IFT6135 - Assignment 3

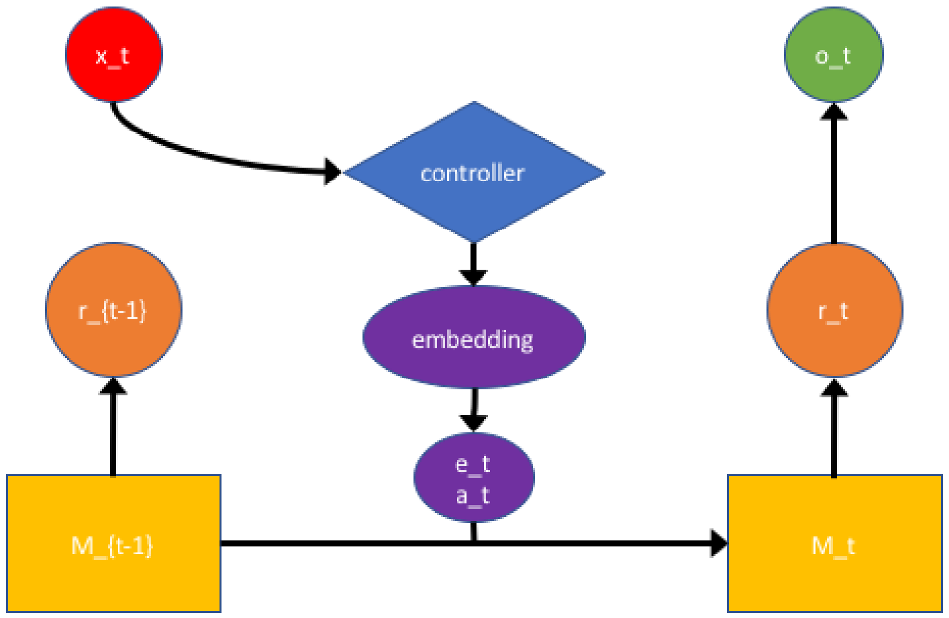
Valentin Thomas, Rémi Le Priol, Salem Lahlou
Monday April 2nd

Filling in the gaps

(a)We constrained the parameters as follow:

- For $\beta \in (0, \infty)$, we use a linear transformation of the output of the controller, followed by the softplus function: $\beta = \log(1 + \exp(W_{h\beta} h))$, with $W_{h\beta} \in \mathbb{R}^{100\times1}$.
- For $\gamma > 1$, same idea, we use a linear transformation of the output of the controller, followed by $1 + softplus$: $\gamma = 1 + \log(1 + \exp(W_{h\gamma} h))$, with $W_{h\gamma} \in \mathbb{R}^{100\times1}$.
- The shift weighting is a vector whose elements should sum up to one. After the linear transformation of the output of the controller, we use a softmax layer: $s = softmax(W_{hs} h)$, with $W_{hs} \in \mathbb{R}^{100\times3}$.
- The interpolation gate should be in $(0, 1)$. So a sigmoid seems like a good choice for the post-linearity transformation: $g = sigmoid(W_{hg} h)$, with $W_{hg} \in \mathbb{R}^{100\times1}$
- The key vector doesn't have to satisfy any constraint whatsoever.

(b)Below is a diagram showing the relationships we implemented. The article does not mention any other relationships.



Implement the neural Turing Machine

(a) Model sizes:

Similar to the paper, we set the NTM memory's size to $N \times M = 128 \times 20 = 2560$.

- NTM with feedforward controller: 17316
- NTM with LSTM controller: 60916
- LSTM baseline: 45309

(b) Training curves

The accuracy mentioned here is averaged over 20 batches of sequences, each batch containing 1000 sequences of length T, with $T = 1..20$ (as in training actually). The accuracy is measured by rounding the outputs: we use a sigmoid layer on the outputs to have actual outputs in $(0, 1)$ - we compare with 0.5 to have byte-like outputs.

Vanilla LSTM

We performed a quick hyperparameter search (learning rate and mini batch size), and the winning pair was ($lr = 0.01, mbsize = 200$). We used Adam (with default secondary hyperparameters) as an optimization algorithm.

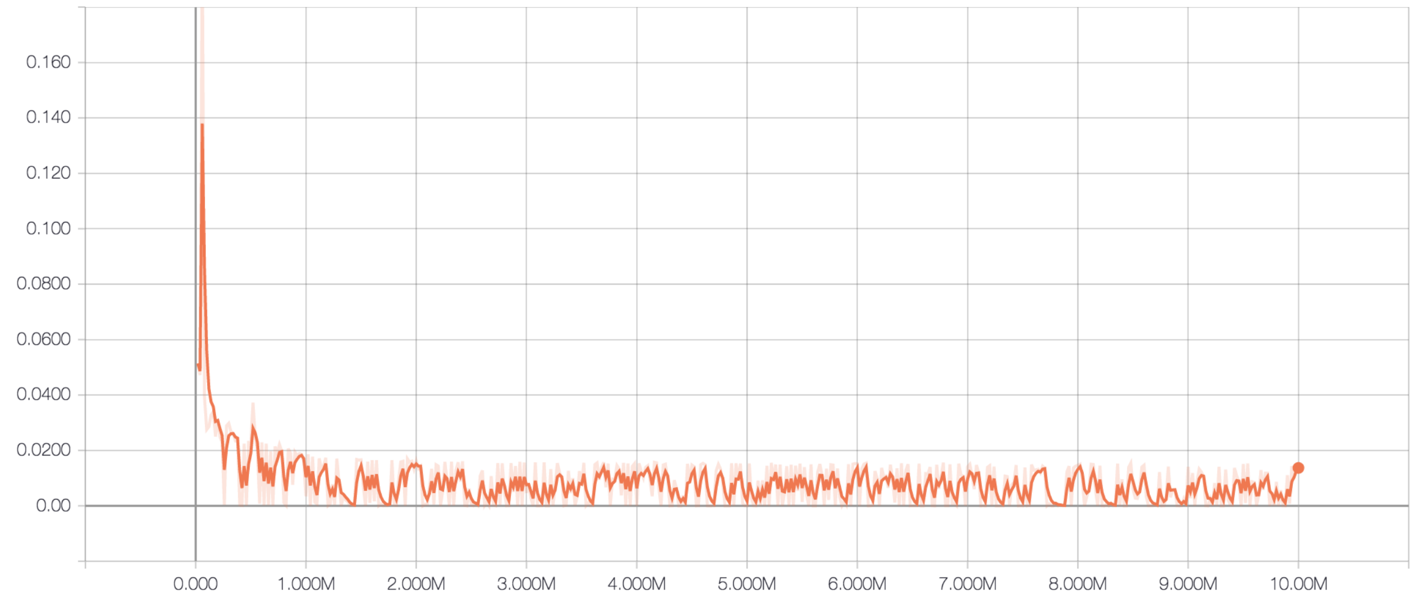


Fig 1 : Loss vs number of sequences seen.

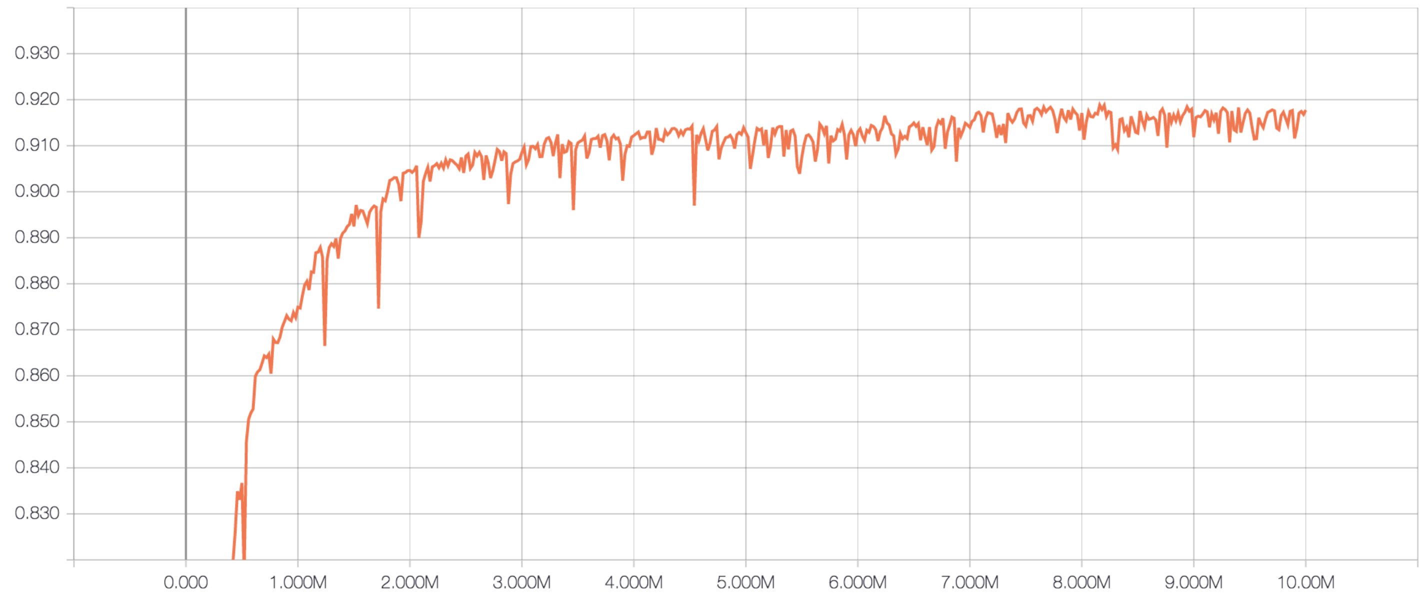


Fig 2 : Accuracy vs number of sequences seen.

It takes millions of sequences to really learn anything. The LSTM from the paper achieves a much better score. That is probably due to the difference in depth and hidden layer size.

Neural Turing Machine

In accordance to what is advised in the paper, we used RMSProp (Alex Graves' version) with a momentum of 0.9 to train the NTM. A quick hyperparameter search made us settle for $lr = 1e-4$. We observed that the model did not learn faster with larger batches. We thus settled for batches of size 1, so as to minimize the sample complexity of the procedure (i.e. the number of samples required to train).

Plotting the results of the three models on the same graph is quite irrelevant since Vanilla LSTM requires **millions** of sequences in order to achieve a reasonable (but not great) performance, whereas both NTM versions only require a few **hundreds**. The NTM model is much better suited for this task than the LSTM.

In the plots below, **the pink curve is the NTM with a feedforward controller, and the grey curve is the NTM with an LSTM controller.**

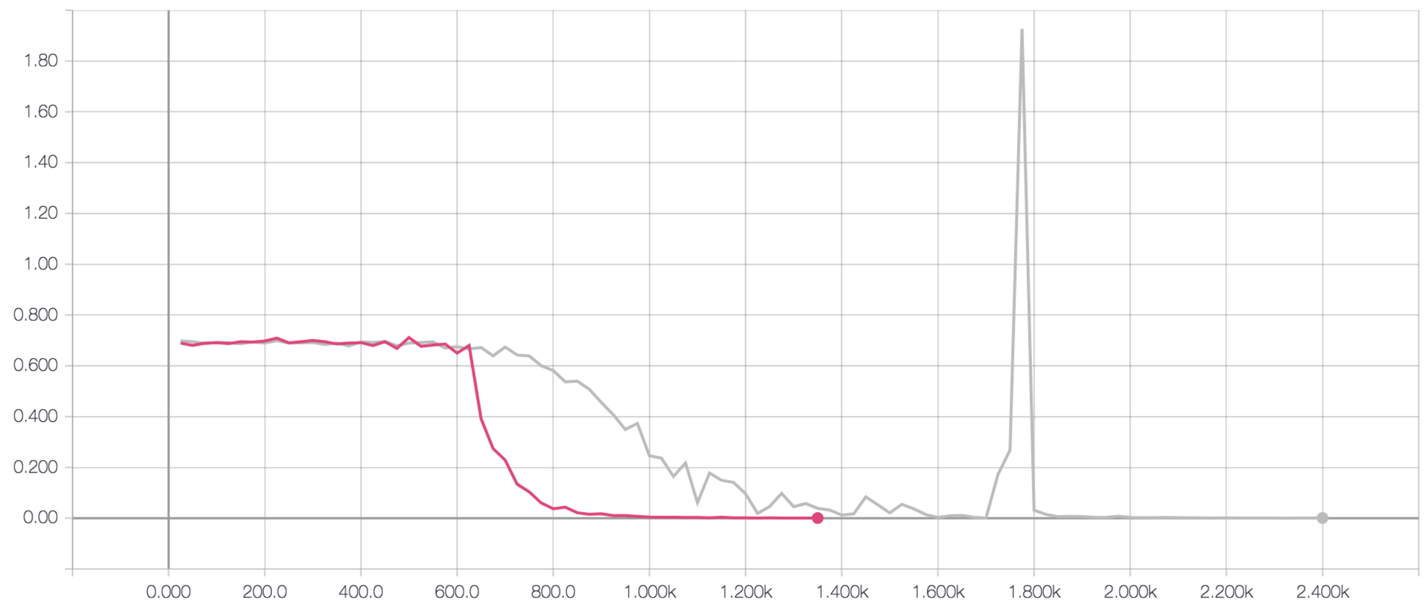


Fig3: Loss vs number of sequences seen. Pink is NTM MLP, Gray is NTM LSTM

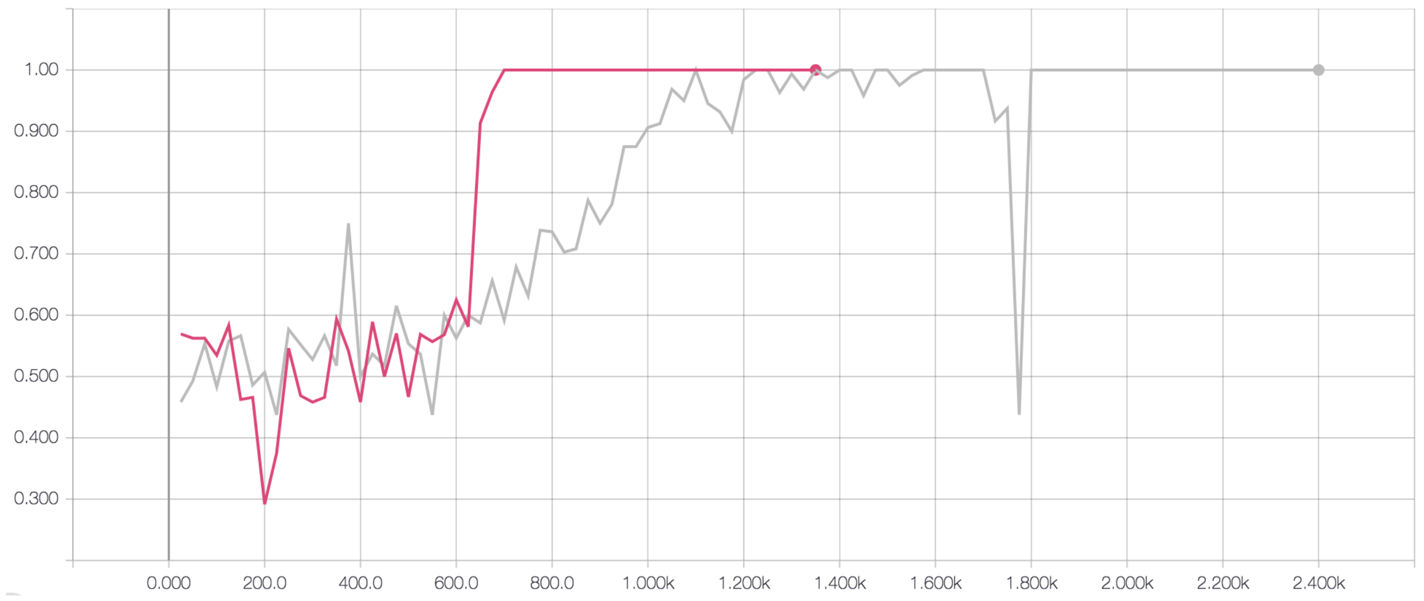


Fig4: Accuracy vs number of sequences seen. Pink is NTM MLP, Gray is NTM LSTM

The feedforward controller learns faster than the LSTM. This may be due to the simplicity of the model. The feedforward network has more than 3 times less parameters to learn. One could argue that it is overly tuned for the copy task.

The LSTM is also more unstable (notice the huge drop in accuracy around 1700 steps).

It is interesting that the NTM model provides a very efficient memory with much fewer additional parameters than the LSTM. The LSTM memory needs matrix multiplication at each step. On contrary the NTM memory requires only vector multiplication.

(c) Generalization to longer sequences

We present here the accuracy obtained for different values of the sequence length ($T \in [[1, 20]]$ and $T \in \{30, 40, \ldots, 100\}$). We averaged over 20 examples for each sequence length to report the accuracy.

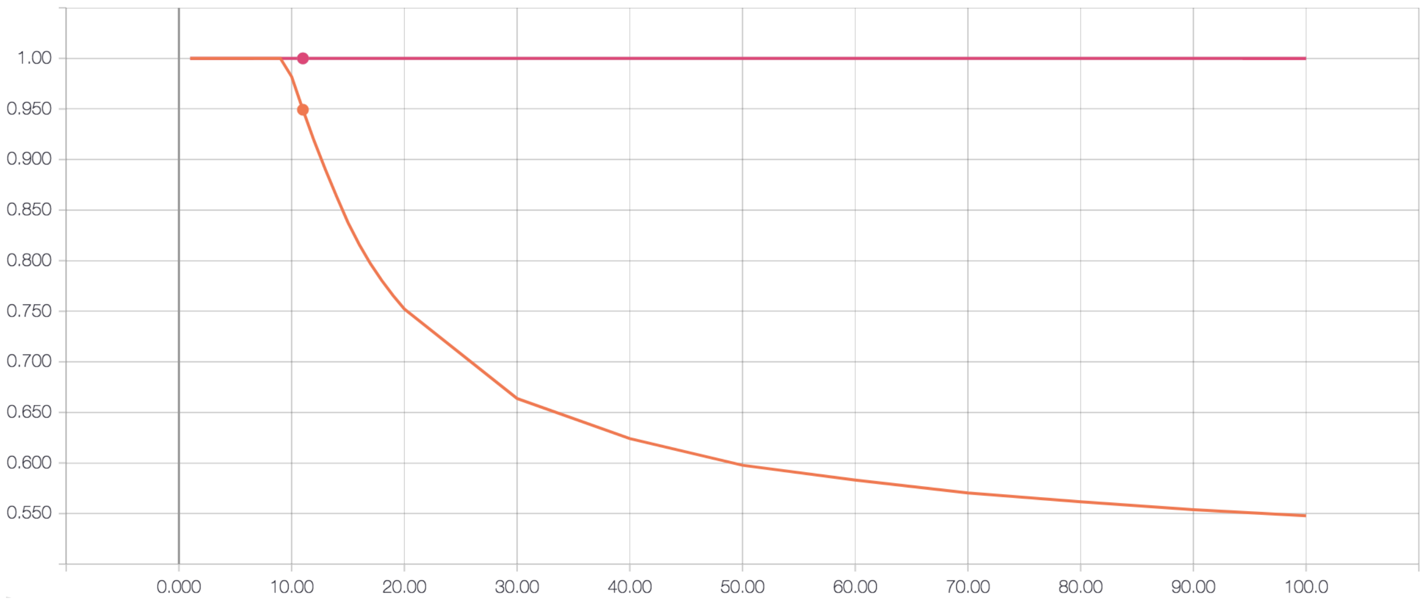


Fig5: Final accuracy vs length of sequences T. In pink, both NTM-MLP and NTM-LSTM. In orange, Vanilla LSTM.

The Neural Turing machine solves the task perfectly while the length of the sequence is smaller than its memory.

We show some examples of the results obtained by the LSTM and the NTM with MLP controller (it is redundant to show examples for NTM with LSTM as it is as perfect as NTM with MLP). The 1st and 3rd rows show true inputs, and the 2nd and 4th rows show the outputs of the model.

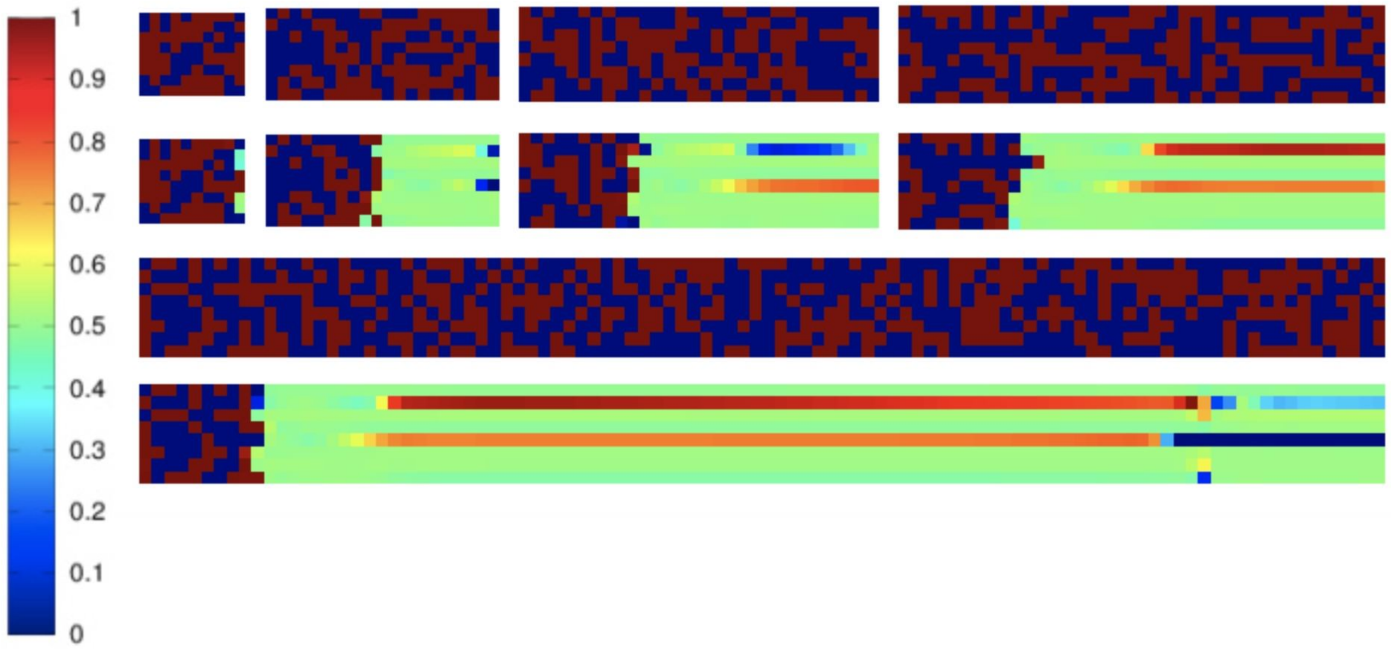


Fig6: Examples with Vanilla LSTM

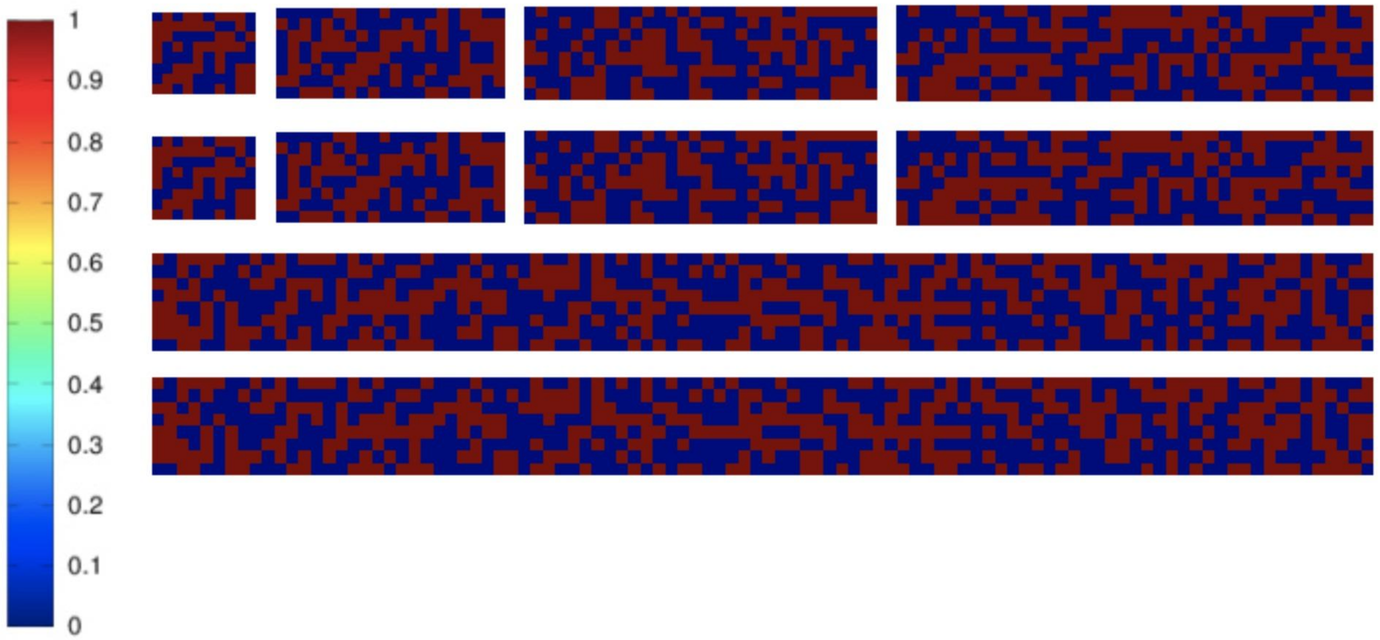


Fig7: Examples with NTM - MLP controller (perfect results also obtained using LSTM controller)

Unsurprisingly, the LSTM doesn't learn very long sequences, and fails to reproduce even sequences which lengths were used during training. This is certainly because we used a hidden state of size 100 (as advised in the homework), which is not enough to store the input values.

The worst performance is on longer sequences. If we don't sample randomly uniformly T in $1..20$, but rather using a probability distribution $p(T)$ proportional to T, we obtain better results on the longer sequences $T = 10..20$.

Both versions of the Neural Turing Machine work perfectly. Not only they achieve perfect accuracy on sequences of higher lengths, but they do so without local errors (i.e. the outputs are either very close to 0 or very close to 1). Magic!

(d) Visualizing the read and write heads/attention

We report the read and write weights for a sequence of length 10.

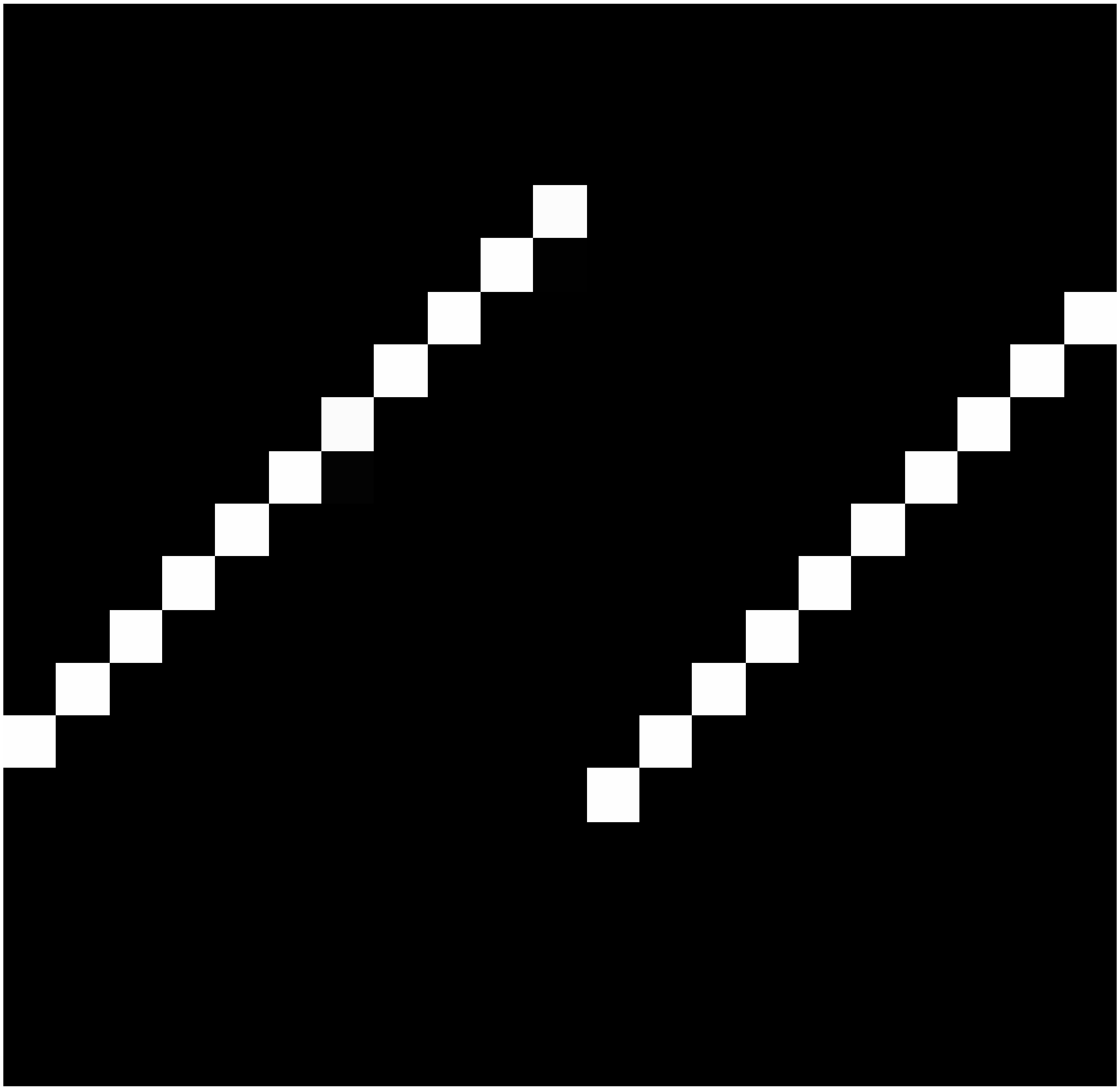


Fig8: Read and write weights
In this image, each column is a fraction of attention weights. The full weights are of size 128 and the full image is too high. The x-axis is the unfolding of the task : write then read.

We report the write attention (left) while the input is fed to the model, then we report the read attention (right) while the model outputs the copy. We observe that the model has learnt the same policy as described in the paper : write each input at a given location while shifting the write head, then read these from the same locations.

(e) Understanding the shift operator

The shift operator is a circular convolution between shift weights and attention weights. By circular we mean that we treat the attention weights as being periodic.

Below is the code allowing both forward and backward shift.

```
import torch


def circular_conv(w, s):
    """Return the cyclic convolution of w with s.

    :param w: attention weights. 2D tensor of size batch*N. The first axis index the
    second axis index the memory.
    :param s: shift weights. 2D tensor of size batch*3. The first axis index the ba
    axis index the shift.
    """
    circular_w = torch.cat([w[:, -1].unsqueeze(1), w, w[:, 0].unsqueeze(1)], 1)
    ans = s[:, 2].unsqueeze(1) * circular_w[:, :-2]
    ans = ans + s[:, 1].unsqueeze(1) * circular_w[:, 1:-1]
    ans = ans + s[:, 0].unsqueeze(1) * circular_w[:, 2:]
    return ans
```

We want to restrict to forward shifts only, while keeping 3 shift weights. A simple way to do so is to change the definition of circular_w, to shift it once to the right.

```
def forward_circular_conv(w, s):
    """Return the cyclic convolution of w with s, while preventing moving backward.

    :param w: attention weights. 2D array of size batch*N. The first axis index the
    second axis index the memory.
    :param s: shift weights. 2D array of size batch*3. The first axis index the bat
    axis index the shift.
    """
    circular_w = torch.cat([w, w[:, :2].unsqueeze(1)], 1)
    ans = s[:, 2].unsqueeze(1) * circular_w[:, :-2]
    ans = ans + s[:, 1].unsqueeze(1) * circular_w[:, 1:-1]
    ans = ans + s[:, 0].unsqueeze(1) * circular_w[:, 2:]
    return ans
```