Explainable AI for Sequence Processing Neural Networks

Gokul S

Student, Department of Computer Science and Engineering, College of Engineering Guindy, 12 Sardar Patel Road, Anna University, Guindy, Chennai, Tamilnadu-600025

Prof. Bapi Raju Surampudi

Professor, Cognitive Science Lab, International Institute of Information Technology, Gachibowli, Hyderabad, Telangana - 500032

Table of Contents

APPENDICES

Explainable AI for Sequence Processing Neural Networks

Gokul S

Student, Department of Computer Science and Engineering, College of Engineering Guindy, 12 Sardar Patel Road, Anna University, Guindy, Chennai, Tamilnadu-600025

Prof. Bapi Raju Surampudi

Professor, Cognitive Science Lab, International Institute of Information Technology, Gachibowli, Hyderabad, Telangana - 500032

Abstract

The project focusses on studying the different approaches to tackle the problem of explaining Sequential Pattern Processing models. A progressive approach is being followed which starts off with the analysis of the architecture and working of the parent network , a Simple RNN (Elman Network). After training and analysing the network's performance, we shift our focus to the actual objective which is to explain models by uncovering the patterns internally learned by the neural network. Toward this goal, we tried to analyse the kind of knowledge obtained by Simple RNN by training it on a corpus of Artificial Grammar Learning using Reber Grammar strings and studying the hidden unit activation patterns. After training (with validation), the model happened to perform extremely well with respect to the test set by correctly predicting the next two possible transitions at every given node. This means that our network can now be approximately treated as an acceptor for the underlying Finite State Automata governing the Reber Grammar. Then, we shifted gears and tried to study the overall grammar learned by the network at a deeper fundamental level. We first retrieved the hidden unit activation patterns corresponding to the entire test set at each timestep after which we performed Agglomerative (hierarchical) clustering on them using Euclidean distance as the metric. The resulting dendrogram happened to cluster each intermediate string based on the current node it has reached. This helps us build trust in our model and reassures us that it has indeed perfectly learned the underlying grammar. In effect , we interpreted the internal hidden unit representations in order to understand machine decisions in a more concrete manner. We

then explore other sequence processing architectures such as Echo State Network (ESN) and Long Short term Memory (LSTM) in a similar manner.

1 INTRODUCTION

Sequence processing neural networks are being viewed at the grass-roots level in order to understand their working at a deeper fundamental level. In order to work on this task, three different types of neural networks are selected and are applied to work on a common Artificial Grammar Learning (AGL) task. Their performances are compared and their internal state space dynamics are analyzed using various techniques in order to obtain a better perception of their working. Specifically, we investigate the Reber Grammar (1976) by means of formal analysis and network simulations and outline an approach to grammar extraction based on the network state-space dynamics.

1.1 Background

The inducement to study sequence processing networks and their underlying grammar comes from the fact that all natural processes have a temporal nature and even if they don't, any problem can be transformed from a non-temporal form to its corresponding temporal form. Language, for instance, is an inherently temporal phenomenon. When we comprehend and produce spoken language, we are processing continuous input streams of indefinite length. And even when dealing with written text we normally process it sequentially, even though we in principle have arbitrary access to all the elements at once. The temporal nature of language is reflected in the metaphors we use; we talk of the flow of conversations, news feeds, and twitter streams, all of which call out the notion that language is a sequence that unfolds in time. This temporal nature is also reflected in the algorithms we use to process language. In contrast, the machine learning approaches applied to sentiment analysis and other classification tasks do not have this temporal nature. These approaches have simultaneous access to all aspects of their input. This is certainly true of feedforward neural networks, including their application to neural language models. Such networks employ fixed-size input vectors with associated weights to capture all relevant aspects of an example at once. This makes it difficult to deal with sequences of varying length, and they fail to capture important temporal aspects of language.

1.2 Problem Definition

A Simple Recurrent Neural Network (SRN) or Elman Network, an Echo State Network (ESN) and a Long Short Term Memory (LSTM) Network are employed to precisely learn the underlying Finite State Automata (FSA) of the Reber Grammar (1976). In short, different types of Sequence Processing Neural Networks are made to work on an Artificial Grammar Learning

(AGL) task. The training is performed on a set of positive (valid) Reber strings and our objective is to work towards the retrieval of the underlying grammar.

1.3 Relevance

Sequential pattern recognition and Grammar Learning (GL) are characteristics of human performance. It is well accepted that neurobiological and functional brain constraints have import antimplications for the characteristics of the language faculty (Hauser et al., 2002). It seems natural to assume that the brain is finite with respect to its memory organization. Now, if one assumes that the brain implements a classical model of language, then it follows immediately from the assumption of a finite memory organization that this model can be implemented in a FSA. In short, a finite-state machine will behave as a Turing Machine as long as the memory limitations are not encountered. One can take the view that classical cognitive models of language are approximate abstract descriptions of brain properties. A complementary perspective is offered by network models of language processing. If one assumes that the brain sustains some level of noise or does not utilize infinite precision processing (either of these assumptions seems necessary for physically realizability), it can be argued on qualitative grounds that 'natural language', viewed as a neurobiological system, might be well-approximated by finite-state behavior (Petersson, 2005a).

2 LITERATURE REVIEW

2.1 Reber Grammar

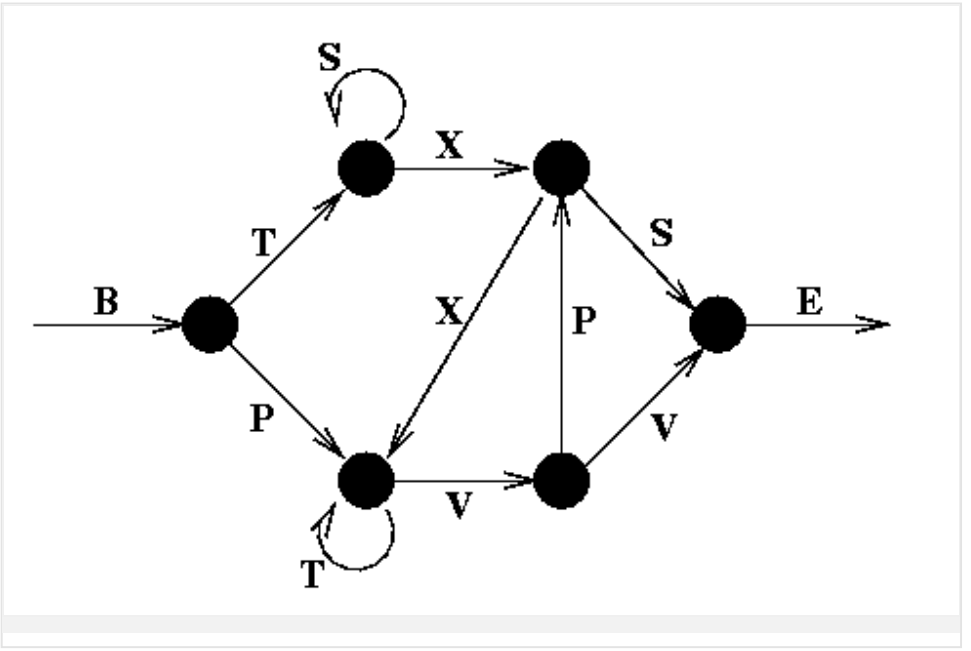

Fig 1 Reber Grammar Transition Diagram

The Reber grammar(1976) is an example of a simple finite state grammar and it has served as a benchmark for Grammar Learning (GL) tasks. We start at B, and move from one node to the next, adding the symbols we pass to our string as we go. When we reach the final E, we stop. If there are two paths we can take, e.g. after T we can go to either S or X, we randomly choose one (with equal probability). In this manner we can generate an infinite number of strings which belong to the rather peculiar Reber language. Note that an **S** can follow a **T**, but only if the immediately preceding symbol was a **B**. A **V** or a **T** can follow a **T**, but only if the symbol immediately preceding it was either a **T** or a **P**. In order to know what symbol sequences are legal, therefore, any system which recognizes Reber strings must have some form of "memory", which can use not only the current input, but also fairly recent history in making a decision.Our task is to predict the next symbol in a sequence generated by the above grammar. If we regard 'prediction' as the task of the network, then this task is not perfectly solvable. E.g: after seeing BTS, the next symbol is not predictable, but the only two candidates are S and X, so both should be predicted in equal measure, and no other should be predicted. In a different context, e.g. after seeing BTXS, the only candidate symbol is an E. The system needs to know more than just the last symbol (S in this example). However, if we regard the job of the

network to be one of grammar induction, that is, to learn the transition probabilities in the grammar, then it can be well learned. We need to be aware that although we are training it to do predictions, we really want it to learn the underlying grammar. It is worth noting that the probabilities of the grammar are only imperfectly realized in the finite set of training data that is available.

2.2 Simple Recurrent Network (SRN) or Elman Network

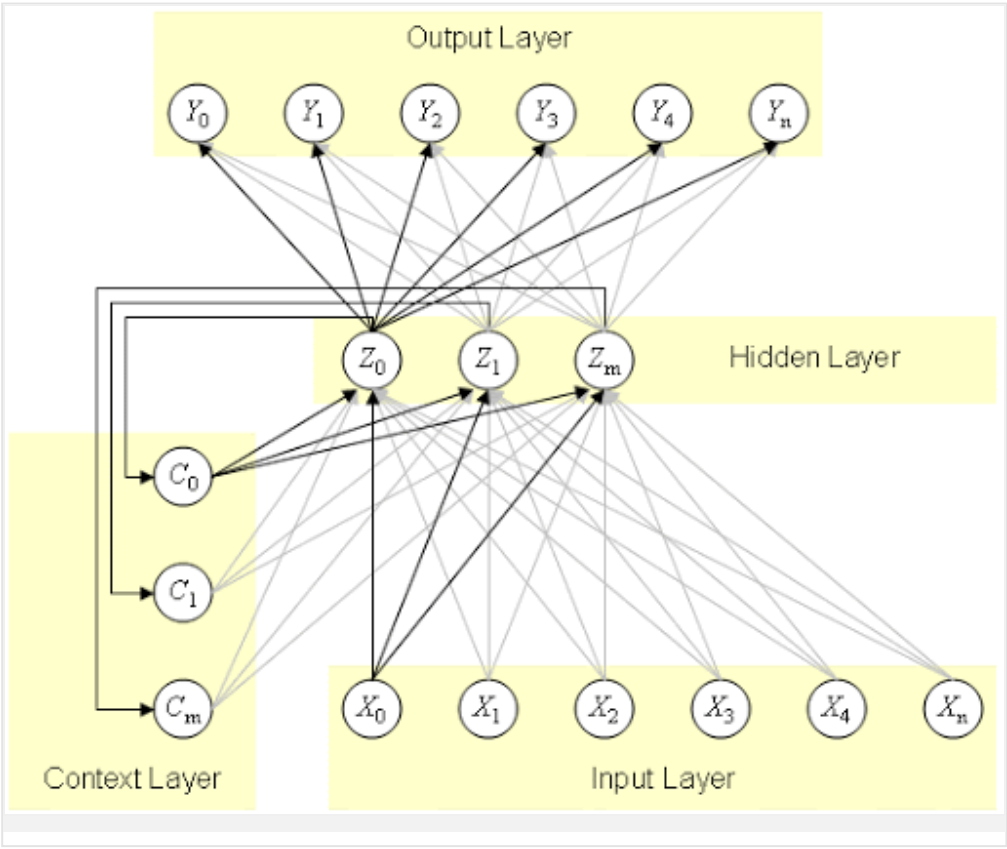

Fig 2 SRN Architecture

An SRN (Elman ,1991) is a three-layer, feed-forward back propagation network with a simple (yet powerful) addition that one of the two parts of the input to the network is the pattern of activation over the network's own hidden units at the previous time step. The Elman network is basically recurrent and designed to learn sequential or time-varying patterns. In this way, algorithms could recognize and predict learned series of values or events. The primary interest in this architecture was for language processing algorithms, but its useful for just about anything involving sequences. The definition of a context revolves around prior internal states, and hence we've added a layer of "context units" to a standard feedforward net. In this

way, the states of the hidden units could be fed back into the hidden units during the next stage of input.

2.3 Echo State Network (ESN)

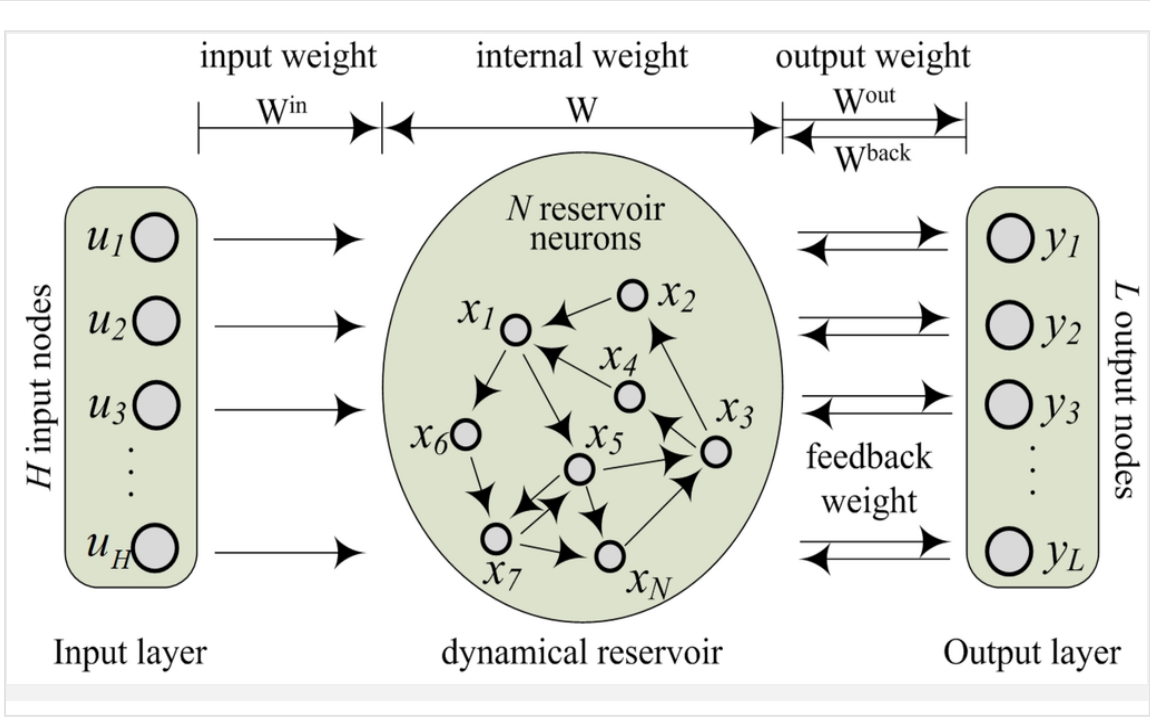

Fig 3 ESN Architecture

Reservoir computing is an extension of neural networks in which the input signal is connected to a fixed (non-trainable) and random dynamical system (the reservoir), thus creating a higher dimension representation (embedding). This embedding is then connected to the desired output via trainable units. Echo state network (ESN) is a type of Recurrent Neural Network, part of the reservoir computing framework, with some particularities. The weights between the input the hidden layer (the 'reservoir') and also the weights of the 'reservoir' are randomly assigned and not trainable. The weights of the output neurons (the 'readout' layer) are trainable and can be learned so that the network can reproduce specific temporal patterns. The hidden layer (or the 'reservoir') is very sparsely connected (typically < 10% connectivity). This reservoir architecture creates a recurrent non-linear embedding of the input which can be then connected to the desired output and these final weights will be trainable.

2.4 Long Short Term Memory (LSTM) Network:

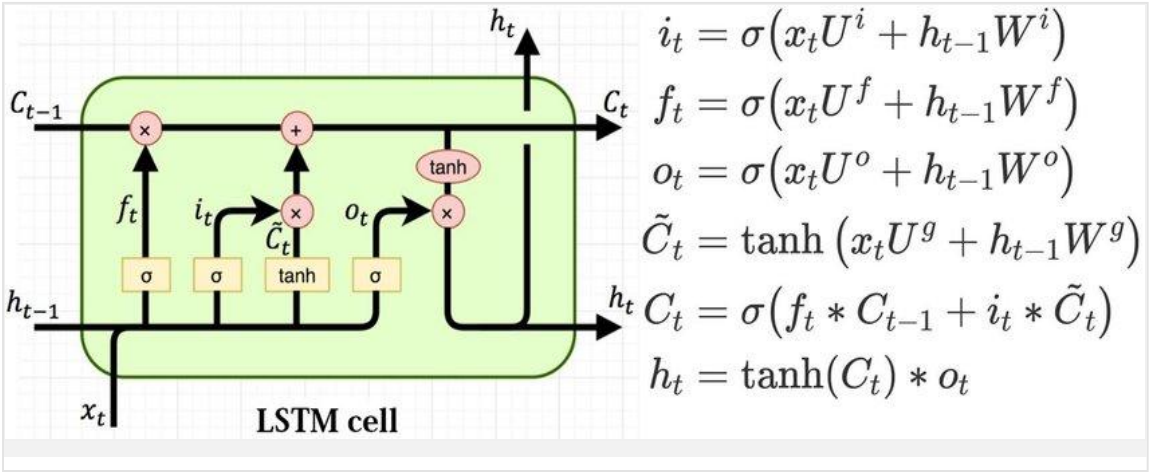

Fig 4 LSTM Cell Architecture

Long short-term memory (LSTM) networks (Hochreiter and Schmidhuber, 1997) divide the context management problem into two sub-problems: removing information no longer needed from the context, and adding information likely to be needed for later decision making. The key to solving both problems is to learn how to manage this context rather than hard-coding a strategy into the architecture. LSTMs accomplish this by first adding an explicit context layer to the architecture (in addition to the usual recurrent hidden layer), and through the use of specialized neural units that make use of *gates* to control the flow of information into and out of the units that comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, and previous hidden layer, and previous context layers. The gates in an LSTM share a common design pattern; each consists of a feed- forward layer, followed by a sigmoid activation function, followed by a pointwise multiplication with the layer being gated. The choice of the sigmoid as the activation function arises from its tendency to push its outputs to either 0 or 1. Combining this with a pointwise multiplication has an effect similar to that of a binary mask. Values in the layer being gated that align with values near 1 in the mask are passed through nearly unchanged; values corresponding to lower values are essentially erased.

3 Methods and Approaches

3.1 Simple Recurrent Network (SRN) Learning Reber Grammar

Network Architecture:

Input Units - 7

Context Units – 3

Hidden Units – 3

Output Units - 7

Working:

Both the input units and context units activate the hidden units and then the hidden units feed forward to activate the output units. The hidden units also feedback to activate the context units. This constitutes the forward activation. Depending on the task, there may or may not be a learning phase in this time cycle. If so, the output is compared with a teacher input and backpropagation of error is used to incrementally adjust connection strengths. Recurrent connections are fixed at 1.0 and are not subject to adjustment. At the next time step **t+1,** the above sequence is repeated. This time the context units contain values which are exactly the hidden unit values at time **t**. In this way, the context units provide the network with memory.

In case of our task (to learn Reber strings) , due to the existence of 7 different alphabets in our grammar, each input element is one-hot encoded to a 7 bit vector which is then given as input to the SRN sequentially at each timestep. Context units are initialised to the value 0. Training of the SRN is performed on a corpus of 2000 valid Reber strings. In order for BPTT to be effective for our task , we use Binary Cross Entropy loss function and the Adam Optimizer. Training is performed for 50 epochs (with validation and early stopping to prevent overfitting)

We then shift our focus to the internal state representation of the SRN. We look at the internal state ,i.e, the vector of hidden unit activation patterns, at each timestep, and try to retrieve the underlying grammar. The entire state matrix is subjected to Agglomerative (Hierarchical) clustering using Euclidean distance as the metric. The resulting dendrogram is then observed to derive inferences about the grammar learned by the network.

3.2 Echo State Network (ESN) learning Reber Grammar

Network Architecture:

A traditional 3-layered Echo State Network is created. Since each character of the Reber Grammar is one-hot encoded to 7 bits , the input layer of the network has seven nodes and the output layer has seven nodes. The Reservoir layer here has 400 neurons which help us in creating a loosely coupled dynamical reservoir. Win and W are the input and the reservoir weight matrices respectively which are sparse(sparsity = 25%) and randomly fixed. Wout is the output weight matrix which is initialized to zeroes and is completely trainable.

Dimensions:

i) Win – (1 + inSize) * resize

ii) W – (resize * resize)

iii) Wout – (1 + inSize + resize) * outsize

Working:

The ESN is trained for 30000 timesteps on valid Reber strings with a Spectral Radius of 0.95 (for Echo State Property to be valid) and a leak rate of 0.3. It is then run in the “Predictive” test mode for 2000 timesteps to measure its performance.

While training, the inputs are fed to the reservoir sequentially and a teacher output is applied to the output units. The reservoir states are captured over time and stored. Once all of the training inputs are applied, a simple application of ridge regression is used between the captured reservoir states and the target outputs. These output weights are then incorporated into the existing network and used for novel inputs. The idea is that the sparse random connections in the reservoir allow previous states to "echo" even after they have passed, so that if the network receives a novel input that is similar to something it trained on, the dynamics in the reservoir will start to follow the activation trajectory appropriate for the input and in that way can provide a matching signal to what it trained on, and if it is well-trained it will be able to generalize from what it has already seen, following activation trajectories that would make sense given the input signal driving the reservoir. The advantage of this approach is in the incredibly simple training procedure since most of the weights are assigned only once and at random. Yet they are able to capture complex dynamics over time and are able to model properties of dynamical systems. It is important to note this proven fact that the Echo State Property of an ESN may only be given if the Spectral Radius of the

reservoir weight matrix is smaller or equal than 1. The Echo State Property means the system forgets its inputs after a limited amount of time. This property is necessary for an ESN to not explode in activity and to be able to learn.

We then concentrate on the reservoir state matrix in order to derive inferences about the learning procedure of the ESN. Since clustering on 400-dimensional data is computationally expensive , we first resort to dimensionality reduction. Principal Component Analysis (PCA) is performed on the reservoir state matrix to reduce the dimensions. After this, Agglomerative (Hierarchical) Clustering is performed on the reservoir state matrix with Euclidean distance as the metric.

3.3 Long Short Term Memory (LSTM) Learning Reber Grammar

Network Architecture:

Input Units - 7

Hidden LSTM Units – 3

Output Units – 7

Working:

The first gate to consider is the forget gate. The purpose of this gate is to delete information from the context that is no longer needed. The forget gate computes a weighted sum of the previous state's hidden layer and the current input and passes that through a sigmoid. This mask is then multiplied by the context vector to remove the information from context that is no longer required. The next task is compute the actual information we need to extract from the previous hidden state and current inputs — the same basic computation we've been using for all our recurrent networks. Next, we generate the mask for the add gate to select the information to add to the current context. Next, we add this to the modified context vector to get our new context vector. The final gate we'll use is the output gate which is used to decide what information is required for the current hidden state (as opposed to what information needs to be preserved for future decisions). Given the appropriate weights for the various gates, an LSTM accepts as input the context layer, and hidden layer from the previous time step, along with the current input vector. It then generates updated context and hidden vectors as output.

In case of our task (to learn Reber strings) , due to the existence of 7 different alphabets in our grammar, each input element is one-hot encoded to a 7 bit vector which is then given as input to the LSTM network sequentially at each timestep. Training is performed on a corpus of 2000 valid Reber strings. In order for BPTT to be effective for our task , we use Binary Cross Entropy loss function and the Adam Optimizer. Training is performed for 50 epochs (with validation and early stopping to prevent overfitting).

In order to understand the effect of the number of LSTM units used in the hidden layer on the performance of the network and the clustering analysis , the same network is also ran using 7 LSTM units and 14 LSTM units in the hidden layer respectively and the results are analysed.We then shift our focus to the internal state representation of the LSTM network. We look at the internal state ,i.e, the vector of hidden unit activation patterns (h_t), at each timestep, and try to retrieve the underlying grammar. The entire state matrix is subjected to Agglomerative (Hierarchical) clustering using Euclidean distance as the metric. The resulting dendrogram is then observed to derive inferences about the grammar learned by the network.

4 Results

4.1 Simple Recurrent Neural Network

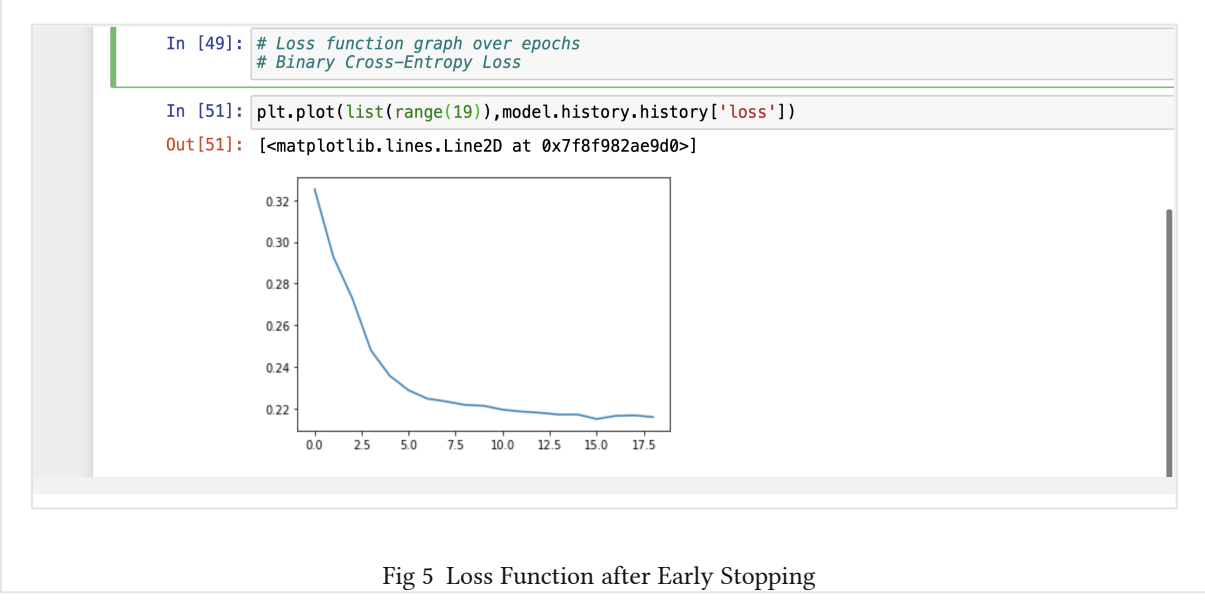

Fig 5 Loss Function after Early Stopping

After training the network for 50 epochs (with validation and early stopping to prevent overfitting) , the average accuracy received is close to 97.5% which is significant taking into consideration how small in size the training corpus was.

```
In [40]: # Analysis of a string

In [41]: print(list(chars))
         print((z.round(decimals=2)[0]))

         ['B', 'E', 'P', 'S', 'T', 'V', 'X']
         [[0.   0.   0.38 0.   0.58 0.03 0.  ]
          [0.   0.   0.   0.   0.72 0.28 0.  ]
          [0.   0.   0.   0.   0.67 0.33 0.  ]
          [0.   0.   0.   0.   0.67 0.33 0.  ]
          [0.   0.   0.   0.   0.68 0.32 0.  ]
          [0.   0.   0.   0.   0.67 0.33 0.  ]
          [0.   0.   0.   0.   0.66 0.34 0.  ]
          [0.   0.   0.98 0.   0.   0.02 0.  ]
          [0.   0.   0.   0.01 0.   0.   0.99]
          [0.   0.   0.   0.   0.61 0.39 0.  ]
          [0.   0.   0.9  0.   0.   0.1  0.  ]
          [0.   0.   0.   0.05 0.   0.   0.95]
          [0.   0.   0.   0.   0.6  0.4  0.  ]
          [0.   0.   0.   0.   0.58 0.42 0.  ]
          [0.   0.   0.69 0.   0.   0.31 0.  ]
          [0.   1.   0.   0.   0.   0.   0.  ]
          [0.15 0.12 0.17 0.15 0.18 0.1  0.12]
          [0.15 0.12 0.17 0.15 0.18 0.1  0.13]
          [0.15 0.13 0.16 0.15 0.17 0.1  0.13]
          [0.15 0.14 0.14 0.16 0.16 0.1  0.14]]

In [65]: temp(X_test[0])

Out[65]: ['BPTTTTTVPXVPXTVVE']

In [66]: y_test[0]

Out[66]: array([[0, 0, 1, 0, 0, 0, 0],
                [0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 0, 1, 0],
                [0, 0, 1, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 1],
                [0, 0, 0, 0, 0, 1, 0],
                [0, 0, 1, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 1],
                [0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 0, 1, 0],
                [0, 0, 0, 0, 0, 1, 0],
                [0, 1, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0]], dtype=int32)
```

Fig 6 Predicted values for a Test String

Looking at the dendrogram after clustering analysis, it is clear that clustering has happened node wise and that each cluster corresponds to the current node in the grammar reached by the string inputted so far.It is clear that the activation patterns are grouped according to the different nodes in the finite state grammar. All patterns that produce a similar prediction are grouped together, independently of the current letter. With only 3 hidden units, representational resources are so scarce that backpropagation forces the network to develop representations that yield a prediction based on the basis of the current node alone, ignoring contributions from the path.

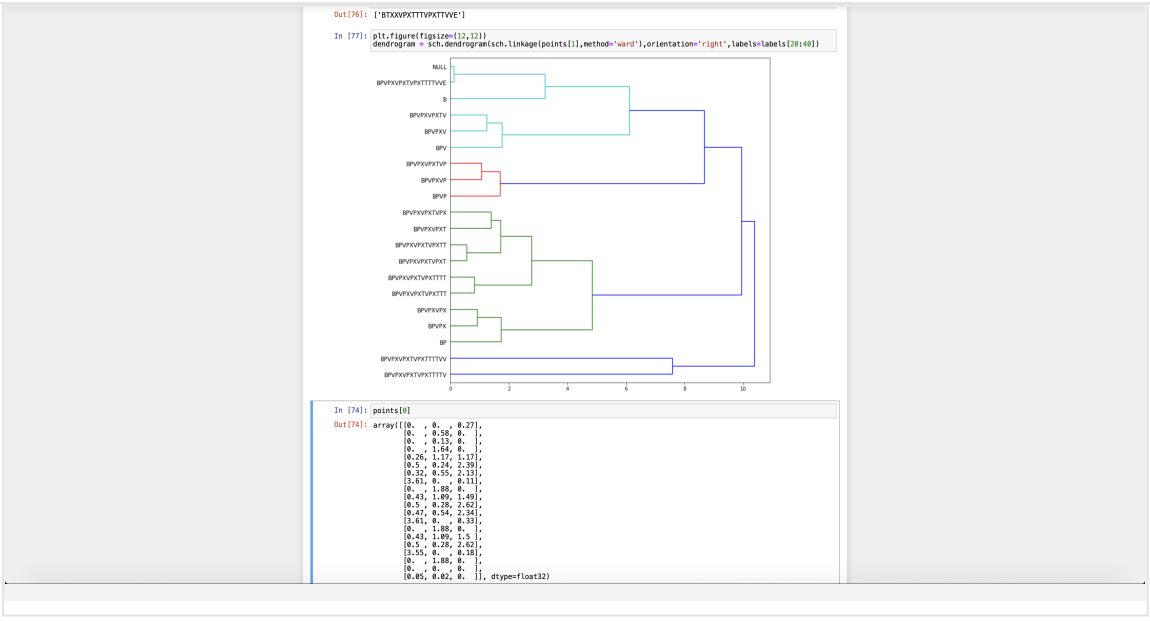

Fig 7 Hierarchical Clustering Dendrogram

4.2 Echo State Network (ESN)

Looking at the performance of the network after training for 30000 timesteps, we can see that it performs extremely well on the test set. Accuracy turns out to be well over 99.9% which is really satisfactory. Note that a prediction is considered correct if given an input, it is able to predict the next two possible transitions in the FSA governing the Reber grammar.

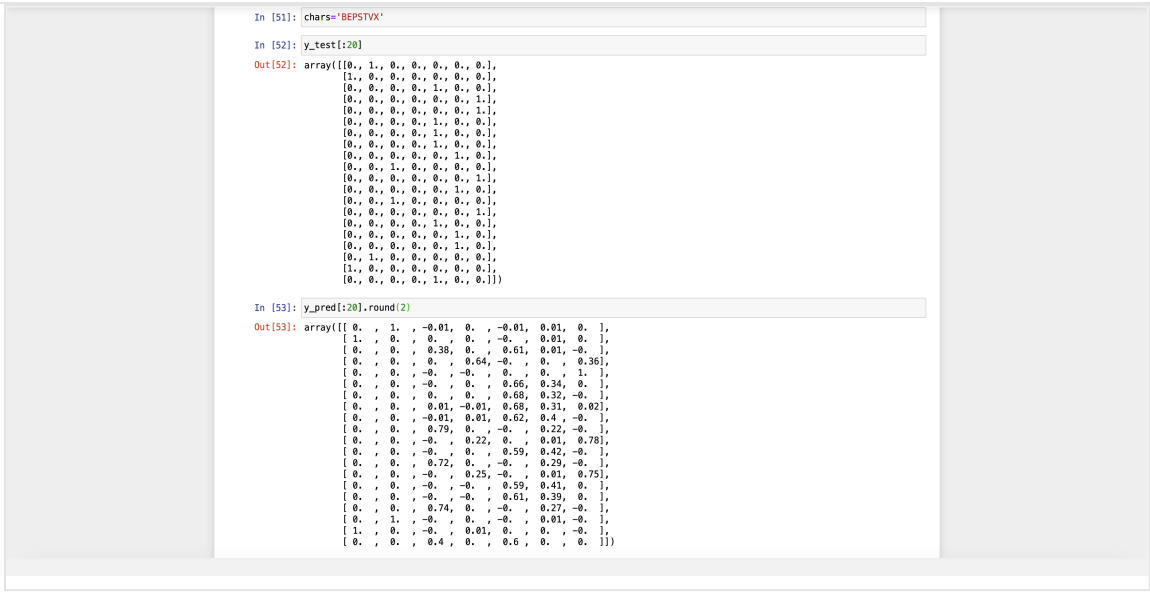

Fig 8 Predicted values for a Test String

Now we retrieve the hidden unit activation patterns (reservoir state matrix) for the entire training period and perform PCA to reduce dimensions. It turns out that PCA happens to be really satisfactory and the entire variance of the reservoir state matrix can be explained by just 6-10 dimensions (from 400 dimensions).

Fig 9 Principal Component Analysis of Reservoir Neuron Activations

Following this, it is subjected to Agglomerative (Hierarchical) clustering using Euclidean distance as the metric. The results turn out to be poor and clustering doesn't happen node wise as we expected. The fact that when we apply PCA on the reservoir state matrix, we are able to explain all the variance in between 6-10 dimensions suggests that the subspace by itself is an encoding of the rules of the Reber Grammar. But to our suprprise, this hypothesis does not turn out to hold when tested upon. The span of the principal axes were also found to contain invalid reber strings.

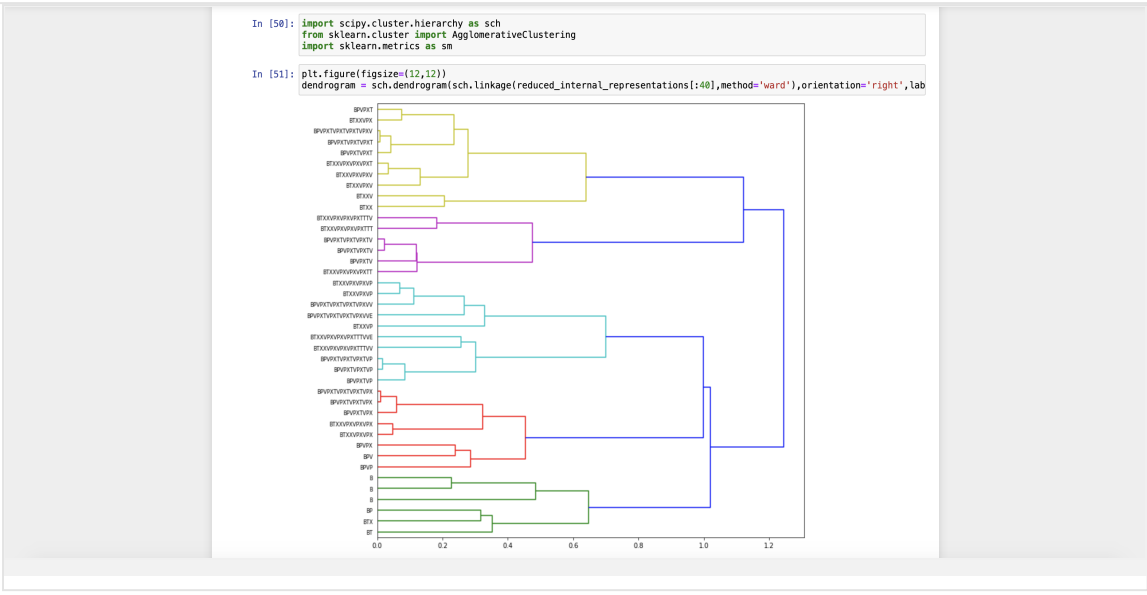

Fig 10 Inappropriate Clustering with Misclassified Strings

This is really shocking and looking at the dendrogram returned by the clustering program once again, it is evident that unlike an SRN, an ESN doesn't learn a sequence by encoding the current state on to its reservoir state matrix. In future studies, we are going to perform Latent Semantic Analysis (LSA) on the output weight matrix, Wout , in order to come to a better understanding of the learning procedure of an Echo State Network.

4.3 Long Short Term Memory Network

After training the network for 50 epochs (with validation and early stopping to prevent overfitting) , the average accuracy received is close to 97% which is significant taking into consideration how small in size the training corpus was. Looking at the dendrogram after clustering analysis, it is clear that clustering has approximately happened node wise and that each cluster corresponds to the current node in the grammar reached by the string inputted so far. It is clear that the activation patterns are grouped according to the different nodes in the finite state grammar. All patterns that produce a similar prediction are grouped together, independently of the current letter. But unlike an SRN, its not perfect and there are a small proportion of strings that are misclustered. Again, with only 3 hidden units, representational resources are so scarce that backpropagation forces the network to develop representations that yield a prediction based on the basis of the current node alone, ignoring contributions from the path.

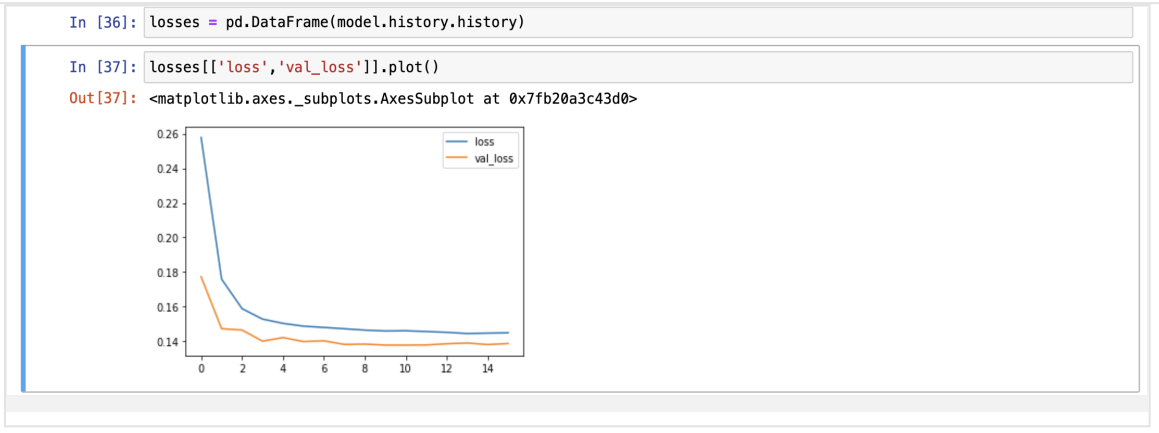

Fig 11 Loss function

While running the network with 7 LSTM units (instead of 3) in the hidden layer , accuracy rises to 98% and clustering happens to be more accurate. When run using 14 LSTM units, accuracy further rises to 99% and clustering happens to be as perfect as an SRN network itself. This shows that the learning procedure of an LSTM network is similar to that of an SRN and that it learns a sequence by encoding the current state on to its hidden state representation.

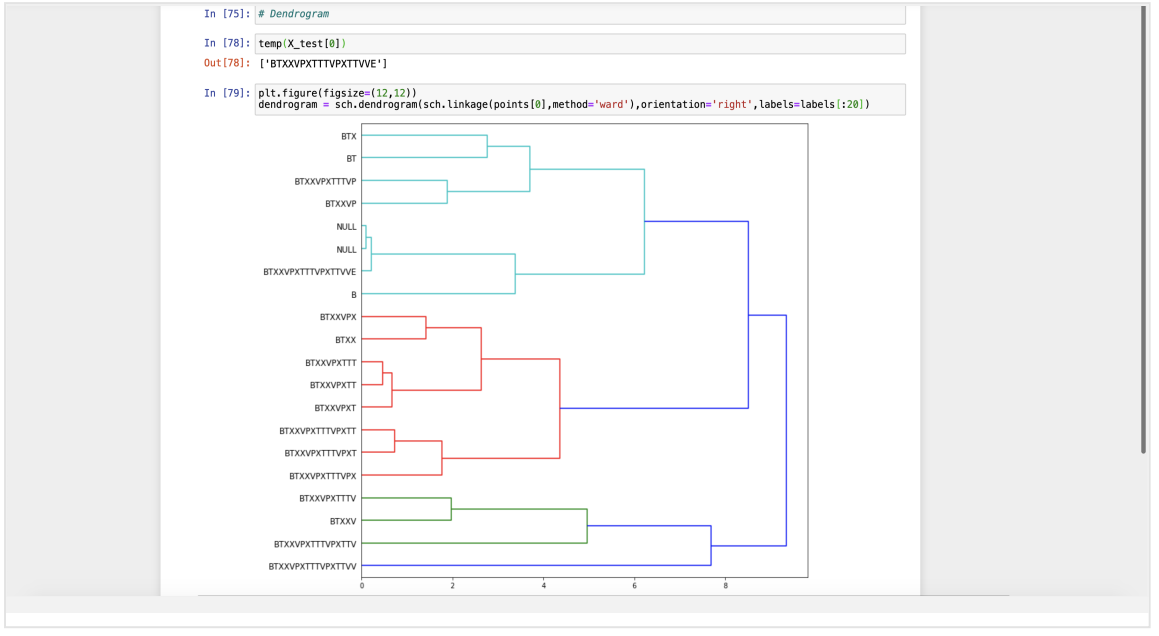

Fig 12 Clustering on Hidden state values of LSTM Cells at each Timestep

5 Discussions

We have attempted to understand better how the simple recurrent network could learn to represent and usecontextual information when presented with structured sequences of inputs. We can see here that that copyingthe state of activation on the hidden layer at the previous time step provided the network with the basicequipment of a finite state machine. When the set of exemplars that the network is trained on comes from afinite state grammar, the network can be used as a recognizer with respect to that grammar. When therepresentational resources are severely constrained, internal representations actually converge on the nodes of the grammar. This is evident from the node wise clustering that the clustering program returned The merepresence of recurrent connections pushed the network to develop hidden layer patterns that capture information about sequences of inputs, even in the absence of training. Encoding of sequential structuredepends on the fact that back propagation causes hidden layers to encode task relevant information. In thesimple recurrent network, internal representations encode not only the prior event but also relevant aspects ofthe representation that was constructed in predicting the prior event from its predecessor. When fed back asinput, these representations provide information that allows the network to maintain prediction relevant featuresof an entire sequence. This is exactly what we illustrated with cluster analysis of the hidden layer patterns.

The results also demonstrate that ESNs have the ability to learn to be sensitive to grammatical structure. The overall performance is comparable with the results of (Elman, 1991, 1993), yet, unlike Simple Recurrent Networks, ESNs perform this task without specialized learned internal representations. Given the prominence that such learned internal representations enjoy in the literature, this is a surprising result. ESNs compensate for this inability to learn useful representations with a large reservoir of input-driven random dynamics that capture some of the statistical regularities of the input. Also, the one-phase learning of ESNs is incapable of making use of a staged learning mechanism. Nevertheless, we see from our results here that such a mechanism is, at least in this case, not essential.

6 Conclusions

Echo State Networks succeeded in producing the probabilities of the next possible transitions from a given node in the grammar. It is not clear at this time whether these results can scale up to full natural human language. However, the results suggest that they are capable of performing the task using a corpus that Elman designed to answer questions fundamental to

modelling language (Elman, 1991). ESNs and SRNs, both fixed- resource systems, were capable of accommodating the subset of natural language used here. Furthermore, ESNs achieve performance comparable (in fact better in our case) to that of SRNs without the staged learning procedure necessary for SRNs.

What makes ESNs particularly interesting is that the hidden layer is driven completely by the input and the dynamics generated by the sparse and randomly weighted connections. While SRNs function by having the hidden layer learn to map functionally identical states into the same region of the representational space, ESNs are incapable of this kind of learning at the hidden layer. Surprisingly, they don't seem to require it. LSTMs perform similar to SRNs but they perform much better when the length of the Reber strings used is large as they perform extremely well in tackling long-term dependencies. This is because they have the concept of a cell state and alterations to it depends on what the network encounters at each timestep unlike an SRN whose memory decays uniformly following a function (mostly exponential, depends on the activation function used).

REFERENCES

[1] J. L. Elman, "Finding structure in time," Cogn. Sci., vol. 14, pp. 179–211, 1990.

[2] D. P. Mandic and J. A. Chambers, "Recurrent Neural Networks for Prediction: Learning Algorithms, Architecture and Stability", Chichester, U.K.: Wiley, 2001.

[4] D. Servan-Schreiber, Cleeremans A., & McClelland, J. L. (1988) , "Encoding Sequential Structure in Simple Recurrent Networks" (Technical Report CMU-CS-183). Pittsburgh,PA: Carnegie Mellon University, School of Computer Science.

[5] Hochreiter, S. and Schmidhuber, J. (1997), "Long short-term memory. *Neural Computation"* , *9*(8), 1735–1780.

[6] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term de- pendencies with gradient descent is difficult", *Neural Networks, IEEE Transactions on*, vol. 5, no. 2, pp. 157–166, 1994.

[7] F. A. Gers and J. Schmidhuber, "LSTM recurrent networks learn simple context free and context sensitive languages," *IEEE Transactions on Neural Networks*, vol. 12, no. 6, pp. 1333–1340, 2001.

[8] Jaeger, H. (2001), "The echo state approach to analysing and training recurrent neural networks." *Technical report GMD report 148*. German National Research Center for Information Technology.

[9] Jaeger, H. (2002). "A tutorial on training recurrent neural networks. Covering BPTT, RTRL, EKF, and the Echo State Network Approach". *GMD report 159*. German National Research Center for Information Technology.

[10] https://numpy.org/

[11] https://matplotlib.org/

[12] https://scipy.org/

[13] https://keras.io/

[14] https://www.tensorflow.org/

[15] https://seaborn.pydata.org/

ACKNOWLEDGEMENTS

First and foremost, I would like to thank God for bestowing upon me this opportunity, good health and an environment supportive enough to complete this fellowship.

I wish to sincerely and whole-heartedly thank **Prof. Bapi Raju Surampudi**, *Head of Cognitive Sciences Lab*, **International Institute of Information Technology, Hyderabad** for agreeing to guide me remotely. I am extremely grateful for his continuous guidance and support through the course of the fellowship. He equipped me with a structured plan to follow and had frequent checks on my progress. His feedback, suggestions and ideas were what made the project what it is. I am glad that this topic was chosen as the basis of the fellowship as it has opened newer windows for exploration. His knowledge, expertise, promptness in identifying a problem and quickness in offering a solution inspires me.

I would wish to convey my heartiest thanks to **Indian Academy of Sciences** for offering the Summer Research Fellowship, which has been a great opportunity for me to venture newer fields and learn so many new concepts. I would like to extend my gratitude to the *Coordinator of Science Education Programme*, **Mr. C.S. Ravi Kumar** for his constant help. Sincere thanks

for being so accessible and patiently responding to our innumerable concerns, thereby easing our way through the fellowship. I would like to thank **Mr. M.R.N Murthy**, *Chairman, Joint Science Education Panel* for allowing all summer fellows continue the fellowships from the comfort of our homes.

I would like to thank my friends' and family's unparalleled support and love that drove me to complete this project. Their motivation and positive feedback brightened my spirits and helped me have a delightful learning experience.

APPENDICES

APPENDIX A: Code for SRN and LSTM

```
# Reber Grammar Implementation using Keras

# Reber Grammar String Generator Class

import random as rnd

class ReberGrammarLexicon(object):

 lexicon = set() #contain Reber words
 graph = [ [(1,'T'), (2,'P')], \
 [(1, 'S'), (3, 'X')], \
 [(2,'T') ,(4, 'V')], \
 [(2, 'X'), (5,'S')], \
 [(3, 'P'),(5, 'V')], \
 [(6,'E')] ] #store the graph

 def __init__(self, num, maxSize = 1000): #fill Lexicon with
num words

 self.maxSize = maxSize

 if maxSize < 5:
 raise NameError('maxSize too small, require maxSize > 4')

 while len(self.lexicon) < num:

 word = self.generateWord()
 if word != None:
```

```
    self.lexicon.add(word)

    def generateWord(self): #generate one word

    c = 2
    currentEdge = 0
    word = 'B'

    while c <= self.maxSize:

    inc = rnd.randint(0,len(self.graph[currentEdge])-1)
    nextEdge = self.graph[currentEdge][inc][0]
    word += self.graph[currentEdge][inc][1]
    currentEdge = nextEdge
    if currentEdge == 6 :
    break
    c+=1

    if c > self.maxSize :
    return None

    return word

import numpy as np

maxsize = 20
inputdim = 7
outputdim = 7
hiddendim = 3

chars='BEPSTVX'

# One-hot encoding generated reber strings

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder

data = ['B', 'E', 'P', 'S', 'T', 'V', 'X']
values = np.array(data)

# integer encode
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(values)

# binary encode
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encode
d), 1)
```

```
onehot_encoded = onehot_encoder.fit(integer_encoded)

def func(word,start=0):
 word = np.array(list(word[start:]))
 encoded = label_encoder.transform(word)
 return onehot_encoded.transform(encoded.reshape(-1,1))

# Padding each one-hot encoded input string to the maximum l
ength

from tensorflow.keras.preprocessing.sequence import pad_sequ
ences

def pad(z):
 z = np.expand_dims(z,axis =0)
 padded = pad_sequences(z,maxlen=20,padding='post',value=
[0.,0.,0.,0.,0.,0.,0.])
 return padded[0]

def preprocessing(dictionary,start=0):
 length = len(dictionary)
 result = []

 for word in dictionary:
 result.extend(pad(func(word,start)))

 return np.array(result).reshape(length,maxsize,7)

generator = ReberGrammarLexicon(2400,maxSize=20)

generator.lexicon

# Train-Test Split

X_train = preprocessing(list(generator.lexicon)[:2000])

y_train = preprocessing(list(generator.lexicon)[:2000],1)

X_test = preprocessing(list(generator.lexicon)[2000:])

y_test = preprocessing(list(generator.lexicon)[2000:],1)

# Model Creation

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,SimpleRNN,TimeDist
ributed,Dropout

```

```
model = Sequential()

model.add(SimpleRNN(hiddendim,activation='relu',return_sequences=True,input_shape=X_train.shape[1:]))

# model.add(LSTM(hiddendim,activation='relu',return_sequences=True,input_shape=X_train.shape[1:]))

model.add(Dropout(0.2))

model.add(TimeDistributed(Dense(outputdim,activation='softmax')))

from tensorflow.keras.callbacks import EarlyStopping

early_stop = EarlyStopping(monitor='val_loss',mode='min',patience=5,verbose=1)

import tensorflow.keras.backend as K
from tensorflow.keras.metrics import top_k_categorical_accuracy
def my_top_k(true, pred, num=2):
 true = K.reshape(true, (-1, outputdim))
 pred = K.reshape(pred, (-1, outputdim))
 return top_k_categorical_accuracy(true, pred, k=num)

model.compile(optimizer='rmsprop',loss='binary_crossentropy',metrics=[my_top_k])

model.fit(x=X_train,y=y_train,batch_size=1,epochs=50,callbacks=[early_stop],validation_split=0.2)

import pandas as pd

losses = pd.DataFrame(model.history.history)

losses[['loss','val_loss']].plot()

score = model.evaluate(X_test, y_test, batch_size=1,
verbose=1)

z = model.predict(X_test)

# Analysis of a string

print(list(chars))
print((z.round(decimals=2)[0]))

```

```
temp(X_test[0])

y_test[0]

def top_2_accuracy(y_test,y_pred):
 i = 0
 k = y_test.shape[0]*y_test.shape[1]
 correct = 0
 test = np.reshape(y_test,(y_test.shape[0]*y_test.shape[1],y_test.shape[2]))
 pred = np.reshape(y_pred,(y_pred.shape[0]*y_pred.shape[1],y_pred.shape[2]))
 while(i<len(test)):

 if (test[i] == np.array((0, 1, 0, 0, 0, 0, 0))).all():
 if (test[i].argmax() == pred[i].argsort()[-1]) | (test[i].argmax() == pred[i].argsort()[-2]):
 correct += 1
 k -= (20%i)
 i = i + (20%i)
 continue

 if (test[i].argmax() == pred[i].argsort()[-1]) | (test[i].argmax() == pred[i].argsort()[-2]):
 correct += 1
 i += 1
 print(correct)
 print(k)
 print(correct/k)

top_2_accuracy(y_test,z)



import matplotlib.pyplot as plt
%matplotlib inline

# Loss function graph over epochs
# Binary Cross-Entropy Loss

plt.plot(list(range(50)),model.history.history['loss'])

# Retrieving back the hidden unit activation paterns

import tensorflow.keras.backend as K

lstm = model.layers[0]

```

```
# Get output from intermediate layer to visualize activation
s
attn_func = K.function(inputs = [model.get_input_at(0), K.le
arning_phase()],
 outputs = [lstm.output])

attn_func.outputs[0]

h = attn_func(X_test)[0]

len(h)

points = (h.round(decimals=2))

points.shape

# Hierarchical Clustering using Scipy

import scipy.cluster.hierarchy as sch
from sklearn.cluster import AgglomerativeClustering
import sklearn.metrics as sm

# Dendrogram

temp(X_test[0])

points[0]

def seq2char(sequence):

 if sequence.any():
 return chars[sequence.argmax()]
 else:
 return ''

def word(X):
 l = X.shape[0]
 result = []

 for i in range(l):
 temp = ''
 for sequence in X[i]:
 temp = temp + seq2char(sequence)
 result.append(temp)

 return result

def temp(X):
```

```
 X = np.expand_dims(X,axis=0)
 l = X.shape[0]
 result = []

 for i in range(l):
 temp = ''
 for sequence in X[i]:
 temp = temp + seq2char(sequence)
 result.append(temp)

 return result

word_list = word(X_test)

def func(word):
 label_list = []
 k=0

 for i in range(1,21):
 if i <= len(word):
 label_list.append(word[:i])
 else:
 label_list.append('NULL')

 k = k + 1

 return label_list

points_temp =
points.reshape(points.shape[0]*points.shape[1],points.shape[
2])

X_test_temp =
X_test.reshape(X_test.shape[0]*X_test.shape[1],X_test.shape[
2])

def label(X):
 result = []

 for i in range(0,len(X),maxsize):
 word_no = i // maxsize

 result.extend(func(word_list[word_no]))

 return result

labels = label(X_test_temp)

plt.figure(figsize=(12,12))
dendrogram = sch.dendrogram(sch.linkage(points_temp,method
```

```
='ward'),orientation='right',labels=labels)

points[0]

# Saving Model

from tensorflow.keras.models import save_model,load_model

model.save("reber.h5")

model = load_model("reber.h5")
```

Code for the Model

APPENDIX B: Code for Echo State Network (ESN)

```
# Implementing Echo State Network for Artificial Grammar Lea
rning(Reber)

# Importing required modules

%matplotlib inline

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import linalg
from ipywidgets import *
from IPython.display import *

# Generating Reber Strings

import random as rnd

class ReberGrammarLexicon(object):

 lexicon = set() #contain Reber words
 graph = [ [(1,'T'), (2,'P')], \
 [(1, 'S'), (3, 'X')], \
 [(2,'T') ,(4, 'V')], \
 [(2, 'X'), (5,'S')], \
 [(3, 'P'),(5, 'V')], \
 [(6,'E')] ] #store the graph
```

```
27.
28.  def __init__(self, num, maxSize = 1000): #fill Lexicon with
    num words
29.
30.  self.maxSize = maxSize
31.
32.  if maxSize < 5:
33.  raise NameError('maxSize too small, require maxSize > 4')
34.
35.  while len(self.lexicon) < num:
36.
37.  word = self.generateWord()
38.  if word != None:
39.  self.lexicon.add(word)
40.
41.  def generateWord(self): #generate one word
42.
43.  c = 2
44.  currentEdge = 0
45.  word = 'B'
46.
47.  while c <= self.maxSize:
48.
49.  if(((currentEdge==3) | (currentEdge==4)) &
    (c<(self.maxSize/9))):
50.  inc=0
51.  else:
52.  inc = rnd.randint(0,len(self.graph[currentEdge])-1)
53.
54.  nextEdge = self.graph[currentEdge][inc][0]
55.  word += self.graph[currentEdge][inc][1]
56.  currentEdge = nextEdge
57.  if currentEdge == 6 :
58.  break
59.  c+=1
60.
61.  if c > self.maxSize :
62.  return None
63.
64.  return word
65.
66. from sklearn.preprocessing import LabelEncoder
67. from sklearn.preprocessing import OneHotEncoder
68.
69. data = ['B', 'E', 'P', 'S', 'T', 'V', 'X']
70. values = np.array(data)
71.
72. # integer encode
```

```
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(values)

# binary encode
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
onehot_encoded = onehot_encoder.fit(integer_encoded)

def func(word,start=0):
 word = np.array(list(word[start:]))
 encoded = label_encoder.transform(word)
 return onehot_encoded.transform(encoded.reshape(-1,1))

def preprocessing(dictionary,start=0):
 length = len(dictionary)
 result = func(list(dictionary)[0])

 for word in list(dictionary)[1:]:
 b = func(word)
 result = np.vstack((result,b))


 return result

generator = ReberGrammarLexicon(2000,maxSize=20)

generator.lexicon

training_set = preprocessing(list(generator.lexicon)[:2000])

training_set.shape

# Setting random seed

def set_seed(seed=None):
 """Making the seed (for random values) variable if None"""

 if seed is None:
 import time
 seed = int((time.time()*10**6) % 4294967295)
 print(seed)
 try:
 np.random.seed(seed)
 print("Seed used for random values:", seed)
 except:
 print("!!! WARNING !!!: Seed was not set correctly.")
 return seed
```

```
120.
121. # Creating Network Class
122.
123. class Network(object):
124.
125.  def __init__(self, trainLen=2000, testLen=2000,
     initLen=100) :
126.  self.initLen = initLen
127.  self.trainLen = trainLen
128.  self.testLen = testLen
129.  self.data = training_set
130.  self.inSize = self.outSize = 7 #Input/Output dimensions
131.  self.resSize = 400 #Reservoir size (prediction)
132.  #self.resSize = 1000 #Reservoir size (generation)
133.  self.a = 1 #Leak rate alpha
134.  self.spectral_radius = 1.25 #Spectral raidus
135.  self.input_scaling = 1. #Input scaling
136.  self.reg = 1e-8 #None #Regularization factor - if None,
137.  #we'd use pseudo-inverse rather than ridge regression
138.
139.  self.mode = 'prediction'
140.  #self.mode = 'generative'
141.
142.  #Change the seed, reservoir performances should be averaged
     accross
143.  #at least 20 random instances (with the same set of paramet
     ers)
144.  seed = None #42
145.
146.  set_seed(seed)
147.
148. nw = Network()
149.
150. # Generating Win,W randomly and then generating X,Ytarget
151.
152. from scipy.sparse import rand
153.
154. def initialization(nw) :
155.
156.  #Weights
157.  #nw.Win = (np.random.rand(nw.resSize,1+nw.inSize)) * nw.inp
     ut_scaling
158.  #nw.W = np.random.rand(nw.resSize,nw.resSize)
159.  nw.Win = np.array(rand(nw.resSize,1+nw.inSize,
     density=0.25, format="csr", random_state=42).todense())
160.  nw.W = np.array(rand(nw.resSize,nw.resSize, density=0.25, f
     ormat="csr", random_state=42).todense())
161.
```

```
 #Matrices
 #Allocated memory for the design (collected states) matrix
 nw.X = np.zeros((1+nw.inSize+nw.resSize,nw.trainLen-nw.initLen))
 #Set the corresponding target matrix directly
 nw.Ytarget = nw.data[None,nw.initLen+1:nw.trainLen+1]

 #Run the reservoir with the data and collect X
 nw.x = np.zeros((nw.resSize,1))

 return(nw)

# Computing spectral radius(biggest of the absolute eigen values of W matrix) and then scaling W matrix using that

def compute_spectral_radius(nw):
 print('Computing spectral radius...',end=" ")
 rhoW = max(abs(linalg.eig(nw.W)[0]))
 print('Done.')
 nw.W *= nw.spectral_radius / rhoW

 return(nw)

# Learning phase

# xn = (1-α)xn + αtanh(Win.un - 1) + W.xn-1

def learning_phase(nw) :
 for t in range(nw.trainLen):
 #Input data
 nw.u = nw.data[t]

 # if (nw.u == np.array((0, 1, 0, 0, 0, 0, 0))).all():
 # nw.x = np.zeros((nw.resSize,1))
 # else:
 nw.x = (1-nw.a)*nw.x + nw.a*np.tanh( np.dot(nw.Win, np.vstack((1,nw.u.reshape(7,1))) ) + np.dot( nw.W, nw.x ) )
 #After the initialization, we start modifying X
 if t >= nw.initLen:
 nw.X[:,t-nw.initLen] = np.vstack((1,nw.u.reshape(7,1),nw.x.reshape(nw.resSize,1)))[:,0]

 return(nw)

# Training output weights using ridge regression
# Wout = (Yt.XT) . (X.XT+reg.I)^-1

def train_output(nw) :
```

```
206.  nw.X_T = nw.X.T
207.  if nw.reg is not None:
208.  # Ridge regression (linear regression with regularization)
209.  nw.Wout = np.dot(np.dot(nw.Ytarget[0].T,nw.X_T),
      linalg.inv(np.dot(nw.X,nw.X_T) + \
210.  nw.reg*np.eye(1+nw.inSize+nw.resSize) ) )
211.  else:
212.  # Pseudo-inverse
213.  nw.Wout = np.dot(nw.Ytarget, linalg.pinv(nw.X) )
214.
215.  return(nw)
216.
217. # Testing in a particular mode
218.
219. def test(nw) :
220.  #Run the trained ESN in a generative mode. no need to initi
      alize here,
221.  #because x is initialized with training data and we continu
      e from there.
222.  nw.Y = np.zeros((nw.testLen,nw.outSize))
223.  nw.u = nw.data[nw.trainLen]
224.  nw.reservoir = np.zeros((nw.testLen,nw.resSize))
225.  for t in range(nw.testLen):
226. # if (nw.u == np.array((0, 1, 0, 0, 0, 0, 0))).all():
227. # nw.x = np.zeros((nw.resSize,1))
228. # else:
229.  nw.x = (1-nw.a)*nw.x + nw.a*np.tanh( np.dot(nw.Win, np.vsta
      ck((1,nw.u.reshape(7,1))) ) + np.dot( nw.W, nw.x ) )
230.
231.  nw.reservoir[t] = nw.x.reshape(nw.resSize,)
232.  nw.y = np.dot(nw.Wout,
      np.vstack((1,nw.u.reshape(7,1),nw.x.reshape(nw.resSize,1)))
      )
233.  nw.Y[t][:] = nw.y.reshape(1,7)
234.  if nw.mode == 'generative':
235.  #Generative mode:
236.  nw.u = nw.y
237.  elif nw.mode == 'prediction':
238.  #Predictive mode:
239.  nw.u = nw.data[nw.trainLen+t+1]
240.  else:
241.  raise(Exception, "ERROR: 'mode' was not set correctly.")
242.
243.  return(nw)
244.
245. def compute_error(nw) :
246.  # Computing MSE for the first errorLen iterations
247.  errorLen = 500
248.  mse = sum( np.square( nw.data[nw.trainLen+1:nw.trainLen+
```

```
errorLen+1] - nw.Y[0,0:errorLen] ) ) / errorLen
 print('MSE = ' + str( mse ))

 return(nw)

def compute_network(nw) :
 nw = initialization(nw)
 nw = compute_spectral_radius(nw)
 nw = learning_phase(nw)
 nw = train_output(nw)
 nw = test(nw)
 nw = compute_error(nw)
 return(nw)

# Definition of the network parameters

select_mode = ToggleButtons(description='Mode:',
 options=['prediction', 'generative'])
var1 = FloatSlider(value=300, min=0, max=1000, step=1, descr
iption='resSize')
var2 = FloatSlider(value=100, min=0, max=2000, step=1, descr
iption='initLen')
var3 = FloatSlider(value=2000, min=0, max=30000, step=1, des
cription='trainLen')
var4 = FloatSlider(value=2000, min=0, max=8000, step=1, desc
ription='testLen')
var5 = FloatSlider(value=1.25, min=0, max=10, step=0.05, des
cription='spectral radius')
var6 = FloatSlider(value=0.3, min=0, max=1, step=0.01, descr
iption='leak rate')
valid = Button(description='Validate')

def record_values(_) :
 clear_output()
 nw.mode=select_mode.value
 nw.resSize=int(var1.value)
 nw.initLen=int(var2.value)
 nw.trainLen=int(var3.value)
 nw.testLen=int(var4.value)
 nw.spectral_radius=float(var5.value)
 nw.a=float(var6.value)
 print("InitLen:", nw.initLen, "TrainLen:", nw.trainLen, "Te
stLen:", nw.testLen)
 print("ResSize:", nw.resSize, "Spectral Radius:", nw.spectr
al_radius, "Leak Rate:", nw.a)
 compute_network(nw)
 return(nw)

```

```
display(select_mode)
display(var1)
display(var2)
display(var3)
display(var4)
display(var5)
display(var6)
display(valid)

valid.on_click(record_values)

y_pred = nw.Y

y_test =
training_set[nw.trainLen+1:nw.trainLen+nw.testLen+1]

def top_2_accuracy(y_test,y_pred):
 k = 0
 for i in range(nw.testLen):
 if (y_test[i].argmax() == y_pred[i].argsort()[-1]) | (y_tes
t[i].argmax() == y_pred[i].argsort()[-2]):
 k += 1
 print(k/nw.testLen)

top_2_accuracy(y_test,y_pred)

chars='BEPSTVX'

y_test[:20]

y_pred[:20].round(2)

from plotly import __version__
import cufflinks as cf
from plotly.offline import download_plotlyjs,init_notebook_m
ode,plot,iplot
init_notebook_mode(connected=True)
cf.go_offline()
from sklearn.decomposition import PCA

pca = PCA(n_components=10).fit(nw.reservoir)

df = pd.DataFrame({'Number of Components':list(range(1,nw.re
sSize+1)),'Cummulative Variance Explained':list(np.cumsum(pc
a.explained_variance_ratio_))})

plt.figure(figsize=(10,8))
df.iplot(kind='scatter',x='Number of Components',y=
```

```
330. 'Cummulative Variance Explained')
331.
332. np.cumsum(pca.explained_variance_ratio_)[:20]
333.
334. reduced_internal_representations = pca.transform(nw.reservoi
     r)
335.
336. reduced_internal_representations[0].round(2)
337.
338. corpus = generator.lexicon
339.
340. def label_generator(corpus):
341.  result = []
342.
343.  for word in corpus:
344.
345.  for i in range(len(word)):
346.  result.append(word[:i+1])
347.
348.  return result
349.
350. labels = label_generator(corpus)
351. test_labels = labels[nw.trainLen+1:nw.trainLen+nw.testLen+1]
352.
353. len(test_labels)
354.
355. import scipy.cluster.hierarchy as sch
356. from sklearn.cluster import AgglomerativeClustering
357. import sklearn.metrics as sm
358.
359. plt.figure(figsize=(12,12))
360. dendrogram = sch.dendrogram(sch.linkage(reduced_internal_rep
     resentations[:40],method='ward'),orientation='right',labels=
     test_labels[:40])
361.
362. nw.Wout.shape
363.
364. from sklearn.decomposition import TruncatedSVD
365.
366. svd = TruncatedSVD(n_components=5)
367.
368. lsa = svd.fit_transform(nw.Wout)
369.
370. lsa
371.
372. print(svd.explained_variance_ratio_)
373.
374. print(svd.singular_values_)
```

```
375.
376. from numpy.linalg import svd as SVD
377.
378. U, S, VT = SVD(nw.Wout,full_matrices=False)
379.
380. print("Left Singular Vectors:")
381. print(U)
382. print()
383. print("Singular Values:")
384. print(np.diag(S))
385. print()
386. print("Right Singular Vectors:")
387. print(VT)
388.
389. # check that this is an exact decomposition
390. # @ is used for matrix multiplication in Py3, use np.matmul
     with Py2
391. print(U @ np.diag(S) @ VT)
392.
393. chars='BEPSTVX'
394. print(list(chars))
395. print(np.round(y_pred,decimals=2)[:20])
396.
397. y_test[:20]
398.
399. # Graph 1: Plotting neurons activations (total)
400.
401. var10 = FloatSlider(value=2000,min=10,max=nw.trainLen-nw.ini
     tLen,step=10,description='time steps')
402. var11 = FloatSlider(value=10, min=1, max=nw.resSize, step=1,
     description='number of neurons')
403. valid = Button(description='Validate')
404.
405. def trace_graph3(_) :
406.  clear_output()
407.  f=int(var10.value)
408.  nb=int(var11.value)
409.  plt.figure(3).clear()
410.  plt.figure(figsize=(10,7))
411.  plt.plot( nw.X[2:2+nb,0:f].T )
412.  print(nw.X.shape)
413.  plt.ylim([-1.1,1.1])
414.  plt.title('Activations $\mathbf{x}(n)$ from Reservoir Neuro
     ns ID 0 to '+str(nb-1)+' for '+str(f)+' time steps')
415.
416. valid.on_click(trace_graph3)
417.
418. display(var10)
```

```
419. display(var11)
420. display(valid)
421.
422. # Graph 2: Plotting single neuron activation
423.
424. var12 = FloatSlider(value=2000,min=10,max=nw.trainLen-nw.ini
     tLen,step=10,description='time steps')
425. var13 = FloatSlider(value=2, min=0, max=nw.resSize-1,
     step=1, description='neuron ID')
426. valid = Button(description='Validate')
427.
428. def trace_graph4(_) :
429.  clear_output()
430.  f=int(var12.value)
431.  num=int(var13.value)
432.  plt.figure(4).clear()
433.  plt.figure(figsize=(10,5))
434.  plt.plot( nw.X[2+num,:f].T )
435.  plt.ylim([-1.1,1.1])
436.  plt.title('Activations $\mathbf{x}(n)$ from Reservoir Neuro
     n ID '+str(num)+' for '+str(f)+' time steps')
437.
438. valid.on_click(trace_graph4)
439.
440. display(var12)
441. display(var13)
442. display(valid)
443.
444. # Graph 3: Output weights at the end of the simulation
445.
446. valid = Button(description='Show')
447.
448. def trace_graph5(_) :
449.  clear_output()
450.  plt.figure(5).clear()
451.  plt.figure(figsize=(12,7))
452.  plt.bar(range(1+nw.inSize+nw.resSize),
     np.squeeze(nw.Wout.T) )
453.  plt.title('Output weights $\mathbf{W}^{out}$')
454.
455. valid.on_click(trace_graph5)
456.
457. display(valid)
```

Code for the Model

APPENDIX C: Q&A Interactive Section using Echo State Network (ESN)

```
# Q&A Session with the Echo State Network

def test_per_datapoint(nw,one_hot_char) :
 #Run the trained ESN in a generative mode. no need to initialize here,
 #because x is initialized with training data and we continue from there.
 #Y = np.zeros((1,nw.outSize))
 nw.u = one_hot_char

 #if (nw.u == np.array((0, 1, 0, 0, 0, 0, 0))).all():
 # nw.x = np.zeros((nw.resSize,1))
 # else:
 nw.x = (1-nw.a)*nw.x + nw.a*np.tanh( np.dot(nw.Win, np.vstack((1,nw.u.reshape(7,1))) ) + np.dot( nw.W, nw.x ) )

 y = np.dot(nw.Wout, np.vstack((1,nw.u.reshape(7,1),nw.x.reshape(nw.resSize,1))) )
 #nw.Y[t][:] = nw.y.reshape(1,7)

 return(nw,y)

print('Valid reber strings start with a "B" and end with an "E"')

while True:
 first = input('Enter the first alphabet: ')
 if (first == 'B') | (first == 'Q'):
 break
 else:
 print('Invalid first alphabet')

if first!='Q':
 char = first
 nw.x = np.zeros((nw.resSize,1))
 y = np.zeros((nw.outSize,1))
 string = first
 while True:
 one_hot_char = func(char)

 (nw,y) = test_per_datapoint(nw,one_hot_char)

```

```
one = int(np.squeeze(y).argsort()[-1])
two = int(np.squeeze(y).argsort()[-2])

possible1 = chars[one]
possible2 = chars[two]

if (possible1 == 'E') | (possible2 == 'E'):
print('E is the only possible next transition')
print('End of string')
print(string + 'E is a valid Reber String...!!!')
break

print(possible1 + ' and ' + possible2 + ' are the next poss
ible transitions')
print('\n')
while True:
char = input('Choose any one: ')
if char == 'Q':
break
elif (char != possible1) & (char != possible2):
print('Invalid input')
else:
break

if char == 'Q':
break

string += char
```

Code for Q&A Section with ESN