Task2-WA3

May 2, 2022

[ ]:
```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False


if IN_COLAB:
  from google.colab import drive
  drive.mount('/content/drive')
  import os
  !pip install faiss-gpu
  import faiss
  !pip install -U nltk
  !pip install -q wordcloud
  import wordcloud
  os.chdir('/content/drive/MyDrive/Documents/Sem6-drive/DL/Assignments/
 ↪3Assignment/')
```

[ ]:
```
import torch
import torch.nn as nn
import torch.functional as F
import torchvision
from torch.utils.data import Dataset, DataLoader
import glob
import numpy as np
import pandas
import matplotlib.pyplot as plt
from torchvision.io import read_image
from PIL import Image
from torchvision import transforms
import pandas as pd
import time
import copy
import torch.optim as optim
from torch.optim import lr_scheduler
from torchvision import datasets, models, transforms
```

```
from PIL import Image
import cv2
from tqdm import tqdm
from sklearn.metrics import confusion_matrix
import itertools
from sklearn.model_selection import train_test_split


from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import time
import os
from tqdm import tqdm
from torch.utils.data import Dataset, DataLoader, SubsetRandomSampler
from torchvision.transforms import ToTensor, Lambda
import inspect
import seaborn as sns
import itertools
if torch.cuda.is_available:
  device = torch.device('cuda:0')
else:
  device = torch.device('cpu')

import re
from collections import Counter

import h5py
import faiss
from os.path import join, exists, isfile, realpath, dirname
from math import log10, ceil
from os import makedirs, remove, chdir, environ
import string
import torch
import torch.nn as nn
import torch.nn.functional as F
from sklearn.neighbors import NearestNeighbors
import numpy as np
import nltk
```

```
from nltk import word_tokenize
from nltk.translate.bleu_score import sentence_bleu

from torch.nn.utils.rnn import pack_padded_sequence
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

[ ]:
```
True
```

1 Parameters

[ ]:
```
batchsize = 16
log_step = 1
save_step = 10
model_path = './model'
num_clusters = 64
dataPath = './data/'
threads = 2
lr = 0.01
lrGamma = 0.5
lrStep = 5
num_epochs = 100
```

2 Helper Functions

[ ]:
```
def get_captions(captions):
    image_to_caption_map = {}
    for caption in captions:
        image= caption.split('\t')[0][:-2]
        image_caption = caption.split('\t')[1].strip()
        if image not in image_to_caption_map:
            image_to_caption_map.update({image: [image_caption]})
        else:
            image_to_caption_map[image].append(image_caption)
    return image_to_caption_map



class Flatten(nn.Module):
    def forward(self, input):
        return input.view(input.size(0), -1)
```

```
class L2Norm(nn.Module):
    def __init__(self, dim=1):
        super().__init__()
        self.dim = dim

    def forward(self, input):
        return F.normalize(input, p=2, dim=self.dim)
```

3 Data

[ ]:
```
class ImageDataset(Dataset):
    def __init__(self, transforms = None , target_transforms = None):
        with  open("./Data/26/image_names.txt", "r") as f:
            image_names = f.readlines()
            image_names = list(map( lambda x: x.strip(), image_names))

        self.image_names = image_names
        self.transforms = transforms
        self.target_transforms = target_transforms

    def __getitem__(self, index):
        image_name = self.image_names[index]
        image = Image.open("./Data/Images/" + image_name)

        if self.transforms is not None:
            image = self.transforms(image)

        return image, index

    def __len__(self):
        return len(self.image_names)
```

[ ]:
```
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

4 CNN Encoder Model

[ ]:
```
# we are using VGG16 as the base model
encoder_dim = 512
encoder = models.vgg16(pretrained=True)
# capture only feature part and remove last relu and maxpool
layers = list(encoder.features.children())[:-2]

# if using pretrained then only train conv5_1, conv5_2, and conv5_3
for l in layers[:-5]:
    for p in l.parameters():
        p.requires_grad = True
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to
/root/.cache/torch/hub/checkpoints/vgg16-397923af.pth

  0%|          | 0.00/528M [00:00<?, ?B/s]
```

[ ]:
```
encoder = nn.Sequential(*layers)
model = nn.Module()
model.add_module('encoder', encoder)

model = model.to(device)
```

5 Cluster Formation

[ ]:
```
def get_clusters(cluster_set):
    nDescriptors = 50000
    nPerImage = 100
    nIm = ceil(nDescriptors/nPerImage)

    sampler = SubsetRandomSampler(np.random.choice(len(cluster_set), nIm,␣
 ↪replace=False))
    data_loader = DataLoader(dataset=cluster_set, batch_size=64, shuffle=False,
                pin_memory=True,
                sampler=sampler)

    if not exists(join(dataPath, 'centroids')):
        makedirs(join(dataPath, 'centroids'))

    initcache = join(dataPath, 'centroids', 'VGG16'  + '_' + str(num_clusters)␣
 ↪+ '_desc_cen.hdf5')
    with h5py.File(initcache, mode='w') as h5:
        with torch.no_grad():
            model.eval()
```

```
            print('====> Extracting Descriptors')
            dbFeat = h5.create_dataset("descriptors",
                        [nDescriptors, encoder_dim],
                        dtype=np.float32)

            for iteration, (input, indices) in enumerate(data_loader, 1):
                input = input.to(device)
                image_descriptors = model.encoder(input).view(input.size(0),␣
↪encoder_dim, -1).permute(0, 2, 1)

                batchix = (iteration-1)*batchsize*nPerImage
                for ix in range(image_descriptors.size(0)):
                    # sample different location for each image in batch
                    sample = np.random.choice(image_descriptors.size(1),␣
↪nPerImage, replace=False)
                    startix = batchix + ix*nPerImage
                    dbFeat[startix:startix+nPerImage, :] =␣
↪image_descriptors[ix, sample, :].detach().cpu().numpy()

                if iteration % 50 == 0 or len(data_loader) <= 10:
                    print("==> Batch ({}/{})".format(iteration,
                        ceil(nIm/batchsize)), flush=True)
                del input, image_descriptors

        print('====> Clustering..')
        niter = 100
        kmeans = faiss.Kmeans(encoder_dim, num_clusters, niter=niter,␣
↪verbose=False)
        kmeans.train(dbFeat[...])

        print('====> Storing centroids', kmeans.centroids.shape)
        h5.create_dataset('centroids', data=kmeans.centroids)
        print('====> Done!')
```

[ ]:
```
whole_train_set = ImageDataset(transforms = preprocess)
print('===> Calculating descriptors and clusters')
# get_clusters(whole_train_set)
```

```
===> Calculating descriptors and clusters
```

6 VLAD

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
from sklearn.neighbors import NearestNeighbors
import numpy as np

# based on https://github.com/lyakaap/NetVLAD-pytorch/blob/master/netvlad.py
class NetVLAD(nn.Module):
    """NetVLAD layer implementation"""

    def __init__(self, num_clusters=64, dim=128,
                 normalize_input=True, vladv2=True):
        """
        Args:
            num_clusters : int
                The number of clusters
            dim : int
                Dimension of descriptors
            alpha : float
                Parameter of initialization. Larger value is harder assignment.
            normalize_input : bool
                If true, descriptor-wise L2 normalization is applied to input.
            vladv2 : bool
                If true, use vladv2 otherwise use vladv1
        """
        super(NetVLAD, self).__init__()
        self.num_clusters = num_clusters
        self.dim = dim
        self.alpha = 0
        self.vladv2 = vladv2
        self.normalize_input = normalize_input
        self.conv = nn.Conv2d(dim, num_clusters, kernel_size=(1, 1),␣
↪bias=vladv2)
        self.centroids = nn.Parameter(torch.rand(num_clusters, dim))

    def init_params(self, clsts, traindescs):
        #TODO replace numpy ops with pytorch ops
        if self.vladv2 == False:
            clstsAssign = clsts / np.linalg.norm(clsts, axis=1, keepdims=True)
            dots = np.dot(clstsAssign, traindescs.T)
            dots.sort(0)
            dots = dots[::-1, :] # sort, descending

            self.alpha = (-np.log(0.01) / np.mean(dots[0,:] - dots[1,:])).item()
            self.centroids = nn.Parameter(torch.from_numpy(clsts))
            self.conv.weight = nn.Parameter(torch.from_numpy(self.
↪alpha*clstsAssign).unsqueeze(2).unsqueeze(3))
            self.conv.bias = None
        else:
            knn = NearestNeighbors(n_jobs=-1) #TODO faiss?
```

```
            knn.fit(traindescs)
            del traindescs
            dsSq = np.square(knn.kneighbors(clsts, 2)[1])
            del knn
            self.alpha = (-np.log(0.01) / np.mean(dsSq[:,1] - dsSq[:,0])).item()
            self.centroids = nn.Parameter(torch.from_numpy(clsts))
            del clsts, dsSq

            self.conv.weight = nn.Parameter(
                (2.0 * self.alpha * self.centroids).unsqueeze(-1).unsqueeze(-1)
            )
            self.conv.bias = nn.Parameter(
                - self.alpha * self.centroids.norm(dim=1)
            )

    def forward(self, x):
        N, C = x.shape[:2]

        if self.normalize_input:
            x = F.normalize(x, p=2, dim=1)  # across descriptor dim

        # soft-assignment
        soft_assign = self.conv(x).view(N, self.num_clusters, -1)
        soft_assign = F.softmax(soft_assign, dim=1)

        x_flatten = x.view(N, C, -1)

        # calculate residuals to each clusters
        vlad = torch.zeros([N, self.num_clusters, C], dtype=x.dtype, layout=x.
↪layout, device=x.device)
        for C in range(self.num_clusters): # slower than non-looped, but lower␣
↪memory usage
            residual = x_flatten.unsqueeze(0).permute(1, 0, 2, 3) - \
                    self.centroids[C:C+1, :].expand(x_flatten.size(-1), -1, -1).
↪permute(1, 2, 0).unsqueeze(0)
            residual *= soft_assign[:,C:C+1,:].unsqueeze(2)
            vlad[:,C:C+1,:] = residual.sum(dim=-1)

        vlad = F.normalize(vlad, p=2, dim=2)  # intra-normalization
        vlad = vlad.view(x.size(0), -1)  # flatten
        vlad = F.normalize(vlad, p=2, dim=1)  # L2 normalize

        return vlad
```

[ ]:
```
net_vlad = NetVLAD(num_clusters=num_clusters, dim=encoder_dim, vladv2=True)
initcache = join(dataPath,'centroids', 'VGG16'  + '_' + str(num_clusters) +␣
↪'_desc_cen.hdf5')
```

```
with h5py.File(initcache, mode='r') as h5:
    clsts = h5.get("centroids")[...]
    traindescs = h5.get("descriptors")[...]
    net_vlad.init_params(clsts, traindescs)
    del clsts, traindescs

model.add_module('pool', net_vlad)
```

[ ]:
```
optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()),␣
↪lr=lr)#, betas=(0,0.9))
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=lrStep,␣
↪gamma=lrGamma)
```

7 Adding a FC layer at the end

[ ]:
```
fc_layers = [nn.Linear(in_features=32768, out_features = 15000), nn.
↪Linear(in_features = 15000, out_features = 300)]
```

[ ]:
```
fc_layer = nn.Sequential(*fc_layers)
```

[ ]:
```
model.add_module('fc', fc_layer)
```

8 Selec the Vocalblory

[ ]:
```
with open('./Data/26/captions.txt', 'r') as f:
            captions = f.readlines()
```

[ ]:
```
image_to_captions_map = get_captions(captions)
```

[ ]:
```
sentences = []
for captions in image_to_captions_map.values():
  sentences += captions
```

[ ]:
```
sentences = [word_tokenize(sentence.lower()) for sentence in sentences]

# Create the vocabulary. Note that we add an <UNK> token to represent words not␣
↪in our vocabulary.
vocabularySize = 10000
word_counts = Counter([word for sentence in sentences for word in sentence])
vocabulary = [e[0] for e in word_counts.most_common(vocabularySize)]
word2index = {word:index for index,word in enumerate(vocabulary)}
```

[ ]:
```
index2word = {index:word for word, index in word2index.items()}
```

9 GloVe Embeddings

[ ]:
```
# RUN ONLY FIRST TIME
# !wget http://nlp.stanford.edu/data/glove.6B.zip
# !unzip glove.6B.zip
# !ls -lat
```

[ ]:
```
vocab,embeddings = [],[]
with open('glove.6B.300d.txt','rt') as fi:
    full_content = fi.read().strip().split('\n')
for i in range(len(full_content)):
    i_word = full_content[i].split(' ')[0]
    i_embeddings = [float(val) for val in full_content[i].split(' ')[1:]]
    if i_word in word2index:
      vocab.append(i_word)
      embeddings.append(i_embeddings)
```

[ ]:
```
vocab_npa = np.array(vocab)
embs_npa = np.array(embeddings)
```

[ ]:
```
del vocab
del embeddings
```

[ ]:
```
vocab_npa.shape, embs_npa.shape
```

[ ]:
```
((7976,), (7976, 300))
```

[ ]:
```
[word2index.pop(k) for k in set(word2index.keys()) - set(vocab_npa.tolist())]
```

[ ]:
```
len(word2index.keys())
```

[ ]:
```
7976
```

[ ]:
```
word2index = {w:i+5 for i, w in enumerate(word2index.keys())}
```

[ ]:
```
#insert '<pad>' and '<unk>' tokens at start of vocab_npa.
vocab_npa = np.insert(vocab_npa, 0, '<pad>')
vocab_npa = np.insert(vocab_npa, 1, '<unk>')
vocab_npa = np.insert(vocab_npa, 2, '<start>')
vocab_npa = np.insert(vocab_npa, 2, '<end>')

print(vocab_npa[:10])
```

```
pad_emb_npa = np.zeros((1,embs_npa.shape[1]))   #embedding for '<pad>' token.
unk_emb_npa = np.mean(embs_npa,axis=0,keepdims=True)    #embedding for '<unk>'␣
 ↪token.
start_emb_npa = np.ones((1, embs_npa.shape[1]))
end_emb_npa = np.ones((1, embs_npa.shape[1]))

#insert embeddings for pad and unk tokens at top of embs_npa.
embs_npa = np.vstack((pad_emb_npa,unk_emb_npa,embs_npa, start_emb_npa,␣
 ↪end_emb_npa))
print(embs_npa.shape)
```

```
['<pad>' '<unk>' '<end>' '<start>' 'the' ',' '.' 'of' 'to' 'and']
(7980, 300)
```

[ ]:
```
word2index.update({'<pad>':0, '<unk>':1, '<start>':2, '<end>':3})
```

[ ]:
```
index2word.update({0:'<pad>', 1:'<unk>', 2:'<start>', 3:'<end>'})
```

[ ]:
```
embs_npa.shape
```

[ ]:
```
(7980, 300)
```

[ ]:
```
len(word2index.keys())
```

[ ]:
```
7980
```

10 Decoder

[ ]:
```
class DecoderRNN(nn.Module):
    def __init__(self, embed_size, hidden_size, vocab_size, num_layers,␣
 ↪max_seq_length=20):
        """Set the hyper-parameters and build the layers."""
        super(DecoderRNN, self).__init__()
        #self.embed = nn.Embedding(vocab_size, embed_size)
        self.embed = torch.nn.Embedding.from_pretrained(torch.
 ↪from_numpy(embs_npa).float())

        self.rnn = nn.RNN(embed_size, hidden_size, num_layers, batch_first=True)
        self.linear = nn.Linear(hidden_size, vocab_size)
        self.max_seg_length = max_seq_length

    def forward(self, features, captions, lengths):
        """Decode image feature vectors and generates captions."""
        embeddings = self.embed(captions)
        embeddings = torch.cat((features.unsqueeze(1), embeddings), 1)
```

```
        packed = pack_padded_sequence(embeddings, lengths, batch_first=True)
        hiddens, _ = self.rnn(packed)
        outputs = self.linear(hiddens[0])
        return outputs

    def sample(self, features, states=None):
        """Generate captions for given image features using greedy search."""
        sampled_ids = []
        inputs = features.unsqueeze(1)
        for i in range(self.max_seg_length):
            hiddens, states = self.rnn(inputs, states)          # hiddens:␣
↪(batch_size, 1, hidden_size)
            outputs = self.linear(hiddens.squeeze(1))            # outputs: ␣
↪(batch_size, vocab_size)
            _, predicted = outputs.max(1)                        # predicted:␣
↪(batch_size)
            sampled_ids.append(predicted)
            inputs = self.embed(predicted)                       # inputs:␣
↪(batch_size, embed_size)
            inputs = inputs.unsqueeze(1)                         # inputs:␣
↪(batch_size, 1, embed_size)
        sampled_ids = torch.stack(sampled_ids, 1)                # sampled_ids:␣
↪(batch_size, max_seq_length)
        return sampled_ids
```

[ ]:
```
class FullDataset(Dataset):
    def __init__(self, transforms = None , target_transforms = None):
        with  open("./Data/26/image_names.txt", "r") as f:
            image_names = f.readlines()
            image_names = list(map( lambda x: x.strip(), image_names))
        with open('./Data/26/captions.txt', 'r') as f:
            captions = f.readlines()
        self.image_names = image_names
        self.image_to_caption_map  = get_captions(captions)
        self.transforms = transforms
        self.target_transforms = target_transforms

    def __getitem__(self, index):
        image_name = self.image_names[index]
        image = Image.open("./Data/Images/" + image_name)

        if self.transforms is not None:
            image = self.transforms(image)

        captions = self.image_to_caption_map[image_name]
        caption = np.random.choice(captions, 1)
```

```
        tokens = nltk.tokenize.word_tokenize(str(caption).lower())
        caption = []
        caption.append(word2index['<start>'])
        caption.extend([word2index.get(token, 1) for token in tokens])
        caption.append(word2index['<end>'])
        target = torch.Tensor(caption)


        return image, target

    def __len__(self):
        return len(self.image_names)
```

[ ]:
```
def collate_fn(data):
    """Creates mini-batch tensors from the list of tuples (image, caption).

    We should build custom collate_fn rather than using default collate_fn,
    because merging caption (including padding) is not supported in default.
    Args:
        data: list of tuple (image, caption).
            - image: torch tensor of shape (3, 256, 256).
            - caption: torch tensor of shape (?); variable length.
    Returns:
        images: torch tensor of shape (batch_size, 3, 256, 256).
        targets: torch tensor of shape (batch_size, padded_length).
        lengths: list; valid length for each padded caption.
    """
    # Sort a data list by caption length (descending order).
    data.sort(key=lambda x: len(x[1]), reverse=True)
    images, captions = zip(*data)

    # Merge images (from tuple of 3D tensor to 4D tensor).
    images = torch.stack(images, 0)

    # Merge captions (from tuple of 1D tensor to 2D tensor).
    lengths = [len(cap) for cap in captions]
    targets = torch.zeros(len(captions), max(lengths)).long()
    for i, cap in enumerate(captions):
        end = lengths[i]
        targets[i, :end] = (cap[:end])
    return images, targets, lengths

def get_loader():
    """Returns torch.utils.data.DataLoader for custom coco dataset."""
    # COCO caption dataset
    dataset = FullDataset(transforms=preprocess)
```

```
    # Data loader for COCO dataset
    # This will return (images, captions, lengths) for each iteration.
    # images: a tensor of shape (batch_size, 3, 224, 224).
    # captions: a tensor of shape (batch_size, padded_length).
    # lengths: a list indicating valid length for each caption. length is␣
 ↪(batch_size).
    data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                              batch_size=batchsize,
                                              shuffle=True,
                                              collate_fn=collate_fn)
    return data_loader
```

11 Training Loop

[ ]:
```
torch.cuda.empty_cache()
```

[ ]:
```
data_loader = get_loader()

# Build the models
encoder = model.to(device)
decoder = DecoderRNN(300, 1024, len(word2index.keys()), num_layers=1)
decoder = decoder.to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
params = list(filter(lambda p: p.requires_grad, decoder.parameters()))
params +=list(filter(lambda p: p.requires_grad, encoder.parameters()))
# params = list(decoder.parameters()) + list(encoder.linear.parameters()) +␣
 ↪list(encoder.bn.parameters())
optimizer = torch.optim.SGD(params, lr=lr)
```

[ ]:
```
checkpoint = torch.load('./models/model-90.ckpt')
encoder.load_state_dict(checkpoint['encoder_state_dict'])
decoder.load_state_dict(checkpoint['decoder_state_dict'])
#optimizer.load_state_dict(checkpoint['optimizer_state_dict'])


model.train()
```

[ ]:
```
# Train the models
total_step = len(data_loader)
for epoch in range(1):
    for i, (images, captions, lengths) in enumerate(data_loader):
```

```
        # Set mini-batch dataset
        images = images.to(device)
        captions = captions.to(device)
        targets = pack_padded_sequence(captions, lengths, batch_first=True)[0]
        #print(targets)
        # Forward, backward and optimize
        # features = encoder(images)

        image_encoding = encoder.encoder(images)
        vlad_encoding = model.pool(image_encoding)
        features = model.fc(vlad_encoding)
        outputs = decoder(features, captions, lengths)
        #print(outputs)
        loss = criterion(outputs, targets)
        decoder.zero_grad()
        encoder.zero_grad()
        loss.backward()
        optimizer.step()

        # Print log info
        if i % log_step == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Perplexity: {:5.
↪4f}'
                  .format(epoch, num_epochs, i, total_step, loss.item(), np.
↪exp(loss.item())))

   if epoch % 10 == 0:
     torch.save({
           'epoch': epoch,
           'encoder_state_dict': encoder.state_dict(),
           'decoder_state_dict': decoder.state_dict(),
           'optimizer_state_dict': optimizer.state_dict(),
           }, f'./models/model-{epoch}.ckpt')

        # Save the model checkpoints
        # if (i+1) % save_step == 0:
        #     torch.save(decoder.state_dict(), os.path.join(
        #         model_path, 'decoder-{}-{}.ckpt'.format(epoch+1, i+1)))
        #     torch.save(encoder.state_dict(), os.path.join(
        #         model_path, 'encoder-{}-{}.ckpt'.format(epoch+1, i+1)))
```

[ ]:
```
torch.save({
           'epoch': epoch,
           'encoder_state_dict': encoder.state_dict(),
           'decoder_state_dict': decoder.state_dict(),
           'optimizer_state_dict': optimizer.state_dict(),
           }, './models/model-100.ckpt')
```

[ ]:
```
for epoch in range():
    for i, (images, captions, lengths) in enumerate(data_loader):

        # Set mini-batch dataset
        images = images.to(device)
        captions = captions.to(device)
        targets = pack_padded_sequence(captions, lengths, batch_first=True)[0]
        #print(targets)
        # Forward, backward and optimize
        # features = encoder(images)

        image_encoding = encoder.encoder(images)
        vlad_encoding = model.pool(image_encoding)
        features = model.fc(vlad_encoding)
        outputs = decoder(features, captions, lengths)
        #print(outputs)
        loss = criterion(outputs, targets)
        decoder.zero_grad()
        encoder.zero_grad()
        loss.backward()
        optimizer.step()

        # Print log info
        if i % log_step == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Perplexity: {:5.
↪4f}'
                  .format(epoch, num_epochs, i, total_step, loss.item(), np.
↪exp(loss.item())))
```

12 Predic the Caption

[ ]:
```
def load_image(image_path, transform=None):
    image = Image.open(image_path).convert('RGB')
    image = image.resize([224, 224], Image.LANCZOS)

    if transform is not None:
        image = transform(image).unsqueeze(0)

    return image

def predict(image_path):
    # # Image preprocessing
    # transform = transforms.Compose([
    #     transforms.ToTensor(),
```

```
    #     transforms.Normalize((0.485, 0.456, 0.406),
    #                          (0.229, 0.224, 0.225))])

    # # Load vocabulary wrapper
    # with open(args.vocab_path, 'rb') as f:
    #     vocab = pickle.load(f)

    # # Build models
    # encoder = EncoderCNN(args.embed_size).eval()  # eval mode (batchnorm uses␣
↪moving mean/variance)
    # decoder = DecoderRNN(args.embed_size, args.hidden_size, len(vocab), args.
↪num_layers)
    # encoder = encoder.to(device)
    # decoder = decoder.to(device)

    # # Load the trained model parameters
    # encoder.load_state_dict(torch.load(args.encoder_path))
    # decoder.load_state_dict(torch.load(args.decoder_path))

    # Prepare an image
    image = load_image(image_path, transform = preprocess)
    image_tensor = image.to(device)

    # Generate an caption from the image
    model.eval()

    image_encoding = model.encoder(image_tensor)
    vlad_encoding = model.pool(image_encoding)
    feature = model.fc(vlad_encoding)
    print(feature)
    sampled_ids = decoder.sample(feature)

    sampled_ids = sampled_ids[0].cpu().numpy()          # (1, max_seq_length)␣
↪-> (max_seq_length)

    # Convert word_ids to words
    sampled_caption = []
    for word_id in sampled_ids:
        word = index2word[word_id]
        sampled_caption.append(word)
        if word == '<end>':
            break
    sentence = ' '.join(sampled_caption)

    # Print out the image and the generated caption
    print (sentence)
    image = Image.open(image_path)
```

```
    plt.imshow(np.asarray(image))
```

[ ]:
```
predict('Data/Images/55470226_52ff517151.jpg')
```

13 IGNORE

[ ]:
```
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
```

[3]:
```
# Run this only if you are using Google Colab
from google.colab import drive
import os

drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
!jupyter nbconvert --to pdf /content/drive/MyDrive/Documents/Sem6-drive/DL/
↪Assignments/3Assignment/Task2-WA3.ipynb
```

```
[NbConvertApp] Converting notebook /content/drive/MyDrive/Documents/Sem6-drive/D
L/Assignments/3Assignment/Task2-A3.ipynb to pdf
[NbConvertApp] Writing 104969 bytes to ./notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', './notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', './notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 90600 bytes to /content/drive/MyDrive/Documents/Sem6-driv
e/DL/Assignments/3Assignment/Task2-A3.pdf
```

[ ]:
```

```

Task2-A3

May 2, 2022

[ ]:
```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False


if IN_COLAB:
  from google.colab import drive
  drive.mount('/content/drive')
  import os
  !pip install faiss-gpu
  import faiss
  !pip install -U nltk
  !pip install -q wordcloud
  import wordcloud
  os.chdir('/content/drive/MyDrive/Documents/Sem6-drive/DL/Assignments/
 ↪3Assignment/')
```

[ ]:
```
import torch
import torch.nn as nn
import torch.functional as F
import torchvision
from torch.utils.data import Dataset, DataLoader
import glob
import numpy as np
import pandas
import matplotlib.pyplot as plt
from torchvision.io import read_image
from PIL import Image
from torchvision import transforms
import pandas as pd
import time
import copy
import torch.optim as optim
from torch.optim import lr_scheduler
from torchvision import datasets, models, transforms
```

```
from PIL import Image
import cv2
from tqdm import tqdm
from sklearn.metrics import confusion_matrix
import itertools
from sklearn.model_selection import train_test_split


from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import time
import os
from tqdm import tqdm
from torch.utils.data import Dataset, DataLoader, SubsetRandomSampler
from torchvision.transforms import ToTensor, Lambda
import inspect
import seaborn as sns
import itertools
if torch.cuda.is_available:
  device = torch.device('cuda:0')
else:
  device = torch.device('cpu')

import re
from collections import Counter

import h5py
import faiss
from os.path import join, exists, isfile, realpath, dirname
from math import log10, ceil
from os import makedirs, remove, chdir, environ
import string
import torch
import torch.nn as nn
import torch.nn.functional as F
from sklearn.neighbors import NearestNeighbors
import numpy as np
import nltk
```

```
from nltk import word_tokenize
from nltk.translate.bleu_score import sentence_bleu

from torch.nn.utils.rnn import pack_padded_sequence
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

[ ]:
```
True
```

1 Parameters

[ ]:
```
batchsize = 16
log_step = 1
save_step = 10
model_path = './model'
num_clusters = 64
dataPath = './data/'
threads = 2
lr = 0.01
lrGamma = 0.5
lrStep = 5
num_epochs = 100
```

2 Helper Functions

[ ]:
```
def get_captions(captions):
    image_to_caption_map = {}
    for caption in captions:
        image= caption.split('\t')[0][:-2]
        image_caption = caption.split('\t')[1].strip()
        if image not in image_to_caption_map:
            image_to_caption_map.update({image: [image_caption]})
        else:
            image_to_caption_map[image].append(image_caption)
    return image_to_caption_map



class Flatten(nn.Module):
    def forward(self, input):
        return input.view(input.size(0), -1)
```

```
class L2Norm(nn.Module):
    def __init__(self, dim=1):
        super().__init__()
        self.dim = dim

    def forward(self, input):
        return F.normalize(input, p=2, dim=self.dim)
```

3 Data

[ ]:
```
class ImageDataset(Dataset):
    def __init__(self, transforms = None , target_transforms = None):
        with  open("./Data/26/image_names.txt", "r") as f:
            image_names = f.readlines()
            image_names = list(map( lambda x: x.strip(), image_names))

        self.image_names = image_names
        self.transforms = transforms
        self.target_transforms = target_transforms

    def __getitem__(self, index):
        image_name = self.image_names[index]
        image = Image.open("./Data/Images/" + image_name)

        if self.transforms is not None:
            image = self.transforms(image)

        return image, index

    def __len__(self):
        return len(self.image_names)
```

[ ]:
```
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

4 CNN Encoder Model

[ ]:
```
# we are using VGG16 as the base model
encoder_dim = 512
encoder = models.vgg16(pretrained=True)
# capture only feature part and remove last relu and maxpool
layers = list(encoder.features.children())[:-2]

# if using pretrained then only train conv5_1, conv5_2, and conv5_3
for l in layers[:-5]:
    for p in l.parameters():
        p.requires_grad = True
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to
/root/.cache/torch/hub/checkpoints/vgg16-397923af.pth

  0%|          | 0.00/528M [00:00<?, ?B/s]
```

[ ]:
```
encoder = nn.Sequential(*layers)
model = nn.Module()
model.add_module('encoder', encoder)

model = model.to(device)
```

5 Cluster Formation

[ ]:
```
def get_clusters(cluster_set):
    nDescriptors = 50000
    nPerImage = 100
    nIm = ceil(nDescriptors/nPerImage)

    sampler = SubsetRandomSampler(np.random.choice(len(cluster_set), nIm,␣
 ↪replace=False))
    data_loader = DataLoader(dataset=cluster_set, batch_size=64, shuffle=False,
                pin_memory=True,
                sampler=sampler)

    if not exists(join(dataPath, 'centroids')):
        makedirs(join(dataPath, 'centroids'))

    initcache = join(dataPath, 'centroids', 'VGG16'  + '_' + str(num_clusters)␣
 ↪+ '_desc_cen.hdf5')
    with h5py.File(initcache, mode='w') as h5:
        with torch.no_grad():
            model.eval()
```

```
            print('====> Extracting Descriptors')
            dbFeat = h5.create_dataset("descriptors",
                        [nDescriptors, encoder_dim],
                        dtype=np.float32)

            for iteration, (input, indices) in enumerate(data_loader, 1):
                input = input.to(device)
                image_descriptors = model.encoder(input).view(input.size(0),␣
↪encoder_dim, -1).permute(0, 2, 1)

                batchix = (iteration-1)*batchsize*nPerImage
                for ix in range(image_descriptors.size(0)):
                    # sample different location for each image in batch
                    sample = np.random.choice(image_descriptors.size(1),␣
↪nPerImage, replace=False)
                    startix = batchix + ix*nPerImage
                    dbFeat[startix:startix+nPerImage, :] =␣
↪image_descriptors[ix, sample, :].detach().cpu().numpy()

                if iteration % 50 == 0 or len(data_loader) <= 10:
                    print("==> Batch ({}/{})".format(iteration,
                        ceil(nIm/batchsize)), flush=True)
                del input, image_descriptors

        print('====> Clustering..')
        niter = 100
        kmeans = faiss.Kmeans(encoder_dim, num_clusters, niter=niter,␣
↪verbose=False)
        kmeans.train(dbFeat[...])

        print('====> Storing centroids', kmeans.centroids.shape)
        h5.create_dataset('centroids', data=kmeans.centroids)
        print('====> Done!')
```

[ ]:
```
whole_train_set = ImageDataset(transforms = preprocess)
print('===> Calculating descriptors and clusters')
# get_clusters(whole_train_set)
```

```
===> Calculating descriptors and clusters
```

6 VLAD

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
from sklearn.neighbors import NearestNeighbors
import numpy as np

# based on https://github.com/lyakaap/NetVLAD-pytorch/blob/master/netvlad.py
class NetVLAD(nn.Module):
    """NetVLAD layer implementation"""

    def __init__(self, num_clusters=64, dim=128,
                 normalize_input=True, vladv2=True):
        """
        Args:
            num_clusters : int
                The number of clusters
            dim : int
                Dimension of descriptors
            alpha : float
                Parameter of initialization. Larger value is harder assignment.
            normalize_input : bool
                If true, descriptor-wise L2 normalization is applied to input.
            vladv2 : bool
                If true, use vladv2 otherwise use vladv1
        """
        super(NetVLAD, self).__init__()
        self.num_clusters = num_clusters
        self.dim = dim
        self.alpha = 0
        self.vladv2 = vladv2
        self.normalize_input = normalize_input
        self.conv = nn.Conv2d(dim, num_clusters, kernel_size=(1, 1),␣
↪bias=vladv2)
        self.centroids = nn.Parameter(torch.rand(num_clusters, dim))

    def init_params(self, clsts, traindescs):
        #TODO replace numpy ops with pytorch ops
        if self.vladv2 == False:
            clstsAssign = clsts / np.linalg.norm(clsts, axis=1, keepdims=True)
            dots = np.dot(clstsAssign, traindescs.T)
            dots.sort(0)
            dots = dots[::-1, :] # sort, descending

            self.alpha = (-np.log(0.01) / np.mean(dots[0,:] - dots[1,:])).item()
            self.centroids = nn.Parameter(torch.from_numpy(clsts))
            self.conv.weight = nn.Parameter(torch.from_numpy(self.
↪alpha*clstsAssign).unsqueeze(2).unsqueeze(3))
            self.conv.bias = None
        else:
            knn = NearestNeighbors(n_jobs=-1) #TODO faiss?
```

```
            knn.fit(traindescs)
            del traindescs
            dsSq = np.square(knn.kneighbors(clsts, 2)[1])
            del knn
            self.alpha = (-np.log(0.01) / np.mean(dsSq[:,1] - dsSq[:,0])).item()
            self.centroids = nn.Parameter(torch.from_numpy(clsts))
            del clsts, dsSq

            self.conv.weight = nn.Parameter(
                (2.0 * self.alpha * self.centroids).unsqueeze(-1).unsqueeze(-1)
            )
            self.conv.bias = nn.Parameter(
                - self.alpha * self.centroids.norm(dim=1)
            )

    def forward(self, x):
        N, C = x.shape[:2]

        if self.normalize_input:
            x = F.normalize(x, p=2, dim=1)  # across descriptor dim

        # soft-assignment
        soft_assign = self.conv(x).view(N, self.num_clusters, -1)
        soft_assign = F.softmax(soft_assign, dim=1)

        x_flatten = x.view(N, C, -1)

        # calculate residuals to each clusters
        vlad = torch.zeros([N, self.num_clusters, C], dtype=x.dtype, layout=x.
↪layout, device=x.device)
        for C in range(self.num_clusters): # slower than non-looped, but lower␣
↪memory usage
            residual = x_flatten.unsqueeze(0).permute(1, 0, 2, 3) - \
                    self.centroids[C:C+1, :].expand(x_flatten.size(-1), -1, -1).
↪permute(1, 2, 0).unsqueeze(0)
            residual *= soft_assign[:,C:C+1,:].unsqueeze(2)
            vlad[:,C:C+1,:] = residual.sum(dim=-1)

        vlad = F.normalize(vlad, p=2, dim=2)  # intra-normalization
        vlad = vlad.view(x.size(0), -1)  # flatten
        vlad = F.normalize(vlad, p=2, dim=1)  # L2 normalize

        return vlad
```

[ ]:
```
net_vlad = NetVLAD(num_clusters=num_clusters, dim=encoder_dim, vladv2=True)
initcache = join(dataPath,'centroids', 'VGG16'  + '_' + str(num_clusters) +␣
↪'_desc_cen.hdf5')
```

```
with h5py.File(initcache, mode='r') as h5:
    clsts = h5.get("centroids")[...]
    traindescs = h5.get("descriptors")[...]
    net_vlad.init_params(clsts, traindescs)
    del clsts, traindescs

model.add_module('pool', net_vlad)
```

[ ]:
```
optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()),␣
↪lr=lr)#, betas=(0,0.9))
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=lrStep,␣
↪gamma=lrGamma)
```

7 Adding a FC layer at the end

[ ]:
```
fc_layers = [nn.Linear(in_features=32768, out_features = 15000), nn.
↪Linear(in_features = 15000, out_features = 300)]
```

[ ]:
```
fc_layer = nn.Sequential(*fc_layers)
```

[ ]:
```
model.add_module('fc', fc_layer)
```

8 Selec the Vocalblory

[ ]:
```
with open('./Data/26/captions.txt', 'r') as f:
            captions = f.readlines()
```

[ ]:
```
image_to_captions_map = get_captions(captions)
```

[ ]:
```
sentences = []
for captions in image_to_captions_map.values():
  sentences += captions
```

[ ]:
```
sentences = [word_tokenize(sentence.lower()) for sentence in sentences]

# Create the vocabulary. Note that we add an <UNK> token to represent words not␣
↪in our vocabulary.
vocabularySize = 10000
word_counts = Counter([word for sentence in sentences for word in sentence])
vocabulary = [e[0] for e in word_counts.most_common(vocabularySize)]
word2index = {word:index for index,word in enumerate(vocabulary)}
```

[ ]:
```
index2word = {index:word for word, index in word2index.items()}
```

9 GloVe Embeddings

[ ]:
```
# RUN ONLY FIRST TIME
# !wget http://nlp.stanford.edu/data/glove.6B.zip
# !unzip glove.6B.zip
# !ls -lat
```

[ ]:
```
vocab,embeddings = [],[]
with open('glove.6B.300d.txt','rt') as fi:
    full_content = fi.read().strip().split('\n')
for i in range(len(full_content)):
    i_word = full_content[i].split(' ')[0]
    i_embeddings = [float(val) for val in full_content[i].split(' ')[1:]]
    if i_word in word2index:
      vocab.append(i_word)
      embeddings.append(i_embeddings)
```

[ ]:
```
vocab_npa = np.array(vocab)
embs_npa = np.array(embeddings)
```

[ ]:
```
del vocab
del embeddings
```

[ ]:
```
vocab_npa.shape, embs_npa.shape
```

[ ]:
```
((7976,), (7976, 300))
```

[ ]:
```
[word2index.pop(k) for k in set(word2index.keys()) - set(vocab_npa.tolist())]
```

[ ]:
```
len(word2index.keys())
```

[ ]:
```
7976
```

[ ]:
```
word2index = {w:i+5 for i, w in enumerate(word2index.keys())}
```

[ ]:
```
#insert '<pad>' and '<unk>' tokens at start of vocab_npa.
vocab_npa = np.insert(vocab_npa, 0, '<pad>')
vocab_npa = np.insert(vocab_npa, 1, '<unk>')
vocab_npa = np.insert(vocab_npa, 2, '<start>')
vocab_npa = np.insert(vocab_npa, 2, '<end>')

print(vocab_npa[:10])
```

```
pad_emb_npa = np.zeros((1,embs_npa.shape[1]))   #embedding for '<pad>' token.
unk_emb_npa = np.mean(embs_npa,axis=0,keepdims=True)    #embedding for '<unk>'␣
 ↪token.
start_emb_npa = np.ones((1, embs_npa.shape[1]))
end_emb_npa = np.ones((1, embs_npa.shape[1]))

#insert embeddings for pad and unk tokens at top of embs_npa.
embs_npa = np.vstack((pad_emb_npa,unk_emb_npa,embs_npa, start_emb_npa,␣
 ↪end_emb_npa))
print(embs_npa.shape)
```

```
['<pad>' '<unk>' '<end>' '<start>' 'the' ',' '.' 'of' 'to' 'and']
(7980, 300)
```

[ ]:
```
word2index.update({'<pad>':0, '<unk>':1, '<start>':2, '<end>':3})
```

[ ]:
```
index2word.update({0:'<pad>', 1:'<unk>', 2:'<start>', 3:'<end>'})
```

[ ]:
```
embs_npa.shape
```

[ ]:
```
(7980, 300)
```

[ ]:
```
len(word2index.keys())
```

[ ]:
```
7980
```

10 Decoder

[ ]:
```
class DecoderRNN(nn.Module):
    def __init__(self, embed_size, hidden_size, vocab_size, num_layers,␣
 ↪max_seq_length=20):
        """Set the hyper-parameters and build the layers."""
        super(DecoderRNN, self).__init__()
        #self.embed = nn.Embedding(vocab_size, embed_size)
        self.embed = torch.nn.Embedding.from_pretrained(torch.
 ↪from_numpy(embs_npa).float())

        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers,␣
 ↪batch_first=True)
        self.linear = nn.Linear(hidden_size, vocab_size)
        self.max_seg_length = max_seq_length

    def forward(self, features, captions, lengths):
        """Decode image feature vectors and generates captions."""
        embeddings = self.embed(captions)
```

```
        embeddings = torch.cat((features.unsqueeze(1), embeddings), 1)
        packed = pack_padded_sequence(embeddings, lengths, batch_first=True)
        hiddens, _ = self.lstm(packed)
        outputs = self.linear(hiddens[0])
        return outputs

    def sample(self, features, states=None):
        """Generate captions for given image features using greedy search."""
        sampled_ids = []
        inputs = features.unsqueeze(1)
        for i in range(self.max_seg_length):
            hiddens, states = self.lstm(inputs, states)          # hiddens:␣
↪(batch_size, 1, hidden_size)
            outputs = self.linear(hiddens.squeeze(1))            # outputs: ␣
↪(batch_size, vocab_size)
            _, predicted = outputs.max(1)                        # predicted:␣
↪(batch_size)
            sampled_ids.append(predicted)
            inputs = self.embed(predicted)                       # inputs:␣
↪(batch_size, embed_size)
            inputs = inputs.unsqueeze(1)                         # inputs:␣
↪(batch_size, 1, embed_size)
        sampled_ids = torch.stack(sampled_ids, 1)                # sampled_ids:␣
↪(batch_size, max_seq_length)
        return sampled_ids
```

[ ]:
```
class FullDataset(Dataset):
    def __init__(self, transforms = None , target_transforms = None):
        with  open("./Data/26/image_names.txt", "r") as f:
            image_names = f.readlines()
            image_names = list(map( lambda x: x.strip(), image_names))
        with open('./Data/26/captions.txt', 'r') as f:
            captions = f.readlines()
        self.image_names = image_names
        self.image_to_caption_map  = get_captions(captions)
        self.transforms = transforms
        self.target_transforms = target_transforms

    def __getitem__(self, index):
        image_name = self.image_names[index]
        image = Image.open("./Data/Images/" + image_name)

        if self.transforms is not None:
            image = self.transforms(image)

        captions = self.image_to_caption_map[image_name]
```

```
        caption = np.random.choice(captions, 1)
        tokens = nltk.tokenize.word_tokenize(str(caption).lower())
        caption = []
        caption.append(word2index['<start>'])
        caption.extend([word2index.get(token, 1) for token in tokens])
        caption.append(word2index['<end>'])
        target = torch.Tensor(caption)


        return image, target

    def __len__(self):
        return len(self.image_names)
```

[ ]:
```
def collate_fn(data):
    """Creates mini-batch tensors from the list of tuples (image, caption).

    We should build custom collate_fn rather than using default collate_fn,
    because merging caption (including padding) is not supported in default.
    Args:
        data: list of tuple (image, caption).
            - image: torch tensor of shape (3, 256, 256).
            - caption: torch tensor of shape (?); variable length.
    Returns:
        images: torch tensor of shape (batch_size, 3, 256, 256).
        targets: torch tensor of shape (batch_size, padded_length).
        lengths: list; valid length for each padded caption.
    """
    # Sort a data list by caption length (descending order).
    data.sort(key=lambda x: len(x[1]), reverse=True)
    images, captions = zip(*data)

    # Merge images (from tuple of 3D tensor to 4D tensor).
    images = torch.stack(images, 0)

    # Merge captions (from tuple of 1D tensor to 2D tensor).
    lengths = [len(cap) for cap in captions]
    targets = torch.zeros(len(captions), max(lengths)).long()
    for i, cap in enumerate(captions):
        end = lengths[i]
        targets[i, :end] = (cap[:end])
    return images, targets, lengths

def get_loader():
    """Returns torch.utils.data.DataLoader for custom coco dataset."""
    # COCO caption dataset
    dataset = FullDataset(transforms=preprocess)
```

```
    # Data loader for COCO dataset
    # This will return (images, captions, lengths) for each iteration.
    # images: a tensor of shape (batch_size, 3, 224, 224).
    # captions: a tensor of shape (batch_size, padded_length).
    # lengths: a list indicating valid length for each caption. length is
 ↪(batch_size).
    data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                              batch_size=batchsize,
                                              shuffle=True,
                                              collate_fn=collate_fn)
    return data_loader
```

11 Training Loop

[ ]:
```
torch.cuda.empty_cache()
```

[ ]:
```
data_loader = get_loader()

# Build the models
encoder = model.to(device)
decoder = DecoderRNN(300, 1024, len(word2index.keys()), num_layers=1)
decoder = decoder.to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
params = list(filter(lambda p: p.requires_grad, decoder.parameters()))
params +=list(filter(lambda p: p.requires_grad, encoder.parameters()))
# params = list(decoder.parameters()) + list(encoder.linear.parameters()) +
 ↪list(encoder.bn.parameters())
optimizer = torch.optim.SGD(params, lr=lr)
```

[ ]:
```
checkpoint = torch.load('./models/model-90.ckpt')
encoder.load_state_dict(checkpoint['encoder_state_dict'])
decoder.load_state_dict(checkpoint['decoder_state_dict'])
#optimizer.load_state_dict(checkpoint['optimizer_state_dict'])


model.train()
```

[ ]:
```
# Train the models
total_step = len(data_loader)
for epoch in range(1):
    for i, (images, captions, lengths) in enumerate(data_loader):
```

```
        # Set mini-batch dataset
        images = images.to(device)
        captions = captions.to(device)
        targets = pack_padded_sequence(captions, lengths, batch_first=True)[0]
        #print(targets)
        # Forward, backward and optimize
        # features = encoder(images)

        image_encoding = encoder.encoder(images)
        vlad_encoding = model.pool(image_encoding)
        features = model.fc(vlad_encoding)
        outputs = decoder(features, captions, lengths)
        #print(outputs)
        loss = criterion(outputs, targets)
        decoder.zero_grad()
        encoder.zero_grad()
        loss.backward()
        optimizer.step()

        # Print log info
        if i % log_step == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Perplexity: {:5.
↪4f}'
                  .format(epoch, num_epochs, i, total_step, loss.item(), np.
↪exp(loss.item())))

    if epoch % 10 == 0:
      torch.save({
            'epoch': epoch,
            'encoder_state_dict': encoder.state_dict(),
            'decoder_state_dict': decoder.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            }, f'./models/model-{epoch}.ckpt')

        # Save the model checkpoints
        # if (i+1) % save_step == 0:
        #     torch.save(decoder.state_dict(), os.path.join(
        #         model_path, 'decoder-{}-{}.ckpt'.format(epoch+1, i+1)))
        #     torch.save(encoder.state_dict(), os.path.join(
        #         model_path, 'encoder-{}-{}.ckpt'.format(epoch+1, i+1)))
```

[ ]:
```
torch.save({
            'epoch': epoch,
            'encoder_state_dict': encoder.state_dict(),
            'decoder_state_dict': decoder.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
```

```
            }, './models/model-100.ckpt')
```

[ ]:

[ ]:

[ ]:
```
for epoch in range():
    for i, (images, captions, lengths) in enumerate(data_loader):

        # Set mini-batch dataset
        images = images.to(device)
        captions = captions.to(device)
        targets = pack_padded_sequence(captions, lengths, batch_first=True)[0]
        #print(targets)
        # Forward, backward and optimize
        # features = encoder(images)

        image_encoding = encoder.encoder(images)
        vlad_encoding = model.pool(image_encoding)
        features = model.fc(vlad_encoding)
        outputs = decoder(features, captions, lengths)
        #print(outputs)
        loss = criterion(outputs, targets)
        decoder.zero_grad()
        encoder.zero_grad()
        loss.backward()
        optimizer.step()

        # Print log info
        if i % log_step == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Perplexity: {:5.
↪4f}'
                  .format(epoch, num_epochs, i, total_step, loss.item(), np.
↪exp(loss.item())))
```

12 Predic the Caption

[ ]:
```
def load_image(image_path, transform=None):
    image = Image.open(image_path).convert('RGB')
    image = image.resize([224, 224], Image.LANCZOS)

    if transform is not None:
        image = transform(image).unsqueeze(0)
```

```
    return image

def predict(image_path):
    # # Image preprocessing
    # transform = transforms.Compose([
    #     transforms.ToTensor(),
    #     transforms.Normalize((0.485, 0.456, 0.406),
    #                          (0.229, 0.224, 0.225))])

    # # Load vocabulary wrapper
    # with open(args.vocab_path, 'rb') as f:
    #     vocab = pickle.load(f)

    # # Build models
    # encoder = EncoderCNN(args.embed_size).eval()  # eval mode (batchnorm uses
↪moving mean/variance)
    # decoder = DecoderRNN(args.embed_size, args.hidden_size, len(vocab), args.
↪num_layers)
    # encoder = encoder.to(device)
    # decoder = decoder.to(device)

    # # Load the trained model parameters
    # encoder.load_state_dict(torch.load(args.encoder_path))
    # decoder.load_state_dict(torch.load(args.decoder_path))

    # Prepare an image
    image = load_image(image_path, transform = preprocess)
    image_tensor = image.to(device)

    # Generate an caption from the image
    model.eval()

    image_encoding = model.encoder(image_tensor)
    vlad_encoding = model.pool(image_encoding)
    feature = model.fc(vlad_encoding)
    print(feature)
    sampled_ids = decoder.sample(feature)

    sampled_ids = sampled_ids[0].cpu().numpy()          # (1, max_seq_length)
↪-> (max_seq_length)

    # Convert word_ids to words
    sampled_caption = []
    for word_id in sampled_ids:
        word = index2word[word_id]
        sampled_caption.append(word)
        if word == '<end>':
```

```
            break
    sentence = ' '.join(sampled_caption)

    # Print out the image and the generated caption
    print (sentence)
    image = Image.open(image_path)
    plt.imshow(np.asarray(image))
```

[ ]:
```
predict('Data/Images/55470226_52ff517151.jpg')
```

13 IGNORE

[ ]:
```
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
```

[2]:
```
# Run this only if you are using Google Colab
from google.colab import drive
import os

drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
!jupyter nbconvert --to pdf /content/drive/MyDrive/Documents/Sem6-drive/RL/
↪Tutorial/5Tut/Tutorial5_new.ipynb
```

Colab Setup

```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False
```

```
if IN_COLAB:
  from google.colab import drive
  drive.mount('/content/drive', force_remount = True)
  import os
  os.chdir('/content/drive/MyDrive/CS6910/Assn3/')
```

Importing and Installing Libraries

```
#inctalling transformers library
! pip install transformers
! pip install sentencepiece
```

```
import cv2
import time
import copy
import glob
import itertools
from PIL import Image

import torch
import torchtext
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from torchvision import transforms
from torchvision import datasets, models
from torchvision.io import read_image
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import re
import tqdm
import nltk
```

```
import unicodedata
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
from transformers import AutoModel, AutoTokenizer

nltk.download('punkt')

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


tokenizer = AutoTokenizer.from_pretrained('ai4bharat/indic-bert')
model = AutoModel.from_pretrained('ai4bharat/indic-bert')

print(device)
```

Building the Vocabulary for Languages

```
dev_en_path  = '/content/drive/MyDrive/CS6910/Assn3/dev.en'
dev_te_path  = '/content/drive/MyDrive/CS6910/Assn3/dev.te'
test_en_path = '/content/drive/MyDrive/CS6910/Assn3/test.en'
test_te_path = '/content/drive/MyDrive/CS6910/Assn3/test.te'


def string_norm(s, lang):
  if lang == 'en':
    s = s.lower().strip()
    s = unicodedata.normalize('NFD', s).encode('ascii', 'ignore').decode()
    s = re.sub(r"([.!?])", r" ", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    s = s.lower()
  else:
    s = re.sub(r"([.!?])", r" ", s.strip())
  return s


PAD = 0
UNK = 1
SOS = 2
EOS = 3

class Vocab:
  def __init__(self, name):
    self.name = name
    self.word2idx = {"PAD":0, "UNK":1, "SOS":2, "EOS":3}
    self.idx2word = {0:"PAD", 1:"UNK", 2:"SOS", 3:"EOS"}
    self.num = 4

  def word2index(self, word):
    return self.word2idx.get(word, UNK)

  def addwords(self, sentence):
    for word in sentence:
```

```
        if word not in self.word2idx:
          self.word2idx[word] = self.num
          self.idx2word[self.num] = word
          self.num += 1

    def get_vocab(self):
      return [key for key in self.word2idx]



def build_vocab(file_path, lang):
    vocab = Vocab(lang)
    with open(file_path) as f:
        lines = f.readlines()
        for line in lines :
            line = string_norm(line,lang)
            if lang == 'en': vocab.addwords(word_tokenize(line))
            else: vocab.addwords(tokenizer.tokenize(string_norm(line, lang)))
    return vocab


envoc = build_vocab(dev_en_path, 'en')
tevoc = build_vocab(dev_te_path, 'te')
```

Creating Embeddings for the Vocabularies

```
#using glove to make en embeddings
glove = torchtext.vocab.GloVe(name = '6B', dim = 100)

en_embed = glove.get_vecs_by_tokens(envoc.get_vocab())
# randomly embed the special symbols
for i in range(4):
  en_embed[i] = torch.randn(100)

# making te embeddings using IndicBERT
voc2emb = model.get_input_embeddings()
te_tokens = tokenizer.convert_tokens_to_ids(tevoc.get_vocab())
te_embed = voc2emb(torch.tensor(te_tokens)).detach()
```

Dataset & Dataloader

```
class TransData(Dataset):
  def __init__(self, en_path, te_path, envoc, tevoc, mlen = 128):
    self.en_path = en_path
    self.te_path = te_path
    self.envoc = envoc
    self.tevoc = tevoc
    self.mlen = mlen

  def __len__(self):
    return len(open(self.en_path).readlines())
```

```
  def __getitem__(self, idx):
    x = [string_norm(line,'en') for line in open(self.en_path).readlines()][idx]
    y = [string_norm(line,'te') for line in open(self.te_path).readlines()][idx]

    x = word_tokenize(string_norm(x, 'en'), 'en')
    y = tokenizer.tokenize(string_norm(y, 'te'), 'te')

    xlen, ylen = len(x), len(y)
    x = torch.tensor(x + [PAD for i in range(self.mlen - xlen)], dtype = torch.long)
    xmask = torch.tensor([1 for i in range(xlen)] + [0 for i in range(self.mlen - xlen)])

    y = torch.tensor(y + [PAD for i in range(self.mlen - ylen)], dtype = torch.long)
    ymask = torch.tensor([1 for i in range(ylen)] + [0 for i in range(self.mlen - ylen)])

    return x,y,len(x),len(y),xmask,ymask


train_data = TransData(dev_en_path, dev_te_path, envoc, tevoc, mlen = 256)
train_data_loader = DataLoader(train_data, batch_size = 5, shuffle = True)
```

Model

```
class Encoder(nn.Module):
  def __init__(self, in_size, emb_size, hid_size):
    super(Encoder, self).__init__()
    self.in_size = in_size
    self.emb_size = emb_size
    self.hid_size = hid_size
    self.embedding = nn.Embedding(in_size, emb_size)
    self.lstm = nn.LSTM(emb_size, hid_size, batch_first=True)

  def forward(self, input, state = None):
    return self.lstm(self.embedding(input), state)


class Decoder(nn.Module):
  def __init__(self, in_size, emb_size, hid_size, out_size):
    super(Decoder, self).__init__()
    self.in_size = in_size
    self.emb_size = emb_size
    self.hid_size = hid_size
    self.out_size = out_size
    self.embedding = nn.Embedding(in_size, emb_size)
    self.lstm = nn.LSTM(emb_size, hid_size, batch_first=True)
    self.fc = nn.Linear(hid_size, out_size)

  def forward(self, input, state=None):
    output, state = self.lstm(self.embedding(input), state)
    output = self.fc(output)
    return output, state
```

Training

```
enc = Encoder(envoc.num,100,256).to(device)
dec = Decoder(tevoc.num,128,256,tevoc.num).to(device)

enc.embedding.weight = nn.Parameter(torch.tensor(en_embed,dtype=torch.float32).to(device))
dec.embedding.weight = nn.Parameter(torch.tensor(te_embed,dtype=torch.float32).to(device))

batch_size = 20
num_epochs = 50
en_optim = torch.optim.Adam (enc.parameters(), lr = 0.00025)
de_optim = torch.optim.Adam (dec.parameters(), lr = 0.00025)


def masked_cross_entropy(logits, target, mask, target_lens):
    length = Variable(torch.LongTensor(target_lens)).cuda()
    batch,seq,cls = logits.shape
    logits_flat = logits.reshape(batch * seq,cls)
    log_probs_flat = F.log_softmax(logits_flat)
    target_flat = target.view(-1, 1)
    losses_flat = -torch.gather(log_probs_flat, dim=1, index=target_flat)
    losses = losses_flat.view(*target.size())
    losses = losses * mask.float()
    loss = losses.sum() / length.float().sum()
    return loss


def train(batch, enc, dec, en_optim, de_optim, max_length = 128):
    en_optim.zero_grad()
    de_optim.zero_grad()
    loss = 0

    x,y,x_len,y_len,x_mask,y_mask = batch
    x,y,y_mask = x.to(device),y.to(device),y_mask.to(device)


    en_output , en_state = enc(x)
    de_input = Variable(torch.LongTensor([SOS] * batch_size).reshape(batch_size,1)).to(device)
    de_outputs, de_state = Variable(torch.zeros(max_length , batch_size, dec.output_size)).to(device), e

    # Run through decoder one time step at a time
    for t in range(128):
        de_output, de_state  = dec(de_input, de_state)
        de_outputs[t] = de_output.permute(1,0,2)
        de_input = y[:,t].reshape(batch_size,-1)


    de_outputs = de_outputs.permute(1,0,2)
    loss = masked_cross_entropy(de_outputs,y,y_mask,y_len)
    loss.backward()
    en_optim.step()
    de_optim.step()

    # # Clip gradient norms
    ec = torch.nn.utils.clip_grad_norm(enc.parameters(), 100)
```

```
        dc = torch.nn.utils.clip_grad_norm(dec.parameters(), 100)

        return loss.item(), ec, dc


    # Training Loop
    for i in range(50):
        running_loss = 0
        for j, batch in enumerate(train_data_loader):
            l,ec,dc = train(batch,enc,dec,en_optim,de_optim,256)
            running_loss += l
            if j % 10 == 0:
                print(f'epoch {i+1} batch {j+1}/{len(train_data_loader)} loss {running_loss/(j+1)}')
        print(f'epoch {i+1} loss {running_loss/len(train_data_loader)}')
```