IFT6135 Assignment3, Practical part

Xiao Fan (20086722)
Zhibin Lu (20091078)

April 2, 2018

Neural Turing Machines

1. Filling in the Gaps

(a)

softplus$(x) = \log(1 + e^x) > 0$, so key strength $\beta \leftarrow$ softplus(β) is in range $(0, \infty)$.

sigmoid$(x) = \frac{1}{1+e^x} \in (0, 1)$, so interpolation gate $g \leftarrow$ sigmoid(g) is constraint to $(0, 1)$.

Shift weighting s is a normalized distribution, so $s \leftarrow$ softmax(s).

Sharpen weighting scalar $\gamma \leftarrow 1 +$ softplus(γ) is in the range $(1, \infty)$.

Erase vector $e \leftarrow$ sigmoid(e) is in the range $(0, 1)$.

(b)

Figure 1 shows the flow diagram of different parts in the Neural Turning Machine. Dependencies of variables are clearly demonstrated in the graph.

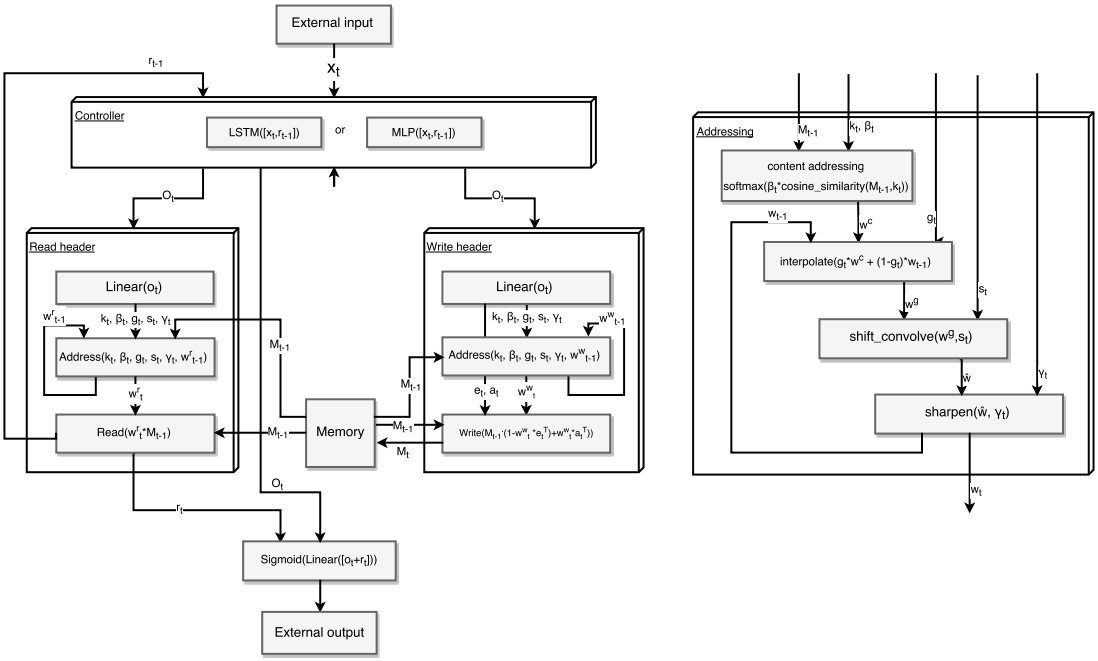

Figure 1: Flow diagram of the Neural Turning Machine (left) and the addressing mechanism (right).

2. Implement the Neural Turing Machine

The implementation is based on algorithm introduced in [1] and we used part of the code source form [2] in our project.

(a) Number of parameters

The number of parameters of LSTM model is 45208.

The number of parameters of Feedforward-NTM model is 13260.

The number of parameters of LSTM-NTM model is 62860.

(b) Hyper-parameter search and learning curves

For all the three models, we used algorithm *RMSProp* for training with momentum of 0.9. We used mini-batch size of 20 and we trained for 50,000 epochs. For Feedforward-NTM and LSTM-NTM model, we used learning rate of 10^{-4} and memory size of 128×20. For LSTM model, we used learning rate of 3×10^{-5}. For all the three models, we used one layer with a dimension of 100.

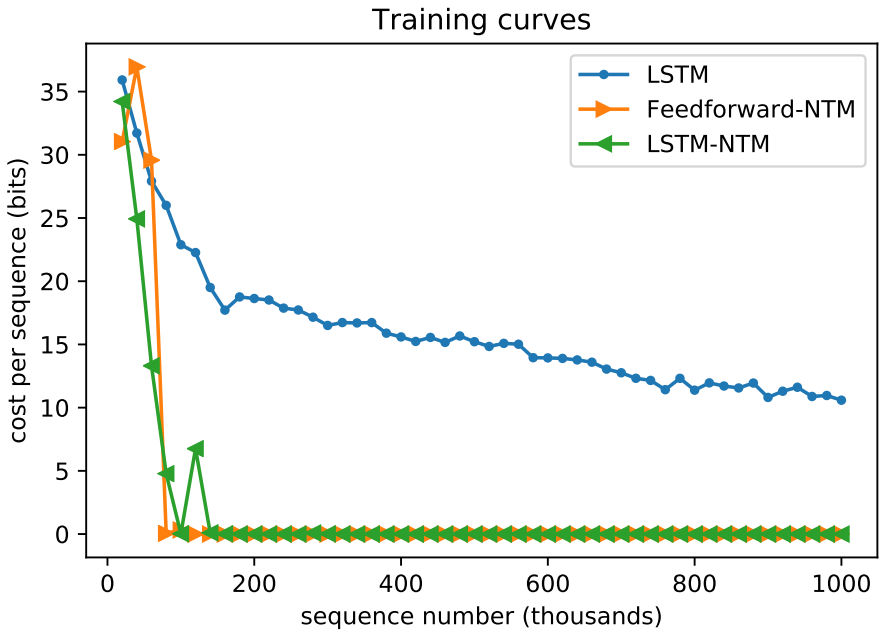

Figure 2: Learning curves for three models : Feedforward-NTM, LSTM-NTM and LSTM.

As seen in figure 2, loss converges for all the three models. Cost of NTM models converge much faster and achieves 0 around $100k$ sequences while LSTM still has a large loss after 1M sequences. NTM models with memory has much better performance than LSTM model. Feedforward-NTM and LSTM-NTM don't have much difference in convergence speed. LSTM-NTM convergences a little faster in the beginning but has a small fluctuation.

(c) Generalization to longer sequences

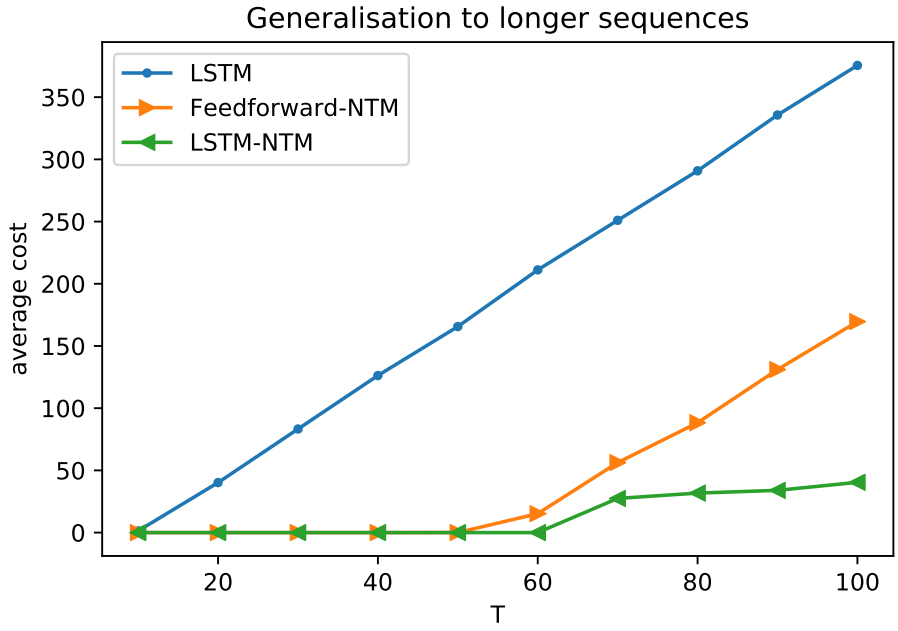

Figure 3: Average loss as a function of T for three models : Feedforward-NTM, LSTM-NTM and LSTM.

NTM models are expected to have better generalization to longer sequences than vanilla LSTM because they have a memory bank that is able to memorize information for longer periods. As we see in the figure 3, both NTM models have lower loss when T is larger than 20. They are able to correctly copy sequences that is no longer than 50 and have little errors for longer sequences. Compared feedforward-NTM and LSTM-NTM, LSTM-NTM model has even better generalization, especially for sequences longer than 50 because LSTM controller in NTM gives the model additional memory capacity. For vanilla LSTM model, the average loss is nearly proportional to the length of sequence, the model has really bad performance in the copy task.

(d) Visualizing the read and write heads/attention

Figure 4 shows the write and read attention during the copy task. The outputs successfully copy the input sequence. We notice the sharp focus of all the weightings on a single location in memory which means the model writes to and reads from one specific location of memory during one time step. During the input phase, at each time step, input information is stocked in one location of memory and the locations are successive which reflects the shift mechanism of location-based addressing. We observed that during the output phase, the read head reads exactly from the same locations where input information is memorized in order to copy the input sequence. This process is similar to how we create and iterate through arrays. We can consider that the NTM model is not just able to copy sequence but has learned the algorithm to copy.

In addition, we caught several values of parameters k, β, g, s, γ while the NTM was running, we observed that the g of write header is close to 0 in the write phase, and the g of read header is close to 0 in the read phase, which also confirms the fact that NTM uses location-based addressing to perform the copy task.

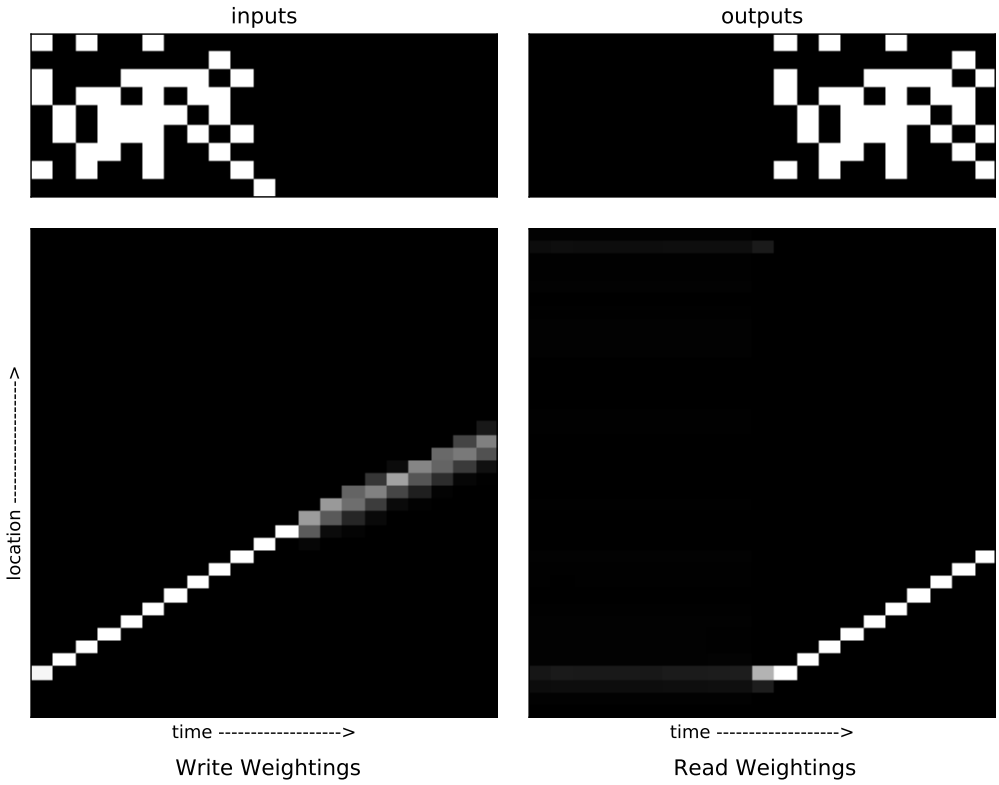

Figure 4: Write and read head/attention visualization. The top column is the inputs to the network (top) and the corresponding write head weightings (bottom). The right column is the outputs of the network (top) and the read head weightings (bottom).

Based on our code and our analysis of the captured parameters, we believe that the logical process learned by our NTM is a bit different from the original paper. The following is our pseudocode.

```
Initialise:
    Move read head to the start location
    Move write head to the same start location
While not input delimiter do:
    Receive input vector
    Move the write head to next location
    Write vector to this location
End While
While true do:
    Move the read head to next location
    Read vector from this location
    Emit this vector to output
End While
```

Listing 1: pseudocode of copy task

(e) Understanding the shift operator

The shift applied to weight by shifting weighting s_t can be expressed as a circular convolution :

$$\tilde{w}_t(i) \leftarrow \sum_{j=0}^{N-1} w_t^g(j) s_t(i-j)$$

As a result, the convolutions can be used to implement the shift operator. Since only shifts of -1, 0 and 1 are allowed, the vector s has length of 3 which indicates shift of -1, 0 and 1 respectively.

```
def _convolve(w, s):
    """Circular convolution implementation."""
    assert s.size(0) == 3
    t = torch.cat( [w[-2:] ,w, w[:2]])
    c = F.conv1d(t.view(1, 1, -1), s.view(1, 1, -1)).view(-1)
    return c[1:-1]
```

Listing 2: Convolution implementation for shift operator

For example, we assume the input weight is $w = [0.1, 0.1, 0.8, 0.1, 0.1, 0.1]$, the attention is in the third position. If $s = [0, 1, 0]$, then the output is still $[0.1, 0.1, 0.8, 0.1, 0.1, 0.1]$ which indicates no shift. If $s = [1, 0, 0]$, the output is $[0.1, 0.1, 0.1, 0.8, 0.1, 0.1]$, the attention has shifted to the right. And if $s = [0, 0, 1]$, the attention is shifted to the left which gives us $[0.1, 0.8, 0.1, 0.1, 0.1, 0.1]$.

If only forward shift is allowed, then the vector s should has length of 2 which indicates shift of 0 and 1. Below is the code modified.

```
def _convolve_forward_shift(w, s):
    assert s.size(0) == 3
    t = torch.cat( [w[-2:] ,w, w[:2]])
    s2=s.clone()
    s2[0] = s2[0]+s2[2]
    s2[2] = 0
    s=s2
    c = F.conv1d(t.view(1, 1, -1), s.view(1, 1, -1)).view(-1)
    return c[1:-1]
```

Listing 3: Only allow forward shifts

References

[1] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *CoRR*, vol. abs/1410.5401, 2014. [Online]. Available: http://arxiv.org/abs/1410.5401

[2] loudinthecloud. Neural turing machines (ntm) - pytorch implementation. [Online]. Available: https://github.com/loudinthecloud/pytorch-ntm