Pointer Networks Using Fast Weights: Novel Architecture[1]

Khushboo Gupta
Syracuse University
khgupta@syr.edu

Apurva Sharma
Syracuse University
apsharma@syr.edu

Abstract

We propose a novel architecture, a modification of Pointer Networks using Fast Weights as encoder and decoder instead of traditional LSTM. Fast Weights support maintaining intermediate details of the sequence, which might be used to draw associations among elements of input sequence. We implement both the variants of Pointer Network – LSTM based and Fast Weights based on two program learning tasks, boundary detection and priority sorting. For boundary detection task, both variants of model had comparable performance on similar dataset while Fast Weights based Pointer Network performs better on generalized dataset. In priority sorting, the LSTM based Pointer Networks performed far better than the other. We conclude that as boundary detection involved recognizing a pattern, Fast Weights were quite essential for the model to draw associations among patterns in the input sequence for higher accuracy in generalization. However, Fast Weights based Pointer Networks don't work well for sorting the input sequences as they generate lot more duplicates in the output. Due to the possible associations among input data, Fast Weights tend to push the decoder towards the same state.

1. Introduction

Pointer networks were originally proposed to solve the problems where the output dictionary for the problem instances are based on the input sequences, and pointer networks learn to generate the conditional probability of the output sequence corresponding to the input positions. Traditionally, they use LSTM as encoder and decoder in their implementation. However, we decided to explore fast weights or fast associative memory to replace the LSTM based encoder and decoder. LSTMs

[1]Boundary Detection source code: https://github.com/khushbooG9/Pointer-Networks-Using-Fast-Weights

Priority Sorting source code: https://github.com/msapurva/Sorting-arrays-using-Pointer_Networks_w_Fast_Weights

are limited to the short-term memory capacity of O(H) [5], where H is no. of the hidden layers, for saving the history of the current sequence. Fast weights [1] provide an advantage over LSTM where they act as intermediates to store information influencing the current sequence to affect the ongoing process, without using up the capacity of hidden activity vectors.

We plan to test fast-weights based pointer networks on two problems: boundary detection and priority sorting. Boundary detection is the task of getting the starting and ending indices of the irregularity pattern in a given sequence based on the prior constraint. Priority sorting is the task of sorting vectors based on their scaler priorities.

Previously there have been multiple works which target detecting object contours from the images [9] or detect boundaries which define the transitions in musical notes using musical annotations [7] and transitions in songs. However, in program learning universe, it is quite unclear if anyone has attempted to solve this exact problem, where in given array/list, we have to predict or find the indices for a subarray. Manipulation of arrays based on indexing is very common task, specially dealing with development of various algorithms. For the same it's necessary to be able to access the indices based on various conditions. This is one example of such cases, where the neural network should learn to predict the starting and ending indices of the pattern (based on constraint) in the list. This attempt may pave path to future attempts which explore the implementation of neural networks being able to perform array like searches and using those searches to perform complicated problems like maximum sum increasing subsequences, replacing a particular item in the array etc. Pointer networks, predicting the output indices and Fast Weights maintaining the associativity among elements of the sequences might be a combination we need to start tackling above kind of problems.

Priority Sort has been attempted in the past via Neural Turing Machines (NTM), which were introduced by Graves et al.[4]. The work presents preliminary experiments showcasing NTM's ability to tackle sorting sequences of predefined lengths but suffers with the drawback of requiring sequence length to be fixed during training. Pointer networks, by Vinyals et al.[8], introduce a way to overcome the fixed length problem and mention sorting as a task that can potentially be tackled through Pointer Networks. The Priority Sorting task here, starts by evaluating Pointer Networks on the sorting task, something not explored in [8]. It further explores Fast Weights based Pointer

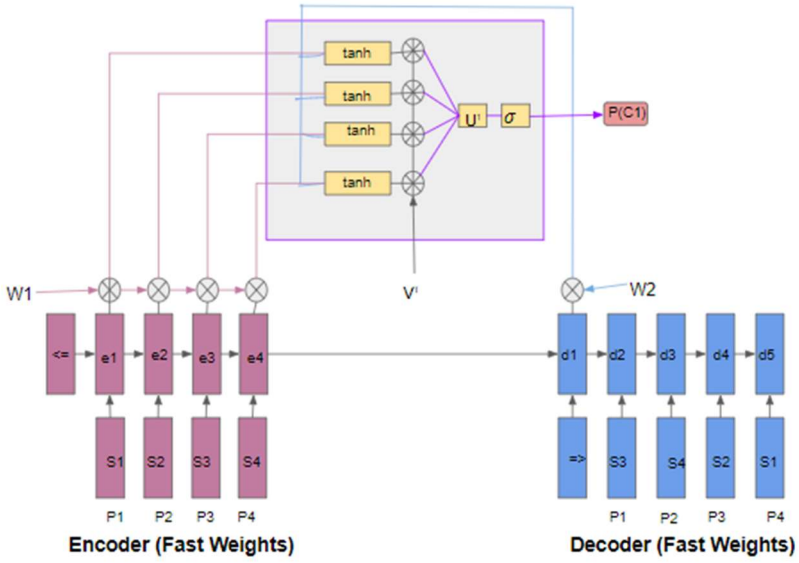
Networks as a way to overcome the drawbacks identified in performance of Pointer Networks on sorting tasks, primarily handling numbers that are close to each other.

2. Methodology

This section outlines in detail about the architecture of the proposed model, underlying update equations and training procedure for both the tasks.

2.1 Architecture

Our proposed novel neural architecture is a modification of original Pointer Networks, where LSTM based encoder and decoder are being replaced with Fast Weights (Fast Associative Memory) based encoder and decoder. Input to the encoder is the zero initial state vector and the first element from the input vector. Once all the elements of the input sequence are converted into encoder, the decoder starts. First input to decoder is the special symbol, which is not supposed to be from any dataset and last encoder state.



To get the attention vector u_j^i and probability distribution over the input sequences from the attention vector, following equations are used -

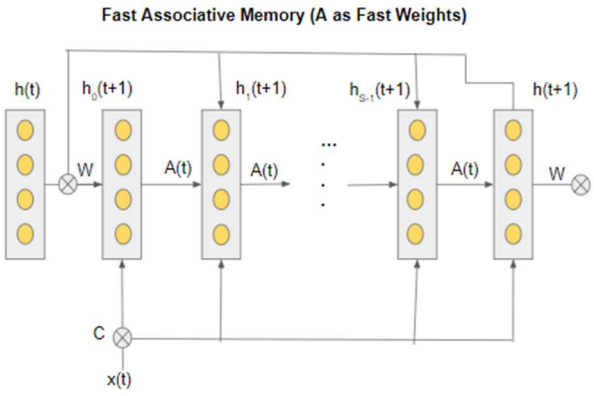
$$u_j^i = v^T \tanh(W_1 e_j + W_2 d_j) \qquad j \in (1, \ldots\ldots, n)$$

$$p(C_i | C_1, \ldots., C_{i-1}, P) = softmax(u^i)$$

where (P, C^P) is the training pair, P is the sequence with n elements and C^P is a sequence of $m(P)$ indices which is target sequence length; each between 1 to n. u^i is the attention vector of length n. v, W_1 and W_2 are the learnable parameters of the trained model. The values e_j and d_j are the encoder and decoder states respectively.

In original Pointer Network paper, the authors have used only vectors as elements of the sequence P, however in our implementation, the elements for the P can be scalars as well as vectors. P can be used as a list of vectors or a vector itself, simulating a list of scalars, which is quite significant to portray them as arrays.

The Fast Weights or Fast Associative Memory Model works as encoder – decoder both. The fast weights layer (A) works as memory of the recent history of hidden state vectors. Every time the input changes, the transition to new hidden state is determined by 3 sources of information – new input using weights C, prior hidden state vector and fast weights A. The fast weights A decay exponentially all the while acting like attention to the recent past. And they have strength of the attention being determined by the scalar product between current hidden vector and previous hidden vector.



The equations involved in the implementation of fast weights are given below –

$$h_0(t+1) = relu([W\, h(t) + C\, x(t)])$$

$h_0(t+1)$ is the starting effective hidden vector or "preliminary" vector during the internal iterative settling phase. W and C are the slow weights matrices, where W maps to hidden vectors and C maps to input. The values $h(t)$ and $x(t)$ are hidden vector and input at time t and $relu$ is the non-

linearity, s is the inner loop for iterative settling phase. We have not added $A(t)$ here as it is zero as well in the start. W and C are learnable parameters for this implementation as well.
For encoder implementation, $h(t)$ for calculation of $h_0(t+1)$ is zero vector (which is always the case with Fast Weights), while for decoder implementation, the $h(t)$ for calculation of $h_0(t+1)$ is the last encoder state. It is being used as decoder starts after encoding process and continues where encoder left off.

$$A(t) = \lambda A(t-1) + \eta h(t)h(t)^T$$

The update rule for fast weight matrix A is given by multiplying the previous fast weight at time $(t-1)$ by decay rate λ, and adding the outer product of hidden vector $h(t)$, multiplied by learning rate of η. Also, at $t = 0$, $A(t-1)$ is zero.

$$h_{s+1}(t+1) = relu(LN[W h(t) + C x(t) + A(t)h_s(t+1)])$$

$h_{s+1}(t+1)$ represents effective hidden state calculated by an inner loop with s iterations, started using the preliminary vector. LN is the Layer Normalization[2], which is applied to protect the hidden state dynamics from exploding and vanishing gradients.

2.1.1 Training Procedure: Boundary Detection

For the boundary detection task, the process involved 3 sets of data – Training set, Test set 1 and Test set 2. Each set has an input-output pair corresponding to the sequence – indices (one hot vectors of sequences' length) pair. Training set was used to train the models. Test set 1 had same kind of problem instances used in the training set. Test set 2 had problem instances which were different from the training set. All 3 sets had 1000 instances as within each set, problems are completely randomized, which helps in preventing the models from overfitting. More information on this in section 2.2.1.
We used 2 models, Pointer Network using LSTM and Pointer Network using Fast Weights, to compare their performance on boundary detection. Training for both models involved Adam optimizer [6] and hidden layer of size 256 in default setup. LSTM based Pointer Networks used

Adam optimizer with learning rate of 0.001 while Fast Weights based Pointer Network used Adam optimizer with 0.005 learning rate. The loss function uses the mean squared error in all the setups. In all the cases, training was done for 500 epochs to check the stability of the model and draw comparisons among 6 different sets of parameters.

2.1.2 Training Procedure: Priority Sort

The overall process has two sub-tasks training an LSTM based Pointer Network, and training Fast Weights based Pointer Network.

For both models, training, development and test data had sequences of varying sizes to enable models to generalize better on arbitrary sequence sizes.

The sequence size range was same for training and development data, while test data had additional larger sequences to test how well the model generalizes.

All setups used cross-entropy loss, normalized to per-output-generated so that all sequence sizes in the training data get equal importance.

For optimization, Adam optimizer has been used with learning rate of 0.001. Other optimizers, like Adagrad were tried but didn't perform as well as Adam. There were other features used across all setups to derive better training performance as batch normalization in Fast Weights. Further, we realized that the representation used for special symbols, like start of a sequence or end of a sequence can have significant impact on model performance. Although the implementation can handle arbitrary long sequences, to optimize training process and reap benefits of batch processing, each mini-batch comprised of same sized sequences only. Mini-batches of different sequence sizes were rotated during training to enable learning on varied sequence sizes.

A favorable representation of special symbols, like start of sequence or end of sequence, does improve learning rate and eventual model performance. We discuss the variations tried and their impact further in later sections.

Lastly, although the total number of instances in training, development and test set were kept same across experiments, the data composition was varied for some to evaluate impact on model performance. Number of epochs vary and depend on how soon the model loss stabilizes.

2.2 Experimental Tasks

For the boundary detection task, Khushboo Gupta was responsible and for list sorting task, Apurva Sharma was responsible.

2.2.1 Boundary Detection

In the boundary detection, for a given input sequence and predefined constraints for the irregularity, the trained network is supposed to be able to return the starting and ending indexes of the irregularity in the given sequence. In the paper for musical structure analysis, Ullrich et al. [5] detect transitions in musical notes by detecting shifts in musical annotations.

However, in this specific case, the boundary detection is being performed on input sequence which is a list of numbers in the range [1,10]. For example, if the input sequence is {1,2,1,3,4,5,6,9,10,7,3,3,1,2} the trained model should return (6,9) following 0-based indexing. In program learning universe, this corresponds to a number of algorithmic problems based on arrays, such as return the indexes of a subsequence in a given sequence and Hill and Valley detection in given array.

For the dataset, we used the low-high-low pattern where low pattern contains integers from 1 to 5 and high pattern contains integers from 6 to 10. To make the data quite varied, we used random generator to get the lengths and contents both for low and high subsequences and then added them to get the final sequence. Low subsequences have lengths from 5 to 10 and high subsequences have lengths 5 to 10 (in test set 1) and lengths 11 to 20 (in test set 2), where test set 1 is the collection of similar sized problem instances as training set while test set 2 contains different sized problem instances to check the generalization ability of the model. Hence the smallest length of the sequence is 5+5+5 = 15 while the longest sequence is 10+20+10=40, which can easily be scaled by the user. We avoided using a seed for the randomizer as it will make all the sequences of same size and content and will hurt the idea of having variable examples.

However, this created a problem to generate the datasets properly compatible with the model for batch training, so we used a frame of maximum length as 60 to add the sequence padded with zeros. While training and testing both, the model ignores the zeros used as padding. For all 3 sets – training, test 1 and test 2, we used 1000 examples of the sequences. To make it clearer, there is one sample of the sequences from the dataset -

```
[ 2  1  5  3  4  2  4  3  5  1  4  5  8  9  8  8  6  8  9  7  7  9  9 10
 10 10  6  9  8  9  6  6  5  5  4  5  4  2  4  4  3  2  2  5  4  3  4  0
  0  0  0  0  0  0  0  0  0  0  0  0]
[ 2  3  1  5  1  4  1  5  3  2  1  5  5  7  7 10  7  7  6  7  9  7  6 10
  9 10 10  8  1  1  3  5  1  4  2  4  3  4  4  4  1  1  5  2  2  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0]
[ 1  5  3  5  4  1  4  4  4  5  2  3  3  8  8  9  8  6  9  9  9  6  9  6
  6 10  6  8  8 10  8  3  4  2  2  4  2  4  3  3  2  4  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0]
[ 2  3  3  2  2  4  2  3  2  1  2  3  3  2  1  4  4 10  9 10  6  9  9 10
  7  8  6  7 10  6  5  3  4  4  1  5  2  4  5  1  4  3  3  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0]
[ 3  2  5  3  4  5  4  1  5  2  3  4  3  2  4  3  2  4  6  7  7 10  9 10
  8  6  7  8  6  8  3  1  1  2  1  5  1  4  1  3  5  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0]
[ 4  2  4  2  1  4  2  3  2  1  2  5  4  1  2  8  7  7 10  7  8 10  7  9
  9  8  4  1  3  1  5  3  3  5  1  4  3  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0]]
```

2.2.2 Priority Sorting

Priority sorting focuses on sorting input sequence vectors with respect to scalar priorities associated with each of the vectors. The task can be thought as comprising two sub-tasks:

a. the ability to sort scalar priorities and

b. the ability to associate vectors associated with each scalar in the sorted output

Since pointer networks directly point to the entire unit (vector + associated priority) learning sub-task b. primarily means learn to ignore the non-priority part of input vector for all calculations. Hence, the author hypothesize that sub-task b. is secondary to sub-task a., focusing on model's ability to learn sorting of input sequences in this exercise. Tackling sub-task b. is suggested as next step.

Sample Input-Output

The sample input to the sorting task is a sequence of floating numbers sampled uniformly from the interval (0,1). The expected final output is the same sequence of elements sorted in ascending order. The below shows four samples of input and output data points.

```
Input: [0.37454012, 0.9507143, 0.7319939, 0.5986585]  > Output: [0.37454012, 0.5986585, 0.7319939, 0.9507143]
Input: [0.15601864, 0.15599452, 0.058083612, 0.8661761]  > Output: [0.058083612, 0.15599452, 0.15601864, 0.8661761]
Input: [0.601115, 0.7080726, 0.020584494, 0.96990985]  > Output: [0.020584494, 0.601115, 0.7080726, 0.96990985]
Input: [0.83244264, 0.21233912, 0.18182497, 0.1834045]  > Output: [0.18182497, 0.1834045, 0.21233912, 0.83244264]
```

While, the above exemplify the input and their expected output, the model uses a more machine suitable format to represent output elements. Each element in output is represented by its position-index in the input sequence. The position index itself is represented using a one-hot vector. For example, the output of the first of the four samples below can be interpreted in the following way: the input has 4 elements to be sorted, hence each output element will be a one-hot vector array of size four. Further, the smallest element in the list is 0.37454012, situated at index 0; hence, the first array-element in output has 1 in index 0 while all other values are 0. The next smallest element is 0.5986585, situated at index 3 in the input array; hence, the second array-element in the output has 1 in index 3 while all other values are 0. Similarly, for third and fourth output array.

```
Input:[0.37454012,0.9507143,0.7319939,0.5986585] > Output:[1. 0. 0. 0.],[0. 0. 0. 1.],[0. 0. 1. 0.],[0. 1. 0. 0.]
Input:[0.15601864,0.15599452,0.058083612,0.8661761] > Output:[0. 0. 1. 0.],[0. 1. 0. 0.],[1. 0. 0. 0.],[0. 0. 0. 1.]
Input:[0.601115,0.7080726,0.020584494,0.96990985] > Output:[0. 0. 1. 0.],[1. 0. 0. 0.],[0. 1. 0. 0.],[0. 0. 0. 1.]
Input:[0.83244264,0.21233912,0.18182497,0.1834045] > Output:[0. 0. 1. 0.],[0. 0. 0. 1.],[0. 1. 0. 0.],[1. 0. 0. 0.]
```

The above format has been used to train all Fast Weights based models.[2]

3. Empirical Results

This section describes the results from both tasks with detailed explanation.

3.1 Boundary Detection

To compare the performance of this model with original Pointer Networks, we ran experiments on Pointer Networks with LSTM and our proposed model, Pointer Networks with Fast Weights and averaged the results from accuracy of each experiment performed 5 times with default parameters. The default parameters for Pointer Networks with LSTM include hidden dimensions of 256 and Adam optimizer with learning rate of 0.001. The default setting for this task and our model is hidden dimensions of 256, S as 1 (where S is the number which determines the number of times inner loop iterations to calculate fast weights between 2 hidden states), Adam optimizer with learning rate of 0.005 and training the model for 500 epochs, with mean squared error as the loss

[2] The input for initial experiments, which were LSTM based pointer networks was slightly different. It used a 2-element vector with first element always zero, except for the starting symbol, and second element having the scaler priority. The sequence starting symbol was vector with 1 in the first element and a do-not-care in the second element. The value of starting symbol was not clear leading to dropping of the starting symbol in Fast Weight based experiments making the input representation a scalar for subsequent experiments.

function. On boundary detection task, for the sequences of maximum length 30 and 60, with high subsequences length from 5 to 10 and 11 to 20 respectively, both models have quite comparative performance. However, when both the models were trained on maximum length of 60 with high subsequences length from 5 to 10 but tested on high subsequences with length 11 to 20(which the model never sees in training), the Pointer Networks model with Fast Weights had better accuracy than the model with LSTM. Table 1 represents the empirically determined accuracy for both models.

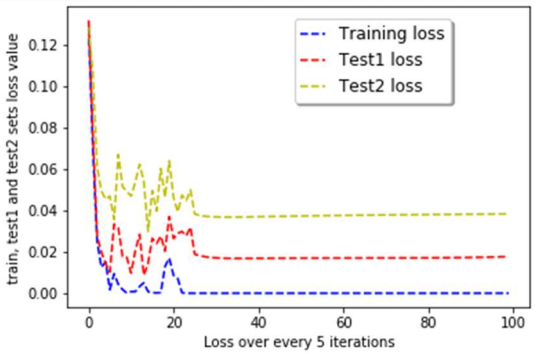
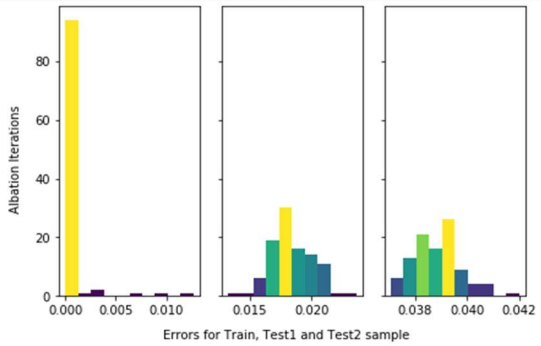
	Pointer Network – LSTM	**Pointer Network – Fast Weights**
Train **Maxlength = 30** **High subsequence range = 5-10** **Test** **Maxlength = 30** **High subsequence range = 5-10**	99.1%	98.9%
Train **Maxlength = 60** **High subsequence range = 11-20** **Test** **Maxlength = 60** **High subsequence range = 11-20**	99.4%	99.4%
Train **Maxlength = 60** **High subsequence range = 5-10** **Test** **Maxlength = 60** **High subsequence range = 11-20**	71.2%	92.6%

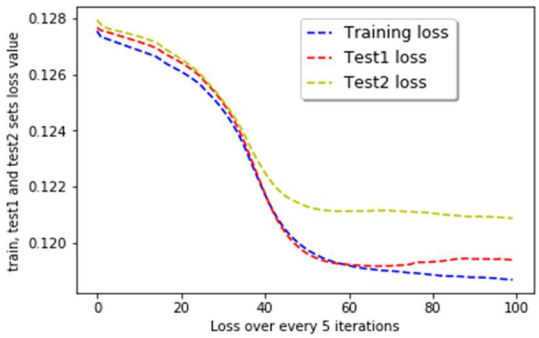
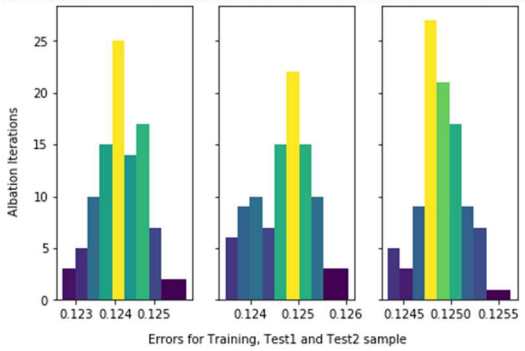
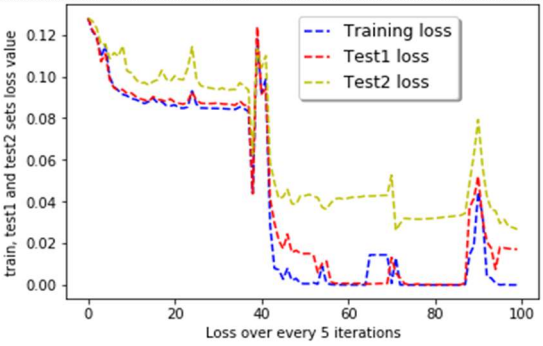
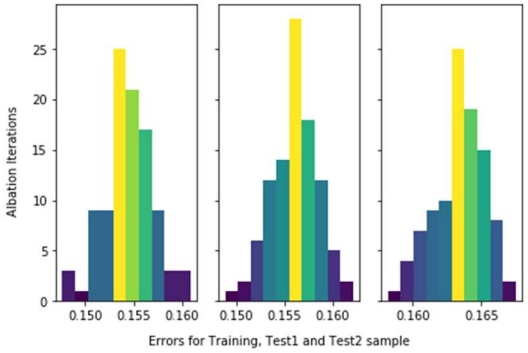
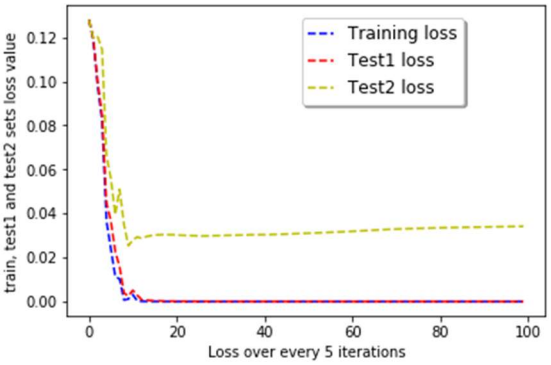
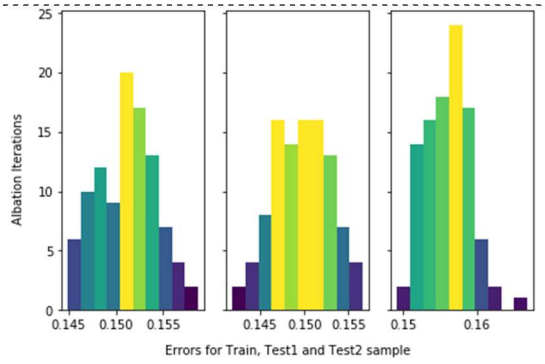
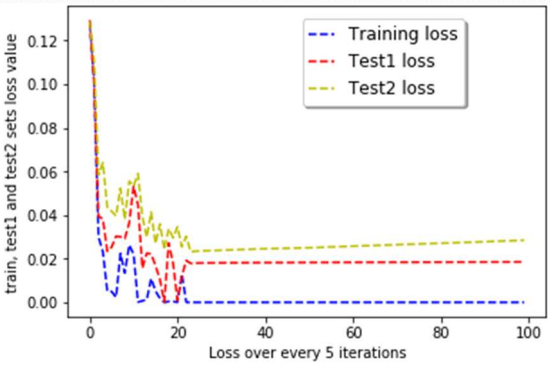
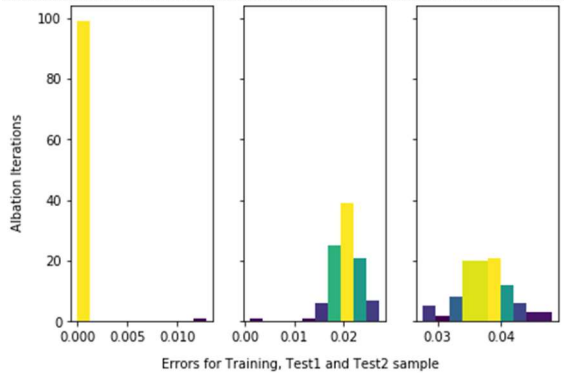
Table 1

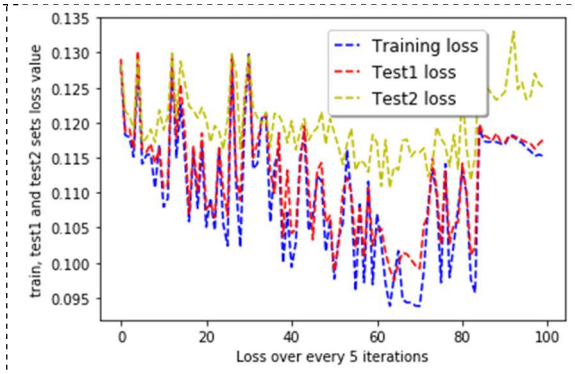
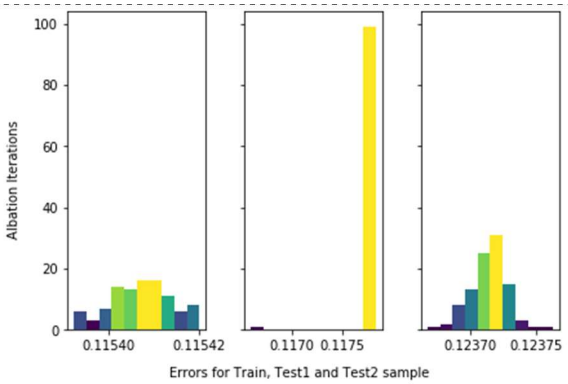
Apart from the above-mentioned default parameters for the model, 5 different hyperparameters to test the model's effectiveness were selected. Those involved hidden dimensions of size 10, 32 and 128; Adagrad optimizer [3] with learning rate of 0.005 and S equal to 3, while keeping other parameters from the default setting intact.

For the ablation study, we chose to add Gaussian noise at the fast weights layer (A) as this layer is quite crucial towards the working of this model. The noise contained uniformly distributed random samples in the range [-10,10]. The ablation was performed 100 times on one sample batch each from the training set, test 1 and test 2 set data, which were originally used to generate the learning curves in each hyperparameter case. The errors, learning curves and their results from the ablation study for each set of parameters on Fast Weights based Pointer Networks are given below.

PARAMETERS	TRAINING LOSS	TEST SET 1 LOSS	TEST SET 2 LOSS
DEFAULT	0.0000006	0.0197821	0.0321987
ADAGRAD	0.1183652	0.1194781	0.1219852
H=10	0.0000097	0.0187811	0.0240927
H=32	0.000071	0.0000098	0.0372106
H=128	0.0000082	0.0220186	0.0278211
S=3	0.1169821	0.1182113	0.126682

Parameters	**Learning Curve**	**Ablation Study**
Default		

Adagrad Optimizer with learning rate of 0.005		
H = 10 (Hidden layer size)		
H = 32 (Hidden layer size)		
H=128 (Hidden layer size)		

S=3 (Fast Weights iterations)		

Apart from the cases where we had default parameters, hidden dimensions as 10, 32 and 128, the model did not learn. However, with hidden dimensions as 10, the performance of model on both test sets was not optimal. It is crucial to know that in the start of the training procedure, the initial errors ranged from 0.12 to 0.14 in all performed experiments. Also, from ablation study, it seems that only the cases with default parameters and hidden dimensions as 128, the trained model continued to perform well, even with the added noise to the fast weights layer, on all three datasets – training, test1 and test2 with errors ranging from 0.000 to 0.05 which was comparable to final errors for each case.

3.2 Priority Sorting

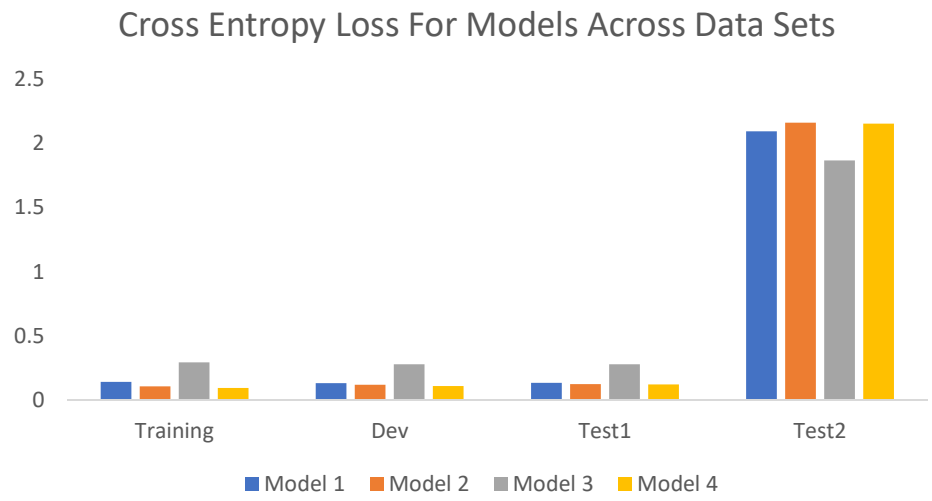
The experiments started with base configuration of data and models and then sequentially modified different aspects to achieve best possible performance. Further, on the sorting task we primarily report Cross-Entropy Loss. Metrics like accuracy are not fair for measuring performance on sorting as they don't give credit for partial correct outputs which is very much required in measuring performance here. We do share sample outputs from different loss levels for examination and analysis of the reader.

Base Data

	Min Seq Size	Max Seq Size	Batch Size	Num. of batches	Total Data
Training	2	5	32	200	25600
Dev	2	5	64	10	2560
Test 1	2	5	64	10	2560
Test 2	6	10	64	10	3200

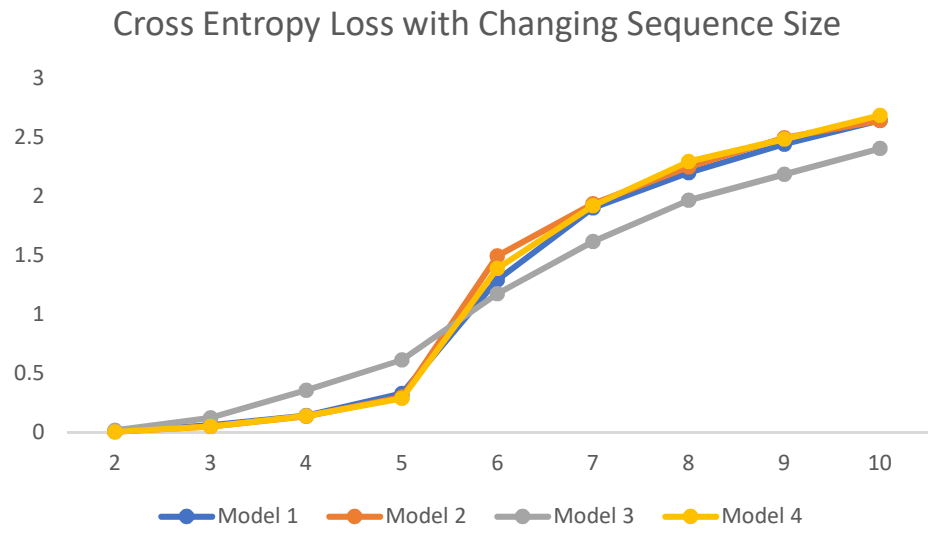
3.2.1 LSTM based Pointer Network

PERFORMANCE SUMMARY (LSTM Based Model)



#	Hidden Dimension	Training	Dev	Test1	Test2	Epochs	Start Symbol
1	**300**	**0.14061**	**0.12957**	**0.13319**	**2.09349**	**8**	**[1, 0]**
2	200	0.10463	0.11927	0.12203	2.16151	19	[1, 0]
3	100	0.29263	0.27704	0.27679	1.86829	8	[1, 0]
4	*300*	*0.09421*	*0.10748*	*0.11946*	*2.15333*	*11*	*[1, 5]*

DETAILED TEST PERFORMANCE (LSTM Based Model)



#	2	3	4	5	6	7	8	9	10
1	**0.00516**	**0.05941**	**0.14044**	**0.32777**	**1.28904**	**1.89913**	**2.19810**	**2.43939**	**2.64177**
2	0.00600	0.04906	0.13600	0.29705	1.49328	1.93468	2.24575	2.49435	2.63947
3	0.01722	0.12318	0.35452	0.61223	1.17285	1.61531	1.96518	2.18535	2.40273
4	*0.00382*	*0.04809*	*0.13969*	*0.28625*	*1.38718*	*1.91792*	*2.29406*	*2.48376*	*2.68375*

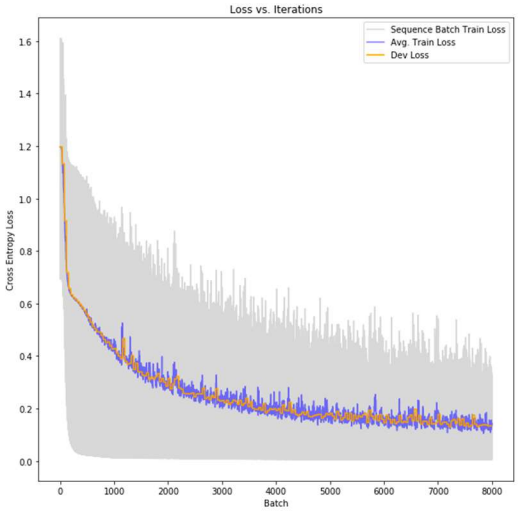
TRAINING DETAILS (LSTM Based Model)

Training Process captures 3 different losses:

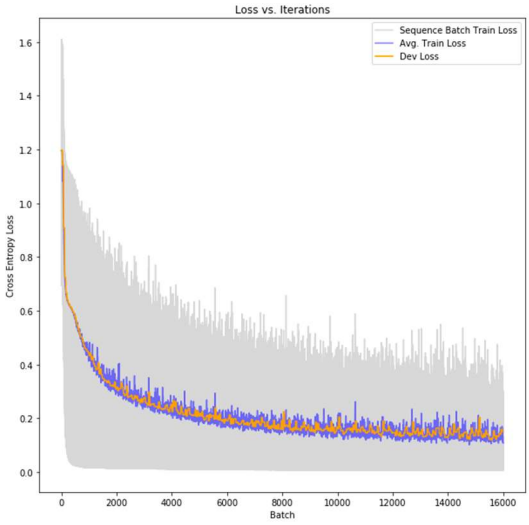
i. Batch Train Loss: Loss on each batch. It fluctuates as each new batch has different size

ii. Avg.Train Loss: Loss on a batch set. A batch set comprises one mini-batch of each size

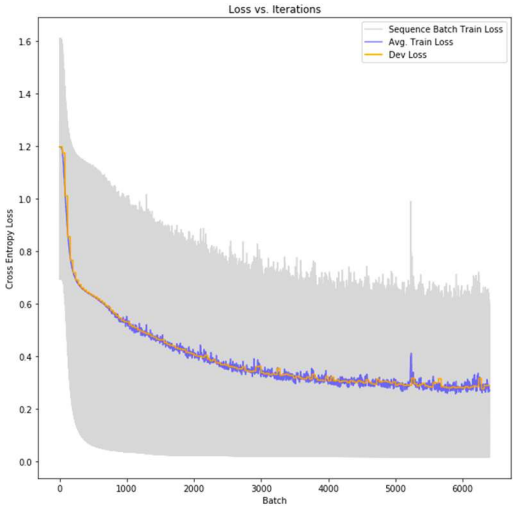
iii. Dev Loss: Average Loss on the entire Development Set



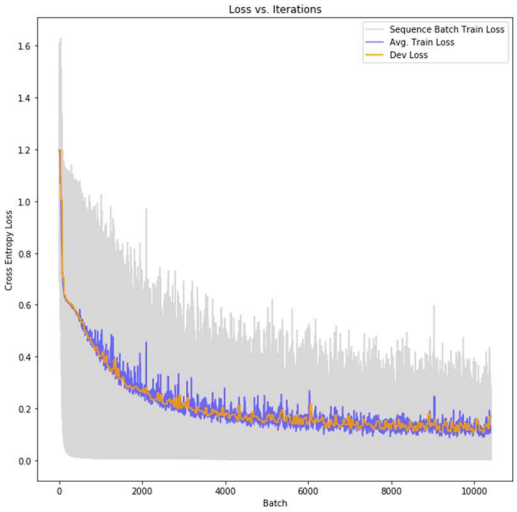
Model 1



Model 2



Model 3



Model 4

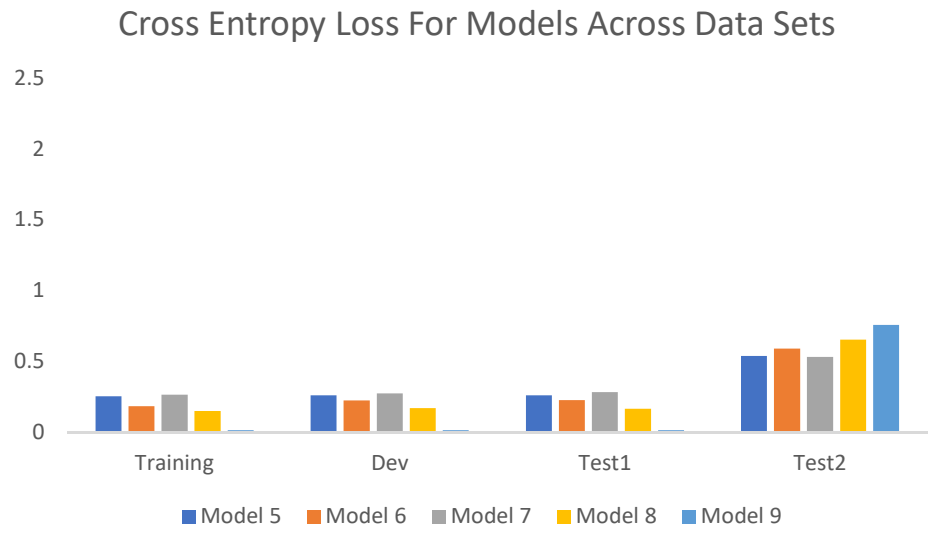
PERFORMANCE ANALYSIS (LSTM Based Model)

- LSTM based networks do a good job of learning sequence sorting, particularly on sequence sizes which they have seen in training data
- Based on experiments the performance starts to fall as the sequence size grows beyond the sequence sizes seen in training data.

- The key performance issue surfaces in form of duplicate entries being output by the model. Basically, the model tends to confuse two close numbers dropping one of them all together. This might be due to the artificial limit imposed of numbers being between 0 and 1. More experiments with wider numeric range are required to check this
- The performance generally seems to go up with increasing hidden dimension size. Hidden dimension of 100 being too low to get any good performance, even on training sizes of 4 and 5, while 300 being sufficient to get excellent performance on sequence size 2-5
- The sharp increase in error for sizes 6 and beyond, seem to hint towards model not learning a meaningful internal representation for sorting sequences, rather a more learned behavior fit to sequence sizes 2-5. Whether further, tuning is required to obtain better performance or better network architecture is needed requires more experimentation
- Lastly, model configurations for model 1-3 used sequence start symbol as vector [1,0] and expected the model to place the special symbol towards the end in the output. The vector, with a second element value of 0 (minimum possible for other vectors in the sequence) coming last creates an anomaly to the sorting behavior might require the Network to learn additional exceptional behavior. To avoid spending network's learning power to handle exception and make the job easier we changed the representation of special symbol to [1,5] with 5 fitting in better as the last value in an ascending sequence. Model 4 is based on this new representation and it does perform better on training and test 1 data showing that initialization values do make a big difference.

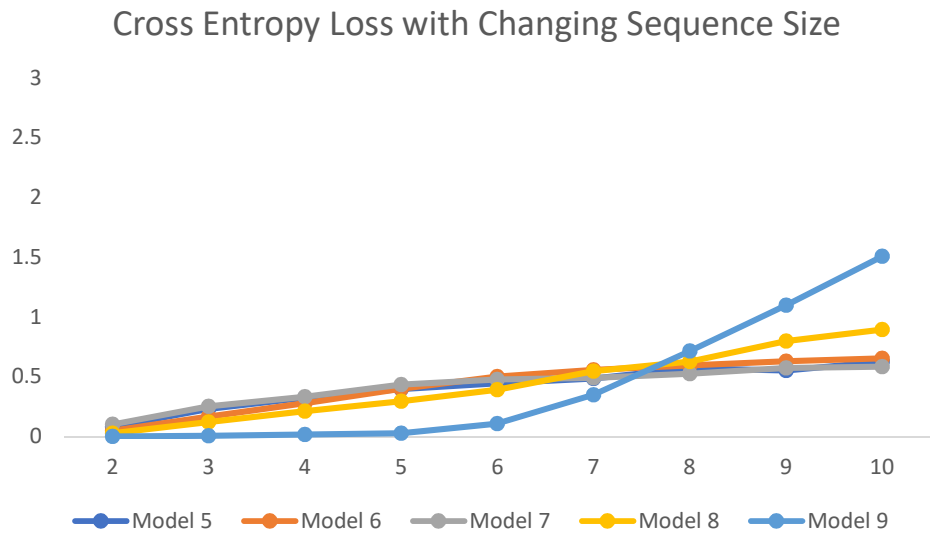
3.2.2 Fast Weights based Pointer Network

PERFORMANCE SUMMARY (Fast Weights Based Model)



#	Hidden Dimension	Training	Dev	Test1	Test2	Epochs	S	Learning Rate	Decay Rate
5	300	0.25254	0.25955	0.26047	0.53783	4	1	0.5	0.9
6	200	0.18239	0.22450	0.22502	0.58851	6	1	0.5	0.9
7	100	0.26472	0.27440	0.28157	0.53156	2	1	0.5	0.9
8	200	0.14928	0.17017	0.16544	0.65309	4	1	0.2	0.6
9	200	0.01401	0.01429	0.01440	0.75721	3	0	-	-

DETAILED TEST PERFORMANCE (Fast Weights Based Model)



#	2	3	4	5	6	7	8	9	10
5	0.08191	0.23344	0.32687	0.39964	0.44741	0.48708	0.57288	0.55420	0.62757
6	0.04985	0.16892	0.27937	0.40194	0.50290	0.55861	0.59642	0.63037	0.65426
7	0.10318	0.25255	0.33473	0.43580	0.47878	0.49090	0.52710	0.57461	0.58642
8	0.02834	0.12375	0.21258	0.29710	0.39290	0.54726	0.62926	0.79978	0.89625
9	0.00185	0.00880	0.01793	0.02900	0.10951	0.35073	0.71726	1.10017	1.50838

TRAINING DETAILS (Fast Weights Based Model)

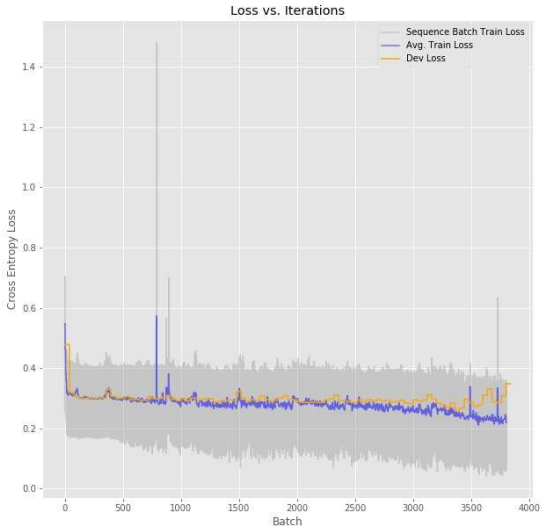
Training Process captures 3 different losses:

i. Train Loss: Loss on each batch. It fluctuates as each new batch has different size

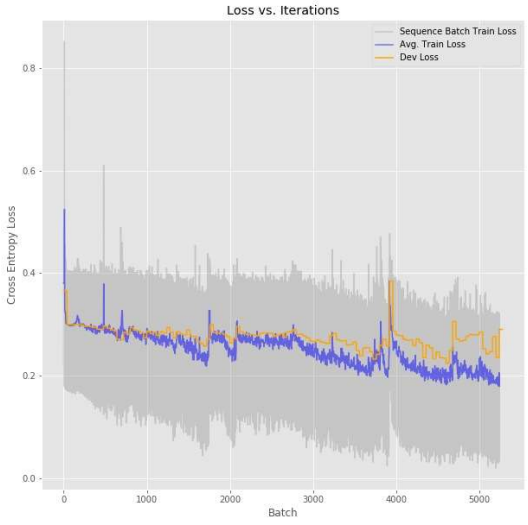
ii. Avg.Train Loss: Loss on a batch set. A batch set comprises one mini-batch of each size

iii. Dev Loss: Average Loss on the entire Development Set

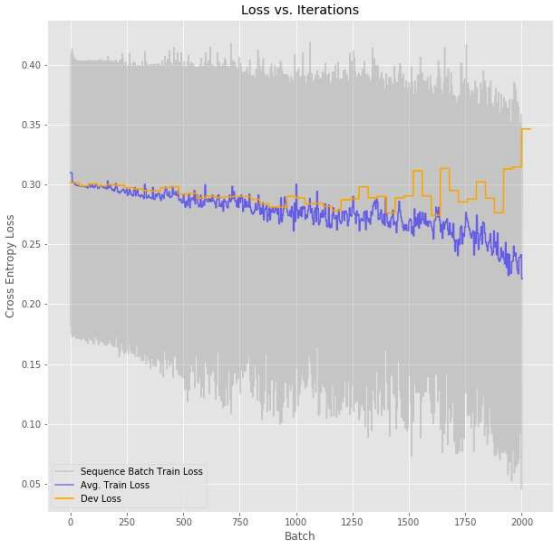
In cases where Dev Loss differs significantly from Avg. train loss, the place where they are aligned is used, avoiding overfitting.



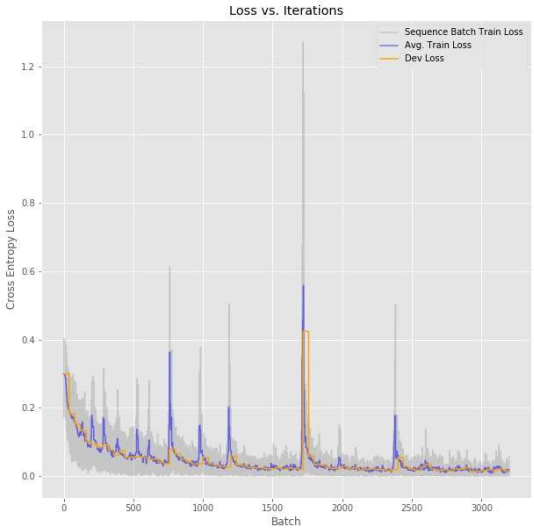
Model 5



Model 6



Model 7

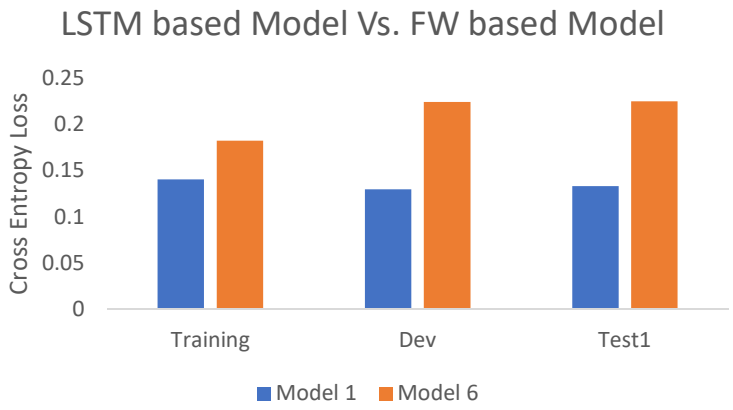


Model 9

PERFORMANCE ANALYSIS (Fast Weights Based Model)

- Comparing models 1, 2 and 3 with models 5, 6 and 7 we can make out that performance of Fast Weights based networks is significantly lower than corresponding LSTM based

pointer networks. Overall impact of Fast Weights seems to deteriorate performance of Pointer Networks on sorting tasks. See figure below



- The above is additionally supported by the fact that reducing the values of Fast Weight's learning rate and decay rate, which will lead to decreasing memory power of Fast Weights on network functioning, tends to improve performance (Model 8). Further, model 9, which completely shuts down the Fast Weights component outperforms all the models up till 8 providing more evidence of the hypothesis that Fast Weights are negatively impacting performance of the network on sorting tasks
- The error in Fast Weights output appear as heavy repetitions in the output with sequences of size 4 or 5 often having only two unique elements
- Model 9 shows better performance than most LSTM models, especially on known sequence sizes, which is surprising as Model 9 is essentially an RNN. Our hypothesis is that it is caused by the bulky vector representation that has been used in LSTM implementation of the code and LSTM networks might benefit with the light, special symbol free, representation that has been used for Fast Weights based implementation

3.2.3 Best Model Configuration's Ablations

- We took the best performing model, model 9, and tried further variations of data and configuration on it to
    i. understand its behavior better and
    ii. see if there is further possibility of improvement

PERFORMANCE SUMMARY (Ablations)

#	Input Dimension	Train Sequence	Batch Size x No. of Batch	Training	Dev	Test1	Test2	Epochs
9	*200*	***2,5***	*32 x 200*	*0.01401*	*0.01429*	*0.01440*	*0.75721*	*3*
10	200	**4,5**	32 x 400	0.04706	0.03409	0.03694	2.07605	4
11	200	**5,5**	32 x 800	0.05606	0.04095	0.04795	7.89380	8
12	**1**	2,5	32 x 200	0.02221	0.01252	0.01311	1.15826	9
13	200	2,5	**8 x 800**	0.01875	0.01771	0.01958	0.88560	1

DETAILED TEST PERFORMANCE (Ablations)

#	2	3	4	5	6	7	8	9	10
10	-	-	0.03283	0.04104	0.36702	1.20910	2.07821	2.97966	3.74626
11	-	-	-	0.04795	3.11076	6.91210	8.80799	9.82490	10.81326
12	0.00133	0.00752	0.01316	0.03042	0.30374	0.81557	1.23127	1.57026	1.87048
13	0.00465	0.01133	0.02389	0.03843	0.15879	0.50689	0.88573	1.25289	1.62373

PERFORMANCE ANALYSIS (Ablations)

- The variations tried were on 3 key parameters:
    i. Training Sequence Length: Reducing the variation in training sequence heavily impacts performance on Test 2 dataset, although the performance on Test 1 doesn't get any significant impact. This clearly shows that to improve generalization on arbitrary sequence lengths training on wider range of sequence is better. restricted our training to 2,5 to accommodate for the restricted GPU resources available to train on but we recommend training on more sequences than 2,5
    ii. The input, before being passed onto the Fast Weight/LSTM/RNN hidden state part, passes through a transformation at the input layer. We tried to reduce the input transformation to see if it is adding any value for scalers. Reducing the size led to reduction in generalization performance. This needs to be investigated further. As next steps we want to analyze the outputs at each step to understand how the model

is behaving internally and if some generalization can be drawn about the embeddings learnt by the system for scalar numbers.

iii. Lastly we tried to see if changing batch size leads to change in overall performance of the model. Based on results of model 13, no significant impact was present on batch size on performance of sorting model

3.2.3 Sample Outputs from Select Models

Model	Input Sequence	Predicted Sequence	Comment
Fast Weights Based Model	0.42932528 0.5543136 0.91265076 0.6928655	0.42932528 0.42932528 0.42932528 0.42932528	Duplicates due to Fast Weights
	0.5211839 0.7436473 0.45335925	0.45335925 0.45335925 0.45335925	Duplicates due to Fast Weights
	0.49157616 0.9199425	0.49157616 0.9199425	Correct only on small outputs
Fast Weights with S=0 Based Model (RNN)	0.5059709 0.3846874 0.02724245	0.02724245 0.3846874 0.5059709	Correct output
	0.8830871 0.37230211 0.2038088 0.7127647 0.8428299	0.2038088 0.37230211 0.7127647 0.8428299 0.8830871	Correct output
	0.9851592 0.95509803 0.28416237 0.56754434 0.18474454 0.7803585	0.18474454 0.28416237 0.56754434 0.7803585 0.95509803 0.95509803	Partially Correct
LSTM Model	-- Special-- 0.09012347 0.58211790 0.18161231 0.01848282	0.01848282 0.09012347 0.18161231 0.58211790 -- Special --	Correct output
	-- Special -- 0.73593640 0.06384878 0.28107760	0.06384878 0.28107760 0.73593640 -- Special --	Correct output
	-- Special -- 0.18411608 0.41264460 0.64268630 0.37089378 0.67645850 0.83634967 0.46989697	0.18411608 0.6764585 0.6426863 0.4126446 0.4126446 -- Special – -- Special – -- Special --	Duplicate values and end symbol

4. Discussion

In this work, we have implemented LSTM and Fast Weights based Pointer Networks on two program learning tasks – boundary detection and priority sorting.

For the boundary detection, both models have comparable performance (with 95% plus accuracy) on same lengths of training and testing high subsequences in dataset. However, Fast Weights based Pointer Networks are able to generalize way better than the LSTM based Pointer Networks on different length subsequences (92% average accuracy vs 72% average accuracy from 5 different experiments on each).

The most challenging part for boundary detection task was forming the datasets such that high pattern in the low-high-low sequence are of variable lengths to produce variable sets of boundary indices output for both training and testing. Although this resulted in having to make the model work based on a maximum length, which performs on sequences padded with zeros. But the model works quite well on various datasets with varying lengths of low and high subsequences within the maximum length.

As it's unclear if this specific kind of boundary detection was previously attempted it's hard to compare it other similar attempts in detecting musical annotation boundaries in songs or object contour detections in pictures. But both Pointer Network variants very well on this task, due to their ability to predict the indices. As hypothesized, not only Fast Weights work well as encoders and decoders with Pointer Networks, they generalized better compared to LSTM based Pointer Networks. This might be due to the ability of Fast Weights to save temporary knowledge to be stored within the layer and make the hidden vectors more relevant to the previous activity, hence being able to remember the pattern of high's and apply it later to the lengthier pattern in the test set 2.

For boundary detection, the ablation study suggests that the values of parameters including hidden layer (128 to 256), S=1 and Adam optimizer with 0.005 learning rate is the best setup. In these setups only, the trained model was able to retain its ability to predict the proper indices pair. In other cases, even if the model learned (as in cases with hidden dimensions as 10 and 32), ablation study proved that the trained model is susceptible to failure, generating errors greater than that of before training.

For future work, we suggest tweaking the model such that it is not limited by the length of inputs, specifically in the case of boundary detection so as to accommodate every kind of variation in lengths, not just of subsequences. It would also be interesting to try implementing basic array manipulation tasks such as searching, extracting subarrays, updating elements in arrays as they all require the neural network to be able to predict and work with indices.

For sorting task, Pointer Networks without Fast Weights were able to achieve a low cross entropy test loss of 0.014 and sorted most known sized sequence correctly; performance on lower unknown size sequences was acceptable too. With Fast Weights the models were not able to achieve good accuracy on training or test data with most outputs having lot of duplicates suggesting Fast Weights might not be best for all types of problems.

The key problem one encounters with Fast Weight based solution for priority sort is repetitions of numbers suggesting that while Pointer Networks do a good job of sorting sequences on known sequence sizes introduction of Fast Weights seems to add noise due to its associative power, leading to lot of duplicate entries. This goes down as the network's Fast Weights are tuned down in power, suggesting Fast Weights might not help all types of tasks.

Further, it has been observed that initializations and representations of special symbols like start or end of sequence symbols and initial input of decoder seem to have significant impact on model performance. Models perform better with representations that align with the overall objective of the system, like a high valued integer as the last output symbol for ascending sort task.

Going forward we want to explore many more options to evaluate impact of model performance. Especially, the impact of using unseen sequence sizes on dev data to improve model performance, using larger sequence sizes to train models and impact of using loss function that favors longer sequences as against shorter sequences. Further, we want to evaluate performance of LSTM models on a lighter 1-valued representation of scalar priorities. Lastly, this exercise has focused primarily on sorting priorities of vectors, we would like to evaluate the best performing models on vector + scalar priority combination as discussed in earlier section.

5. Bibliography

[1] Jimmy Ba, Geoffrey Hinton, Volodymyr Mnih, Joel Z. Leibo and Catalin Ionescu. Using Fast Weights to Attend to the Recent Past. *arXiv:1610.06258v3*, 2016. https://arxiv.org/pdf/1610.06258.pdf.

[2] Jimmy Lei Ba, Jamie Ryan Kiros and Geoffrey E. Hinton. Layer Normalization. *arXiv preprint arXiv:1607.06450,* 2016. https://arxiv.org/pdf/1607.06450.pdf.

[3] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of Machine Learning Research*: *2121-2159,* 2011. http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf .

[4] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014. https://arxiv.org/pdf/1410.5401.pdf.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long-Short Term Memory. In *Neural Computations 9(8),* pages 1735 -1780, 1997. https://www.bioinf.jku.at/publications/older/2604.pdf.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980,* 2014. https://arxiv.org/pdf/1412.6980.pdf .

[7] Karen Ullrich, Jan Schlüter, and Thomas Grill. Boundary Detection in Music Structure Analysis using Convolutional Neural Networks. In *The International Society of Music Information Retrieval*, pages 417-422, 2014. https://grrrr.org/pub/ullrich_schlueter_grill-2014-ismir.pdf.

[8] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015. https://arxiv.org/pdf/1506.03134.pdf.

[9] Jimei Yang, Brian Price, Scott Cohen, Honglak Lee, and Ming-Hsuan Yang. Object contour detection with a fully convolutional encoder-decoder network. *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* pages 193-202*,* 2016. https://eng.ucmerced.edu/people/jyang44/papers/cvpr16_contour_final.pdf .