CIS 700 Project 1

Yijie Zhao
10th March 2020

1 Introduction

This project contains 2 parts:

- Part I reproduces the result of Ask Me Anything: Dynamic Memory Networks for Natural Language Processing for Question and Answering on the bAbI dataset.
- Part II changes the GloVe word embedding used in the original paper to ELMo word embeddings and implements the new model on task 1 and 17 of the bAbI dataset.

2 Part I: Reproducing Results

The relevant files are contained in the folder **Part_1_reproduce_results**. Instruction on how to run the code and a detailed description of each file that is contained in this folder can be found in **README.txt**. Below is a table of the accuracy on the 20 tasks of the bAbI dataset recorded in the paper and reproduced in this project:

Table 1: Question and answering task accuracy comparison on bAbI dataset

Task	Paper DMN Accuracy	Reproduced Accuracy
1	100	100
2	98.2	16.50
3	95.2	19.43
4	100	100
5	99.3	99.51
6	100	90.33
7	96.9	99.02
8	96.5	/
9	100	82.91
10	97.5	97.13
11	99.9	100
12	100	100
13	99.8	100
14	100	100
15	100	100
16	99.4	100
17	59.6	47.75
18	95.3	89.36
19	34.5	9.47
20	100	100

Findings:

- As can be seen in the table above, task 1, 4-7, 9-18, and 20 achieve roughly the same accuracy as those recorded in the paper. Some are slightly higher than that in the papr (e.g. task 16: the reproduced accuracy is 100 %, while the paper records an accuracy of 99.4 %). However, the difference is not significant. The reproduced accuracy among different tasks aligns with intuition: for harder tasks such as task 17: Positional Reasoning and task 19: Path Finding, the accuracy is much lower than those easier tasks such as task 1: Single Supporting Fact.

- As marked in red above, we notice task 2: Two Supporting Facts, 3: Three Supporting Facts, and 19: Path Finding have a much lower reproduced accuracy as compared to those recorded in the paper. The results for task 2 and 3 seem suspicious since they are quite different from the result of task 1: One Supporting Fact, which is similar but easier than task 2 and 3. From the set of result in the original paper, it seems unlikely that tasks dealing with more than one supporting facts become significantly harder than task with just one supporting fact. Since the set of code reproduces the majority of the tasks roughly correctly, we also doubt the issue comes from the code. Thus, we are unsure what could cause such a discrepancy for these tasks.
- We try to run the model on task 8: Lists/Sets multiple times; however, we keep on getting errors. This could be because this part of the data was loaded in incorrectly since the same set of code works well on all other tasks. However, since all other 19 tasks are reproducable, we believe leaving this one out is of no big harm.

3 Part II: Dynamic Memory Network with ELMo Embeddings

In the original paper, the authors used GloVe embeddings of the text as the input of the dynamic memory network. In the Pytorch implementation of the network we used in part I, the authors first load in the pre-trained GloVe word embeddings (glove.6B.300d.txt) and the bAbI dataset, create dictionaries that map words to their embeddings and vice versa in **dataset.py** and then run the model (described in **main.py, run.py, model.py**) using above as input.

In this part, we decide to change the input word embeddings from GloVe to ELMo, which is a deep contextualized word representation that models both (1) complex characteristics of word use (e.g., syntax and semantics), and (2) how these uses vary across linguistic contexts. We still use the set of codes from part I as the backbone. The specific changes made to each file is as follows:

- **dataset.py**
  As mentioned above, this file is originally used to build a list of words that have appeared in the dataset and several dictionaries that map words to their GloVe embeddings and vice versa (the file introduces index as a simpler way to record words, but we will not go into detail about this here).

  The overall idea of the changes made here is that we first use Tensorflow Hub to calculate the word embedding of each word per sentence. Then we store each word with a special suffix in the dictionary that maps words to embeddings or vice versa. The suffix consists of the name of the file (e.g. qa1_test.txt) and line number of this sentence, and the index of the word in this sentence. Thus, the suffix is unique per word occurred in the file. Since we noticed that many sentences (such as "Where is Sandra") occurs multiple times in the text file, to reduce computation complexity, when processing a new sentence, we first check if the entire sentence has occurred before. If so, we would not add each word of the sentence again in the dictionary with a new suffix since the calculated ELMo word embeddings would be the same as the sentence that has occurred before and already stored in the dictionaries. We do, however, increment the count of the word in the dictionary by 1 as in the original code. Specifically:

  We first load in Tensorflow Hub, which is used to calculate the ELMo word embeddings per word of each sentence. This can be seen in line 15-16 of the code file:

  ```
  elmo = hub.Module("https://tfhub.dev/google/elmo/2", trainable=True)
  init = tf.compat.v1.global_variables_initializer()
  ```

  Then, when initializing dictionaries, as compared to part I, we introduce a new dictionary called sen2fileidx, which maps sentence in the text file to a list [file, line_index], which indicates the file name and the line number of which this sentence is first seen. If a sentence has occurred before, then we know the ELMo embedding of each word in this sentence must be the same as before. Therefore, there is no need to re-calculate. This newly introduced dictionary sen2fileidx is used to keep track of what sentences we have seen so far. This can be seen in line 56:

  ```
  self.sen2fileidx = {}
  ```

  Additionally, since ELMo does not have specific embeddings for punctuation, we set the embedding of '?', '.' to a vector of all zeros and initialize them in the dictionaries that map index to word, word to index, and index

to vector. This can be seen in line 40 - 48:

```
# vector for period .
self.word2idx['.'] = 0

... [OMITTED - check dataset.py]

self.idx2vec.append([0.0] * self.config.word_embed_dim)
```

In the original code, the function build_word_dict is used to loop through all text files in the bAbI dataset and create a list of all the words appeared in the dataset. In addition to this, we include the task of keeping track of what sentences have appeared in the dataset. That is, we update the dictionary sen2fileidx here. This can be seen in line 72 - 112:

```
def update_init_dict(line, file, line_idx):
    words = nltk.word_tokenize(line)
    if '.' in words or '?' in words:
        words = words[:-1]

    ... [OMITTED - check dataset.py]

            self.init_word_dict[word_suffix] = (
                self.init_word_dict[word_suffix][0],
                self.init_word_dict[word_suffix][1] + 1)
```

We also change line 100, 104, 108, where the function update_init_dict has been called since this function is now modified and takes in different parameters.

Function map_dict has been heavily changed. This is because in the original code, the dictionaries (idx2word, word2idx, and idx2vec) are updated in the function get_pretrained_word, which goes into the folder of GloVe embeddings to retrive the embedding per word, but this function is no longer applicable in this setting (since we can no longer map a word to a single word embedding now) and thus **deleted**. We move the task of updating the above dictionaries to this function map_dict. This function is called per line of story, question, answer inside function process_data. We also change the input parameters of this function and newly include a parameter called type, which indicates whether the input text is a question, story, or answer. We then add the index corresponding to '.' or '?' to the output correspondingly. The entire function is nearly completely re-written. The changes in the function map_dict can be seen in line 115 - 153:

```
def map_dict(self, line, type):
    words = nltk.word_tokenize(line)
    if '.' in words or '?' in words:
        words = words[:-1]

    ... [OMITTED - check dataset.py]

    elif type == 'story':
        output.append(self.word2idx['.'])
    return output
```

We also change line 181, 185, 202, which are the three places that have called the function map_dict, which has been modified with different input parameters now.

Since function map_dict is changed for a specific use of converting words to index, we introduce a new function called map_dict2 that is used for converting index to word. Note that since we store words with a special suffix in the dictionaries, in this function we also convert the stored words back to their original version (e.g. from john_qa1_test.txt:10 to john). This can be seen in line 319 - 325:

```
def map_dict2(self, key_list):
    output = []
    for key in key_list:
```

```
... [OMITTED - check dataset.py]

    return output
```

In the original code, the function decode_data uses map_dict to convert indices back to words. Now, instead of map_dict, we use map_dict2 that we just created. This can be seen in line 329 - 331.

In class Config, we changed self.word_embed_dim from 300 (GloVe) to 1024 (ELMo). This is in line 345.

- **model.py**
  When comparing the predicted answers with the target answers, we convert the words with suffix back to the original words and then do the comparison. This change can be seen in line 222 - 227:

```
acc = np.array([1.0 for _ in range(outputs.size(0))])
for target, topk in zip(target_list, topk_list):
    acc *= np.array([float(k == tk[0] or k == -100) \
            for (k, tk.split('_')[0]) in zip(target, topk)])
    # print(acc)
acc = np.mean(acc) * 100
```

- **run.py, main.py**
  Since we preserved the overall structure of the original code by making a large number of changes in dataset.py (we preserved dictionaries such as word2idx, idx2word, word2vec, although this is not the most efficient solution), no change was necessary in these two files.

Since using Tensorflow Hub to retrieve the ELMo word embeddings for the input text files is very time-consuming, we only implemented the ELMo version of the network on task 1: Single Supporting Fact, and task 17: Positional Reasoning of the bAbI dataset. We specifically chose these two since one is hard (with 59.6 % accuracy as recorded in the paper) and one is easy (with 100 % accuracy). The results can be seen in the following table:

Table 2: Question and answering task accuracy comparison on bAbI dataset [ELMo]

Task	Paper DMN Accuracy	GloVe-based Accuracy	ELMo-based Accuracy
1	100	100	100
17	59.6	47.75	50.39

Findings

- From the table above, we notice that the ELMo-based dynamic memory network performs equally well as the GloVe-based dynamic memory network on task 1: Single Supporting Fact, which is an easy task.
- We also notice the ELMo-based dynamic memory network performs slightly better than that of the GloVe-based on the hard task: Positional Reasoning. This aligns with our intuition since ELMo word embeddings take into account the context of which the word lives in and thus is more 'accurate' in describing the meaning of the word in a sentence as compared to GloVe. This, we believe, helps the network better understands hard tasks and thus achieves a higher accuracy overall. However, we can not make solid conclusion on which is better since, due to time constraint, we only ran the experiment once.