1. config.py

# -\*- coding: utf-8 -\*-

'''

Filename : config.py

Function : All the parameter settings used in TransText model, users could change any as they need.

'''

import torch

from code.utils.utils import CheckPath

class Config():

def \_\_init\_\_(self):

# Input&Output data arguments

self.inputPath = './data/Input/'

self.entity2idFile = './data/Input/entity2id.txt'

self.relation2idFile = './data/Input/relation2id.txt'

self.triplesFile = './data/Input/train.txt'

self.pretrainW2VFile = './data/Input/pretrain\_w2v.txt'

self.pretrainE2VFile = './data/Input/pretrain\_e2v.txt'

self.pretrainR2VFile = './data/Input/pretrain\_r2v.txt'

self.pretrainEncoder = "./data/model/TransText\_cnn.param"

self.pretrainModel = "./data/model/TransText.param"

self.outputPath = './data/Output/'

self.saveW2VFile = './data/Output/word2vec.txt'

self.saveE2VFile = './data/Output/entity2vec.txt'

self.saveR2VFile = './data/Output/relation2vec.txt'

# Middle source path/argument

self.trainDataPath = './source/train\_triples.pkl'

self.evalDataPath = './source/eval\_triples.pkl'

self.parameterPath = './source/parameters.pkl'

self.modelPath = "./source/model/"

# Dataloader arguments

self.batchSize = 1024

self.shuffle = True

self.numWorkers = 0

self.dropLast = False

self.repProba = 0.5

#self.exproba = 0.5

# paramters:

self.WORD\_EMB\_DIM = 100

self.KG\_EMB\_DIM = 100

self.splitRate = 0.1

self.evaluate = False

# Model and training general arguments

self.TransText = {"EmbeddingDim": 100,

"KgDim": 100,

"Margin": 1.0,

"Alpha": 0.01,

"L": 2}

# Encoder and training general arguments

self.lstm = {"hiddenDim": 100,

"biDirection": True}

self.cnn = {"hiddenDim": 100,

"kernelSizes": [3,4,5]}

self.maxSentenceLen = 100

self.maxSentenceNum = 5

self.usegpu = torch.cuda.is\_available()

self.modelName = "TransText"

self.encoderName = 'cnn'

self.optimizer = "Adam"

self.evalMethod = "MR"

self.simMeasure = "L2"

self.modelSaveType = "param"

self.weightDecay = 0

self.epochs = 500

self.evalEpoch = 1

self.learningRate = 0.01

self.lrdecay = 0.96

self.lrdecayEpoch = 5

self.loadEncoder = False

# Check Path

self.CheckPath()

def CheckPath(self):

# Check files

CheckPath(self.entity2idFile)

CheckPath(self.relation2idFile)

CheckPath(self.triplesFile)

# Check dirs

CheckPath(self.inputPath, raise\_error=False)

CheckPath(self.outputPath, raise\_error=False)

CheckPath(self.modelPath, raise\_error=False)

2. GenerateData.py

# -\*- coding:utf-8 -\*-

'''

Filename : GenerateData.py

Function : Package all the data into a pkl file

Pacakge all the parameters/embeddings into a pkl file

'''

from code.process.LoadData import loadTerm2id, loadTrainTriples, loadPretrainW2V, loadPretrainE2V, splitData

import pickle

from config import Config

if \_\_name\_\_ == '\_\_main\_\_':

args = Config()

# step 1 Transform raw input to standard format

# track 1 load entity2id

print ('INFO : loading entity2id...')

entityDict = loadTerm2id (dataPath = args.entity2idFile)

# track 2 load relation2id

print ('INFO : loading relation2id...')

relationDict = loadTerm2id (dataPath = args.relation2idFile)

# track 3 load Triples and translate them into ID for training

print ('INFO : loading all training triples...')

allTriples,wordDict = loadTrainTriples (

dataPath = args.triplesFile,

entityDict = entityDict,

relationDict = relationDict,

wordDict = {}

)

# track 4 load pretrained word2vec

print ('INFO : loading pretrained word2vec...')

wordEmbedding = loadPretrainW2V(

embedPath= args.pretrainW2VFile,

wordDict = wordDict,

embeddingDim = args.WORD\_EMB\_DIM

)

# track 5 load pretrained entity2vec

print ('INFO : loading pretrained entity2vec...')

entityEmbedding = loadPretrainE2V(

embedPath = args.pretrainE2VFile,

entityDict = entityDict,

embeddingDim = args.KG\_EMB\_DIM

)

# track 6 load pretrained relation2vec

print ('INFO : loading pretrained relation2vec...')

relationEmbedding = loadPretrainE2V(

embedPath = args.pretrainR2VFile,

entityDict = relationDict,

embeddingDim = args.KG\_EMB\_DIM

)

#step 2 Split Data:

if args.evaluate:

print ('INFO : spliting data into training set and testing set...')

allTriples,evalTriples = splitData(

allTriples = allTriples,

splitRate = args.splitRate)

pickle.dump (evalTriples,open(args.evalDataPath,'wb'))

print ('INFO : dumping all the preprocessing data...')

#step 3 Dump all the file into PKL

pickle.dump (allTriples,open(args.trainDataPath,'wb'))

pickle.dump ([wordEmbedding,entityEmbedding,relationEmbedding,wordDict,entityDict,relationDict],open(args.parameterPath, 'wb'))

3. Train.py

# -\*- coding :utf-8 -\*-

'''

Filename : Train.py

Function : Train the triples

Stage1: prepare the data

Stage2: create the model

Stage3: fit the model

Stage4: evaluate on the validation set if nesessary (MR)

Stage5: save the model with the best MR or the last epoch

'''

import os

import json

import torch

import codecs

import pickle

import argparse

import numpy as np

from config import Config

from code.utils import utils

from code.models import TransText

from code.utils import evaluation

from code.dataloader.dataloader import tripleDataset

from code.utils.TensorDevice import LongTensorDevice,FloatTensorDevice,VariableDevice,ParameterDevice

from torch.utils.data import DataLoader

from torch.autograd import Variable

from tensorboardX import SummaryWriter

os.environ['CUDA\_VISIBLE\_DEVICES']='0'

args = Config()

'''

Generate the dataloader: training, has negative sample generations

'''

def prepareDataloader(args):

# Initialize dataset and dataloader

dataset = tripleDataset(dataPath=args.trainDataPath)

# entityDictPath=args.entpath,

# relationDictPath=args.relpath)

dataset.paddingSentences(maxLen = args.maxSentenceLen,

maxSentenceNum=args.maxSentenceNum) #padding

dataset.generateNegSamples(changeProba=args.repProba)

dataloader = DataLoader(dataset,

batch\_size=args.batchSize,

shuffle=args.shuffle,

num\_workers=args.numWorkers,

drop\_last=args.dropLast)

return dataloader

def prepareEvalDataloader(args):

dataset = tripleDataset(dataPath=args.evalDataPath)

dataset.paddingSentences(maxLen = args.maxSentenceLen,

maxSentenceNum=args.maxSentenceNum) #padding

dataloader = DataLoader(dataset,

batch\_size=args.batchSize,

shuffle=False,

drop\_last=False)

return dataloader

'''

cut off the learning rate every i epoch

'''

def adjust\_learning\_rate(optimizer, decay):

for param\_group in optimizer.param\_groups:

param\_group['lr'] \*= decay

'''

class of training stage

'''

class trainTriples():

def \_\_init\_\_(self, args):

self.args = args

def prepareData(self):

print("INFO : Prepare dataloader")

self.dataloader = prepareDataloader(self.args)

if self.args.evaluate:

self.evalloader = prepareEvalDataloader(self.args)

def prepareModel(self):

print("INFO : Init model %s"%self.args.modelName)

if self.args.modelName == "TransText":

self.model = TransText.TransText(parameterPath = self.args.parameterPath,

encoderName = self.args.encoderName,

embeddingDim=self.args.TransText["EmbeddingDim"],

kgDim = self.args.TransText["KgDim"],

margin=self.args.TransText["Margin"],

L=self.args.TransText["L"],

alpha= self.args.TransText["Alpha"])

else:

print("ERROR : No model named %s"%self.args.modelName)

exit(1)

if self.args.usegpu:

self.model.cuda()

def loadPretrainEncoder(self):

if self.args.modelName == "TransText":

print("INFO : Loading pre-training entity and relation embedding!")

self.model.initialWeight(parameterPath = self.args.parameterPath)

else:

print("ERROR : Model %s is not supported!"%self.args.modelName)

exit(1)

def loadPretrainModel(self):

if self.args.modelName == "TransText":

print("INFO : Loading pre-training model.")

modelType = os.path.splitext(self.args.premodel)[-1]

if modelType == ".param":

self.model.load\_state\_dict(torch.load(self.args.premodel))

elif modelType == ".model":

self.model = torch.load(self.args.premodel)

else:

print("ERROR : Model type %s is not supported!")

exit(1)

else:

print("ERROR : Model %s is not supported!" % self.args.modelName)

exit(1)

def fit(self):

EPOCHS = self.args.epochs

LR = self.args.learningRate

OPTIMIZER = self.args.optimizer

if OPTIMIZER == "Adam":

optimizer = torch.optim.Adam(self.model.parameters(),

weight\_decay=self.args.weightDecay,

lr=LR)

else:

print("ERROR : Optimizer %s is not supported."%OPTIMIZER)

exit(1)

# Training, GLOBALSTEP and GLOBALEPOCH are used for summary

minLoss = float("inf")

bestMR = float("inf")

GLOBALSTEP = 0

GLOBALEPOCH = 0

seed = 2019

print("INFO : Using seed %d" % seed)

for epoch in range(EPOCHS):

GLOBALEPOCH += 1

STEP = 0

print("="\*20+"EPOCHS(%d/%d)"%(epoch+1, EPOCHS)+"="\*20)

for posX, negX, sentencesX, sentenceMasking, headMasking, tailMasking in self.dataloader:

# Allocate tensor to devices

posX = LongTensorDevice (posX, use\_cuda = args.usegpu)

negX = LongTensorDevice (negX,use\_cuda = args.usegpu)

sentencesX = LongTensorDevice (sentencesX,use\_cuda = args.usegpu)

sentenceMasking = LongTensorDevice (sentenceMasking,use\_cuda = args.usegpu)

headMasking = LongTensorDevice (headMasking,use\_cuda = args.usegpu)

tailMasking = LongTensorDevice (tailMasking,use\_cuda = args.usegpu)

# Normalize the embedding if neccessary

# self.model.normalizeEmbedding()

# Calculate the loss from the model

loss = self.model(posX, negX, sentencesX, sentenceMasking, headMasking, tailMasking)

if self.args.usegpu:

lossVal = loss.cpu().item()

else:

lossVal = loss.item()

# Calculate the gradient and step down

optimizer.zero\_grad()

loss.backward()

optimizer.step()

# Print infomation and add to summary

if minLoss > lossVal:

minLoss = lossVal

print("[TRAIN-EPOCH(%d/%d)-STEP(%d)]Loss:%.4f, minLoss:%f"%(epoch+1, EPOCHS, STEP, lossVal, minLoss))

STEP += 1

GLOBALSTEP += 1

if GLOBALEPOCH % self.args.lrdecayEpoch == 0:

adjust\_learning\_rate(optimizer, decay=self.args.lrdecay)

if GLOBALEPOCH % self.args.evalEpoch == 0 and self.args.evaluate:

MR = evaluation.MREvaluation(evalloader=self.evalloader,

modelName=self.args.modelName,

simMeasure=args.simMeasure,

model = self.model,

usegpu = self.args.usegpu)

print("[EVALUATION-EPOCH(%d/%d)]Measure method %s, eval %.4f"% \

(epoch+1, EPOCHS, self.args.evalMethod, MR))

# Save the model if new MR is better

if MR < bestMR:

bestMR = MR

self.saveModel()

self.dumpEmbedding()

if not self.args.evaluate:

self.saveModel()

self.dumpEmbedding()

def saveModel(self):

if self.args.modelSaveType == "param":

path = os.path.join(self.args.modelPath, "{}\_ent{}\_rel{}.param".format(self.args.modelName, getattr(self.args, self.args.modelName)["EmbeddingDim"], getattr(self.args, self.args.modelName)["EmbeddingDim"]))

torch.save(self.model.state\_dict(), path)

elif self.args.modelSaveType == "full":

path = os.path.join(self.args.modelPath, "{}\_ent{}\_rel{}.model".format(self.args.modelName, getattr(self.args, self.args.modelName)["EmbeddingDim"], getattr(self.args, self.args.modelName)["EmbeddingDim"]))

torch.save(self.model, path)

else:

print("ERROR : Saving mode %s is not supported!"%self.args.modelSave)

exit(1)

def dumpEmbedding(self):

# save word embedding, entity embedding and relation embedding as txt file

entWeight = self.model.entityEmbedding.weight.detach().cpu().numpy()

relWeight = self.model.relationEmbedding.weight.detach().cpu().numpy()

worWeight = self.model.wordEmbedding.weight.detach().cpu().numpy()

entityNum, entityDim = entWeight.shape

relationNum, relationDim = relWeight.shape

wordNum, wordDim = worWeight.shape

entsave = os.path.join(self.args.saveE2VFile)

relsave = os.path.join(self.args.saveR2VFile)

worsave = os.path.join(self.args.saveW2VFile)

with codecs.open(entsave, "w", encoding="utf-8") as fp:

fp.write("{} {}\n".format(entityNum, entityDim))

for embed in entWeight:

fp.write("{}\n".format(" ".join(embed.astype(np.str))))

with codecs.open(relsave, "w", encoding="utf-8") as fp:

fp.write("{} {}\n".format(relationNum, relationDim))

for embed in relWeight:

fp.write("{}\n".format(" ".join(embed.astype(np.str))))

with codecs.open(worsave, "w", encoding="utf-8") as fp:

fp.write("{} {}\n".format(wordNum, wordDim))

for word in self.model.wordDict:

embed = worWeight[self.model.wordDict[word]]

fp.write("{} {}\n".format(word," ".join(embed.astype(np.str))))

# save the encoder parameter

encodersave = os.path.join(self.args.outputPath, self.args.encoderName + ".param")

torch.save (self.model.encoder.state\_dict(),encodersave)

if \_\_name\_\_ == "\_\_main\_\_":

# Print args

utils.printArgs(args)

# Create and train model

trainModel = trainTriples(args)

trainModel.prepareData()

trainModel.prepareModel()

if args.loadEncoder:

trainModel.loadPretrainEncoder()

trainModel.fit()

4. dataloader.py

# -\*- coding: utf-8 -\*-

'''

Filename : dataloader.py

Function : pack the data into dataloader for batch learning

'''

import time

import sys

import math

import json

import numpy as np

import pandas as pd

import pickle

from torch.utils.data import \*

from code.utils.TensorDevice import LongTensorDevice,FloatTensorDevice,VariableDevice,ParameterDevice

from tqdm import tqdm

import copy

class tripleDataset(Dataset):

def \_\_init\_\_(self, dataPath):

super(Dataset, self).\_\_init\_\_()

# Load entity-index dict and relation-index dict

allTriples = pickle.load(open(dataPath,'rb'),encoding = 'iso-8859-1')

self.triples = allTriples

# Generate the positive instance

self.positiveSamples = [[triple.head,triple.relation,triple.tail] for triple in self.triples]

'''

Name: paddingSentences

Function: padding all the sentences to a fixed length

padding all the triples to a fixed number of sentences

generate the position embedding of head entity & tail entity in sentences

generate the masking tensor of sentences & headEntities & tailEntities

'''

def paddingSentences(self,maxLen=0,maxSentenceNum=0):

# if the maxLen or maxSentenceNum not given, calculate the max one

tempMaxLen = 0

tempMaxSentenceNum = 0

self.sentenceSamples = []

self.sentenceMaskings = []

self.headEntityMaskings = []

self.tailEntityMaskings = []

print ('INFO: Calculate the max sentence number and the max sentence length.')

for triple in tqdm(self.triples):

if len(triple.instance) > tempMaxSentenceNum:

tempMaxSentenceNum = len (triple.instance)

for instance in triple.instance:

if len(instance.sentence) > tempMaxLen:

tempMaxLen = len(instance.sentence)

if not maxLen:

maxLen = tempMaxLen

if not maxSentenceNum:

maxSentenceNum = tempMaxSentenceNum

# padding sentence with zero

print ('INFO: Generating and padding the sentences to the max sentence number and max sentence length.')

for triple in tqdm(self.triples):

newTriple = []

sMaskings = []

headEntityMasking = []

tailEntityMasking = []

for j,instance in enumerate(triple.instance):

if maxLen>=len(instance.sentence):

newSentence = instance.sentence + [0]\*(maxLen-len(instance.sentence))

sMasking = [1]\*len(instance.sentence) + [0]\*(maxLen-len(instance.sentence))

else:

newSentence = instance.sentence[:maxLen]

sMasking = [1]\*maxLen

headMasking = [0]\*maxLen

tailMasking = [0]\*maxLen

try:

headMasking [instance.offsetStartHead:min(maxLen,instance.offsetEndHead)] = [1]\* (min(maxLen,instance.offsetEndHead) - instance.offsetStartHead)

except:

pass

try:

tailMasking [instance.offsetStartTail:min(maxLen,instance.offsetEndTail)] = [1]\* (min(maxLen,instance.offsetEndTail) - instance.offsetStartTail)

except:

pass

newTriple.append(newSentence)

sMaskings.append(sMasking)

headEntityMasking.append (headMasking)

tailEntityMasking.append (tailMasking)

if j >= maxSentenceNum - 1:

break

if len(triple.instance) < maxSentenceNum:

for j in range (maxSentenceNum-len(triple.instance)):

newTriple.append([0]\*maxLen)

sMaskings.append([0]\*maxLen)

headEntityMasking.append ([0]\*maxLen)

tailEntityMasking.append ([0]\*maxLen)

self.sentenceSamples.append(newTriple) #sampleNum \* sentenceNum \* sentenceLen

self.sentenceMaskings.append (sMaskings)

self.headEntityMaskings.append(headEntityMasking)

self.tailEntityMaskings.append(tailEntityMasking)

def generateNegSamples(self, changeProba=0.5): #, headSeed=0, tailSeed=0):

assert changeProba >= 0 and changeProba <= 1.0

# Generate negtive samples from positive samples

print("INFO : Generate negtive samples from positive samples.")

self.negativeSamples = copy.deepcopy(self.positiveSamples)

# Replacing head or tail

allSampleNum = range(len(self.positiveSamples))

for i,sample in enumerate(self.negativeSamples):

headProbaDistribution = np.random.uniform(low=0.0, high=1.0)

if headProbaDistribution < changeProba:

shuffleHead = np.random.choice(allSampleNum)

self.negativeSamples[i][0] = self.positiveSamples[shuffleHead][0]

else:

shuffleTail = np.random.choice(allSampleNum)

self.negativeSamples[i][2] = self.positiveSamples[shuffleTail][2]

'''

Used to transform CSV data to index-form

==> csvData : Input CSV data

==> repDict : A dict like {column\_name : dict(entity\_dict)}.

The keys are names of the csv columns, the corresponding

value is entity/relation dictionary which used to transform

entity/realtion to index.

'''

def \_\_len\_\_(self):

return len(self.positiveSamples)

def \_\_getitem\_\_(self, item):

if hasattr(self, "negativeSamples"):

return np.array(self.positiveSamples[item]), np.array(self.negativeSamples[item]),\

np.array(self.sentenceSamples[item]), np.array(self.sentenceMaskings[item]),\

np.array(self.headEntityMaskings[item]), np.array(self.tailEntityMaskings[item])

else:

return np.array(self.positiveSamples[item]), np.array(self.sentenceSamples[item]),\

np.array(self.sentenceMaskings[item]), np.array(self.headEntityMaskings[item]),\

np.array(self.tailEntityMaskings[item]),

5. TransText.py

# -\*- coding: utf-8 -\*-

'''

Filename : TransText.py

Function : Define the knowledge representation learning model

Input: Positive triples in knowledge base

Generated negative triples

A bag of sentences for each triple

Sentences&entities masking of each triple

'''

import torch

import codecs

import numpy as np

import pickle

import torch.nn as nn

import torch.nn.functional as F

from config import Config

from code.models.Encoder import cnnEncoder,lstmEncoder

from code.utils.TensorDevice import FloatTensorDevice,LongTensorDevice,VariableDevice,ParameterDevice

args = Config()

class TransText(nn.Module):

def \_\_init\_\_(self, parameterPath, encoderName, embeddingDim,kgDim,usegpu = True, margin=1.0, L=2,alpha = 0.01):

super(TransText, self).\_\_init\_\_()

assert (L == 1 or L == 2)

self.model = "TransText"

self.margin = margin

self.L = L

self.alpha = alpha

self.usegpu = usegpu

wordEmbedding,entityEmbedding,relationEmbedding,wordDict,entityDict,relationDict = pickle.load (open(parameterPath,'rb'),encoding = 'iso-8859-1')

self.wordDict = wordDict

self.entityDict = entityDict

self.relationDict = relationDict

self.entityEmbedding = nn.Embedding(num\_embeddings=entityEmbedding.shape[0],

embedding\_dim=embeddingDim)

self.entityEmbedding.weight = ParameterDevice(Tensor=entityEmbedding,

requires\_grad = True,

use\_cuda =usegpu)

self.relationEmbedding = nn.Embedding(num\_embeddings=relationEmbedding.shape[0],

embedding\_dim=embeddingDim)

self.relationEmbedding.weight = ParameterDevice(Tensor=relationEmbedding,

requires\_grad = True,

use\_cuda =usegpu)

self.wordEmbedding = nn.Embedding(num\_embeddings=wordEmbedding.shape[0],

embedding\_dim=embeddingDim)

self.wordEmbedding.weight = ParameterDevice(Tensor=wordEmbedding,

requires\_grad = True,

use\_cuda =usegpu)

self.distfn = nn.PairwiseDistance(L)

self.mseLoss = nn.MSELoss()

if encoderName == 'lstm':

self.encoder = lstmEncoder (embeddingDim = embeddingDim,

biDirection = args.lstm['biDirection'],

hiddenDim = args.lstm['hiddenDim'],

kgDim = kgDim,

usegpu = usegpu)

elif encoderName == 'cnn':

self.encoder = cnnEncoder (embeddingDim = embeddingDim,

kernelSizes = args.cnn['kernelSizes'],

hiddenDim = args.cnn['hiddenDim'],

kgDim = kgDim,

usegpu = usegpu)

else:

print("ERROR : No encoder named %s"%encoderName)

exit(1)

'''

This function used to calculate score, steps follows:

Step1: Split input as head, relation and tail index column

Step2: Transform index tensor to embedding tensor

Step3: Calculate MSE loss of text encoded entity/relation embeddings and triple encoded entity/relation embeddings

Step4: Calculate distance as final score

'''

def scoreOp(self, inputTriple, sentenceFeature,headEntityFeature = None,tailEntityFeature = None):

# Step1

# head : shape(batch\_size, 1)

# relation : shape(batch\_size, 1)

# tail : shape(batch\_size, 1)

batchSize = inputTriple.size()[0]

head, relation, tail = torch.chunk(input=inputTriple,

chunks=3,

dim=1)

# Step2

# head : shape(batch\_size, 1, embedDim)

# relation : shape(batch\_size, 1, embedDim)

# tail : shape(batch\_size, 1, embedDim)

head = torch.squeeze(self.entityEmbedding(head), dim=1)

tail = torch.squeeze(self.entityEmbedding(tail), dim=1)

relation = torch.squeeze(self.relationEmbedding(relation), dim=1)

textRelation = torch.squeeze(sentenceFeature, dim=1)

# Step 3

# Calculate loss of the alignment

# if there is no entity (cut off), replace it with an all-zero embeddings

try:

outputHead = self.mseLoss(head,headEntityFeature)

except:

outputHead = VariableDevice(Tensor = np.zeros(batchSize),use\_cuda = self.usegpu,requires\_grad = False)

try:

outputTail = self.mseLoss(tail,tailEntityFeature)

except:

outputTail = VariableDevice(Tensor = np.zeros(batchSize),use\_cuda = self.usegpu,requires\_grad = False)

outputRelation = self.mseLoss(relation,textRelation)

# Step4 and Step4

# output : shape(batch\_size, embedDim) ==> shape(batch\_size, 1)

outputEntity = self.distfn(head+relation, tail)

return outputEntity, outputHead,outputTail, outputRelation

'''

In every training epoch, the entity embedding could be normalize

Step1: Get numpy.array from embedding weight

Step2: Normalize array

Step3: Assign normalized array to embedding

'''

def normalizeEmbedding(self):

embedWeight = self.entityEmbedding.weight.detach().cpu().numpy()

embedWeight = embedWeight / np.sqrt(np.sum(np.square(embedWeight), axis=1, keepdims=True))

self.entityEmbedding.weight.data.copy\_(torch.from\_numpy(embedWeight))

# embedWeight = self.relationEmbedding.weight.detach().cpu().numpy()

# embedWeight = embedWeight / np.sqrt(np.sum(np.square(embedWeight), axis=1, keepdims=True))

# self.relationEmbedding.weight.data.copy\_(torch.from\_numpy(embedWeight))

'''

Input:

posX : (torch.tensor)The positive triples tensor, shape(batch\_size, 3)

negX : (torch.tensor)The negtive triples tensor, shape(batch\_size, 3)

posSentence : (torch.tensor)The sentences tensor, shape(batch\_size, sentence\_num,max\_len)

sentenceMasking : (torch.tensor)The sentence Masking tensor, where the padding words are 0, shape(batch\_size, sentence\_num,max\_len)

headMasking : (torch.tensor)The head entity masking tensor, where the words in head entity are 1, shape(batch\_size, sentence\_num,max\_len)

tailMasking : (torch.tensor)The tail entity masking tensor, where the words in tail entity are 1, shape(batch\_size, sentence\_num,max\_len)

'''

def forward(self, posX, negX, posSentence, sentenceMasking, headMasking, tailMasking):

size = posX.size()[0]

# Calculate score

posSentence = self.wordEmbedding(posSentence)

sentenceFeature,headEntityFeature,tailEntityFeature = self.encoder(posSentence,sentenceMasking,headMasking,tailMasking)

posEntityScore, headScoreLoss, tailScoreLoss, relationScoreLoss = self.scoreOp(posX,sentenceFeature,headEntityFeature,tailEntityFeature)

negEntityScore, \_, \_,\_ = self.scoreOp(negX,sentenceFeature)

# Get margin ranking loss

lossEntity = torch.sum(F.relu(input=posEntityScore-negEntityScore+self.margin))/size

lossHead = torch.sum(headScoreLoss)/size

lossTail = torch.sum(tailScoreLoss)/size

lossRelation = torch.sum(relationScoreLoss)/size

return lossEntity + (lossHead + lossTail + lossRelation) \* self.alpha

'''

Evaluation forward:

without negative instance,calculate the distance between entity1&relation&entity2, igonore the alignment loss

'''

def evalForward(self, posX, posSentence, sentenceMasking, headMasking, tailMasking):

size = posX.size()[0]

# Calculate score

posSentence = self.wordEmbedding(posSentence)

sentenceFeature,headEntityFeature,tailEntityFeature = self.encoder(posSentence,sentenceMasking,headMasking,tailMasking)

posEntityScore, \_,\_,\_ = self.scoreOp(posX,sentenceFeature,headEntityFeature,tailEntityFeature)

# Get margin ranking loss

return posEntityScore

'''

Used to load pretraining encoder.

'''

def initialWeight(self, encoderPath):

print("INFO : Loading pretrained encoder.")

self.encoder.load\_state\_dict(torch.load(encoderPath))

6. Encoder.py

# -\*- coding: utf-8 -\*-

'''

Filename : Encoder.py

Function : Define the sentence encoders

Input: a bag of sentences of a triple

Output: a feature of sentences with size of batch\_size\*kgDim\*1

a feature of head entity with size of batch\_size\*kgDim\*1

a feature of tail entity with size of batch\_size\*kgDim\*1

'''

import torch

import codecs

import numpy as np

import torch.nn as nn

import torch.nn.functional as F

from code.utils.TensorDevice import LongTensorDevice,FloatTensorDevice,VariableDevice,ParameterDevice

np.random.seed(2019)

torch.manual\_seed(2019)

torch.cuda.manual\_seed(2019)

mySeed = np.random.RandomState(2019)

'''

encoder: lstm

steps1: Every sentences pass through an LSTM encoder

steps2: Use maxpooling to catch each sentence feature

steps3: Use meanpooling to catch the feature of all the sentences

steps4: Average all the entity words in sentences to generate entity feature

'''

class lstmEncoder(nn.Module):

def \_\_init\_\_(self, embeddingDim,hiddenDim,kgDim,biDirection, usegpu =True):

super(lstmEncoder, self).\_\_init\_\_()

self.embeddingDim = embeddingDim

self.hiddenDim = hiddenDim

self.kgDim = kgDim

self.usegpu = usegpu

self.lstm = nn.LSTM(input\_size = embeddingDim, hidden\_size = hiddenDim, bidirectional =biDirection,dropout = 0.5, batch\_first = True)

self.linearW = ParameterDevice(mySeed.uniform(-0.01, 0.01, ( self.hiddenDim \* 2, self.kgDim )),use\_cuda=usegpu, requires\_grad = True)

self.linearb = ParameterDevice(mySeed.uniform(-0.01, 0.01, ( 1, self.kgDim )),use\_cuda=usegpu, requires\_grad = True)

def forward (self, sentences,sentenceMasking,headMasking,tailMasking):

batchSize = sentences.size()[0]

sentenceLen = sentences.size()[2]

sentenceNum = sentences.size()[1]

sentences = sentences.view(-1,sentenceLen,self.embeddingDim) #batchsize\*sentenceNum\*maxlen\*inputDim -> (batchsize\*sentenceNum)\*maxlen\*inputDim

outputSentences,\_ = self.lstm (sentences) #(batchsize\*sentenceNum)\*maxlen\*(hiddenDim\*2)

outputSentences = outputSentences.view(batchSize,sentenceNum,sentenceLen,self.hiddenDim\*2) #batchsize\*sentenceNum\*maxlen\*(hiddenDim\*2)

outputSentences = torch.mul(outputSentences,sentenceMasking.unsqueeze(-1)) #masking the padding words

#print (outputSentences.size())

# cut out the head entity

headTotalLength = torch.sum(headMasking,dim = (1,2)).unsqueeze(1) # batchsize\*1

zeroTensor = torch.eq (headTotalLength,LongTensorDevice(Tensor = np.zeros(headTotalLength.size()), use\_cuda = self.usegpu))

headTotalLength = headTotalLength + zeroTensor

outputHead = torch.sum(torch.mul(outputSentences,headMasking.unsqueeze(-1)),dim = (1,2)) / headTotalLength #batchsize\*(hiddenDim\*2)

headFeature = outputHead.unsqueeze(1) #batchsize\*1\*(hiddenDim\*2)

finalHeadFeature = torch.bmm(headFeature, self.linearW.unsqueeze(0).expand(batchSize,-1,-1)) + self.linearb.unsqueeze(0).expand(batchSize,-1,-1)

# cut out the tail entity

tailTotalLength = torch.sum(tailMasking,dim = (1,2)).unsqueeze(1)

zeroTensor = torch.eq (tailTotalLength,LongTensorDevice(Tensor = np.zeros(tailTotalLength.size()), use\_cuda = self.usegpu))

tailTotalLength = tailTotalLength + zeroTensor

outputTail = torch.sum(torch.mul(outputSentences,tailMasking.unsqueeze(-1)),dim = (1,2)) / tailTotalLength

tailFeature = outputTail.unsqueeze(1)

finalTailFeature = torch.bmm(tailFeature, self.linearW.unsqueeze(0).expand(batchSize,-1,-1)) + self.linearb.unsqueeze(0).expand(batchSize,-1,-1)

featureSentences = F.max\_pool2d (outputSentences.transpose(1,3),kernel\_size = (sentenceLen,1)) #batchSize\*(hiddenDim\*2)\*1\*sentenceNum

# featureSentence = F.avg\_pool1d (featureSentences,kernel\_size = sentenceNum).transpose(1,2) #batchSize\*1\*(hiddenDim\*2)

realSentenceNum = torch.sum(sentenceMasking[:,:,0],dim = 1).unsqueeze(1).unsqueeze(1).expand(batchSize,1,self.hiddenDim\*2)

featureSentence = torch.sum(featureSentences,dim = 3).transpose(1,2)/realSentenceNum #batchSize\*1\*(hiddenDim\*2)

finalFeature = torch.bmm(featureSentence, self.linearW.unsqueeze(0).expand(batchSize,-1,-1)) + self.linearb.unsqueeze(0).expand(batchSize,-1,-1) #batchSize\*1\*(hiddenDim\*2)

return finalFeature.squeeze(1),finalHeadFeature.squeeze(1),finalTailFeature.squeeze(1)

'''

encoder: lstm

steps1: Every sentences pass through an CNN encoder

steps2: Use maxpooling to catch each sentence feature

steps3: Use meanpooling to catch the feature of all the sentences

steps4: Average all the entity words in sentences to generate entity feature

'''

class cnnEncoder(nn.Module):

def \_\_init\_\_(self, embeddingDim,hiddenDim,kgDim,kernelSizes,usegpu =True):

super(cnnEncoder, self).\_\_init\_\_()

self.embeddingDim = embeddingDim

self.hiddenDim = hiddenDim

self.kgDim = kgDim

self.kernelSizes = kernelSizes

self.usegpu = usegpu

self.cnnList = nn.ModuleList([nn.Conv1d(embeddingDim,hiddenDim, kernel\_size = size,padding = int((size-1)/2)) for size in kernelSizes])

self.linearW = ParameterDevice(mySeed.uniform(-0.01, 0.01, ( self.hiddenDim \* len(kernelSizes), self.kgDim )),use\_cuda=usegpu, requires\_grad = True)

self.linearb = ParameterDevice(mySeed.uniform(-0.01, 0.01, ( 1, self.kgDim )),use\_cuda=usegpu, requires\_grad = True)

def forward (self, sentences,sentenceMasking,headMasking,tailMasking):

batchSize = sentences.size()[0]

sentenceLen = sentences.size()[2]

sentenceNum = sentences.size()[1]

sentences = sentences.view(-1,sentenceLen,self.embeddingDim).transpose(1,2) #batchsize\*sentenceNum\*maxlen\*inputDim -> (batchsize\*sentenceNum)\*inputDim\*maxlen

outputSentences = []

for kernel,cnn in zip(self.kernelSizes,self.cnnList): #(batchsize\*sentenceNum)\*maxlen\*(hiddenDim\*2)

if not kernel%2:

outputSentence = cnn(torch.cat([sentences,VariableDevice(Tensor = np.zeros((batchSize\*sentenceNum,self.embeddingDim,1)),requires\_grad = False,use\_cuda = self.usegpu)],2))

else:

outputSentence = cnn(sentences)

outputSentences.append(outputSentence)

outputSentences = torch.cat(outputSentences,1)

outputSentences = outputSentences.view(batchSize,sentenceNum,self.hiddenDim \* len(self.kernelSizes),sentenceLen).transpose(2,3) #batchsize\*sentenceNum\*maxlen\*(hiddenDim\*2)

outputSentences = torch.mul(outputSentences,sentenceMasking.unsqueeze(-1)) #masking the padding words

#print (outputSentences.size())

# cut out the head entity

headTotalLength = torch.sum(headMasking,dim = (1,2)).unsqueeze(1) # batchsize\*1

zeroTensor = torch.eq (headTotalLength,LongTensorDevice(Tensor = np.zeros(headTotalLength.size()), use\_cuda = self.usegpu))

headTotalLength = headTotalLength + zeroTensor

outputHead = torch.sum(torch.mul(outputSentences,headMasking.unsqueeze(-1)),dim = (1,2)) / headTotalLength #batchsize\*(hiddenDim\*2)

headFeature = outputHead.unsqueeze(1) #batchsize\*1\*(hiddenDim\*2)

finalHeadFeature = torch.bmm(headFeature, self.linearW.unsqueeze(0).expand(batchSize,-1,-1)) + self.linearb.unsqueeze(0).expand(batchSize,-1,-1)

# cut out the tail entity

tailTotalLength = torch.sum(tailMasking,dim = (1,2)).unsqueeze(1)

zeroTensor = torch.eq (tailTotalLength,LongTensorDevice(Tensor = np.zeros(tailTotalLength.size()), use\_cuda = self.usegpu))

tailTotalLength = tailTotalLength + zeroTensor

outputTail = torch.sum(torch.mul(outputSentences,tailMasking.unsqueeze(-1)),dim = (1,2)) / tailTotalLength

tailFeature = outputTail.unsqueeze(1)

finalTailFeature = torch.bmm(tailFeature, self.linearW.unsqueeze(0).expand(batchSize,-1,-1)) + self.linearb.unsqueeze(0).expand(batchSize,-1,-1)

featureSentences = F.max\_pool2d (outputSentences.transpose(1,3),kernel\_size = (sentenceLen,1)) #batchSize\*(hiddenDim\*2)\*1\*sentenceNum

# featureSentence = F.avg\_pool1d (featureSentences,kernel\_size = sentenceNum).transpose(1,2) #batchSize\*1\*(hiddenDim\*2)

realSentenceNum = torch.sum(sentenceMasking[:,:,0],dim = 1).unsqueeze(1).unsqueeze(1).expand(batchSize,1,self.hiddenDim\* len(self.kernelSizes))

featureSentence = torch.sum(featureSentences,dim = 3).transpose(1,2)/realSentenceNum #batchSize\*1\*(hiddenDim\*2)

finalFeature = torch.bmm(featureSentence, self.linearW.unsqueeze(0).expand(batchSize,-1,-1)) + self.linearb.unsqueeze(0).expand(batchSize,-1,-1) #batchSize\*1\*(hiddenDim\*2)

return finalFeature.squeeze(1),finalHeadFeature.squeeze(1),finalTailFeature.squeeze(1)

7. DataLoader

# -\*- coding: utf-8 -\*-

'''

Filename : LoadData.py

Function : Preprocess source data and transform to standard fotmat

See GenerateData.py for the steps of preprocess

'''

import re

import os

import json

import codecs

import numpy as np

import pandas as pd

from collections import Counter

from code.process.data import Instance,Triple

import string

''' Load Entity2id & Relation2id file '''

def loadTerm2id (dataPath):

termDict = {}

with open (dataPath,'r') as f:

for line in f:

line = line.strip().split('\t')

termName = line[0]

termNum = line[1]

termDict[termName] = termNum

return termDict

''' clean the word '''

def normWord (word):

word = word.strip().lower()

word = re.sub(u'\s+', '', word, flags=re.U) # match all the blank character

word = word.replace("--", "-")

word = re.sub("\"+", '"', word)

if word.isdigit():

word = '1'

else:

temp = word

for char in word:

if char not in string.printable:

temp = temp.replace(char, '\*')

word = temp

return word

''' Translate sentence into ID in wordDict'''

def sentence2id (sentence,wordDict,headStart,headEnd,tailStart,tailEnd):

sentenceID = []

headID = []

tailID = []

# replace entity with a special name

head = sentence[headStart:headEnd]

tail = sentence[tailStart:tailEnd]

if headStart< tailStart:

sentence = sentence[:headStart] + ' HEADENTITY ' + sentence[headEnd:tailStart] + ' TAILENTITY ' + sentence[tailEnd:]

else:

sentence = sentence[:tailStart] + ' TAILENTITY ' + sentence[tailEnd:headStart] + ' HEADENTITY ' + sentence[headEnd:]

# normalize sentence

for c in string.punctuation:

sentence = sentence.replace(c,' '+c+' ')

head = head.replace(c,' '+c+' ')

tail = tail.replace(c,' '+c+' ')

sentence = sentence.split()

head = head.split()

tail = tail.split()

# normalize entity

for word in head:

word = normWord(word)

if word not in wordDict:

wordDict[word] = len(wordDict) + 1 #the ID zero should be retained for padding words

headID.append(wordDict[word])

for word in tail:

word = normWord(word)

if word not in wordDict:

wordDict[word] = len(wordDict) + 1

tailID.append(wordDict[word])

# calculate the position of head/tail entity in the sentence

# translate sentence with words into sentence with ids

for i,word in enumerate(sentence):

if word == 'HEADENTITY':

newHeadStart = i

newHeadEnd = i + len(headID)

sentenceID.extend(headID)

continue

if word == 'TAILENTITY':

newTailStart = i

newTailEnd = i + len(tailID)

sentenceID.extend(tailID)

continue

word = normWord(word)

if word not in wordDict:

wordDict[word] = len(wordDict) + 1

sentenceID.append(wordDict[word])

return sentenceID,newHeadStart,newHeadEnd,newTailStart,newTailEnd,wordDict

'''

Load train.txt File into a list

train.txt format:

head \t tail \t relation \t sentenceNums

offset1\_start1 \t offset1\_end1 \t offset1\_start2 \t offset1\_end2 \t sentence1

offset2\_start1 \t offset2\_end1 \t offset2\_start2 \t offset2\_end2 \t sentence2

...

head \t tail \t relation \t sentenceNums

offset1\_start1 \t offset1\_end1 \t offset1\_start2 \t offset1\_end2 \t sentence1

offset2\_start1 \t offset2\_end1 \t offset2\_start2 \t offset2\_end2 \t sentence2

offset3\_start1 \t offset3\_end1 \t offset3\_start2 \t offset3\_end2 \t sentence3

...

'''

def loadTrainTriples (dataPath,entityDict,relationDict,wordDict):

triples = [] # store all the triples with texts

with open (dataPath,'r') as f:

for line in f:

line = line.strip().split('\t')

head = int(entityDict[line[0]])

tail = int(entityDict[line[1]])

relation = int(relationDict[line[2]])

sentenceNum = int(line[3])

triple = Triple (

head = head,

tail = tail,

relation = relation,

instance = []

)

for \_ in range (sentenceNum):

sentenceInform = f.readline().strip().split('\t')

offsetStartHead = int(sentenceInform[0])

offsetEndHead = int(sentenceInform[1])

offsetStartTail = int(sentenceInform[2])

offsetEndTail = int(sentenceInform[3])

sentence = sentenceInform[4].strip() #.split(' ')

sentence,headStart,headEnd,tailStart,tailEnd,wordDict = sentence2id(sentence = sentence,

headStart = offsetStartHead,

headEnd = offsetEndHead,

tailStart = offsetStartTail,

tailEnd = offsetEndTail,

wordDict = wordDict)

instance = Instance (

offsetStartHead = headStart,

offsetEndHead = headEnd,

offsetStartTail = tailStart,

offsetEndTail = tailEnd,

sentence = sentence

)

triple.instance.append(instance)

triples.append(triple)

return triples,wordDict

'''

Generate the embedding matrix of word embedding

If the pretrain embeding is not given, the word embedding is randomly initialized from -0.5 to 0.5

'''

def loadPretrainW2V (embedPath,wordDict,embeddingDim):

wordNum = len(wordDict) + 1 #plus one for 0(padding word)

wordEmbedding = np.random.uniform(-0.5,0.5,size = (wordNum,embeddingDim))

wordEmbedding[0] = np.zeros(shape = embeddingDim ,dtype = 'float32')

word2vec = {}

if os.path.exists(embedPath):

with open (embedPath,'r') as f:

for line in f:

line = line.strip().split()

word = line[0].lower() #uncased

embed = np.asarray(line[1:],dtype = 'float32')

word2vec[word] = embed

for word in wordDict:

if word in word2vec:

wordEmbedding[wordDict[word]] = word2vec[word]

return wordEmbedding

'''

Generate the embedding matrix of entity & relation embedding

If the pretrain embeding is not given, the word embedding is randomly initialized from -0.5 to 0.5

'''

def loadPretrainE2V (embedPath,entityDict,embeddingDim):

entityNum = len(entityDict)

entityEmbedding = np.random.uniform(-0.5,0.5,size = (entityNum,embeddingDim))

entityEmbedding[0] = np.zeros(shape = embeddingDim ,dtype = 'float32')

if os.path.exists(embedPath):

with open (embedPath,'r') as f:

for line in f:

line = line.strip().split()

entityID = int (line[0]) #format: entityID \t 0.1 \t 0.2 \t 0.15 ...

embed = np.asarray(line[1:],dtype = 'float32')

entityEmbedding[entityID] = embed

return entityEmbedding

''' Split all data into train & evaluation data'''

def splitData (allTriples,splitRate):

tripleNum = len(allTriples)

np.random.shuffle(allTriples)

splitTrainNum = int(splitRate\*tripleNum)

return allTriples[splitTrainNum:],allTriples[:splitTrainNum]

8. data.py

# -\*- coding: utf-8 -\*-

'''

Filename : Data.py

Function : define the class of Instance & Triple

'''

import re

import os

import json

import codecs

import numpy as np

import pandas as pd

from collections import Counter

class Instance (object):

def \_\_init\_\_(self,offsetStartHead,offsetEndHead,offsetStartTail,offsetEndTail,sentence):

self.offsetStartHead = offsetStartHead

self.offsetEndHead = offsetEndHead

self.offsetStartTail = offsetStartTail

self.offsetEndTail = offsetEndTail

self.sentence = sentence

class Triple (object):

def \_\_init\_\_(self,head,tail,relation,instance = []):

self.head = head

self.tail = tail

self.relation = relation

self.instance = instance #A list, store a bag of sentences

9. util.py

# -\*- coding: utf-8 -\*-

'''

Filename : utils.py

Function : check and print arguments.

'''

import re

import os

import json

import codecs

'''

Used to check the existence of the path. If the path doesn't

exist, raise error if raise\_error is True, or make the path.

'''

def CheckPath(path, raise\_error=True):

if not os.path.exists(path):

if raise\_error:

print("ERROR : Path %s does not exist!" % path)

exit(1)

else:

print("WARNING : Path %s does not exist!" % path)

print("INFO : Creating path %s." % path)

os.makedirs(path)

print("INFO : Successfully making dir!")

return

'''

Used to print arguments on the screen.

'''

def printArgs(args):

print("="\*20 + "Arguments" + "="\*20)

argsDict = vars(args)

for arg, value in argsDict.items():

print("==> {} : {}".format(arg, value))

print("="\*50)

10. evaluation.py

# -\*- coding: utf-8 -\*-

'''

Filename : evaluation.py

Function : When the args.evaluation is true, use this file to evaluate the model in current epoch.

Input: evaluation dataset

Output: evaluation score ==> MR

'''

from tqdm import tqdm

import numpy as np

import torch

from torch.utils.data import dataloader

from torch.autograd import Variable

import copy

from code.utils.TensorDevice import LongTensorDevice,FloatTensorDevice,VariableDevice,ParameterDevice

'''

Evaluation for TransText

'''

def evalTransText(triples, sentences,sentenceMasking,headMasking,tailMasking,simMeasure, model,usegpu):

triples = triples.numpy()

sampleNum = triples.shape[0]

allRelationNum = len(model.relationDict)

rank = 0

triplesX = LongTensorDevice (triples, use\_cuda = usegpu)

sentencesX = LongTensorDevice (sentences,use\_cuda = usegpu)

sentenceMasking = LongTensorDevice (sentenceMasking,use\_cuda = usegpu)

headMasking = LongTensorDevice (headMasking,use\_cuda = usegpu)

tailMasking = LongTensorDevice (tailMasking,use\_cuda = usegpu)

realScores = model.evalForward (triplesX,sentencesX, sentenceMasking,headMasking,tailMasking).detach().cpu().numpy() # N\*1

for entity in range(allRelationNum):

candidateEntity = np.array([entity]\*sampleNum)

candidateTriples = copy.deepcopy(triples) # N\*3

candidateTriples[:,2] = candidateEntity

triplesX = LongTensorDevice (candidateTriples, use\_cuda = usegpu)

candidateScore = model.evalForward (triplesX,sentencesX,sentenceMasking,headMasking,tailMasking)

candidateScore = candidateScore.detach().cpu().numpy()

judgeMatrix = candidateScore-realScores

if simMeasure == "L2" or simMeasure == "L1":

judgeMatrix[judgeMatrix > 0] = 0

judgeMatrix[judgeMatrix < 0] = 1

elif simMeasure == "dot" or simMeasure == "cos":

judgeMatrix[judgeMatrix > 0] = 1

judgeMatrix[judgeMatrix < 0] = 0

else:

print("ERROR : Similarity measure is not supported!")

exit(1)

rank += np.sum(judgeMatrix)

return rank, sampleNum

'''

Now, only MR metric is available

'''

def MREvaluation(evalloader:dataloader, modelName, model, simMeasure="L2", usegpu = True):

R = 0

N = 0

for triple,sentences,sentenceMasking,headMasking,tailMasking in evalloader:

if modelName == "TransText":

r,n = evalTransText(triple,sentences,sentenceMasking,headMasking,tailMasking, simMeasure, model,usegpu)

else:

print("ERROR : The %s evaluation is not supported!" % model)

exit(1)

R += r #sum of rank

N += n #sum of the instance

return (R / N) #MeanRank

11. TensorDevice.py

# -\*- coding: utf-8 -\*-

'''

Filename : TensorDevice.py

Function : Package Parameter，Variable，Tensor for easy use of cuda

'''

import os

import numpy as np

import torch

import torch.nn as nn

from torch.autograd import Variable

np.random.seed(2019)

torch.manual\_seed(2019)

torch.cuda.manual\_seed(2019)

mySeed = np.random.RandomState(2019)

def LongTensorDevice(Tensor,use\_cuda):

if use\_cuda:

return torch.LongTensor(Tensor).cuda()

else:

return torch.LongTensor(Tensor)

def FloatTensorDevice(Tensor,use\_cuda):

if use\_cuda:

return torch.FloatTensor(Tensor).cuda()

else:

return torch.FloatTensor(Tensor)

def VariableDevice (Tensor,requires\_grad,use\_cuda):

if use\_cuda:

return Variable(torch.FloatTensor(Tensor).cuda(),requires\_grad =requires\_grad)

else:

return Variable(torch.FloatTensor(Tensor),requires\_grad =requires\_grad)

def ParameterDevice (Tensor,requires\_grad,use\_cuda):

if use\_cuda:

return nn.Parameter(torch.FloatTensor(Tensor).cuda(),requires\_grad =requires\_grad)

else:

return nn.Parameter(torch.FloatTensor(Tensor),requires\_grad =requires\_grad)