Student Name: Rahul Deshmukh ¶

ECE 595 Machine Learning II

Project 5-part-2: NTM

You might need to modify the third line in the code cell below, to make sure you cd to the actual directory which your ipynb file is located in.

Caution: due to the nature of this project's setup, everytime you want to rerun some code cell below, please click **Runtime -> Restart and run all**; this operation clears the computational graphs and the local variables but allow training and testing data that are already loaded from google drive to stay in the colab runtime space. Please do **not** do the following if you just wish to rerun code: click Runtime -> reset all runtimes, and then click Runtime -> Run all; it will remount your google drive, and remove the training and testing data already loaded in your colab runtime space. **Runtime -> Restart and run all** automatically avoids remounting the drive after the first time you run the notebook file; the loaded data can usually stay in your colab runtime space for many hours.

Loading the training and testing data after remounting your google drive takes 30 - 40 minutes.

In [0]:

```
from google.colab import drive
drive.mount("/content/gdrive/", force_remount=True)
%cd gdrive/My Drive/ML2/Project-5/
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_
id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&re
direct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%
20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.goog
leapis.com%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.ph
otos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /content/gdrive/
/content/gdrive/My Drive/ML2/Project-5
```

In [0]:

```
from utils import OmniglotDataLoader, one_hot_decode, five_hot_decode
import tensorflow as tf
import argparse
import numpy as np
%tensorflow_version 1.x
print(tf.__version__)
```

The default version of TensorFlow in Colab will soon switch to TensorFlow 2.x.
We recommend you upgrade (https://www.tensorflow.org/guide/migrate) now or ensure your notebook will continue to use TensorFlow 1.x via the `%tensorflow_version 1.x` magic: more info (https://colab.research.google.com/notebooks/tensorflow_version.ipynb).

```
1.15.0
```

The following class `MANNCell` is the core of the memory-augmented neural network (MANN). You will implement the main parts of it in Tensorflow 2.0.

Before any technical discussion of how the MANNCell should operate, let us look at what it should do on a general level. Suppose we have an input batch of 16 episodes of image samples, with each episode being of equal length of 50. Based on the design of the rest of the project (which we have already implemented for you), MANNCell should be called 50 times, each time having 16 input samples (along with the offseted labels), and outputting 16 output labels. More specifically, the MANNCell should produce classification labels $[\hat{y}_0^t, \ldots, \hat{y}_{15}^t]$ for all 16 iteration-t image samples batch $[x_0^t + \text{null}, x_1^t + y_0^t, \ldots, x_{15}^t + y_{14}^t]$ ("+" means concatenation) every time it is called; for your information, it is the class NTMOneShotLearningModel (already implemented below) that actually calls MANNCell 50 times. Your job is to make sure that at a single iteration t (where $t = 0, 1, 2, \ldots, 49$), MANNCell correctly parses the input arguments, produce the correct read and write weights w_t^r, w_t^w, correctly retrieve from and write to the memory to form M_t, and use the right material to get the logits for classification (they will be used for computing the labels and cross-entropy values in NTMOneShotLearningModel), and return the right states that will be used in the next iteration $t + 1$.

Let us look at the input arguments of the method `call(self, inputs, states)` of this class first:

- The `inputs` variable shall have the following shape: `(batch_size, image_size+num_classes)`.
  - It corresponds to the $[x_0^t + \text{null}, x_1^t + y_0^t, \ldots, x_{15}^t + y_{14}^t]$ above, for some iteration $t = 0, 1, \ldots, 49$.
  - `inputs[p,:]` is the p-th image in the batch `inputs` (note that the images are flattened to 1D tensors, and the labels are one-hot encoded).
- The `states` variable is a dictionary that has the following set of keys: `{'controller_state', 'read_vector_list', 'w_r_list', 'w_u', 'M'}`
  - `controller_state` is the state of the controller in iteration $t - 1$; if $t - 1 < 0$, then it is just zero-filled. As it is an LSTM cell, `controller_state` is of the form `[(batch_size, rnn_size),(batch_size, rnn_size)]` (technically speaking its shape is `(2, batch_size, rnn_size)`). The two `(batch_size, rnn_size)`-shaped entries in it correspond to the cell state and the hidden state of the LSTM. We will mostly be treating the LSTM controller as a black-box in this project, so we do not need to pay much attention to the details of its states. If interested, you can read about the LSTM cell's technical details in [tf.keras.layers.LSTMCell (https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTMCell)](https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTMCell).
  - `read_vector_list` is the list of read vectors r_{t-1} which we obtained in the previous iteration $t - 1$ in the episode; if $t - 1 < 0$, then the read vector list is initialized to be an arbitrary one-hot vector. It is of the shape `(head_num, batch_size, memory_vector_dim)`. Basically, `read_vector_list[i,p,:]` is the $(t - 1)$-th-iteration read vector of the i-th read head for the p-th input sample in the batch.
  - `w_r_list` is the list of read weights w_{t-1}^r which we obtained in the previous iteration $t - 1$ in the episode; if $t - 1 < 0$, then the read weights list is initialized to be an arbitrary one-hot vector. It is of the shape `(head_num, batch_size, num_memory_slots)`. Basically, `w_r_list[i,p,:]` is the $(t - 1)$-th-iteration read weight of the i-th read head for the p-th input sample in the batch.
  - `w_u` is the list of memory usage weights w_{t-1}^u which we obtained in the previous iteration $t - 1$ in the episode; if $t - 1 < 0$, then the usage weights list is is initialized to be an arbitrary one-hot vector. It is of the shape `(batch_size, num_memory_slots)`. Basically, `w_u[p,:]` is the $(t - 1)$-th-iteration memory usage weight of the p-th input sample in the batch.
  - `M` is the memory content from the previous iteration $t - 1$; if $t - 1 < 0$, then the memory is just zero-filled. It is of shape `(batch_size, num_memory_slots, memory_vector_dim)`. Basically, `M[p,j,:]` is the j-th memory vector in the memory block for the p-th sample in the batch from iteration $t - 1$, and `M[p,:,:]` is the memory block for the p-th sample in the batch, where the memory block is a 2D structure that has `num_memory_slots` memory vectors, each vector of length `memory_vector_dim`.

Now let us look at some of the technical details of the MANNCell. First, we discuss the main ingredients of the MANNCell, and initialization of the relevant units.

- The input arguments of the class initialization method `__init__` have already been specified, they will be used to initialize relevant structures in the class.
- `self.controller` : this is the controller of the MANN cell that is responsible for interfacing with the memory M. We recommend using `tf.keras.layers.LSTMCell` with `units=rnn_size` for initialization. For its technical details, see [tf.keras.layers.LSTMCell (https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTMCell)](https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTMCell).
- `self.controller_output_to_read_keys` , `self.controller_output_to_write_keys` , `self.controller_output_to_alphas` : the LSTM controller's output structure (we will discuss what its inputs should be later) is of the form [controller_output, controller_cell_and_hidden_states]. We need a mapping that maps the controller_output to the read keys, write keys and the interpolation coefficient α_t's, which will then be used for interacting with the memory. Three `tf.keras.layers.Dense` layers (one for producing read keys, one for write keys, one for the α_t's) are sufficient, though you are welcome to try out more complicated structures.
  - **Remark 1**: each access to memory involves `head_num` number of heads, if you wish, you could just initialize `self.controller_output_to_read_keys` with `units=self.memory_vector_dim*self.head_num` and apply `tf.split` to the output of the dense layer along `axis=1` and `num_or_size_splits=head_num` in the `call` method (similar for the other two dense layers).
  - ***From now on, we assume that you are following Remark 1 above in your implementation***.
- `self.controller_output_to_logits` : it should be a dense layer that will be used to map the concatenated controller_output + read_vector_list to the logits that will be used for obtaining the classification labels of the inputs and computing the cross entropy values. Thus, initialize it with `units=self.num_classes` .

Finally, we discuss how to implement the method `call` . The following discussion is only one way of implementing the method, please feel free to deviate from it. However, ***we do suggest you to at least read through the discussion once***, as we have already implemented parts of the method and the whole training loop for you, and incompatibility between the data structures could cause the code to not run or have buggy outputs.

- **Caution**: even though most of the discussion below that involve tensors are treated either element-wise or vector-wise, in your implementation please utitlize tensorflow matrix operations as much as possible, as it can avoid strange bugs and increase the speed of your model.
- As described before, the input arguments of the `call` method are `inputs` and `states` .
  - Parse `state` to obtain `prev_controller_state` , `prev_read_vector_list` , `prev_w_r_list` , `prev_w_u` , `prev_M` that come from the previous iteration $t-1$. You may assume that they are zero-filled if $t=0$.
- Constructing the controller's input had been implemented for you.
  - The controller's output will be of the form `(controller_output, controller_states)` .
  - Why do you think we should involve `prev_read_vector_list` in the controller's input?
- Now pass `controller_output` to the dense layers we discussed before, and obtain the read keys, write keys and the interpolation coefficients.
  - Following the suggestion in the Remark 1 above, after applying `tf.split` to the dense layers' outputs, the shapes of your `read_key_list` and `write_key_list` should both be `(head_num, batch_size, memory_vector_dim)` , and the shape of `alpha_list` should be `(head_num, batch_size, 1)` . As an example, `read_key_list[i,p,:]` should be the memory read key for the i-th read head for the p-th sample in the batch.
- Before computing the read and write weights and interact with memory, we need to compute `prev_w_lu` , the least used weights from the previous iteration $t-1$.
  - You need to fill in the code for method `compute_w_lu` . To compute `prev_w_lu` , note that for the p-th sample in the batch in the previous iteration $t-1$, `prev_w_lu[p,:]` is a vector of binary values with length `num_memory_slots` : defining
    $$s(\text{prev\_w\_u}[p,:],k) = \text{the } k\text{-th smallest entry in prev\_w\_u}[p,:]$$
    we have
    $$\text{prev\_w\_lu}[p,i] = 0, \quad \text{if prev\_w\_u}[p,i] > s(\text{prev\_w\_u}[p,:], \text{head\_num})$$
    and
    $$\text{prev\_w\_lu}[p,i] = 1 \quad \text{otherwise}$$
  - Here is one way to implement `compute_w_lu` . Given input argument `prev_w_u` the usage weight from the previous iteration $t-1$ (it has shape `(batch_size, num_memory_slots)`), use `tf.math.top_k` to obtain the desired set of indices from `prev_w_u` (so you should have a `batch_size` number of index sets, each set is of size `head_num` ; the overall structure should be of shape `(batch_size, head_num)`). Then use `tf.one_hot` and `tf.reduce_sum` to expand these indices into `prev_w_lu` , which should have shape `(batch_size, num_memory_slots)` .
    - From the set of indices with size `(batch_size, head_num)` you used for computing `prev_w_lu` , remember to also construct and return the index corresponding to *the smallest* entry in `prev_w_u[p,:]` for every p (this index also correspond to the memory slot that was least used for the p-th sample in the previous iteration); so your returned indices will have size `(batch_size, 1)` .
    - You may find [tf.math.top_k (https://www.tensorflow.org/api_docs/python/tf/math/top_k)](https://www.tensorflow.org/api_docs/python/tf/math/top_k), [tf.one_hot (https://www.tensorflow.org/api_docs/python/tf/one_hot)](https://www.tensorflow.org/api_docs/python/tf/one_hot) and [tf.reduce_sum (https://www.tensorflow.org/api_docs/python/tf/math/reduce_sum)](https://www.tensorflow.org/api_docs/python/tf/math/reduce_sum) useful.

- Now we proceed to compute the read and write weights w_t^r and w_t^w.
  - For the p-th sample in the batch, recall that the read key `read_key_list[m,p,:]` is for the m-th read head for that sample, and `prev_M[p,j,:]` is the j-th memory vector for the p-th sample from the previous interation $t-1$. Then the memory **read** weight `w_r_list[m,p,:]` for the m-th read head for the p-th sample is a 1D tensor

with length `num_memory_slots` , with entries

$$\text{w\_r\_list}[m,p,i] = \frac{\exp(K(\text{prev\_M}[p,i,:], \text{read\_key\_list}[m,p,:]))}{\sum_{j=0}^{\text{num\_memory\_slots}-1} \exp(K(\text{prev\_M}[p,j,:], \text{read\_key\_list}[m,p,:]))}$$

where $i=0,1,...,\text{num_memory_slots}-1\$, and \begin{equation} K(x, y) = \frac{x\cdot y}{\Vert x \Vert_2 \Vert y \Vert_2 + \epsilon} \end{equation}

- ϵ is there to ensure numerical stability. $\epsilon = 10^{-8}$ seems to be a good choice.
- You might find some of the following tensorflow operations useful: [tf.matmul (https://www.tensorflow.org/api_docs/python/tf/linalg/matmul)](https://www.tensorflow.org/api_docs/python/tf/linalg/matmul), [tf.norm (https://www.tensorflow.org/api_docs/python/tf/norm)](https://www.tensorflow.org/api_docs/python/tf/norm), [tf.expand_dims (https://www.tensorflow.org/api_docs/python/tf/expand_dims)](https://www.tensorflow.org/api_docs/python/tf/expand_dims), [tf.squeeze (https://www.tensorflow.org/api_docs/python/tf/squeeze)](https://www.tensorflow.org/api_docs/python/tf/squeeze), [tf.math.exp (https://www.tensorflow.org/api_docs/python/tf/math/exp)](https://www.tensorflow.org/api_docs/python/tf/math/exp)
- In the suggested setup, the method `compute_read_weights` 's return shape should be `(batch_size, num_memory_slots)` , and `w_r_list` should have shape `(head_num, batch_size, num_memory_slots)` .

  - Given the p-th sample in the batch, the memory **write** weight `w_w_list[m,p,:]` for the m-th write head for that sample is of the general form:

    $$\text{w\_w\_list}[m,p,i] = \text{Sigmoid}(\text{alpha\_list}[m,p,0]) \times \text{prev\_w\_r\_list}[m,p,i] + (1 - \text{Sigmoid}(\text{alpha\_list}[m,p,0$$

    where $i = 0, \ldots, \text{num\_memory\_slots-1}$.

    - In our suggested setup, method `compute_write_weights` 's return shape should be `(batch_size, num_memory_slots)` , so `w_w_list` should have shape `(head_num, batch_size, num_memory_slots)` .

- Let us read from memory `prev_M` now.
  - As we have `w_r_list` with shape `(head_num, batch_size, num_memory_slots)` , to obtain the read vectors, simply carry out the following: for the m-th read head for the p-th sample,

    $$\text{read\_vector\_list}[m,p,:] = \sum_{j=0}^{\text{num\_memory\_slots}-1} \text{w\_r\_list}[m,p,j] \times \text{prev\_M}[p,j,:]$$

    where `read_vector_list` has shape `(head_num, batch_size, memory_vector_dim)` .

    - Please remember that computing with matrices (in contrast to using some kind of for loop) can usually make you code run faster.

- Having obtained the write weights `w_w_list` , we are closer to accessing the content of the memory now. But before that, rememeber that we got a set of indices of size `(batch_size, 1)` from the method `compute_w_lu` that indicated the least used memory slot in the previous iteration $t-1$? We are going to use them to zero out *the least used slot* in the memory first, before the writing operations.
  - One way of implementation: apply `tf.one_hot` to the set of indices of size `(batch_size, 1)` to obtain a matrix `E` of size `(batch_size, num_memory_slots)` containing one-hot vectors, where `E[p,j]` is 1 if the j-th memory slot for the p-th sample in the previous iteration was least used. Then we just need to compute the new memory along the line of $M * (1-E)$. So we have obtained `M_erased` , with shape `(batch_size, num_memory_slots, memory_vector_dim)` .

- Now we can write to memory:
  - Recall that we have already computed `write_key_list` and `w_w_list` with shapes `(head_num, batch_size, memory_vector_dim)` and `(head_num, batch_size, num_memory_slots)` respectively. To write to `M_erased` with the m-th write head for the p-th sample, simply compute

    $$\text{M\_written}[p,i,:] = \text{M\_erased}[p,i,:] + \text{w\_w\_list}[m,p,i] \times \text{write\_key\_list}[m,p,:]$$

    - You might find [tf.matmul (https://www.tensorflow.org/api_docs/python/tf/linalg/matmul)](https://www.tensorflow.org/api_docs/python/tf/linalg/matmul) and [tf.expand_dims (https://www.tensorflow.org/api_docs/python/tf/expand_dims)](https://www.tensorflow.org/api_docs/python/tf/expand_dims) useful here.

- Finally, update the usage weight w_t^u following the formula: for the p-th sample in the batch,

$$\text{w\_u}[p,:] = \text{self.gamma} \times \text{prev\_w\_u}[p,:] + \sum_{i=0}^{\text{head\_num}-1} \text{w\_r\_list}[i,p,:] + \sum_{i=0}^{\text{head\_num}-1} \text{w\_w\_list}[i,p,:]$$

where `w_u` has shape `(batch_size, num_memory_slots)` , and `self.gamma` is a manually defined free parameter of the model, which we have already set for you.

- Finally, we update the `state` dictionary , and feed [controller's output + the read vector list] to `self.controller_output_to_logits` which will be used for obtaining the labels for the input samples (already written for you) . Please ensure that all the relevant tensors have the correct shape and content.

In [0]:
```
class MANNCell(tf.keras.layers.AbstractRNNCell):
  def __init__(self, rnn_size, num_memory_slots, memory_vector_dim, head_num,
num_classes=5, gamma=0.95, **kwargs):
    super().__init__(**kwargs)
    ################ Setup ###############################################
    self.rnn_size = rnn_size
    # number of memory slots
    self.num_memory_slots = num_memory_slots
    # size of each memory slot
    self.memory_vector_dim = memory_vector_dim
    self.head_num = head_num
    # memory access head number is the same for both read and
    # write in our setup
    self.write_head_num = head_num
    # decay parameter for computing the usage weights
    self.gamma = gamma

    self.num_classes = num_classes
    ########################################################################

    # Controller RNN layer, we use an LSTM
    # Recommended: tf.keras.layers.LSTMCell
    self.controller = tf.keras.layers.LSTMCell(units=self.rnn_size)

    # controller_output
    #          -> read_key (batch_size, head_num*memory_vector_dim)
    #          -> write_key (batch_size, head_num*memory_vector_dim)
    #          -> alpha (batch_size, head_num), interpolation coefficient for
 writing to memory
    #
    # We suggest units=self.memory_vector_dim*self.head_num for initializing t
he dense layers
    # for read key and write keys, and units=self.head_num for the dense layer
for alpha,
    # and apply tf.split along axis=1 in the call method
    self.controller_output_to_read_keys =  tf.keras.layers.Dense(units=self.me
mory_vector_dim*self.head_num, use_bias=True)
    self.controller_output_to_write_keys = tf.keras.layers.Dense(units=self.me
mory_vector_dim*self.head_num, use_bias=True)
    self.controller_output_to_alpha = tf.keras.layers.Dense(units=self.head_nu
m, use_bias=True)

    # This is the dense layer for mapping the controller output + read vector
 list to
    # logits (which will then be used for computing the labels and cross-entro
py values
    # in NTMOneShotLearningModel). So initialize it with units=self.num_classe
s.
    self.controller_output_to_logits = tf.keras.layers.Dense(units=self.num_cl
asses, use_bias=True)

  @property
  def state_size(self):
    return self.rnn_size

  # This initializes the dictionary states in MANNCell, and returns the initia
l state.
  # Please do not change it.
  def zero_state(self, batch_size, rnn_size, dtype):
    one_hot_weight_vector = np.zeros([batch_size, self.num_memory_slots])
    one_hot_weight_vector[..., 0] = 1
```

```
    one_hot_weight_vector = tf.constant(one_hot_weight_vector, dtype=tf.float32)
    initial_state = {
            'controller_state': [tf.zeros((batch_size, rnn_size)), tf.zeros((batch_size, rnn_size))],
            'read_vector_list': [tf.zeros([batch_size, self.memory_vector_dim])
                                  for _ in range(self.head_num)],
            'w_r_list': [one_hot_weight_vector for _ in range(self.head_num)],
            'w_u': one_hot_weight_vector,
            'M': tf.constant(np.ones([batch_size, self.num_memory_slots, self.memory_vector_dim]) * 1e-6, dtype=tf.float32)
        }
    return initial_state

  def call(self, inputs, states):
    # read vectors from the previous iteration, extract from states
    prev_read_vector_list = states['read_vector_list']
    # state of controller from previous iteration t-1, extract from states
    prev_controller_state = states['controller_state']
    # Obtain the list of w^r_{t-1}, M_{t-1}, and w^u_{t-1}, extract from states

    prev_w_r_list = states['w_r_list']
    prev_M = states['M']
    prev_w_u = states['w_u']

    # Controller output form the parameters of the read and write vectors
    controller_input = tf.concat([inputs] + prev_read_vector_list, axis=1)
    controller_output, controller_state = self.controller(inputs=controller_input, states=prev_controller_state)

    # Map the controller_output to the read_keys, write_keys, and alphas
    read_keys = self.controller_output_to_read_keys(controller_output)
    write_keys =  self.controller_output_to_write_keys(controller_output)
    alphas = self.controller_output_to_alpha(controller_output)

    # We have head_num heads per access to memory (same number of heads for read and write),
    # so split the parameters obtained above into head_num groups,
    # tf.split is useful here (try splitting along axis=1. Why?)
    read_key_list = tf.tanh(tf.split(read_keys, self.head_num, axis=1))
    write_key_list = tf.tanh(tf.split(write_keys, self.head_num, axis=1))
    sig_alpha = tf.sigmoid(tf.split(alphas, self.head_num, axis=1))

    # For every p-th sample in the batch (from iteration t-1), compute the index
    # corresponding to least used memory slot in prev_M[p,:,:], return as prev_indices.
    # Also compute w^lu_{t-1}, return as prev_w_lu.
    # Please fill in the method self.compute_w_lu.
    prev_indices, prev_w_lu = self.compute_w_lu(prev_w_u)

    # Setup read and write weights
    w_r_list = []
    w_w_list = []
    # We obtain read and write weights for each head
    for i in range(self.head_num):
      # Obtain READ weights
      # Please fill in the method self.compute_read_weights
      w_r = self.compute_read_weights(read_key_list[i], prev_M)
      # Obtain WRITE weights
      # Please fill in the method self.compute_write_weights
```

```
      w_w = self.compute_write_weights(sig_alpha[i], prev_w_r_list[i], prev_w_
lu)
      # Note: w_r_list is of shape (head_num, batch_size, num_memory_slots),
      # and same for w_w_list
      w_r_list.append(w_r)
      w_w_list.append(w_w)
      # print(np.shape(w_w_list[0]))
      # print(np.shape(write_key_list[0]))

    # Read from memory M_{t-1}, using the w_r_list
    read_vector_list = []
    # Iterate over each head
    for i in range(self.head_num):
      # Fill in, compute read_vector
      # read_vector_list should have shape (head_num, batch_size, memory_vecto
r_dim)
      read_vector = tf.reduce_sum(tf.expand_dims(w_r_list[i], dim=2) * prev_M,
axis=1)
      read_vector_list.append(read_vector)


    # Set least used memory slot in prev_M to ZERO, make use of prev_indices!
    M_erased = prev_M * tf.expand_dims(1. - tf.one_hot(prev_indices[:, -1], se
lf.num_memory_slots), dim=2)

    # Write to memory, form M_t, using the w_w_list and write_keys
    # Iterate over each head
    for i in range(self.head_num):
      # Fill in
      M_written = M_erased + tf.matmul(tf.expand_dims(w_w_list[i], axis=2),
                                       tf.expand_dims(write_key_list[i], axis=
1))

    # Compute usage weights w^u_t for the current iteration
    w_u = self.gamma * prev_w_u + tf.add_n(w_r_list) + tf.add_n(w_w_list)

    # Concatenate controller's output and the read memory
    # content, they are then fed into a dense layer to obtain the logits,
    # which will be used for obtaininig labels and computing the  cross-entrop
y
    # values in NTMOneShotLearningModel below
    mann_output = tf.concat([controller_output] + read_vector_list, axis=1)
    logits = self.controller_output_to_logits(mann_output)

    state = {
        'controller_state': controller_state,
        'read_vector_list': read_vector_list,
        'w_r_list': w_r_list,
        'w_w_list': w_w_list,
        'w_u': w_u,
        'M': M_written,
    }

    return logits, state

  def compute_read_weights(self, read_key, prev_M):
    # Fill in
    read_key = tf.expand_dims(read_key, axis=2)
    # Compute the inner products, norms
    inner_product = tf.matmul(prev_M, read_key)
    read_key_norm = tf.sqrt(tf.reduce_sum(tf.square(read_key), axis=1, keep_di
ms=True))
```

```
        M_norm = tf.sqrt(tf.reduce_sum(tf.square(prev_M), axis=2, keep_dims=True))
        norm_product = M_norm * read_key_norm
        # Compute the exp(K(M,key))'s
        K = tf.squeeze(inner_product / (norm_product + 1e-8))
        K_exp = tf.exp(K)
        # Obtain read weights
        w_r = K_exp / tf.reduce_sum(K_exp, axis=1, keep_dims=True)
        return w_r

    def compute_write_weights(self, sig_alpha, prev_w_r, prev_w_lu):
        # Compute the write weights
        # Fill in
         w_w = sig_alpha * prev_w_r + (1. - sig_alpha) * prev_w_lu
         return w_w

    def compute_w_lu(self, prev_w_u):
          _, indices = tf.nn.top_k(prev_w_u, k=self.num_memory_slots)
          prev_w_lu = tf.reduce_sum(tf.one_hot(indices[:, -self.head_num:], depth=
self.num_memory_slots), axis=1)
          return indices, prev_w_lu
```

Already implemented, no need to change.

This class is part of the training loop.

In [0]:
```python
class NTMOneShotLearningModel():
  def __init__(self, model, n_classes, batch_size, seq_length, image_width, image_height,
                rnn_size, num_memory_slots, rnn_num_layers, read_head_num, write_head_num, memory_vector_dim, learning_rate):
    self.output_dim = n_classes

    # Note: the images are flattened to 1D tensors
    # The input data structure is of the following form:
    # self.x_image[i,j,:] = jth image in the ith sequence (or, episode)
    self.x_image = tf.placeholder(dtype=tf.float32, shape=[batch_size, seq_length, image_width * image_height])
    # Model's output label is one-hot encoded
    # The data structure is of the following form:
    # self.x_label[i,j,:] = one-hot label of the jth image in
    #               the ith sequence (or, episode)
    self.x_label = tf.placeholder(dtype=tf.float32, shape=[batch_size, seq_length, self.output_dim])
    # Target label is one-hot encoded
    self.y = tf.placeholder(dtype=tf.float32, shape=[batch_size, seq_length, self.output_dim])

    if model == 'LSTM':
      # Using a LSTM layer to serve as the controller, no memory
      def rnn_cell(rnn_size):
        return tf.nn.rnn_cell.BasicLSTMCell(rnn_size)
      cell = tf.nn.rnn_cell.MultiRNNCell([rnn_cell(rnn_size) for _ in range(rnn_num_layers)])
      state = cell.zero_state(batch_size=batch_size, dtype=tf.float32)
    elif model == 'MANN':
      # Using a MANN network as the controller, with memory
      cell = MANNCell(rnn_size, num_memory_slots, memory_vector_dim,
                                    head_num=read_head_num)
      state = cell.zero_state(batch_size=batch_size, rnn_size=rnn_size, dtype=tf.float32)

    cell
    self.state_list = [state]
    # Setup the NTM's output
    self.o = []

    # Now iterate over every sample in the sequence
    for t in range(seq_length):
      output, state = cell(tf.concat([self.x_image[:, t, :], self.x_label[:, t, :]], axis=1), state)
      output = tf.nn.softmax(output, axis=1)
      self.o.append(output)
      self.state_list.append(state)
    # post-process the output of the classifier
    self.o = tf.stack(self.o, axis=1)
    self.state_list.append(state)

    eps = 1e-8
    # cross entropy, between model output labels and target labels
    self.learning_loss = -tf.reduce_mean(
        tf.reduce_sum(self.y * tf.log(self.o + eps), axis=[1, 2])
    )

    self.o = tf.reshape(self.o, shape=[batch_size, seq_length, -1])
    self.learning_loss_summary = tf.summary.scalar('learning_loss', self.learning_loss)
```

```
    with tf.variable_scope('optimizer'):
      self.optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
      self.train_op = self.optimizer.minimize(self.learning_loss)
```

The training and testing functions

In [0]:

```
def train(learning_rate, image_width, image_height, n_train_classes, n_test_cl
asses, restore_training, \
         num_epochs, n_classes, batch_size, seq_length, num_memory_slots, augm
ent, save_dir, model_path, tensorboard_dir):

  # We always use one-hot encoding of the labels in this experiment
  label_type = "one_hot"

  # Initialize the model
  model = NTMOneShotLearningModel(model=model_path, n_classes=n_classes,\
                    batch_size=batch_size, seq_length=seq_length,\
                    image_width=image_width, image_height=image_height, \
                    rnn_size=rnn_size, num_memory_slots=num_memory_slots,\
                    rnn_num_layers=rnn_num_layers, read_head_num=read_head_num
,\
                    write_head_num=write_head_num, memory_vector_dim=memory_ve
ctor_dim,\
                    learning_rate=learning_rate)
  print("Model initialized")
  data_loader = OmniglotDataLoader(
      image_size=(image_width, image_height),
      n_train_classses=n_train_classes,
      n_test_classes=n_test_classes
  )
  print("Data loaded")
  # Note: our training loop is in the tensorflow 1.x style
  with tf.Session() as sess:
    if restore_training:
      saver = tf.train.Saver()
      ckpt = tf.train.get_checkpoint_state(save_dir + '/' + model_path)
      saver.restore(sess, ckpt.model_checkpoint_path)
    else:
      saver = tf.train.Saver(tf.global_variables())
      tf.global_variables_initializer().run()
    train_writer = tf.summary.FileWriter(tensorboard_dir + '/' + model_path, s
ess.graph)
    print("1st\t2nd\t3rd\t4th\t5th\t6th\t7th\t8th\t9th\t10th\tepoch\tloss")
    for b in range(num_epochs):
      # Test the model
      if b % 100 == 0:
        # Note: the images are flattened to 1D tensors
        # The input data structure is of the following form:
        # x_image[i,j,:] = jth image in the ith sequence (or, episode)
        # And the sequence of 50 images x_image[i,:,:] constitute
        # one episode, and each class (out of 5 classes) has around 10
        # appearances in this sequence, as seq_length = 50 and
        # n_classes = 5, as specified in the code block below
        # See the details in utils.py, OmniglotDataLoader class
        x_image, x_label, y = data_loader.fetch_batch(n_classes, batch_size, s
eq_length,
                                  type='test',
                                  augment=augment,
                                  label_type=label_type)
        feed_dict = {model.x_image: x_image, model.x_label: x_label, model.y:
y}
        output, learning_loss = sess.run([model.o, model.learning_loss], feed_
dict=feed_dict)
        merged_summary = sess.run(model.learning_loss_summary, feed_dict=feed_
dict)
        train_writer.add_summary(merged_summary, b)
        accuracy = test(seq_length, y, output)
```

```
        for accu in accuracy:
          print('%.4f' % accu, end='\t')
        print('%d\t%.4f' % (b, learning_loss))

      # Save model per 2000 epochs
      if b%2000==0 and b>0:
        saver.save(sess, save_dir + '/' + model_path + '/model.tfmodel', globa
l_step=b)

      # Train the model
      x_image, x_label, y = data_loader.fetch_batch(n_classes, batch_size, seq
_length, \
                                type='train',
                                augment=augment,
                                label_type=label_type)
      feed_dict = {model.x_image: x_image, model.x_label: x_label, model.y: y}
      sess.run(model.train_op, feed_dict=feed_dict)

# Fill in this function. You might not need seq_length (the length of an episo
de)
# as an input, depending on your setup
# Note: y is the true labels, and of shape (batch_size, seq_length, 5)
# output is the network's classification labels
def test(seq_length, y, output):
  correct = [0] * seq_length
  total = [0] * seq_length
  y_decode = np.argmax(y,axis=-1)
  output_decode = np.argmax(output,axis = -1)
  for i in range(np.shape(y)[0]):
    y_i = y_decode[i]
    output_i = output_decode[i]
    class_count = {}
    for j in range(seq_length):
        if y_i[j] not in class_count:
            class_count[y_i[j]] = 0
        class_count[y_i[j]] += 1
        total[class_count[y_i[j]]] += 1
        if y_i[j] == output_i[j]:
            correct[class_count[y_i[j]]] += 1
  return [float(correct[i]) / total[i] if total[i] > 0. else 0. for i in range
(1, 11)]
```

In [0]:

```
restore_training = False
label_type = "one_hot"
n_classes = 5
seq_length = 50
augment = True
read_head_num = 4
batch_size = 16
num_epochs = 10000
learning_rate = 1e-3
rnn_size = 200
image_width = 20
image_height = 20
rnn_num_layers = 1
num_memory_slots = 128
memory_vector_dim = 40
shift_range = 1
write_head_num = 4
test_batch_num = 100
n_train_classes = 220
n_test_classes = 60
save_dir = './save/one_shot_learning'
tensorboard_dir = './summary/one_shot_learning'
model_path = 'MANN'
train(learning_rate, image_width, image_height, n_train_classes, n_test_classe
s, restore_training, \
         num_epochs, n_classes, batch_size, seq_length, num_memory_slots, augm
ent, save_dir, model_path, tensorboard_dir)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow_cor
e/python/ops/resource_variable_ops.py:1630: calling BaseResourceVariable.__ini
t__ (from tensorflow.python.ops.resource_variable_ops) with constraint is depr
ecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
WARNING:tensorflow:From <ipython-input-3-a9faa67fc0c8>:157: calling reduce_sum
_v1 (from tensorflow.python.ops.math_ops) with keep_dims is deprecated and wil
l be removed in a future version.
Instructions for updating:
keep_dims is deprecated, use keepdims instead
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow_cor
e/python/util/dispatch.py:180: calling expand_dims (from tensorflow.python.op
s.array_ops) with dim is deprecated and will be removed in a future version.
Instructions for updating:
Use the `axis` argument instead
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow_cor
e/python/ops/math_grad.py:1424: where (from tensorflow.python.ops.array_ops) i
s deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
Model initialized
Entered Dataloader
10.0% data loaded.
20.0% data loaded.
30.0% data loaded.
40.0% data loaded.
50.0% data loaded.
60.0% data loaded.
70.0% data loaded.
80.0% data loaded.
90.0% data loaded.
100.0% data loaded.
Data loaded
1st     2nd     3rd     4th     5th     6th     7th     8th     9th     10th
epoch   loss
0.2375  0.1750  0.2250  0.1646  0.1899  0.2468  0.2083  0.2000  0.0980  0.1739
0       80.8004
0.2250  0.2000  0.1750  0.1125  0.2532  0.1818  0.1884  0.2308  0.2000  0.1522
100     80.4795
0.1875  0.2125  0.2000  0.2000  0.2278  0.2405  0.2192  0.2540  0.2885  0.2889
200     80.1958
0.2125  0.2500  0.1750  0.2000  0.2152  0.1974  0.1972  0.2787  0.2727  0.2045
300     80.3394
0.1500  0.1625  0.2405  0.2436  0.1948  0.3378  0.2192  0.1324  0.2667  0.2917
400     80.4481
0.1875  0.2125  0.1899  0.1646  0.2564  0.2468  0.1389  0.1846  0.1786  0.2093
500     80.5164
0.1375  0.2000  0.1500  0.1875  0.1646  0.2179  0.2667  0.1846  0.1724  0.2553
600     80.4993
0.1500  0.1625  0.1875  0.1375  0.2152  0.1842  0.1644  0.0968  0.1091  0.1429
700     80.6959
0.1500  0.2125  0.2125  0.2125  0.2125  0.2436  0.2192  0.2000  0.2143  0.2766
800     80.4178
0.1750  0.2000  0.2000  0.2125  0.1948  0.1579  0.0946  0.1406  0.2037  0.2500
900     80.5100
0.1875  0.1250  0.2625  0.2025  0.2821  0.2078  0.2297  0.2381  0.1538  0.2889
1000    80.4365
0.1750  0.2250  0.1875  0.2250  0.2250  0.2000  0.1268  0.2222  0.2105  0.1818
1100    80.4770
0.2500  0.1625  0.2500  0.2000  0.2152  0.1622  0.2059  0.2063  0.2353  0.2093
```

```
1200    80.5454
0.1625  0.1875  0.2375  0.2000  0.1875  0.1667  0.2267  0.2188  0.2143  0.2093
1300    80.5353
0.3000  0.1625  0.1750  0.2025  0.1646  0.2162  0.2432  0.2656  0.1731  0.2308
1400    80.4826
0.1875  0.2250  0.1375  0.2000  0.2152  0.1974  0.3714  0.2623  0.2407  0.2000
1500    80.3926
0.1750  0.2250  0.1772  0.1923  0.1688  0.1351  0.2000  0.1818  0.3585  0.2143
1600    80.5057
0.2250  0.1500  0.2500  0.1538  0.1184  0.1528  0.2258  0.2931  0.2453  0.1277
1700    80.4544
0.1500  0.1875  0.2152  0.2152  0.2051  0.1711  0.1429  0.2923  0.2105  0.1957
1800    80.5208
0.2125  0.2375  0.1500  0.2025  0.1392  0.2667  0.2429  0.2615  0.1724  0.2292
1900    80.5019
0.2000  0.2375  0.2625  0.2125  0.2051  0.1818  0.1667  0.2031  0.1356  0.2093
2000    80.4673
0.1625  0.2250  0.1750  0.1625  0.1875  0.1948  0.2000  0.1791  0.2143  0.2791
2100    80.4427
0.2500  0.2000  0.2405  0.2278  0.2152  0.2368  0.2429  0.1695  0.1346  0.2391
2200    80.4432
0.2250  0.2125  0.1875  0.2000  0.2375  0.2025  0.1667  0.2029  0.2500  0.2381
2300    80.5018
0.1625  0.1750  0.1750  0.1625  0.1875  0.2308  0.2535  0.3333  0.2321  0.3023
2400    80.1846
0.1875  0.2000  0.3000  0.2250  0.3000  0.3377  0.2917  0.3788  0.3000  0.3800
2500    75.4757
0.1750  0.2250  0.2375  0.3125  0.3500  0.3846  0.4028  0.3065  0.3846  0.3571
2600    72.9792
0.1750  0.2750  0.3250  0.4500  0.4684  0.4805  0.5143  0.5538  0.5536  0.6341
2700    67.1641
0.1750  0.2625  0.4125  0.4375  0.5128  0.5325  0.5942  0.5167  0.5818  0.5714
2800    61.2447
0.1625  0.2875  0.4625  0.6076  0.5195  0.6000  0.6912  0.6190  0.6545  0.6818
2900    55.8116
0.1125  0.4000  0.5375  0.5250  0.6250  0.5974  0.6849  0.6769  0.5926  0.6750
3000    52.9030
0.1500  0.3500  0.5000  0.5750  0.5696  0.5867  0.6056  0.4844  0.6379  0.6279
3100    56.1624
0.1750  0.4250  0.5000  0.6000  0.5750  0.5443  0.5467  0.5231  0.6296  0.6667
3200    55.0711
0.1375  0.5500  0.5125  0.5250  0.6456  0.7143  0.6351  0.6818  0.6481  0.7317
3300    51.7110
0.1250  0.2875  0.5000  0.6625  0.6026  0.5867  0.5833  0.5217  0.6780  0.6364
3400    55.1640
0.1375  0.3250  0.5125  0.5443  0.5526  0.5921  0.6765  0.5574  0.6545  0.6250
3500    53.1516
0.1125  0.3875  0.5125  0.6375  0.6962  0.6081  0.6866  0.7097  0.7544  0.6444
3600    49.9464
0.0875  0.4375  0.5250  0.6750  0.6538  0.5921  0.5833  0.7424  0.6786  0.6809
3700    51.4954
0.1625  0.4125  0.5375  0.5250  0.5949  0.7123  0.7059  0.6769  0.6167  0.8261
3800    49.9989
0.2250  0.4000  0.5750  0.6329  0.5190  0.6154  0.6528  0.5152  0.6333  0.5952
3900    52.1865
0.1000  0.4500  0.5500  0.6375  0.6500  0.7179  0.7500  0.6721  0.6923  0.6829
4000    47.6378
0.1250  0.4250  0.5000  0.6962  0.6582  0.6892  0.6438  0.7656  0.6271  0.7381
4100    46.7389
0.1375  0.5250  0.5875  0.6125  0.6250  0.7143  0.6429  0.7031  0.7778  0.7333
4200    48.5371
0.2125  0.4750  0.5125  0.6582  0.6410  0.6538  0.7432  0.6875  0.8824  0.7174
```

```
4300    44.4176
0.1125  0.4250  0.5875  0.5250  0.6500  0.6410  0.6104  0.7143  0.6441  0.6889
4400    51.0243
0.2125  0.5125  0.6203  0.7595  0.7895  0.7083  0.7941  0.7458  0.7091  0.7778
4500    42.8843
0.1750  0.4250  0.5875  0.5375  0.7273  0.6579  0.7222  0.8548  0.7091  0.7857
4600    44.6277
0.1750  0.4875  0.6625  0.8228  0.6456  0.6667  0.7000  0.7619  0.7407  0.6429
4700    44.6863
0.2125  0.6500  0.6750  0.7000  0.7125  0.6933  0.7059  0.7541  0.7636  0.8372
4800    41.0942
0.2000  0.5000  0.7000  0.7342  0.7821  0.7534  0.7000  0.8254  0.6780  0.6364
4900    42.5702
0.2750  0.4875  0.7375  0.7375  0.7975  0.7792  0.8143  0.9016  0.6471  0.8222
5000    38.5396
0.1500  0.5375  0.6875  0.7215  0.8077  0.8333  0.7879  0.8644  0.8364  0.9111
5100    35.2695
0.2625  0.5250  0.7500  0.7949  0.6923  0.8026  0.7083  0.7761  0.7966  0.8367
5200    40.3246
0.2000  0.6000  0.6875  0.7125  0.7051  0.7945  0.8169  0.8308  0.8302  0.9333
5300    37.1751
0.1500  0.6000  0.6835  0.6709  0.7468  0.7922  0.8310  0.7869  0.9231  0.7209
5400    38.1173
0.1875  0.5625  0.7000  0.7875  0.8250  0.7468  0.7973  0.8676  0.8276  0.8163
5500    35.5382
0.2250  0.4875  0.7875  0.7000  0.7792  0.8267  0.7639  0.8261  0.7627  0.8511
5600    38.9233
0.1875  0.5750  0.6375  0.7375  0.6962  0.8000  0.7463  0.7742  0.8070  0.7500
5700    39.4545
0.2500  0.6125  0.7500  0.7625  0.8228  0.7532  0.8310  0.8594  0.8182  0.8636
5800    35.6592
0.1625  0.5875  0.8250  0.7875  0.8125  0.8718  0.7945  0.7879  0.8070  0.7955
5900    35.3042
0.2000  0.6750  0.7000  0.8000  0.7750  0.8205  0.8429  0.7705  0.8361  0.9200
6000    33.8930
0.2375  0.5750  0.7750  0.7375  0.7975  0.8052  0.7639  0.8413  0.8036  0.8444
6100    33.0218
0.2375  0.5625  0.7625  0.7750  0.8228  0.8718  0.8421  0.8676  0.8393  0.8667
6200    32.9017
0.1875  0.6125  0.7375  0.8500  0.8481  0.8205  0.8133  0.8209  0.8704  0.8636
6300    32.8444
0.2750  0.6250  0.8354  0.8987  0.8701  0.8933  0.8904  0.9692  0.8103  0.8571
6400    28.6679
0.2000  0.6000  0.8125  0.7848  0.8077  0.8108  0.8209  0.8636  0.9123  0.9167
6500    31.9572
0.2375  0.7125  0.7625  0.8000  0.7949  0.8630  0.7971  0.8689  0.8364  0.8776
6600    31.8547
0.2375  0.5750  0.7722  0.7722  0.7179  0.7361  0.8154  0.7377  0.8333  0.7143
6700    37.8935
0.2250  0.6250  0.7750  0.8101  0.8333  0.8133  0.8592  0.7812  0.8113  0.7805
6800    31.4689
0.2625  0.5500  0.8250  0.8101  0.8077  0.8846  0.8493  0.8254  0.7273  0.8095
6900    30.2163
0.1375  0.6000  0.6875  0.6625  0.7949  0.7333  0.8310  0.8000  0.8421  0.8000
7000    38.3565
0.1875  0.6125  0.7250  0.7375  0.8846  0.8649  0.8507  0.8571  0.7407  0.9111
7100    31.4974
0.1875  0.6250  0.7750  0.7875  0.7848  0.8333  0.9167  0.8939  0.7963  0.8125
7200    32.3718
0.2000  0.7750  0.8250  0.8500  0.8462  0.8533  0.9118  0.8281  0.9649  0.9348
7300    27.3881
0.1750  0.6875  0.7250  0.7250  0.8462  0.8000  0.8732  0.8857  0.8364  0.7857
```

```
7400    32.0973
0.2625  0.5500  0.6750  0.8875  0.7875  0.8421  0.8219  0.8000  0.8730  0.8600
7500    33.2255
0.1500  0.6250  0.8000  0.8750  0.8625  0.8961  0.8533  0.8358  0.7925  0.8409
7600    29.7367
0.3250  0.6000  0.7500  0.8500  0.8125  0.8590  0.8841  0.9242  0.8364  0.8649
7700    30.8136
0.3125  0.6750  0.7875  0.8375  0.7875  0.9200  0.8889  0.9048  0.8364  0.8478
7800    29.4853
0.1750  0.6250  0.8375  0.7750  0.8500  0.8571  0.8784  0.8971  0.9167  0.8776
7900    29.5538
0.3250  0.5875  0.7595  0.8333  0.8312  0.8767  0.9286  0.8281  0.8103  0.8889
8000    29.3469
0.2500  0.6750  0.7875  0.7875  0.8500  0.8816  0.8630  0.9231  0.8545  0.8913
8100    28.7347
0.3000  0.7750  0.9000  0.8987  0.8831  0.8158  0.8630  0.9091  0.8750  0.8750
8200    25.8044
0.2125  0.6000  0.8625  0.7595  0.9231  0.9342  0.8611  0.8923  0.8400  0.8750
8300    27.7273
0.2625  0.7125  0.8250  0.8625  0.8625  0.8816  0.8630  0.9062  0.8519  0.8936
8400    27.3845
0.2500  0.6750  0.8750  0.9241  0.8861  0.8974  0.9155  0.8788  0.8814  0.8936
8500    24.6426
0.2250  0.6250  0.7250  0.7875  0.8718  0.8571  0.7838  0.8939  0.8846  0.9091
8600    32.3150
0.2375  0.7000  0.7250  0.8375  0.8974  0.8400  0.9014  0.8413  0.8868  0.8372
8700    28.7230
0.2250  0.6875  0.8625  0.8481  0.8590  0.7922  0.8611  0.8889  0.8654  0.8140
8800    28.7483
0.2625  0.5875  0.8625  0.8333  0.9359  0.8933  0.8986  0.9032  0.9091  0.8776
8900    25.7919
0.2000  0.6875  0.8750  0.8608  0.8846  0.8933  0.8732  0.8923  0.8868  0.9111
9000    27.9055
0.2500  0.7125  0.8375  0.8354  0.9103  0.8784  0.8657  0.8689  0.8750  0.8636
9100    27.0997
0.2250  0.7000  0.8500  0.8125  0.9125  0.8734  0.9079  0.9104  0.9298  0.8864
9200    26.1590
0.1875  0.6750  0.8125  0.9125  0.8500  0.9342  0.8378  0.9242  0.8929  0.8936
9300    26.9306
0.2250  0.6375  0.8500  0.8375  0.8625  0.8816  0.9167  0.9219  0.9298  0.7826
9400    26.4441
0.3250  0.7500  0.8875  0.8250  0.8625  0.8718  0.9315  0.9048  0.8113  0.8936
9500    25.1266
0.1875  0.7125  0.8375  0.8625  0.8462  0.9067  0.8919  0.8548  0.8929  0.9111
9600    28.3779
0.2750  0.7500  0.8987  0.9114  0.9620  0.9079  0.9315  0.9265  0.9273  0.8810
9700    21.2627
0.2750  0.6750  0.7750  0.8875  0.8718  0.8961  0.8169  0.8939  0.9464  0.8222
9800    26.5269
0.3250  0.6125  0.8000  0.8250  0.8750  0.8553  0.8873  0.9000  0.8846  0.8750
9900    28.4682
```

Comments on implementation and NTM learning algorithm:

1. Implementation: I was able to understatnd the ideas discussed in the Handout much clearly while going through the implementation. Also, the choice of having separate varaibles for computing read keys, write keys and interpolation coefficient makes the implementation identical to the equations dicussed in the handout.
2. The use of cosine similarity in this case is well suited when compared to other similarity measures. As cosine similarity measure does not place any constraints on the input vectors and always produces a output which is bounded which makes it well suited. On the other hand, the use of other similarity measures such as Earth Movers Distance and Jaccard Similarity are not directly applicable in this case as we would need a normalized vector for the former and a way of converting the real values to countable integers for the latter. Furthermore, cosine similarity comes with a small computation cost when compared to other measures.
3. Memory read write process: I beleive reading the memory before writing makes sense intuitively. However, it would be interesting to see if the performance improves if we reverse the process.