Production Rule Recurrent Net

Bojian Han
Carnegie Mellon University
Pittsburgh, PA 15213
bojianh@andrew.cmu.edu

Abstract

In this paper, I proposed a new neural network architecture capable of simulating ACT-R production rules after training the network using data generated from these production rules. This paper will demonstrate the ability of the network to learn the ACT-R model from *addition.lisp* [4]. This paper will also explore different data representation of numbers and their effects on test results.

1 Introduction

ACT-R architecture contain both symbolic and subsymbolic components. The symbolic components in ACT-R provides an abstract representation of how the brain encodes knowledge while the subsymbolic components such as chunk activation and production rule utility in ACT-R provides an abstract representation of the lower level neural computation [5].

There have been prior works to provide an connectionism implementation of ACT-R in the past. Lebiere, et. al have implemented a connectionist version of ACT-R architecture using recurrent neural networks to show it is indeed possible[1]. More recently, Stocco, et. al have focused more on building a more biologically based neural network architecture with various routing connections in attempt to simulate the basal ganglia [2, 3].

In this paper, I have decided to incorporate ideas for both of the previous work to implement *Production Rule Recurrent Net* to model ACT-R production rules with the left hand side of the network (*LHS Net*) more resembling the work by Lebiere, et. al. and the right hand side of the network (*RHS Net*) more resembling the work by Stocco, et. al. Additionally, there is an addition of a comparison layer to facilitate network training using backpropagation through time (BPTT).

2 Production Rule

To first understand the network architecture, it is important to first understand production rules as stated in this section.

There are two forms of knowledge representation in ACT-R, declarative knowledge and procedural knowledge[4, 5]. Declaraive knowledge refers to facts we are conscious of and can describe this to others. Procedural knowledge refers to knowledge we display in our behavior that we are not conscious of. In ACT-R, declarative knowledge is represented as chunks and procedural knowledge is represented as production rules.

A production rule can be represented as a if-then rule or a condition-action pair. A production rule is fired if all the conditions of the rule are satified. Then the action of the production rule will be performed. I have included 4 production rules from *addition.lisp*[4]. The conditions of the rule are to the left hand side before the ==> symbol. The actions of the rule are to the right hand side of the ==> symbol. For instance, *Initialize-Addition* rule will be fired if the *arg1* and *arg2* are some numbers and *sum* is *nil*. Note that *nil* does not mean 0, instead it meant that this slot is not assigned.

Production Rule 1 Initialize-Addition

```
(P initialize-addition
   =goal>
      arg1          =num1
      arg2          =num2
      sum           nil
  ==>
   =goal>
      sum           =num1
      count         0
   +retrieval>
      first        =num1
)
```

Production Rule 2 Terminate-Addition

```
(P terminate-addition
   =goal>
      count         =num
      arg2          =num
      sum           =answer
  ==>
   =goal>
      count         nil
)
```

Production Rule 3 Increment-Count

```
(P increment-count
   =goal>
      sum           =sum
      count         =count
   =retrieval>
      first         =count
      second        =newcount
  ==>
   =goal>
      count         =newcount
   +retrieval>
      first        =sum
)
```

ACT-R architecture contains a number of modules with their correponding buffers. In the production rule *Terminate-Addition*, *=goal>* refers to the goal module buffer. Each of the buffer contains a chunk which is a set of slot-value pairs. For instance, *count* is the slotname in goal module buffer and *=num* is the value assigned to this slot. Note that the = prefixed in front of *num* means that *=num* is a local variable instead of an actual value. This allows this production to match arbitrary values as well as perform comparison between variables. For instance, in order for *Terminate-Addition* to fire, both *count* and *arg2* must equal to *=num*. This is equivalent to *count==arg2* in other programming languages. Hence, if *count=2*, *arg2=2* and *sum=7*, this production rule will fire but if *count=2*, *arg2=3* and *sum=7*, this production will not fire. Additionally, in the production rule *Increment-Sum*, the $-$ sign in $-$ *arg2 =count* means not equal to. Hence, this is equivalent to *count!=arg2* in other programming languages.

There are additional notations in ACT-R such as request for a buffer in $+$*retrieval*$>$ and $>$ and $<$ sign instead of $-$ sign for comparison but those are not in the scope of this paper for understanding this network model.

Production Rule 4 Increment-Sum

```
(P increment-sum
   =goal>
      sum          =sum
      count        =count
    - arg2         =count
   =retrieval>
      first        =sum
      second       =newsum
  ==>
   =goal>
      sum          =newsum
   +retrieval>
      first       =count
)
```

Table 1: Addition of $5 + 2 = 7$

Production Rule	Left Hand Side (Condition)						Right Hand Side (Action)					
	Goal				Retrieval		Goal				Retrieval	
	Arg1	Arg2	Count	Sum	First	Second	Arg1	Arg2	Count	Sum	First	Second
Initialize-Addition	5	2	nil	nil	nil	nil	5	2	0	5	5	nil
Increment-Sum	5	2	0	5	5	6	5	2	0	6	0	nil
Increment-Count	5	2	0	6	0	1	5	2	1	6	6	nil
Increment-Sum	5	2	1	6	6	7	5	2	1	7	1	nil
Increment-Count	5	2	1	7	1	2	5	2	2	7	2	nil
Terminate-Addition	5	2	2	7	2	3	5	2	0	7	2	3

Using the 4 production rules as shown above (1, 2, 3, 4), we can perform addition of numbers as shown in Table 1. In this example, the ACT-R model performed a sequence of steps to add 2 to 5 to form 7 initially starting at *Initialize-Addition* and ending at *Terminate-Addition*. The final output will be value in slot *sum* in the goal module buffer. Note that the request for retrieval is not shown in the table. In this example, the retrieval acts as a successor function which increment 1 on *first* and put the value in *second*.

3 Production Rule Recurrent Net Model

The proposed model attempts to simulate ACT-R production rule system, effectively simulating behavior of the condition-action pair in the ACT-R production rule. In order to accomplish this task, the *Production Rule Net* is split into 2 sub-networks *LHS Net* and *RHS Net*. The function of *LHS Net* is to correctly classify which production rule should respond using the input. The function of *RHS Net* is to correctly generate the response of the production rule right hand side given the input and the correct production rule.

3.1 Left Hand Side Network (LHS Net)

The *LHS Net* architecture is shown in Figure 1. The input to the network is shown on the diagram as *LHS Layer* or l_t at time t. For the addition task in *addition.lisp*, the size of l_t is 6. There is a hidden layer *Comparison Layer* or c_t which accomplish the task of comparison between variables. The motivation behind the c_t is that variable comparison is a rather difficult task since checking if two variables are equal is at least as difficult as the XOR task which requires one hidden layer in a multi-layer perception (MLP). The size of this layer is $O(N^2)$ where N is the size of the input. Hence the size of this layer for the addition task is 36. The output layer is *Fire Layer* or p_t which represents the probability of different production rule firing given the input. The size of this layer equal to the number of production rules in the ACT-R model which is 4 in this case.

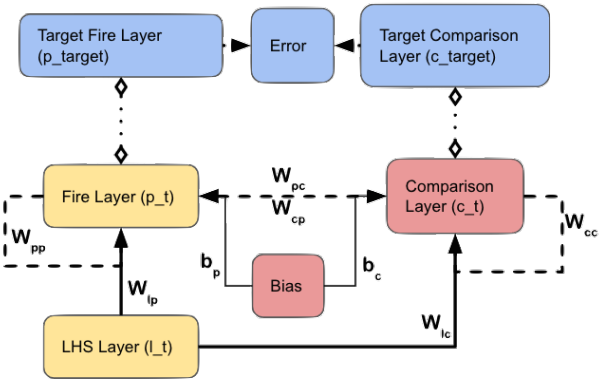

Figure 1: Left Hand Side Network Architecture (LHS Net)

The loss function for this network is shown in Equation 1.

$$L(l_{1...T};\theta) = \frac{1}{T}\sum_{t}\frac{1}{N^2}||c_{target} - c_t||^2 + \frac{1}{P}||p_{target} - p_t||^2 \tag{1}$$

T is the total number of time step to run *LHS Net*. N is the number of input units to l_t or $|l_t|$. P is the number of production rule in the ACT-R model or $|p_t|$. c_{target} refers to the target comparison matrix and p_{target} refers the target production rule one-hot vector. θ refers to the set of weight matrices $\{W_{lc}, W_{lp}, W_{cc}, W_{cp}, W_{pc}, W_{pp}, b_c, b_p\}$.

The *Comparison layer* c_t at time t is computed in Equation 2.

$$c_t = \sigma(W_{lc}l_t + W_{cc}c_{t-1} + W_{pc}p_{t-1} + b_c) \tag{2}$$

σ refers to the sigmoid function. W_{lc} refers to the weight matrix between the l_t and c_t. W_{cc} refers to the recurrent weight matrix between itself on the previous timestep. W_{pc} refers to the recurrent weight matrix between p_{t-1} which the production rule vector on the previous timestep. b_c is the bias to c_t.

The *Fire layer* p_t at time t is computed in Equation 3.

$$p_t = \sigma(W_{lp}l_t + W_{cp}c_t + W_{pp}p_{t-1} + b_p) \tag{3}$$

W_{lp} refers to the weight matrix between the l_t and p_t. W_{cp} refers to the feedfoward weight matrix between the c_t and p_t. W_{pp} refers to the recurrent weight matrix between the itself on the previous timestep. b_p is the bias to p_t.

The reason for the recurrent weight connections is to simulate the decision making process in the basal ganglia which involve different production rules competing to fire by trying to increase its own probability of firing and suppressing the other production rules from firing.

3.2 Right Hand Side Network (RHS Net)

The *RHS Net* architecture is shown in Figure 2. The input to the network is l_t and p_t from the *LHS Net*. The whole network is essentially a gated multi-layer perception. *Broadcast layer* b_t is a hidden layer that is broadcasted from the input l_t. *Gating layer* g_t is the gating that is applied onto the b_t to form *Broadcast Gated layer* bg_t. The size of b_t, g_t and bg_t are all $O(N^2)$ relative to the input l_t and hence 36 in this addition task. The output layer is *RHS Layer* r_t which is of size $O(N)$ and 6 in this case. The reason for the *Broadcast layer* is provide the necessary information pathways for all pairwise reassignment of slot values in the right side of the ACT-R production rule which is $O(N^2)$ in input size. The reason for *Gating layer* is to provide the right level of suppression so that

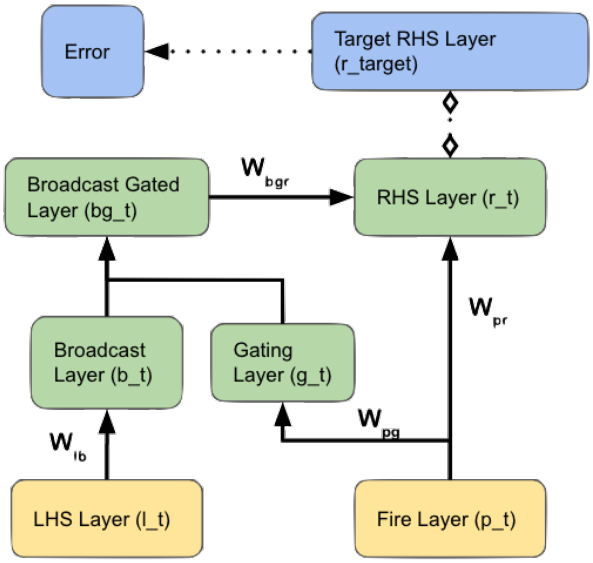

Figure 2: Right Hand Side Network Architecture (RHS Net)

reassignment of slot values will be correct instead of summation of all the values from the *Broadcast layer*.

The loss function for this network is shown in Equation 4.

$$L(l_{1...T}, p_{1...T}; \theta) = \frac{1}{N}||r_{target} - r_T||^2 \tag{4}$$

T is still the total number of time step to run *LHS Net*. N is the number of input units to l_t or $|l_t|$. r_{target} refers to the target right side output of the correct production rule. r_T refers to the target output at the last timestep T. θ refers to the set of weight matrices $\{W_{lb}, W_{pg}, W_{pr}, W_{bgr}\}$.

The *RHS layer* r_t at time t is computed in Equation 5.

$$r_t = W_{bgr}(W_{lb}l_t \odot \sigma(W_{pg}p_t)) + W_{pr}p_t \tag{5}$$

Note that the computation for r_t contained the computation for the layers b_t, g_t and bg_t. W_{lb} is the weight matrix from l_t to b_t. W_{pg} is the weight matrix from p_t to g_t. W_{bgr} is the weight matrix from bg_t to r_t. W_{pr} is the weight matrix from p_t to r_t. The symbol $\odot$ stands for element-wise multiplication. Note that the *Gating layer* g_t is conditioned on the production rule and the *Broadcast layer* b_t is conditioned on the input. The reason for W_{pr} is that this will allow the network to simulate ACT-R production in setting certain values in the right side that is not directly from the input such as setting *count* to 0 in *Initialize-Addition*.

3.3 Whole Network

The diagram for the whole is shown in Figure 3 by merging *LHS Net* and *RHS Net* into a single network.

4 Methods

4.1 Data Representation

There are 3 data representations used in this paper. The first one is the naive representation. In this representation, the integers from 0 to 10 is mapped uniformly into the real number interval of [0,1].

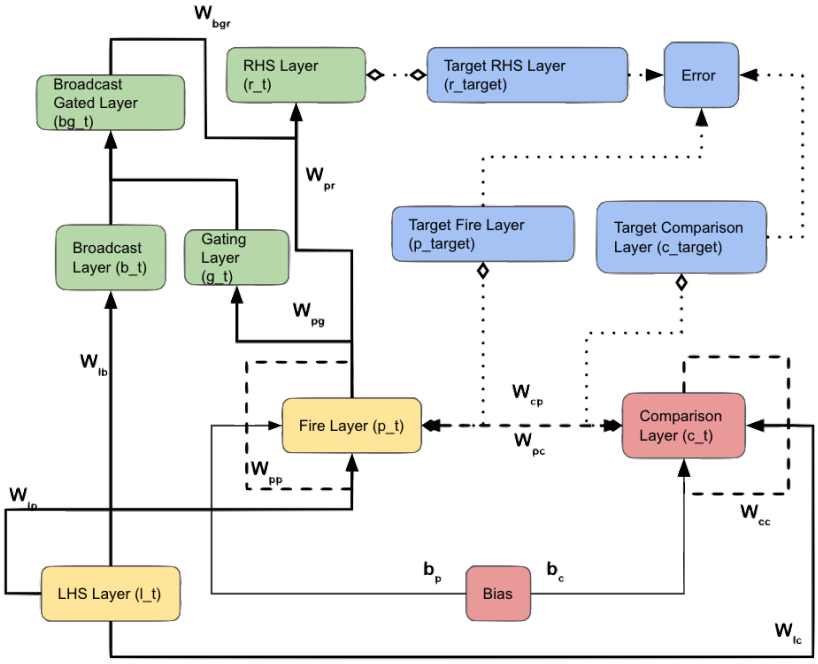

Figure 3: Whole Network. The yellow boxes are layers in *LHS Net* and *RHS Net*. The red boxes are layers only in *LHS Net*. The green boxes are layers only in *RHS Net*. The blue boxes refer to targets for backpropagation of errors. The bold solid arrows refer to feedforward connections. The dashed arrows refer to recurrent connections. The thin solid arrows refer to bias connections. The thin dotted diamond arrows refer to pathways for backpropagation of errors.

For instance, the number 5 is mapped to 0.5 and the number 2 is mapped to 0.2. *nil* is mapped to -1 in this representation. For values that are not a number in ACT-R chunks, all the possible values are first indexed and this index will be mapped uniformly to the real number interval of [0,1]. The benefits of this representation is that this will preserve the ordering properties of the integers as compared to a more distributed representation. The downside however is that this representation will introduce ordering properties for the values which are not numbers which can lead to inaccuracies. Another additional benefit of this representation is that it is computationally less expensive to implement the network using this representation since the *Comparison layer* c_t is of $O(N^2)$ in size and the recurrent weight W_{cc} is $O(N^4)$ in size ($36 \times 36 = 1296$ for this task). If one-hot encoding is used instead, the size of input l_t will be $6 \times 12 = 72$ since there are 11 integers from 0 to 10 inclusive and 1 value for *nil*. This means that the resultant weight matrix will have $72^4 = 26,873,856$ parameters which makes much harder and longer to train.

The second data representation (repeated encoding) is to simply repeat the the naive representation several times in order to boost accuracy. This can work because the input weights are random and hence the network can use more parameters to discriminate between decimal values. In this case, if we repeat the input vector by 3, we will be increasing N to 18 from 6 for this task.

The last data representation (nil encoding) is to separate out *nil* from the input vector and allow the integers to map to a bigger interval. In this representation, each variable will be represented by a vector of 3 real numbers from the interval of [-1,1]. The first number maps to either -1 or 1 depending on if the value is *nil* or not. The second number is result after mapping integer 0 to 10 uniformly to the interval [-1,1] and the third number is mapping the integer uniformly from 0 to 10 in reverse to [-1,1] in attempt to maximize the differences in representation. For instance, the number 7 will be mapped to the vector [1, 0.4, -0.4].

4.2 Generating Data

In order to generate training data and testing data for the *Production Rule Recurrent Net*, I have to first parse the production file to create a data structure containing the details of the production rule such as the slot-value pairs in total across the chunks of all the modules in the file as well as local variable binding for comparison purposes. The parsing of the production rule is done in Python.

Generation of random data for production rules is done in the following way. The first step is to randomly select a production rule. For the left hand side of the production rule, the code sequentially go through each slot-value pair and generate random values that fit the constraints of the condition. If the slot-value pair contains a fixed value, then that value is automatically assigned to be part of the data. If the slot-value pair contains a local variable instead, the local variable is first given a random value and the the results of that variable is saved. When the local variable is encountered at the again at some new slot-value pair, the local variable will be assigned to the new slot-value pair. If the comparison uses the not-equal-to operator $-$, the value is first assigned to that slot-value pair and later replaced off with a different random value. All these steps are done to ensure that the resultant data indeed satisfy all the conditions of the production rule. The resultant vector will be fed into l_t.

The right hand side of the production rule will follow from the local variable saved from the left hand side of the production rule as well as static values assigned by the production rule. This vector will be fed into r_{target} for computing the loss function. The comparison matrix is generated by taking the pairwise comparisons values between the inputs. This can be approximated by first tiling the input vector by N, taking the tiled input minus the tiled input transposed, and then multiply by a huge constant (100) and clipping the values between 0 and 1. This vector will be fed into c_{target} for computing the loss function. Lastly, one-hot vector for the production rule is also generated to be fed into p_{target} for computing the loss function. Note that the one-hot vector is equivalent of 100% probability of the correct production rule and 0% for all other production rules.

4.3 Training

Production Rule Recurrent Net is written in a Python package called Theano and trained using backpropagation through time (BPTT) algorithm. The training is done using Stochastic Gradient Descent (SGD) using a batch size of 10. There is no momentum and the network is trained for 6000 epoches. In each epoch, a batch of 10 random training data is generated using the data generation code. Due to the complexity of the network and the gating mechanism, there are different learning rate and decay parameters for the different weights as shown in Table 2. The reason for the large learning rate to c_t is for the network to focus on linear portion of the sigmoid curve for positive values and the horizontal portion of the simgoid curve for the negative values. Large learning rate to g_t is to allow the gating matrix to push to 0 and 1 values of the sigmoid function. I considered using ReLu as the nonlinear function for c_t and g_t but could not find the right set of learning parameters for it.

Table 2: Learning Rate and Decay Rate for Various Weights

Set of Weights	**Learning Rate**	**Decay Rate**
$\{W_{lp}, W_{lb}, W_{cp}, W_{pp}, W_{pr}, W_{bgr}, b_p\}$	3	0.999
$\{W_{lc}, W_{cc}, W_{pc}, b_c\}$	100	0.995
$\{W_{pg}\}$	1000000	0.999

4.4 Testing

Testing is done on all 66 unique addition tasks in which the sum ranges from 0 to 10. For each addition task, the network will be run repeatedly using the output vector from the previous run after retrieval as the new input vector. Retrieval done using the value in slot *first* in the retrieval buffer and compute the value for the slot by incrementing the value before mapping by 1 (0.1 in the mapped version). The running of the network will stop once the network outputs *Terminate-Addition* as the production rule or if the network timed-out after it failed to stop after $2\times$*arg2*$+2$ steps which is the

correct number of production rules fired in the ACT-R for this model. The breakdown is as follows, 1 for *Initialize-Addition*, 1 for *Terminate-Addition*, $2\times arg2$ for *Increment-Sum* and *Increment-Count* loop. The resultant output vector will be used to compute the actual *sum* value rounded to the nearest integer. The test error is computed by taking the absolute difference between the actual answer and the nearest integer outputed by the network.

5 Results

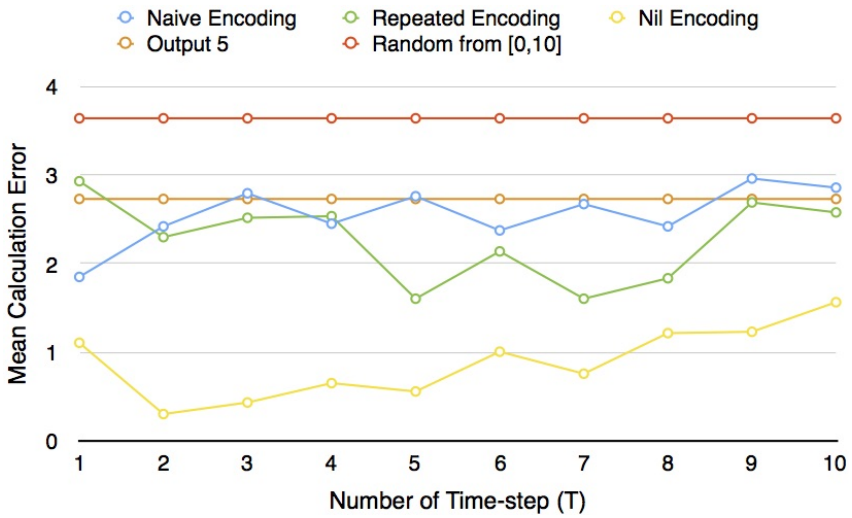

Figure 4: Plot of mean absolute error of computation for the 3 different data representation types averaged over 5 different training instances. The 3 models are benchmarked against an algorithm that only outputs 5 and an algorithm that outputs a number randomly from 0 to 10.

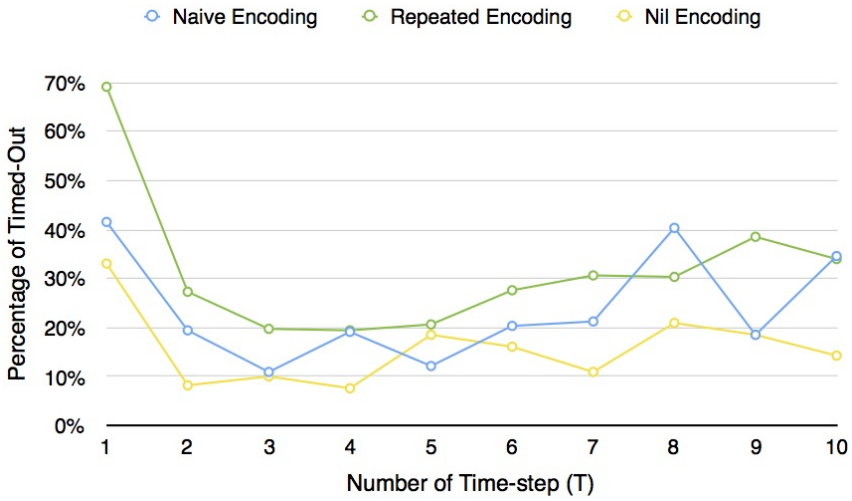

Figure 5: Plot of percentage of timed-out for the 3 different data representation types averaged over 5 different training instances.

5.1 Different Data Representation Types

The network is trained with all 3 encoding types with the parameter total time step T set from 1 to 10 as shown in Figure 4 and Figure 5. In order to compare the performance between different encoding types and different total time-step T, the network is trained on the same set of 60000

randomly generated training data. The network performance is benchmarked by the performance of an algorithm that just outputs 5 and an algorithm that outputs an integer sampled uniformly from [0,10]. The performance at each point is the average performance over 5 different training instances using the same type of parameters.

The plot shows that the Naive Encoding has similar performance as the algorithm that just outputs 5 with the exception of $T = 1$. Repeated Encoding has slightly better performance when the total number of time-step T is between 5 and 8. The Nil Encoding achieved the best performance out of the 3 data representation types with an average error of around 0.88 averaged across T with the best performance at $T = 2$ with an average error of about 0.30. This shows that using the Nil Encoding is indeed better than the rest of the data representation schemes. However, it is unclear at this stage whether the performance gain is a result of separating out the *nil* or using the normal and reverse mapping or a combination of both. The results suggest that it is probably indeed better to have a more distributed but sparse representation for numbers.

Another interesting result from this is that more time-step T does not translate to better performance on testing. For instance, the best performance of Nil Encoding is at $T = 2$ while the best performance of Repeated Encoding is at $T = 5$. The Timed-Out plot on Figure 5 showed a similar trend. If $T = 1$, the percentage of Timed-Out is much higher compared to the rest, demonstrating that recurrent weight connections are indeed useful. However as T increases beyond a threshold, the percentage of Timed-Out increases. This suggest the production rule *Terminate-Addition* is not fired at the right time which can lead to performance errors because the network performed different sequence of production rules instead. This also seem to suggest that having bigger T lead to the network spending more time choosing the correct production rule which may mean that network might 'overthink' the problem and choose the wrong production rule *Increment-Sum* instead of *Terminate-Addition*.

5.2 Effects of *arg1*, *arg2* and *arg1+arg2*

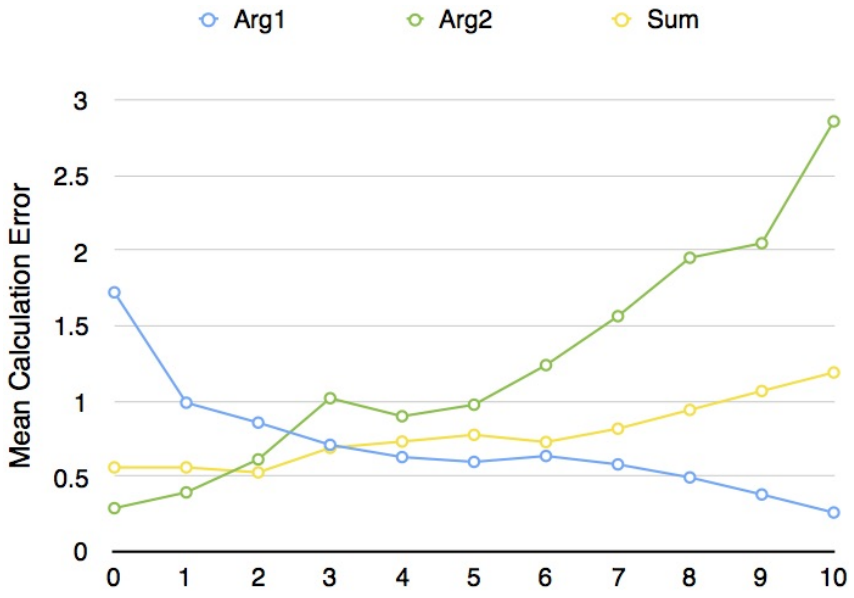

Figure 6: Plot of mean absolute error of computation for the effects of *arg1*, *arg2* and *arg1+arg2* averaged over all the training instances for Nil Encoding.

I did further analysis on the results of Nil Encoding to figure out the effects of *arg1*, *arg2* and *arg1+arg2* on the performance and percentage of timed-out training instances of the network as shown in Figure 6 and Figure 7.

The mean calculation error for *arg2* increased as *arg2* increased. The same effect is observed on the timed-out percentage as well. This makes sense because as the number of iterations that the network

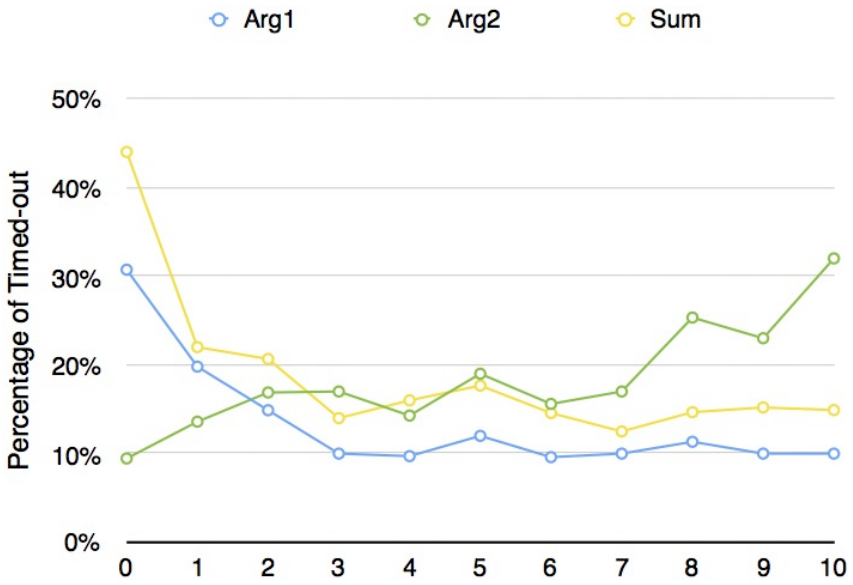

Figure 7: Plot of percentage of timed-out for the effects of *arg1*, *arg2* and *arg1*+*arg2* averaged over all the training instances for Nil Encoding.

runs is proportionate to *arg2*. A bigger *arg2* means that the network will run for more iterations and accumulate more errors for each iteration.

The mean calculation error for *arg1* decreased as *arg1* increased and similar effect is seen on the timed-out percentage. One possible reason is that in the 66 unique addition task, *arg1*+*arg2* is between the interval [0,10], as a result, the average *arg2* value for a small *arg1* is higher than the average *arg2* value for a big *arg1*. For instance, there are 11 addition tasks for *arg1*=*0* and the average value of *arg2* is 5.5 but there is only 1 addition task for *arg1*=*10* and the average value of *arg2* is 0 instead. Therefore, the trend of *arg1* is the opposite to that of *arg2*.

The mean calculation error for *arg1*+*arg2* increases slightly as *arg1*+*arg2* increases which does make sense because the average *arg2* value increases as *arg1*+*arg2*. However, the intriguing result is that timed-out percentage plot shows the opposite trend as seen in Figure 7 which means the network is more likely to fail to fire *Terminate-Addition* when *arg1*+*arg2* is small. It is currently unclear what exactly lead to this effect.

6 Discussion

In this paper, I have demonstrated that it is possible to use a neural network model to simulate ACT-R production rules, showing that symbolic processing in ACT-R can be implemented in a biological plausible manner. However, this network demonstrated that it is possible for errors in matching for the correct production rule and errors in generating the correct response for the right hand side output in which the current ACT-R does not have.

The results showed that data representation types does affect the type of testing performance with Nil Encoding with $T = 2$ having the lowest errors among all the other training instances with different representation and T parameter.

We can relate some of the results of this network to behaviors in real humans. The phenomenon of network timeout may be used to explain why people can get stuck in a thinking loop if they fail to differentiate between values between different slots. Performance differences due to different total time-step T can be used to explain why humans are less accurate if they overthink a problem.

On a sidenote, it is interesting to note that ACT-R model can be used to simulate code in other programmming languages. Hence it is possible to reduce code in other programming languages to

ACT-R model and then reduce from ACT-R model to this network. This means that this network is able to execute short snippets of code with a limited memory.

7 Acknowledgement

I would like to thank Professor John Anderson and Research Faculty Christian Lebiere for their guidance on this project. I would also like to thank Dan Bothell for answering many questions related to ACT-R programs and interface.

References

[1] Lebiere, C., Anderson, J. R. (1993). A connectionist implementation of the ACTR production system. In Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society (pp. 635 640). Mahwah, NJ: Erlbaum.

[2] Stocco, A. et al. (2010) Conditional routing of information to the cortex: a model of the basal ganglias role in cognitive coordination. Psychol. Rev. 117, 541574

[3] Stocco, A., Lebiere, C., O'Reilly, R. C., Anderson, J. R. (2010). The role of the anterior prefrontal-basal ganglia circuit as a biological instruction interpreter. In A. V. Samsonovich, K. R. Jhannsdttir, A. Chella, B. Goertzel (Eds.) Biologically Inspired Cognitive Architectures 2010. Frontiers in Artificial Intelligence and Applications, (pp. 153-162). Amsterdam, The Netherlands: IOS Press.

[4] ACT-R Research Group. (2013). Tutorial Units. http://act-r.psy.cmu.edu/software/

[5] Anderson JR. How can the human mind occur in the physical universe? 1. New York, NY: Oxford University Press; 2007.