Code for Task 1

May 2, 2022

```
[ ]: from google.colab import drive
     drive.mount('/content/drive')
```

```
[ ]: import torch
     import pandas as pd
     import numpy as np
     from PIL import Image
     import cv2
     import os
     from torchvision import transforms,models
     import pickle
     import random
     torch.manual_seed(42)
     path='/content/drive/MyDrive/image_captioning_dataset/'
     from torchtext.data.metrics import bleu_score
```

```
[ ]: import torch
     import pandas as pd
     import numpy as np
     from PIL import Image
     import cv2
     import os
     from torchvision import transforms,models
     import pickle
     import random
     torch.manual_seed(42)
     path='/content/drive/MyDrive/image_captioning_dataset/'
     from torchtext.data.metrics import bleu_score
```

```
[ ]: def initialize_glove():
       num=0
       word2idx = {}
       vectors = []

       lines=open('/content/drive/MyDrive/glove.6B.300d.txt', 'rb').readlines()
       print(len(lines))
       for line in lines:
           line = line.decode().split()
```

```
      word = line[0]
      word2idx[word] = num
      num = num + 1
      vect = np.append(np.array(line[1:]),0.0).astype(np.float)
      vectors.append(vect)

  lines=open('/content/drive/MyDrive/captions.txt', 'rb').readlines()
  vocab={}
  vocab["<pad>"]=0
  vocab["<start>"]=1
  vocab["<end>"]=2
  num=3
  for line in lines:
    cap=line.decode().split("\t")[1]
    for word in cap.split():
      word=word.lower()
      if word not in vocab:
        vocab[word]=num
        num=num+1


  word_embeddings=np.zeros((len(vocab),301))
  words={}
  for word in vocab:
    if word in word2idx:
      word_embeddings[vocab[word]]=vectors[word2idx[word]]
    else:
      if word=="<pad>":
        word_embeddings[vocab[word]]=np.zeros(301)
      else:
        word_embeddings[vocab[word]]=np.random.normal(0.5, size=301)
        if(word=="<start>"):
          word_embeddings[vocab[word],300]=0.5
        if(word=="<end>"):
          word_embeddings[vocab[word],300]=1.0
    words[vocab[word]]=word
  print(word_embeddings.shape)
  return vocab, word_embeddings ,words

vocab, word_embeddings ,words= initialize_glove()
```

[ ]:

```
class DatasetLoader(torch.utils.data.Dataset):
  def __init__(self,vocab, word_embeddings,caption_length ,path , mode):
    self.path=path
    self.mode=mode
    self.vocab=vocab
    self.caption_length=caption_length
    self.word_embeddings=word_embeddings
    np.random.seed(42)
    indices=np.arange(4000)
    np.random.shuffle(indices)
    indices_temp=indices[:int(0.8*4000)]
    train_indices=np.append(indices_temp*5,[indices_temp*5+1,indices_temp*5+2,indices_temp*5+3,indices_temp*5+4])
    test_indices=indices[int(0.8*4000):]



    if(mode=='train'):
      self.indices=train_indices
      self.caption_transform=transforms.Compose([transforms.ToTensor()])
      self.image_transform=transforms.Compose([
                                              transforms.ToPILImage(),
                                              transforms.Resize((256,256)),
                                              transforms.ToTensor(),
                                              transforms.Normalize([0.5276365, 0.508226 , 0.4184626], [0.27150184, 0.26589277, 0.28562558])
                                            ])

    if(mode=='test'):
      self.indices=test_indices
      self.caption_transform=transforms.Compose([transforms.ToTensor()])
      self.image_transform=transforms.Compose([
                                              transforms.ToPILImage(),
                                              transforms.Resize((256,256)),
                                              transforms.ToTensor(),
                                              transforms.Normalize([0.5276365, 0.508226 , 0.4184626], [0.27150184, 0.26589277, 0.28562558])
                                            ])

  def __len__(self):
    return len(self.indices)

  def __getitem__(self,idx):
    if(self.mode =='train') :
      path=self.path+'image'+str(self.indices[idx]//5)+'.jpg'
      image=self.image_transform(np.array(cv2.imread(path)))
      path=self.path+'captions'+str(self.indices[idx]//5)+'.txt'
      caption=open(path,'r').readlines()[self.indices[idx]%5].lower().split()
      caption.insert(0,'<start>')
      caption.append('<end>')
      while(len(caption)<self.caption_length):
        caption.append('<pad>')
      if(len(caption)>self.caption_length):
        caption=caption[:self.caption_length]
        caption[self.caption_length-1]='<end>'
      caption_embbedings=np.zeros((len(caption)-1,301))
      target_embbedings=np.zeros((len(caption)-1,1))
      for i in range(len(caption)-1):
        caption_embbedings[i]=self.word_embeddings[vocab[caption[i]]]
        target_embbedings[i][0]=float(vocab[caption[i+1]])
    #target_embbedings[len(caption)][0]=float(vocab['<pad>'])
      return image.float() , self.caption_transform(caption_embbedings).float(),␣
↪self.caption_transform(target_embbedings).float()
    if(self.mode=='test'):
      path=self.path+'image'+str(self.indices[idx])+'.jpg'
      image=self.image_transform(np.array(cv2.imread(path)))
      path=self.path+'captions'+str(self.indices[idx])+'.txt'
      caption=open(path,'r').readlines()
      for i in range(len(caption)) :
        caption[i]=caption[i].lower().split()
      return image.float(),caption
```

[ ]:

```
class NetVlad(torch.nn.Module):
  def __init__(self, num_clusters, desciptor_dimension, beta):
    super(NetVlad, self).__init__()
    self.num_clusters=num_clusters
    self.descriptor_dimension=desciptor_dimension
    self.beta=beta
    self.cluster_centres=torch.nn.Parameter(torch.rand(num_clusters,␣
↪desciptor_dimension))
    self.netvlad=torch.nn.Conv2d(desciptor_dimension, num_clusters,␣
↪kernel_size=(1,1), bias=True)
  def _init_params(self):
    self.netvlad.weight = torch.nn.Parameter((2.0 * self.beta * self.
↪cluster_centres).unsqueeze(-1).unsqueeze(-1))
    self.netvlad.bias = torch.nn.Parameter(- self.beta * self.cluster_centres.
↪norm(dim=1))
  def forward(self,x):
    shape=x.shape
    a_k=torch.nn.functional.softmax(self.netvlad(x).view(shape[0],self.
↪num_clusters,-1),dim=1)
    x_flatten = x.view(shape[0], shape[1], -1)
    difference = x_flatten.expand(self.num_clusters, -1, -1, -1).permute(1, 0,␣
↪2, 3) - self.cluster_centres.expand(x_flatten.size(-1), -1, -1).permute(1, 2,␣
↪0).unsqueeze(0)
    difference =difference * a_k.unsqueeze(2)
    output = difference.sum(dim=-1)
    output = torch.nn.functional.normalize(output, p=2, dim=2)
    output = output.view(x.size(0), -1)
    output = torch.nn.functional.normalize(output, p=2, dim=1)
    return output
    #part of NETVLAD code borrwed from https://github.com/lyakaap/
 ↪NetVLAD-pytorch/blob/master/netvlad.py


class CNN_Resnet(torch.nn.Module):
  def __init__(self):
      super(CNN_Resnet, self).__init__()
      cnn = models.resnet50(pretrained=True)
      for param in cnn.parameters():
            param.requires_grad_(False)
      modules = list(cnn.children())[:-2]
      self.resnet = torch.nn.Sequential(*modules)
  def forward(self,x):
    return self.resnet(x)


class RNN(torch.nn.Module):
  def __init__(self ,hidden_dim, num_layers,input_dim):
    super(RNN, self).__init__()
    self.hidden_dim=hidden_dim
    self.num_layers=num_layers
    self.input_dim=input_dim
    self.rnn = torch.nn.RNN(input_dim, hidden_dim, num_layers, batch_first=True)
    self.fc=torch.nn.Linear(hidden_dim,input_dim)
  def forward(self,image,caption,device):
    c0 = image.requires_grad_().to(device)
    h0 = image.requires_grad_().to(device)
    out, hn=self.rnn(caption,h0)
    return out

class Image_Captioning(torch.nn.Module):
  def __init__(self,vocab,word_embbedings,words,device):
    super(Image_Captioning, self).__init__()
    self.device=device
    self.resnet=CNN_Resnet()
    self.netvlad=NetVlad(16,2048,0.5)
    self.vocab=vocab
    self.words=words
    self.word_embbedings=word_embbedings
    self.linear1 = torch.nn.Linear(2048*16, 4096)
    self.rnn=RNN(4096,1,301)
    self.linear2= torch.nn.Linear(4096,len(vocab))
    self.soft=torch.nn.Softmax(dim=2)
  def forward(self,image,caption,device):
    cnn=self.resnet(image)
    vlad=self.netvlad(cnn)
    hidden=self.linear1(vlad)
    out=self.rnn(hidden.unsqueeze(0),caption.squeeze(1),device)
    out=self.linear2(out)
    return out

  def predict(self,image):
    prediction=[]
    caption=[]
    caption.append('<start>')
    while(len(caption)<19):
      caption.append('<pad>')
    count=0
    while (True):
      caption_embbedings=np.zeros((len(caption),301))
      for i in range(len(caption)):
        caption_embbedings[i]=word_embeddings[vocab[caption[i]]]

      caption_embbedings=transforms.Compose([transforms.
 ↪ToTensor()])(caption_embbedings)
      caption_embbedings=caption_embbedings.float().to(self.device)
      out = self.forward(image.to(self.device),caption_embbedings,self.device)
  #print(out)
      idxs = torch.argmax(out, dim = 2).cpu().numpy()
  # print(idxs)
      new_word=words[idxs[0,count]]
      if(new_word!='<start>' and new_word!='<end>' and new_word!='<pad>'):
        prediction.append(new_word)
      count=count+1
  #print(count)
      if(new_word=='<end>' or count==19):
        break
      caption[count]=new_word

    return prediction
```

[ ]:

```
def train_one_epoch(model,dataset,criterion,optimizer,epoch,device):
  model.train()
  model.to(device)
  train_loss = []
  acc = []

  for i,(image,input, target) in enumerate(dataset):
        image,input, target = image.to(device), input.to(device), target.
 ↪to(device)
        output = model(image,input,device)
        target = target.type(torch.LongTensor)
        target=target.to(device)
        optimizer.zero_grad()
        #print(target.shape)
        loss = criterion(output.permute(0,2,1), target.squeeze(3).squeeze(1))
        loss.backward()
        optimizer.step()
        #idxs = torch.argmax(output, dim = 1)
        #acc.append(accuracy(idxs, target))
        train_loss.append(loss.item())

    #wandb.log({
     #   'epoch': epoch,
      #  'train_loss':np.mean(train_loss),
       # "train_acc": np.mean(acc),
    #})
        if (i%20==0) :
           print(f'step - {i} Train loss - {np.mean(train_loss)}')
  print(f'Epoch - {epoch}\tTrain loss - {np.mean(train_loss)}')

def save_checkpoint(state,  path):
    f = open(path, 'w')

    torch.save(state,path)
    f.close()
```

[ ]:

```
torch.manual_seed(998244353)
trainloader=DatasetLoader(vocab, word_embeddings,20 ,path,'train')
train_dataset=torch.utils.data.DataLoader(trainloader,batch_size=64,␣
 ↪shuffle=True,num_workers=2)
device='cuda'
model=Image_Captioning(vocab,word_embeddings,words,device).float()
learning_rate=0.001
num_epochs=10
criterion = torch.nn.CrossEntropyLoss().to(device)
params_to_update = []
for name,param in model.named_parameters():
  if param.requires_grad == True:
    print(name)
    params_to_update.append(param)

optimizer = torch.optim.Adam(params_to_update, lr=learning_rate)
#model.load_state_dict(torch.load('/content/drive/MyDrive/model.
 ↪pt',map_location=torch.device('cpu')))
```

```
for epoch in range(num_epochs):
  train_one_epoch(model,train_dataset,criterion,optimizer,epoch,device)
  save_checkpoint(model.state_dict(), '/content/drive/MyDrive/model_hyp_rnn.pt')
```

[ ]:

```
testloader=DatasetLoader(vocab, word_embeddings,20, path,'test')
test_dataset=torch.utils.data.DataLoader(testloader,batch_size=1,␣
 ↪shuffle=True,num_workers=2)
true_captions=[]
predicted_captions=[]
length=len((test_dataset))
for i, (image,true_caption) in enumerate(test_dataset):
  if (i%80==0):
    print(f'{i/length*100}% captions predicted')
  true_captions.append([[i[0] for i in caption] for caption in true_caption ])
  predicted_captions.append(model.predict(image))
print('prediction complete')
```

[ ]:

```
from torchtext.data.metrics import bleu_score
print(f'The bleu1 score is␣
 ↪{bleu_score(predicted_captions,true_captions,max_n=1,weights=[1])}')
print(f'The bleu2 score is␣
 ↪{bleu_score(predicted_captions,true_captions,max_n=2,weights=[0,1])}')
print(f'The bleu3 score is␣
 ↪{bleu_score(predicted_captions,true_captions,max_n=3,weights=[0,0,1])}')
print(f'The bleu4 score is␣
 ↪{bleu_score(predicted_captions,true_captions,max_n=4,weights=[0,0,0,1])}')
```

Code for Task 2

May 2, 2022

[ ]:

```python
from google.colab import drive
drive.mount('/content/drive')
```

[ ]:

```python
import torch
import pandas as pd
import numpy as np
from PIL import Image
import cv2
import os
from torchvision import transforms,models
import pickle
import random
torch.manual_seed(42)
path='/content/drive/MyDrive/image_captioning_dataset/'
from torchtext.data.metrics import bleu_score
```

[ ]:

```python
import torch
import pandas as pd
import numpy as np
from PIL import Image
import cv2
import os
from torchvision import transforms,models
import pickle
import random
torch.manual_seed(42)
path='/content/drive/MyDrive/image_captioning_dataset/'
from torchtext.data.metrics import bleu_score
```

[ ]:

```python
def initialize_glove():
  num=0
  word2idx = {}
  vectors = []

  lines=open('/content/drive/MyDrive/glove.6B.300d.txt', 'rb').readlines()
  print(len(lines))
  for line in lines:
      line = line.decode().split()
      word = line[0]
      word2idx[word] = num
      num = num + 1
      vect = np.append(np.array(line[1:]),0.0).astype(np.float)
      vectors.append(vect)

  lines=open('/content/drive/MyDrive/captions.txt', 'rb').readlines()
  vocab={}
  vocab["<pad>"]=0
  vocab["<start>"]=1
  vocab["<end>"]=2
  num=3
  for line in lines:
    cap=line.decode().split("\t")[1]
    for word in cap.split():
      word=word.lower()
      if word not in vocab:
        vocab[word]=num
        num=num+1


  word_embeddings=np.zeros((len(vocab),301))
  words={}
  for word in vocab:
    if word in word2idx:
      word_embeddings[vocab[word]]=vectors[word2idx[word]]
    else:
      if word=="<pad>":
        word_embeddings[vocab[word]]=np.zeros(301)
      else:
        word_embeddings[vocab[word]]=np.random.normal(0.5, size=301)
        if(word=="<start>"):
          word_embeddings[vocab[word],300]=0.5
        if(word=="<end>"):
          word_embeddings[vocab[word],300]=1.0
    words[vocab[word]]=word
  print(word_embeddings.shape)
  return vocab, word_embeddings ,words

vocab, word_embeddings ,words= initialize_glove()
```

[ ]:

```
class DatasetLoader(torch.utils.data.Dataset):
  def __init__(self,vocab, word_embeddings,caption_length ,path , mode):
    self.path=path
    self.mode=mode
    self.vocab=vocab
    self.caption_length=caption_length
```

```
   self.word_embeddings=word_embeddings
   np.random.seed(42)
   indices=np.arange(4000)
   np.random.shuffle(indices)
   indices_temp=indices[:int(0.8*4000)]
   train_indices=np.
↪append(indices_temp*5,[indices_temp*5+1,indices_temp*5+2,indices_temp*5+3,indices_temp*5+4])
   test_indices=indices[int(0.8*4000):]



   if(mode=='train'):
     self.indices=train_indices
     self.caption_transform=transforms.Compose([transforms.ToTensor()])
     self.image_transform=transforms.Compose([
                                               transforms.ToPILImage(),
                                               transforms.Resize((256,256)),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.5276365,␣
↪0.508226 , 0.4184626], [0.27150184, 0.26589277, 0.28562558])
                                             ])

   if(mode=='test'):
     self.indices=test_indices
     self.caption_transform=transforms.Compose([transforms.ToTensor()])
     self.image_transform=transforms.Compose([
                                               transforms.ToPILImage(),
                                               transforms.Resize((256,256)),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.5276365,␣
↪0.508226 , 0.4184626], [0.27150184, 0.26589277, 0.28562558])
                                             ])

 def __len__(self):
   return len(self.indices)

 def __getitem__(self,idx):
   if(self.mode =='train') :
     path=self.path+'image'+str(self.indices[idx]//5)+'.jpg'
     image=self.image_transform(np.array(cv2.imread(path)))
     path=self.path+'captions'+str(self.indices[idx]//5)+'.txt'
     caption=open(path,'r').readlines()[self.indices[idx]%5].lower().split()
     caption.insert(0,'<start>')
     caption.append('<end>')
     while(len(caption)<self.caption_length):
       caption.append('<pad>')
     if(len(caption)>self.caption_length):
       caption=caption[:self.caption_length]
       caption[self.caption_length-1]='<end>'
     caption_embbedings=np.zeros((len(caption)-1,301))
     target_embbedings=np.zeros((len(caption)-1,1))
     for i in range(len(caption)-1):
       caption_embbedings[i]=self.word_embeddings[vocab[caption[i]]]
       target_embbedings[i][0]=float(vocab[caption[i+1]])
   #target_embbedings[len(caption)][0]=float(vocab['<pad>'])
     return image.float() , self.caption_transform(caption_embbedings).float(),␣
↪self.caption_transform(target_embbedings).float()
   if(self.mode=='test'):
     path=self.path+'image'+str(self.indices[idx])+'.jpg'
     image=self.image_transform(np.array(cv2.imread(path)))
     path=self.path+'captions'+str(self.indices[idx])+'.txt'
     caption=open(path,'r').readlines()
     for i in range(len(caption)) :
       caption[i]=caption[i].lower().split()
     return image.float(),caption
```

[ ]:

```
class NetVlad(torch.nn.Module):
  def __init__(self, num_clusters, desciptor_dimension, beta):
    super(NetVlad, self).__init__()
    self.num_clusters=num_clusters
    self.descriptor_dimension=desciptor_dimension
    self.beta=beta
    self.cluster_centres=torch.nn.Parameter(torch.rand(num_clusters,␣
 ↪desciptor_dimension))
    self.netvlad=torch.nn.Conv2d(desciptor_dimension, num_clusters,␣
 ↪kernel_size=(1,1), bias=True)
  def _init_params(self):
    self.netvlad.weight = torch.nn.Parameter((2.0 * self.beta * self.
 ↪cluster_centres).unsqueeze(-1).unsqueeze(-1))
    self.netvlad.bias = torch.nn.Parameter(- self.beta * self.cluster_centres.
 ↪norm(dim=1))
  def forward(self,x):
    shape=x.shape
    a_k=torch.nn.functional.softmax(self.netvlad(x).view(shape[0],self.
 ↪num_clusters,-1),dim=1)
    x_flatten = x.view(shape[0], shape[1], -1)
    difference = x_flatten.expand(self.num_clusters, -1, -1, -1).permute(1, 0,␣
 ↪2, 3) - self.cluster_centres.expand(x_flatten.size(-1), -1, -1).permute(1, 2,␣
 ↪0).unsqueeze(0)
    difference =difference * a_k.unsqueeze(2)
    output = difference.sum(dim=-1)
    output = torch.nn.functional.normalize(output, p=2, dim=2)
    output = output.view(x.size(0), -1)
    output = torch.nn.functional.normalize(output, p=2, dim=1)
    return output
    #part of NETVLAD code borrwed from https://github.com/lyakaap/
 ↪NetVLAD-pytorch/blob/master/netvlad.py


class CNN_Resnet(torch.nn.Module):
  def __init__(self):
      super(CNN_Resnet, self).__init__()
      cnn = models.resnet50(pretrained=True)
      for param in cnn.parameters():
            param.requires_grad_(False)
      modules = list(cnn.children())[:-2]
      self.resnet = torch.nn.Sequential(*modules)
  def forward(self,x):
    return self.resnet(x)


class RNN_LSTM(torch.nn.Module):
  def __init__(self ,hidden_dim, num_layers,input_dim):
    super(RNN_LSTM, self).__init__()
    self.hidden_dim=hidden_dim
    self.num_layers=num_layers
    self.input_dim=input_dim
    self.lstm = torch.nn.LSTM(input_dim, hidden_dim, num_layers,␣
 ↪batch_first=True)
    self.fc=torch.nn.Linear(hidden_dim,input_dim)
  def forward(self,image,caption,device):
    c0 = image.requires_grad_().to(device)
    h0 = image.requires_grad_().to(device)
    out, (hn,cn)=self.lstm(caption,(h0,c0))
    return out

class Image_Captioning(torch.nn.Module):
  def __init__(self,vocab,word_embbedings,words,device):
    super(Image_Captioning, self).__init__()
    self.device=device
    self.resnet=CNN_Resnet()
    self.netvlad=NetVlad(16,2048,0.5)
    self.vocab=vocab
    self.words=words
    self.word_embbedings=word_embbedings
    self.linear1 = torch.nn.Linear(2048*16, 4096)
    self.lstm=RNN_LSTM(4096,1,301)
    self.linear2= torch.nn.Linear(4096,len(vocab))
    self.soft=torch.nn.Softmax(dim=2)
  def forward(self,image,caption,device):
    cnn=self.resnet(image)
    vlad=self.netvlad(cnn)
    hidden=self.linear1(vlad)
    out=self.lstm(hidden.unsqueeze(0),caption.squeeze(1),device)
    out=self.linear2(out)
    return out

  def predict(self,image):
    prediction=[]
    caption=[]
    caption.append('<start>')
    while(len(caption)<19):
      caption.append('<pad>')
    count=0
    while (True):
      caption_embbedings=np.zeros((len(caption),301))
      for i in range(len(caption)):
        caption_embbedings[i]=word_embeddings[vocab[caption[i]]]

      caption_embbedings=transforms.Compose([transforms.ToTensor()])(caption_embbedings)
      caption_embbedings=caption_embbedings.float().to(self.device)
      out = self.forward(image.to(self.device),caption_embbedings,self.device)
  #print(out)
      idxs = torch.argmax(out, dim = 2).cpu().numpy()
  # print(idxs)
      new_word=words[idxs[0,count]]
      if(new_word!='<start>' and new_word!='<end>' and new_word!='<pad>'):
        prediction.append(new_word)
      count=count+1
  #print(count)
      if(new_word=='<end>' or count==19):
        break
      caption[count]=new_word

    return prediction
```

[ ]:

```
def train_one_epoch(model,dataset,criterion,optimizer,epoch,device):
  model.train()
  model.to(device)
  train_loss = []
  acc = []

  for i,(image,input, target) in enumerate(dataset):
        image,input, target = image.to(device), input.to(device), target.
 ↪to(device)
        output = model(image,input,device)
        target = target.type(torch.LongTensor)
        target=target.to(device)
        optimizer.zero_grad()
        #print(target.shape)
        loss = criterion(output.permute(0,2,1), target.squeeze(3).squeeze(1))
        loss.backward()
        optimizer.step()
        #idxs = torch.argmax(output, dim = 1)
        #acc.append(accuracy(idxs, target))
        train_loss.append(loss.item())

    #wandb.log({
     #   'epoch': epoch,
      #  'train_loss':np.mean(train_loss),
       # "train_acc": np.mean(acc),
    #})
        if (i%20==0) :
           print(f'step - {i} Train loss - {np.mean(train_loss)}')
  print(f'Epoch - {epoch}\tTrain loss - {np.mean(train_loss)}')

def save_checkpoint(state,  path):
    f = open(path, 'w')

    torch.save(state,path)
    f.close()
```

[ ]:

```
trainloader=DatasetLoader(vocab, word_embeddings,20 ,path,'train')
train_dataset=torch.utils.data.DataLoader(trainloader,batch_size=64,␣
 ↪shuffle=True,num_workers=2)
device='cuda:0'
model=Image_Captioning(vocab,word_embeddings,words,device).float()
learning_rate=0.001
num_epochs=20
criterion = torch.nn.CrossEntropyLoss().to(device)
params_to_update = []
for name,param in model.named_parameters():
  if param.requires_grad == True:
    print(name)
    params_to_update.append(param)

optimizer = torch.optim.Adam(params_to_update, lr=learning_rate)
#model.load_state_dict(torch.load('/content/drive/MyDrive/model.
 ↪pt',map_location=torch.device('cpu')))
for epoch in range(num_epochs):
  train_one_epoch(model,train_dataset,criterion,optimizer,epoch,device)
  save_checkpoint(model.state_dict(), '/content/drive/MyDrive/model_final.pt')
```

[ ]:

```
testloader=DatasetLoader(vocab, word_embeddings,20, path,'test')
test_dataset=torch.utils.data.DataLoader(testloader,batch_size=1,␣
 ↪shuffle=True,num_workers=2)
true_captions=[]
predicted_captions=[]
length=len((test_dataset))
for i, (image,true_caption) in enumerate(test_dataset):
  if (i%80==0):
    print(f'{i/length*100}% captions predicted')
  true_captions.append([[i[0] for i in caption] for caption in true_caption ])
  predicted_captions.append(model.predict(image))
print('prediction complete')
```

[ ]:

```
from torchtext.data.metrics import bleu_score
print(f'The bleu1 score is␣
 ↪{bleu_score(predicted_captions,true_captions,max_n=1,weights=[1])}')
print(f'The bleu2 score is␣
 ↪{bleu_score(predicted_captions,true_captions,max_n=2,weights=[0,1])}')
print(f'The bleu3 score is␣
 ↪{bleu_score(predicted_captions,true_captions,max_n=3,weights=[0,0,1])}')
print(f'The bleu4 score is␣
 ↪{bleu_score(predicted_captions,true_captions,max_n=4,weights=[0,0,0,1])}')
```

Code for task 3

May 2, 2022

[ ]:

```
# -*- coding: utf-8 -*-


from torch.autograd import Variable
import torch
import random
import wandb
# wandb.init(project = 'assign3', name = 'mt_final')
import numpy as np
import pickle
import torchtext
from collections import Counter
from torchtext.vocab import Vocab
import pickle
import io
import math
import time
import torch.nn as nn
from tqdm import tqdm
from nltk import word_tokenize
from transformers import AutoModel, AutoTokenizer, BertTokenizerFast
from torch.nn.utils.rnn import pad_sequence
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchtext import vocab
from torchtext.data.utils import get_tokenizer
import numpy as np
import spacy
spacy_eng = spacy.load("en")


PAD_IDX_EN = 0
BOS_IDX_EN = 1
EOS_IDX_EN = 2
UNK_IDX_EN = 3
GLOVE_TEXT_PATH = 'glove.6B.300d.txt'
```

```
def add_specials(vocab):
    vocab["<unk>"] = UNK_IDX_EN
    vocab["<pad>"] = PAD_IDX_EN
    vocab["<bos>"] = BOS_IDX_EN
    vocab['<eos>'] = EOS_IDX_EN
    return vocab

def save_pickle(data, path):
  with open(path, 'wb') as f:
    pickle.dump(data, f)
def load_pickle(path):
  with open(path, 'rb') as f:
    return pickle.load(f)

def load_embeds_enc(root_dir):
    embeddings_index = dict()
    f = open(root_dir)
    c =4
    for line in f:
        values = line.split()
        word = values[0]
        embeddings_index[word] = c
        c +=1
    f.close()
    return embeddings_index

# Part of the model code borrowed from https://pytorch.org/tutorials/beginner/
 ↪torchtext_translation_tutorial.html
class Encoder(nn.Module):
    def __init__(self,weights_matrix, vocab_size= 400000, emb_dim=300,␣
 ↪hidden_size=300):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(vocab_size, emb_dim)
        self.embedding.weight.requires_grad = True
        self.embedding.load_state_dict({'weight': weights_matrix})
        self.lstm = nn.LSTM(emb_dim, hidden_size, 1)

    def forward(self, input, hidden):
        embed = self.embedding(input)
        output, (hidden, cell) = self.lstm(embed, hidden)
        return hidden, cell

    def first_hidden(self, batch_size, device):
        return (Variable(torch.cuda.FloatTensor(1, batch_size, self.hidden_size).
 ↪zero_()).to('cuda:0'),
                Variable(torch.cuda.FloatTensor(1, batch_size, self.hidden_size).
 ↪zero_()).to('cuda:0'))


class Decoder(nn.Module):
    def __init__(self, weights_matrix, vocab_size= 400000, emb_dim=128,␣
 ↪hidden_size=300):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_dim)
        self.embedding.weight.requires_grad = False
        self.embedding.load_state_dict({'weight': weights_matrix})
        self.lstm = nn.LSTM(emb_dim, hidden_size, 1)
        self.output_dim = vocab_size
        self.linear = nn.Linear(hidden_size, vocab_size)
        #self.softmax = nn.Softmax(dim=1)

    def forward(self, input, hidden, cell):
        #print("L38", input.shape)
        embed = self.embedding(input)
        embed = embed.unsqueeze(0)
        #print("L40",embed.shape, hidden[0].shape, hidden[1].shape)
        output, (hidden, cell) = self.lstm(embed, (hidden, cell))
        #print(output.shape)
        output = output.squeeze(0)
        #print(output.shape)
        linear = self.linear(output)
        #softmax = self.softmax(linear)
        return linear, hidden, cell

class Seq2Seq(nn.Module):
    def __init__(self, input_size, output_size, in_emb_dim, dec_emb_dim,␣
 ↪weight_matrix_enc, weight_matrix_dec, device):
        super(Seq2Seq, self).__init__()

        self.encoder = Encoder(vocab_size = input_size, emb_dim = in_emb_dim,␣
 ↪weights_matrix = weight_matrix_enc)
        self.decoder = Decoder(vocab_size = output_size, emb_dim = dec_emb_dim,␣
 ↪weights_matrix = weight_matrix_dec)
        self.device = device

    def forward(self,
                src,
                trg,
                teacher_forcing_ratio):
        # teacher_forcing_ratio = 0 --> Eval
        # teacher_forcing_ratio = 1 --> TRAIN
        batch_size = src.shape[1]
        max_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim
        src = src.to(self.device)
        outputs = torch.zeros(max_len, batch_size, trg_vocab_size).to(self.
 ↪device)
        first_hidden = self.encoder.first_hidden(batch_size, self.device)
        hidden, cell = self.encoder(src, first_hidden)

        input = trg[0,:]

        for t in range(1, max_len):
            output, hidden, cell = self.decoder(input, hidden, cell)
            outputs[t] = output
            teacher_force = random.random() < teacher_forcing_ratio
            top1 = output.argmax(1)
            input = (trg[t] if teacher_force else top1)

        return outputs


en_tokenizer = get_tokenizer('spacy', language='en')
bert_model = AutoModel.from_pretrained('ai4bharat/indic-bert')
bert_tokenizer = AutoTokenizer.from_pretrained('ai4bharat/indic-bert')

en_bert_model = AutoModel.from_pretrained('bert-base-uncased')
en_bert_tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')


train_filepaths = ['/scratch/tanay/exp/en-gu/train.en', '/scratch/tanay/exp/
 ↪en-gu/train.gu']
val_filepaths = ['/scratch/tanay/exp/en-gu/dev.en', '/scratch/tanay/exp/en-gu/
 ↪dev.gu']
test_filepaths = ['/scratch/tanay/exp/en-gu/test.en', '/scratch/tanay/exp/en-gu/
 ↪test.gu']

def entokenizer(text_list, tokenizer):
  if isinstance(text_list, str):
    text_list = [text_list]
  tokenized_text = []
  for text in text_list:
    ls= []
    for tok in tokenizer(text.strip()):
      ls.append(tok.lower())
    tokenized_text.append(ls)
  return tokenized_text

def gu_tokenizer(text, tokenizer):
    return tokenizer(text, add_special_tokens=False)['input_ids']


def build_vocab(filepath, lang, _tokeniz):
  vocab = {}
  c = 4
  with io.open(filepath, encoding="utf8") as f:
    data = f.readlines()
    for i in tqdm(range(0, len(data), 512), desc = f'Building vocab {lang}'):
        if lang == 'en':
          for k in entokenizer(data[i:i + 512], en_tokenizer):
            # print(k)
            # exit()
            for token in k:
              if token not in vocab:
                vocab[token] = c
                c +=1
        elif lang == 'gu':
          # print(counter)
          for k in gu_tokenizer(data[i: i + 512], _tokeniz):
            for token in k:
              if token not in vocab:
                vocab[token] = c
                c +=1
  return vocab

if False:
    #gu_vocab = build_vocab(train_filepaths[1], lang = 'gu')
    en_vocab = build_vocab(train_filepaths[0], lang = 'en', _tokeniz =␣
 ↪en_bert_tokenizer)

    #specials=['<unk>', '<pad>', '<bos>', '<eos>'])
    en_vocab = add_specials(en_vocab)
    #gu_vocab = add_specials(gu_vocab)

    #save_pickle(gu_vocab, 'vocab_gu2.pkl')
    save_pickle(en_vocab, 'vocab_en3.pkl')


gu_vocab = load_pickle('vocab_gu.pkl')
en_vocab = load_pickle('vocab_en.pkl')



def read_data(filepaths, k = 1):
  with open(filepaths[0], 'r') as fen:
    en_data = fen.readlines()
  with open(filepaths[1],'r') as fgu:
    gu_data = fgu.readlines()
  en_data2 = []
  for i in en_data[:int(len(en_data)*k)]:
    en_data2.append(i.strip())
  gu_data2 = []
  for i in gu_data[:int(len(gu_data)*k)]:
    gu_data2.append(i.strip())
  return en_data2, gu_data2

def data_process_val_test(path, k ):
  data = [];a =0
  en_data2, gu_data2 = read_data(path, k = k)
  assert len(en_data2) == len(gu_data2), f"EN{len(en_data2)}/GU{len(gu_data2)}"
  for i in tqdm(range(0,len(en_data2), 512), desc = f'Running'):
    c =0
    en_list =[];gu_list =[]
    for en in entokenizer(en_data2[i:i +512], en_tokenizer):␣
 ↪#gu_tokenizer(gu_data2[i:i + 512], en_bert_tokenizer):
      tk =[]
      for token in en:
        if token in en_vocab:
          tk.append(en_vocab[token])
        else:
          a +=1
          tk.append(en_vocab["<unk>"])
      en_tensor_ = torch.tensor(tk,
                        dtype=torch.long)
      en_list.append(en_tensor_)
    for gu in gu_tokenizer(gu_data2[i:i + 512], bert_tokenizer):
      tk =[]
      for token in gu:
        if token in gu_vocab:
          tk.append(gu_vocab[token])
        else:
          c +=1
          tk.append(gu_vocab["<unk>"])

      gu_tensor_ = torch.tensor(tk,
                        dtype=torch.long)
      gu_list.append(gu_tensor_)

    assert len(en_list) == len(gu_list)
    for i,j in zip(en_list, gu_list):
      data.append((i, j))
    del en_list
    del gu_list
  print(len(data), a, c)
  return data


def data_process(path):
  data = []
  en_data2, gu_data2 = read_data(path, k = 1)
  assert len(en_data2) == len(gu_data2), f"EN{len(en_data2)}/GU{len(gu_data2)}"
  for i in tqdm(range(0,len(en_data2), 512), desc = f'Running'):
    c =0
    en_list =[];gu_list =[]
    for en in entokenizer(en_data2[i:i +512], en_tokenizer):
        en_tensor_ = torch.tensor([en_vocab[token] for token in en],
                        dtype=torch.long)
        en_list.append(en_tensor_)

    for gu in gu_tokenizer(gu_data2[i:i + 512], bert_tokenizer):
        gu_tensor_ = torch.tensor([gu_vocab[token] for token in gu],
                          dtype=torch.long)
        gu_list.append(gu_tensor_)

    assert len(en_list) == len(gu_list)
    for i,j in zip(en_list, gu_list):
      data.append((i, j))
    del en_list
    del gu_list
  print(len(data))
  return data

train_data = data_process(train_filepaths)
val_data  = data_process_val_test(val_filepaths, 1)
test_data  = data_process_val_test(test_filepaths, 1)

import gc
gc.collect()


def load_embeds_dec(model, tokenizer, vocab, embed_dim= 128):
    vocab_to_embedding_convertor = model.get_input_embeddings()
    # pass the tokens to get the embeddings
    embeddings_index = {}
    for tokens in tqdm(vocab):
      try:
        embeddings = vocab_to_embedding_convertor(torch.tensor(tokens))
      except:
        print(tokens)
        embeddings = np.random.normal(scale=0.5, size=(embed_dim, ))
      embeddings_index[tokens] = embeddings

    return embeddings_index

def load_embeds_enc(root_dir):
    embeddings_index = {}
    f = open(root_dir)

    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

    f.close()
    return embeddings_index

def load_embed_weights_enc(embeddings_index, embed_dim, vocab, vocab_size):
    matrix_len = vocab_size
    print("ENC", vocab_size)
    weights_matrix = np.zeros((matrix_len, embed_dim))
    words_found = 0
    for word,i in vocab.items():
        try:
            weights_matrix[i] = embeddings_index[word]
            words_found += 1
        except:
            weights_matrix[i] = np.random.normal(scale=0.5, size=(embed_dim, ))
    print(words_found/vocab_size)
    weights_matrix = torch.tensor(weights_matrix)
    return weights_matrix


def load_embed_weights_dec(embeddings_index, embed_dim, vocab, vocab_size):

    matrix_len = vocab_size
    print("DEC", vocab_size)
    weights_matrix = np.zeros((matrix_len, embed_dim))
    words_found = 0
    for word,i in tqdm(vocab.items(), desc = 'DEC'):
        try:
            weights_matrix[i] = embeddings_index[word]
            words_found += 1
        except:
            weights_matrix[i] = np.random.normal(scale=0.5, size=(embed_dim, ))
    print(words_found/vocab_size)
    weights_matrix = torch.tensor(weights_matrix)
    return weights_matrix

embeddings_index = load_embeds_enc(GLOVE_TEXT_PATH)␣
 ↪#load_embeds_dec(en_bert_model, en_bert_tokenizer, en_vocab)
weights_matrix = load_embed_weights_enc(embeddings_index, 300,␣
 ↪en_vocab,len(en_vocab)) #load_embed_weights_dec(embeddings_index, 128,␣
 ↪en_vocab, len(en_vocab))

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
BATCH_SIZE = 128

embeddings_index_dec = load_embeds_dec(bert_model, bert_tokenizer, gu_vocab)
weight_matrix_dec = load_embed_weights_dec(embeddings_index_dec, 128, gu_vocab,␣
 ↪len(gu_vocab))

print("STARTING TO CREATE DATALOADERS")
def generate_batch(data_batch, max_len = 40):
  gu_batch, en_batch = [], []
  for gu_item, en_item in data_batch:
    gu_batch.append(torch.cat([torch.tensor([BOS_IDX_EN]), gu_item[:max_len],␣
 ↪torch.tensor([EOS_IDX_EN])], dim=0))
    en_batch.append(torch.cat([torch.tensor([BOS_IDX_EN]), en_item[:max_len],␣
 ↪torch.tensor([EOS_IDX_EN])], dim=0))

  gu_batch = pad_sequence(gu_batch, padding_value=PAD_IDX_EN)
  en_batch = pad_sequence(en_batch, padding_value=PAD_IDX_EN)
  return gu_batch, en_batch

train_iter = DataLoader(train_data, batch_size=BATCH_SIZE,
                         shuffle=True, collate_fn=generate_batch)
valid_iter = DataLoader(val_data, batch_size=BATCH_SIZE,
                        shuffle=True, collate_fn=generate_batch)
test_iter = DataLoader(test_data, batch_size=BATCH_SIZE,
                       shuffle=False, collate_fn=generate_batch)


INPUT_DIM = len(en_vocab)
ENC_EMB_DIM = 300
DEC_EMB_DIM = 128
OUTPUT_DIM = len(gu_vocab)

model = Seq2Seq(INPUT_DIM,
                OUTPUT_DIM,
                ENC_EMB_DIM,
                DEC_EMB_DIM,
                weights_matrix,
                weight_matrix_dec, device)
```

```
params_to_update = model.parameters()
params_to_update = []
for name,param in model.named_parameters():
    if param.requires_grad == True:
        params_to_update.append(param)

optimizer = optim.Adam(params_to_update, lr = 0.001)


criterion = nn.CrossEntropyLoss(ignore_index=0)


def evaluate(model: nn.Module,
             iterator: torch.utils.data.DataLoader,
             criterion: nn.Module):

    epoch_loss = 0
    outputs =[];gold =[];inputs =[]
    with torch.no_grad():

        for _, (src, trg) in enumerate(iterator):
            src, trg = src.to(device), trg.to(device)

            output = model(src, trg, teacher_forcing_ratio = 0)
            output = output[1:].view(-1, output.shape[-1])
            trg = trg[1:].view(-1)
            loss = criterion(output, trg)
            # output = output.argmax(dim =2)
            # outputs.append(output.cpu().numpy())
            # gold.append(trg.cpu().numpy())
            # inputs.append(src.cpu().numpy())

            epoch_loss += loss.item()
    save_pickle(outputs, 'test_ped.pkl')
    save_pickle(gold, 'test_gold.pkl')
    save_pickle(inputs, 'test_sc.pkl')
    return epoch_loss / len(iterator)


def run_model(model,
          train_iterator,
          valid_iterator,
          optimizer,
          criterion,
          teacher_forcing_ratio,
          best_val):
```

```
    model.train()

    decay = 0.999
    for _, (src, trg) in tqdm(enumerate(train_iterator)):
        src, trg = src.to(device), trg.to(device)

        optimizer.zero_grad()
        output = model(src, trg, teacher_forcing_ratio = teacher_forcing_ratio)

        teacher_forcing_ratio = teacher_forcing_ratio*decay

        output = output[1:].view(-1, output.shape[-1])
        trg = trg[1:].view(-1)

        loss = criterion(output, trg)
        loss.backward()

        optimizer.step()
        _loss = loss.item()

        print(f'Step: {_} \t Loss: {_loss}')
        wandb.log(
          {
            "step":_,
            "train_step_loss": _loss
          }
        )
        if _%200 ==0:
          val_loss= evaluate(model, valid_iterator, criterion)
          print(f'Step: {_} \t Val Loss: {val_loss}')
          wandb.log(
          {
            "step":_,
            "val_step_loss": val_loss
          }
          )
          if val_loss < best_val:
              torch.save(model.state_dict(), f'model_{epoch}_decay_final.pth')
              best_val = val_loss

    return teacher_forcing_ratio, best_val

N_EPOCHS = 30
best_val = float('inf')
teacher_forcing_ratio =1
for epoch in range(N_EPOCHS):
    teacher_forcing_ratio, best_val = run_model(model, train_iter, valid_iter,␣
 ↪optimizer, criterion, teacher_forcing_ratio=teacher_forcing_ratio, best_val=␣
 ↪best_val)

# model.load_state_dict(torch.load('model_2_decay_final.pth'))
# print('model loaded')
# model.to(device)
# model.eval()
test_loss = evaluate(model, test_iter, criterion)

print(f'| Test Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss):7.3f} |')
```