Albert Lee and Tom Wu

Neural Turing Machine Assignment Solutions

1. Note that most of this code can be transferred directly from `train_copy.py` once you understand how the NTM is trained. Changes are highlighted in red.

```
def make_model(
                input_size=8,
                output_size=8,
                mem_size=128,
                mem_width=20,
                hidden_size=100):

        P = Parameters()

        # Build the controller and model
        ctrl = controller.build(P,input_size,output_size,mem_size,mem_width,hidden_size)
        predict = model.build(P,mem_size,mem_width,hidden_size,ctrl)
        input_seq = T.matrix('input_sequence')
        [M_curr,weights,output] = predict(input_seq)

        # Return a Theano function for the NTM
        test_fun = theano.function(
                        inputs=[input_seq],
                        outputs=[weights,output]
                )

        return P,test_fun
```

2a. Changes are made to `build_head_curr` in `model.py`

```
def build_head_curr(weight_prev,M_curr,head,input_curr):
        """
        Implement addressing mechanism shown in Figure 2 in paper.
        Also return add and erase vectors computed by head.
        """
        # input_curr is hidden layer from controller
        # this is passing the hidden layer into the heads layer
        # which computes key, beta, g, shift, erase, and add
        # as outputs (see head_params in head.py)
        key,beta,g,shift,erase,add = head(input_curr)

        # 3.3.1 Focusing b Content (Equation (5))
        weight_c = U.vector_softmax(beta * similarity(key,M_curr)) # part i
        weight_c.name = "weight_c"

        # 3.3.2 Focusing by Location (Equation (7))
        weight_g      = g * weight_c + (1 - g) * weight_prev # part ii
        weight_g.name = "weight_g"
```

```
# Equation (8)
weight_shifted = shift_convolve(weight_g,shift) # part iii
# Normalize weights
weight_curr = weight_shifted / T.sum(weight_shifted)

return weight_curr,erase,add
```

The NTM was trained for 50,000 episodes and the trained model was evaluated on the following sequence lengths: 10, 20, 50 and 120. The plots for the input, the expected output and the actual output are shown in Figures 1, 2, and 3.

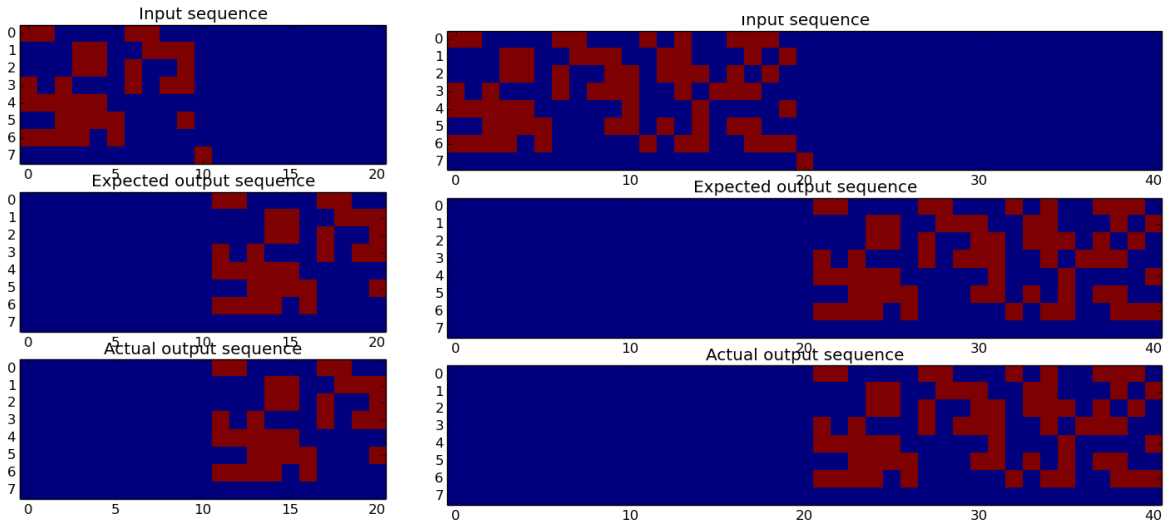

Figure 1: Test result for the NTM with sequences of lengths 10 (left) and 20 (right)

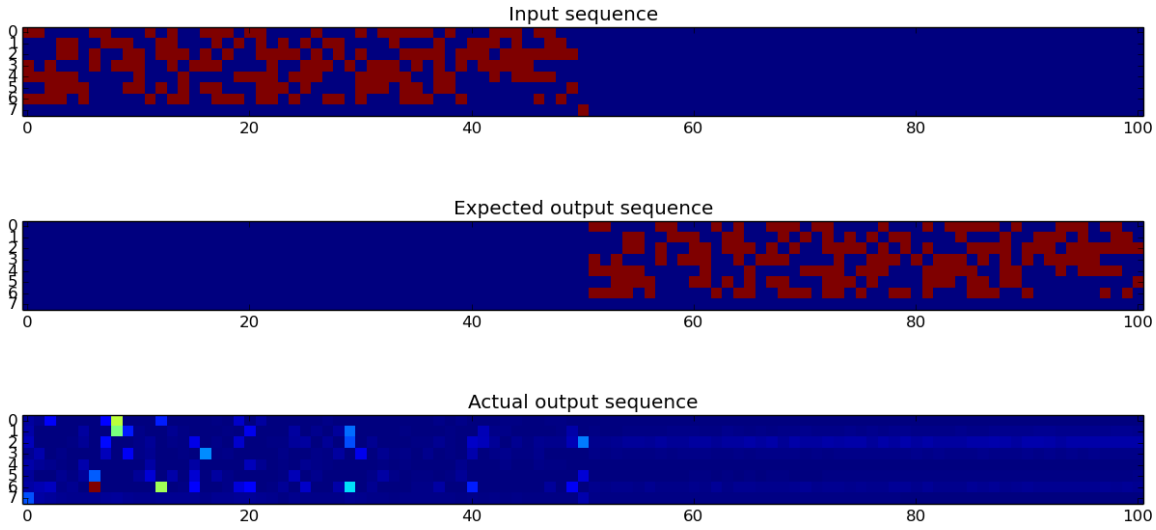

Figure 2: Test result for the NTM with sequences of length 50

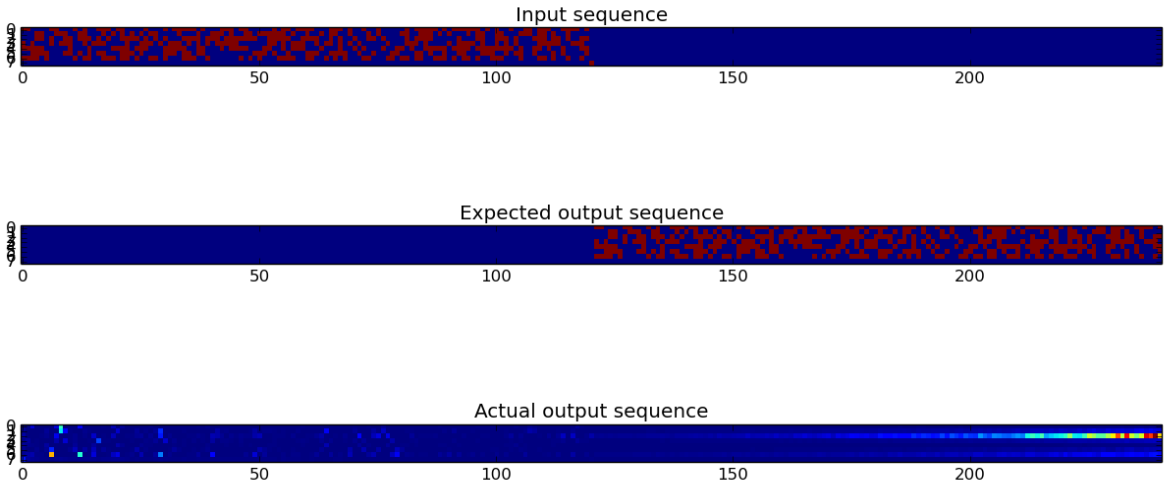

Figure 3: Test result for the NTM with sequences of length 120

The average error rate for each of the above test cases are displayed in Table 1.

Table 1: Average error rate of the trained NTM

Sequence Length	Average Error Rate
10	0.018%
20	0.020%
50	21.5%
120	21.7%

From the figures and the table, it is obvious to see that the trained NTM can copy smaller sequences almost perfectly. However, the NTM fails to effectively copy any data for longer sequences. This can be attributed to the absence of the sharpening feature in the NTM. Without the sharpening of weights, after a significant number of convolutions during training, the weights cannot focus on a particular location any more.

2b. In `head.py`, we add weights and biases for gamma (the sharpening factor)

```
P["W_%d_gamma"] = U.initial_weights(input_size)
P["b_%d_gamma"] = 0.
```

and calculate and return gamma in `head_params`

```
_gamma_t = T.dot(x,P["W_%d_gamma"]) + P["b_%d_gamma"]
gamma_t = T.nnet.softplus(_gamma_t) + 1.
...
return key_t,beta_t,g_t,shift_t,gamma_t,erase_t,add_t
```

To `build_head_curr` in `model.py` we add:

```
key,beta,g,shift,gamma,erase,add = head(input_curr)
...
weight_sharp   = weight_shifted ** gamma
weight_curr    = weight_sharp / T.sum(weight_sharp)
```

After the sharpening mechanism was added, the NTM was trained for 50,000 episodes and the trained model was evaluated on the following sequence lengths: 10, 20, 50 and 120. The plots for the input, the expected output and the actual output are shown in Figures 4, 5 and 6:

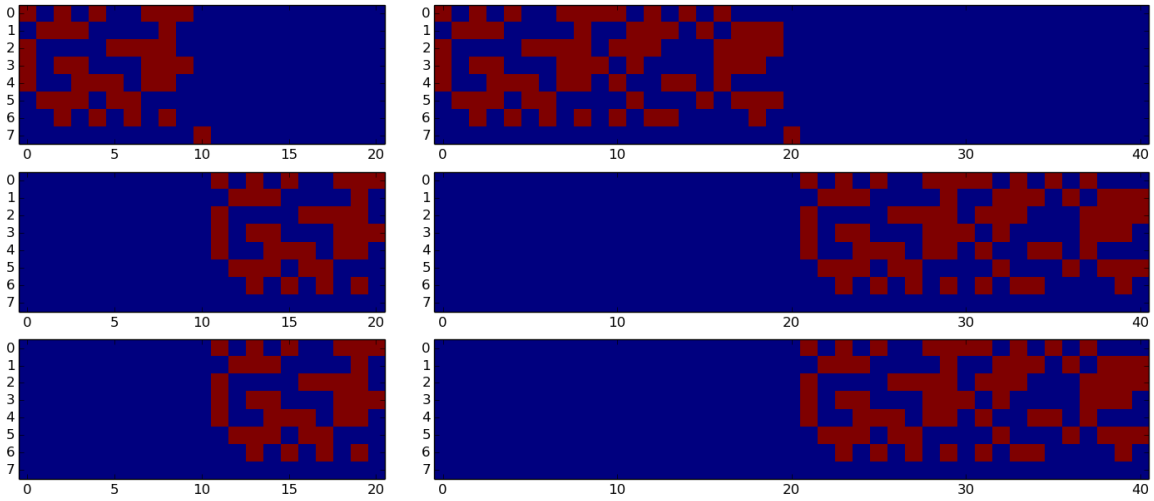

Figure 4: Test result for lengths 10 (left) and 20 (right), **with sharpening**

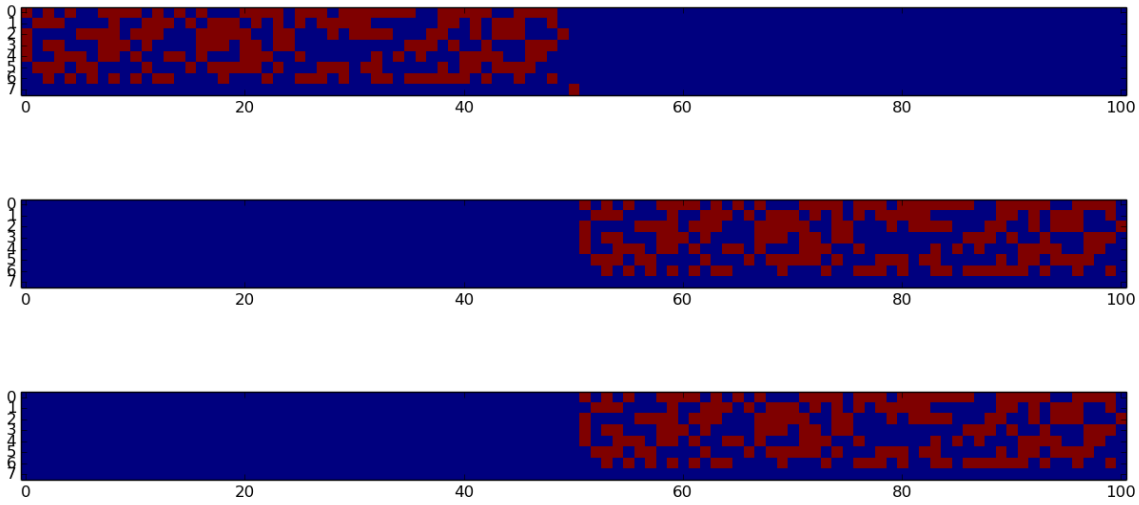

Figure 5: Test result for lengths 50, **with sharpening**

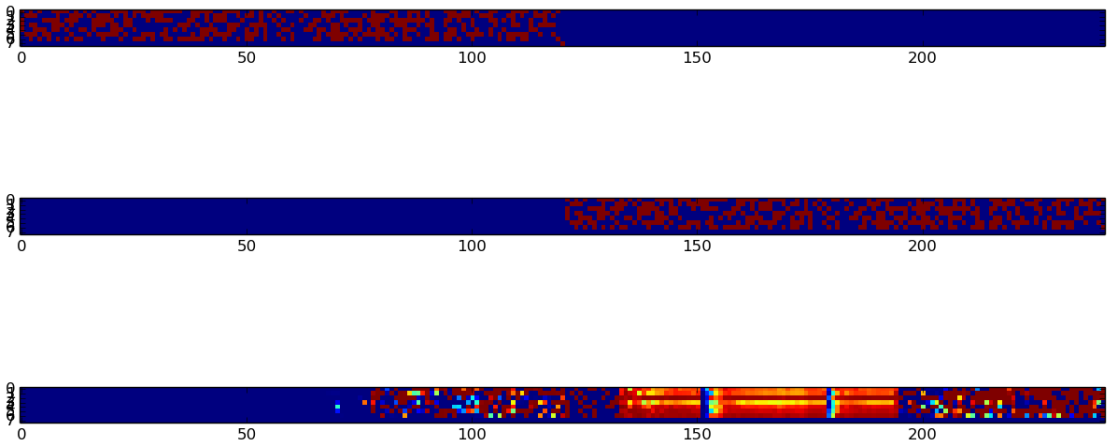

Figure 6: Test result for lengths 120, **with sharpening**

After sharpening was added, the NTM correctly copies sequences up to a length of 50 very accurately, as demonstrated by Figure 5. This is significantly superior to the NTM without sharpening (which fails to copy any data at length 50, as in Figure 2). However, even with sharpening, the NTM is still not able to generalize well to a sequence of length 120 (see Figure 6). The results are summarized in Table 2 below.

Table 2: Average error rate with and without sharpening

Sequence Length	Average Error Rate (**without sharpening**)	Average Error Rate (**with sharpening**)
10	0.018%	0.0163%
20	0.020%	0.014%
50	21.5%	0.0139%
120	21.7%	29%

3a. As noted in the paper, content-based addressing is used for the copy task to jump to the start of the sequence while location-based addressing is used to move along the sequence (pg. 12). Looking at the pseudocode that summarizes the NTM's operations in the copy task, we can see no other operations needing content-based addressing. Therefore, we can simplify the NTM specifically for the copy task by having a hard-coded weight vector that lets the NTM jump to the start of the sequence (the vector is of the form [1, 0, 0, 0, …]) and this vector is then interpolated as before with the weights from the previous step for the NTM to determine whether it needs to jump or just move along the sequence. By replicating the part of content-based addressing

needed for the copy task, the NTM doesn't need to learn the key or key weighting vectors and instead only needs to learn when to use the gated interpolation on a precise jump to the start.

3b. In `head.py` we can remove all code pertaining to `key` and `beta`. In `build_head_curr` in `model.py` we change:

```
key,beta,g,shift,gamma,erase,add = head(input_curr)

# 3.3.1 Focusing b Content
weight_c = U.vector_softmax(beta * similarity(key,M_curr))
```

to

```
g,shift,gamma,erase,add = head(input_curr)

weight_c = np.zeros((mem_size,))
weight_c[0] = 1
weight_c = theano.shared(weight_c)
```

After the content-based addressing mechanism was replaced, the NTM was trained for 50,000 episodes and the trained model was evaluated on the following sequence lengths: 10, 20, 50 and 120. The plots for the input, the expected output and the actual output are shown in Figures 7, 8 and 9:

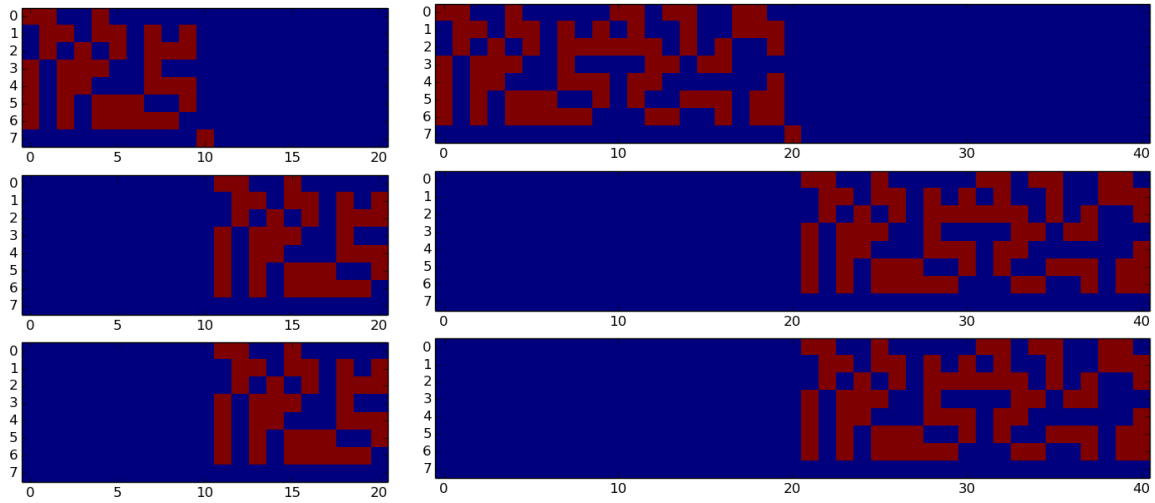

Figure 7: Test result for lengths 10 (left) and 20 (right), **without content-based addressing**

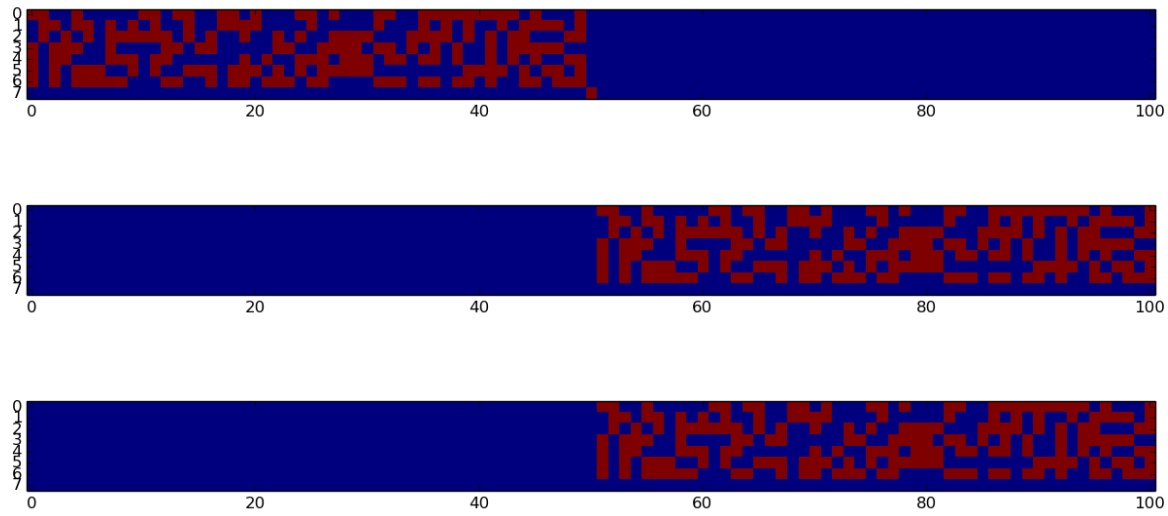

Figure 8: Test result for lengths 50, **without content-based addressing**

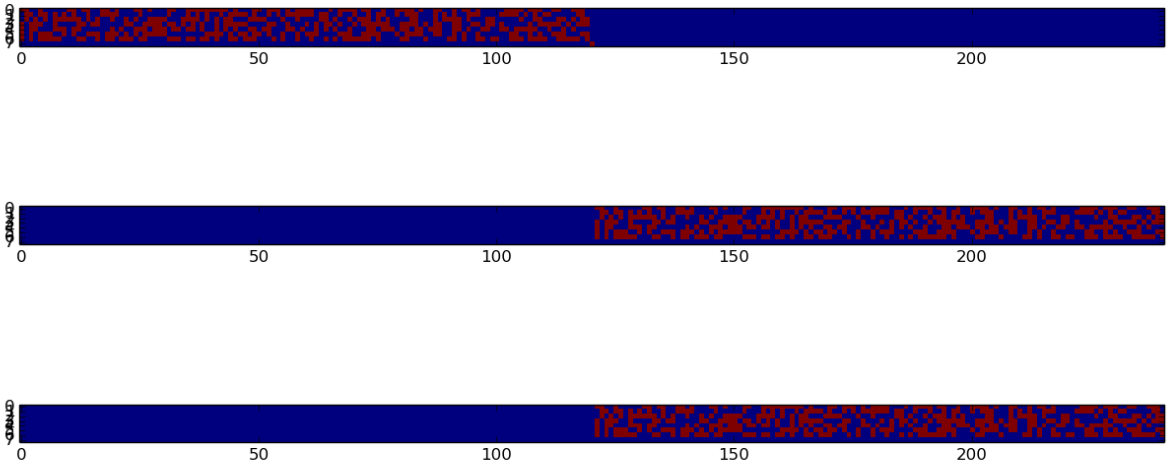

Figure 9: Test result for lengths 120, **without content-based addressing**

It is easy to see that after we replaced the content-addressing mechanism without changing any other parameter, the NTM successfully learned to perform the copy task for sequences even up to 120 units. This is because the new addressing mechanism helps the read/write head jump back to the very beginning of the sequence without needing to learn content-addressing mechanism. The average error rates are summarized in Table 3:

Table 3: Average error rate with and without content-based addressing

Sequence Length	Error Rate (**with content addressing**)	Error Rate (**without content addressing**)
10	0.0163%	0.0136%

20	0.014%	0.0137%
50	0.0139%	0.0133%
120	29%	0.0139%

c. The input sequence, the expected output sequence and the actual output sequence are plotted for a sequence length of 150 in Figure 10.

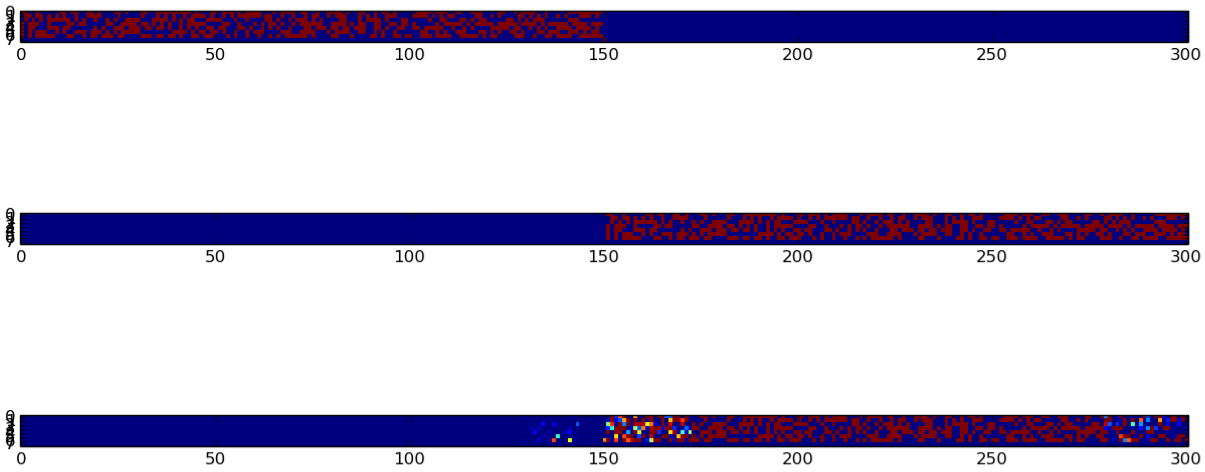

Figure 10: Test result for the NTM with sequences of lengths 150

The actual output matches the expected output for the majority of the sequence. However, they differ significantly around the first 20 columns and the last 20 columns of the sequence. The reason for this abnormal behavior is that the length of the memory is only 128, while the length of the sequence to copy is 150. For smaller sequences, the NTM learns to write each column of the input sequence to memory sequentially, and reads all of them sequentially afterward. When the sequence length exceeds the memory's length, the NTM's addressing mechanism will go back to the start of the memory after exhausting the memory, thus corrupting both the start of the sequence and the end of the sequence.

4.

We did the following experiments on the NTM model from the end of question 3 (with a hard-coded content addressing vector):

Increasing the memory size from 128 to 200:

We found that by increasing the memory size, the NTM was still able to successfully learn how to use this increased memory to learn to copy longer sequences. We saw that the NTM was successfully able to learn to copy sequences of length 150 (which it was unable to before with a memory of 128) as well as 200 (see Figures 11 and 12). We still find as expected though that once the sequence length exceeds the memory capacity (see Figure 13)., we start having errors similar to those described in Part 2c.

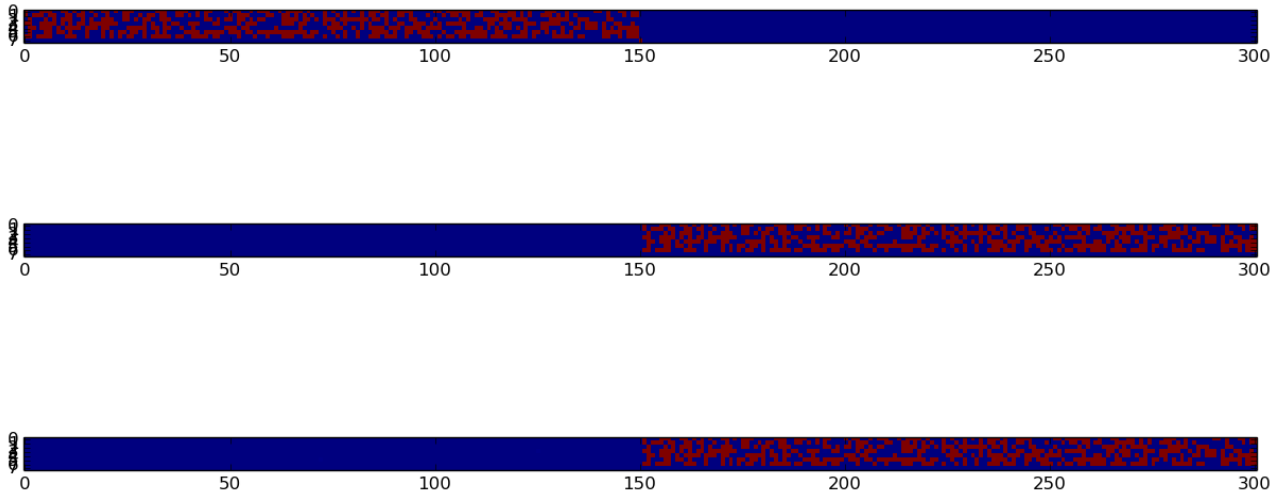

Figure 11: Test result for length 150, **with increased memory size**

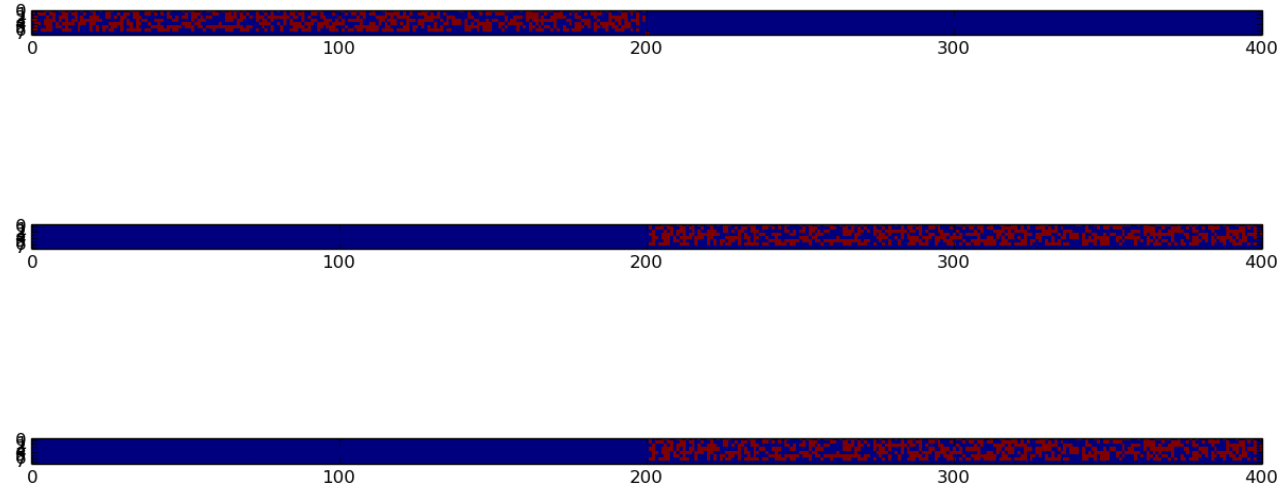

Figure 12: Test result for length 200, **with increased memory size**

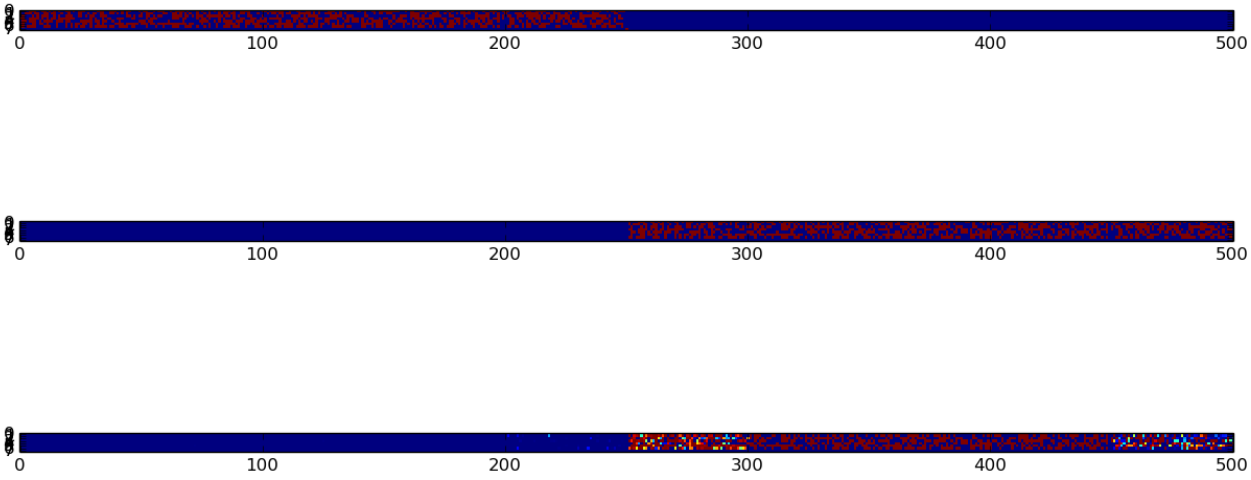

Figure 13: Test result for length 250, **with increased memory size**. The NTM fails to copy the beginning and the end of the sequence because its length exceeds the memory size.

Changing the update rule:
The current update rule uses adadelta with momentum (alpha=0.95).
Changing to rmsprop, we found that the NTM consistently had a higher cost function and was unable to learn to copy even for a sequence of length 10.
Changing to adadelta with less momentum (alpha=0.5), we found that the NTM performed significantly worse than adadelta with alpha=0.95 but still better than rmsprop. The average error rates of the NTM with the two momentum coefficients are compared in Table 4:

Table 4: Average error rate with alpha = 0.95 and alpha = 0.5

Sequence Length	Error Rate (**alpha = 0.95**)	Error Rate (**alpha = 0.5**)
10	0.0136%	0.0887%
20	0.0137%	0.0913%
50	0.0133%	0.1017%
120	0.0139%	0.133%

Changing the maximum length sequence seen during training:
When training, we currently give the NTM an input sequence up to length 20. Changing the max length to 10, we observed slightly higher error rates but still very good performance on the copy task. Despite having only seen an input-output pair of length

10, the NTM was still able to generalize to copying sequences of length 120 with little error. Changing the max length to 5, we observed significantly higher error rates at the start of each copied sequence for input sequences of length 50 and higher, but was surprisingly still able to generalize pretty well up to sequences of 120 after some errors at the sequence start (e.g. Figure 14). We believe that this may be due to the NTM not having fully generalized when to jump to the start of the sequence (using the hard-coded start weight vector) during the interpolation step from seeing such short input sequences, while having learned to generalize the sequential copy operation since the sequential copy operation is very simple and can easily be learned even from seeing many short sequences. The average error rates of both scenarios are shown in Table 5 below:

Table 5: Average error rate with different max lengths

Sequence Length	Error Rate (**max length = 20**)	Error Rate (**max length = 10**)	Error Rate (**max length = 5**)
10	0.0136%	0.0279%	0.0621%
20	0.0137%	0.0295%	0.114%
50	0.0133%	0.0307%	0.506%
120	0.0139%	0.0296%	0.552%

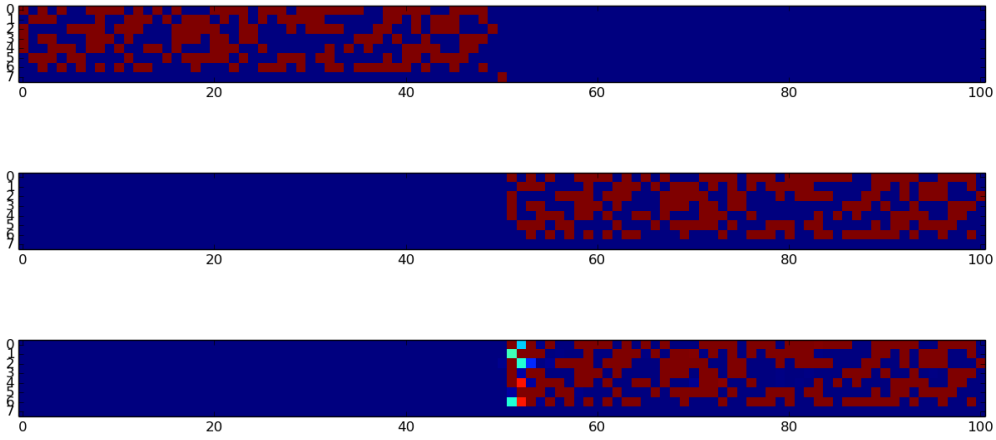

Figure 13: Test result for length 50, **with maximum training sequence length = 5**. The NTM produces noticeable errors the beginning of the sequence.

Changing training method: from curriculum learning to combined curriculum learning (with probability p=0.1 for each input, give a random length sequence rather than the

length as perscribed by regular curriculum learning, based off of method described in http://arxiv.org/abs/1410.4615)

```
if np.random.random() > 0.1:
        length = np.random.randint(int(20 * (min(counter,25000)/float(25000))**2) +1) + 1
else:
        length = int(np.random.random()*20)+1
```

We found this method of curriculum learning to perform significantly worse than the standard curriculum learning procedure, unable to even learn to copy a input sequence of length 10, as shown in Figure 14. It is interesting to note that the NTM was able to learn that the first half of the output sequence should be zeros and that the start of the output sequence is special compared to the rest.

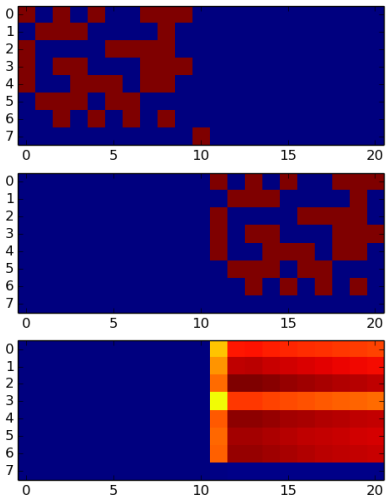

Figure 14: Test result for length 10, **with a combined curriculum training method**

Changing controller architecture:

We added a second hidden layer to the controller network with 100 hidden units. We found that performance was significantly worse, but from looking at the plots in Figure 15 and Figure 16, the NTM is clearly learning to generalize an incorrect interpretation of the copy task by outputting alternating vectors of all 1s and vectors of all 0s. We hypothesize that this may be due to us introducing 10,000 extra parameters with the additional hidden layer, making it much harder to train the network and find an appropriate setting for all the parameters for the copy task.

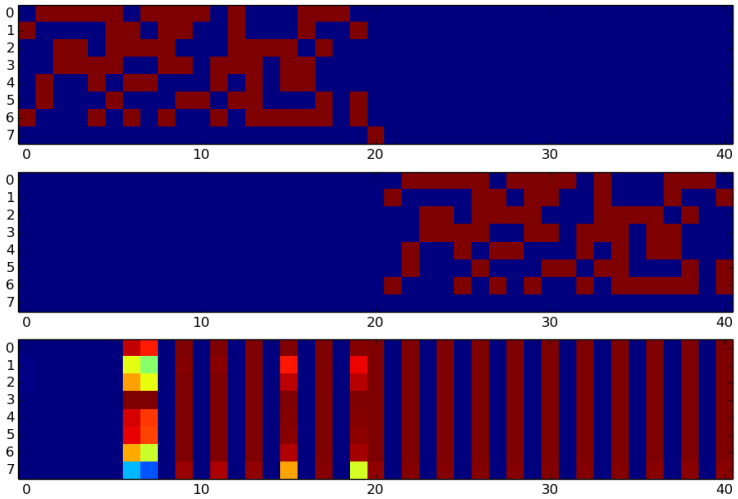

Figure 15: Test result for length 20, **with a second hidden layer**

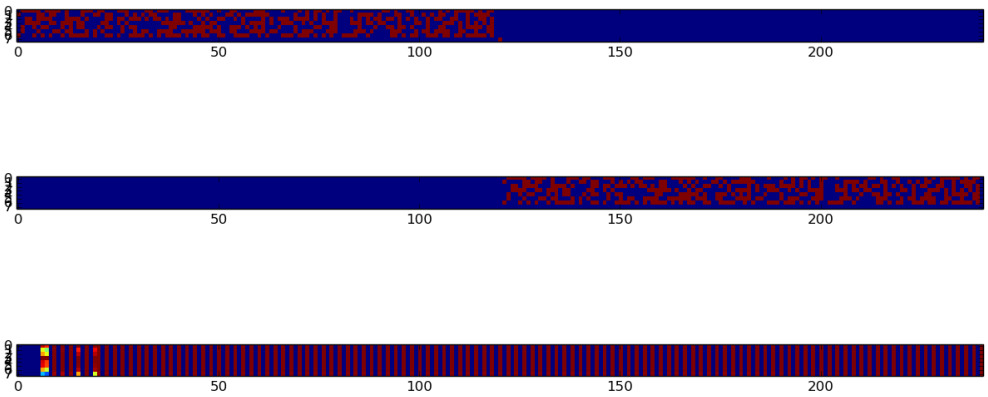

Figure 16: Test result for length 150, **with a second hidden layer**

Appendix

Notes:
Full code for each question can be found online at https://github.com/alee101/598c-project/.
The start code can be found on branch ‘startcode’.
The solution up to the end of question 2a can be found on branch ‘nosharpening’.
The solution up to the end of question 2b can be found on branch ‘master’.
The solution up to the end of question 3 can be found on branch ‘nocontentaddressing’.

Code was adapted from https://github.com/shawntan/neural-turing-machines, simplified, and annotated.

Contributions:
Both of us worked on understanding, simplifying, and annotating the existing solution. Albert wrote up most of the report and Tom generated all the plots and tables for each part.