```
!pip3 install http://download.pytorch.org/whl/cu80/torch-0.3.0.post4-cp36-cp36m-linux_x86_64.whl
```

```
Collecting torch==0.3.0.post4 from http://download.pytorch.org/whl/cu80/torch-0.3.0.post4-cp36-cp36m-linux_x86_64.whl
  Downloading http://download.pytorch.org/whl/cu80/torch-0.3.0.post4-cp36-cp36m-linux_x86_64.whl (592.3MB)
    100% |████████████████████████████████| 592.3MB 62.0MB/s
Requirement already satisfied: pyyaml in /usr/local/lib/python3.6/dist-packages (from torch==0.3.0.post4) (3.13)
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from torch==0.3.0.post4) (1.14.6)
Installing collected packages: torch
Successfully installed torch-0.3.0.post4
```

```
!pip3 install torchvision
```

```
Collecting torchvision
  Downloading https://files.pythonhosted.org/packages/ca/0d/f00b2885711e08bd71242ebe7b96561e6f6d01fdb4b9dcf4d37e2e13c5e1
    100% |████████████████████████████████| 61kB 2.9MB/s
Requirement already satisfied: torch in /usr/local/lib/python3.6/dist-packages (from torchvision) (0.3.0.post4)
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from torchvision) (1.14.6)
Collecting pillow>=4.1.1 (from torchvision)
  Downloading https://files.pythonhosted.org/packages/85/5e/e91792f198bbc5a0d7d3055ad552bc4062942d27eaf75c3e2783cf64eae5
    100% |████████████████████████████████| 2.0MB 12.4MB/s
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from torchvision) (1.11.0)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.6/dist-packages (from torch->torchvision) (3.13)
Installing collected packages: pillow, torchvision
  Found existing installation: Pillow 4.0.0
    Uninstalling Pillow-4.0.0:
      Successfully uninstalled Pillow-4.0.0
Successfully installed pillow-5.4.1 torchvision-0.2.1
```

Importing the basic libraries

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.optim as optim
import torch.nn.functional as F
import nltk
import random
import numpy as np
from collections import Counter, OrderedDict
import nltk
from copy import deepcopy
import os
import re
import unicodedata
flatten = lambda l: [item for sublist in l for item in sublist]

from torch.nn.utils.rnn import PackedSequence, pack_padded_sequence
random.seed(1024)
```

Checking if GPU is available or not.

```
USE_CUDA = torch.cuda.is_available()
gpus = [0]
torch.cuda.set_device(gpus[0])

FloatTensor = torch.cuda.FloatTensor if USE_CUDA else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if USE_CUDA else torch.LongTensor
ByteTensor = torch.cuda.ByteTensor if USE_CUDA else torch.ByteTensor
```

Splitting the training dataset into batches.

```
def getBatch(batch_size, train_data):
    random.shuffle(train_data)
    sindex=0
    eindex=batch_size
    while eindex < len(train_data):
        batch = train_data[sindex: eindex]
        temp = eindex
        eindex = eindex + batch_size
        sindex = temp
        yield batch

    if eindex >= len(train_data):
        batch = train_data[sindex:]
        yield batch
```

Padding all the sentences in order to make all the sentences of same length. Hence adding a unique character after all the sentences until the maximum legnth of the sentence is reached

```
def pad_to_batch(batch, w_to_ix):
    fact,q,a = list(zip(*batch))
    max_fact = max([len(f) for f in fact])
    max_len = max([f.size(1) for f in flatten(fact)])
    max_q = max([qq.size(1) for qq in q])
    max_a = max([aa.size(1) for aa in a])

    facts, fact_masks, q_p, a_p = [], [], [], []
    for i in range(len(batch)):
        fact_p_t = []
        for j in range(len(fact[i])):
            if fact[i][j].size(1) < max_len:
                fact_p_t.append(torch.cat([fact[i][j], Variable(LongTensor([w_to_ix['<PAD>']] * (max_len - fact[i][j].size(1
            else:
                fact_p_t.append(fact[i][j])

        while len(fact_p_t) < max_fact:
            fact_p_t.append(Variable(LongTensor([w_to_ix['<PAD>']] * max_len)).view(1, -1))

        fact_p_t = torch.cat(fact_p_t)
        facts.append(fact_p_t)
        fact_masks.append(torch.cat([Variable(ByteTensor(tuple(map(lambda s: s ==0, t.data))), volatile=False) for t in fact

        if q[i].size(1) < max_q:
            q_p.append(torch.cat([q[i], Variable(LongTensor([w_to_ix['<PAD>']] * (max_q - q[i].size(1)))).view(1, -1)], 1))
        else:
            q_p.append(q[i])

        if a[i].size(1) < max_a:
            a_p.append(torch.cat([a[i], Variable(LongTensor([w_to_ix['<PAD>']] * (max_a - a[i].size(1)))).view(1, -1)], 1))
        else:
            a_p.append(a[i])

    questions = torch.cat(q_p)
    answers = torch.cat(a_p)
    question_masks = torch.cat([Variable(ByteTensor(tuple(map(lambda s: s ==0, t.data))), volatile=False) for t in questions

    return facts, fact_masks, questions, question_masks, answers
```

Preparing the sentences for the model input. If the word in not there in out vocabulary then is labelled as UNK

```
def prepare_sequence(seq, to_index):
    idxs = list(map(lambda w: to_index[w] if to_index.get(w) is not None else to_index["<UNK>"], seq))
    return Variable(LongTensor(idxs))
```

Reading the data from the text file which is used for training the model. s is added in front of all the sentence to demarkate starting of a new sentence

```
data = open('qa5_three-arg-relations_train.txt').readlines()
data = [d[:-1] for d in data]
train_data = []
fact=[]
qa=[]
for d in data:
    index=d.split(' ')[0]
    if(index=='1'):
        fact=[]
        qa=[]
    if('?' in d):
        temp = d.split('\t')
        ques = temp[0].strip().replace('?', '').split(' ')[1:] + ['?']
        ans=temp[1].split() + ['</s>']
        temp_s = deepcopy(fact)
        train_data.append([temp_s, ques, ans])
    else:
        fact.append(d.replace('.', '').split(' ')[1:] + ['</s>'])


fact,q,a = list(zip(*train_data))
vocab = list(set(flatten(flatten(fact)) + flatten(q) + flatten(a)))
```

Converting the sentences (text) into numbers

```
word_to_index={'<PAD>': 0, '<UNK>': 1, '<s>': 2, '</s>': 3}
for vo in vocab:
    if word_to_index.get(vo) is None:
        word_to_index[vo] = len(word_to_index)
index_to_word = {v:k for k, v in word_to_index.items()}


for s in train_data:
    for i, fact in enumerate(s[0]):
        s[0][i] = prepare_sequence(fact, word_to_index).view(1, -1)
    s[1] = prepare_sequence(s[1], word_to_index).view(1, -1)
    s[2] = prepare_sequence(s[2], word_to_index).view(1, -1)
```

Creating the dynamic neural network architecture using GRU and Linear Dense connected layers.

```
class DMN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dropout_p=0.1):
        super(DMN, self).__init__()

        self.hidden_size=hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.fact_gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.ques_gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.attn_weights = nn.Sequential(nn.Linear(4*hidden_size, hidden_size), nn.Tanh(), nn.Linear(hidden_size, 1), nn.Sc

        self.epsisodic_grucell = nn.GRUCell(hidden_size, hidden_size)
        self.memory_grucell = nn.GRUCell(hidden_size, hidden_size)
        self.ans_grucell = nn.GRUCell(2*hidden_size, hidden_size)

        self.ans_fc = nn.Linear(hidden_size, output_size)

        self.dropout = nn.Dropout(dropout_p)

    def init_hidden(self, inputs):
        hidden = Variable(torch.zeros(1, inputs.size(0), self.hidden_size))
        return hidden.cuda() if USE_CUDA else hidden

    def init_weight(self):
        nn.init.xavier_uniform(self.embedding.state_dict()['weight'])

        for name, param in self.fact_gru.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)
        for name, param in self.ques_gru.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)
        for name, param in self.attn_weights.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)
        for name, param in self.epsisodic_grucell.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)
        for name, param in self.memory_grucell.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)
        for name, param in self.ans_grucell.state_dict().items():
            if 'weight' in name: nn.init.xavier_normal(param)

        nn.init.xavier_normal(self.ans_fc.state_dict()['weight'])
        self.ans_fc.bias.data.fill_(0)

    def forward(self, facts, facts_masks, question, question_masks, num_decode, episodes=3, is_training=True):
        #input module
        concated=[]
        for fact, fact_mask in zip(facts, facts_masks):
            embedded = self.embedding(fact)
            if(is_training):
                embedded = self.dropout(embedded)
            hidden = self.init_hidden(fact)
            output, hidden = self.fact_gru(embedded, hidden)
            hidden_real = []
            for i, o in enumerate(output):
                length = fact_mask[i].data.tolist().count(0)
                hidden_real.append(o[length-1])
            concated.append(torch.cat(hidden_real).view(fact.size(0), -1).unsqueeze(0))
        encoded_facts = torch.cat(concated)
        #question module
        hidden=self.init_hidden(question)

        embedded = self.embedding(question)
        if(is_training):
                embedded = self.dropout(embedded)
        output, hidden = self.ques_gru(embedded, hidden)

        if is_training == True:
```

```
            real_question = []
            for i, o in enumerate(output): # B,T,D
                real_length = question_masks[i].data.tolist().count(0)

                real_question.append(o[real_length - 1])

            encoded_question = torch.cat(real_question).view(questions.size(0), -1) # B,D
        else: # for inference mode
            encoded_question = hidden.squeeze(0) # B,D

        #episodic memory module

        memory = encoded_question
        T_C = encoded_facts.size(1)
        B = encoded_facts.size(0)
        for i in range(episodes):
            hidden = self.init_hidden(encoded_facts.transpose(0, 1)[0]).squeeze(0) # B,D
            for t in range(T_C):

                z = torch.cat([
                                encoded_facts.transpose(0, 1)[t] * encoded_question, # B,D , element-wise product
                                encoded_facts.transpose(0, 1)[t] * memory, # B,D , element-wise product
                                torch.abs(encoded_facts.transpose(0,1)[t] - encoded_question), # B,D
                                torch.abs(encoded_facts.transpose(0,1)[t] - memory) # B,D
                            ], 1)
                g_t = self.attn_weights(z) # B,1 scalar
                hidden = g_t * self.epsisodic_grucell(encoded_facts.transpose(0, 1)[t], hidden) + (1 - g_t) * hidden

            e = hidden
            memory = self.memory_grucell(e, memory)

        # Answer Module
        answer_hidden = memory
        start_decode = Variable(LongTensor([[word_to_index['<s>']] * memory.size(0)])).transpose(0, 1)
        y_t_1 = self.embedding(start_decode).squeeze(1) # B,D

        decodes = []
        for t in range(num_decode):
            answer_hidden = self.ans_grucell(torch.cat([y_t_1, encoded_question], 1), answer_hidden)
            decodes.append(F.log_softmax(self.ans_fc(answer_hidden),1))
        return torch.cat(decodes, 1).view(B * num_decode, -1)
```

Defining the hyperparameters

```
HIDDEN_SIZE = 80
BATCH_SIZE = 64
LR = 0.001
EPOCH = 50
NUM_EPISODE = 3
EARLY_STOPPING = False
```

Initializing the DMN model.

```
model = DMN(len(word_to_index), HIDDEN_SIZE, len(word_to_index))
model.init_weight()
if USE_CUDA:
    model = model.cuda()

loss_function = nn.CrossEntropyLoss(ignore_index=0)
optimizer = optim.Adam(model.parameters(), lr=LR)
```

Training the model on the dataset and using CrossEntropyLoss along with Adam optimizer.

```
 for epoch in range(EPOCH):
    losses = []
    if EARLY_STOPPING:
        break

    for i,batch in enumerate(getBatch(BATCH_SIZE, train_data)):
        facts, fact_masks, questions, question_masks, answers = pad_to_batch(batch, word_to_index)

        model.zero_grad()
        pred = model(facts, fact_masks, questions, question_masks, answers.size(1), NUM_EPISODE, True)
        loss = loss_function(pred, answers.view(-1))
        losses.append(loss.data.tolist()[0])

        loss.backward()
        optimizer.step()
```

```
        if i % 100 == 0:
            print("[%d/%d] mean_loss : %0.2f" %(epoch, EPOCH, np.mean(losses)))

            if np.mean(losses) < 0.01:
                EARLY_STOPPING = True
                print("Early Stopping!")
                break
            losses = []
```

```
/usr/local/lib/python3.6/dist-packages/torch/nn/modules/container.py:67: UserWarning: Implicit dimension choice for soft
  input = module(input)
[0/50] mean_loss : 3.83
[0/50] mean_loss : 1.29
[1/50] mean_loss : 0.69
[1/50] mean_loss : 0.65
[2/50] mean_loss : 0.65
[2/50] mean_loss : 0.65
[3/50] mean_loss : 0.65
[3/50] mean_loss : 0.65
[4/50] mean_loss : 0.63
[4/50] mean_loss : 0.63
[5/50] mean_loss : 0.61
[5/50] mean_loss : 0.62
[6/50] mean_loss : 0.61
[6/50] mean_loss : 0.62
[7/50] mean_loss : 0.61
[7/50] mean_loss : 0.62
[8/50] mean_loss : 0.60
[8/50] mean_loss : 0.61
[9/50] mean_loss : 0.59
[9/50] mean_loss : 0.61
[10/50] mean_loss : 0.51
[10/50] mean_loss : 0.49
[11/50] mean_loss : 0.39
[11/50] mean_loss : 0.39
[12/50] mean_loss : 0.35
[12/50] mean_loss : 0.39
[13/50] mean_loss : 0.40
[13/50] mean_loss : 0.36
[14/50] mean_loss : 0.29
[14/50] mean_loss : 0.31
[15/50] mean_loss : 0.33
[15/50] mean_loss : 0.30
[16/50] mean_loss : 0.29
[16/50] mean_loss : 0.27
[17/50] mean_loss : 0.24
[17/50] mean_loss : 0.18
[18/50] mean_loss : 0.13
[18/50] mean_loss : 0.15
[19/50] mean_loss : 0.16
[19/50] mean_loss : 0.14
[20/50] mean_loss : 0.12
[20/50] mean_loss : 0.14
[21/50] mean_loss : 0.17
[21/50] mean_loss : 0.14
[22/50] mean_loss : 0.12
[22/50] mean_loss : 0.14
[23/50] mean_loss : 0.17
[23/50] mean_loss : 0.13
[24/50] mean_loss : 0.08
[24/50] mean_loss : 0.08
[25/50] mean_loss : 0.03
[25/50] mean_loss : 0.03
[26/50] mean_loss : 0.01
Early Stopping!
```

```
torch.save(model, 'DMN.pkl')
# Uncomment to load the existing model
# model = torch.load('DMN.pkl')
```

```
/usr/local/lib/python3.6/dist-packages/torch/serialization.py:158: UserWarning: Couldn't retrieve source code for contai
  "type " + obj.__name__ + ". It won't be checked "
```

Creating a function for padding the new incoming text for predictions

```
def pad_to_fact(fact, x_to_ix): # this is for inference

    max_x = max([s.size(1) for s in fact])
    x_p = []
    for i in range(len(fact)):
        if fact[i].size(1) < max_x:
            x_p.append(torch.cat([fact[i], Variable(LongTensor([x_to_ix['<PAD>']] * (max_x - fact[i].size(1))))).view(1, -1)]
        else:
            x_p.append(fact[i])
```

```
    fact = torch.cat(x_p)
    fact_mask = torch.cat([Variable(ByteTensor(tuple(map(lambda s: s ==0, t.data))), volatile=False) for t in fact]).view(fa
    return fact, fact_mask
```

Reading the test prediction file applying the same pre processing on the test data.

```
data = open('qa5_three-arg-relations_test.txt').readlines()
data = [d[:-1] for d in data]
test_data = []
fact=[]
qa=[]
for d in data:
    index=d.split(' ')[0]
    if(index=='1'):
        fact=[]
        qa=[]
    if('?' in d):
        temp = d.split('\t')
        ques = temp[0].strip().replace('?', '').split(' ')[1:] + ['?']
        ans=temp[1].split() + ['</s>']
        temp_s = deepcopy(fact)
        test_data.append([temp_s, ques, ans])
    else:
        fact.append(d.replace('.', '').split(' ')[1:] + ['</s>'])


for t in test_data:
    for i, fact in enumerate(t[0]):
        t[0][i] = prepare_sequence(fact, word_to_index).view(1, -1)

    t[1] = prepare_sequence(t[1], word_to_index).view(1, -1)
    t[2] = prepare_sequence(t[2], word_to_index).view(1, -1)
```

Checking the accuracy of the model on the testing data.

```
accuracy = 0
for t in test_data:
    fact, fact_mask = pad_to_fact(t[0], word_to_index)
    question = t[1]
    question_mask = Variable(ByteTensor([0] * t[1].size(1)), requires_grad=False).unsqueeze(0)
    answer = t[2].squeeze(0)

    model.zero_grad()
    pred = model([fact], [fact_mask], question, question_mask, answer.size(0), NUM_EPISODE, False)
    if pred.max(1)[1].data.tolist() == answer.data.tolist():
        accuracy += 1

print(accuracy/len(test_data) * 100)
```

```
/usr/local/lib/python3.6/dist-packages/torch/nn/modules/container.py:67: UserWarning: Implicit dimension choice for soft
  input = module(input)
97.89999999999999
```

```
t = random.choice(test_data)
fact, fact_mask = pad_to_fact(t[0], word_to_index)
question = t[1]
question_mask = Variable(ByteTensor([0] * t[1].size(1)), requires_grad=False).unsqueeze(0)
answer = t[2].squeeze(0)

model.zero_grad()
pred = model([fact], [fact_mask], question, question_mask, answer.size(0), NUM_EPISODE, False)

print("Facts : ")
print('\n'.join([' '.join(list(map(lambda x: index_to_word[x],f))) for f in fact.data.tolist()]))
print("")
print("Question : ",' '.join(list(map(lambda x: index_to_word[x], question.data.tolist()[0]))))
print("")
print("Answer : ",' '.join(list(map(lambda x: index_to_word[x], answer.data.tolist()))))
print("Prediction : ",' '.join(list(map(lambda x: index_to_word[x], pred.max(1)[1].data.tolist()))))
```

```
Facts :
Mary moved to the hallway </s> <PAD>
Jeff moved to the office </s> <PAD>
Jeff grabbed the football there </s> <PAD>
Bill moved to the bathroom </s> <PAD>
Mary travelled to the bathroom </s> <PAD>
Mary went to the kitchen </s> <PAD>
Bill travelled to the hallway </s> <PAD>
Jeff put down the football </s> <PAD>
```

```
Mary moved to the bathroom </s> <PAD>
Jeff journeyed to the garden </s> <PAD>
Jeff travelled to the bathroom </s> <PAD>
Fred went to the hallway </s> <PAD>
Fred went to the bedroom </s> <PAD>
Bill grabbed the milk there </s> <PAD>
Fred travelled to the office </s> <PAD>
Bill put down the milk </s> <PAD>
Fred picked up the football there </s>
Bill got the milk there </s> <PAD>
Jeff went back to the hallway </s>
Bill handed the milk to Jeff </s>
Fred travelled to the garden </s> <PAD>
Jeff passed the milk to Bill </s>

Question :  Who gave the milk to Bill ?

Answer :  Jeff </s>
Prediction :  Jeff </s>
/usr/local/lib/python3.6/dist-packages/torch/nn/modules/container.py:67: UserWarning: Implicit dimension choice for soft
  input = module(input)
```

Start coding or generate with AI.