Implementing a Neural Turing Machine

Jared Nielsen[1]

***Abstract*— Auxiliary memory models enable long-term recall, improved one-shot learning, and the potential to learn algorithms. The Neural Turing Machine has shown success at copying data sequences, repeat copying, and even a rudimentary sort algorithm. We implement it and seek to replicate the paper's results.**

I. INTRODUCTION

The Neural Turing Machine (NTM) consists of three components: the controller, the read/write heads, and the memory. It can be used as a stand-in replacement for an RNN module, since its general form merely has input, state, and output. However, the NTM is notoriously difficult to train. The original paper was released without code in 2014, and as recent as 2018, a paper called "Implementing Neural Turing Machines" was accepted to ICANN. The fact that it took four years to reproduce results and achieve stable training shows that this is a difficult problem. Yet further work such as Memory-Augmented Neural Networks (MANN) and the Differentiable Neural Computer (DNC) show promise, so it's best to start from the foundation.

II. PROJECT OVERVIEW

A. Goals

Since the NTM does not achieve state-of-the-art results on any significant tasks yet, the scope of this project is exploration. The NTM is a fantastic baseline for further development of auxiliary memory models, so I wrote one from scratch and implemented an existing open-source version.

B. Results

- Created feedforward controller architecture.
- Implemented differentiable memory-addressing scheme.
- Developed copy-task dataset with variable-length sequences.
- Made LSTM baseline for comparison.
- Visualized memory contents, read vectors, read/write weightings.
- Tested the effect of small vs large batch sizes on the NTM.
- Merged best-practice techniques into an existing but unstable PyTorch framework.
- Achieved 100% accuracy on the copy task for length-20 sequences.

[1]Jared is at Brigham Young University in Provo, Utah, jaredtnielsen at gmail.com

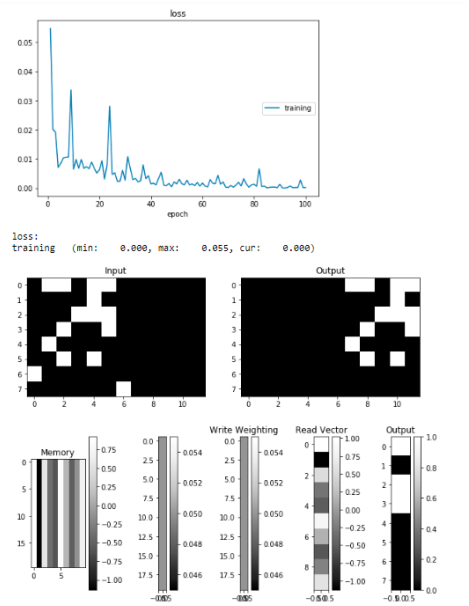


Fig. 1. Successful copy task. On the input (left middle), the first column represents the start code and the final non-blank column is the end code. The NTM reads in the input, outputting nothing, until it reaches the end code. Then it must output the entire sequence it has seen.

III. EXPERIMENTS

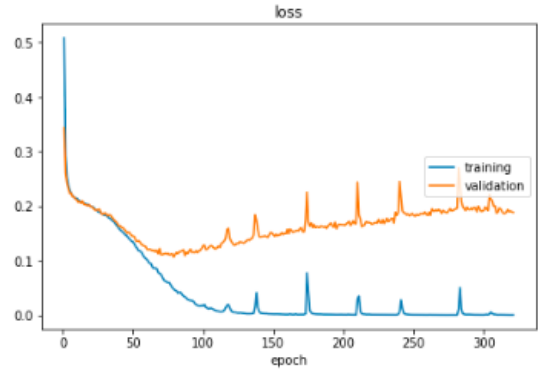
First we train an LSTM on the copy task as a baseline.

A lot of work was put into creating extensible datasets as I continue to explore auxiliary memory models. Since we're using 8-bit vectors (and two of those bits are start/end codes, so they're actually 6-bit vectors), I was worried that there might be overlap between the train set and test set. Dealing with small datasets, there's a definite danger of simply memorizing the training data. However, even though there are only $2^6 = 64$ unique 6-bit vectors, a sequence of 5 vectors concatenated will have $64^5 = 1.07$ *billion* unique possibilities.

A. LSTM Baseline

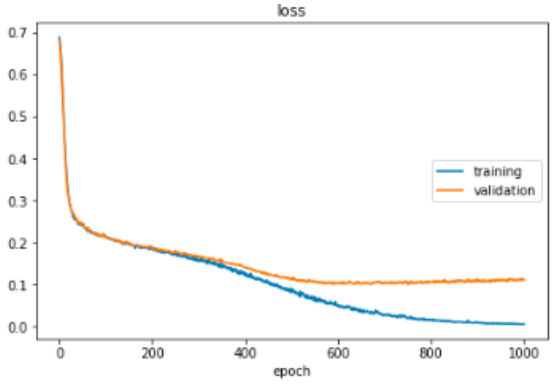
A surprising amount of time was spent optimizing the LSTM. Since my natural language processing experience is limited, it was a wonderful opportunity to implement padding, packed sequences, and batching in conjunction with a handmade dataset.

Most NTM implementations have a default batch size of 1. Is this because it leads to better generalization, or is it simply the accepted default? I compared an LSTM with batch size 1 to an LSTM with batch size 32 on the copy task. The




```
loss:
training   (min:    0.000, max:    0.508, cur:    0.000)
validation (min:    0.106, max:    0.344, cur:    0.188)
```

Fig. 2. LSTM with batch size 1. Faster convergence, easily overfits.




```
loss:
training   (min:    0.004, max:    0.688, cur:    0.005)
validation (min:    0.099, max:    0.679, cur:    0.110)
```

Fig. 3. LSTM with batch size 32. Slower convergence, better test performance.

larger batch size took longer to train but had better validation performance.

B. NTM

I successfully implemented the overall network architecture and differentiable memory addressing scheme. Because of the model complexity and number of hyperparameters to tune (memory size, memory width, number of read/write heads, head architecture, etc.), I did not get my homemade NTM working.

I used the memory initialization scheme from `https://github.com/MarkPKCollier/NeuralTuringMachine` combined with the PyTorch NTM implementation at `https://github.com/loudinthecloud/pytorch-ntm`. These achieved much improved results.

IV. CONCLUSION

Neural Turing Machines are hard to train! Constant memory initialization as opposed to a learned memory bias stabilize training. Using a batch size larger than 1 is important

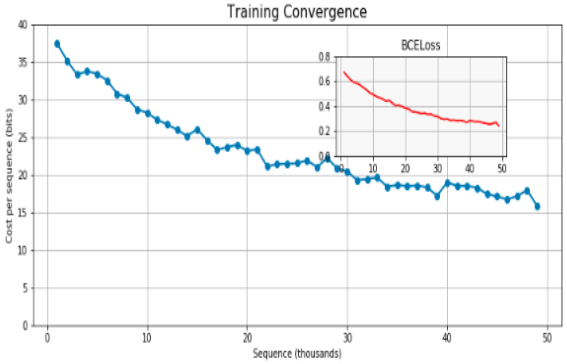


Fig. 4. Neural Turing Machine cost per iteration on copy task.

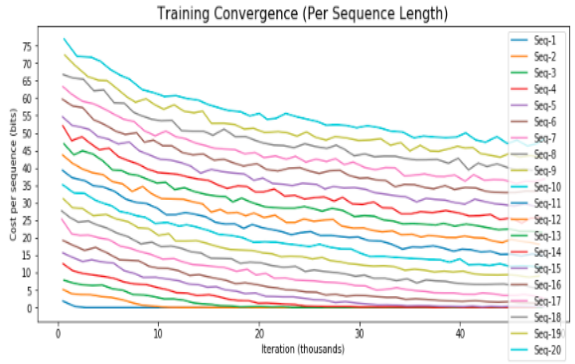


Fig. 5. Neural Turing machine generalization capability. Longer sequences are harder to copy, but we see improvement across the board through training.

for generalization abilities. My model would often learn to output a uniform distribution over read/write weightings, somehow memorizing everything in the controller. This can be seen in Figure 1. I'm not sure why this is. There are many moving parts in the NTM, so investigating it is beyond the scope of this project, but it's promising future work!

For further work, I would like to investigate curriculum learning. Does starting with shorter sequences and then later training on longer sequences offer an improvement over pure randomization?

I will also explore MANNs, which are an extension of NTMs and proposed solution for the problem of one-shot learning. The DNC also seems promising.

V. CODE

To see the LSTM baseline, dataset creation, and differentiable memory addressing, visit `https://github.com/jarednielsen/research/blob/master/ntm/Neural%20Turing%20Machine.ipynb`.