PURDUE UNIVERSITY
ECE 595 MACHINE LEARNING II

Project 5: One-Shot Learning with Neural Turing Machines

Due Date: Dec 6th, 2019

1 INTRODUCTION

There are two major components which we will learn about in this project: a special one-shot learning task, and the Neural Turing Machine. We give a general introduction to the two in the sections below.

1.1 One-Shot Learning

The learning task which we focus on in this project is a somewhat unorthodox one, different from the usual image classification or generation tasks you had already seen before in this class.

Consider a dataset $\mathscr{D} = \{(\boldsymbol{x}_t, y_t)\}_{t=0}^{T-1}$, where $\boldsymbol{x}_t \in \mathbb{R}^d$ is some image and $y_t \in \{1,...,K\}$ is the label for that image. We assume $T \gg K$. Now consider the following sequence of tuples:

$$(\boldsymbol{x}_0, \text{null}), (\boldsymbol{x}_1, y_0), ..., (\boldsymbol{x}_{T-1}, y_{T-2}). \tag{1.1}$$

Suppose we present this sequence of tuples to you one by one: at iteration $t = 0$, you see the image $\boldsymbol{x}_0$, at iteration $t = 1$, you see $\boldsymbol{x}_1$ and the label y_0 (the true label for the previous image $\boldsymbol{x}_0$), and so on. Moreover, at each iteration t, we ask you to tell the label of the image $\boldsymbol{x}_t$. Of course, for the first time you encounter an image from class k (say at iteration t_0), you would have no idea what its label is so you will have to randomly guess one, but because at iteration $t_0 + 1$ you are given its true label, the next time you see an image from class k, the probability of you giving the correct label for it increases. This is the task we focus on in this project - a game of memorization. An illustration of this task is provided in Figure 1.1 below.

But you might ask, what does this task have to do with one-shot learning? The truth is, this task is only a very specific problem under the umbrella of one-shot learning. Essentially, if there is a network that is capable of performing the above task well, then that network is learning about the sample-label binding of the dataset very quickly; if for every class $k \in \{1,...,K\}$, this network is capable of classifying an image from the class correctly upon the second time seeing a sample from that class, then this network indeed learned to solve this task in "one shot". It is very difficult to obtain this ideal network, but we will see later that we can train a Neural Turing Machine that can get pretty close to it.

Finally, we define the term "episode" to be the presentation of a dataset $\mathscr{D}$ (like the one defined above) in the "label-offset" sequence form as in (1.1).

1.2 Neural Turing Machine (NTM)

An NTM is basically a neural network that interfaces with a memory module. What makes it particularly interesting is that in addition to the controller module (which is usually a differentiable feedforward network or some recurrent network), its memory read and write operations that extract and manipulate memory content are defined by smooth differentiable functions.

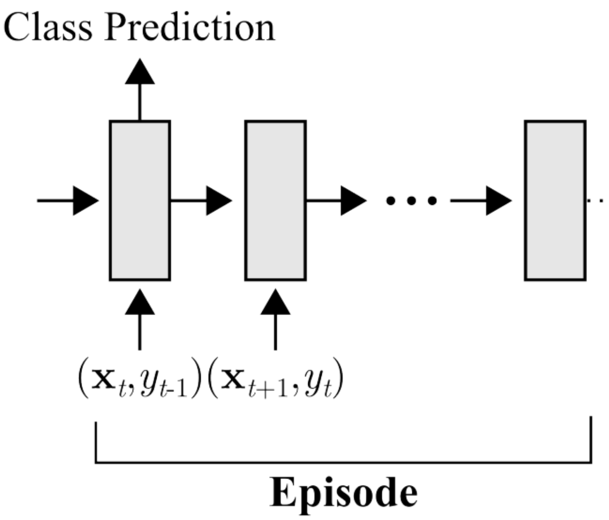

Figure 1.1: Task Setup

1.2.1 Overall Structure

Concretely speaking, the NTM $\mathscr{N}$ is composed of a controller $\mathscr{C}$ and a memory block $\boldsymbol{M} \in \mathbb{R}^{n_M \times l_M}$; $\boldsymbol{M}$ is basically a 2D matrix, n_M is the number of memory slots, and each memory slot is a vector of size l_M. Let us consider an episode of inputs like in (1.1). For an input sample $(\boldsymbol{x}_t, y_{t-1})$ at iteration t in the episode of length T, $\mathscr{C}$ computes a memory read key $\boldsymbol{k}_t$ through a differentiable operation, and the read key is then used to retrieve *memory read vector* $\boldsymbol{r}_t$ from the memory block $\boldsymbol{M}_{t-1}$ (the memory block is updated at every iteration in the episode, we will discuss it later; the subscript $t-1$ indicates that this is the memory block from the previous iteration). We may say that the read key is used to "drive" the memory read head(s) of this Turing machine. This retrieved memory content $\boldsymbol{r}_t$, in combination with the controller's output that resulted from the input sample, are then passed to some differentiable layer to produce the classification label $\hat{y}_t$ of $\boldsymbol{x}_t$. Furthermore, the controller also computes some memory write key(s) $\boldsymbol{a}_t$ that is then used to write content to memory $\boldsymbol{M}_{t-1}$ to form $\boldsymbol{M}_t$; so $\boldsymbol{a}_t$ "drives" the write head(s) of this Turing machine. Finally, relevant parameters of the current iteration are then stored in memory, and retrieved for use in the next iteration $t+1$ when $\mathscr{C}$ encounters input $(\boldsymbol{x}_{t+1}, y_t)$.

Let us now examine the details of how the NTM operates.

1.2.2 Memory Reading

The NTM reads from memory first, then it writes to memory.

We assume from now on that the controller $\mathscr{C}$ is an LSTM layer. This is mainly for maintaining consistency between the conventions adopted in this document and in the ipynb file.

At iteration t, given the LSTM controller $\mathscr{C}$'s internal state $\boldsymbol{s}_{t-1}$ from the previous iteration (if $t = 0$, $\boldsymbol{s}_{t-1}$ is zero-filled), and $[(\boldsymbol{x}_t, y_t) + \boldsymbol{r}_{t-1}]$ ("+" indicates concatenation; we will discuss how to compute $\boldsymbol{r}_{t-1}$ later), $\mathscr{C}$ outputs a tuple $(\boldsymbol{o}_t, \boldsymbol{s}_t)$, where $\boldsymbol{o}_t$ denotes the output values of the LSTM, and $\boldsymbol{s}_t$ denotes the internal states of the LSTM. A dense layer f_d^1 is then used to compute the read key $\boldsymbol{k}_t$ in the following manner:

$$f_d^1 : \mathbb{R}^{\text{size of } \boldsymbol{o}_t} \to \mathbb{R}^{l_M},\ \boldsymbol{o}_t \mapsto \boldsymbol{k}_t \tag{1.2}$$

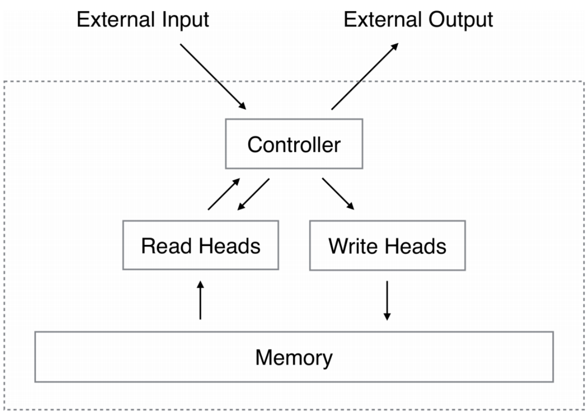

Figure 1.2: Basic Structure of a Neural Turing Machine

The read key is then used to compute the *memory read weight* $\boldsymbol{w}_t^r \in \mathbb{R}^{n_M}$ defined as follows:

$$w_t^r[i] = \frac{\exp(K(\boldsymbol{k}_t, \boldsymbol{M}_{t-1}[i]))}{\sum_{j=0}^{n_M-1} \exp(K(\boldsymbol{k}_t, \boldsymbol{M}_{t-1}[j]))}, \quad i = 0, 1, \dots, n_M - 1 \tag{1.3}$$

where $\boldsymbol{M}_{t-1}[i] \in \mathbb{R}^{l_M}$ denotes the i-th memory vector in $\boldsymbol{M}_{t-1}$, and $K : \mathbb{R}^{l_M} \times \mathbb{R}^{l_M} \to \mathbb{R}$ is the cosine similarity measure

$$K(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = \frac{\boldsymbol{\theta}_1 \cdot \boldsymbol{\theta}_2}{\|\boldsymbol{\theta}_1\|_2 \|\boldsymbol{\theta}_2\|_2 + \epsilon}, \tag{1.4}$$

and ϵ is some small number (e.g. 10^{-8}) for numerical stability. Finally, the memory read vector $\boldsymbol{r}_t \in \mathbb{R}^{l_M}$ is computed by

$$\boldsymbol{r}_t = \sum_{j=0}^{n_M-1} w_t^r[i] \boldsymbol{M}_{t-1}[i]. \tag{1.5}$$

Therefore, the memory read head places more “attention” on the i-th memory vector $\boldsymbol{M}_{t-1}[i]$ by having $w_t^r[i]$ closer to 1, and this is achieved by $\boldsymbol{k}_t$ being “similar” to $\boldsymbol{M}_{t-1}[i]$ (in the cosine similarity sense).

Additionally, we remark that there can be *multiple* memory read heads per iteration t. In other words, we can have n_r read heads, so we compute $\boldsymbol{k}_t(0), \dots, \boldsymbol{k}_t(n_r - 1)$ (using n_r corresponding dense layers $f_d^{1,0}, \dots, f_d^{1,n_r-1}$), obtain the corresponding read weights $\boldsymbol{w}_t^r(0), \dots, \boldsymbol{w}_t^r(n_r - 1)$, and finally get the read vectors $\boldsymbol{r}_t(0), \dots, \boldsymbol{r}_t(n_r - 1)$. This increase in the capacity of our NTM could potentially help the NTM learn faster and better in comparison to having just one read head.

Finally, when $t = 0$, we assume that the relevant parameters are initialized to either be zero-filled or some arbitrary one-hot vector, e.g. $\boldsymbol{M}_{t-1}$ would be a zero matrix, $\boldsymbol{r}_{t-1}$ would be an arbitrary one-hot vector and $\boldsymbol{o}_{t-1}$ would be zero-filled for $t = 0$.

1.2.3 Memory Writing

First, denote the *number of memory write heads* to be n_w. **Throughout this project, we assume that the number of read heads is equal to the number of write heads, i.e.,** $n_r = n_w \geq 1$.

Writing to memory is somewhat more complicated. It adopts a mechanism called “least recently used access”. It works as follows:

1. Using the controller $\mathscr{C}$'s output $\boldsymbol{o}_t$, apply a dense layer $f_d^{2,m}$ to it to obtain an *interpolation* parameter for the m-th write head

$$f_d^{2,m} : \mathbb{R}^{\text{size of } \boldsymbol{o}_t} \to \mathbb{R}, \quad \boldsymbol{o}_t \mapsto \alpha_t(m) \tag{1.6}$$

2. Apply a dense layer $f_d^{3,m}$ to $\mathscr{C}$'s output $\boldsymbol{o}_t$ to obtain the write key $\boldsymbol{a}_t(m) \in \mathbb{R}^{l_M}$ for the m-th write head:

$$f_d^{3,m} : \mathbb{R}^{\text{size of } \boldsymbol{o}_t} \to \mathbb{R}^{l_M}, \ \boldsymbol{o}_t \mapsto \boldsymbol{a}_t(m) \tag{1.7}$$

3. Obtain the *usage weights* $\boldsymbol{w}_{t-1}^u \in \mathbb{R}^{n_M}$ from the previous iteration $t-1$ (if $t=0$, $\boldsymbol{w}_{t-1}^u$ is just initialized to some arbitrary one-hot vector). We then compute the *least used weight* $\boldsymbol{w}_{t-1}^{lu} \in \mathbb{R}^{n_M}$ from the previous iteration. We first define $s(\boldsymbol{w}_{t-1}^u, k)$ = the k-th smallest entry in $\boldsymbol{w}_{t-1}^u$. Then

$$\boldsymbol{w}_{t-1}^{lu}[i] = \begin{cases} 0, & \text{if } w_{t-1}^u[i] > s(\boldsymbol{w}_{t-1}^u, n_w) \\ 1, & \text{if } w_{t-1}^u[i] \le s(\boldsymbol{w}_{t-1}^u, n_w) \end{cases} \quad, i = 0, 1, \ldots, n_M - 1$$

4. Before writing to memory, we first zero out the memory slot in $\boldsymbol{M}_{t-1}$ that was the least used from previous iteration. In other words, set $\boldsymbol{M}_{t-1}[i] = \boldsymbol{0}$ for i corresponding to the index of the smallest entry in $\boldsymbol{w}_{t-1}^u$.
5. Now we can compute the *memory write weight* $\boldsymbol{w}_t^w(m)$ *for the m-th write head*:

$$\boldsymbol{w}_t^w(m) = \text{Sigmoid}(\alpha_t(m))\boldsymbol{w}_t^r(m) + (1 - \text{Sigmoid}(\alpha_t(m)))\boldsymbol{w}_{t-1}^{lu} \tag{1.8}$$

6. We write to $\boldsymbol{M}_{t-1}$ to form $\boldsymbol{M}_t$ by carrying out the following operation for all $m = 0, \ldots, n_w - 1$:

$$\boldsymbol{M}_t[i] = \boldsymbol{M}_{t-1}[i] + w_t^w(m)[i] \times \boldsymbol{a}_t(m) \tag{1.9}$$

7. Finally, we compute the usage weight $\boldsymbol{w}_t^u$ for the current iteration:

$$\boldsymbol{w}_t^u = \gamma \boldsymbol{w}_{t-1}^u + \sum_{m=0}^{n_r-1} \boldsymbol{w}_t^r(m) + \sum_{m=0}^{n_w-1} \boldsymbol{w}_t^w(m) \tag{1.10}$$

   where $\gamma \in (0,1)$ is a manually defined free parameter.

We can gain some intuition about the NTM's mechanism of writing to memory by first examining (1.9). $\boldsymbol{a}_t(m)$ is the content that is to be written to $\boldsymbol{M}_{t-1}$, and $\boldsymbol{w}_t^w(m)$ chooses which slots in the memory the network wants to write to. But how is $\boldsymbol{w}_t^w(m)$ computed? From (1.8), it basically interpolates between $\boldsymbol{w}_t^r(m)$, which indicates the locations that were the most recently read from, and $\boldsymbol{w}_{t-1}^{lu}$, which indicates the least used memory slots from before. This leads to the following two questions. One, why should $\boldsymbol{w}_t^w(m)$ interpolate between two such parameters, and two, why does $\boldsymbol{w}_{t-1}^{lu}$ indicate the least used memory slots from previous iterations?

To answer the first question, note that the memory writing mechanism has the name "least recently used access". The reason we wish to involve the read weights $\boldsymbol{w}_t^r(m)$ from the current iteration is based on the assumption that the memory slots which were read from in the current iteration has low likelihood to be read in the next few iterations, so we can manipulate the content of these slots without worrying too much about losing valuable information. The reason we involve $\boldsymbol{w}_{t-1}^{lu}$ is more straightforward: if some memory slots are rarely used (compared to the other slots) from the previous iterations, we should feel free to manipulate them without too much worry. Now for the second question, the answer really boils down to examining the definition of $\boldsymbol{w}_{t-1}^{lu}$ presented above. If we have $n_w = n_r = 2$, then basically $\boldsymbol{w}_{t-1}^{lu}[i]$ is 0 if i corresponds to the index of the second/first smallest entry in $\boldsymbol{w}_{t-1}^u$, $\boldsymbol{w}_{t-1}^{lu}[i]$ is 1 otherwise; so we can think of $\boldsymbol{w}_{t-1}^{lu}$ as an indicator variable of the least used locations from before. Looking at the definition of $\boldsymbol{w}_t^u$ in (1.10), we see that $\boldsymbol{w}_{t-1}^u[i]$ indicates how heavy the i-th memory slot $\boldsymbol{M}_{t-1}[i]$ had been used (i.e. written to and/or read from) up until the previous iteration, so we are justified in using $\boldsymbol{w}_{t-1}^u[i]$ to define $\boldsymbol{w}_{t-1}^{lu}$, and for using it to execute step 4 above.

1.3 Implementation

The data loading and training loop had already been implemented for you in this project, you are only required to write two major chunks of code in the `one_shot_learning_Student.ipynb` - the network accuracy testing function `test`, and the class defining the core of the Neural Turing Machine: `MANNCell`. However, it could be very helpful to know the general working mechanism of the network, and the structure of the data being fed into your `MANNCell` and `test`. Since detailed information for implementation had already been provided for some of the programs in the notebook file, we will not dive into too much technical detail here.

1. The one-shot learning task for this project will be performed using the Omniglot dataset[1]. The Omniglot dataset consists of over 1600 separate classes with only a few examples per class; it is often called the "transpose of MNIST". The actual dataset we use will be a 280-classes subset of the Omniglot dataset, with 220 classes used for training, and 60 classes for testing. **Throughout this project, in each episode, we place the restriction that there can be at most 5 classes (out of 220 for training, 60 for testing) of images present.**
   a) We discuss here how we construct samples in an episode $[(\boldsymbol{x}^0, \text{null}), ..., (\boldsymbol{x}^{T-1}, y^{T-2})]$. Suppose we are in the training phase. We first sample 5 classes of data from the full 220-classes training dataset; call these 5 classes $\mathscr{D}_0, ..., \mathscr{D}_4$. Then we assign one-hot labels to these 5 classes, i.e., the $y_t^\prime s$ are now length-5 one-hot vectors. Now, we sample the episode: for any $t \in \{0, .., T-1\}$, $(\boldsymbol{x}_t, y_t)$ has equal chances of belonging to any one of the five classes, and the image $\boldsymbol{x}_t$ is randomly sampled from that randomly chosen class. Therefore, in an episode of length T with T reasonably large, we expect the number of images from any one of the five classes to be close to $T/5$. Finally, we offset the labels by one step (and we take null to be a length-5 zero vector), and we apply some data augmentation techniques to the images to reduce overfitting. A testing episode is constructed in a similar manner. For implementation details, see the class `OmniglotDataLoader` in the python file `utils.py` provided.
2. During training and testing, the overall NTM network $\mathscr{N}$ will be fed *batches* of episodes. Suppose the batch size is b. Then a single batch of input episodes is of the form

$$[[(\boldsymbol{x}_0^0, \text{null}), ..., (\boldsymbol{x}_0^{T-1}, y_0^{T-2})], ..., [(\boldsymbol{x}_{b-1}^0, \text{null}), ..., (\boldsymbol{x}_{b-1}^{T-1}, y_{b-1}^{T-2})]] \tag{1.11}$$

   and will have shape `(batch_size, sequence_length, image_size+5)` (notice that the images are flattend to 1D, and concatenated with the length-5 one-hot label vectors). $\mathscr{N}$'s final output on this batch of episodes will be $[[\hat{y}_0^0, ..., \hat{y}_0^{T-1}], ..., [\hat{y}_{b-1}^0, ..., \hat{y}_{b-1}^{T-1}]]$, having shape `(batch_size, sequence_length, 5)`. This batch of outputs and the true labels are then used to compute the cross-entropy values - note that we do gradient descent *per episode*, not per iteration in the episode. This way of utilizing gradient descent helps pushing the network's policy of memory usage toward the ideal one - encode sample $\boldsymbol{x}_t$ and store it in memory in iteration t, and bind its encoded information in the memory to its true label y_t at the next iteration $t+1$, finally, store this binding in memory for future use.
3. Leaving the actual implementation of the NTM aside for a moment, what exactly is the metric for measuring the performance of the network, and how do we implement it? This leads us to part 1 of this project.

[1] https://github.com/brendenlake/omniglot

2 Part 1: Network Accuracy Testing

Recall from Section 1.3 that, given a batch of input episodes

$$[[(\boldsymbol{x}_0^0, \text{null}), ..., (\boldsymbol{x}_0^{T-1}, y_0^{T-2})], ..., [(\boldsymbol{x}_{b-1}^0, \text{null}), ..., (\boldsymbol{x}_{b-1}^{T-1}, y_{b-1}^{T-2})]] \tag{2.1}$$

having shape `(batch_size, sequence_length, image_size+5)`, the output of the network will be a batch of labels for each episode:

$$[[\hat{y}_0^0, ..., \hat{y}_0^{T-1}], ..., [\hat{y}_{b-1}^0, ..., \hat{y}_{b-1}^{T-1}]] \tag{2.2}$$

having shape `(batch_size, sequence_length, 5)`. Also note that the true labels of the set of input episodes $[[y_0^0, ..., y_0^{T-1}], ..., [y_{b-1}^0, ..., y_{b-1}^{T-1}]]$ will also have the same shape as the network output.

The following is the desired functionality of function `test`:

1. Input arguments: the output labels of the network $\mathcal{N}$ having the form in (2.2) (where the network's input was a batch of testing episodes like in (2.1), and the true labels of the batch of testing episodes.
2. We measure the accuracy of the network in the following way: within the i-th episode in the batch of b episodes, define

   $$\text{ct}_i(\text{j-th instance}) = \sum_{k \in I_i(j)} \mathbb{1}\{\text{Correct prediction the j-th time } \mathcal{N} \text{ sees a sample from class } k \text{ in episode } i\} \tag{2.3}$$

   where $I_i(j) \subseteq \{0, ..., 4\}$ indicates the set of classes that have at least j samples in episode i, and $\mathbb{1}\{E\} = 1$ if event E is true, and 0 otherwise. Moreover, define

   $$\text{ACC}(\text{j-th instance}) = \frac{\sum_{i=0}^{b-1} \text{ct}_i(\text{j-th instance})}{\sum_{i=0}^{b-1} |I_i(j)|} \times 100 \tag{2.4}$$

3. Return ACC(1st instance), ..., ACC(10th instance) as a 1D list, with length 10.

Deliverables for Part 1:

1. Unzip the `NTM_Student.zip` file on your computer, and upload it to your google drive. You can then proceed to edit `Project_5_part_1_Student.ipynb`. Please read the suggestions in the ipynb first.
2. Fill in the function `test` in `Project_5_part_1_Student.ipynb`.
3. Run the notebook. Note that in this ipynb, we have implemented a very simple LSTM network (with no memory) as $\mathcal{N}$ for you to test your function `test`. Please allow the training of the basic LSTM network $\mathcal{N}$ to run for 5,000 epochs (if you wish, try running it for 10,000 epochs). You should be able to see that the accuracy of this network never exceeds 40%.
4. Remember to write your name at the start of the ipynb.

3 PART 2: NEURAL TURING MACHINE

In this part, you need to implement the class `MANNCell` in `one_shot_learning_Student.ipynb`.

Since the detailed technical implementation suggestions are written in the notebook file, we will only give a general overview of what `MANNCell` is expected to do. Suppose, for the sake of concreteness, that 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, and outputting 16 output labels. More specifically, the `MANNCell` should produce classification labels $[\hat{y}_0^t, ..., \hat{y}_{15}^t]$ for all 16 iteration-t image samples in the batch $[\boldsymbol{x}_0^t + \text{null}, \boldsymbol{x}_1^t + y_1^{t-1}, ..., \boldsymbol{x}_{15}^t + y_{15}^{t-1}]$ ("+" means concatenation, null is just a length-5 zero vector) every time it is called; for your information, it is the class `NTMOneShotLearningModel` (already implemented) that actually calls `MANNCell` 50 times. Your job is to make sure that at a single iteration t (where $t = 0, 1, 2, ..., 49$), `MANNCell` correctly parses the input arguments, produce the correct read and write weights $\boldsymbol{w}_t^r, \boldsymbol{w}_t^w$ for every sample in the batch, correctly retrieve from and write to the memory blocks, and use the right material to get the logits for classification (they will be used for computing the labels $[\hat{y}_0^t, ..., \hat{y}_{15}^t]$ and cross-entropy values in `NTMOneShotLearningModel`), and return the right states that will be used in the next iteration $t + 1$.

One more point to keep in mind: The discussion in Section 1 of this document is based on the NTM operating on one single episode of samples, while in our actual implementation, we always deal with a *batch* of episodes. Of course, the memory blocks and relevant parameters are separate for each episode, i.e., memory, read keys, write keys, read weights, write weights, usage weights, least used weights and other per-episode parameters are *not* shared across episodes.

Deliverables for Part 2:

1. Copy the `test` function you implemented in `Project_5_part_1_Student.ipynb` to `one_shot_learning_Student.ipynb`.
2. Fill in the code for the class `MANNCell` in `one_shot_learning_Student.ipynb`. Please at least read through the suggestions given in the notebook once, as we have already implemented parts of the class and the whole training loop for you, and incompatibility between the data structures/content already implemented and the ones you produce could cause the code to not run or have buggy outputs.
3. Report the best ACC(1st instance), ..., ACC(10th instance) that your NTM model can reach. You should be able to outperform the simple LSTM network from part 1 by a large margin in around 10,000 epochs of training, but feel free to train it for less epochs.
   Remark: In the first few thousand of epochs, the model's loss could stay very high with little change. Please have patience, if your implementation is correct, it should most likely start decreasing before you reach 10,000 epochs of training. Please note that it could happen that your implementation is correct but the loss just does not change (training this network requires a bit of luck!); we suggest restarting the training (so with a different random initialization of the weights) if you do not see the loss changing for 5,000 to 7,000 epochs; you might need to check your code though if a number of restarts still do not help.
4. Having gone through the exercise, in your opinion, does every step in our implementation make sense? For instance, do you think if we should have used more complicated structures for computing the read keys, write keys and the interpolation coefficients? Maybe we should have written to the memory before reading from it? Maybe a better measure of similarity than the cosine similarity in computing the reading weights? Does zeroing out the least used memory slot from before really make sense? A better mechanism of writing to memory than the "least recently used access"? Is there any hope of utilizing the experience and intuition we gained about NTMs and the one-shot learning task here on normal classification problems? Please elaborate.
5. If you are curious, try to sample a few episodes from the dataset, each episode of length 50, and test yourself on them. Can you outperform the NTM?
6. Please write your answers in a text cell at the end of the notebook. Also, please remember to write your name at the start of the ipynb.