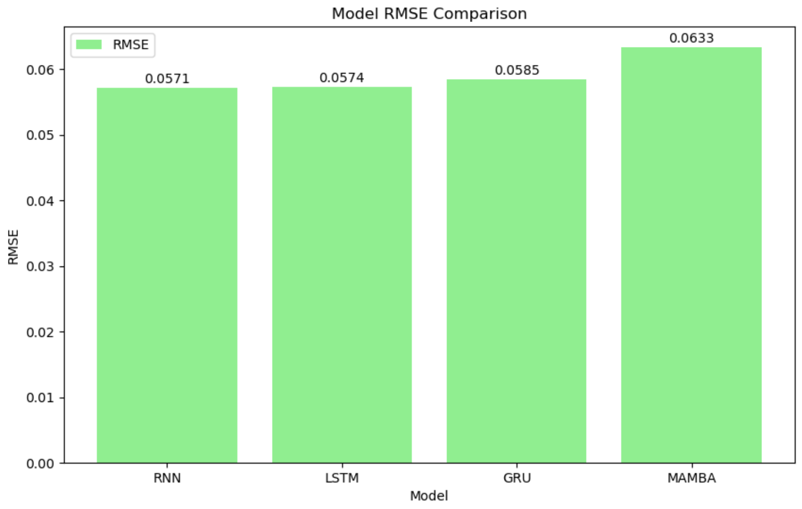


Figure 7: The root mean squared error (RMSE) is compared between the optimal model of each architecture. We see no model in particular with a major difference in performance.

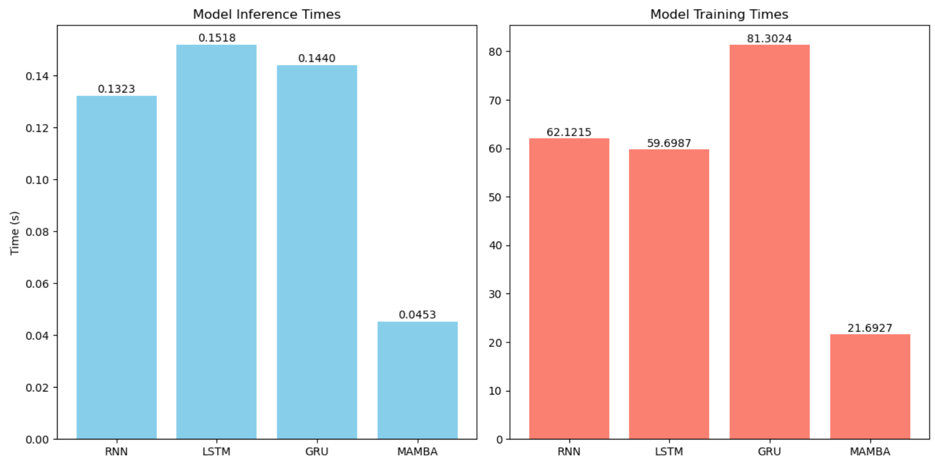


Figure 8: The inference times for the optimal model of each architecture are plotted as bar graphs. Training time is over a single training sessions while inference time is an average of 100 inference sessions.

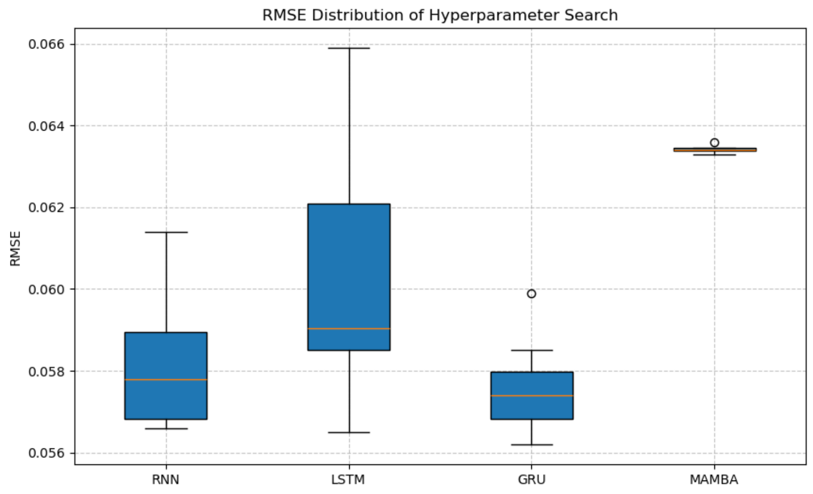


Figure 9: The root mean squared error (RMSE) of all models tested during a hyper parameter search.

A.2 Additional Tables

Model	num_layers	hidden_dim	learning_rate	RMSE
RNN	2	32	0.001	0.0569
RNN	2	32	0.0005	0.0566
RNN	2	64	0.001	0.0614
RNN	2	64	0.0005	0.0581
RNN	3	32	0.001	0.0575
RNN	3	32	0.0005	0.0589
RNN	3	64	0.001	0.0566
RNN	3	64	0.0005	0.0591
LSTM	2	32	0.001	0.0587
LSTM	2	32	0.0005	0.0621
LSTM	2	64	0.001	0.0582
LSTM	2	64	0.0005	0.0586
LSTM	3	32	0.001	0.0594
LSTM	3	32	0.0005	0.0659
LSTM	3	64	0.001	0.0565
LSTM	3	64	0.0005	0.0621
GRU	2	32	0.001	0.0570
GRU	2	32	0.0005	0.0599
GRU	2	64	0.001	0.0578
GRU	2	64	0.0005	0.0570
GRU	3	32	0.001	0.0562
GRU	3	32	0.0005	0.0585
GRU	3	64	0.001	0.0578
GRU	3	64	0.0005	0.0563
MAMBA	2	–	0.001	0.0633
MAMBA	2	–	0.0005	0.0634
MAMBA	3	–	0.001	0.0636
MAMBA	3	–	0.0005	0.0634

Table 1: RMSE results for RNN, LSTM, GRU, and MAMBA models with varying hyperparameters. The MAMBA API being borrowed did not have a hidden dimension configuration variable available.