Microprocessor Modeling with RNNs: A Hippocampal Analogy Approach

Aditya Nanda Kishore, EE20B062 ; Srinivasan Kidambi, EE21B139

Guided by: Prof. Srinivasa Chakravarthy, Vigneswaran Chandrasekaran

May, 2023

Abstract

The hippocampus is a region in the brain that plays an important role in memory formation and navigation. It is involved in the consolidation of information from short-term memory to long-term memory and is also crucial for spatial navigation and mapping of the environment. Hippocampus is made of recurrent layers of memory processing units that can help it achieve the same. Utilizing the hippocampal analogy as inspiration for this computational model, we have the potential to model microprocessor behaviour that can recall necessary information through the use of learned parameters.

1. Introduction

The primary goal at hand is to model the microprocessor behaviour. When a sequence of instructions is given as input to the model, it must be able to compile the set of instructions given and output the results accordingly. Considering the engineering freedom in this problem statement. The model's complexity depends on the dataset being captured. As a first step, a stack model can be employed, where data is sequentially provided and subsequently retrieved in reverse order.

1.1. *Problem Description*

To implement the stack in microprocessor context, these steps were followed through.

1. Creating the microprocesssor dataset. One data point would be a sequence of instructions as input and the corresponding outputs as output.
2. Next step is to convert the input sequences into a suitable format for the RNN model. For example, to model the execution of instructions, one can represent each instruction as a sequence of binary values or as one-hot encoded vectors.
3. The training data should consist of input sequences and corresponding target sequences that represent the expected output of the microprocessor. Next step is to train the RNN model using a suitable loss function and an appropriate training algorithm. Various approaches, models and the results obtained are attached below.

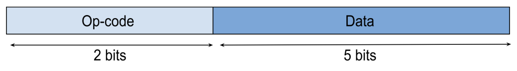

Figure 1: Stack-Instruction

2. Dataset

We have worked on two datasets primarily, the Stack Dataset and an instruction set architechture that incorporates addressing.

2.1. *Stack Dataset*

The stack dataset doesn't require addressing. So an op-code and the data are the only necessary things in the instruction. An intruction is modelled as shown in Figure 1. Here there are only 3 opcodes- 00(Store), 01(Maintain), 10(Retrieve).

$$Input\ Sequence\ X_s = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 18 \\ 7 \\ 64 \\ 64 \end{bmatrix}$$

$$Output\ Sequence\ Y_s = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 96 \\ 96 \\ 7 \\ 18 \end{bmatrix}$$

The above example has 4 instructions, where in the first two instructions we are asking it to store two data points of 5-bits each and in the next two instruction it has been asked to retrieve. Each instruction can be looked as a binary encoded sequence and can be converted into a unique value

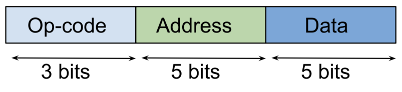

Figure 2: Load-Instruction

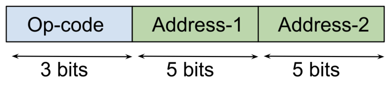

Figure 3: ALU-Instruction

by converting it to decimal value later. Advantage of this modelling is one need not fix the number of columns in the matrix, helping us accomodate variable data and addressing size later. '96' is just a buffer output to account for padding.

2.1.1. Nomenclature for the stack dataset

- Data Size: Corresponds to number of bits of data. Worked with 5 bits of data in each instruction here.
- Store Length: Corresponds to half the number of rows of the matrix. The maximum number of store instructions an input sequence can have. We have worked with this as 8 in the beginning and then moved to variable size store length. It corresponds to the maximum input length of RNN later.

2.2. ***ISA for the Microprocessor Dataset***

There are two kinds of instructions we are going to work with here. Load/Store instructions and ALU instructions. These are the opcodes below for all the instructions.

Op-code	Instruction
000	Store
001	Retrieve
010	ADD
011	SUB
100	AND
101	OR
110	XOR
111	Buffer

Load/Retrieve Instruction: Figure 2
ALU Instruction: Figure 3

$$Input\ Sequence = \begin{bmatrix} 000 & 01010 & 10110 \\ 000 & 01011 & 00110 \\ 110 & 01011 & 01011 \\ 100 & 01010 & 01010 \\ 001 & 01010 & 00000 \\ 001 & 01011 & 00000 \end{bmatrix} = \begin{bmatrix} 342 \\ 358 \\ 6507 \\ 4426 \\ 1344 \\ 1376 \end{bmatrix}$$

$$Output\ Sequence = \begin{bmatrix} 111 & 00000 & 00000 \\ 111 & 00000 & 00000 \\ 111 & 00000 & 00000 \\ 111 & 00000 & 00000 \\ 000 & 00000 & 10110 \\ 000 & 00000 & 00000 \end{bmatrix} = \begin{bmatrix} 7168 \\ 7168 \\ 7168 \\ 7168 \\ 22 \\ 0 \end{bmatrix}$$

2.2.1. Nomenclature for the Microprocessor dataset

- Data Size: Corresponds to number of bits of data. Worked with 5 bits of data in each instruction here.
- Address Size: Corresponds to number of addresses available in the "Register File". Worked with 5 bits of addressing here.
- To ease the complexity and testing the model's capacity, Every input sequence has 6 instructions, where first two instructions are store and the next two instructions are ALU operations over the addressing above and the last two instructions are retrieving the data stored in the addresses used.

3. Approaches to solve the stack problem.

3.1. ***A Hippocampal RNN Approach using Flip-Flop Neuron***

- **Task:** Stack Operation modelled as Sequential Bits Prediction
- **Data:** $(X_{si}, Y_{si})_{i=1}^{N}$ These pairs are randomly generated. Code for the data set generator is attached in the code section below.
- **Model:** X_{si} and Y_{si} correspond to one training example. Call them x, y respectively and let x_i be the i^{th} row of x. This is passed into Flip Flop RNN(FF) that outputs a representation of every instruction after every pass. The output of the layer layer in the diagram is a 7 dimensional vector $(y_i)_i^7 = 1$ where every element indicates the probability of output being 1 at that position.

$$\begin{aligned} s_i^1 &= ReLU(W concat[e(s_i^2), x_i) + b] \\ h_i^1 &= FF(h_{i-1}^1, concat[s_i^1, h_{i-1}^2]) \\ h_i^2 &= FF(h_{i-1}^2, h_i^1) \\ s_i^2 &= Uh_i^2 + b \\ P(y_i) &= \sigma(s_i^2) \end{aligned}$$

- **Loss:** Cross Entropy Loss.
- **Results:** Training loss wasn't able to converge. Model's capacity was too low for such a complicated problem. Accuracy was 70 %. Since this is a bit prediction problem, this is not a great accuracy.

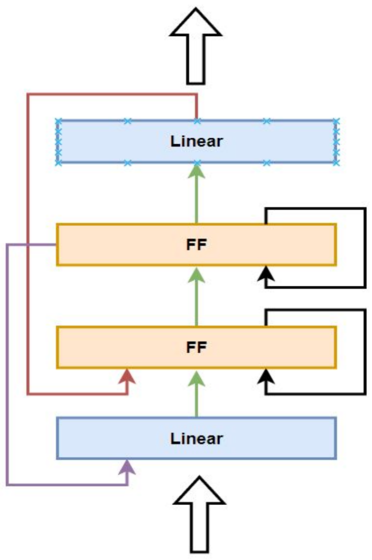

Figure 4: FF Model

3.2. *Encoder-Decoder approach with the FF model*

In order to increase model's capacity, we converted the above model as Encoder and Passed the Encoded states into a similar decoder.

- **Task:** Stack Operation modelled as Sequential Bits Prediction
- **Data:** $(X_{si}, Y_{si})_{i=1}^{N}$ These pairs are randomly generated. Code for the data set generator is attached in the code section below.
- **Model:** Let T be the maximum input length.

  Encoder:

$$\begin{aligned} s_{ei}^1 &= ReLU(W concat[e(s_{ei}^2), x_i] + b) \\ h_{ei}^1 &= FF(h_{e,i-1}^1, concat[s_{ei}^1, h_{e,i-1}^2]) \\ h_{ei}^2 &= FF(h_{e,i-1}^2, h_{ei}^1) \\ s_{ei}^2 &= ReLU(U h_{ei}^2 + b) \end{aligned}$$

  Decoder:

$$\begin{aligned} s_{eT}^2 &= h_{d0} \\ s_{di}^1 &= ReLU(W_d concat[e(h_{d,i-1}^2), (y_{i-1})] + b) \\ h_{di}^1 &= FF(h_{d,i-1}^1, concat[s_{di}^1, h_{d,i-1}^2]) \\ h_{d,i}^2 &= FF(h_{d,i-1}^2, h_{di}^1) \\ s_{di}^2 &= V h_{di}^2 + b \\ P(y_i) &= \sigma(s_{di}^2) \end{aligned}$$

- **Loss:** Cross Entropy Loss.

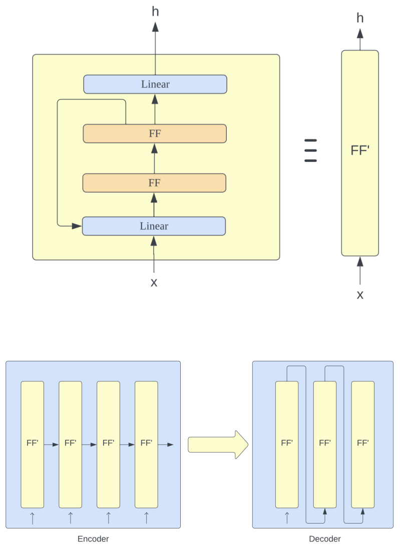

Figure 5: Model Architechture - 3.2

- **Results:** Though the loss was much better than the previous model's, Training loss was still unable to converge and is very unstable now. Results are attached in Figure 6. Unexpectedly, Model's capacity is still too low. We can also attribute low accuracy to this being a bit prediction problem.

3.3. *Encoder-Decoder with Attention*

To modify the task as sequence prediction approach from bit prediction we used binary to decimal encoded input,output sequences as our training data now. We have also incorporated attention to hidden states of encoder. Instead of using the FF' cell we used in the previous method, we wanted to test with the LSTMs. Incase LSTMs fail, we planned to incorporate FF'.

- **Task:** Stack Operation modelled as Sequential Prediction
- **Data:** $(X_{si}, Y_{si})_{i=1}^{N}$ These pairs are randomly generated. Code for the data set generator is attached in the code section below.
- **Model:** Let T be the maximum input length.

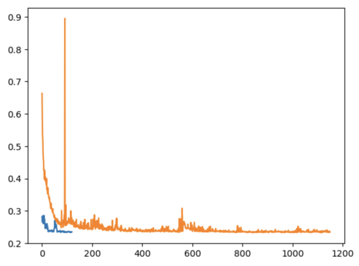

Figure 6: Training Loss vs Iters in Approach 3.2

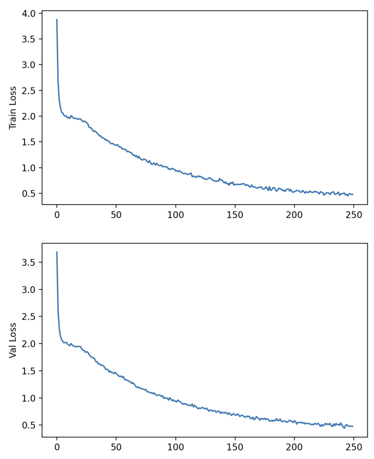

Figure 7: Constant Length Stack

Encoder: Encoder has 2 layers of Bidirectional LSTMs.

$$e_i = e(x_i)$$
$$h_i^1 = LSTM(h_{i-1}^1, e_i)$$
$$h_i^2 = LSTM(h_{i-1}^2, h_i^1)$$

Attention Decoder: Let h_j^2 correspond to h_j

$$e_{ij} = ReLU(Wconcat[h_j, s_{j-1}] + b)$$
$$[\alpha_{ij}] = Softmax([e_{ij}])$$
$$c_j = \Sigma_i \alpha_{ij} h_j$$
$$s_i^1 = LSTM(s_{i-1}^1, concat[c_j, e[\hat{y}_{i-1}]])$$
$$s_i^2 = LSTM(s_{i-1}^2, concat[c_j, s_i^1])$$
$$P(y_i/[y]_{j=1}^{i-1}) = Softmax(Vs_i^2 + b)$$
$$\hat{y}_i = argmaxP(y_i/[y]_{j=1}^{i-1})$$

- **Loss:** NLL Loss.
- **Results:** Model has successfully predicted the sequences in reverse. Results are attached below. Training Loss was converging much faster and accuracy on test data was 85% for store length 8. This leads to the question of its capacity on Microprocessor data. Model was predicting correct outputs up to a sequence length of 20. Variable length stack(where every training example is of different length)'s training was more stochastic but the outputs were fine.

4. Approaches to solve the Microprocessor Problem

Though the stack problem seemed very simple to implement, In the context of binary data, no simple model had the capacity to solve the problem at hand. Attention was successfully able to solve the problem. We

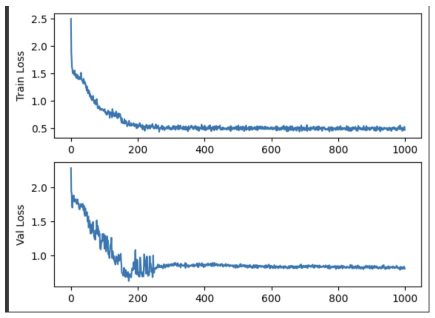

Figure 8: Variable Length Stack

tested the same model with the microprocessor data as discussed above i.e., Data set now is $(X_{mi}, Y_{mi})_{i=1}^N$ and the model,loss is same as Approach 3.3. These pairs are also randomly generated. Code for the data set generator is attached in the code section below.

Turns out this model doesn't have enough capacity to incorporate microprocessor data. Training loss wasn't converging for any set of hyper-parameters tried.

5. Further Work

We have to keep in mind that this is a high-level overview, and the implementation details would depend on the specific microprocessor architecture and the level of detail we want to capture. Additionally, the success of this approach would depend on the complexity of the microprocessor. Stack model was simple enough, but 8-op microprocessor isn't. We plan on using a different encoding scheme(currently we are using binary to decimal encoding scheme) after performing attention experiments on the model. We also plan on increasing it's accuracy by using Recurrent Memory Transformers in our further work.

From all of the experiments above, we have various results that very well agrees with the theory. Whereas we were unable to find answers for the phenomenon such as

- As expected Vanilla FF model failed for large sequences. But Vanilla FF model failing to at least accommodate small sequences is questionable.
- FF' Vanilla Encoder-Decoder failing to converge is also questionable as the number of parameters is much higher in this model than in the approach 3.1.

Hence, Some Interesting Experiments on the previous models include

- We can modify the Vanilla FF model into Encoder - Decoder Seq2Seq Attention and compare its results on Stack data with the LSTM Model
- Looking at the attention map of the microprocessor model and infer why is attention failing.
- Analysing the Levenshtein distance between true and predicted sequences of stack to get a better measure of accuracy for the model we created.

6. Conclusion

In our research, we have achieved notable progress in the implementation of both constant and variable length stack models utilizing recurrent neural networks. While the constant length stack model demonstrated promising results, we encountered challenges when applying the same approach to the microprocessor model. Despite incorporating attention mechanisms, the microprocessor model exhibited difficulty in converging effectively. Nevertheless, we consider this endeavor as a significant step in our pursuit of accurately modeling the intricate architecture of microprocessors.

7. References

- "The Flip-flop neuron – A memory efficient alternative for solving challenging sequence processing and decision making problems." Sweta Kumari, C Vigneswaran, V. Srinivasa Chakravarthy
- "Neural Machine Translation by Jointly Learning to Align and Translate" Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio

Code

Codes for all the above models and datasets can be found here