Universität Hamburg
Department Informatik
Knowledge Technology, WTM

Neural Turing Machines

Seminar Paper

Neural Networks

Vincent Rolfs, Yiyao Wei
vincent.rolfs@studium.uni-hamburg.de
9wei@informatik.uni-hamburg.de

July 17, 2020

Abstract

Contents

1 Introduction

Around the '50s, the mathematical computer model Turing machines along with Von Neumann architecture accelerate the birth of digital computers. The basic principle of Von Neumann architecture is that elementary operations, logical flow control, and external memory are the three fundamental mechanisms for the design of general-purpose digital computers [10]. This trend (later termed symbolism) not only inspired computer scientists and engineers to build amazing devices such as digtial computers, but also enriched psychologies and many other researchers to try to understand knowledge processing and to explain how cognition implemented in the human brain. At that time, people believe that the human cognitive process or thinking is just a pure symbol-manipulation process that follows a set of instructions.

The next movement in AI and cognitive science is connectionism, which claims that information is coded in a distrubuted fashion in the neural substrate. Artificial Neural networks are one of the most successful modeling techniques out of connectionsim from nowadays point of view. Around the same time, the study of human cognition from empirical psychology and neuroscience started to reveal the mechanism of working memory, and its limitations.

With the advent of recurrent neural networks (RNNs), artificial networks start to have some short term memory. However, the amount of internal memory is far from efficient for solving complicated tasks. Even the powerful LSTM (long short-term memory) still suffers from forgetting after a short period of time [1]. Moreover, the memory inside the networks is difficult to access and interact with externally.

Neural Turing Machine (NTM) is a novel machine learning architecture that augments a traditional neural network with external memory and aims to enable the network to learn an algorithm that interacts with this memory. This is done in a differentiable way so that methods like gradient descent are applicable. In other words, all operations in this architecture are differentiable so we can optimize the model with backpropagation. The architecture of this hybrid system is analogous to Turing machines or Von Neumann architecture, so the authors call it Neural Turing Machines, just like what Turing did to finite state machines (coupling an infinitely long memory tape to finite state machines). According to the experiment results, NTMs outperform LSTMs on all five tasks [4]. They are copying, repeat copying, associative recall, dynamic N-Grams and priority sort.

nange this ac-
rding to our
periments

In this report, our goal is to investigate the capabilities of NTMs,with a comparison between other memory augmented neural networks. The focus is on replicating the results from [3] on copying, repeat copying, and associative recall, as well as implementing the sorting task.

2 Background

To better understand NTMs and its root, we first talk about what the normal Turing machines are, and its impact on modern computer and on theoretical computer science. We then give a brief history of Memory Augmented Neural Networks, including the original paper by Graves et. al that proposed Neural Turing Machines [4], the paper by Collier and Beel that presents a stable implementation of NTMs [3], as well as the paper by Gravel et. al that introduced the Differentiable Neural Computer (DNC), an extension of the NTM [5]. Furthermore, we explain how a NTM works.

2.1 Turing Machines

First introduced by the mathematician Alan Turing in 1936, Turing Machines [8] are hypothetical devices that can simulate any computable algorithm. In Turing's original paper, he described the term "automatic machine" as a finite state machine that equips a "head" moving in the direction of left or right to write to and read symbols from a tape. Tapes are composed of many individual cells. Each symbol goes into exactly one cell. And there is a finite number of symbols that can be written to the tape, denoted by 0 and 1. We can see the tape here as an external memory for the finite state machine. Based on what it reads from the tape and instruction from current state, the head can make the next movement on the tape. For example, the head can move to the next cell to the right and write a symbol on it, erase a symbol on current position or just move 3 cells left, and so on. Theoretically, Turing machines are capable of performing the logic of arbitrary computations as long as the tape is infinite long.

Turing machines have many types: deterministic, non-deterministic, single head, multiple heads, and so on. All types of Turing machines can be stimulated by a universal Turing machine on arbitrary input [9]. This means no matter which type of Turing machines, they all share the same definition and properties of the universal Turing machine. Nevertheless, the concept of Turing Machines is one of the fundamental models for the arithmetic operation of the digital computer.

2.2 Memory Augmented Neural Networks

2.2.1 Long Short-Term Memory

LSTM is a variant of recurrent neural networks (RNN). Unlike Turing Machines, the family of RNNs introduces directed circles (internal vectors) to store memory of past events and external inputs. One advantage of RNNs is that interanl states are dynamic and distributed cross the network. The dynamic state is the key because

the property unlocks the potential of context-dependent computation. In addition, the distribution of states also allows RNNs to do significantly larger memory and computational operations [4]. As a result, RNNs like LSTM can better handle data in temporal space over other machine learning algorithms. What's more, RNNs are been proven to be *Turing-complete* [7], means that in principle we can use RNNs to solve arbitrary computational problems. Note that by assumption, the memory space and running time do not have a limit. Although Turing-complete machines are guaranteed to output a result, we don't know how much memory it needs, and how long it takes.

Since LSTM is a powful RNN, it is the ideal baseline for comparasion between memory augmented neural nets. We use vanilla LSTM cell directly from TensorFlow.

2.2.2 Neural Turing Machines

Gradient-based neural networks perform optimization using gradient descent, which requires the differentiability of the function it observes. NTM [4] takes advantage of it. By coupling an external memory space, the neural networks can simulate an end-to-end differentiable computer. In other words, external memory is essential for neural nets to learn to perform programs that computers can stimulate, but machine learning models are struggling with. Like Turing machines, the external memory is independent of heads and acts as a knowledge base. In contrast to Turing machines and digital computers, NTMs interact with memory by portions, instead of by a single cell.

Since the memory matrix is very large, it is better to focus on a specific region of the memory when heads interact with memory. In order to do this, the author introduces a selective (soft) attention model to weighting over portions of the memory. This is done by the controller. At each timestep, the controller returns a parameterized distribution over the locations in the memory matrix. In the view of Turing machines, these parameterized outputs of neural nets are essentially the "heads" since they are responsible for processing read and write operations. Both read and write heads receive its own normalized weighting matrix over the locations in the memory. The heads then use the inforamtion to determine which the head interacts at each location.

Additionally, this internal weighting can be computed follow either or both content-based and location-based addressing mechanisms. The benefit of using two addressing mechanisms for weighting is to enable distinct modes of the interaction between the heads and the memory matrix. If the controller only uses the content key, the interaction looks like getting information from an associative map. If the controller only uses the location, the interaction is more like iterative shifting (list iterator). If using both, the interaction then would be slicing an array from a matrix according to the indexes we have. Hence, each mode can be seen as a

simulation of certain kinds of data structures and accessors.

Despite the fact that the code of original NTMs is not open source with the publishing of the paper [4] in 2014, many researchers were excited about it and gave their own implementations [1][2][3][4][5][6][7] written in a variety of machine learning frameworks (Theano, TensorFlow and PyTorch). However, many of these implementations are not very stable. Occasionally people who tried to their implementation reportd failure on training due to the NaN value of gradients. Finally in 2018, [3] open-sourced their successful implementation of NTMs. After a series of careful experimentation. They argue that the stability and performance of NTMs highly depend on how NTMs initialize the memory contents. Results show that NTMs initializing memory contents with a small constant values converge on average 2 times faster, compared to other initialization schemes that they've tried (learned initialization and random initialization).

2.2.3 Differentiable Neural Computer

Differentiable Neural Computer [5] is an extension of the NTM. With all the properties of NTMs, DNC updated the addressing mechanisms Write more...

3 Approach

In this section, we will introduce four tasks that we used to test the capabilities of the NTMs. Our goal is to investigate the capabilities of NTMs. We not only try to replicate the results in [3], but also evaluate NTMs on a novel sorting task. The first three tasks, "Copy", "Repeat Copy", and "Associative Recall", are originally proposed in the original paper [4] and are tested in the implementation [3], where they provide a simple way to test to what extent the controller can learn to correctly interact with its memory.

For the fourth tasks, we wanted to test to what extent the NTMs is capable of learning an algorithm for an everyday programming problem: Sorting.

In conclusion, the tasks in our experiments are "Copy", "Repeat Copy", "Associative Recall" and "Sorting".

Occasionally, we found that the implementation paper [3] did not provide complete details for how the tasks were set up. In these cases, we consulted the code in the

[1] https://github.com/snowkylin/ntm
[2] https://github.com/chiggum/Neural-Turing-Machines
[3] https://github.com/yeoedward/Neural-Turing-Machine
[4] https://github.com/loudinthecloud/pytorch-ntm
[5] https://github.com/camigord/Neural-Turing-Machine
[6] https://github.com/snipsco/ntm-lasagne
[7] https://github.com/carpedm20/NTM-tensorflow

implementation directly [2]. We try to give all the details for reproducing the experiments in this paper.

For implementing the tasks, we used the existing NTM implementation announced in [3] and added our own code on top of this in order to make the testing more streamlined, clean and reproducible. However, since the code from [3] was written for TensorFlow version 1, we opted to use a version of the code that was ported to TensorFlow version 2 by another user [11].

3.1 Copy

The idea of the copy task is to present the Neural Turing Machine with a long sequence of information, which the network has to reproduce as precisely as possible. This directly tests the NTM's ability to store and reproduce information.

3.1.1 Network inputs

uld add some ample images, used in [2]

More precisely, an input to the network contains a sequence of 8-bit binary vectors plus a delimiter flag at the end of each vector. Thus, each vector contains nine elements, each of which is equal to 0 or 1. The last value is the delimiter flag and is always zero, the other values are sampled uniformly at random.

The length of the vector sequence, i.e. the amount of vectors in the input, is between 1 and 20 inclusive and is also sampled uniformly at random. Because the network cannot know how many input vectors are expected, we always include a final delimiter vector, which has nine elements, all of which are 1. In this way, the network can determine when the input sequence has ended, because only the final delimiter vector has its last bit set to 1.

We conclude that an input always consists of 2 to 21 vectors containing 9 bits, including the delimiter vector.

3.1.2 Target outputs

The target sequence, i.e. the desired output sequence, consists of exactly the input sequence, except for the last vector and without the last bit of each vector (i.e. the delimiters are removed). The target sequence therefore consists of 1 to 20 vectors each containing 8 bits.

During the output phase, the network is presented only with zeros as the input.

3.1.3 Batching

The training of the network was done in batches. This means that the calculation of the loss function includes multiple input/output pairs at the same time. In our case, the batch size was 32, meaning that the network was presented with 32 input sequences before its weights were updated. Note that within one batch, all sequences have the same length. We trained the network for $31,300$ batches, meaning that the network was presented with $1,001,600$ sequences of varying length.

3.1.4 Training

For training, we used a standard Adam optimizer with a learning rate of 0.001. We also used global norm clipping [6] for the gradients, with a threshold norm of 50.

For the loss function, we used binary cross-entropy for every bit in the output separately. These values are then summed and the result is divided by the batch size (32).

3.1.5 Validation

During training, we continually evaluated the performance of the network on a validation set. For this, the validation set was created before the training started and independently from the training data. The validation set contained 640 pairs of input and target sequences in the same format as for training (described above in subsubsection 3.1.1). Note that no batching was performed, i.e. the batch size for validation was 1. This means that the lengths of each of the 640 sequences are independent from each other (in contrast with the training data, where all sequences in a batch must have the same length).

The validation was performed every 200 batches. The error metric we used for validation was the average number of incorrect predictions per sequence. This means that for each sequence, the number of mistakes is counted. These mistake counts are then summed and divided by 640.

3.2 Repeat Copy

The repeat copy task is very similar to the copy task, but is made more challenging by the fact that the network has to repeat its output a fixed number of times. We will describe the differences to the copy task in more detail.

3.2.1 Network inputs

An input to the network consists of a sequence of 8-bit vectors plus two delimiter flags at the end of each vector. Therefore, each vector contains ten elements, where the first eight elements are sampled uniformly from $\{0, 1\}$ and the last two are always zero.

At the end of the sequence, we add a delimiter vector containing ten elements. This vector indicates to the network that the input sequence is now over, and also defines how often the network should repeat the input sequence. The first nine elements of the delimiter vector are always 1. The last value is a number n uniformly uniformly sampled from the set $\{0.1, 0.2, \ldots, 0.9, 1.0\}$. This number indicates to the network how often it should repeat its input. A value n means that the network should repeat $10n$ times. In other words, the number of repetitions is uniformly sampled from the set $\{1, \ldots, 10\}$.

One input sequence can contain between 1 and 10 input vectors, plus the delimiter vector. The sequence length is sampled uniformly at random. The final input dimensions are therefore between 2 and 11 vectors, each containing 10 elements.

3.2.2 Target outputs

As explained before, the target outputs are identical to the inputs without delimiter flags and without the delimiter vector, repeated $10n$ times. The target output dimension is therefore 1 to 10 vectors, each containing 8 elements.

3.3 Associative Recall

While the previous two tasks can be used to study if the NTM is able to interact correctly with its memory, the associative recall tasks tests whether it can learn associations between its input elements. This is done by showing a sequence of itemsto the network, and a query item which was part of the sequence. The task for the network is to output the item in the sequence that comes directly after the query item.

efine item

3.3.1 Network inputs

The input sequence contains a sequence of 6-bit vectors plus a delimiter flag at the end of each vector. The vectors in the sequence thus have seven elements, the first 6 of which are sampled uniformly at random from the set $\{0, 1\}$. The seventh and last value is always zero. We mentally group these vectors in groups of three contiguous vectors. We call a group of three contiguous vectors an item. The number of vectors is always divisible by three and between 6 and 18 – in fact, the

number of items is sampled uniformly at random from the set $\{2, 3, \ldots, 6\}$. Items do not overlap.

After the input vectors, the input sequence contains a delimiter vector consisting of seven elements, all of which are 1.

After this delimiter vector, the input sequence includes one of the items from the input vectors, called the query item. The query item will never be the last item from the input vectors.

After the three vectors from the query item, the input sequence includes another delimiter vector.

We conclude that all vectors in the input sequence have 7 elements, and the number of vectors in the sequence is always between $6+1+3+1 = 11$ and $18+1+3+1 = 23$

Better summarized in words, e.g. 6 to 18 input vectors, 3 query vectors, and 2 delimiter vectors

.

3.3.2 Target outputs

The target sequence always consist of three vectors, which all have 6 bits. These are the vectors from the item directly after the query item.

3.4 Sorting

Better emphasize more on this part. For the input, what is the length of each vector, and the number of vectors in a sequence? Will the vectors in a sequence are sorted? Are there same vectors in a sequence? For training and validation, how to calculate loss, and what error metric is used?

We now quickly describe a fourth task which was devised by us in order to test the NTM on an everyday problem: Sorting. The task is easily described: It works exactly like the copy task, except that the target outputs are sorted in lexicographical order. For example, given the input vectors $[1, 0, 0], [0, 1, 1], [1, 0, 1]$, the correct output would be $[0, 1, 1], [1, 0, 0], [1, 0, 1]$. We first sort by the first bit. Any ties are resolved by comparing the second bit. Any ties here are resolved by comparing the third bit, and so on.

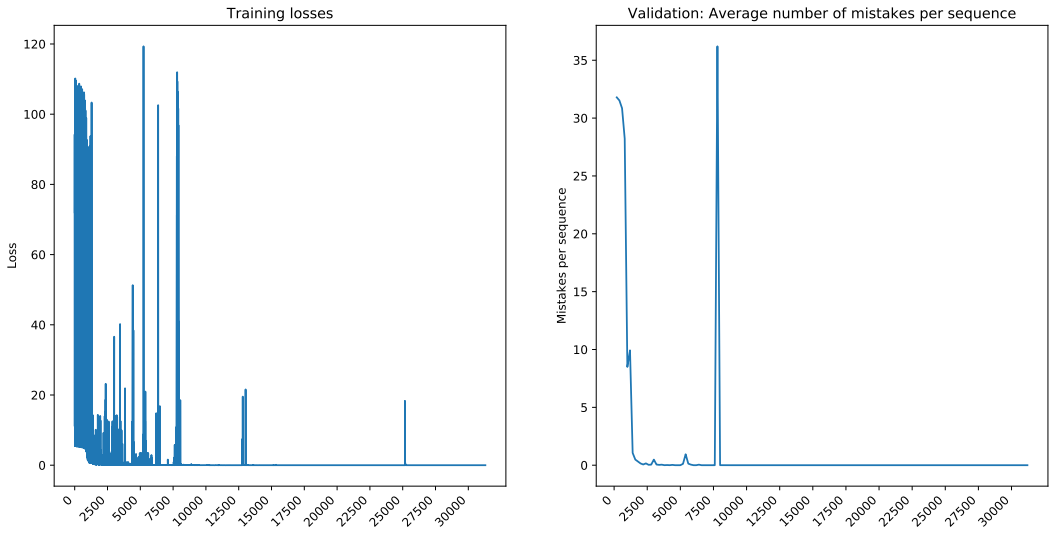

Figure 1: Losses and mistakes made for the Copy task

4 Results

For each of the four tasks, we recorded the loss and the validation metric over the course of the training. As explained before, the loss used was the summed binary cross-entropy of every output bit, divided by the batch size. For validation, we used a validation set of 640 sequences and calculated the average number of (bitwise) mistakes per sequence.

For the Copy task pictured in section 4, we can see that the Neural Turing Machine is able to learn to replicate its output. This is achieved without overfitting: The validation metric drops to zero about as quickly as the training loss. However, we observe that both training loss and validation metric sometimes shoot to high values for short amounts of time. We don't know if the same behaviour can be observed in the implementation in [3], because there, the median losses and validations of 10 runs where shown, which can hide such effects. It must be emphasized that these artifacts occur after a perfect training loss was achieved, so in a productive setting, they could be tolerable, because training would stop before they can occur. In fact this could be an example of "overtraining": Especially the validation metric shows the artifcat only after the task could already be considered solved.

The Repeat Copy task in Figure 4 and the Associative Recall task in Figure 4 show very similar results compared to the Copy task: The tasks are solved without overfitting, but some artifacts in losses and validation occur.

The Sorting task Figure 4 shows a different picture: Here the training takes much

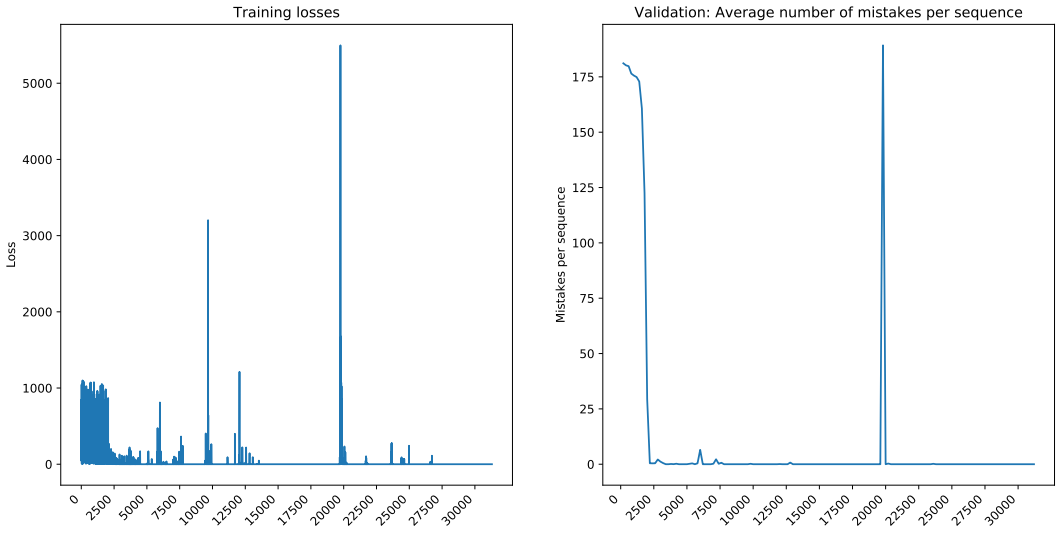

Figure 2: Losses and mistakes made for the Repeat Copy task

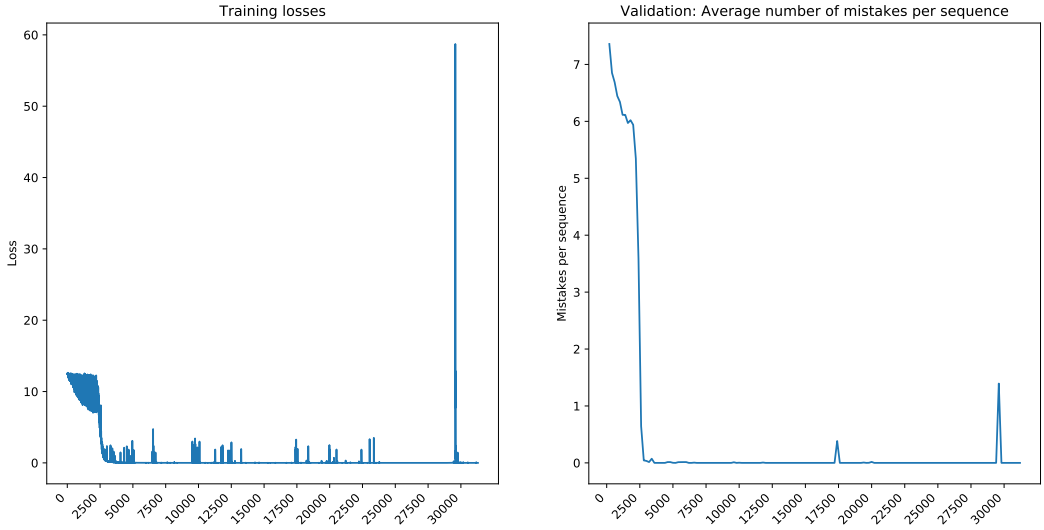

Figure 3: Losses and mistakes made for the Associative Recall task

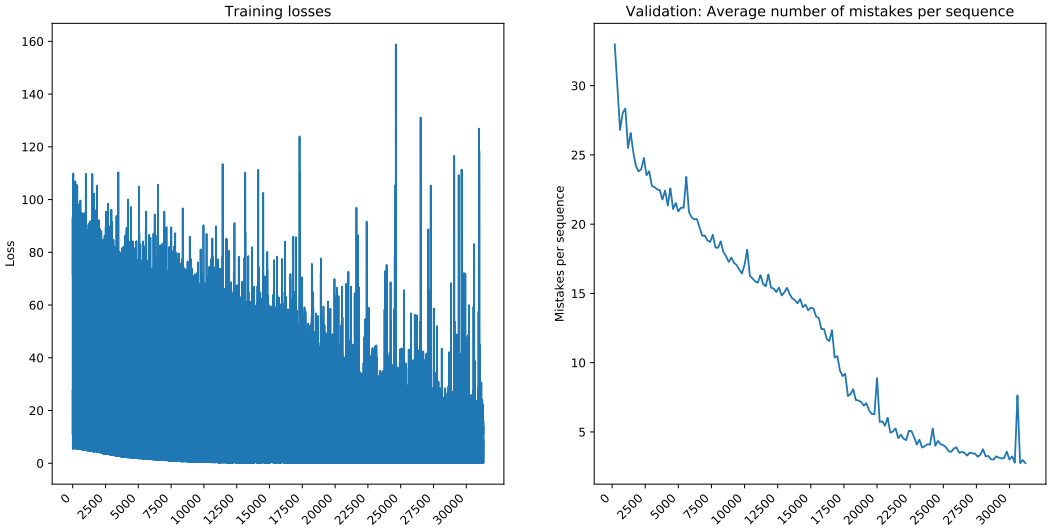

Figure 4: Losses and mistakes made for the Sorting task

longer and the task is not perfectly solved at the end. However, it is nearly solved: The NTM reaches a validation of an average 2.75 mistakes per sequence. Since a sequence consists of 1 to 20 vectors, each with 8 bit, this is an average mistake rate of $\frac{2.75}{10.5 \cdot 8} \approx 3.3\%$. It is possible that the performance could be further improved through longer training. Furthermore, we performed not finetuning of parameters at all – we used the default parameters from the implementation in [3].

It is interesting to note that not spikes or artifacts occur for the validation metric recorded for the Sorting task. This might be because no "overtraining" has occured here – the task of sorting a sequence of vectors seems to be more difficult than the other tasks, and takes much longer to train.

5 Discussion

We answer our research question based on the results.

Compared to the graphs in [3], note that we show a training time that is twice as long and we labelled the x-axis differently: We use the batch index for the label, while [3] uses the sequence number, which is just the batch index times the batch size (32).

We check if we were able to replicate the results from [3]. Possibly we will also check the performance of the DNC (from [5]) compared to the NTM.

We will discuss reasons for the performance of the NTM on the tasks.

6 Conclusion

We discuss the contribution and limitation of our work, and further direction of this field.

References

[1] Gaurav Arora, Afshin Rahimi, and Timothy Baldwin. Does an LSTM forget more than a CNN? an empirical study of catastrophic forgetting in NLP. In *Proceedings of the The 17th Annual Workshop of the Australasian Language Technology Association*, pages 77–86, Sydney, Australia, 4–6 December 2019. Australasian Language Technology Association.

[2] Mark Collier. NeuralTuringMachine. `https://github.com/MarkPKCollier/NeuralTuringMachine`, 2018.

[3] Mark Collier and Joeran Beel. Implementing neural turing machines. *International Conference on Artificial Neural Networks, ICANN.*, 2018.

[4] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014.

[5] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, October 2016.

[6] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[7] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449, 1992.

[8] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[9] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society*, 2(1):544–546, 1938.

[10] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[11] wchen342. NeuralTuringMachine - TF 2.0. `https://github.com/wchen342/NeuralTuringMachine`, 2019.