
Update 2019-05-26: Google has integrated our NTM implementation into the official TensorFlow release. For more details read here: <https://www.scss.tcd.ie/joeran.beel/blog/2019/05/25/google-integrates-our-neural-turing-machine-implementation-in-tensorflow/>

For a description of our implementation and experimental results please see the pre-print of our paper which will appear as a conference paper at ICANN 2018: <https://arxiv.org/abs/1807.08518>

Our key contribution is not to implement a Neural Turing Machine in code but to make training stable and reliable. We do not observe the slow learning or gradients becoming NaN that other implementations have reported.

You can cite the paper as follows:

```
1 @article{collierbeel2018ntms, title={Implementing Neural Turing
  Machines,
2 author={Collier, Mark and Beel, Joeran},
3 journal={International Conference on Artificial Neural Networks, ICANN
  .}, year={2018}}
```

This work was done with Joeran Beel Ussher Assistant Professor in Intelligent Systems at the Adapt Centre, Trinity College Dublin as part of my undergraduate thesis at Trinity College Dublin.

Neural Turing Machine

This repository contains a stable, successful Tensorflow implementation of a Neural Turing Machine which has been tested on the Copy, Repeat Copy and Associative Recall tasks from the original paper.

Usage

```
1 from ntm import NTMCell
2
3 cell = NTMCell(num_controller_layers, num_controller_units,
4               num_memory_locations, memory_size,
5               num_read_heads, num_write_heads, shift_range=3, output_dim=
6               num_bits_per_output_vector,
7               clip_value=clip_controller_output_to_value)
8
9 outputs, _ = tf.nn.dynamic_rnn(
10    cell=cell,
11    inputs=inputs,
12    time_major=False)
```

The implementation is derived from <https://github.com/snowkylin/ntm>, another open source NTM implementation. We make small but meaningful changes to the linked code that have a large effect on making our implementation more reliable to train and faster to converge as well as being easier to integrate with Tensorflow. Our contribution is: - We compare three different memory initialization schemes and find that initializing the memory contents of a Neural Turing Machine to small constant values works much better than random initialization or backpropagating through memory initialization. - We clip the outputs from the NTM controller into a range, which helps with optimization difficulties. - The NTMCell implements the Tensorflow RNNCell interface so can be used directly with `tf.nn.dynamic_rnn`, etc. - We never see loss go to NaN as some other implementations report. - We implement 3 of the 5 tasks from the NTM paper. We run many experiments and report convergence speed and generalization performance for our implementation, compared to an LSTM, a DNC and for 3 memory contents initialization schemes.

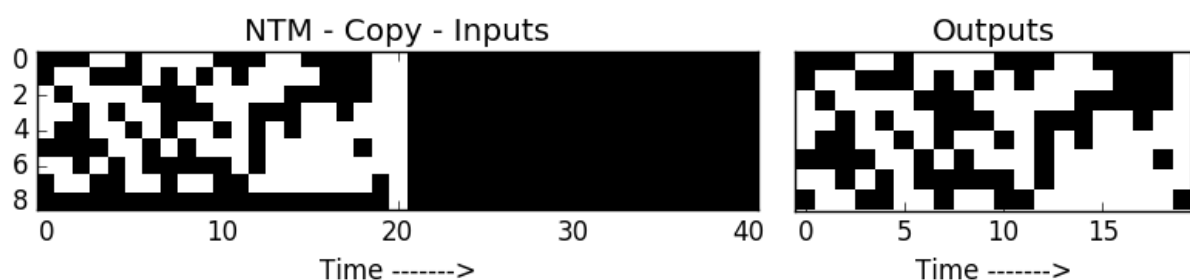
Sample Outputs

Below are some sample outputs on the Copy and Associative Recall tasks. We replicated the hyperparameters from the original paper for the 2 tasks:

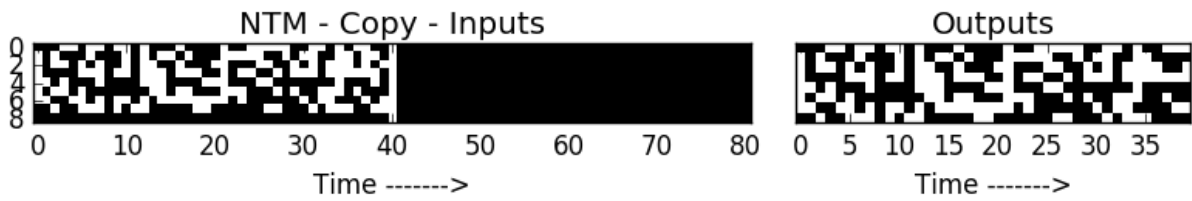
- Memory Size: 128 X 20
- Controller: LSTM - 100 units
- Optimizer: RMSProp - learning rate = 10^{-4}

The Copy task network was trained on sequences of length sampled from $\text{Uniform}(1,20)$ with 8-dimensional random bit vectors. The Associative Recall task network was trained on sequences with the number of items sampled from $\text{Uniform}(2,6)$ each item consisted of 3 6-dimensional random bit vectors.

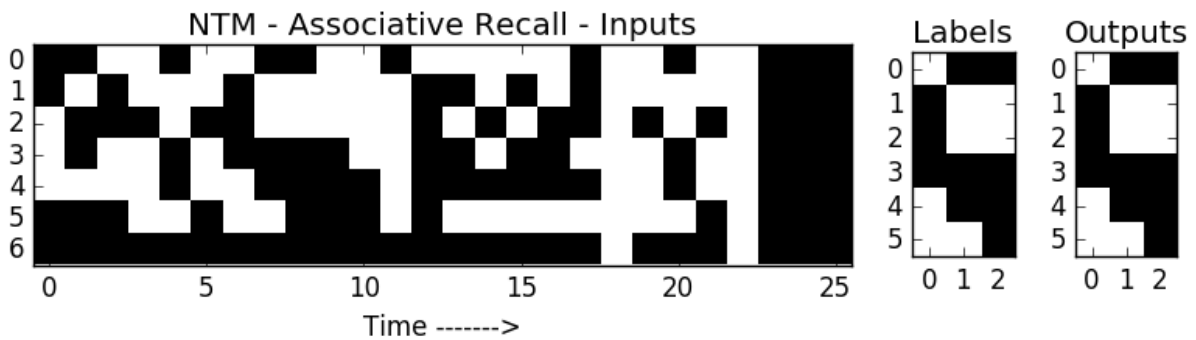
Example performance of NTM on Copy task with sequence length = 20 (output is perfect):



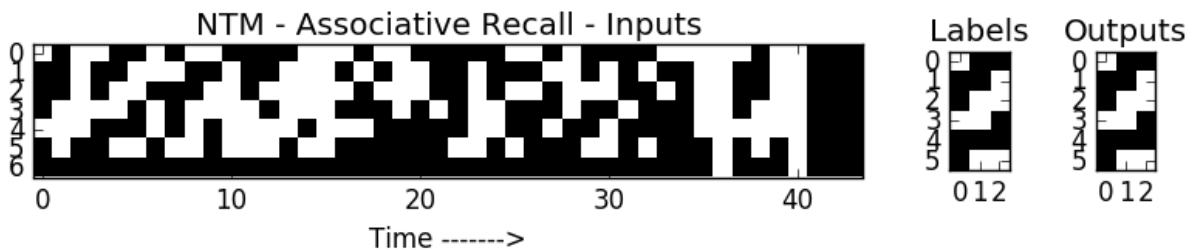
Example performance of NTM on Copy task with sequence length = 40 (network only trained on sequences of length up to 20 - performance degrades on example after 36th input):



Example performance of NTM on Associative Recall task with 6 items (output is perfect):



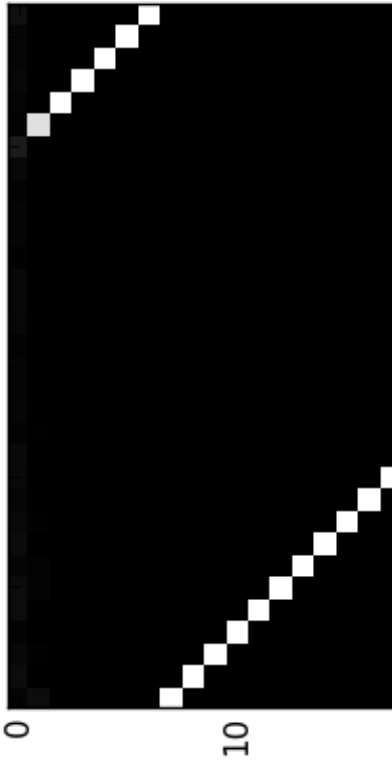
Example performance of NTM on Associative Recall task with 12 items (despite only being trained on sequences of up to 6 items to network generalizes perfectly to 12 items):



In order to interpret how the NTM used its external memory we trained a network with 32 memory locations on the Copy task and graphed the read and write head address locations over time.

As you can see from the below graphs, the network first writes the sequence to memory and then reads it back in the same order it wrote it to memory. This uses both the content and location based addressing capabilities of the NTM. The pattern of writes followed by reads is what we would expect of a reasonable solution to the Copy task.

Write He



Write head locations of NTM with 32 memory locations trained on Copy task:

Read Head



Read head locations of NTM with 32 memory locations trained on Copy task:

Further results on memory initialization comparison and learning curves to come...