# Sequential Analysis of Epileptic Seizures Using a LSTM Classifier

Randy Suarez Rodes
University of Texas at Dallas
Richardson, TX
rxs179030@UTDallas.edu

Derek Carpenter
University of Texas at Dallas
Richardson, TX
dxc190017@UTDallas.edu

***The following report illustrates an analysis of Epileptic Seizure data using a Recurrent Neural Network with a Long Short-Term memory model. The model has been improved using various optimization techniques such as an addition of AMSGrad and AdaGrad optimizers along with the stacking of model layers. The model will be trained by a sequential dataset containing relevant features about the brain activity before and after epileptic seizures. Finally, the model will be used to predict status of patient.***

## I. Introduction

The use of neural networks for predictive classification is becoming an industry standard. With the wide availability of data, we can start to build these models to help us predict things that would otherwise be difficult to see. One type of neural network is a Recurrent Neural Network (RNN) that is used specifically for sequential and time series prediction. One practical example is the use of an RNN to predict what will happen to the stock market, as we have a significant amount of sequential data telling us what has happened in the past.

We will be performing a similar task, using a RNN to classify the epileptic status of a patient. Sequential data has been obtained from sensors wired to a patient's brain. This data classifies the status of the patient, whether the subject is having a epileptic seizure or not, at each second of the experiment. For our implementation we will use various improvements, such as the use of Long Short-Term Memory (LSTM) model and the addition of the AMSGrad & AdaGrad optimizers along with stacking LSTM layers.

## II. Approach

First, we must preprocess the data so that it can be properly understood by our model. To achieve this, we convert epileptic classes to a hot-one encoding and shape our feature sets into number of sequences, length of sequence, number of attributes.

Next, we must build our model, we have decided on an LSTM. Our LSTM utilizes series of states and gates that produce a set of sequential predictions.

This model is trained using a traditional neural network approach of performing epochs where there is a forward pass to calculate error and a backward pass to adjust the weights. Our goal is to optimize this LSTM using additional features, we will explore the use of an AdaGrad and AMSGrad solvers to increase our model performance. We also will implement a stacked LSTM to further increase performance. Finally, we will run a series of experiments fine tuning the hyper parameters of the model to achieve the best model for epileptic prediction.

## III. Dataset Overview and Preprocessing

To train our LSTM we will use a dataset obtained from the UCI Machine Learning Repository. The dataset is titled "Epileptic Seizure Recognition Data Set" [5] The dataset contains a set of time-series readings from a study performed on the brain activity of 500 subjects. Each entity is one second of time for a particular patient and contains the classification of the subject. The dataset contains 5 classes, however for our purposes, we will convert this to a binary class {Seizure, No Seizure}. On this approach the dataset is imbalanced, being 2300 observations the presence of Seizure vs 9200 No Seizure. Below is a brief summary of the dataset.

TABLE I. Dataset Description

| Epileptic Seizure Dataset | Description |
|---|---|
| Dataset Type | Time Series/Sequential |
| Dataset Output | Binary Classification |
| Number of Attributes | 178 |
| Number of Unique Entities | 11,500 |
| Number of Correlated Entities | 500 |

## IV. Implementing a RNN using a LSTM Model

A RNN is a neural network that is used for sequential time series prediction. It is a collection of states with some associated time t, where each state depends on all the previous states. We will first analyze the structure and theory of a vanilla RNN, then show how to improve it using an LSTM.

### A. *Vanilla RNN Structure*

With a RNN, the data is input at its corresponding time. For example, the 4th item of a sequential data set will be input at t = 4. Figure 1 shows the structure of a vanilla RNN [1]. The computation of the green state squares is determined by the method used. For our project we will use a LSTM.

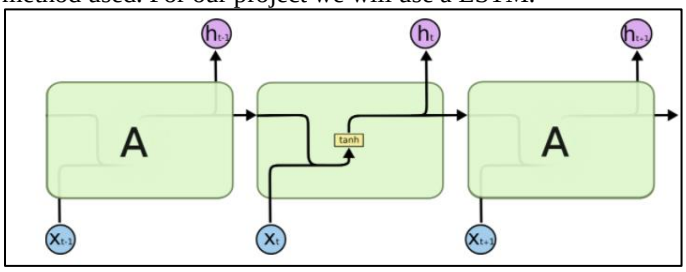


Fig. 1. Vanilla RNN Structure

In this vanilla example we have the following equations:

$$Z_t = X_t.W_{t,x} + W_{t,t-1}.A_{t-1} + Bias_t \quad (1)$$

$$A_t = \tanh(Z_t) \quad (2)$$

$$h_t = \tanh(W_{h_t}.A_t + Bias_{h_t}) \quad (3)$$

Where $Z_t$ is state A at time t, $A_t$ is the output of state A at time t, and $h_t$ is the prediction at time t. Here the methods of computation are like a traditional neural network, where we multiply inputs by weights and send the net through some activation function. In this case a tanh activation function is used.

This model has one issue, vanishing gradients. When we backpropagate this model, each weight update at state t will become less impactful as the back propagation continues down the state sequence. To solve this, we will implement a Long Short-Term Memory (LSTM) model to help keep only relevant data as we build our model.

### *B. LSTM Structure and Implimentation*

To overcome this issue with vanishing gradients, which essentially gives the model only short-term memory, we will use a LSTM model to keep relevant long-term information. This is done by setting up a system of gates that chooses which information is important to keep, and that which can be forgotten. Figure 2 below shows a graphical representation of an LSTM [1].

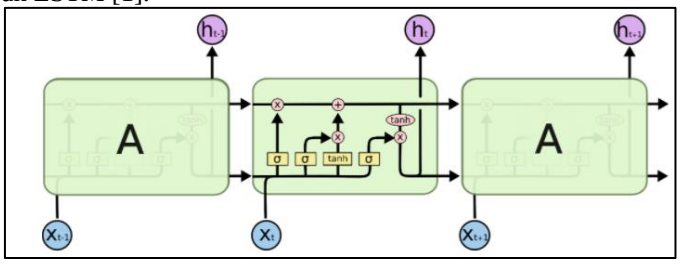


Fig. 2. Vanilla LSTM Structure

#### *The Forward Pass*

Looking at the top arrow of the state, we can see this as a flow of information across the state. The gates are neural nets that decide what information should be input to the flow as the LSTM propagates forward. This ensures only relevant information is sent to the next state. The forward pass of an LSTM is similar to an ANN, where we compute each state and send that state information to the next state.

Here we will compute 4 different gates that will be stored as part of a particular time state $Z_t$ as seen from equations 4 -7 [6].

$$a_t = \tanh(W_{xc}.X_t + U_{hc}.h_{t-1} + b_c \quad (4)$$

$$i_t = \sigma(W_{xi}.X_t + U_{hi}.h_{t-1} + b_i) \quad (5)$$

$$f_t = \sigma(W_{xf}.X_t + U_{hf}.h_{t-1} + b_f) \quad (6)$$

$$o_t = \sigma(W_{xo}.X_t + U_{ho}.h_{t-1} + b_o) \quad (7)$$

Where $a_t$ is the input modulation gate, $i_t$ is the input gate, $f_t$ is the forget gate, and $o_t$ is the output gate. These four gates make up our particular time state, Zt, which we will compute for every time step in our series. We have 23 seconds of data for each patient, 1 data point per second so we will have 23 states Z. On our implementation and results we set 23 as the default length of the sequence, although this parameter can be customized.

At each state we will also have a cell state, which we will use to control information from state to state. Here we compute the cell by Equation 8 below.

$$c_t = f_t * c_{t-1} + i_t * a_t \quad (8)$$

Where $c_t$ is the cell at time t and $c_{t-1}$ is the previous state cell. We can see how it is being affected by the forget gate that we mentioned earlier. This is a critical part in the LSTM that helps with long term memory. We are now eliminating irrelevant information using this forget gate.

The last piece of information needed for our state in the forward pass is the hidden state, which is defined by the below equation.

$$h_t = \tanh(c_t) * o_t \quad (9)$$

Finally, our prediction at state t can be calculated by equation 10.

$$Y_t = softmax(W_{out} * h_t + b_{out}) \quad (10)$$

Where softmax is an activation function that can be chosen selectively from other functions such as sigmoid. We have found favorable results using softmax.

#### *Backpropagation*

With the forward pass completed, we must now backpropagate the model to update the weights based on the error obtained after the forward pass. We can apply our weight update equations to adjust the weights as we back propagate the LSTM. First, we must calculate our deltas at each time step. We can do this using equations 11-18.

$$dv_t = Y_t - y_t \quad (11)$$

Where $y_t$ is the true value (one hot encoded in classes) from the training data and $Y_t$ is the predicted probabilities from Equation 10.

$$dh_t = Wo_t * dv_t \quad (12)$$

$$dc_t = dh_t * o_t(1 - \tanh^2(c_t)) \quad (13)$$

$$da_t = dc_t * i_t * \frac{d}{do}\tanh(a_t) \quad (14)$$

$$di_t = dc_t * a_t * \mathrm{d}\sigma(\mathrm{i}_t) \quad (15)$$

$$df_t = dc_t * c_t * \mathrm{d}\sigma(\mathrm{f}_t) \quad (16)$$

$$do_t = dh_t * \tanh(state_{t+1}) * d\sigma(\mathrm{o}_t) \quad (17)$$

$$da_t = i_t * \mathrm{dc}_t \quad (18)$$

The gradient which respect of each weight will be an accumulation of the gradients at each time step. From these deltas, we can calculate the accumulation of gradients using Equation 19-22[6].

$$dWxo = \sum_{t=1}^{T} o_t(1-o_t)\,\mathrm{x}_t * \mathrm{do}_t \quad (19)$$

$$dWxi = \sum_{t=1}^{T} i_t(1-i_t)\,\mathrm{x}_t * \mathrm{di}_t \quad (20)$$

$$dWxf = \sum_{t=1}^{T} f_t(1-f_t)\,\mathrm{x}_t * \mathrm{df}_t \quad (21)$$

$$dWxc = \sum_{t=1}^{T} (1-a_t)\,\mathrm{x}_t * \mathrm{da}_t \quad (22)$$

Similarly, we can calculate U updates using equations 23-26[6].

$$dUho = \sum_{t=1}^{T} o_t(1-o_t)\,\mathrm{h}_{t-1} * \mathrm{do}_t \quad (23)$$

$$dUhi = \sum_{t=1}^{T} i_t(1-i_t)\,\mathrm{h}_{t-1} * \mathrm{di}_t \quad (24)$$

$$dUhf = \sum_{t=1}^{T} f_t(1-f_t)\,\mathrm{h}_{t-1} * \mathrm{df}_t \quad (25)$$

$$dUhc = \sum_{t=1}^{T} (1-a_t^2)\,\mathrm{h}_{t-1} * \mathrm{da}_t \quad (26)$$

The output layer gradients can be obtained using Equations 27-28[6]. (Here . represents dot product)

$$db\_output = \sum_{t=1}^{T} dv_t \quad (27)$$

$$dW_{output} = \sum_{t=1}^{T} dv_t\,.\,transpose(\mathrm{h}_t) \quad (28)$$

Now that we have all the necessary information to update the weight as we back propagate the LSTM and we can adjust our model weights. To do this we will use the standard update equation using the learning rate and an AdaGrad or AMSGrad solver. Equation 30-34 shows these applications.

$$W = W - learning\ rate * \ dW \quad (30)$$

$$U = U - \ learning\ rate * \ dU \quad (31)$$

$$b = b - \ learning\ rate * \ db \quad (32)$$

$$W_{out} = W_{out} - \ learning\ rate * \ dW\_output \quad (33)$$

$$b_{out} = b_{out} - \ learning\ rate \ * \ db\_output \quad (34)$$

Where each of the deltas is sent through an AdaGrad or AMSGrad solver which is discussed in section C.

This completes one epoch of our LSTM model.

### C. *AdaGrad and AMSGrad Optimizer*

To improve our model, we have added AdaGrad and AMSGrad Optimizers. These additions improve the process of updating the weights, and is more stable than the vanilla update and avoiding local minimums.

#### *AdaGrad Solver*

For further improvements we have added the AdaGrad solver to our model as defined by the following equations.

$$new\_weight = w_t - \frac{\alpha}{\sqrt{v_t+\epsilon}} * \frac{dL}{dw_t} \quad (35)$$

$$v_t = v_{t-1} + \left[\frac{dL}{dw_t}\right]^2 \quad (36)$$

TABLE II. ADAGRAD PARAMETERS

| Parameter | Value |
|---|---|
| $\epsilon$ | 10^-7 |
| $\alpha$ | 0.4 |

#### *AMSGrad Solver*

The AMSGrad solver uses a combination of momentum and RMSprop by using the exponential moving average, $v$, and ensuring that each time $v$ is updated it is larger than $v_{t-1}$.[7]

$$new\_weight = w_t - \frac{\alpha}{\sqrt{v_t+\epsilon}} * m_t \quad (37)$$

$$v_{t,hat} = \max\left(v_{t-1,hat}, v_t\right) \quad (38)$$

$$m_t = \beta_1 * m_{t-1} + (1-\beta_1) * \frac{dL}{dw_t} \quad (39)$$

$$v_t = \beta_2 * v_{t-1} + (1-\beta_2) * \left[\frac{dL}{dw_t}\right]^2 \quad (40)$$

Where $\beta_1, \beta_2, \alpha, eps$ are all hyper parameters that are tuned to optimize model performance. We have chosen the following values for our parameters for a single layer LSTM.

TABLE III. AMSGRAD PARAMETERS

| Parameter | Value |
|---|---|
| $\epsilon$ | 10^-7 |
| $\alpha$ | 0.02 |
| $\beta_1$ | 0.9 |
| $\beta_2$ | 0.999 |

### D. *Stacking LSTM Layers*

To further improve our model, we implemented a stacked LSTM model. Figure 3 shows a representation of how the model implements stacked layers [8]. In the depth dimension of the stacked LSTM network, the output of the $(L-1)^{th}$ LSTM layer at time $t$ is $h^{L-1}{}_t$ . This output is treated as the input, $x^{L}{}_t$ of the $L^{th}$ layer. In the backward process the role of Equation (11)

is played by a gradient incoming from the L[th] layer towards the L-1[th] layer can be obtained using Equation (41).

$$dx_t^L = \sum_{t=1}^{T} W^L \, . dgates^L \qquad (41)$$

Where $W^L$ are the weights on the layer L of the inputs for the inputs $x_t^L$ as in Equations (4-7) and $dgates^L$ are the deltas (15-18) on L[th] layer.

Our implementation fixes a base layer to receive the sequences inputs, and offers the option to create additional layers by passing a list of the number of neurons of the subsequent upper layers.

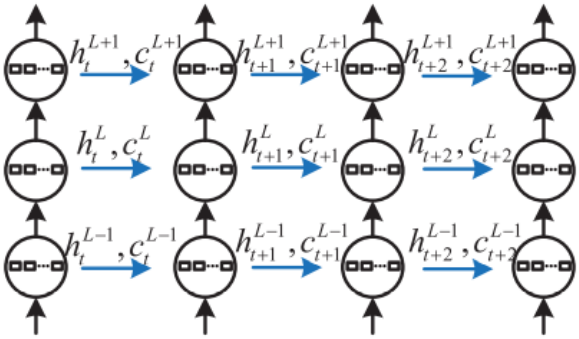

Fig. 3. LSTM with stacked layer communication.

## V. Analysis and Results

To evaluate our results, we performed a series of experiments using different hyperparameters and solvers to determine performance. We varied the learning rate, solver, and number of stacked layers. For each experiment we reported the accuracy (a), cross entropy error (CEL), precision (p), recall (r) and f score (f-s). On the layers 1 represents our base layer receiving 178 attributes and [8,4] represents stacked layers of 8 and 4 neurons respectively.

TABLE IV. Model Experiment Results

| Exp # | Parameter | | | Data Set | a | CEL | f-s | p | r |
|---|---|---|---|---|---|---|---|---|---|
| | *Layer* | *Solver* | *η* | | | | | | |
| **1** | 1 | Vanilla | 0.1 | Train | 0.933 | 0.158 | 0.815 | 0.901 | 0.744 |
| | | | | Test | 0.923 | **0.194** | 0.791 | **0.887** | 0.714 |
| **2** | 1 | AdaGrad | 0.4 | Train | 0.947 | 0.139 | 0.858 | 0.912 | 0.810 |
| | | | | Test | 0.913 | 0.256 | 0.762 | 0.866 | 0.681 |
| **3** | 1 | AMSGrad | 0.02 | Train | 0.942 | 0.173 | 0.845 | 0.906 | 0.791 |
| | | | | Test | 0.915 | 0.229 | 0.768 | 0.869 | 0.689 |
| **4** | 1+ [8,4] | Vanilla | 0.03 | Train | 0.964 | 0.123 | 0.907 | 0.928 | 0.886 |
| | | | | Test | **0.929** | 0.237 | **0.820** | 0.857 | **0.785** |
| **5** | 1+ [8,4] | AdaGrad | 0.3 | Train | 0.954 | 0.154 | 0.878 | 0.948 | 0.816 |
| | | | | Test | 0.887 | 0.369 | 0.678 | 0.811 | 0.582 |
| **6** | 1+ [8,4] | AMSGrad | 0.007 | Train | 0.956 | 0.184 | 0.882 | 0.946 | 0.826 |
| | | | | Test | 0.908 | 0.261 | 0.748 | 0.854 | 0.665 |

For each of these experiments we have plotted the log-loss vs epochs as the model is trained for both the test and training test. From Figures 5-7 it can be seen that we hit a minimum around 30 epochs with about 10-20% error for most of the models and that the error decay fairly quickly. A good f-score on each experiment is evidence of the ability of our model to classify imbalanced data.

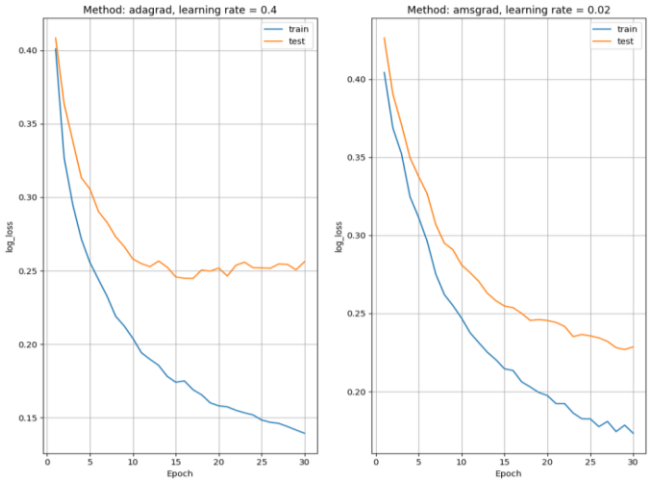


Fig. 4. Non-stacked LSTM performance AdaGrad(left) & AMSGrad(right)

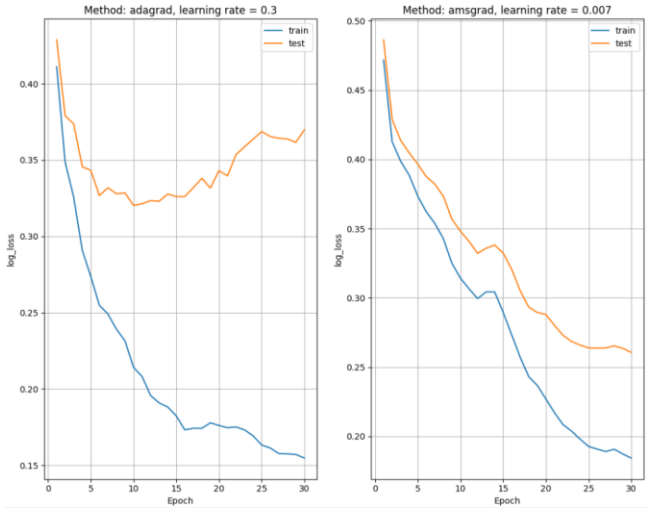


Fig. 5. Stacked LSTM performance AdaGrad(left) & AMSGrad(right)

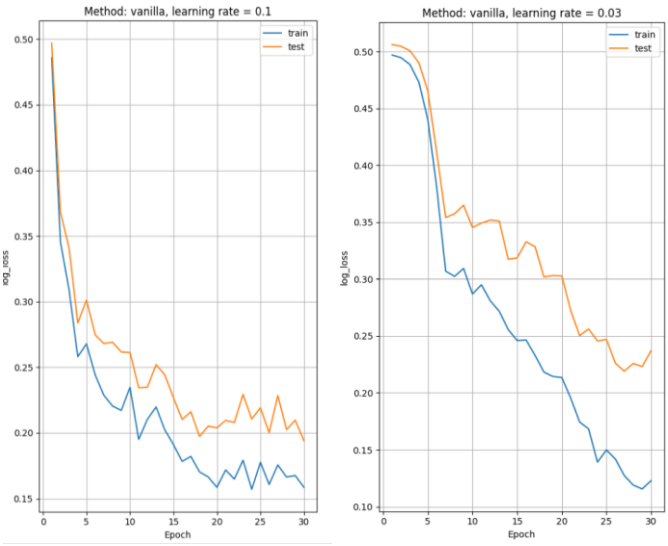


Fig. 6. Vanilla LSTM Performance of non-stacked(left) & stacked(right)

## VI. Conclusion

To close our investigation, we present the following conclusions:

- Based on our experiments our model correctly predicts epileptic seizure over 90% of the time using the vanilla LSTM and AMSGrad optimizer on both, single and stacked LSTM models in just a few number of epochs.

- On all experiments we obtained a high f-score, reflecting the ability of our model to correctly classify an imbalanced data set.

- We have implemented a versatile and powerful LSTM network, able to be customized with several stacked layers and number of neurons.


## Acknowledgment

Thank you to Professor Nagar for a great semester.


## References


[1] Gomez, A. (2016). Backpropogating an LSTM: A Numerical Example.J. https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9

[2] Jeong, J. (2019). The Most Intuitive and Easiest Guide for Recurrent Neural Network. https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-recurrent-neural-network-873c29da73c7.

[3] Olah, C. (2015). *Understanding LSTM Networks.* http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[4] Seo, J. D. (2018). *Propagation on simple RNN/LSTM.* https://towardsdatascience.com/back-to-basics-deriving-back-propagation-on-simple-rnn-lstm-feat-aidan-gomez-c7f286ba973d.

[5] Wu, Q. (2017). Epileptic Seizure Recognition.

[6] Chen, G. (2018). A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation. https://arxiv.org/pdf/1610.02583.pd

[7] 10 Stochastic Gradient Descent Optimisation Algorithms + Cheat Sheet | by Raimi Karim | Towards Data Science

[8] Yu, Yong. A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures. neco_a_01199 (mitpressjournals.org)