SS-4290: Computer Science Final Year Project Weekly Report

**Week 9 (Due 29th September 2021)**

Muhammad Syauqi Waiz bin Haji Sufri

17B9082

Objectives:

1. Learn PyTorch.
2. Implement EMN.

1. Learn PyTorch.

Installed PyTorch on my machine using PyCharm. Learning PyTorch basics by watching tutorials. Some of the tutorials I tried:

```
import torch

#initialize two sets of tensors
x = torch.tensor([5,3])
y = torch.Tensor([2,1])

#multiple the two arrays
print("Multiplication of the two arrays: ")
print(x*y)

#create an tensors of 5 elements in 2 arrays:
x = torch.zeros([2,5])
print("New tensor: ")
print(x)

#check the size of tensors
print("Tensor size: ")
print(x.shape)

#Assign random elements in tensor
y = torch.rand([2,5])
print("Random elements assigned:")
print(y)

#View the different size of the tensors:
print("Viewing different Tensor's size ([1, 10]: ")
print(y.view([1,10]))

print("Print y again and it returns back to original size: ")
print(y)

#Change the size of tensor
y = y.view([1,10])
print("Change the size of tensor: ")
print(y)
```

Output:

```
Multiplication of the two arrays:
tensor([10.,  3.])
New tensor:
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
Tensor size:
torch.Size([2, 5])
Random elements assigned:
tensor([[0.3381, 0.5986, 0.6800, 0.5777, 0.6456],
        [0.7322, 0.8552, 0.8719, 0.3825, 0.5167]])
Viewing different Tensor's size ([1, 10]:
tensor([[0.3381, 0.5986, 0.6800, 0.5777, 0.6456, 0.7322, 0.8552, 0.8719, 0.3825,
         0.5167]])
Print y again and it returns back to original size:
tensor([[0.3381, 0.5986, 0.6800, 0.5777, 0.6456],
        [0.7322, 0.8552, 0.8719, 0.3825, 0.5167]])
Change the size of tensor:
tensor([[0.3381, 0.5986, 0.6800, 0.5777, 0.6456, 0.7322, 0.8552, 0.8719, 0.3825,
         0.5167]])

Process finished with exit code 0
```

Tensor default type is float, hence why the elements has float type. Tensor is put simply, numpy but run in a GPU.

Also, did a quick tutorial on using torchvision. Torchvision is a library for Computer Vision:

```python
import torch
import torchvision
from torchvision import transforms, datasets
import matplotlib.pyplot as plt

#Training and testing datasets downloaded from The Internet
#Two sets of tensors: One containing the image; the other the label
train = datasets.MNIST("", train=True, download=True, transform =
transforms.Compose([transforms.ToTensor()]))

test = datasets.MNIST("", train=False, download=True, transform =
transforms.Compose([transforms.ToTensor()]))

#Train and test the data with batch size of 10
trainset = torch.utils.data.DataLoader(train, batch_size=10, shuffle=True)
testset = torch.utils.data.DataLoader(test, batch_size=10, shuffle=True)

#Display the train dataset:
for data in trainset:
    print(data)
    break

#Assign the first image in the tensor to a variable
x, y = data[0][0], data[1][0]
print(y)

#Display the dimension of the image
print(data[0][0].shape)

#Display the image
```

```
plt.imshow(data[0][0].view(28,28))
plt.show()
```

Output:

```
[tensor([[[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]],


        [[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]],


        [[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]],
```

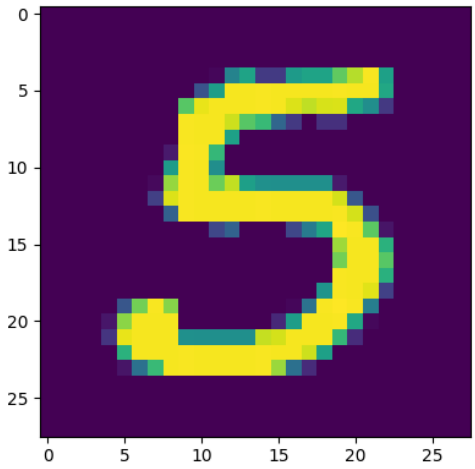
```
        [[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]],


        [[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]],


        [[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]]]), tensor([5, 7, 2, 0, 7, 1, 4, 2, 4, 8])]
tensor(5)
torch.Size([1, 28, 28])

Process finished with exit code 0
```

Tensor(5) is displayed:



2. Implement EMN

EMN Theory:

Given a list of input sentences with a query “Where is milk?”

1. Ali went to the kitchen.
2. Ahmad went to the kitchen.
3. Ali picked up the milk.
4. Ali traveled to the office.
5. Ali left the milk.
6. Ali went to the bathroom.

Input sentences are stored in memory, m:

| Memory, m | Sentence |
|---|---|
| 1 | Ali went to the kitchen. |
| 2 | Ahmad went to the kitchen. |
| 3 | Ali picked up the milk. |
| 4 | Ali traveled to the office. |
| 5 | Ali left the milk. |
| 6 | Ali went to the bathroom. |

First query, q will locate the sentence in the memory, m1 that is most relevant to it by performing a scoring function, s1. Then q and m1 will combine to form a new query:

$$(q, m1)$$

It will perform another scoring function, s1 to find the most relevant sentence, m2 and combine with q to become:

$$(q, m1, m2)$$

where it will be instead be combined with scoring function, s2 to locate a word, W which is essentially the answer to the query.

The process is as follow:

$$o1 = \text{softmax } s1(q, mi)$$

Query, q = “Where is milk?”

s1 = scoring function

o1 = index of the memory with the most relevant

Based on the example input sentences given above, the most relevant is “Ali left the milk”. Hence, m1 = “Ali left the milk” is mi

Then follow by second inference:

$$o2 = \text{softmax } s1([q, m1], mi)$$

The most relevant is “Ali traveled to the office”, and combine together:

o = [q, m1, m2] = [“where is milk?”, “Ali left the milk.”, “Ali travelled to the office.”]

Finally, the answer will be given by:

$$r = \text{softmax } s2([q, m1, mi], w)$$

w = All words in the dictionary

r = response

s2 = scoring function

Hence, the response and the answer in this case would be “office”.

Code implementation:

Model.py from https://github.com/zshihang/MemN2N/blob/master/model.py:

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as I


class MemN2N(nn.Module):

    def __init__(self, params, vocab):
        super(MemN2N, self).__init__()
        self.input_size = len(vocab)
        self.embed_size = params.embed_size
        self.memory_size = params.memory_size
        self.num_hops = params.num_hops
        self.use_bow = params.use_bow
        self.use_lw = params.use_lw
        self.use_ls = params.use_ls
        self.vocab = vocab

        # create parameters according to different type of weight tying
        pad = self.vocab.stoi['<pad>']
        self.A = nn.ModuleList([nn.Embedding(self.input_size,
self.embed_size, padding_idx=pad)])
        self.A[-1].weight.data.normal_(0, 0.1)
        self.C = nn.ModuleList([nn.Embedding(self.input_size,
self.embed_size, padding_idx=pad)])
        self.C[-1].weight.data.normal_(0, 0.1)
        if self.use_lw:
            for _ in range(1, self.num_hops):
                self.A.append(self.A[-1])
                self.C.append(self.C[-1])
            self.B = nn.Embedding(self.input_size, self.embed_size,
padding_idx=pad)
            self.B.weight.data.normal_(0, 0.1)
            self.out = nn.Parameter(
                I.normal_(torch.empty(self.input_size, self.embed_size), 0,
0.1))
            self.H = nn.Linear(self.embed_size, self.embed_size)
            self.H.weight.data.normal_(0, 0.1)
        else:
            for _ in range(1, self.num_hops):
                self.A.append(self.C[-1])
                self.C.append(nn.Embedding(self.input_size, self.embed_size,
padding_idx=pad))
                self.C[-1].weight.data.normal (0, 0.1)
            self.B = self.A[0]
            self.out = self.C[-1].weight

        # temporal matrix
        self.TA = nn.Parameter(I.normal_(torch.empty(self.memory_size,
self.embed_size), 0, 0.1))
        self.TC = nn.Parameter(I.normal_(torch.empty(self.memory_size,
```

```
self.embed_size), 0, 0.1))

    def forward(self, story, query):
        sen_size = query.shape[-1]
        weights = self.compute_weights(sen_size)
        state = (self.B(query) * weights).sum(1)

        sen_size = story.shape[-1]
        weights = self.compute_weights(sen_size)
        for i in range(self.num_hops):
            memory = (self.A[i](story.view(-1, sen_size)) *
weights).sum(1).view(
                *story.shape[:-1], -1)
            memory += self.TA
            output = (self.C[i](story.view(-1, sen_size)) *
weights).sum(1).view(
                *story.shape[:-1], -1)
            output += self.TC

            probs = (memory @ state.unsqueeze(-1)).squeeze()
            if not self.use_ls:
                probs = F.softmax(probs, dim=-1)
            response = (probs.unsqueeze(1) @ output).squeeze()
            if self.use_lw:
                state = self.H(response) + state
            else:
                state = response + state

        return F.log_softmax(F.linear(state, self.out), dim=-1)

    def compute_weights(self, J):
        d = self.embed_size
        if self.use_bow:
            weights = torch.ones(J, d)
        else:
            func = lambda j, k: 1 - (j + 1) / J - (k + 1) / d * (1 - 2 * (j +
1) / J)     # 0-based indexing
            weights = torch.from_numpy(np.fromfunction(func, (J, d),
dtype=np.float32))
        return weights.cuda() if torch.cuda.is_available() else weights
```

Currently still attempting to understand the codes and make sense of it before moving on to the next step of the codes (i.e main.py etc) and write a code for MEMO.

End