Neural Turing Machines: Programming Language

Justin Garrigus
University of North Texas
Department of Computer Science and Engineering
justingarrigus@my.unt.edu

Abstract

Standard deep learning models learn transformations on data that reduce complex but abstract representations of a problem into simple, separable categories. Recurrent neural networks (RNNs) process sequences of data with respect to time in such a way that inputs at one time step could modify outputs at a much later time step. Long short-term memory (LSTM) is one of the most popular forms of RNN, but it has limitations at generalizing operations when given inputs that exceed the length for which it was trained for. The neural Turing machine (NTM) model fixes this by combining a neural network with an external addressable memory bank which the model can use to store intermediate data while it focuses on learning a general algorithm instead of a specific transformation, allowing it to generalize to much longer inputs. This paper proposes distributing individual tasks across a collection of discrete NTM units which are combined through an NTM-controller. This is utilized to create a novel programming language built entirely from deep learning models: each line of a user program is passed through an encoder-decoder to scrape arguments and a linear classifier to apply attention to a set of NTM units representing individual instructions. This work shows a pseudo-curriculum learning approach that can used as a building block to create fully differentiable computers.

1. Introduction

Traditional deep learning tasks focus on finding ways to change the representation of data by iteratively extracting features that identify aspects of an input in such a way that a given example can be uniquely identified or separated from other examples. Deep learning models are typically constructed from a collection of discrete components, where recurrent networks involve cycles between components and feedforward networks do not. Each component contains a single operation that takes a matrix, vector, or scalar as input, and modifies it in some way with parameters to yield a transformed output. The goal of training the network is to learn parameters that minimize a final cost function between training examples the network has seen and examples the network expects to see in the future. Different methods are used during training to improve the model's ability to generalize across examples and to prevent models from overfitting parameters to observed examples.

While usual deep learning tasks find representations of data, the means by which data is operated on is usually fixed. Training does not typically change the number of operations being performed, and the order of operations among data and the ways in which layers are connected are constant. For example, training an image-classification network like AlexNet simply consists of repeatedly applying slight modifications to scalar constants used in dot-product operations in order to improve the similarity between the output of the network and a target vector. The ultimate goal of this domain of network is to discover a purely-numerical transformation on a collection of input values, but this has limitations when it comes to generalizing. AlexNet would perform very poorly if the resolution of the input image was doubled without retraining, since the numerical transformations it was meant to express do not apply to larger images, similar to how a graphical 2D to 3D projection matrix cannot generalize to transforming higher-dimensional vectors without reconstructing a new matrix entirely.

In this sense, an algorithm can be interpreted as a superset of a transformation. Algorithms contain abstract instructions used to guide and generate transformations. A sufficiently-complex algorithm could express the interrelation between pixels on an image of any resolution by perhaps defining rules to create transformations on a per-image basis, which an architecture like AlexNet is incapable of doing on its own.

The neural Turing machine (NTM) [1] is an example of such an architecture that has the ability of learning algorithms instead of transformations. This is expressed by the fact it can generalize to much larger inputs than it has seen during training: when given a simple task like copying input values to an output stream, it performs equally well on inputs of size 80 when it was only trained on inputs up to

size 20. Conversely, a standard LSTM network would degrade very quickly when it encounters inputs that are larger than it had seen during training.

The original paper that introduces the NTM focused on teaching the model fairly simple tasks, like copying data, set-retrieval, and n-gram prediction. This paper expands on previous work by introducing 8 new tasks, each dealing with simple elementary operations on an input matrix similar in concept to assembly instructions. Each task is contained within its own NTM model, which are all connected to an NTM controller. A programming language is created that takes user-inputted strings, with strings containing commands corresponding to individual NTM components, which the controller accepts as input in order to control the activations on its NTM components. The resulting model consists of 10 individually-trained sub-networks that combine to form a simple interpreted programming environment.

The rest of the paper is organized as follows. Section 2 describes the background and related work in neural Turing machines, sequence-learning, and curriculum learning. Section 3 details the proposed architecture. Section 4 gives the datasets and metrics used to train and judge the performance of each model. Section 5 discusses the implementation of each model and roadblocks that were solved. Section 6 provides experimental results in training and inference of each model. Section 7 gives the conclusion.

2. Related Work

Recurrent neural networks (RNNs) [2] are Turing-complete, meaning they are capable of solving any computational problem [3]. Long short-term memory (LSTM) is the most popular method of implementing RNN, and it is often used as a benchmark to compare sequence-processing tasks against [4]. LSTMs contain stored variables alongside weights, and use gates to control both the flow of data and the memory contained within. Due to their internal memory structure and shortcut connections, LSTMs are much better at preserving time-dependent long-term data requirements than traditional feedforward networks can, though this has limitations when generalizing to inputs that far exceed their training examples. Furthermore, in the context of modern-day computers under the von Neumann architecture [5], an LSTM could be visualized as a program containing small stored variables like registers.

The neural Turing machine [1] fixes the shortcomings of standard RNN networks by encouraging long-term connections through an RNN controller accessing an external addressable memory bank, depicted in Fig. 1. This allows the controller to use their own stored variables to represent algorithmic details instead of input features necessary for transformation. NTMs are fully differentiable and end-to-end trainable due to a fuzzy attentional mechanism on the read and write heads. When the controller wants to access memory, it must access all combined rows within its memory matrix at once, using a softmax probability distribution to control which cells take priority.

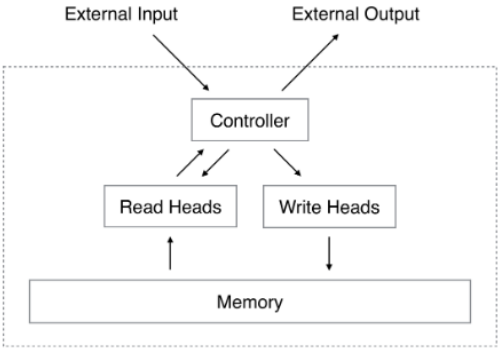


Figure 1. Neural Turing Machine Architecture. The controller is either a feedforward network or LSTM, and it processes the inputs while modifying its memory bank with its write head. The output of the network is generated from a linear layer connected to the controller and the read head.

Several other NTM variants have been proposed since the original's introduction in 2014. [6] extend the model to support new tasks explicitly involving data structures like binary trees and linked-lists, and use a more-sophisticated "pointer" memory-addressing mode that remains fully-differentiable. [7] removes the full-differentiability of the model in order to improve performance by reducing memory accesses and utilize reinforcement learning to train on simple tasks, and [8] further develops the model for reward-based learning to succeed on maze-traversals. [9] introduce the Lie-access neural Turing machine which improves on the model's ability to index memory cells relative in location to each other through positional transformations like shifts and rotations. [10] and [11] are both direct improvements of the original paper, changing the organization of recurrent connections and the addressing mechanism in order to solve problems that can be visualized as relational graphs like graph-traversals and question-answering.

Besides neural Turing machines, this paper focuses on creating a programming interface using NTMs as instructions, alongside a feedforward network for classification of instruction types [12] and an LSTM encoder-decoder for scraping arguments from commands [13]. Using individually-trained submodels can be visualized as a subset of curriculum learning [14], and training them jointly or teaching them to utilize the same addressable memory space like the typical von Neumann achitecture is a valid area for future work.

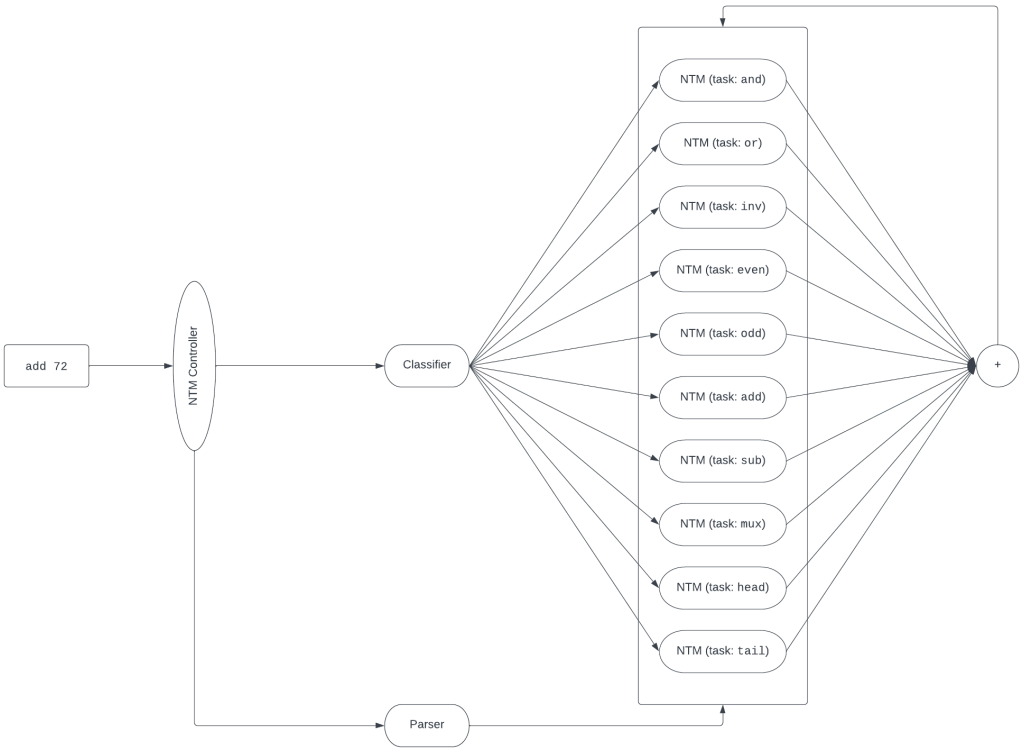


Figure 2. Architecture overview. A controller takes individual instructions as input, passing them to a classifier and a parser. The classifier calculates attention on the NTM components, and the parser turns an optional parameter from the command into a vector.

3. Architecture Overview

The ultimate goal of the project was to make a fully-differentiable interpreter of an input programming language. Although the result is not end-to-end trainable, certain restrictions were taken to ensure that future conversion into a differentiable model was possible.

The proposed model reinterprets "tasks" from the original paper [1] as "instructions", defining a discrete operation on a matrix of data to yield a new matrix of data. Each instruction is showed in table 2. Instructions are inputted line-by-line into an NTM Controller, which contains a classifier and parser submodel. We imposed a challenge on ourselves to require that the input commands be represented as characters rather than pre-processed numbers: it is up to the classifier and parser to recognize patterns between these characters and to convert them into formats it can use.

The classifier is a simple feedforward network with `n` hidden layers and `m` hidden units in each layer. It determines an input and output attention for each NTM unit, where the sum of each attention must equal 1. Since each instruction yields a matrix of the same dimensions, the matrices can be pairwise-added together. The degree to which matrices are accumulated is accomplished through a softmax function which determines how much of a given NTM to accumulate into the final resulting matrix. For example, an input command "`add 72`" should ideally lead to a classification vector `[0, ..., 0, 1, 0, ..., 0]`, where the 1 cooresponds to the index of the `add` NTM unit. The classification vector is multiplied against each NTM output, and the results are summed together and passed through a sigmoid nonlinearity to yield the resulting matrix.

The parser scrapes the optional argument from the command input and returns a binary string. Each instruction accepts an input matrix, but some instructions like `add`, `head`, and `mux` require parameters to be placed in specific rows or columns. The binary string returned from the parser is passed to each NTM unit's input, with some units ignoring the input and some units copying it. We found that an encoder-decoder LSTM model based on [15] with `i` hidden layers and `j` hidden units in each layer works best for the parser.

4. Datasets and Metrics

The output of the language interface depends entirely on the performance of its sub-models. Therefore, the performance of each sub-model will be reported individually. This project does not rely on any external datasets, as all tasks are essentially simple instructions that are easily gen-

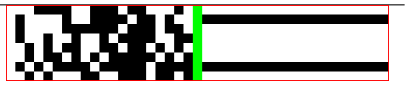
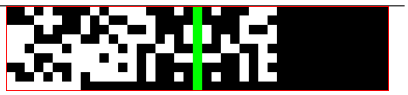
Command	Parameters	Description	Example (Input/Output)
add	$[0, 2^{width}]$	Enables rows which align with the parameter mask, ignores all other rows.	
sub	$[0, 2^{width}]$	Disables rows which align with the parameter mask, ignores all other rows.	
and	None	Enables columns only if all bits within an input column are enabled.	
or	None	Enables columns only if they contain at least one enabled bit within an input column.	
even	None	Enables a column only if it contains an even number of enabled inputs.	
odd	None	Enables a column only if it contains an odd number of set inputs.	
head	$[0, length]$	Copies *n* columns from the front, where *n* is a mask in the last row.	
tail	$[0, length]$	Copies *n* columns from the back, where *n* is a mask in the last row.	
inv	None	Flips (inverts) each bit in the input.	
mux	$[0, width]$	Copies a single input row across all output rows, given by a mask.	

Table 1. List of instructions. Commands are in the format "`command [param]`" (e.g., "`add 173`"), where the optional `param` falls within a given range. For the Example column, the image to the left of the green divider represents the inputs, and the image to the right of the green divider represents the outputs. Each output has the same dimension (in this case, 8x20), but inputs which contain parameters may need to specify an additional row or column. The `add`, `sub`, and `mux` instructions include a parameter in the last column, with a separator between the input stream and the parameter. The `head` and `tail` instructions include a parameter in the last row. The `and`, `or`, `even`, `odd`, and `inv` instructions do not require parameters.

erated manually, and all commands follow a finite grammar.

4.1. NTM Tasks

Each task contains a generator function that produces an input and expected output matrix. In order to improve the performance of each NTM, the input matrices may not come from a completely random distribution (i.e., the `and` task must have a high probability of generating completely-enabled columns in order for the model to learn to associate these columns to enabled outputs, while the `or` task does not have this same requirement).

Similarly, instructions with column-parameters like `add` and `mux` require a delimiter bit to be placed on the second-to-last column. Instructions with row-parameters like `head` and `tail` do not require delimiter bits, as the parameter can be passed directly into the final input row. Finally, the `mux` task is unique in that proper semantics specify that only one bit of the parameter column should be set; due to the net-

work consisting of only deep learning models, there is nothing preventing the user from specifying more values to the parameter channel, but this results in undefined behavior.

4.2. NTM Controller

In order to reduce the amount of required pre-processing to the data (as too much pre-processing would negate the novelty of the model), input commands are passed to the two components of the NTM Controller and processed individually. For both sub-models, a generator function retrieves a random command from the list of commands, adds a random parameter if applicable, and converts the string to a list of numbers (such that each character in the command corresponds to an index into a predefined vocabulary). As such, a string like "`add 72`" with vocabulary "`?abcdefghijklmnopqrstuvwxyz 0123456789`" may become `[1, 4, 4, 27, 35, 30]`. Due to the small scope of the language grammar, an exhaustive list of input commands can be generated.

For this project, the classifier sub-model is a linear feed-forward network with `n` hidden layers and `m` hidden units in each layer. The parser sub-model is an LSTM encoder-decoder with `i` hidden layers and `j` hidden units in each layer. The exact values of `n`, `m`, `i`, and `j` are determined via iterative optimization, detailed in section 5.

4.3. Metrics

Each sub-model is judged for performance using the standard Cross Entropy Loss. The size of the datasets are comparatively very small, and the bounds of the inputs are ultimately known in advance, but this means that regularization to unseen inputs is not an issue (for example, a perfect parser component is possible if the LSTM learned to map the different inputs directly to their associated outputs, with no creative transformation required). A greater concern is generalization to inputs of different lengths: the original NTM paper [1] emphasized their ability to generalize to inputs drastically larger than the model trained for, so we follow this idea by focusing on performance as a factor of input length as well.

5. Implementation

Our approach was implemented in PyTorch. In order to reduce the optimization space and ensure that all models have a chance to converge, early-stopping is applied to each training regimen, detailed in Algorithm 1. The training procedure continues to traverse until the validation loss stops improving over a set number of epochs. A goal (required) validation loss is also specified such that the model will repeatedly attempt to improve itself (e.g., by increasing how many hidden layers it contains) on validation stagnation until its best validation loss is below the goal. Also, in this method, no upper-limit to the epoch count is specified due to tasks like `mux` which seem to stagnate over long stretches of time before suddenly improving and converging to zero loss.

Algorithm 1 Early-stopping to improve performance

Require: $epoch_{stop} > 0$
Require: $val_{req} \geq 0 \ \ \& \ \ val_{req} \leq 1$
  $epoch_{idx} \leftarrow 0$
  $val_{best} \leftarrow 0$
  $val_{idx} \leftarrow 0$
  loop
    $val_{err} \leftarrow train\_one\_epoch()$
    if $val_{err} < val_{best}$ **then**
      $val_{best} \leftarrow val_{err}$
      $val_{idx} \leftarrow epoch_{idx}$
      $save\_model()$
    else if $epoch_{idx} - val_{idx} > epoch_{stop}$ **then**
      if $val_{err} > val_{req}$ **then**
        $update\_layer\_configuration()$
        $retrain()$
        return
      else
        break
      end if
    end if
    $epoch_{idx} \leftarrow epoch_{idx} + 1$
  end loop

5.1. NTM

The NTM portion extended an existing repository [16], which attempted to follow the original paper [1] as close as possible and managed to match the authors' performance on several tasks. The methodology of this section was as follows: (1) implement data generators for each new task, (2) define custom NTM model configurations to explore in an automated search, and (3) leave the model training over several nights until they converged. Since the NTM model was adopted from an external repository, there were several sub-optimal implementation details, the most important of which was the fact the NTMs must be trained on the CPU only. We attempted to transfer the code to the GPU, but this incurred a significant increase in training time due to high communication latency.

The NTM's controller was internalized as a PyTorch LSTM component with a binary cross entropy criterion (since each output cell essentially represents a binary classifier) and a root mean square propagation optimizer. The memory is simply a two-dimensional PyTorch tensor, where reads and writes consist of matrix multiplications with attentions from the read and write heads. The heads themselves are a single linear layer with sigmoid nonlinearity. It is important to note that the heads do not use a softmax non-

linearity: this helps, for instance, the read head to equally address multiple cells at once and accumulate the results if it requires, which was used extensively for the N-gram task in the original paper [1].

5.2. Classifier

The classifier submodel consists only of pairings of linear layers, ReLU nonlinearities, and a final softmax. The softmax is important to allow the architecture to produce a single normalized matrix after each task NTM calculates their own matrix. While only one command is supplied by the user to the programming interpreter at a time, every NTM unit is invoked together, and the classifier should ensure the model properly ignores the NTM units which do not coincide with the user's intentions.

Commands are passed to the classifier as lists of integer embeddings cooresponding to indexes into a predefined vocabulary. To train the classifier, stochastic gradient descent was used as an optimizer along with a cross entropy loss criterion. These are the most common combination of optimizer and criterion, and most recommended in use for feedforward networks as a "default" option. The early-stop tactic specified in algorithm 1 is still used, but the number of unique training examples is very small so it results in negligible time savings. The most optimal results with respect to parameter count was found to be one hidden layer of length 128 with a learning rate of 0.1.

5.3. Parser

The parser submodel is based off the LSTM encoder-decoder specified in [15]. While better, more modern architectures exist for this task, we wanted to stick to a simpler model that could be more easily implemented for manual training. Since the model's source and target vocabularies are relatively small (with the output of the parser being a binary string of fixed length) we attempted to use an identical feedforward structure to the classifier, the results of which are detailed in section 6.

Both the encoder and decoder contained a PyTorch embedding layer fed into an LSTM module and dropout, though the decoder had an additional linear layer to coerce the LSTM into the correct shape. The optimizer used was Adam (which is more popular with NLP tasks like sequence generation) and the criterion was cross entropy loss. The encoder and decoder both contain 4 LSTM layers each with a hidden dimension of 512. Additionally, the embedding dimensions were 512, and the dropout was 0.5. To avoid exploding gradients typical to recurrent networks, the gradients were clipped prior to stepping forward the optimizer during each training iteration.

6. Results

Since each component is discrete and trained individually, this section describes the training and inference results independently of each other. When combined, the architectural performance is proportional to the performances of each component.

6.1. NTM

Table 2 gives, for each task, the loss and number of sequences required to train, and the size and layer count of the associated controller. Each controller was started at a default controller size and layer count of 100 and 1, respectively. Due to the high training time required to gain acceptable performance on each model, the early-stopping algorithm depicted in Algorithm 1 was essential to automate training.

Several of the tasks are expressed as pairs, like `add` and `sub`. Since these pairs have very similar algorithms, it is natural to assume they would have equivalent layer counts and controller sizes, but the `even` and `odd` tasks curiously do not exhibit this, as the `odd` task contains 54x more parameters than the `even` task and still contains much worse performance, shown in 3 to contain volatile validation losses at times. Besides this, every other task has less than 0.01 error, with some tasks attaining perfect results. The early-stopping method employed in this project demonstrates its benefit again when it comes to the `mux` task shown in 4: gradual progress made at different sequence intervals prevented the model from exiting, and the sudden drop in loss allowed the model to converge much later than any other task.

Additionally, 5 shows an example of how a task generalizes to inputs of length it hasn't seen before. For the `mux` task, the NTM was trained on sequences between 5 and 20 values in length. The figure shows the model is more than capable of generalizing to higher sequence lengths in certain cases while maintaining a low loss. The sudden jumps in loss are possibly due to algorithmic gaps in the NTM controller components.

6.2. Classifier

The classifier demonstrated good performance with respect to its size, with its history shown in 6. Due to the tiny dataset size, its fair to assume the model memorized the dataset. Although, the validation and training losses interestingly never met to the same point despite coming from the same datasets. As mentioned previously, it was attempted to use ascii character embeddings for input strings rather than indexes into an arbitrarily-defined vocabulary string, but this never managed to converge regardless of model size. This is expected, as neural networks tend to perform better when inputs are closer to 0, and ascii alphanumeric characters are much higher than vocabulary indexes.

Task	Loss	Sequences	Parameters	Controller Size	Controller Layers
add	0.00004	22000	62860	100	1
sub	0.00068	17000	62860	100	1
and	0.00978	36000	18121860	800	4
or	0.00000	20000	808260	250	2
even	0.00065	50000	62460	100	1
odd	0.65477	30000	3297060	400	3
inv	0.00032	22000	62460	100	1
mux	0.00000	73000	62860	100	1
head	0.00002	26000	62860	100	1
tail	0.00003	26000	527460	200	2

Table 2. Training results for each task. Gives the loss from a single training example, the number of sequences it took to train the model to become optimal, the number of parameters, the size of the controller component, and the number of layers within the controller. Each task began with a controller size and layer count of 100 and 1, respectively. By following algorithm 1, each task was trained to continuously increment their size and layer count until they could not improve any further.

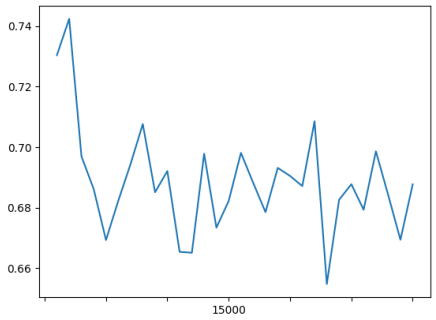


Figure 3. Training history for the `mux` task.

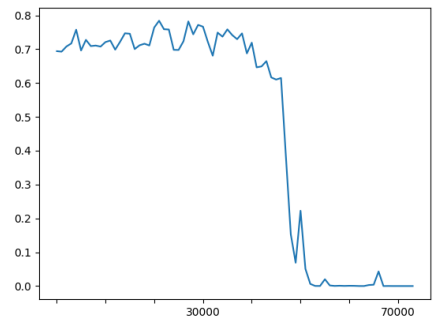


Figure 4. Training history for the `mux` task.

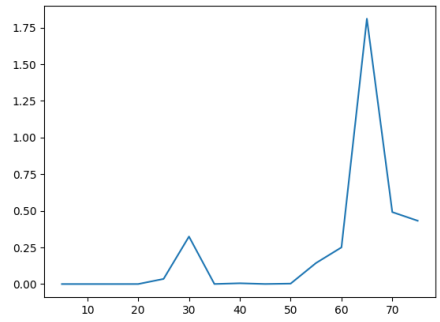


Figure 5. Generalizing to long inputs for the `mux` task

6.3. Parser

At a glance, the parser training history shown in 7 achieved results closer to what was expected for the classifier: the validation and training losses come from the same dataset, so they converge to the same point. Although, the accuracy of the model never reached a perfect score like the classifier, meaning that some bits in the outputted binary sequence were incorrect in some way.

On closer inspection, this appears to be a foundational failure of the encoder-decoder: since the vocabulary consists only of two output tokens (”0” and ”1”), swapping the order of some tokens or misplacing a token would result in a small performance drop in a real language model, but it should result in very poor performance for a model meant to act as an interface to a programming language meant to be axiomatically sound. The parser architecture used in this paper was adopted from [15], which originally used it in the task of English to German caption translations. Understandably, swapping a noun and a verb in English may result in a grammatically-incorrect but still-readable translation, but swapping two adjacent bits in a binary string would result in an entirely-new binary string. Therefore, future work should use a different parser with possibly a different crite-

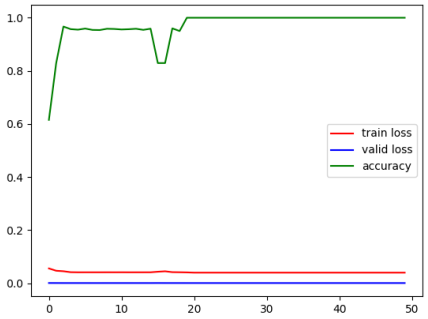


Figure 6. Training history for the classifier.

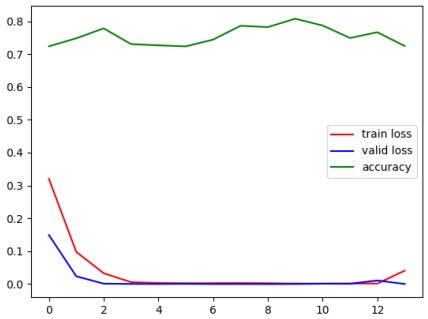


Figure 7. Training history for the parser.

rion to encourage the model to keep binary values in their correct orders. Regardless, the model achieves a high performance of 73%, considering its a language model trained on converting decimal digits into binary masks.

7. Conclusion

This paper proposed a new programming language interface built entirely from neural turing machines, encoder-decoders, and feedforward networks. Each network was trained individually and combined to yield a functional language of 10 instructions. While the model does fail to yield the correct output in some cases due to an incorrect parsing of binary strings from a user-inputted command, an improper attention from the classifier, or an incorrect output matrix generated from an NTM unit, the model produces correct matrices in most cases. Future work should expand on this concept to make the architecture fully-differentiable, and different submodels like a transformer for the parser and NTM controller should be experimented with to possibly observe better performance.

References

[1] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, 1410.5401, 2014. 1, 2, 3, 5, 6

[2] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: an introduction. *The MIT Press*. 2014. 2

[3] Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132-150, 1992. 2

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735-1780, 1997. 2

[5] John von Neumann. First draft of a report on the edvac. 1945. 2

[6] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *ERCIM News*. 2015. 2

[7] Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines. *arXiv*. 2016. 2

[8] Rasmus B. Greve and Emil J. Jacobsen. Evolving neural turing machines for reward-based learning. *GECCO*, pages 117-124, 2016. 2

[9] Greg Yang and Alexander M. Rush. Lie-access neural turing machines. *CoRR*, 1611.02854, 2016. 2

[10] Alex Graves and Greg Wayne et. al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538:471-476, 2016. 2

[11] Caglar Gulcehre, Sarath Chandar, Kyunghyun Cho, and Yoshua Bengio. Dynamic neural turing machine with soft and hard addressing schemes. *arXiv*. 2016. 2

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. *MIT Press*. 2016. 2

[13] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *Conference on Empirical Methods in Natural Language Processing*. 2014. 2

[14] Yoshua Bengio, Jérôme Louradour, Jonan Collobert, and Jason Weston. Curriculum learning. *Association for Computing Machinery*, pages 41-48. 2009. 2

[15] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. *Advances in neural information processing systems*, 27. 2014. 3, 6, 7

[16] Guy Zana, Jules Gagnon-Marchand, and Mark Goldstein. Pytorch neural turing machine. *GitHub*, https://github.com/loudinthecloud/pytorch-ntm. 5