Neural Turing Machine with Diagonal Structured State Space Model (NTM-S4D)

Kerim Erekmen, Hauke Bünning

12.06.2024

Abstract

Sequence models are crucial for representing real-world data, which often exists sequentially. State space models such as the Structured State Space Model (S4) are adept at capturing long-range dependencies but exhibit deficiencies in sample efficiency and encounter difficulties with a uniform learning rate across all parameters. Moreover, Neural Turing Machines (NTMs) frequently experience stability issues when processing longer sequences. To address these challenges, we introduce NTM-S4D (Neural Turing Machine with Diagonal Structured State Space Model), which augments S4 with external memory and attentional mechanisms. Our findings demonstrate that NTM-S4D significantly enhances sample efficiency, captures long-range dependencies more effectively, and improves stability for longer sequences, surpassing the performance of the NTM model.

1 Introduction

When addressing sequence tasks, particularly those encountered in real-world applications, the length of sequences and the presence of long-range dependencies (LRDs) pose significant challenges [1]. Unlike computers, humans utilize their memory to recall past experiences and make informed decisions. This ability to filter and apply relevant memories is a complex mechanism that is difficult for computers to replicate. In this paper, we examine two prominent approaches to learning LRDs in sequences: **Neural Turing Machines** (NTM) [3] and **Structured State Space Sequence Models** (S4) [5].

Neural Turing Machines utilize a dedicated memory bank to store and retrieve experiences, providing a flexible, powerful, although computational expensive way to handle LRDs. Conversely, S4 models keep track of inputs by compressing them into a state, offering a more compact representation, that, if necessary, can approximately reconstruct the input history. Our goal is to combine these successful approaches into a new model, NTMwS4, which integrates the memory bank of NTM with the state representation of S4.

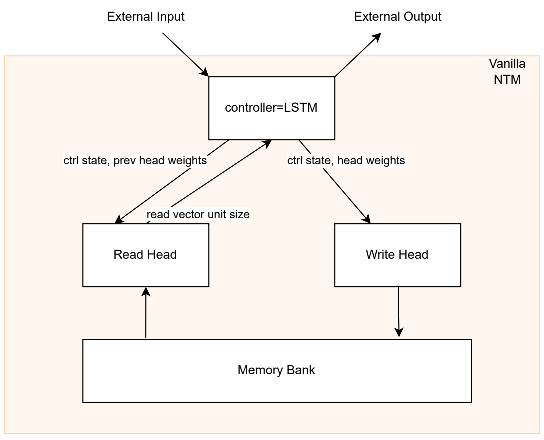

Figure 1: The basic architecture of a Neural Turing Machine [3], Including the controller, read/write heads, a memory bank and arrows to signal paths of data.

Specifically, NTM-S4D replaces the LSTM [7] controller in NTM with the S4 model, aiming to enhance the understanding of LRDs. This novel hybrid approach seeks to leverage the strengths of both methods, providing a more robust solution for sequence tasks.

2 Background

2.1 Neural Turing Machines

The Neural Turing Machine (NTM) [3] architecture extends conventional neural networks with an external memory matrix that allows them to perform tasks requiring a complex memory and a time-sensitive context, while offering the advantage of being end-to-end differentiable. The architecture includes a controller C for the neural network and read/write heads that interact with this memory matrix. During training, the read/write heads learn to focus on the memory.

Controller. The controller is usually an RNN that processes inputs and generates outputs along with read/write instructions. Given input x_t at time step t, and the previous hidden state h_{t-1}, the controller produces the new hidden state h_t and intructions k_t, β_t, g_t, s_t, and γ_t for the heads. As well as the new output according to state h_t

Memory. The memory is a matrix M_t with dimensions N x M, where N is the number of memory slots and M is the size of each slot. Its functionality is similar to what RAM is to a computer. As shown in 1.

Read and Write Heads. Both read and write Heads generate parameters to interact with the memory. These parameters include keys k_t, strengths β_t, the weight vector w_t^r and others controlling how to focus on memory slots. More than one each can be used, similarly to Turing machines. The **Read Head** takes the current input and instructions from the controller and, based on them, generates an attention vector w_t^r. Therefore yielding the reading procedure, $r_t = \sum_i w_t^r(i) M_t(i)$.

The write Head is seperated into two distinct processing steps, the *erase* and *add* steps.

Erase: The erase vector e_t determines how much of each memory cell should be erased:

$$\tilde{M}_t = M_{t-1} \circ \left[E - w_t^{\mathrm{w}\top} e_t\right] \tag{1}$$

Add: The add vector a_t contains new information to be written:

$$M_t = \tilde{M}_t + w_t^{\mathrm{w}\top} a_t \tag{2}$$

Combining both steps, the update rule for the memory matrix is:

$$M_t = M_{t-1} \circ \left[E - w_t^{\mathrm{w}\top} e_t\right] + w_t^{\mathrm{w}\top} a_t \tag{3}$$

2.1.1 Memory Addressing.

To retain the desired differentiability, Neural Turing Machines (NTM) address the discrete nature of memory by employing a categorical distribution over memory locations. Two addressing mechanisms are utilized to achieve this.

Content-based Addressing. Content-based Addressing means that locations in the memory are chosen, that are, in content, closest to the key vector k_t. The measure being cosine similarity. The resulting similarity vector is multiplied with β, the key strength. Afterwards a softmax function is applied to yield a distribution, resulting in the content-based attention vector w_t^C.

Location-based Addressing. Location-based Addressing is generally focussed on regions of memory locations around a current location and is split into several parts 2:

Gated Interpolation. Gated Interpolation controls the impact of the weights of the previous time step on the current weights. Therefore the controller outputs a parameter g that acts as following:

$$w_t^g = g_t w_t^C + (1 - g_t) w_{t-1} \tag{4}$$

Convolutional Shift. The Convolutional Shift mechanism utilizes a distinct component of the controller output, specifically the shifting kernel s_t. A circular, one-dimensional convolution is applied to s_t and w_t^g. As described by:

$$\tilde{w}_t(i) = \sum_{j=0}^{N-1} w_t^g(j) s_t(i - j) \tag{5}$$

Sharpening. Sharpening describes the final addressing step and serves to sharpen the weight vector by applying the scalar parameter γ to the weight vector:

$$\boldsymbol{w}_t(i) = \frac{\tilde{\boldsymbol{w}}_t(i)^{\gamma_t}}{\sum_j \tilde{\boldsymbol{w}}_t(j)^{\gamma_t}} \tag{6}$$

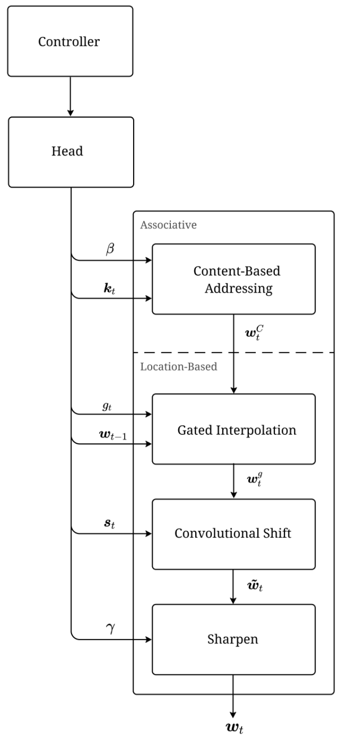

Figure 2: The Memory Addressing pipeline in the head of the NTM (`https://www.niklasschmidinger.com/posts/2019-12-25-neural-turing-machines/`)

2.2 State Space Models

State Space Models (SSMs)[9] are defined as parameterized mappings from input signals $u(t)$ to output signals $y(t)$. These SSMs are linear, time-invariant systems that can be represented as a convolution operation. SSMs are mathematical models used to describe the evolution of a systems state over time. Figure 3 represents the architecture of a continuous, time-invariant SSM, where $\boldsymbol{A}$ is the state matrix of size NxN, $\boldsymbol{B}$ is the control matrix of size Nx1, $\boldsymbol{C}$ is the output matrix of size 1xN and $\boldsymbol{D}$ is the command matrix. These four matrices account for the learnable parameters. Therefore 3 can be reduced to the following equation:

$$\begin{aligned} K(t) &= \boldsymbol{C}e^{t\boldsymbol{A}}\boldsymbol{B} \\ y(t) &= (K * u)(t) \end{aligned} \tag{7}$$

or expressed as a continuous convolution:

$$y(t) = (K * u)(t) = \int_0^\infty \boldsymbol{C}e^{\tau\boldsymbol{A}}\boldsymbol{B}u(t-\tau)d\tau \tag{8}$$

To be able to use the convolutional structure on a computer it is necessary to discretize and therefore approxmiate the integral from 8. Starting with the parameters, one of the methods used in [4] is the **Bilinear method**, which is

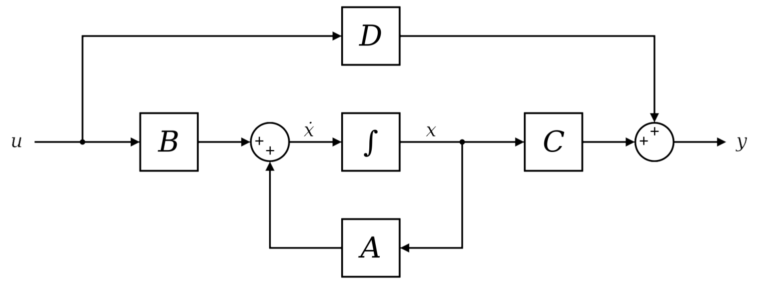

Figure 3: State Space Model

defined as 9 [1]:

$$\begin{aligned}\overline{\boldsymbol{A}} &= (\boldsymbol{I} - \Delta/2\boldsymbol{A})^{-1}(\boldsymbol{I} + \Delta/2\boldsymbol{A}) \\ \overline{\boldsymbol{B}} &= (\boldsymbol{I} - \Delta/2\boldsymbol{A})^{-1} \cdot \Delta\boldsymbol{B}\end{aligned} \tag{9}$$

From this the output of the discrete-time SSM comes to 10:

$$y = u * \overline{K} \qquad \text{Where} \quad \overline{K} = (\boldsymbol{C}\overline{\boldsymbol{B}}, \boldsymbol{C}\overline{\boldsymbol{A}\boldsymbol{B}}, ..., \boldsymbol{C}\overline{\boldsymbol{A}}^{L-1}\overline{\boldsymbol{B}}) \tag{10}$$

Allowing for convolution of the kernel with an entire input sequence, which increases training speed. However during inference a RNN version can be used instead of the kernel to be able to take advantage of the hidden states and computational efficiency, since we need only $O(1)$ operation to compute the next state. The RNN architecture is very similar to 3, using the already trained matrices $\overline{A}, \overline{B}, \overline{C}$. Resulting in 14

$$\begin{aligned}x'_t &= \overline{\boldsymbol{A}}x_{t-1} + \overline{\boldsymbol{B}}u_t \\ y_t &= \overline{\boldsymbol{C}}x_t + \overline{\boldsymbol{D}}u_t\end{aligned} \tag{11}$$

2.3 S4 and S4D

If we use a primitive method to initialize $\boldsymbol{A}$, such as random initialization, this can lead to computational and performance issues [5, 4]. The reparameterization in the S4 and S4D model is to transform the state space model matrices into a form that makes the computations more efficient and stable. S4 and S4D are therefore variants of the SSM model where the choice of discretization mentioned above, the way $\boldsymbol{A}$ is defined and strutured and **initiated** is one of the points that distinguish these variants from each other. A core component of **S4** is the initialization of the SSM state matrix $\boldsymbol{A}$ with a specific matrix, the so-called **HiPPO matrix** (High-Order Polynomial Projection Operator) 12, which was shown empirically to be crucial for S4s ability to process long sequences [6]. The original S4 model has a **diagonal plus low-rank** (DPLR) matrix $\boldsymbol{A}$, while S4D has only a **diagonal**

[1]Note that the bar over the matrix was introduced by Gu et al. [5] and denotes the discrete case.

matrix $\boldsymbol{A}$. This matrix can be calculated using advanced mathematical methods and will only be briefly introduced here.

State Matrix. The **HIPPO matrix** is a special strutured matrix that makes it possible to compress the past history of a signal into a state that contains enough information to approximately reconstruct the history.

$$A = \begin{pmatrix} 1 & 2 & & & & \cdots & & & \\ -1 & 2 & 3 & & & \cdots & & & \\ 1 & -3 & 3 & 4 & & \cdots & & & \\ -1 & 3 & -5 & 4 & 5 & \cdots & & & \\ 1 & -3 & 5 & -7 & 5 & 6 & \cdots & & \\ -1 & 3 & -5 & 7 & -9 & 6 & 7 & \cdots & \\ 1 & -3 & 5 & -7 & 9 & -11 & 7 & 8 & \cdots \\ -1 & 3 & -5 & 7 & -9 & 11 & -13 & 8 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{12}$$

$$A_{nk} = \begin{cases} (-1)^{n-k}(2k+1) & \text{if } n > k \\ k+1 & \text{if } n = k \\ 0 & \text{if } n < k \end{cases} \tag{13}$$

The primary challenge in computing the discrete-time State Space Model (SSM) lies in the repeated matrix multiplication by $\overline{\boldsymbol{A}}$, as illustrated in the convolution operation. To address this issue, [4] introduced various methods, namely SSKernelDPLR and SSKernelDiag, designed to produce computationally efficient matrices for further processing. Specifically, the **HIPPO matrix** can be decomposed into either a **diagonal plus low-rank** (DPLR) matrix, resulting in the S4 method, or a purely **diagonal matrix**, which approximates the **HIPPO matrix**, resulting in the S4D method. The way the matrix A represents the latest memory history is mainly by keeping track of the coefficients of a Legendre polynomial [2] The principal conclusion to be drawn is that, among all variations, the configuration of the matrix is of importance to the model's performance, where $\boldsymbol{A}$ with HiPPO is significantly better than the initialization as a random matrix, as the HiPPO matrix keeps track of the coefficients of the Legendre polynomials. The forward pass in the covolution mode takes place in frequency space, since the convolution in time space is a multiplication in frequency space, where the convolution operations are made more efficient by the Fast Fourier Transform.

[2]Legendre polynomials are a system of orthogonal polynomials that can be reduced to the very related orthogonal basis vectors in linear algebra, where the basis vectors are the span of a vector space. This means that all elements in the vector space can be written as a linear combination of these basis vectors, where the coefficients of the linear combination with respect to the basis vectors are called components/coordinates. Consequently, Legendre polynomials can decompose functions, or rather a function is a linear combination of these Legendre polynomials (see Blog post).

3 Approach

3.1 Cached NTM

As the computational complexity of the Neural Turing Machine (NTM) increases with the size of the memory, due to the read and write heads interacting with all memory cells at each time step, we have introduced a minor modification to the standard NTM implementation. In our version, memory interaction during training occurs only a certain percentage of the time. Specifically, prior to each interaction with the memory at each time step, we perform the following operation:

$$M_t = M_{t-1} \circ \left[E - w_t^w e_t^\top\right] + w_t^w a_t^\top$$

$$r_t = \sum_i w_t^r(i) M_t(i)$$

The interaction with memory is determined by:

$$\mathit{interact} = \mathbf{1}_{\{R<M\}}$$

where $R \sim \mathcal{U}(0,1)$ and $M \in [0,1]$. We define the following functions:

$$\mathrm{Read}(M_t, r_t, \mathit{interact}) \in \{0,1\}$$

$$\mathrm{Write}(M_t, r_t, \mathit{interact}) \in \{0,1\}$$

In this modification, M_t is the updated memory at time t, r_t is the read vector, and *interact* is a binary indicator that determines whether memory interaction occurs based on a random variable R and the threshold M. One remark is that this operation is unfortunately not differentiable as the indicator function occurs in this operation. One could use **Policy gradient Methods** [8] to differentiate this part of the operation using reinforcement learning mechanisms, but that is a matter for future work. The computation graph retains its structure as the previous read vectors are incorporated into the subsequent timestep, under the assumption of locality in the data sequence [10]. Since the controller is not stateless, i.e. the hidden state evolves over time with respect to previous states, the controller operates as a cache memory within the NTM. This adjustment aims to reduce the computational load by limiting memory interactions during training.

3.2 NTM-S4D

3.2.1 Variant 1

The modification of the Neural Turing Machine involves replacing the Long Short-Term Memory controller with a state space model, specifically an S4D layer. In this new architecture, homogeneous states are stored in the memory. More specifically, the coefficients of the Legendre polynomial, derived from the HiPPO matrix, are stored as hidden states. These hidden state representations in state space models are more effective for capturing long-range dependencies in time series data.

The read and write heads learn to attend to these hidden states, focusing on the most relevant parts. While transformers excel at this task, SSMs typically struggle with content- and context-aware learning [9]. One noteworthy observation is the ability of SSMs to train in convolutional mode. However, this mode is not utilized in this architecture because the model ideally interacts with memory at each time step. Hence in NTM-S4D we use the recurrent relation formula 14 to train the model. Further research is needed to explore how memory and convolution-specific kernels can be used for particular sequence lengths. We tranform the operations of the new state space model into read and write (only to store the hidden states into the memory):

$$\begin{aligned} \textbf{Read:}\quad x'_t &= \overline{\boldsymbol{A}}x_{t-1} + \overline{\boldsymbol{B}}(u_t + \sum_i w^r_t(i)M_t(i)) \\ &= \overline{\boldsymbol{A}}x_{t-1} + \overline{\boldsymbol{B}}(u_t + r_t) \\ &= \overline{\boldsymbol{A}}x_{t-1} + \overline{\boldsymbol{B}}\tilde{u}_t \\ y_t &= \overline{\boldsymbol{C}}x_t \end{aligned} \tag{14}$$

$$\begin{aligned} \textbf{Write:}\quad m &= \sqrt{\Re(x'_t)^2 + \Im(x'_t)^2} \\ p &= \angle x'_t \\ \text{embedding} &= \text{linear}(m, p) \end{aligned} \tag{15}$$

where $\tilde{u}_t := (u_t + r_t)$.

In the write operation, it is necessary to extract the real and imaginary components of the hidden state. By capturing both the magnitude and phase, the information of the data from the complex domain is preserved within the real domain. The embedding is computed via a neural network and subsequently utilized within the pipeline, as delineated in 2, for writing into the memory.

3.2.2 Variant 2

Variant 2 is computationally more efficient due to its utilization of the convolution operation, which can be parallelized. Consequently, this NTM-S4D variant offers parallelizability in contrast to Variant 1, which requires interaction with the memory at each time step. In Variant 2, interaction with the memory occurs only after processing the entire sequence using the convolution kernel. The output of the S4D block is then used in the heads to interact with the memory. Therefore, memory interaction happens once per sequence, unlike in the vanilla NTM or Variant 1, where interaction occurs at each time step. This process is visualized in Figure 4. One significant aspect of Variant 2 is its simplicity in comparison to Variant 1, both in mathematical formulation and in implementation.

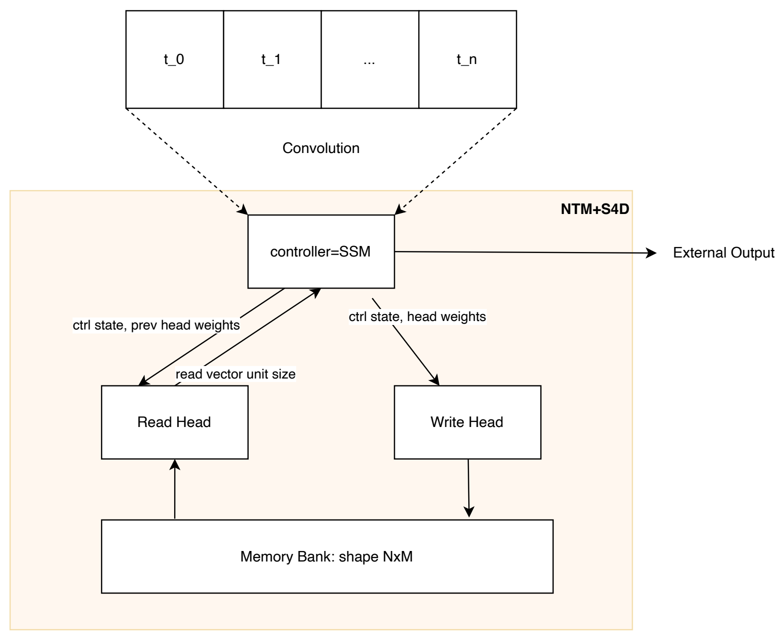

Figure 4: Variant 2 of NTM-S4D

4 Results

Setup. The task employed in this study is a modification of the classic MNIST dataset [2]. Specifically, the dataset was adapted to be sequential, where each image is represented as a sequence of pixels rather than as an entire image at once. A similar variant of MNIST was utilized by Gu et al. [5] in their paper introducing S4. The training setup is consistent across all architectures. The models are trained using the RMSprop optimizer with a learning rate (lr) of 0.0004, α of 0.9, and momentum of 0.95. Additionally, as indicated in the figure, the AdamW optimizer is used with a learning rate of 0.0004, where the hidden state is adjusted with a lower learning rate. Model validation is performed after each epoch on a distinct dataset from the training set. In this study, the models were trained for five epochs, exclusively utilizing variant 2 throughout the experiments.

4.1 Sequential MNIST Task

The sequential MNIST dataset shows a task with long-range dependencies and a high gradient flow through time. The pixel values of each image are fed to the model sequentially to mimic the properties of a time series, where the model must predict the fed image at the end of time. Due to time constraints, the full sequence length of 784 (28*28 pixels) was not used here, but a shorter version such as 64 (8*8 pixels) and 256 (16*16 pixels), labeled MNIST-64 and MNIST-256. These

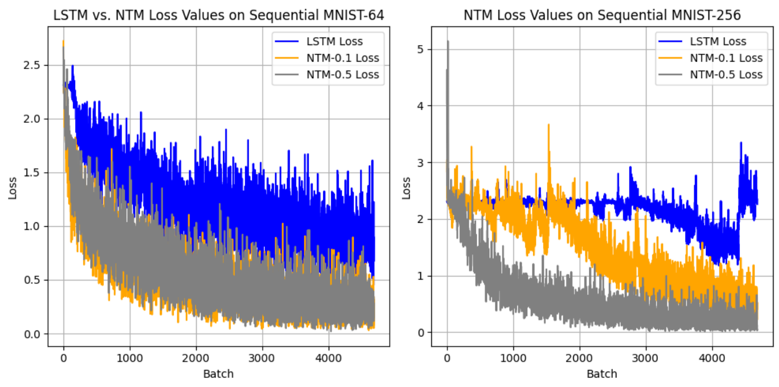

Figure 5: Loss over 5000 batch training for a LSTM and two variants of the Cached NTM, one with 0.5 and the other with 0.1 as memory interaction frequency. The first graph focusses on the MNIST - 64 dataset while the second one displays loss on MNIST-256.

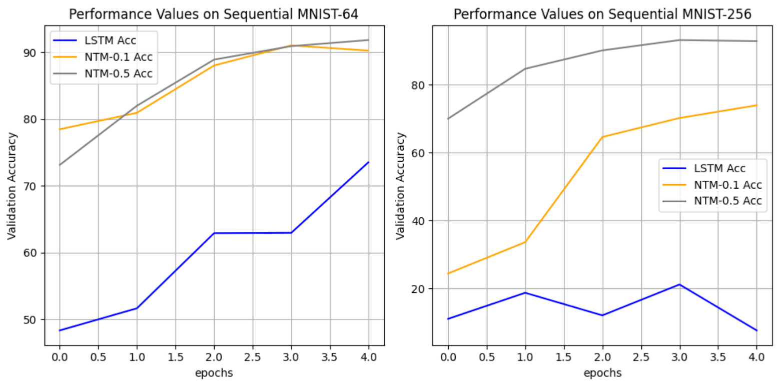

Figure 6: Validation accuracy over 5 epochs for a LSTM and two variants of the Cached NTM, one with 0.5 and the other with 0.1 as memory interaction frequency. The first graph focusses on the MNIST - 64 dataset while the second one displays performance on the MNIST-256.

were created by a transformation in order to retain the information contained as complete as possible. As can be seen in some of the tests, even a sequence length of 64 or 256 is sufficient to detect significant differences in performance and trends.

4.2 Cached NTM performance

To test the baseline performance of the cached NTM approach, it will be compared against firstly the base LSTM and itself with different levels of memory interaction. The metrics used to compare performance are loss over amount of batches seen 5 and validation accuracy per epoch 6.

The results depicted in Figure 5 clearly illustrate that the NTM-Cached approach achieves loss convergence to zero significantly faster than the vanilla LSTM. In the MNIST-64 run, both NTM-Cached variants demonstrate similar convergence speeds. However, in the MNIST-256 run, the NTM-Cached variant interacting with its memory 50% of the time converges faster and more stably. This improved performance is likely due to the longer sequence length, allowing for more accurate modeling of sequences in memory. Increased interaction with memory enhances the NTM's ability to capture long-range dependencies, a trend that would likely be even more pronounced on the MNIST-784 dataset and with the baseline NTM. Additionally, the results underscore the lower inductive bias of the NTM, which, unlike the parameter-bound LSTM, generalizes better with minimal optimization. This conclusion is further supported by the consistent training setup used throughout the experiments.

4.3 NTM-S4D performance

Figure 7 illustrates the loss values of various models, including LSTM, NTM-0.1, NTM-0.5, and NTM-S4D, on the Sequential MNIST-256 dataset over a series of batches. For the purpose of this analysis, we will focus on the comparison between NTM-S4D and NTM-0.5.

At the beginning NTM-S4D, appears to have a slightly higher initial loss compared to NTM-0.5. NTM-S4D demonstrates a rapid decrease in loss values in the initial phase of training, indicating a faster convergence rate. This is likely due to its efficient interaction with the memory, which is optimized to occur only once per sequence. This strategy contrasts with the more frequent memory interactions in NTM-0.5. Over the batches, NTM-S4D maintains a consistent downward trend in loss values, showcasing its robust performance and efficient caching mechanism. NTM-S4D exhibits a more stable training process with fewer fluctuations in loss values compared to NTM-0.5. The grey line for NTM-0.5 shows more significant oscillations, indicating variability in performance. This instability could be attributed to the less efficient memory interaction mechanism of NTM-0.5. By the end of the training period, NTM-S4D achieves lower loss values than NTM-0.5, reinforcing its superiority in handling sequential data. The final loss values for NTM-S4D are consistently lower, suggesting better generalization and learning efficiency. The performance of NTM-S4D can be attributed to its more compact representation of history through the use of the Legendre polynomial and its integration with the reads via attentional processes.

Furthermore figure 7 illustrates the loss values of two different configurations of the NTM-S4D model on the Sequential MNIST-784 dataset. NTM-S4D-AdamW-HiddenSlow shows a rapid decrease in loss values in the early stages of training, converging more quickly than NTM-S4D-RMSProp. This indicates that the AdamW optimizer, when used in conjunction with the hidden slow variant, facilitates a more efficient initial learning process. However, the application of consistent learning rates across all parameters in NTM-S4D-RMSProp suggests that employing state space models and memory mechanisms stabilizes the training of Neural Turing Machines. Furthermore, it demonstrates that state space models can be effectively utilized with uniform learning rates, when augmenting with memory.

Figure 8 presents a comparative performance analysis of LSTM, NTM, and

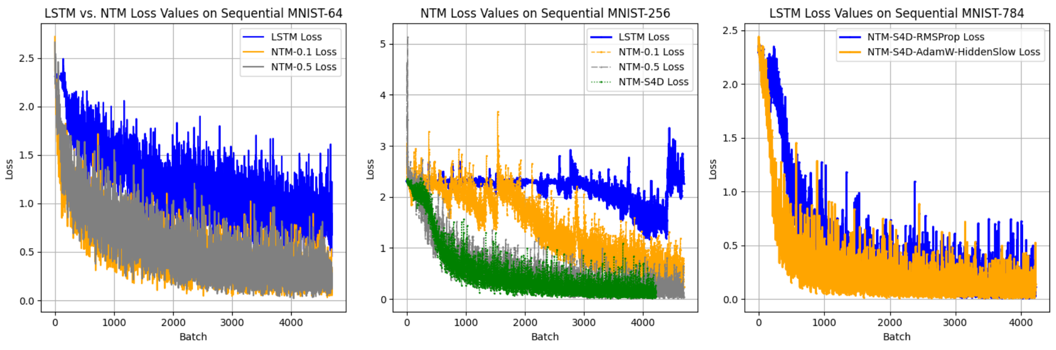

Figure 7: Comparative Loss Values of LSTM, NTM and NTM-S4D on Sequential MNIST.

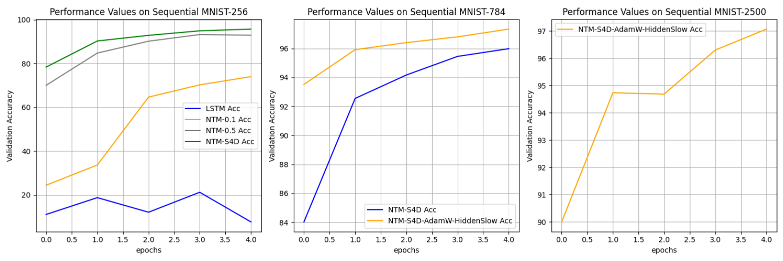

Figure 8: Comparative Validation Values of LSTM, NTM and NTM-S4D on Sequential MNIST.

NTM-S4D models on various Sequential MNIST datasets. In the 256-Sequential MNIST scenario, the NTM-S4D model demonstrates superior performance over the other methods, particularly evident after five epochs. For the more complex 784-Sequential MNIST scenario, the NTM-S4D paired with RMNSprop exhibits consistent performance as the sequence length increases. Notably, the NTM-S4D achieves higher accuracy compared to a single S4D Block, reaching approximately 90% after just one epoch of training [5]. Additionally, the combination of the AdamW optimizer and slower hidden learning rates yields the best overall performance. Further it can be seen that, as the sequence length increases, the models performance does not decrease. This figure underscores the superior performance of various NTM-S4D configurations compared to the base models in terms of both loss and validation accuracy across different scales of Sequential MNIST datasets.

5 Conclusion

A Neural Turing Machine (NTM) equipped with state-space models and state-space models with extended memory shows superior performance compared to all baseline models. Performance analysis of LSTM, NTM and NTM-S4D models on sequential MNIST datasets reveals several important findings. The NTM-S4D model outperforms the other models on all tasks and shows significant improvements after only a few epochs, indicating the sampling efficiency of these models. For the more challenging 784-sequential MNIST dataset, the NTM-S4D model shows robust performance, especially when using the AdamW optimizer with slower hidden learning rates. However, using a constant learning rate to train state space models with memory augmentation also proves to be effective. These results emphasize the effectiveness of the NTM-S4D configurations in achieving superior loss and validation accuracy across different dataset types, demonstrating their clear advantage over traditional models.

References

[1] Mark Collier and Joeran Beel. Implementing neural turing machines. 2018.

[2] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[3] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[4] Albert Gu, Karan Goel, Ankit Gupta, and Christopher Ré. On the parameterization and initialization of diagonal state space models. *Advances in Neural Information Processing Systems*, 35:35971–35983, 2022.

[5] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

[6] Albert Gu, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Ré. How to train your hippo: State space models with generalized orthogonal basis projections. *arXiv preprint arXiv:2206.12037*, 2022.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[8] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[10] Liang Yuan, Chen Ding, Wesley Smith, Peter Denning, and Yunquan Zhang. A relational theory of locality. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(3):1–26, 2019.