The Neural Turing Machine: Introduction, Implementation and Experiments

Florian Unger

August 21, 2017

In the following we try to roughly explain the idea behind the Neural Turing Machine as described in [1]. As that paper did not provide a lot of details necessary for implementing it and also lacks an implementation itself, this work aims to fill the gap by implementing the Neural Turing Machine as a Keras recurrent layer. In the end we test our implementation on a simple copy task and describe our findings.

Whenever text appears in `this_style`, it refers to functions or parameters in the code, which can be found at GitHub at `https://github.com/flomlo/ntm_keras/`. The source code is extensivly documented and can be regarded as additional reading matter should questions about implementation details arise.

1 Introduction

The core idea of the Neural Turing Machine is to allow a neural net access to actual computer RAM, as such is often available in abundance and using long short term memory (LSTMs) for memory intensive task quickly becomes quite a burden, both in numerical and computational complexity.

This is achieved in resemblance to a classical Turing Machine: The controller in a classical Turing Machine is in a memory/state-loop: At every step the Turing Machine decides what to write on the memory and where to move its head based on its state and what it has read from the memory this step. It is also one of the remarkable features of the classical Turing Machine that the controller does not know about the size of the memory (in comparison to, e.g., register machines), as the memory addressing mechanism is only based on head movement on the famous infinite long tape.

Those two ideas are the key features of the Neural Turing Machine, hence the name. The controller becomes a differentiable function reading memory input and emitting memory output and adressing data. This is modelled via a neural net. Now the difficulty lies in differentiable read and write access to the memory, together with differentiable adressing mechanisms, so that everything can be trained via backpropagation through time.

1.1 Components and hyperparameters

As we have implemented the Neural Turing Machine as a Keras recurreny layer, it accepts input of shape `input_dim` $= d$ and emits output of dimension `output_dim` $= d'$ in every step[1].

[1]Via setting `return_sequences=False` the output shape could consist of a single vector instead of a sequence. Unfortunately, this is currently broken.

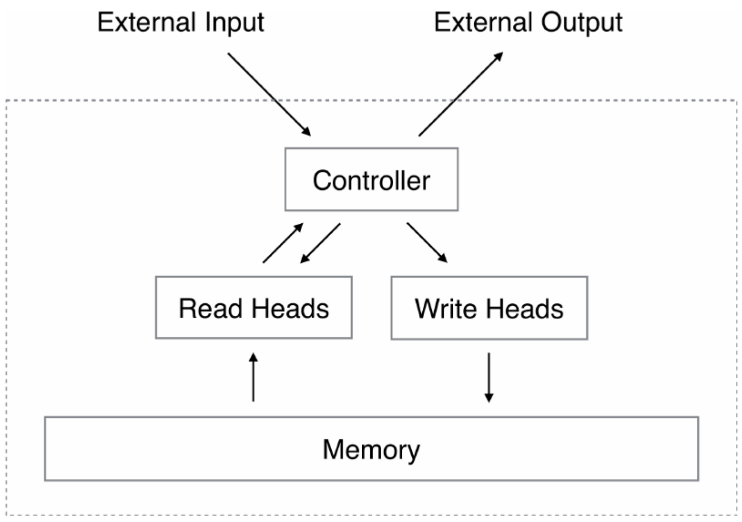

Figure 1: A rough graphical representation of the Neural Turing Machine as shown in [1]. Everything inside the dashed line is considered the Neural Turing Machine.

Initialising a Neural Turing Machine also requires some hyperparameters, namely the shape of the memory given by depth $m =$ `m_depth` and the width $n =$ `n_slots`, the range of head movement $r =$ `shift_range` and the number of read and write heads[2] h_r respectivly h_w. For technical reasons[3] the `batch_size` must be passed, too.

A Neural Turing Machine now consists of a `controller`, memory `M` and (logically) read and write heads, implemented through functions for reading (`_read_from_memory`), writing (`_write_to_memory`) and the adressing mechanism (`_get_weight_vector`). All this is now explained in further detail.

1.2 Controller

The controller is the trainable part of the Neural Turing Machine which is responsible for both computation and memory management. It can be of any kind, e.g. a single layer of LSTM or many dense layers. This is influenced by the general architecture in mind: One might think of either a more classical von-Neumann architecture with a big complex controller and strict separate memory, or more of a non-traditional mixture of memory and computation as is the case with recurrent neural networks. In the latter case one only takes a very small controller, ideally (for internal cache enabling computations requiring more input than what can be read via the expensive write heads) one or two layers of LSTM.

The controller shape is as follows: The input shape depends on the layer input from the outside and the input read from the memory (calculated in the last step), resulting in

$$\texttt{controller\_input\_shape} = d + h_r * m.$$

The output shape depends on the layer output dimension d', the addressing data for reading $(m + r + 3)$ and the adressing data for writing along with actual content to write $(m + r + 3 + 2 * m)$. All in all the controller output shape results in

$$\texttt{controller\_output\_shape} = d' + (h_r + h_w) * (m + r + 3) + h_w * 2 * m.$$

For convenience, there is a helper function `controller_input_output_shape` calculating that.

[2]Currently it is fixed to a single read head and a single write head.

[3]A stateful controller needs to save its state in a tensor, which has to be initialised at build time.

In our code, a controller is passed to the Neural Turing Machine as a parameter `controller_model` (if `None` is passed, a single dense layer is used). Stateful controllers, e.g. recurrent ones, have to carry around their own state.[4] It is also assumed that the output of the controller was activated via sigmoid, or at least lies in $(0, 1) \subseteq \mathbb{R}$ for each node and is about 0.5 when it is not yet trained.

1.3 Memory adressing mechanism

Note that the controller does not know about the width n of the memory.[5] This is due to the memory addressing design. One can read about it in more detail in section 3.3 in [1], therefore we concentrate on a rough overview and implementation difficulties. At every timestep, in contrast to the classical Turing Machine, the Neural Turing Machine interacts with the whole memory matrix, weighted via a probability vector $w_t \in \mathbb{R}^n$ for every head. This weight vector is calculated in dependence of the controller output, the memory and the last weight vector w_{t-1} (see Figure 2 of [1]). This is done to allow for both content based addressing and location based addressing and works as follows:

The controller emits for each head a key vector $k \in \mathbb{R}^m$, three scalars $\beta, g, \gamma \in \mathbb{R}$ and a probability vector $s \in \mathbb{R}^r$ called the shift vector. Additionally, one has the afore mentioned old weight vector w_{t-1} and the memory $M_{t-1} \in \mathbb{R}^{m \times n}$ from the last step.

That data is transformed into the current weight vector w_t by first calculating a cosine similiary search of the key vector k on the old memory M_{t-1} and softmaxing the result. Then the resulting probabilty vector is combined via a convex combination parameterised by g with the old location vector w_{t-1}. Now we may shift this intermediary result as far as the shift range r allows it, calculated with s. After all this treatment the resulting probability vector has to be sharpened again using the parameter γ.

While in theory this is a perfectly fine process, as every step guarantees that a probability vector is produced and thus also the composition. In practice however it is numerically quite problematic. The cosine simililarity produces `NaN` whenever a null vector is part of the calculation and can be numerically unstable. The resharpening in the end can lead to `Inf/Inf` if γ is too big. This is combatted in our implementation by modyfing cosine similary by adding ϵ in case of zero input[6], and taking care that both inputs of cosine similarity are not initialised zero. The problems with γ are, for now, combatted via simple clipping.[7]

1.4 Memory interaction

Reading is pretty straightforward: We use the aquired weight vector $w_r \in \mathbb{R}^n$ and multiply it with the memory.

For writing, the controller additionally emits an erase vector $e \in (0, 1)^m$, (the range being important[8]), and deletes by multiplying the memory M with $1 - w_w * e$. After

[4]In Keras recurrent layers, set `stateful=True` to enable this behaviour.

[5]This is one of the coolest features of them all, truly earning the model the right to be called Neural Turing Machine and not just Neural Register Machine. One can take the weights of a trained model and just implement them into a model with bigger (wider) memory! This was tested on fully trained models and works like a charm. In theory, this could also been done one-the-fly during training, but it is difficult to determine when this is necessary.

[6]There is probably a more perfect solution out there.

[7]Also looking for a better solution.

[8]A bug where the erase vector could become negative bug creeped around for a month, invalidating the results of every experiment done in that time and radically improving the quality of the model more than anything else after being fixed.

erasing the add vector $a \in \mathbb{R}^m$ [9] is added after being weighted with w_r.

The whole procedure reminds of LSTMs and has the same purpose.

2 Experiments

2.1 Copy

Here we try to replicate the copy experiment found in [1]. The task consists of copying a sequence of variable length (but between 5 and 100), where the input sequence consists of 8 bits information and 2 bits flag ($d = 10$) (as seen in Figure 6 of [1]) and the output should consist of the copied sequence without the flags ($d' = 8$). Testing is done at sequence lengths 5,10,20,40,80, while training is only done at random sequence lengths between 5 and 20. Training is done over 1000 epochs consisting of 10 steps per epoch with a batchsize of 100, amounting to 1 million sequences in total.

The performance criteria is bitwise accuracy on the part where the copied sequence is to be expected. It is also interesting to look at generalisation: Could a model copy for sequence parts (much) greater than the ones trained for?

Besides the corrupted accuracy reported in TensorBoard[10], binary accuracy on the sequence is calculated and reported for each sequence length before and after training, too.

The loss is calculated via binary crossentropy, weighted 1 at the position where we expect the sequence and 0.001 everywhere else, which is also padded with 0.5 in order to avoid any bias. All the networks performed worse if focusing only on the important part. The optimizer in all experiments was Adams with gradient clipping and a learning rate of 5×10^{-4}. If the Neural Turing Machine was used, $m = 20$ and $n = 128$. Each experiment was repeated as least ten times. The experiments were conducted on a NVIDIA GeForce 1050 Ti using `float32`.[11]

2.1.1 Stateless FeedForwardNetwork

As a quick sanity check we tested a simple FFW without a hidden layer. As to be expected, it was not able to learn anything, as it doesnt have state. At least it was quick.

2.1.2 LSTM

We tested a 3-Layer LSTM with 256 units each layer (as given by Table 3 in [1]) using Keras default values for initialisation. Each run took about 70 minutes.

In comparison to experiments with the Neural Turing Machine it must be noted that every run performed almost the same, i.e. loss and accuracy were tighly grouped together. The accuracies for the respective sequence lengths were 5: 1, 10: ca. 0.999, 20: ca. 0.95, 40: ca. 0.50, 80: ca. 0.50. This shows that it works quite well, but not perfect, with decreasing accuracy with increasing sequence length. It also fails completely at generalising.

[9]It might be tempting to change the output to something strictly positive, as that would make zero vectors in the memory impossible thus avoiding numerical errors at cosine similiarity. However, the models trained with that performed (on average) much worse and it drastically increased the variance of model performance.

[10]Which unfortunately maxes out at ≈ 0.3, as accuracy measurement in Keras can not be weighted. It is however still good for a quick look: perfect would be around 0.3, complete failure at 0.15, anything outside that range indicates a `NaN`.

[11]Here we would like to thank the computer science department of the Friedrich Alexander University Erlangen-Nuernberg, which graciously donated several hundred computing hours on 5 GPUs of aforementioned type without even being asked, never even complaining about it.

2.1.3 Neural Turing Machine with 1-Layer FFW Controller

We build a 2-Head Neural Turing Machine with a dense single layer controller. Each run took about 100 minutes. After finding even the most hidden bugs in the implementation, this model performed, with some exceptions, very well: It converged quickly, with very sharp and sudden declines in loss and often reached perfect scores for every sequence length.[12] In about 30 experiments, about 20 reached that level. The rest had each interesting behavior: Quite a few performed better on longer sequences than on short ones, a closer inspection[13] revealed that this was due to problems at the beginning of the sequence.[14] Others performed like a better LSTM, performing perfect on training sequence length, but a lot worse in longer sequences.[15] There were even some which performed with an accuracy of about 0.7 for every possible sequence length, so far remaining a mystery. Besides one model which crashed to a `NaN`, the most worrying were those which achieved perfect scores for several hundred epochs, only to become completely useless (but without numerical errors!) in just one epoch, which happened two times. So far no explanation has been found, it is not clear if this is due to the theoretical model or due to the implementation.[16]

Note that all these runs were performed with the same initial weights, the differences are due to the nondeterministic nature of the stochastic gradient descent.

2.1.4 Neural Turing Machine with 1-Layer LSTM Controller

We build a Neural Turing Machine with a 1-Layer LSTM controller, initialised with Keras default values (but with sigmoid activation instead of tanh). Training time was also about 100 minutes. As the copy task only uses unary operations, the models performed very much like their stateless brothers. At least the implementation seems to work.

2.1.5 Neural Turing Machine with 2-Layer FFW Controller

We build a Neural Turing Machine with a 2-Layer dense controller, which had a hidden layer size of 150. Training time also was about 100 minutes.[17] The models performed almost the same as their smaller brothers.

3 Conclusion

All in all the Neural Turing Machine is a neat idea, worth further research. The copy experiment shows that despite its fickleness, it is fundamentally able to reasonable use computer memory, even for something advanced like for-loop variables, as shown in chapter

[12]The only limit was the width of the memory n. In one case we even replaced the memory with a larger memory with $n = 512$, while retaining the trained weights. This modified model now worked for sequences up to this length flawlessly.

[13]You can do that too! Just enable `verbose = True` in `lengthy_test` and suddenly the network is evaluated after every training epoch, but not only that, it also shows bitwise accuracy and accuracy for every single timestep.

[14]This indicates problems on edge cases of index arithmetik. If that is not human-like artificial intelligence, what is?

[15]A possible explanation could be very unsharp weight vectors, generating deeper memory at the cost of width.

[16]One wild hypothesis would be that this is due to an insufficient loss function: If the output of the model is shifted by just one timestep, the loss function would rate it as complete garbish.

[17]A test with a really big 5-Layer controller still took almost the same time. This indicates that the (computational) performance issues compared with LSTM might lie entirely in the implemenation of the memory loop. As this was not a priority so far, it is quite possible that is a noticeable potential for performance optimisations.

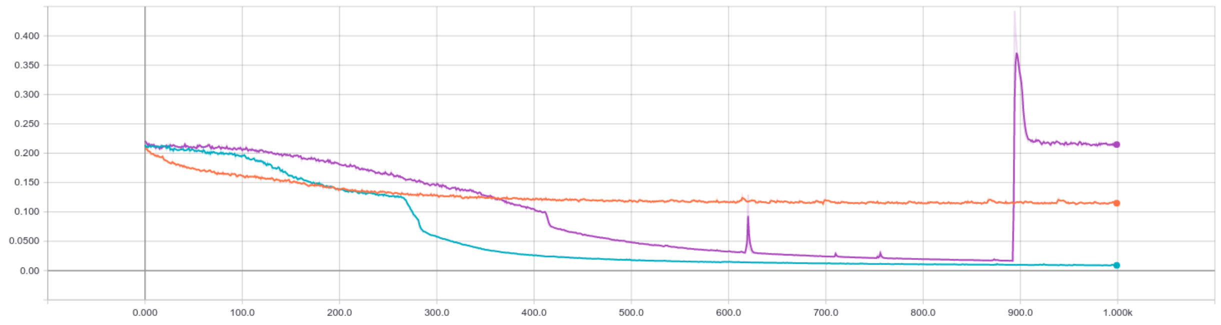

Figure 2: A plot of the loss. The orange line describes a LSTM, the cyan and purple line Neural Turing Machines with a single layer dense controller. Characterstic are the very sharp drops in the value of the loss function and the, in case of cyan, very very low final loss. The purple line however belongs to a model which mysteriously lost all its knowledge in a single step.

4.2 in [1]. The generalisation ability together with the independence of the controller of the memory width n has the potential to make training much quicker if properly used. Note that content based addressing system was not utilised at all[18] in our copy test, but chapter 4.3 in [1] claims it works well, too.

There is reason to believe that the idea still has issues. In the spiritual successor, the Differential Neural computer described in [2] a few are named. For example the inability to perform effective high level memory management and its tendency to overywrite important things, or the low-level way of implementing the idea of temporal memory management via location shifts.

There is also the issue of training the Neural Turing Machine with methods other than supervised learning. There are papers working on that, e.g. [3], but it seems to be incredible difficult.

But nonetheless, by having a backend neutral implementation as a Keras recurrent layer more people might discover applications for it. A small model with an LSTM controller could be a very viable replacement option for a pure LSTM layer.

References

[1] Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural Turing Machines”. In: *CoRR* abs/1410.5401 (2014). URL: `http://arxiv.org/abs/1410.5401`.

[2] Alex Graves et al. “Hybrid computing using a neural network with dynamic external memory”. In: *Nature* 538.7626 (2016), pp. 471–476.

[3] Wojciech Zaremba and Ilya Sutskever. “Reinforcement learning neural turing machines”. In: *arXiv preprint arXiv:1505.00521* 419 (2015).

[18] This can actually be visualised: Using `view_weights.py` on a folder full of saved model weights as generated by an appropriate Keras callback generates an interactive slider through the controller weights represented as 30×100 grayscale pictures. This is quite relaxing to look at.