ASSIGNMENT 3: NEURAL TURING MACHINES [IFT6135]

JOSEPH D. VIVIANO, MARZI MEHDIZAD, JOHNATHAN GUYMONT

1. Filling the Gaps

Writing Parameters
In the writing head we have two parameters ($\boldsymbol{e_t}$ and $\boldsymbol{a_t}$). Writing involves two steps : erasing and adding
Erasing: In order to erase the old data a writing head needs the erase vector $\boldsymbol{e_t}$ of size M whose elements are values between 0 and 1, in addition to our length-N normalized weight vector w_t. The erase vector is used in conjunction with the weight vector to specify which elements in a row should be erased, left unchanged, or something in between. If the weight vector tells us to focus on a row, and the erase vector tells us to erase an element, the element in that row will be erased.

$$\tilde{M}_t(i) \leftarrow M_{t-1}\left[\mathbf{1} - w_t(i)\boldsymbol{e_t}\right]$$

Remark: In order to update the value of $_{e_t}$ we need to use the Sigmoid function after each back propagation to keep the elements of $\boldsymbol{e_t}$ between 0 and 1.
Adding: After M_{t1} has been converted to $\tilde{M}$, the write head uses a length-M add vector $\boldsymbol{a_t}$ to complete the writing step.

$$M_t(i) \leftarrow \tilde{M}_t + w_t(i)\boldsymbol{a_t}.$$

Addressing Parameters
We have a 4 stage-process and 5 parameters ($\boldsymbol{k_t}, \boldsymbol{\beta_t}, \boldsymbol{g_t}, \boldsymbol{s_t}, \boldsymbol{\gamma_t}$) to get the controller output.
Stage 1 (Content addressing parameters): The first stage is to generate a weight vector, based on how similar each row of memory is to a vector $\boldsymbol{k_t}$ emitted by controller. We introduce a the vector W_t^c as the *content vector*. The content vector allows the controller to choose values which are similar to previous values, which is called content based addressing.
For each head , the controller produces a *key vector* $\boldsymbol{k_t}$ that will be compared to each row of M_t by using a measure. this measure can be the euclidean measure , but here the authors used the cosine measure defined as follows

$$K(u, v) := \frac{u.v}{||u||.||v||}$$

The ith row of the content weight vector is obtained by the following formula

$$w_t^c(i) \leftarrow \frac{\exp\left(\boldsymbol{\beta_t} K(\boldsymbol{k_t}, M_t(i))\right)}{\sum_j \exp\left(\boldsymbol{\beta_t} K(\boldsymbol{k_t}, M_t(j))\right)}.$$

The parameter $\boldsymbol{\beta_t}$ which is called *key strength*, is a scalar constrained in the $(0, \infty)$. It is used to show that how focused the content weight should be. We have two cases for the values of $\boldsymbol{\beta_t}$:

Date: March 2018.

i: If $\boldsymbol{\beta_t} \leq 1$: In this case the values of weight vector spread over a large area.
ii: If $\boldsymbol{\beta_t} > 1$: In this case, As the values of $\boldsymbol{\beta}_t$ get larger and larger then the weight vector will be concentrated on the most similar row in memory.
Example: let $\boldsymbol{k}_t = (3 \quad 2 \quad 1)$ and the memory matrix is as follows:

$$M_t = \begin{bmatrix} 1 & 2 & 3 & 1 & 0 & 1 \\ 1 & 1 & 2 & 4 & 0 & 0 \\ 2 & 4 & 1 & 5 & 1 & 0 \end{bmatrix}$$

If $\boldsymbol{\beta_t} \to 0$ then

$$w_t^c \to (0.16 \quad 0.16 \quad 0.16 \quad .16 \quad 0.16 \quad 0.16).$$

Let $\boldsymbol{\beta_t} = 5$, then

$$w_t^c = (0.16 \quad 0.1 \quad 0.47 \quad 0.08 \quad 0.13 \quad 0.17).$$

Let $\boldsymbol{\beta_t} = 50$

$$w_t^c = (0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0)$$

Remark: To update the value of $\boldsymbol{\beta_t}$ in each back propagation of our model we use the *Softplus* funtion over the value $\boldsymbol{\beta_t}$ obtained by back propagation process to have get a non-negative value. So the updated beta will be:

$$\boldsymbol{\beta_t'} = log(1 + e^{\boldsymbol{\beta_t}})$$

Stage 2 (interpolation parameter): Sometimes the our variables are stored in different addresses and we don't care about their values , because the controller will retrieve them from and perform the desired algorithm. So the variables are addressed by location and not the content. So in this stage we decide when we should use or ignore the content-based addressing . We introduce the scalar $\boldsymbol{g_t} \in (0, 1)$ named *interpolation gate* that combine w_t^c with the previous step weight vector w_{t-1} and produce the gated weighting w_t^g defined as follows

$$w_t^g \leftarrow \boldsymbol{g_t} w_t^c + (1 - \boldsymbol{g_t}) w_{t-1}$$

So as we observe in this formula:
i: If $\boldsymbol{g_t} \to 0$ then the weight in previous stage is chosen and w_t^c is completely ignored.
ii: if $\boldsymbol{g_t} \to 1$ then the content base addressing is selected and the previous step's weight will be ignored.
Example: Let $w_t^c = (0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0)$ and $w_{t-1} = (0.9 \quad 0.1 \quad 0 \quad 0 \quad 0 \quad 0)$
If $g_t \to 1$ then

$$w_t^g \to (0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0)$$

If $g_t = 0.5$ then

$$w_t^g = (0.45 \quad 0.05 \quad 0.5 \quad 0 \quad 0)$$

If $g_t \to 0$ then

$$w_t \to (0.9 \quad 0.1 \quad 0 \quad 0 \quad 0 \quad 0).$$

Remark: To update the value of $\boldsymbol{g_t}$ in each back propagation of our model we use the *Sigmoid* function over the value $\boldsymbol{g_t}$ obtained by back propagation process to have get a value between 0 and 1. So the updated $\boldsymbol{g_t}$ will be:

$$\boldsymbol{g_t'} = sigmoid(\boldsymbol{g_t})$$

Stage 3(Convolutional shift parameter): After interpolation stage, each head emits a shift weighting $\boldsymbol{s_t}$. This parameter makes the controller able to shift concentrating to other rows. The range of shift is given as a parameter. For example

if it is between -1 and 1, then the shift range size is 3 (shift forward a row(+1), stay in the same row (0), shift backward a row(-1)).
We perform a shift modulo N such that a shift forward for the last row of the memory shift the head's attention to the the first rows.
the following convolutional shift is performed to produce the shifted weight $\tilde{w}_t$

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

Example: let $\boldsymbol{s_t}$ have three elements corresponding to the degree to which shifts of -1, 0 and 1 are performed, assume that we are working on the element with index 3, and w_t^g has 5 entries. So we have
let $w_t^g = (0.45 \quad 0.05 \quad 0.5 \quad 0 \quad 0)$
If $s_t = (1 \quad 0 \quad 0)$ (we note that $\sum_i s_t(i) = 1$) then we have

$$\tilde{w_t} = (0.05 \quad 0.5 \quad 0 \quad 0 \quad 0.45)$$

which means that all element shifted to the left.
If $s_t = (0 \quad 0 \quad 1)$ (we note that $\sum_i s_t(i) = 1$) then we have

$$\tilde{w_t} = (0 \quad 0.45 \quad 0.05 \quad 0.5 \quad 0)$$

which means that all element shifted to right. If $s = (0.5 \quad 0 \quad 0.5)$ Then

$$\tilde{w_t} = (0.25 \quad 0.425 \quad 0.025 \quad 0.25 \quad 0 \quad 0.225)$$

which means that each number gives half of its value to the right and the other half to the left.
Stage 4(sharpening): The fourth stage, sharpening is used to prevent the shifted weight $\tilde{w}_t$ from blurring, so we use a scalar $\boldsymbol{\gamma_t} > 1$ to sharpen the weight obtained from the previous stage.

$$w_t(i) \leftarrow \frac{(\tilde{w}_t(i))^{\gamma_t}}{\sum_j (\tilde{w}_t(j))^{\gamma_t}}$$

Example:
let $\tilde{w}_t = (0 \quad 0.45 \quad 0.05 \quad 0.5 \quad 0 \quad 0)$ If $\gamma_t = 0$ then

$$w_t = (0.16 \quad 0.16 \quad 0.16 \quad 0.16 \quad 0.16 \quad 0.16)$$

In this case (when the value of $\gamma_t < 1$), Then w_t will be more blurred. If $\gamma_t = 5$ then

$$w_t = (0 \quad 0.37 \quad 0 \quad 0.62 \quad 0 \quad 0).$$

In this case the values are more sharpened but not enough.
If $\gamma_t = 50$ then

$$w_t = (0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0)$$

In this case since the value of γ_t is large enough then the array will be completely sharpened.
Remark: To update the value of $\boldsymbol{\gamma_t}$ in each back propagation of our model we use the *Softplus* function over the value $\boldsymbol{\gamma_t}$ obtained by back propagation process to have get a value non-negative. So the updated $\boldsymbol{\gamma_t}$ will be:

$$\boldsymbol{\gamma_t'} = 1 + Softplus(\boldsymbol{\gamma_t})$$

We add 1 to the Softplus function to have the updated value bigger than 1, in order of sharpening and not blurring.

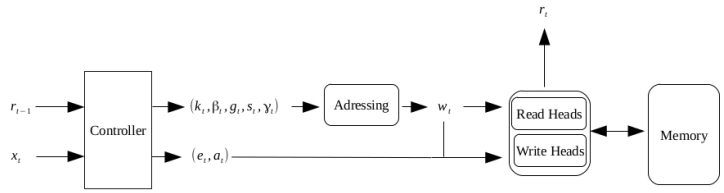

FIGURE 1. Ineraction between the controller and read/write heads.

2. Implementing the Neural Turing Machine

Our implementation was forked from `https://github.com/loudinthecloud/pytorch-ntm`. The code can be found at `https://github.com/josephdviviano/pytorch-ntm`.

2.1. **Parameters.** As requested, all models made use of a single 100-node layer (for the NTM models, this layer was in the controller, for the LSTM, this layer was the hidden layer). Both NTM models used a single read and write head.

(1) LSTM-NTM: number of parameters = 62860
(2) MLP-NTM: number of parameters = 13260
(3) LSTM (baseline): number of parameters = 45408

2.2. **Hyperparameters Chosen.** The same hyperparameters were used for all models to facilitate comparison. All models were trained on 40000 sequences, with a random length between 1 and 20. We used the rmsprop optimizer, with a learning rate of 0.0004, momentum of 0.9, and alpha of 0.95. The loss was binary cross entropy.

Please see figures 2-4 for the convergence of the three models (LSTM-NTM, MLP-NTM, and LSTM baseline) using these hyperparameters, respectively.

Furthermore, the curve per sequence size can be seen for each in figures 5-7:

It can be seen that the MLP-NTM model converges fastest, followed by the LSTM-NTM. The LSTM model improves over time, but does not converge nearly as quickly as either NTM model, replicating the 'qualitative' difference between methods noted in the original paper. We speculate that the MLP-NTM converges slightly faster because training an MLP generally takes less batches, and is less sensitive to sample variance, than any recurrent model including LSTM (of similar size). The baseline LSTM model likely did not converge as quickly because of the slow leanring rate. The NTM variants likely do not fare well with higher learning rates due to the powerful influence of the memory matrix on the gradients, which the baseline LSTM model does not have access to. However, it is also possible that the

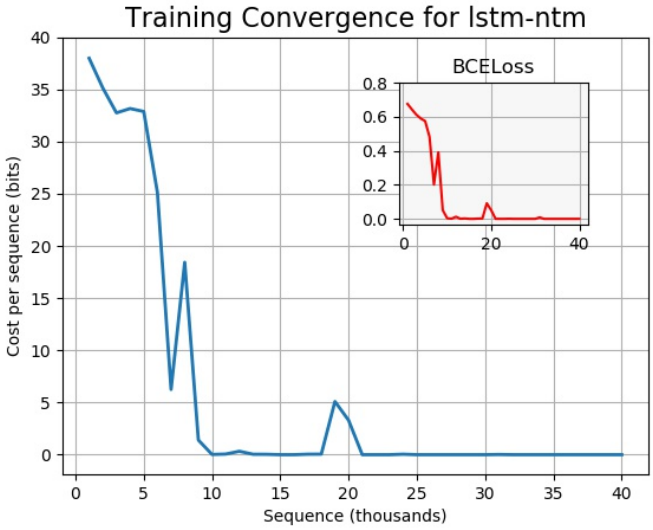

FIGURE 2. Training curve of LSTM-NTM.

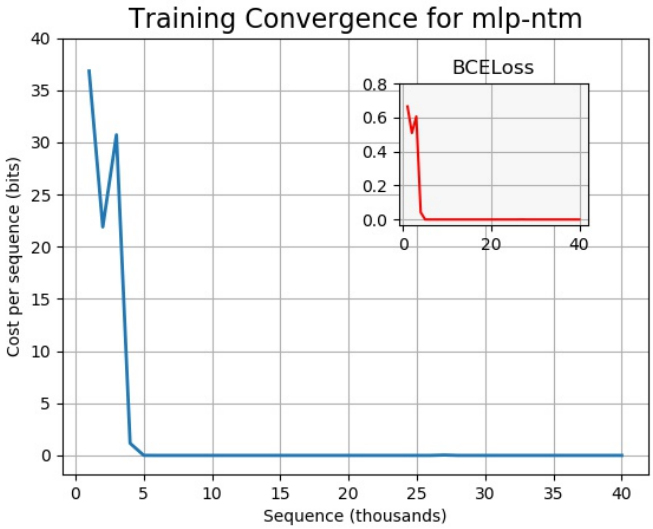

FIGURE 3. Training curve of MLP-NTM.

baseline LSTM learning rate is approximately optimial, and the model is simply learning the task as quickly as is possible for that architecture.

2.3. **Generalization to Longer Sequences.** For these three trained models, we tested their ability to correctly copy binary sequences of length $T \in 10, 20, ..., 100$, 20 times for each T. We plot the mean and standard deviation across those 20

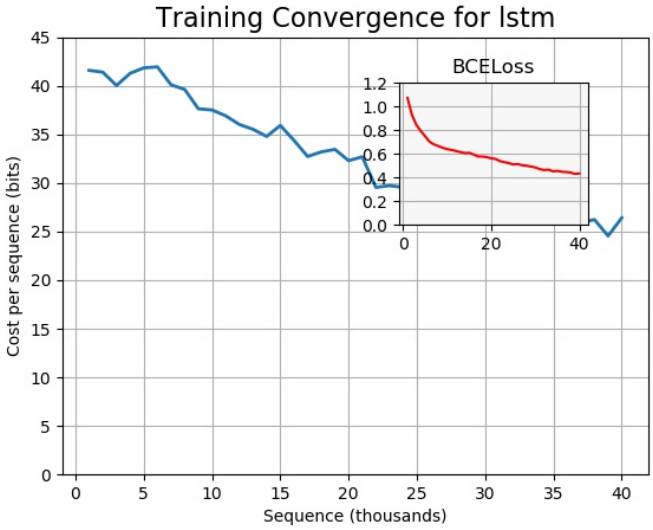

FIGURE 4. Training curve of LSTM baseline.

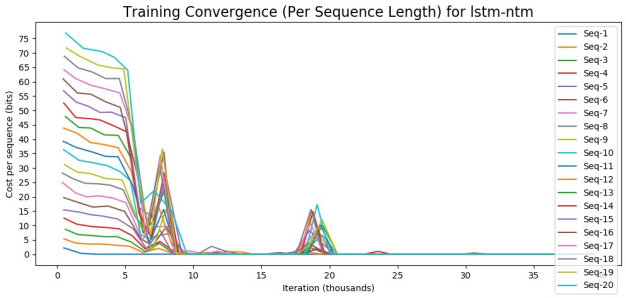

FIGURE 5. Length-specific training curves of the LSTM-NTM.

samples per T below for the LSTM-NTM, MLP-NTM, and LSTM in figures 8-10:

In this experiment, we expected both of the NTM models to be able to generalize to all longer sequences, and the LSTM model to rapidly degrade as sequence length gets longer. This is because we know and expect recurrent models like the LSTM to have trouble composing long sequences due to the vanishing/exploding gradients problem, which leads to all recurrent models (to varying degrees) not being able to learn very long-term dependencies. The NTM model gets around this in a clever way: the memory module can store 'chunks' of sequences that the controller can then compose to generate a much larger sequence than could typically be held in just the controller network.

This observation leads us to our results. Figure 8 clearly shows that the MLP-NTM begins to falter as the sequence length reaches $T = 50$, while, the LSTM-NTM fares

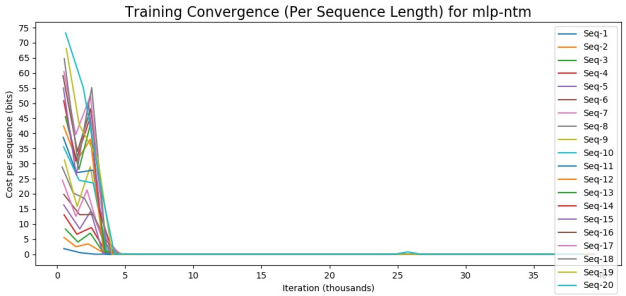

FIGURE 6. Length-specific training curves of the MLP-NTM.

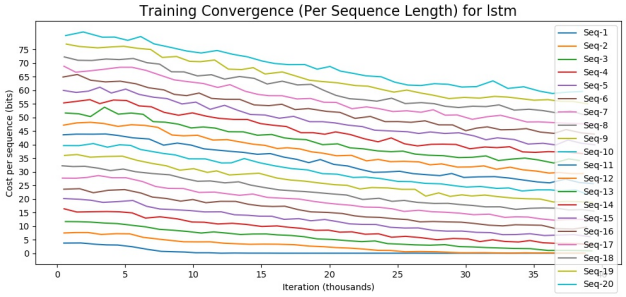

FIGURE 7. Length-specific training curves of the LSTM baseline.

well at all sequence lengths, only showing a small amount of weakness at $T = 100$. The graph suggests good performance of the LSTM-NTM for sequences much longer than $T = 100$. We speculate that this is because using a LSTM as the controller allows the NTM to compose longer sequences of data held in the memory matrix than a MLP controller given a fixed controller size (in this case, one hidden layer of 100 nodes), since the LSTM is designed to learn sequences. Said slightly differently, the optimial configuration for this task uses an LSTM for the controller, which can learn how to sequence the 'chunked' memories held in the memory networks effectively, for tasks that are sequential such as this one. We speculate that for complex recognition tasks, an MLP or even convolutional architecture in the controller would be useful, where the features stored in the memory matrix represent sets of very high-order features that would require an otherwise massive convolutional network to store effctively.

Unsuprisingly, the baseline LSTM model's performance quickly degrades the sequence length increases, starting from $T = 10$.

2.4. **Visualization of Read/Write Heads (Attention).** We need a graph of the read/write heads.

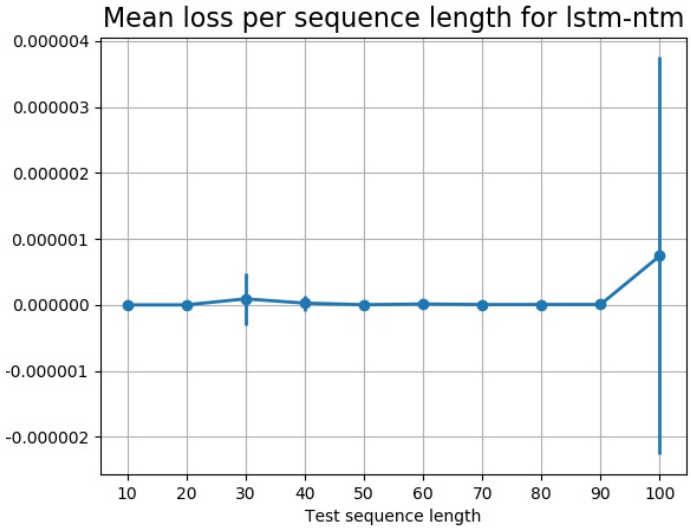

FIGURE 8. Generalization to longer sequences for the LSTM-NTM.

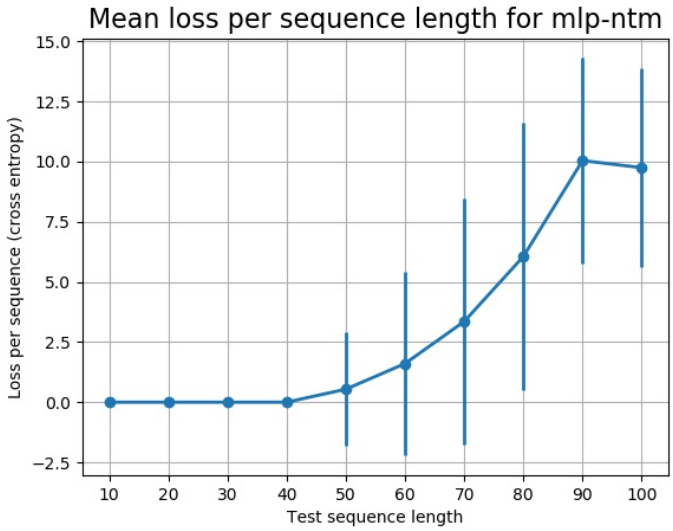

FIGURE 9. Generalization to longer sequences for the MLP-NTM.

2.5. **Understanding the Shift Operator.** We need the snippet of code using the shift operator, modification, etc.

UNIVERSITÉ DE MONTRÉAL
Email address: joseph@viviano.ca, jonathan@guymont.com, marzieh.mehdizadeh@gmail.com

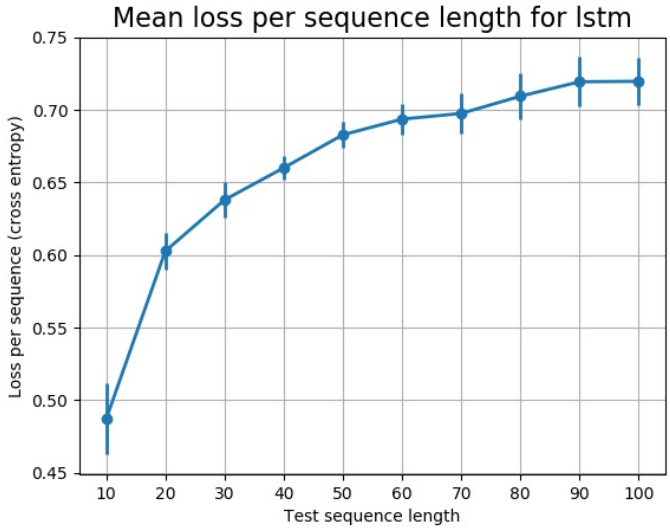

FIGURE 10. Generalization to longer sequences for the LSTM baseline.