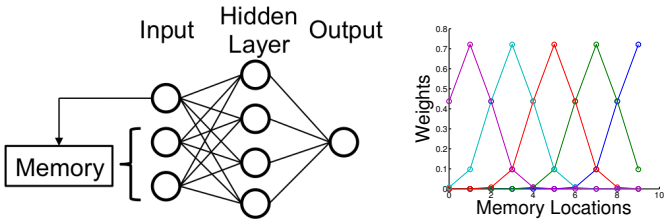
Neural Memory Networks

David Biggs and Andrew Nuttall

I. Introduction

CONTEXT is vital in formulating intelligent classifications and responses, especially under uncertainty. In a standard feed-forward neural network (FFNN), context comes in the form of information encoded in the input vector and trained in weight parameters. However, useful information can also be present in the temporal nature of the input vectors, or from past internal states of a network. Future outputs can achieve better accuracy by observing transient trends in the input data, or by utilizing key memories from distant inputs which could be crucial to formulating a correct output. By providing a neural network with an architecture for storing and maintaining memories this additional context can be effectively leveraged.

A simple implementation of memory in a neural network would be to write inputs to external memory and use this to concatenate additional inputs into a neural network. For noisy analog inputs, memory inputs pulled from Gaussian distributions can act to preprocess and filter the data. Fig. 1 shows a schematic and memory weight distribution of a FFNN with external memory augmentation.



(a) Network schematic

(b) Weights of memory matrix

Fig. 1: Architecture of FFNN with simple memory implementation

This architecture was implemented to predict the next sequence in a noisy sinusoidal signal. Ten previous inputs were stored in memory, from which five additional inputs were drawn from Gaussian weights on the memory. The network is trained with a single sinusoid then tested with a sum of three sinusoids of varying frequencies with errors around 10% for training and 15% for testing. The convergence and input/output waveforms are shown in Fig. 2.

Network architectures with delayed inputs or additional inputs drawn from memory can be useful tools, but are limited in functionality since the memory architecture is predetermined. A more powerful memory architecture would store memory inside the network to allow the network to learn how to utilize memory via read and write commands.

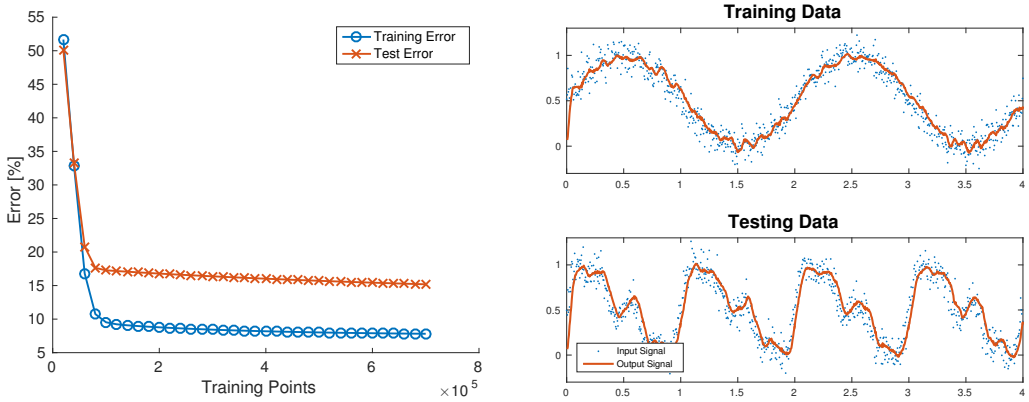
II. Related Work

Neural networks were first introduced over 60 years ago as one of the first learning algorithms. Recurrent neural networks (RNN) were then introduced in the 1980's to better process sequential inputs by maintaining an internal state between inputs. Long-Short-Term-Memory (LSTM) improved upon RNN's in the late 1990's by adding logic gates to read, write, and forget internal memory states. Recently, Weston et al. [1] defined a framework for neural networks to interact with external memory in order to read and write long term memory. In their paper, Westin implements an RNN with memory for textual story processing in which several actors move between rooms and while carrying and deposit objects. In their results they showed that with memory their network outperformed a similar RNN and LSTM without memory access at answering questions about a story. Around the same time, work was published by Graves et al. [2] in which they implement an algorithm they call the *Neural Turing Machine* (NTM). The NTM algorithm has memory structures called *memory heads* that can be accessed in order to read, write, and erase data. They ran several test on their NTM, one of which was teaching the network to store then copy sequences of numbers. They trained their network up to sequences of twenty 8 bit numbers and tested up to sequences of 120 numbers with good results. This project differs from related work by implementing internal neural memory by assigning a set of memory nodes (memory bank) to each standard neuron, to be discussed in the following section. With this approach a richer memory can be achieved by not only maintaining key memories, but by retaining memory histories with associated temporal information.

III. Methodology

The neural memory network (NMN) architecture is comprised of an FFNN with a dedicated internal memory bank associated with each standard neuron in each of the hidden layers. The input and output layers are composed of only standard nodes, while the hidden layers are composed of standard nodes with their respective memory nodes. In this formulation, the output of each node is given by a sigmoid function of an affine combination of its inputs. The output of a node is used as a linear switch to activate the memories dedicated to that node. During forward propagation the neural network can learn to call upon these memories as needed to provide additional information in making classifications. Figure 3 depicts the general network layout for a simple memory neural network.

Training and testing of the algorithm is comprised of three key steps:



(a) Error convergence (b) Waveforms

Fig. 2: Signal prediction results from memory augmented FFNN

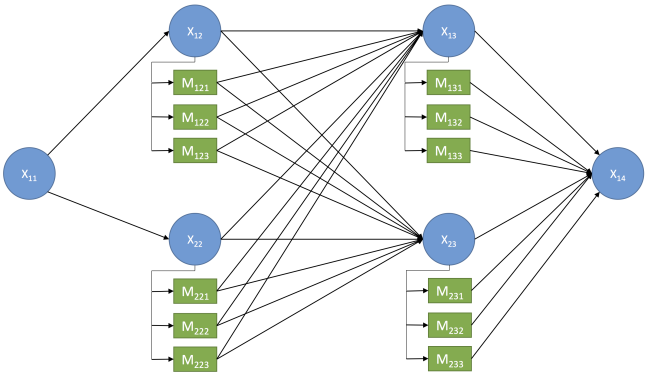


Fig. 3: A neural memory network has input and output layers composed of standard sigmoid nodes and hidden layers composed of both standard and memory nodes. Each standard node in a hidden layer has a memory bank associated with it, and each memory bank is composed of multiple memory nodes. The outputs of standard nodes are used as switches to activate memories when called upon.

- *Forward propagation* of the input across each network layer to form the output
- *Back propagation* of the errors across each layer starting at the output to perform gradient descent updates on network parameters
- *Memory storage* in each layer of nearby neuron outputs as an orthogonal set in a higher dimensional space

A. Forward Propagation

Forward propagation in an NMN functions in a similar manner to that of an FFNN, with an augmented output vector from each hidden layer. The output of each layer i is given by a two-step equation containing outputs from both standard neurons and memory neurons,

$$\mathbf{x}^{(i)} = f\left(\mathbf{W}^{(i-1)}\mathbf{x}^{(i-1)} + \mathbf{b}^{(i-1)}\right) \tag{1}$$

$$\mathbf{x}^{(i)} := \left[\mathbf{x}_1^{(i)}, \mathbf{x}_1^{(i)}\mathbf{M}_1^{(i)\top}, ..., \mathbf{x}_k^{(i)}, \mathbf{x}_k^{(i)}\mathbf{M}_k^{(i)\top}\right]^\top \tag{2}$$

The output state vector for each layer's standard nodes is augmented with the output of the layer's memory nodes. The output of each memory node is the product of the content of that memory slot and the output of the standard node it is attached to. With this approach the standard node acts as a linear switch to turn memories on or off as desired. During forward propagation in the hidden layers the outputs of a hidden layer only go to standard nodes, memory nodes only receive input from their respective standard node in the same layer.

B. Backward Propagation and Gradient Descent

The network is trained by updating the parameters $\mathbf{W}$ and $\mathbf{b}$ using gradient descent with the back-propagation technique. In the training process, an input is forward propagated through the neural network to produce an hypothesis $h(\mathbf{x_j})$ that is then compared to a known solution $\mathbf{y}$. The error is calculated as

$$\epsilon = \frac{1}{2}||\mathbf{h}(\mathbf{x}^{(\text{output})}) - \mathbf{y}||_2^2. \tag{3}$$

Performing gradient descent on an NMN requires different considerations to be given to the set of first and last layers and the set of hidden layers due to the network's asymmetry. To perform gradient descent on the weights the partial derivatives w.r.t $w_{j,k}^{(i)}$ are taken as,

$$\frac{\partial \epsilon}{\partial \mathbf{W}_{jk}^{(i)}} = \frac{\partial \epsilon}{\partial \mathbf{h}(\mathbf{x}_j^{(i)})} \frac{\partial \mathbf{h}(\mathbf{x}_j^{(i)})}{\partial \mathbf{x}_j^{(i)}} \frac{\partial \mathbf{x}_j^{(i)}}{\partial \mathbf{W}_{jk}^{(i)}} \tag{4}$$

Evaluating this expression for the final layer I of weights yields

$$\frac{\partial \epsilon}{\partial \mathbf{W}_{jk}^{(I)}} = \left(\mathbf{h}(\mathbf{x}_j^{(I)}) - \mathbf{y}\right) \mathbf{h}(\mathbf{x}_j^{(I)})(1 - \mathbf{h}(\mathbf{x}_j^{(i)}))\mathbf{x}^{I-1} \tag{5}$$

Back propagation is implemented in the standard way by storing gradients in the variable δ and passed to previous layers. This process starts at the output layer as,

$$\delta^{(\text{output})} = (\mathbf{x}^{(\text{output})} - \mathbf{y})\mathbf{x}^{(\text{output})}(1 - \mathbf{x}^{(\text{output})}) \tag{6}$$

The errors of the subsequent layers are weighted by the parameters

$$\delta^{(i)} = \mathbf{W}^{(i+1)}\delta^{(i+1)}\mathbf{x}^{(i+1)}(1-\mathbf{x}^{(i+1)}) \tag{7}$$

There are two paths back to the primary neurons, so the errors of each hidden layer must be summed across memory nodes

$$\delta_j^{(i)} = \delta_j^{(i)} + \sum_k \delta_{j+k}^{(i)}\mathbf{M}_{j+k}^{(i)} \tag{8}$$

The updates on $\mathbf{W}$ and $\mathbf{b}$ are then,

$$\mathbf{W}^{(i)} := \mathbf{W}^{(i)} - \alpha\delta^{(i)}\mathbf{x}^{(i+1)} \tag{9}$$

$$\mathbf{b}^{(i)} := \mathbf{b}^{(i)} - \alpha\delta^{(i)} \tag{10}$$

C. Memory Architecture

Every node in the hidden layers contains a static memory bank. Each unique memory in a node is orthogonal to all other memories in that node, such that no information is lost when the memories are added together. A collection of memories is then represented as a single vector in a higher dimensional space called a composite memory. The direction of the vector dictates what memories are present, and the magnitude of the vector when projected onto each memory's axis determines the order in which the memories occurred. When a node fires the current composite vector that is stored in memory is scaled according to some scheduled decay and then the new memory is added. The new memory is created by an orthogonal mapping of the outputs the node and its τ nearest neighbors. Nearest neighbors can be defined in a spatial sense to be nodes in the same layer with close indices, or it can also be defined to include nodes in previous and future layers. The memory bank update equation for each node is given by

$$\mathbf{M}_j^{(i)} := \gamma^{\left\lfloor x_j^{(i)} \right\rceil}\mathbf{M}_j^{(i)} + g\left(\left[x_{j-\tau/2}^{(i)}, \ldots x_j^{(i)}, \ldots, x_{j+\tau/2}^{(i)}\right]\right) \tag{11}$$

where $g(x) : \mathcal{R}^{\tau+1} \rightarrow \mathcal{R}^{2^{\tau+1}}$, is a function mapping the outputs of its τ nearest neighbors to $2^{\tau+1}$ dimensional space, and γ is the given decay schedule which is turned on and off by the rounded output of the standard node. When a node fires $\gamma^{\left\lfloor x_j^{(i)} \right\rceil}$ evaluates to γ, and if it does not fire $\gamma^{\left\lfloor x_j^{(i)} \right\rceil}$ evaluates to 1.

During the training phase memories are allowed to decay to zero quickly to provide the network the plasticity to converge to its desired orientation. During use memories can be retained indefinitely, or according to any desired decay schedule. With an asymptotic memory decay schedule, transient information can be retained for the more recent memories, and any remaining memories which have hit the asymptotic magnitude are still retained, but their time ordering can no longer be compared to other memories which have also hit the asymptotic decay magnitude.

IV. Experiments/Results

The NMN algorithm was implemented and then tested with several test data sets:

- *Sorted Classification*
- *Classification and Recall*

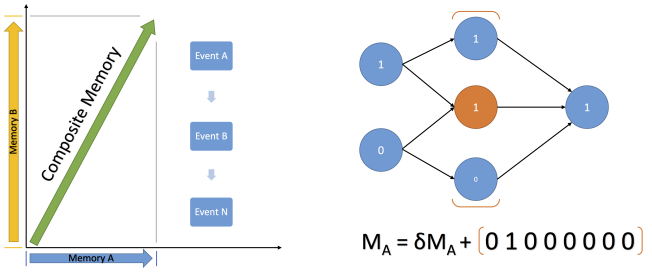


Fig. 4: Each standard node stores a composite memory in its dedicated memory bank. A composite memory is the sum of all of the orthogonal memories associated with that node, such that the original memories are retained by projecting the composite memory onto different dimensions. In the nearest neighbor approach the activation permutations of a cell's nearest neighbors is mapped to a higher dimensional orthogonal memory and then added to the cell's composite memory.

- *Memory Recall*
- *Extended Recall*

Each of the test sets are meant to evaluate the various functionality of the algorithm. For comparison, a standard FFNN (NMN with memory turned off) and an RNN available in the MatLab Neural Network toolbox (*layrecnet()*) were also trained and evaluated with the test data sets. *layrecnet*() has one hidden layer with feedback and one output layer; a block diagram of is shown in Fig. 6.

A. Sorted Classification

A data set composed of points sampled from 10 randomly generated multivariate Gaussian distributions, with the goal being to classify which distribution a point was sampled from. Samples were drawn from the distributions in a stationary order. Due to overlap in the distributions a standard feed forward neural net was only able to achieve 4.5% classification error. By implementing memory the neural network was able to identify that there was a pattern to the input points and reduces classification error to 0.7%, an improvement of 600%. An RNN with a single feedback delay was also tested on the data, with classification error of 1.8%. Convergence results are shown in Fig. 7.

B. Classification and Recall

The task of this data set was to classify an input and also state the previous two classifications it made, in the order it made them. A memory-less neural network can perform no better than converging to the most prolific output on this type of task. A memory based neural network with the same parameters was able to achieve 7% classification error on the same data set. An RNN with a single feedback delay was also tested on the data, with classification error of $< 0.1\%$. Convergence results are shown in Fig. 8.

C. Memory Recall

The networks were given sequences of 5 bit numbers to write into memory, followed by empty input vectors of zeros

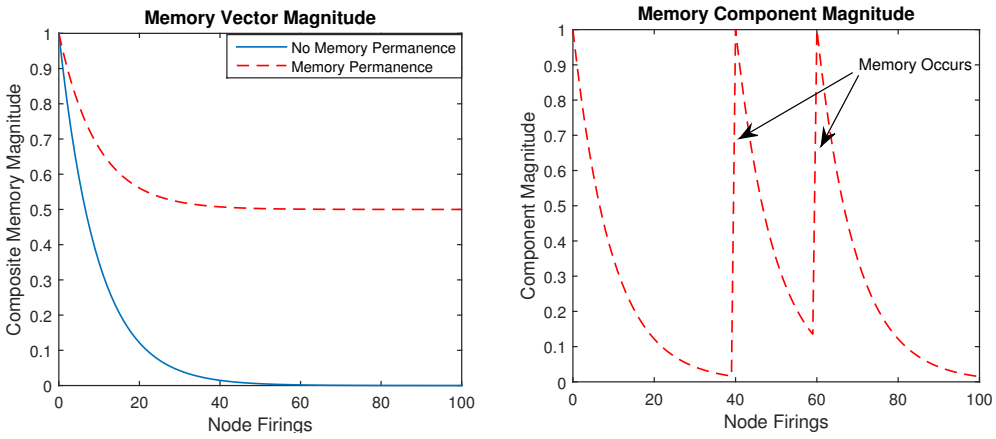


Fig. 5: Every time a standard node is activated that memories associated with that node have their magnitude decremented according to some decay schedule to allow for the retention of temporal information. Memories can either decay out of existence, or can be given some asymptotic decay.

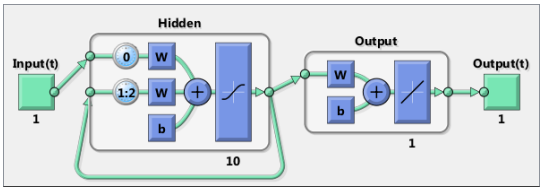


Fig. 6: Block diagram of *layrecnet*(), a layered recurrent neural network function available in the MatLab Neural Network Toolbox.

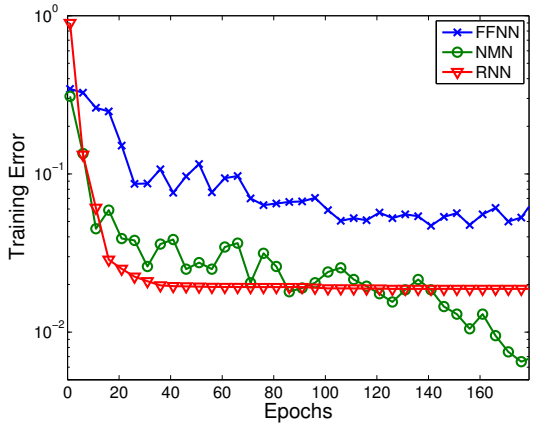


Fig. 7: A comparison of error convergence between an FNN, NMN, and RNN on a classification task demonstrates the ability of the NMN to leverage temporal patterns in the input data.

of the same series length to signify to the networks to read and output the stored sequences. The NMN algorithm was set with one internal memory layer to process the data. Both memory writing procedures were tested, external writing which recorded the input layer upon a standard neuron firing, and nearest neighbor which recorded nearby neuron outputs as outlined above. Convergence results for sequences of length two are shown in Fig. 9.

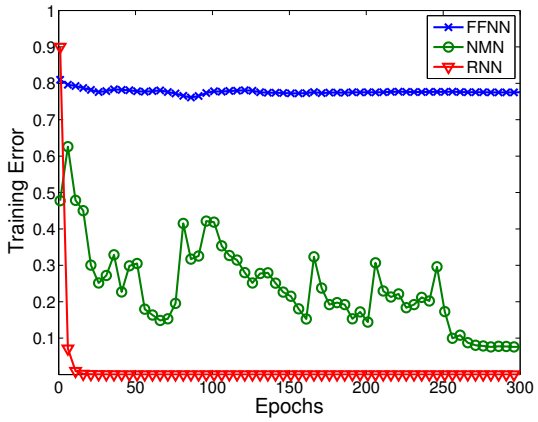


Fig. 8: A convergence plot for the classification and recall task, where outputs are given by the current classification and the previous two classifications.

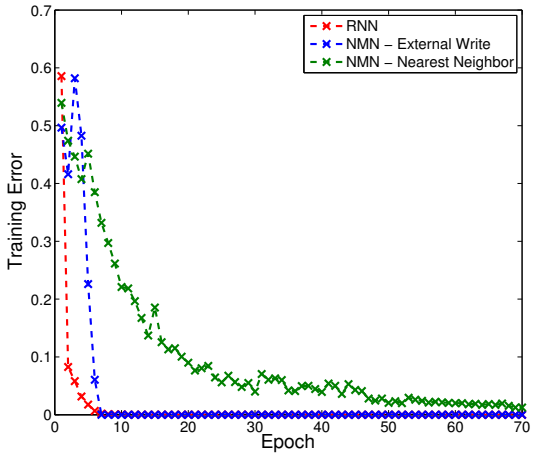


Fig. 9: Convergence plot of the memory recall task for the various networks, RNN, NMN with an input write function, and NMN with nearest neighbor write function.

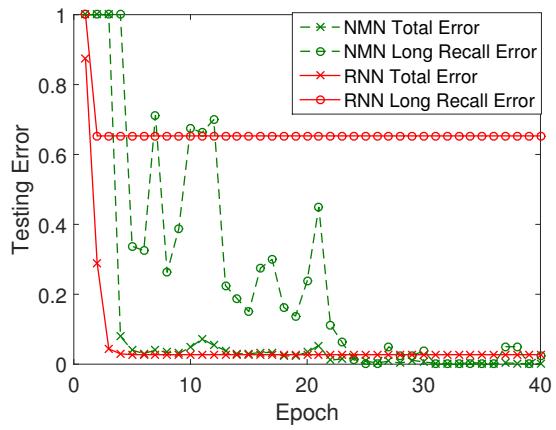


Fig. 10: Convergence plot for the extended recall task. In this data set, some classifications depend only on the current inputs, but some depend on inputs from 25 time units ago. A standard RNN is unable to perform those long term classifications better than randomly guessing, while the NMN is able to achieve near perfect long term recall.

D. Extended Recall

In the extended recall data set the task was to perform classifications, with some classifications depending only upon the current input, and some classifications depending on information from 25 inputs prior. Both the RNN and the NMN were able to achieve perfect testing performance on the classification task where the outputs only depending upon the current input. However, in the case of outputs depending upon data from 25 inputs ago the RNN was only able to achieve random guess (66% error) levels of performance, while the NMN was able to achieve near perfect long term recall. The NMN in this setup was composed of 2 hidden layers. Convergence results for all four error rates are shown in Fig. 10.

V. Conclusion

We have developed and demonstrated a feed-forward neural network algorithm that stores memory associated with standard hidden layer neurons. The benefit of neural memory is shown to give context to the algorithm while computing pattern recognition and time series data sets. Memories are stored with a higher dimensional mapping to ensure orthogonality, and memories are decremented upon recall to maintain a time history. The neural memory network was evaluated with several data sets of classification and recall and against a similar feed-forward neural network without memory and a commercial recurrent neural network. The results show that although indeed the neural memory network indeed learns to utilize memory in order to solve problems requiring context, in the current implementation it does not outperform the commercial recurrent neural network for most tasks. On the other hand, neural memory allow for the network to excel at long term memory storage and recall, a functionality that escapes the standard recurrent neural network.

VI. Future Work

The algorithms described in this paper are only a first approach towards the NMN architecture. Future developments will depend upon un-linking the read and write operators, memory scaling issues, training schedules, and the general optimization techniques used to perform gradient descent.

In this implementation the NMN learns to recall information from memory as needed, but it is not explicitly trained on how and when to write information. To combat this, in our current implementation we have tied together the write and read commands so that any potentially useful information is saved to memory in the appropriate memory slots as determined by the read command. However, to get a more robust performance the read and write commands need to be learned separately and independently.

In the current nearest neighbor approach, the size of the memory banks scale according to 2^τ, which can cause major performance issues if τ becomes too large. This constraint is introduced to maintain the orthogonal nature of the memories to avoid the problem of having to learn to reference specific memories. With orthogonal memories, a composite memory is referenced instead of a single component memory. Instead of this approach, the permutations of the nearest neighbors can be used to address a specific memory location in a nodes memory bank through a binary to linear mapping, thus eliminating the orthogonal constraint and the 2^τ scaling law. An example output of a memory node with 2 nearest neighbors with this approach could be given by

$$\begin{aligned}\mathbf{M_1} = (1-x_3)(1-x_2)x_1M_{1,1}+\\ (1-x_3)x_2x_1M_{1,2}+\\ x_3(1-x_2)x_1M_{1,3}+\\ x_3x_2x_1M_{1,4}\end{aligned} \tag{12}$$

which is a differentiable expression for selecting a memory from a learned location. In this expression x_2 and x_3 are the outputs of the two nearest neighbors. This expression could be further processed with some type of focusing technique, but this approach allows for linear memory scaling and does impose an orthogonal constraint.

To allow for more accurate comparisons between the NMN and other off the shelf algorithms the optimization techniques used to perform gradient descent on the NMN can be improved upon to allow for better convergence. The implementation of a dynamic step-size achieved via a line search could prove to be very effective.

References

[1] Weston, Jason, Sumit Chopra, and Antoine Bordes. *Memory networks*. arXiv preprint arXiv:1410.3916 (2014).
[2] Graves, Alex, Greg Wayne, and Ivo Danihelka. *Neural Turing Machines*. arXiv preprint arXiv:1410.5401 (2014).