Exploring the Capabilities and Possible Applications of Neural Turing Machines

Ryan Cooper
Computer Science and Engineering
New Mexico Institute of Mining and Technology
Socorro, New Mexico
Email: ryan.cooper@student.nmt.edu

Brandon Fleming
Computer Science and Engineering
New Mexico Institute of Mining and Technology
Socorro, New Mexico
Email: brandon.fleming@student.nmt.edu

***Abstract*—The neural turing machines (NTM) is a class of learner that was introduced in 2014 by Google DeepMind. The NTM adds a "working memory" to the computational unit in a traditional artificial neuron, essentially causing the neuron to not only act on the input provided to the neuron, but also acting as a controller to its own working memory set. The NTM has been shown to not only solve turing-problems, but is hypothesized to be a super-turing approximation model [1]. The NTM has spawned significant research in memory-augmented computing and allows classical deep-learning to be applied to algorithmic processes. We explore open-source implementations of NTMs and analyze the extent of these capabilities, comparing these to the shortcomings of classical memory models like recurrent LSTM and GRU.**

***Index Terms*—Machine Learning, Neural Turing Machine, Python, Recurrent Neural Networks**

I. Introduction

A. Classical Machine Learning Limitations

Neumann defines computation programs that are constructed based on three fundamental mechanisms [1]:

1) Initial operations (e.g. arithmetic)
2) Logical flow control
3) External memory

With respect to the success made in complicated data modeling, machine learning usually applies logical flow control by ignoring the external memory. Here, RNNs networks outperform other learning machine methods with a learning capability. Moreover, it is obvious that RNNs, are Turing-Complete[3] and provided that they are formatted in a correct manner, they would be able to simulate different methods. Any advance in RNNs capabilities can provide solutions for algorithmic tasks by applying a big memory. RNNs utilize bidirectional associative memory [2] in order to maintain temporal (or spatial) relationships between input, and as such, lend themselves for comparison to an explicit memory model.

B. Introduction to Neural Turing Machines

Neural Turing Machines (NTMs) were originally introduced by Alex Graves in 2014 [3]. The basic premise of this model type is that rather than the neural nodes acting on the data directly, the model learns how to interact with input on an external memory (tape), similar to in an actual Turing machine. This architecture is visualized in *Figure 1*. For the purpose of the tests described in this paper, the controller is exclusively recurrent. Graves presents an argument for the validity of recurrent controllers in the initial paper, and as such, we felt it unnecessary to explore the differences between feed-forward and recurrent controllers.

Neural Turing Machines pose an interesting solution to problems – rather than learning a black-box model, why not fit a well understood branch of computing to it instead. Moreover, this architecture lends itself to learn fundamental algorithms (sequence of operations / procedure) rather than pure input relationships.

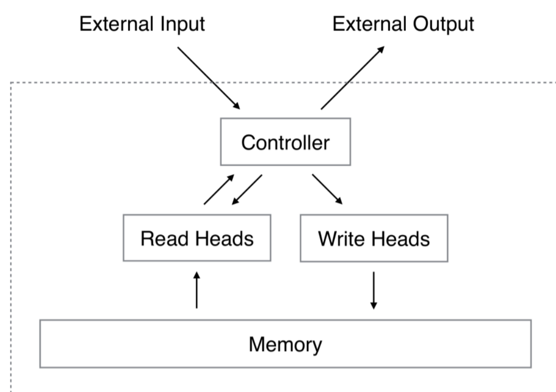

Fig. 1. Basic architecture of Neural Turing Machines

II. Approach

For the purpose of this project, we wished to explore the effects of the application of the external memory module on a procedural task and attempt to rationalize the abstraction that we observe. Two problems are used to realize this goal; the copy task and basic binary numeracy.

A. Copy Task

In the paper by Graves [3], one of the original dataset that they used to show the capabilities of the NTM model is the copy task. The copy task is a function such that returns the original input, that is, the identity function. In classical Turing machines, this is a trivial model, as shown in *Figure 2*. The copy task takes in a set of n bytes, this represents the data to be copied and outputs a set of n bytes.

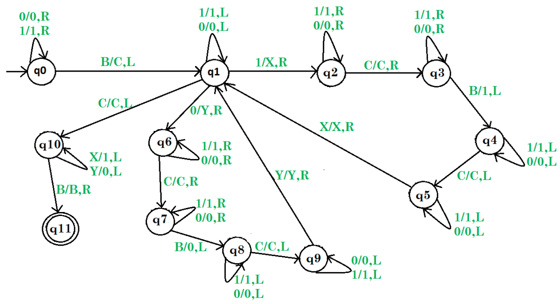

Fig. 2. Turing machine of Copy Task

B. Binary Numeracy

More recent developments in the field of NTMs is shown by a paper implementing basic binary numeracy [4]. We attempt to replicate their results by the addition of a sequence of n, k-sized binary strings, with no consideration for overflow. This allows us to see how the NTM scales with the addition of more bit strings in the sequence, to show generalizability. The data generator is given in *Appendix III*.

C. Implementation

As much of this research is relatively new, we adapted some open-source libraries to suit our needs and developed two classes on top of TensorFlow's `NTMCell` class (See *Appendix I*). The first of the classes is `NTMCopyModel`, which implements the copy model architecture in both LSTM and NTM fundamental cells in Tensorflow. For the memory, a reusable external list is used for the controller to act on. The input and output shapes are identical. This implementation is given in *Appendix II*.

Secondly, the `NTMNumeracyModel` is defined very similarly, but with the output shape varying to `(batch_size, 1, vector_size)` while the input remains as `(batch_size, sequence_length, vector_size)`, as in the copy model. This class can be seen in *Appendix III*.

III. Results

A. Copy Task

Trains were made for the copy task using a few different sequence lengths and epochs. Each attempt was made on CPU rather than with GPU acceleration due to the RAM intensive properties of the model.

Early training attempts were run over a max length of 8 bits, with a total of twenty thousand epochs. This often converged well as shown in *Figure 3*, but failed to generalize the overall algorithm and rather. This was revealed through testing as it showed that the underlying algorithm failed to learn and thus, a higher training sequence length was necessary.

The settings we found optimal to learn this algorithm was when trained over one hundred thousand epochs with a max

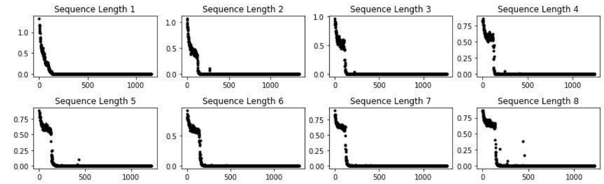

Fig. 3. Training over a max sequence length of 8

sequence length of 16, with each vector being 8 bits (1 byte) long. The loss of this training is shown in *Figure 4*. These graphs show that this particular session converged rather quickly in the training process over one hundred thousand epochs. We determine through cross-validation that the model was not overfit, but rather converged to a local minimum in terms of loss as seen through *Figure 5*. This graph shows that even as we increase the sequence length well above that untrained number, the algorithm shows nearly zero loss over 100 samples of testing.

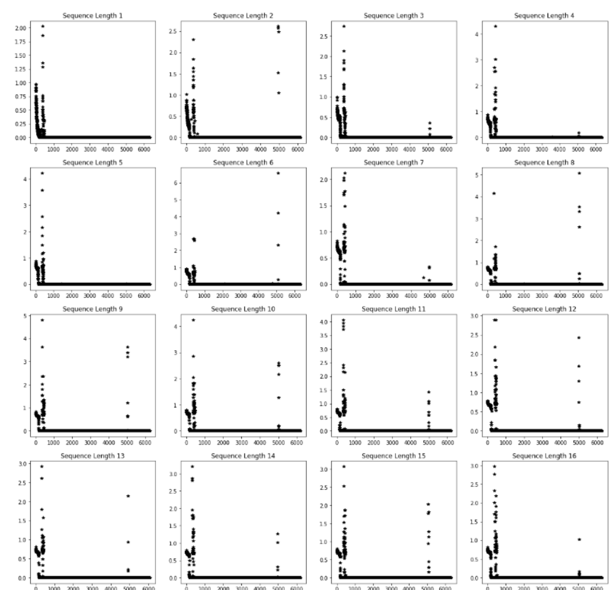

Fig. 4. Graph of copy task training loss over various training sequence lengths, up to sequence length of 16 with NTM

As a reference, an equivalently sized LSTM model (when compared to the controller) as trained over an equivalent number of epochs using the same procedural data generation. The training loss of this model can be seen in *Figure 6* and the testing loss is seen in *Figure 7*.

B. Basic Numeracy

For the numeracy task, we focused on binary addition over n, k-sized binary strings, with no consideration for overflow. Initially, the training max sequence size was set to 8, that is, $n = 8, k = 8$. This task is fairly straight forward in Von Neumann architecture as it is just simple iteration, but we observed that both NTM and LSTM struggled with a controller topology of 128 fully recurrent nodes densely connected in 3

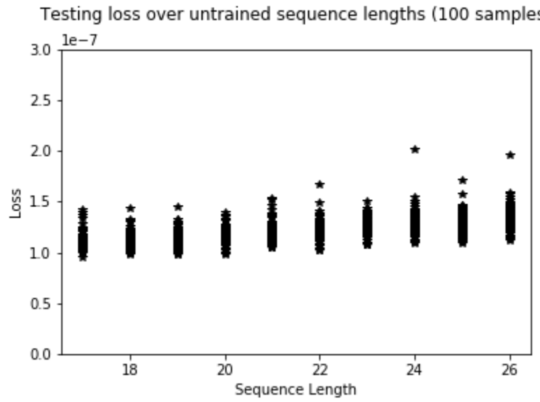

Fig. 5. Graph of copy task test loss over various unseen testing sequence lengths, up to sequence length of 27 with NTM

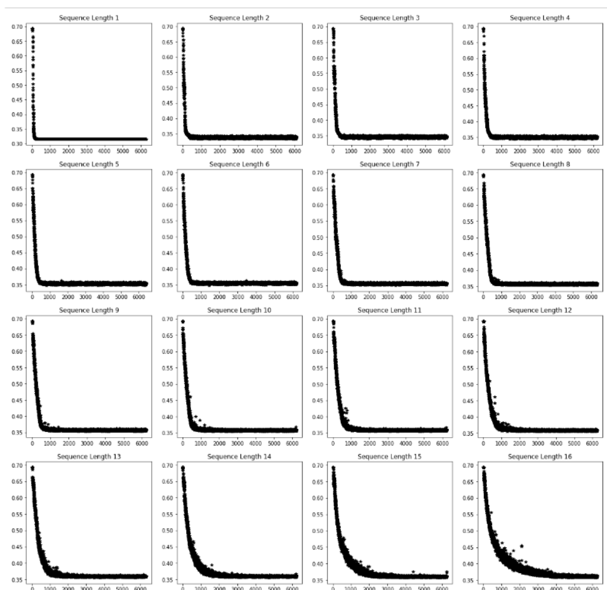

Fig. 6. Graph of copy task training loss over various training sequence lengths, up to sequence length of 16 with LSTM

layers. The training loss is seen in *Figure 8*, and the same type of cross validation test loss is seen in *Figure 9*, going up to a max sequence of 12.

We observe the same, but worse for LSTM.

IV. Discussion and Conclusion

Training with a sequence length of 8 performed very well over most data including sequences of length 9, but failed to generalize to larger sequences. Some data sequences such as all zeroes or half zeroes performed badly as well, revealing that the model failed to generalize training set. With this in mind, we considered expanding the max sequence length up to 16. As shown in the results, this helped to generalize the algorithm and solved the edge cases where the dataset was homogeneous. This can be seen in *Figures 10 and 11* as the data and output are equivalent, for both all ones and all zeros respectively.

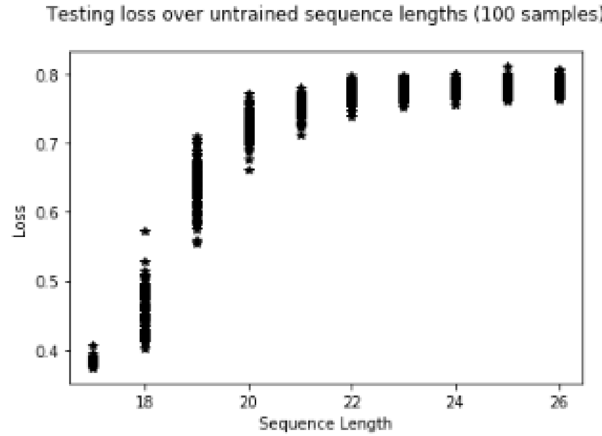

Fig. 7. Graph of copy task test loss over various unseen testing sequence lengths, up to sequence length of 27 with LSTM

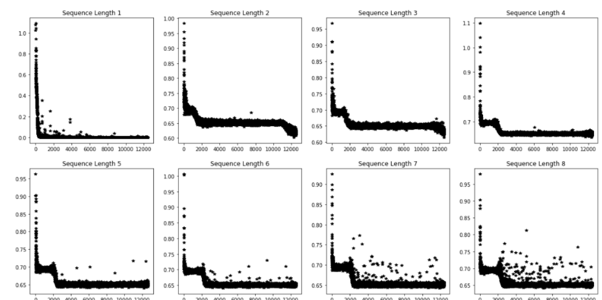

Fig. 8. Loss over training sequence lengths in numeracy training in numeracy example

This model was then tested with unseen sequence lengths up to size 27, which is 11 longer than was originally trained on. As was shown in the results, this model was able to generalize well.

This generalizability will allow us to directly compare the NTM to the LSTM in order to analyze differences.

While the LSTM trained quicker than the NTM, average final training loss is significantly higher in the LSTM (0.39) model than the NTM (9.29×10^{-8}) model for all sequence lengths. As shown in the results for the LSTM model, it failed to generalize the underlying algorithm, but moreover, even the dataset as a whole. This is likely due to the size of the RNN itself. In the NTM, the controller was 128 fully recurrent nodes densely connected in 3 layers. This network topology as an LSTM is likely too small to generalize this complex relationship. This may be the case, but illustrates the point exactly: the addition of external memory in a similarly-sized network allows it to focus on procedural retention rather than be bothered by numeric values (in the case of the copy task). This not only shows the capabilities NTM, but opens the door for learning of underlying algorithms or procedures, rather than nonlinear relationships between input and output.

These results illustrate the advantage that NTM models have over Recurrent Neural Networks, and even more evidently

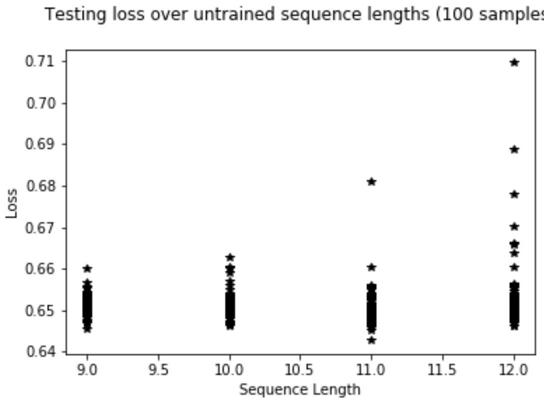

Fig. 9. Shows the loss growth as we expand the unseen scope of the test sequence length in numeracy example

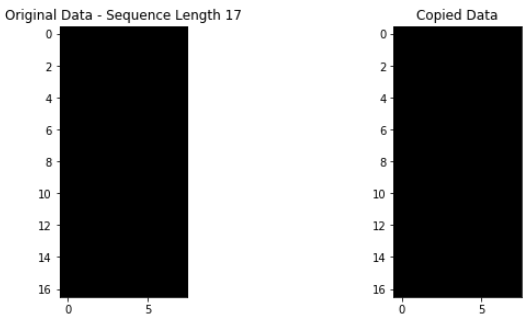

Fig. 10. Output when training with all ones (note, coloring is dynamic, so since the two images match, the input and output are identical)

when the loss is viewed in depth. As can be seen in *Figure 6*, convergence occurs rather quickly and the LSTM model trains in a shorter time when compared to the NTM model. A major difference is shown in the convergence value, as even over several training iterations, the loss for LSTM never improved past 0.3. The LSTM model converges quickly to a loss value much higher than that of the NTM, around 0.38 over 4 tests, at approximately 7 orders of magnitude different. As such, the merits of an NTM model have been illustrated by performance against the modern RNNs. The primary difference is the bidirectional associative memory [2] that exists in RNNs versus the explicit memory in the NTM architecture. In an RNN model, a form of memory is introduced to the system by a recurrent connection. This connection can alter its weight so the previous state alters the next and induces a memory upon the system. In a NTM however, a model can store data directly in a memory area, allowing usage of a true memory between iterations. Through experimentation, this difference shows the loss to converge at a much lower value than a traditional RNN. By comparing *Figure 7* to *Figure 7*, it is shown that NTM can sustain loss on tests with values outside of the testing length showing successful learning of copy task, whereas the LSTM model loss grows steadily after expanding past the training set.

As for the numeracy task, we speculate that the issue of

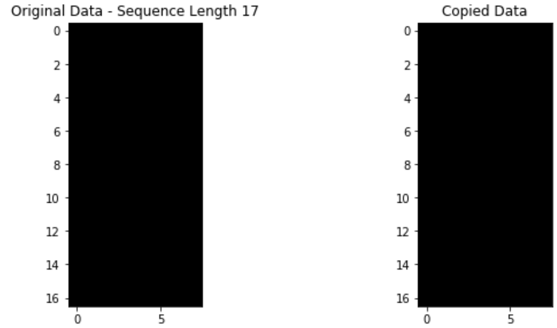

Fig. 11. Output when training with all zeros (note, coloring is dynamic, so since the two images match, the input and output are identical)

non-convergence is derived from the size of both the recurrent controller and the memory not being large enough in the NTM. We would test larger sizes, but our hardware was maxed out as is; so this will need to be explored more on more powerful machines in the future.

A. Future Work

An immediate question arises with the legitimacy of the NTM architecture - can the underlying Turing machine be extracted from the distributed NTM controller representation. At first, one may hastily say no, however, it would reason to stand that extraction is possible through enumeration of all possible input values of the machine and define a language $\mathcal{L}$ for the machine $\mathcal{T}$. In order to reconstruct the controller automaton, a dense graph would be generated, using ϵ-transitions to each of the input values, handling each input value as a separate case. This automaton could then be reduced to simple form. Much more research needs to be done on this however, as this is all speculation.

References

[1] J. V. Neumann, “First draft of a report on the edvac,” 1945.
[2] B. Kosko, “Bidirectional associative memories,” *IEEE Trans. Syst. Man Cybern.*, vol. 18, no. 1, pp. 49–60, Jan. 1988. [Online]. Available: http://dx.doi.org/10.1109/21.87054
[3] A. Graves, G. Wayne, and I. Danihelka, “Neural turing machines,” 2014.
[4] J. Castellini, “Learning numeracy: Binary arithmetic with neural turing machines,” 2019.
[5] S. M. Faradonbeh and F. Safi-Esfahani, “A review on neural turing machine,” 2019.
[6] M. Collier and J. Beel, “Implementing neural turing machines,” 2018.
[7] GeeksforGeeks, “Turing machine for copying data,” Jun 2018. [Online]. Available: https://www.geeksforgeeks.org/turing-machine-for-copying-data/

Appendix

I. NTM Cell [6]

```python
class NTMCell():
    def __init__(self, rnn_size, memory_size, memory_vector_dim, read_head_num, write_head_num,
                 addressing_mode='content_and_loaction', shift_range=1, reuse=False, output_dim=None):
        self.rnn_size = rnn_size
        self.memory_size = memory_size
        self.memory_vector_dim = memory_vector_dim
        self.read_head_num = read_head_num
        self.write_head_num = write_head_num
        self.addressing_mode = addressing_mode
        self.reuse = reuse
        self.controller = tf.nn.rnn_cell.BasicRNNCell(self.rnn_size)
        self.step = 0
        self.output_dim = output_dim
        self.shift_range = shift_range

    def __call__(self, x, prev_state):
        prev_read_vector_list = prev_state['read_vector_list']      # read vector in Sec 3.1 (the content that is
                                                                    # read out, length = memory_vector_dim)
        prev_controller_state = prev_state['controller_state']      # state of controller (LSTM hidden state)

        # x + prev_read_vector -> controller (RNN) -> controller_output
        controller_input = tf.concat([x] + prev_read_vector_list, axis=1)
        with tf.variable_scope('controller', reuse=self.reuse):
            controller_output, controller_state = self.controller(controller_input, prev_controller_state)

        # controller_output     -> k (dim = memory_vector_dim, compared to each vector in M, Sec 3.1)
        #                       -> beta (positive scalar, key strength, Sec 3.1)                -> w^c
        #                       -> g (scalar in (0, 1), blend between w_prev and w^c, Sec 3.2)  -> w^g
        #                       -> s (dim = shift_range * 2 + 1, shift weighting, Sec 3.2)      -> w^~
        #                            (not memory_size, that's too wide)
        #                       -> gamma (scalar (>= 1), sharpen the final result, Sec 3.2)     -> w    * num_heads
        # controller_output     -> erase, add vector (dim = memory_vector_dim, \in (0, 1), Sec 3.2)     * write_head_num

        num_parameters_per_head = self.memory_vector_dim + 1 + 1 + (self.shift_range * 2 + 1) + 1
        num_heads = self.read_head_num + self.write_head_num
        total_parameter_num = num_parameters_per_head * num_heads + self.memory_vector_dim * 2 * self.write_head_num
        with tf.variable_scope("o2p", reuse=(self.step > 0) or self.reuse):
            o2p_w = tf.get_variable('o2p_w', [controller_output.get_shape()[1], total_parameter_num],
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))
            o2p_b = tf.get_variable('o2p_b', [total_parameter_num],
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))
            parameters = tf.nn.xw_plus_b(controller_output, o2p_w, o2p_b)
        head_parameter_list = tf.split(parameters[:, :num_parameters_per_head * num_heads], num_heads, axis=1)
        erase_add_list = tf.split(parameters[:, num_parameters_per_head * num_heads:], 2 * self.write_head_num, axis=1)

        # k, beta, g, s, gamma -> w

        prev_w_list = prev_state['w_list']  # vector of weightings (blurred address) over locations
        prev_M = prev_state['M']
        w_list = []
        p_list = []
        for i, head_parameter in enumerate(head_parameter_list):

            # Some functions to constrain the result in specific range
            # exp(x)                -> x > 0
            # sigmoid(x)            -> x \in (0, 1)
            # softmax(x)            -> sum_i x_i = 1
            # log(exp(x) + 1) + 1   -> x > 1

            k = tf.tanh(head_parameter[:, 0:self.memory_vector_dim])
            beta = tf.sigmoid(head_parameter[:, self.memory_vector_dim]) * 10        # do not use exp, it will explode!
            g = tf.sigmoid(head_parameter[:, self.memory_vector_dim + 1])
            s = tf.nn.softmax(
                head_parameter[:, self.memory_vector_dim + 2:self.memory_vector_dim + 2 + (self.shift_range * 2 + 1)]
            )
            gamma = tf.log(tf.exp(head_parameter[:, -1]) + 1) + 1
            with tf.variable_scope('addressing_head_%d' % i):
                w = self.addressing(k, beta, g, s, gamma, prev_M, prev_w_list[i])     # Figure 2
            w_list.append(w)
            p_list.append({'k': k, 'beta': beta, 'g': g, 's': s, 'gamma': gamma})

        # Reading (Sec 3.1)

        read_w_list = w_list[:self.read_head_num]
        read_vector_list = []
        for i in range(self.read_head_num):
            read_vector = tf.reduce_sum(tf.expand_dims(read_w_list[i], dim=2) * prev_M
```

```
, axis=1)
            read_vector_list.append(read_vector)

        # Writing (Sec 3.2)

        write_w_list = w_list[self.read_head_num:]
        M = prev_M
        for i in range(self.write_head_num):
            w = tf.expand_dims(write_w_list[i], axis=2)
            erase_vector = tf.expand_dims(tf.sigmoid(erase_add_list[i * 2]), axis=1)
            add_vector = tf.expand_dims(tf.tanh(erase_add_list[i * 2 + 1]), axis=1)
            M = M * (tf.ones(M.get_shape()) - tf.matmul(w, erase_vector)) + tf.matmul(w, add_vector)

        # controller_output -> NTM output

        if not self.output_dim:
            output_dim = x.get_shape()[1]
        else:
            output_dim = self.output_dim
        with tf.variable_scope("o2o", reuse=(self.step > 0) or self.reuse):
            o2o_w = tf.get_variable('o2o_w', [controller_output.get_shape()[1], output_dim],
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))
            o2o_b = tf.get_variable('o2o_b', [output_dim],
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))
            NTM_output = tf.nn.xw_plus_b(controller_output, o2o_w, o2o_b)

        state = {
            'controller_state': controller_state,
            'read_vector_list': read_vector_list,
            'w_list': w_list,
            'p_list': p_list,
            'M': M
        }

        self.step += 1
        return NTM_output, state

    def addressing(self, k, beta, g, s, gamma, prev_M, prev_w):

        # Sec 3.3.1 Focusing by Content

        # Cosine Similarity

        k = tf.expand_dims(k, axis=2)
        inner_product = tf.matmul(prev_M, k)
        k_norm = tf.sqrt(tf.reduce_sum(tf.square(k), axis=1, keep_dims=True))
        M_norm = tf.sqrt(tf.reduce_sum(tf.square(prev_M), axis=2, keep_dims=True))
        norm_product = M_norm * k_norm
        K = tf.squeeze(inner_product / (norm_product + 1e-8))                   # eq (6)

        # Calculating w^c

        K_amplified = tf.exp(tf.expand_dims(beta, axis=1) * K)
        w_c = K_amplified / tf.reduce_sum(K_amplified, axis=1, keep_dims=True)  # eq (5)

        if self.addressing_mode == 'content':                                   # Only focus on content
            return w_c

        # Sec 3.3.2 Focusing by Location

        g = tf.expand_dims(g, axis=1)
        w_g = g * w_c + (1 - g) * prev_w                                         # eq (7)

        s = tf.concat([s[:, :self.shift_range + 1],
                       tf.zeros([s.get_shape()[0], self.memory_size - (self.shift_range * 2 + 1)]),
                       s[:, -self.shift_range:]], axis=1)
        t = tf.concat([tf.reverse(s, axis=[1]), tf.reverse(s, axis=[1])], axis=1)
        s_matrix = tf.stack(
            [t[:, self.memory_size - i - 1:self.memory_size * 2 - i - 1] for i in range(self.memory_size)],
            axis=1
        )
        w_ = tf.reduce_sum(tf.expand_dims(w_g, axis=1) * s_matrix, axis=2)      # eq (8)
        w_sharpen = tf.pow(w_, tf.expand_dims(gamma, axis=1))
        w = w_sharpen / tf.reduce_sum(w_sharpen, axis=1, keep_dims=True)        # eq (9)

        return w

    def zero_state(self, batch_size, dtype):
        def expand(x, dim, N):
            return tf.concat([tf.expand_dims(x, dim) for _ in range(N)], axis=dim)

        with tf.variable_scope('init', reuse=self.reuse):
            state = {
                # 'controller_state': self.controller.zero_state(batch_size, dtype),
                # 'read_vector_list': [tf.zeros([batch_size, self.memory_vector_dim])
                #                      for _ in range(self.read_head_num)],
                # 'w_list': [tf.zeros([batch_size, self.memory_size])
                #            for _ in range(self.read_head_num + self.write_head_num)],
                # 'M': tf.zeros([batch_size, self.memory_size, self.memory_vector_dim])
                'controller_state': expand(tf.tanh(tf.get_variable('init_state', self.rnn_size,
                                    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))),
                                    dim=0, N=batch_size),
                'read_vector_list': [expand(tf.nn.softmax(tf.get_variable('init_r_%d' % i, [self.memory_vector_dim],
                                    initializer=tf.random_normal_initializer(
```

```
          mean=0.0, stddev=0.5))),
                                                 dim=0, N=
          batch_size)
                                     for i in range(self.
          read_head_num)],
                    'w_list': [expand(tf.nn.softmax(
          tf.get_variable('init_w_%d' % i, [self.
          memory_size],

          initializer=tf.random_normal_initializer(
          mean=0.0, stddev=0.5))),
                                                 dim=0, N=
          batch_size) if self.addressing_mode == '
          content_and_loaction'
                                     else tf.zeros([
          batch_size, self.memory_size])
                                     for i in range(self.
          read_head_num + self.write_head_num)],
                    'M': expand(tf.tanh(tf.
          get_variable('init_M', [self.memory_size,
          self.memory_vector_dim],

          initializer=tf.random_normal_initializer(
          mean=0.0, stddev=0.5))),
                                                 dim=0, N=
          batch_size)
                  }
              return state
```

II. NTM Copy Task Model

```
class NTMCopyModel():
    def __init__(self, args, seq_length, reuse=
    False):
        self.x = tf.placeholder(name='x', dtype=
    tf.float32, shape=[args.batch_size,
    seq_length, args.vector_dim])
        self.y = self.x
        eof = np.zeros([args.batch_size, args.
    vector_dim + 1])
        eof[:, args.vector_dim] = np.ones([args.
    batch_size])
        eof = tf.constant(eof, dtype=tf.float32)
        zero = tf.constant(np.zeros([args.
    batch_size, args.vector_dim + 1]), dtype=tf
    .float32)

        if args.model == 'LSTM':
            def rnn_cell(rnn_size):
                return tf.nn.rnn_cell.
    BasicLSTMCell(rnn_size, reuse=reuse)
            cell = tf.nn.rnn_cell.MultiRNNCell([
    rnn_cell(args.rnn_size) for _ in range(args
    .rnn_num_layers)])
        elif args.model == 'NTM':
            cell = NTMCell(args.rnn_size, args.
    memory_size, args.memory_vector_dim, 1, 1,

    addressing_mode='content_and_location',
                                    reuse=reuse,
                                    output_dim=
    args.vector_dim)

        state = cell.zero_state(args.batch_size,
     tf.float32)
        self.state_list = [state]
        for t in range(seq_length):
            output, state = cell(tf.concat([self
    .x[:, t, :], np.zeros([args.batch_size, 1])
    ], axis=1), state)
            self.state_list.append(state)
        output, state = cell(eof, state)
        self.state_list.append(state)

        self.o = []
        for t in range(seq_length):
            output, state = cell(zero, state)
            self.o.append(output[:, 0:args.
    vector_dim])
            self.state_list.append(state)
        self.o = tf.sigmoid(tf.transpose(self.o,
     perm=[1, 0, 2]))

        eps = 1e-8
        self.copy_loss = -tf.reduce_mean(   #
    cross entropy function
            self.y * tf.log(self.o + eps) + (1 -
     self.y) * tf.log(1 - self.o + eps)
        )
        with tf.variable_scope('optimizer',
    reuse=reuse):
            self.optimizer = tf.train.
    RMSPropOptimizer(learning_rate=args.
    learning_rate, momentum=0.9, decay=0.95)
            gvs = self.optimizer.
    compute_gradients(self.copy_loss)
            capped_gvs = [(tf.clip_by_value(grad
    , -10., 10.), var) for grad, var in gvs]
            self.train_op = self.optimizer.
    apply_gradients(capped_gvs)
        self.copy_loss_summary = tf.summary.
    scalar('copy_loss_%d' % seq_length, self.
    copy_loss)
```

III. NTM Numeracy Task

```
class NTMNumercyModel():
    def __init__(self, args, seq_length, reuse=
    False):
        self.x = tf.placeholder(name='x', dtype=
    tf.float32, shape=[args.batch_size,
    seq_length, args.vector_dim])
        self.y = tf.placeholder(name='y', dtype=
    tf.float32, shape=[args.batch_size, 1, args
    .vector_dim])
        eof = np.zeros([args.batch_size, args.
    vector_dim + 1])
        eof[:, args.vector_dim] = np.ones([args.
    batch_size])
        eof = tf.constant(eof, dtype=tf.float32)
        zero = tf.constant(np.zeros([args.
    batch_size, args.vector_dim + 1]), dtype=tf
    .float32)

        if args.model == 'LSTM':
            def rnn_cell(rnn_size):
                return tf.nn.rnn_cell.
    BasicLSTMCell(rnn_size, reuse=reuse)
            cell = tf.nn.rnn_cell.MultiRNNCell([
    rnn_cell(args.rnn_size) for _ in range(args
    .rnn_num_layers)])
        elif args.model == 'NTM':
            cell = NTMCell(args.rnn_size, args.
    memory_size, args.memory_vector_dim, 1, 1,

    addressing_mode='content_and_location',
                                    reuse=reuse,
                                    output_dim=
    args.vector_dim)

        state = cell.zero_state(args.batch_size,
     tf.float32)
        self.state_list = [state]
        for t in range(seq_length):
            output, state = cell(tf.concat([self
    .x[:, t, :], np.zeros([args.batch_size, 1])
    ], axis=1), state)
            self.state_list.append(state)
        output, state = cell(eof, state)
        self.state_list.append(state)
```

```python

        self.o = []
        for t in range(seq_length):
            output, state = cell(zero, state)
            self.o.append(output[:, 0:args.vector_dim])
            self.state_list.append(state)
        self.o = tf.sigmoid(tf.transpose(self.o, perm=[1, 0, 2]))

        eps = 1e-8
        self.copy_loss = -tf.reduce_mean(   # cross entropy function
            self.y * tf.log(self.o + eps) + (1 - self.y) * tf.log(1 - self.o + eps)
        )
        with tf.variable_scope('optimizer', reuse=reuse):
            self.optimizer = tf.train.RMSPropOptimizer(learning_rate=args.learning_rate, momentum=0.9, decay=0.95)
            gvs = self.optimizer.compute_gradients(self.copy_loss)
            capped_gvs = [(tf.clip_by_value(grad, -10., 10.), var) for grad, var in gvs]
            self.train_op = self.optimizer.apply_gradients(capped_gvs)
        self.copy_loss_summary = tf.summary.scalar('copy_loss_%d' % seq_length, self.copy_loss)


def bool2int(x):
    y = 0
    for i,j in enumerate(x):
        y += j<<i
    return y

def int2bool(x, seq_length):
    r = np.array([int(_) for _ in bin(x)[2:]]).astype(np.uint32)
    l = len(r)
    for _ in range(seq_length - l):
        r = np.insert(r, 0, 0.)
    return r

def compute_y(x):
    seq_length = x.shape[-1]
    y = []
    for row in x:
        r = [0 for _ in range(seq_length)]
        for entry in row:
            v = bool2int(entry[::-1])
            r = int2bool(v + bool2int(r[::-1]), seq_length)

        y.append([r[::-1][:seq_length][::-1]])
    return np.array(y, dtype=np.uint32)

def generate_data(batch_size, seq_length, vector_size):
    x = np.random.randint(0, 2, size=[batch_size, seq_length, vector_size]).astype(np.uint32)
    return x, compute_y(x)
```