Project Code

Table of Contents

[1. Utils 2](#_Toc209925161)

[1.1. utils.py 2](#_Toc209925162)

[2. createInputFiles.py 10](#_Toc209925163)

[3. dataLoader.py 11](#_Toc209925164)

[4. Models 13](#_Toc209925165)

[4.1. encoder.py 13](#_Toc209925166)

[4.2. decoder.py 14](#_Toc209925167)

[4.3. lstmNoAttention.py 18](#_Toc209925168)

[4.4. transformerDecoder.py 22](#_Toc209925169)

[4.5. transformerDecoderAttVis.py 26](#_Toc209925170)

[5. Training and Testing Scripts 33](#_Toc209925171)

[5.1. train.py 33](#_Toc209925172)

[5.2. trainMultiGPU.py 44](#_Toc209925173)

[5.3. test.py 59](#_Toc209925174)

[6. caption.py 65](#_Toc209925175)

[7. makingGraphs.py 79](#_Toc209925176)

# 1. Utils

## 1.1. utils.py

import os

import numpy as np

import torch

import h5py

import json

# from scipy.misc import imread, imresize

from PIL import Image

from tqdm import tqdm

from collections import Counter

from random import seed, choice, sample

# The helper functions create\_input\_files, AverageMeter, clip\_gradient, save\_checkpoint,

# adjust\_learning\_rate and accuracy are adapted from the codebase of the original study (Ramos et al., 2024).

# Link to their GitHub repository: https://github.com/Leo-Thomas/ConvNeXt-for-Image-Captioning/tree/main

# The original study (Ramos et al., 2024) seem to have adapted their code from another repository (Vinodababu, 2019)

# which is a popular open source implementation of the 'Show, Attend and Tell' paper (Xu et al., 2015).

# Link to the (Vinodababu, 2019) repository: https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning

# The save\_checkpoint function is modified to include parameters relevant to my study.

# The accuracy function is modified to support multi-GPU training in my study.

def create\_input\_files(dataset, karpathy\_json\_path, image\_folder, captions\_per\_image, min\_word\_freq, output\_folder,

max\_len=100):

"""

Creates input files for training, validation, and test data.

:param dataset: name of dataset, one of 'coco', 'flickr8k', 'flickr30k'

:param karpathy\_json\_path: path of Karpathy JSON file with splits and captions

:param image\_folder: folder with downloaded images

:param captions\_per\_image: number of captions to sample per image

:param min\_word\_freq: words occuring less frequently than this threshold are binned as <unk>s

:param output\_folder: folder to save files

:param max\_len: don't sample captions longer than this length

"""

assert dataset in {'coco', 'flickr8k', 'flickr30k'} # Ensure dataset is one of the expected values

# Read Karpathy JSON

with open(karpathy\_json\_path, 'r') as j:

data = json.load(j)

# Read image paths and captions for each image

train\_image\_paths = []

train\_image\_captions = []

val\_image\_paths = []

val\_image\_captions = []

test\_image\_paths = []

test\_image\_captions = []

word\_freq = Counter()

for img in data['images']:

captions = []

for c in img['sentences']:

# Update word frequency

word\_freq.update(c['tokens'])

if len(c['tokens']) <= max\_len:

captions.append(c['tokens'])

if len(captions) == 0:

continue

path = os.path.join(image\_folder, img['filepath'], img['filename']) if dataset == 'coco' else os.path.join(

image\_folder, img['filename'])

if img['split'] in {'train', 'restval'}:

train\_image\_paths.append(path)

train\_image\_captions.append(captions)

elif img['split'] in {'val'}:

val\_image\_paths.append(path)

val\_image\_captions.append(captions)

elif img['split'] in {'test'}:

test\_image\_paths.append(path)

test\_image\_captions.append(captions)

# Sanity check

assert len(train\_image\_paths) == len(train\_image\_captions)

assert len(val\_image\_paths) == len(val\_image\_captions)

assert len(test\_image\_paths) == len(test\_image\_captions)

# Create word map (A dictionary that maps each word to a unique index)

words = [w for w in word\_freq.keys() if word\_freq[w] > min\_word\_freq]

word\_map = {k: v + 1 for v, k in enumerate(words)}

word\_map['<unk>'] = len(word\_map) + 1

word\_map['<start>'] = len(word\_map) + 1

word\_map['<end>'] = len(word\_map) + 1

word\_map['<pad>'] = 0

# Create a base/root name for all output files

base\_filename = dataset + '\_' + str(captions\_per\_image) + '\_cap\_per\_img\_' + str(min\_word\_freq) + '\_min\_word\_freq'

# Save word map to a JSON

with open(os.path.join(output\_folder, 'WORDMAP\_' + base\_filename + '.json'), 'w') as j:

json.dump(word\_map, j)

# Sample captions for each image, save images to HDF5 file, and captions and their lengths to JSON files

seed(123)

for impaths, imcaps, split in [(train\_image\_paths, train\_image\_captions, 'TRAIN'),

(val\_image\_paths, val\_image\_captions, 'VAL'),

(test\_image\_paths, test\_image\_captions, 'TEST')]:

with h5py.File(os.path.join(output\_folder, split + '\_IMAGES\_' + base\_filename + '.hdf5'), 'a') as h: # This opens an HDF5 file for storing images in the current split (train/val/test).

# Make a note of the number of captions we are sampling per image

h.attrs['captions\_per\_image'] = captions\_per\_image

# Create dataset inside HDF5 file to store images (The images dataset is created to store the images as 3x256x256 arrays)

images = h.create\_dataset('images', (len(impaths), 3, 256, 256), dtype='uint8')

print("\nReading %s images and captions, storing to file...\n" % split)

enc\_captions = []

caplens = []

for i, path in enumerate(tqdm(impaths)): # tqdm(impaths) is a progress bar that shows how much of the list has been processed

# Sample captions

if len(imcaps[i]) < captions\_per\_image:

captions = imcaps[i] + [choice(imcaps[i]) for \_ in range(captions\_per\_image - len(imcaps[i]))] # If the image has fewer captions than needed, the code will randomly duplicate captions from the existing ones

else:

captions = sample(imcaps[i], k=captions\_per\_image) # If the image has enough captions (5 or greater), it will randomly sample the required number of captions

# Sanity check

assert len(captions) == captions\_per\_image

img = Image.open(impaths[i])

if img.mode != 'RGB':

img = img.convert('RGB')

img = img.resize((256, 256), Image.BICUBIC)

img = np.array(img)

if len(img.shape) == 2:

img = img[:, :, np.newaxis]

img = np.concatenate([img, img, img], axis=2)

img = img.transpose(2, 0, 1) # Convert to (C, H, W) format for PyTorch

assert img.shape == (3, 256, 256)

assert np.max(img) <= 255

# Save image to HDF5 file

images[i] = img

for j, c in enumerate(captions):

# Encode captions

enc\_c = [word\_map['<start>']] + [word\_map.get(word, word\_map['<unk>']) for word in c] + [

word\_map['<end>']] + [word\_map['<pad>']] \* (max\_len - len(c))

# Find caption lengths

c\_len = len(c) + 2

enc\_captions.append(enc\_c)

caplens.append(c\_len)

# Sanity check

assert images.shape[0] \* captions\_per\_image == len(enc\_captions) == len(caplens)

# Save encoded captions and their lengths to JSON files

with open(os.path.join(output\_folder, split + '\_CAPTIONS\_' + base\_filename + '.json'), 'w') as j:

json.dump(enc\_captions, j)

with open(os.path.join(output\_folder, split + '\_CAPLENS\_' + base\_filename + '.json'), 'w') as j:

json.dump(caplens, j)

class AverageMeter(object):

"""

Keeps track of most recent, average, sum, and count of a metric.

"""

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

self.reset()

def reset(self):

self.val = 0

self.avg = 0

self.sum = 0

self.count = 0

def update(self, val, n=1):

self.val = val

self.sum += val \* n

self.count += n

self.avg = self.sum / self.count

def clip\_gradient(optimizer, gradClip):

"""

Clips gradients computed during backpropagation to avoid explosion of gradients.

:param optimizer: optimizer with the gradients to be clipped

:param grad\_clip: clip value

"""

for group in optimizer.param\_groups:

for param in group['params']:

if param.grad is not None:

param.grad.data.clamp\_(-gradClip, gradClip)

def save\_checkpoint(dataName, epoch, epochsSinceImprovement, encoderSaved, decoderSaved, encoderOptimizer, decoderOptimizer,

bleu4, isBest, results, lstmDecoder, startingLayer, encoderLr, pretrainedEmbeddingsName):

"""

Saves model checkpoint.

:param data\_name: base name of processed dataset

:param epoch: epoch number

:param epochs\_since\_improvement: number of epochs since last improvement in BLEU-4 score

:param encoder: encoder model

:param decoder: decoder model

:param encoder\_optimizer: optimizer to update encoder's weights, if fine-tuning

:param decoder\_optimizer: optimizer to update decoder's weights

:param bleu4: validation BLEU-4 score for this epoch

:param is\_best: is this checkpoint the best so far?

"""

state = {'epoch': epoch,

'epochsSinceImprovement': epochsSinceImprovement,

'bleu-4': bleu4,

'encoder': encoderSaved,

'decoder': decoderSaved,

'encoderOptimizer': encoderOptimizer.state\_dict() if encoderOptimizer else None,

'decoderOptimizer': decoderOptimizer.state\_dict(),

'results': results}

if lstmDecoder is True:

filename = 'checkpoint\_LSTM\_Finetuning' + str(startingLayer) + '\_' + str(encoderLr) + '\_' + dataName + '.pth.tar'

else:

filename = 'checkpoint\_Transformer\_Finetuning' + str(startingLayer) + '\_' + str(encoderLr) + '\_' + pretrainedEmbeddingsName + '\_' + dataName + '.pth.tar'

torch.save(state, filename)

# If this checkpoint is the best so far, store a copy so it doesn't get overwritten by a worse checkpoint

if isBest:

torch.save(state, 'BEST\_' + filename)

def adjust\_learning\_rate(optimizer, shrink\_factor):

"""

Shrinks learning rate by a specified factor.

:param optimizer: optimizer whose learning rate must be shrunk.

:param shrink\_factor: factor in interval (0, 1) to multiply learning rate with.

"""

print("\nDECAYING learning rate.")

for param\_group in optimizer.param\_groups:

param\_group['lr'] = param\_group['lr'] \* shrink\_factor

print("The new learning rate is %f\n" % (optimizer.param\_groups[0]['lr'],))

def accuracy(scores, targets, k, gpu):

"""

Computes top-k accuracy, from predicted and true labels.

:param scores: scores from the model

:param targets: true labels

:param k: k in top-k accuracy

:return: top-k accuracy

"""

batch\_size = targets.size(0)

\_, ind = scores.topk(k, 1, True, True)

correct = ind.eq(targets.view(-1, 1).expand\_as(ind))

correct\_total = correct.view(-1).float().sum() # 0D tensor

# return correct\_total.item() \* (100.0 / batch\_size)

if gpu == 'multi':

return correct\_total.item(), batch\_size

elif gpu == 'single':

return correct\_total.item() \* (100.0 / batch\_size)

# The preprocessDecoderOutputForMetrics is contribution of my study. It is used the align the predicted logits,

# generated sequences and ground truth captions in the case of forward without teacher forcing to compute the

# evaluation metrics.

def preprocessDecoderOutputForMetrics(predictions, sequences, encodedCaptions, end\_token\_idx, pad\_token\_idx, maxDecodeLen):

batchSize = predictions.size(0)

allFilteredPredictedLogitsList = []

allFilteredTargetIdsList = []

totalValidTokenCount = 0

actualDecodeLengths = []

for i in range(batchSize):

currentDecodeLength = 0

if (sequences[i] == end\_token\_idx).any():

endIndex = (sequences[i] == end\_token\_idx).nonzero(as\_tuple=True)[0][0].item()

currentDecodeLength = endIndex + 1

else:

currentDecodeLength = maxDecodeLen

actualDecodeLengths.append(currentDecodeLength)

predictedLogitsSliced = predictions[i, :currentDecodeLength, :]

groundTruthIdsSliced = encodedCaptions[i, 1:1 + currentDecodeLength]

nonPaddingMask = (groundTruthIdsSliced != pad\_token\_idx)

predictedLogitsFiltered = predictedLogitsSliced[nonPaddingMask]

groundTruthIdsFiltered = groundTruthIdsSliced[nonPaddingMask]

numValidTokensInSequence = groundTruthIdsFiltered.numel()

if numValidTokensInSequence == 0:

continue

allFilteredPredictedLogitsList.append(predictedLogitsFiltered)

allFilteredTargetIdsList.append(groundTruthIdsFiltered)

totalValidTokenCount += numValidTokensInSequence

# Concatenate all filtered tensors to get the final flattened output

finalFilteredPredictedLogits = torch.cat(allFilteredPredictedLogitsList, dim=0) # (N\_total\_valid, vocab\_size)

finalFilteredTargetIds = torch.cat(allFilteredTargetIdsList, dim=0) # (N\_total\_valid,)

return finalFilteredPredictedLogits, finalFilteredTargetIds, totalValidTokenCount, actualDecodeLengths

# 2. createInputFiles.py

from utils.utils import create\_input\_files

# This script uses create\_input\_files function from utils/utils.py to process the dataset and generate the

# input files for training, validation, and testing. It is adapted from the original codebase of the

# study (Ramos et al., 2024).

create\_input\_files(dataset='coco',

karpathy\_json\_path='cocoDataset/caption\_datasets/dataset\_coco.json',

image\_folder='cocoDataset/trainval2014',

captions\_per\_image=5,

min\_word\_freq=5,

output\_folder='cocoDataset/inputFiles',

max\_len=50)

# 3. dataLoader.py

import h5py

import json

import torch

from torch.utils.data import Dataset

import os

# This class to load images, caption and their lengths is adapted from the codebase of the original study (Ramos et al., 2024).

# Link to their GitHub repository: https://github.com/Leo-Thomas/ConvNeXt-for-Image-Captioning/tree/main

# The original study (Ramos et al., 2024) seem to have adapted their code from another repository (Vinodababu, 2019)

# which is a popular open source implementation of the 'Show, Attend and Tell' paper (Xu et al., 2015).

# Link to the (Vinodababu, 2019) repository: https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning

# The original class is modified to support multiple workers, lazy loading of images to avoid OOM issues and faster loading

# which is a contribution of this study.

class CaptionDataset(Dataset):

def \_\_init\_\_(self, dataFolder, dataName, split, transform=None):

self.split = split

assert self.split in {'TRAIN', 'VAL', 'TEST'}

self.dataFolder = dataFolder

self.dataName = dataName

# Store path instead of opening hdf5

self.h5\_path = os.path.join(dataFolder, self.split + '\_IMAGES\_' + dataName + '.hdf5')

self.h = None # lazy open

# Load captions fully into memory

with open(os.path.join(dataFolder, self.split + '\_CAPTIONS\_' + dataName + '.json'), 'r') as j:

self.captions = json.load(j)

with open(os.path.join(dataFolder, self.split + '\_CAPLENS\_' + dataName + '.json'), 'r') as j:

self.caplens = json.load(j)

# Load captions\_per\_image from file attribute

with h5py.File(self.h5\_path, 'r') as h:

self.cpi = h.attrs['captions\_per\_image']

self.dataset\_len = len(h['images'])

self.transform = transform

self.dataset\_size = len(self.captions)

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

if self.h is None:

self.h = h5py.File(self.h5\_path, 'r')

self.imgs = self.h['images']

img = torch.FloatTensor(self.imgs[i // self.cpi] / 255.)

if self.transform is not None:

img = self.transform(img)

caption = torch.LongTensor(self.captions[i])

caplen = torch.LongTensor([self.caplens[i]])

if self.split == 'TRAIN':

return img, caption, caplen

else:

all\_captions = torch.LongTensor(

self.captions[((i // self.cpi) \* self.cpi):(((i // self.cpi) \* self.cpi) + self.cpi)])

return img, caption, caplen, all\_captions

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

return self.dataset\_size

# 4. Models

## 4.1. encoder.py

import torch

from torch import nn

import torchvision

from torchvision.models import ConvNeXt\_Base\_Weights

import torch.nn.functional as F

# This ConvNeXt based encoder class is adapted from the codebase of the original study (Ramos et al., 2024).

# Link to their GitHub repository: https://github.com/Leo-Thomas/ConvNeXt-for-Image-Captioning/tree/main

# The original study (Ramos et al., 2024) seem to have adapted their code from another repository (Vinodababu, 2019)

# which is a popular open source implementation of the 'Show, Attend and Tell' paper (Xu et al., 2015).

# Link to the (Vinodababu, 2019) repository: https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning

class Encoder(nn.Module):

def \_\_init\_\_(self, encoded\_image\_size=7):

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

self.enc\_image\_size = encoded\_image\_size

convnext = torchvision.models.convnext\_base(weights=ConvNeXt\_Base\_Weights.IMAGENET1K\_V1)

self.convnext = convnext.features

self.adaptive\_pool = nn.AdaptiveAvgPool2d((encoded\_image\_size, encoded\_image\_size))

self.fine\_tune()

def forward(self, images):

out = self.convnext(images) # (batch\_size, 768, image\_size/32, image\_size/32)

out = self.adaptive\_pool(out) # (batch\_size, 768, encoded\_image\_size, encoded\_image\_size)

out = out.permute(0, 2, 3, 1) # (batch\_size, encoded\_image\_size, encoded\_image\_size, 768)

return out

def fine\_tune(self, fine\_tune=True, startingLayer=7): # A starting layer parameter is added to allow fine-tuning

for p in self.convnext.parameters(): # from specific layers in this stidy

p.requires\_grad = False

for c in list(self.convnext.children())[startingLayer:]:

for p in c.parameters():

p.requires\_grad = fine\_tune

## 4.2. decoder.py

import torch

from torch import nn

import torchvision

from torchvision.models import ConvNeXt\_Base\_Weights

import torch.nn.functional as F

# This LSTM + Attention based decoder class is adapted from the codebase of the original study (Ramos et al., 2024).

# Link to their GitHub repository: https://github.com/Leo-Thomas/ConvNeXt-for-Image-Captioning/tree/main

# The original study (Ramos et al., 2024) seem to have adapted their code from another repository (Vinodababu, 2019)

# which is a popular open source implementation of the 'Show, Attend and Tell' paper (Xu et al., 2015).

# Link to the (Vinodababu, 2019) repository: https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning

# This includes the Attention class, the DecoderWithAttention class with all its methods except the

# forwardWithoutTeacherForcing method which is a contribution of my study.

class Attention(nn.Module):

def \_\_init\_\_(self, encoder\_dim, decoder\_dim, attention\_dim):

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

self.encoder\_att = nn.Linear(encoder\_dim, attention\_dim) # linear layer to transform encoded image

self.decoder\_att = nn.Linear(decoder\_dim, attention\_dim) # linear layer to transform decoder's output

self.full\_att = nn.Linear(attention\_dim, 1) # linear layer to calculate values to be softmax-ed

self.relu = nn.ReLU()

self.softmax = nn.Softmax(dim=1) # softmax layer to calculate weights

def forward(self, encoder\_out, decoder\_hidden):

att1 = self.encoder\_att(encoder\_out) # (batch\_size, num\_pixels, attention\_dim)

att2 = self.decoder\_att(decoder\_hidden) # (batch\_size, attention\_dim)

att = self.full\_att(self.relu(att1 + att2.unsqueeze(1))).squeeze(2) # (batch\_size, num\_pixels)

alpha = self.softmax(att) # (batch\_size, num\_pixels)

attention\_weighted\_encoding = (encoder\_out \* alpha.unsqueeze(2)).sum(dim=1) # (batch\_size, encoder\_dim)

return attention\_weighted\_encoding, alpha

class DecoderWithAttention(nn.Module):

def \_\_init\_\_(self, attention\_dim, embed\_dim, decoder\_dim, vocab\_size, device, encoder\_dim=1024, dropout=0.5):

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

self.encoder\_dim = encoder\_dim

self.attention\_dim = attention\_dim

self.embed\_dim = embed\_dim

self.decoder\_dim = decoder\_dim

self.vocab\_size = vocab\_size

self.dropout = dropout

self.attention = Attention(encoder\_dim, decoder\_dim, attention\_dim) # attention network

self.embedding = nn.Embedding(vocab\_size, embed\_dim) # embedding layer

self.dropout = nn.Dropout(p=self.dropout)

self.decode\_step = nn.LSTMCell(embed\_dim + encoder\_dim, decoder\_dim, bias=True) # decoding LSTMCell

self.init\_h = nn.Linear(encoder\_dim, decoder\_dim) # linear layer to find initial hidden state of LSTMCell

self.init\_c = nn.Linear(encoder\_dim, decoder\_dim) # linear layer to find initial cell state of LSTMCell

self.f\_beta = nn.Linear(decoder\_dim, encoder\_dim) # linear layer to create a sigmoid-activated gate

self.sigmoid = nn.Sigmoid()

self.fc = nn.Linear(decoder\_dim, vocab\_size) # linear layer to find scores over vocabulary

self.init\_weights() # initialize some layers with the uniform distribution

self.device = device

def init\_weights(self):

self.embedding.weight.data.uniform\_(-0.1, 0.1)

self.fc.bias.data.fill\_(0)

self.fc.weight.data.uniform\_(-0.1, 0.1)

def init\_hidden\_state(self, encoder\_out):

mean\_encoder\_out = encoder\_out.mean(dim=1)

h = self.init\_h(mean\_encoder\_out) # (batch\_size, decoder\_dim)

c = self.init\_c(mean\_encoder\_out)

return h, c

def forwardWithTeacherForcing(self, encoder\_out, encoded\_captions, caption\_lengths):

batch\_size = encoder\_out.size(0)

encoder\_dim = encoder\_out.size(-1)

vocab\_size = self.vocab\_size

# Flatten image

encoder\_out = encoder\_out.view(batch\_size, -1, encoder\_dim) # (batch\_size, num\_pixels, encoder\_dim)

num\_pixels = encoder\_out.size(1)

# Sort input data by decreasing lengths; why? apparent below

caption\_lengths, sort\_ind = caption\_lengths.squeeze(1).sort(dim=0, descending=True)

encoder\_out = encoder\_out[sort\_ind]

encoded\_captions = encoded\_captions[sort\_ind]

# Embedding

embeddings = self.embedding(encoded\_captions) # (batch\_size, max\_caption\_length, embed\_dim)

# Initialize LSTM state

h, c = self.init\_hidden\_state(encoder\_out) # (batch\_size, decoder\_dim)

# We won't decode at the <end> position, since we've finished generating as soon as we generate <end>

# So, decoding lengths are actual lengths - 1

decode\_lengths = (caption\_lengths - 1).tolist()

# Create tensors to hold word predicion scores and alphas

predictions = torch.zeros(batch\_size, max(decode\_lengths), vocab\_size).to(self.device)

alphas = torch.zeros(batch\_size, max(decode\_lengths), num\_pixels).to(self.device)

# At each time-step, decode by

# attention-weighing the encoder's output based on the decoder's previous hidden state output

# then generate a new word in the decoder with the previous word and the attention weighted encoding

for t in range(max(decode\_lengths)):

batch\_size\_t = sum([l > t for l in decode\_lengths])

attention\_weighted\_encoding, alpha = self.attention(encoder\_out[:batch\_size\_t],

h[:batch\_size\_t])

gate = self.sigmoid(self.f\_beta(h[:batch\_size\_t])) # gating scalar, (batch\_size\_t, encoder\_dim)

attention\_weighted\_encoding = gate \* attention\_weighted\_encoding

h, c = self.decode\_step(

torch.cat([embeddings[:batch\_size\_t, t, :], attention\_weighted\_encoding], dim=1),

(h[:batch\_size\_t], c[:batch\_size\_t])) # (batch\_size\_t, decoder\_dim)

preds = self.fc(self.dropout(h)) # (batch\_size\_t, vocab\_size)

predictions[:batch\_size\_t, t, :] = preds

alphas[:batch\_size\_t, t, :] = alpha

return predictions, encoded\_captions, decode\_lengths, alphas, sort\_ind

# This method adapts the forward with teacher forcing method from (Vinodababu, 2019) to implement forward without

# teacher forcing. This is a contribution of my study.

def forwardWithoutTeacherForcing(self, encoder\_out, wordMap, maxDecodeLen):

batch\_size = encoder\_out.size(0)

encoder\_dim = encoder\_out.size(-1)

vocab\_size = self.vocab\_size

encoder\_out = encoder\_out.view(batch\_size, -1, encoder\_dim) # (batch\_size, num\_pixels, encoder\_dim)

num\_pixels = encoder\_out.size(1)

h, c = self.init\_hidden\_state(encoder\_out) # (batch\_size, decoder\_dim)

start\_token\_idx = wordMap['<start>']

end\_token\_idx = wordMap['<end>']

inputs = torch.LongTensor([start\_token\_idx] \* batch\_size).to(self.device)

inputs = self.embedding(inputs) # (batch\_size, embed\_dim)

predictions = torch.zeros(batch\_size, maxDecodeLen, vocab\_size).to(self.device)

alphas = torch.zeros(batch\_size, maxDecodeLen, num\_pixels).to(self.device)

sequences = torch.zeros(batch\_size, maxDecodeLen, dtype=torch.long).to(self.device) # To store predicted IDs

# Track finished sequences (those that have predicted the <end> token)

finished = torch.zeros(batch\_size, dtype=torch.bool).to(self.device) # False for all

# Decoding loop

for t in range(maxDecodeLen):

active\_indices = (~finished).nonzero(as\_tuple=False).squeeze(1) # (number\_of\_currently\_active\_sentences,)

if len(active\_indices) == 0:

break # All sequences finished early

attention\_weighted\_encoding, alpha = self.attention(encoder\_out[active\_indices], h[active\_indices])

gate = self.sigmoid(self.f\_beta(h[active\_indices]))

attention\_weighted\_encoding = gate \* attention\_weighted\_encoding

h\_new, c\_new = self.decode\_step(

torch.cat([inputs[active\_indices], attention\_weighted\_encoding], dim=1),

(h[active\_indices], c[active\_indices]))

preds = self.fc(self.dropout(h\_new)) # (active\_batch\_size, vocab\_size)

predictions[active\_indices, t, :] = preds

alphas[active\_indices, t, :] = alpha

predicted\_ids = preds.argmax(dim=1) # (active\_batch\_size) # Greedy prediction: choose the word with the highest probability

sequences[active\_indices, t] = predicted\_ids # stores the generated captions in the form of indices

finished[active\_indices] |= predicted\_ids == end\_token\_idx # Update finished flags

inputs[active\_indices] = self.embedding(predicted\_ids) # # Prepare inputs for the next step

h[active\_indices] = h\_new # Update hidden and cell states for active sequences

c[active\_indices] = c\_new

return predictions, alphas, sequences

def forward(self, teacherForcing, encoder\_out, encoded\_captions=None, caption\_lengths=None, wordMap=None, maxDecodeLen=None):

if teacherForcing is True:

predictions, encoded\_captions, decode\_lengths, alphas, sort\_ind = self.forwardWithTeacherForcing(encoder\_out, encoded\_captions, caption\_lengths)

return predictions, encoded\_captions, decode\_lengths, alphas, sort\_ind

elif teacherForcing is not True:

predictions, alphas, sequences = self.forwardWithoutTeacherForcing(encoder\_out, wordMap, maxDecodeLen)

return predictions, alphas, sequences

## 4.3. lstmNoAttention.py

import torch

from torch import nn

import torchvision

from torchvision.models import ConvNeXt\_Base\_Weights

import torch.nn.functional as F

# This LSTM without Attention based decoder class is a replication of the DecoderWithAttention class in decoder.py

# with the attention mechanism removed which is explored in this study as a baseline.

# The citations in decoder.py also apply to this class.

class DecoderWithoutAttention(nn.Module):

def \_\_init\_\_(self, embed\_dim, decoder\_dim, vocab\_size, device, encoder\_dim=1024, dropout=0.5):

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

self.encoder\_dim = encoder\_dim

self.embed\_dim = embed\_dim

self.decoder\_dim = decoder\_dim

self.vocab\_size = vocab\_size

self.dropout = dropout

self.embedding = nn.Embedding(vocab\_size, embed\_dim) # embedding layer

self.dropout = nn.Dropout(p=self.dropout)

self.decode\_step = nn.LSTMCell(embed\_dim, decoder\_dim, bias=True) # decoding LSTMCell

self.init\_h = nn.Linear(encoder\_dim, decoder\_dim) # linear layer to find initial hidden state of LSTMCell

self.init\_c = nn.Linear(encoder\_dim, decoder\_dim) # linear layer to find initial cell state of LSTMCell

self.fc = nn.Linear(decoder\_dim, vocab\_size) # linear layer to find scores over vocabulary

self.init\_weights() # initialize some layers with the uniform distribution

self.device = device

def init\_weights(self):

"""

Initializes some parameters with values from the uniform distribution, for easier convergence.

"""

self.embedding.weight.data.uniform\_(-0.1, 0.1)

self.fc.bias.data.fill\_(0)

self.fc.weight.data.uniform\_(-0.1, 0.1)

def init\_hidden\_state(self, encoder\_out):

"""

Creates the initial hidden and cell states for the decoder's LSTM based on the encoded images.

:param encoder\_out: encoded images, a tensor of dimension (batch\_size, num\_pixels, encoder\_dim)

:return: hidden state, cell state

"""

mean\_encoder\_out = encoder\_out.mean(dim=1)

h = self.init\_h(mean\_encoder\_out) # (batch\_size, decoder\_dim)

c = self.init\_c(mean\_encoder\_out)

return h, c

def forwardWithTeacherForcing(self, encoder\_out, encoded\_captions, caption\_lengths):

batch\_size = encoder\_out.size(0)

encoder\_dim = encoder\_out.size(-1)

vocab\_size = self.vocab\_size

# Flatten image

encoder\_out = encoder\_out.view(batch\_size, -1, encoder\_dim) # (batch\_size, num\_pixels, encoder\_dim)

num\_pixels = encoder\_out.size(1)

# Sort input data by decreasing lengths; why? apparent below

caption\_lengths, sort\_ind = caption\_lengths.squeeze(1).sort(dim=0, descending=True)

encoder\_out = encoder\_out[sort\_ind]

encoded\_captions = encoded\_captions[sort\_ind]

# Embedding

embeddings = self.embedding(encoded\_captions) # (batch\_size, max\_caption\_length, embed\_dim)

# Initialize LSTM state

h, c = self.init\_hidden\_state(encoder\_out) # (batch\_size, decoder\_dim)

# We won't decode at the <end> position, since we've finished generating as soon as we generate <end>

# So, decoding lengths are actual lengths - 1

decode\_lengths = (caption\_lengths - 1).tolist()

# Create tensors to hold word predicion scores and alphas

predictions = torch.zeros(batch\_size, max(decode\_lengths), vocab\_size).to(self.device)

for t in range(max(decode\_lengths)):

batch\_size\_t = sum([l > t for l in decode\_lengths])

h, c = self.decode\_step(

embeddings[:batch\_size\_t, t, :],

(h[:batch\_size\_t], c[:batch\_size\_t])) # (batch\_size\_t, decoder\_dim)

preds = self.fc(self.dropout(h)) # (batch\_size\_t, vocab\_size)

predictions[:batch\_size\_t, t, :] = preds

return predictions, encoded\_captions, decode\_lengths, sort\_ind

def forwardWithoutTeacherForcing(self, encoder\_out, wordMap, maxDecodeLen): # This method adapts the forward with teacher forcing method

batch\_size = encoder\_out.size(0) # from (Vinodababu, 2019) to implement forward without

encoder\_dim = encoder\_out.size(-1) # teacher forcing. This is a contribution of my study.

vocab\_size = self.vocab\_size

encoder\_out = encoder\_out.view(batch\_size, -1, encoder\_dim) # (batch\_size, num\_pixels, encoder\_dim)

h, c = self.init\_hidden\_state(encoder\_out) # (batch\_size, decoder\_dim)

start\_token\_idx = wordMap['<start>']

end\_token\_idx = wordMap['<end>']

inputs = torch.LongTensor([start\_token\_idx] \* batch\_size).to(self.device)

inputs = self.embedding(inputs) # (batch\_size, embed\_dim)

predictions = torch.zeros(batch\_size, maxDecodeLen, vocab\_size).to(self.device)

sequences = torch.zeros(batch\_size, maxDecodeLen, dtype=torch.long).to(self.device) # To store predicted IDs

# Track finished sequences (those that have predicted the <end> token)

finished = torch.zeros(batch\_size, dtype=torch.bool).to(self.device) # False for all

# Decoding loop

for t in range(maxDecodeLen):

active\_indices = (~finished).nonzero(as\_tuple=False).squeeze(1) # (number\_of\_currently\_active\_sentences,)

if len(active\_indices) == 0:

break # All sequences finished early

h\_new, c\_new = self.decode\_step(

inputs[active\_indices],

(h[active\_indices], c[active\_indices]))

preds = self.fc(self.dropout(h\_new)) # (active\_batch\_size, vocab\_size)

predictions[active\_indices, t, :] = preds

predicted\_ids = preds.argmax(dim=1) # (active\_batch\_size) # Greedy prediction: choose the word with the highest probability

sequences[active\_indices, t] = predicted\_ids # stores the generated captions in the form of indices

finished[active\_indices] |= predicted\_ids == end\_token\_idx # Update finished flags

inputs[active\_indices] = self.embedding(predicted\_ids) # # Prepare inputs for the next step

h[active\_indices] = h\_new # Update hidden and cell states for active sequences

c[active\_indices] = c\_new

return predictions, sequences

def forward(self, teacherForcing, encoder\_out, encoded\_captions=None, caption\_lengths=None, wordMap=None, maxDecodeLen=None):

if teacherForcing is True:

predictions, encoded\_captions, decode\_lengths, sort\_ind = self.forwardWithTeacherForcing(encoder\_out, encoded\_captions, caption\_lengths)

return predictions, encoded\_captions, decode\_lengths, sort\_ind

elif teacherForcing is not True:

predictions, sequences = self.forwardWithoutTeacherForcing(encoder\_out, wordMap, maxDecodeLen)

return predictions, sequences

## 4.4. transformerDecoder.py

import torch.nn as nn

import math

import torch

import gensim.downloader as api

from gensim.models import KeyedVectors

import numpy as np

import gzip

# The PositionalEncoding class is adapted from a Datacamp tutorial on how to build a Transformer

# using PyTorch (Sarkar, 2025).

# Link to tutorial: https://www.datacamp.com/tutorial/building-a-transformer-with-py-torch

class PositionalEncoding(nn.Module):

def \_\_init\_\_(self, embed\_dim, maxLen):

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

pe = torch.zeros(maxLen, embed\_dim)

position = torch.arange(0, maxLen, dtype=torch.float).unsqueeze(1)

div\_term = torch.exp(torch.arange(0, embed\_dim, 2).float() \* (-math.log(10000.0) / embed\_dim))

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

pe = pe.unsqueeze(0) # shape (1, max\_len, embed\_dim)

self.register\_buffer('pe', pe)

def forward(self, x):

# x shape: (batch\_size, seq\_len, embed\_dim)

x = x + self.pe[:, :x.size(1)]

return x

def loadPretrainedWordEmbeddings(wordMap, pretrained\_embeddings\_path, embed\_dim):

newEmbeddingMatrix = np.zeros((len(wordMap), embed\_dim))

if pretrained\_embeddings\_path == 'wordEmbeddings/word2vec-google-news-300.gz':

# This line is adapted from a GeeksForGeeks tutorial (GeeksforGeeks, 2025).

# Link to tutorial: https://www.geeksforgeeks.org/nlp/pre-trained-word-embedding-in-nlp/

pretrainedEmbeddings = KeyedVectors.load\_word2vec\_format(pretrained\_embeddings\_path, binary=True)

else:

pretrainedEmbeddings = KeyedVectors.load\_word2vec\_format(pretrained\_embeddings\_path, binary=False)

for word, idx in wordMap.items():

if word in pretrainedEmbeddings:

newEmbeddingMatrix[idx] = pretrainedEmbeddings[word]

return torch.tensor(newEmbeddingMatrix, dtype=torch.float)

# The TransformerDecoder class is a contribution of this study. The Datacamp tutorial (Sarkar, 2025)

# was used to understand the general structure of the transformer decoder whereas the TransformerDecoderLayer

# and TransformerDecoder classes from the PyTorch documentation were used to implement this class.

# 1. PyTorch. TransformerDecoderLayer - PyTorch 2.8 documentation.

# Available at: https://docs.pytorch.org/docs/stable/generated/torch.nn.TransformerDecoderLayer.html

# 2. PyTorch. TransformerDecoder - PyTorch 2.8 documentation.

# Available at: https://docs.pytorch.org/docs/stable/generated/torch.nn.TransformerDecoder.html

class TransformerDecoder(nn.Module):

def \_\_init\_\_(self, embed\_dim, decoder\_dim, vocab\_size, maxLen, device, wordMap, pretrained\_embeddings\_path, fine\_tune\_embeddings,

dropout=0.5, encoder\_dim=1024, num\_heads=8, num\_layers=6):

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

self.encoder\_dim = encoder\_dim

self.decoder\_dim = decoder\_dim

self.embed\_dim = embed\_dim

self.vocab\_size = vocab\_size

if pretrained\_embeddings\_path == 'wordEmbeddings/word2vec-google-news-300.gz':

num\_heads = 6

self.num\_heads = num\_heads

self.num\_layers = num\_layers

self.dropout = dropout

if pretrained\_embeddings\_path and wordMap:

pre\_trained\_embeddings\_tensor = loadPretrainedWordEmbeddings(wordMap, pretrained\_embeddings\_path, embed\_dim)

if pre\_trained\_embeddings\_tensor.shape[1] != embed\_dim:

print('Dimension mismatch for pre-trained embeddings')

self.embedding = nn.Embedding(vocab\_size, embed\_dim)

else:

self.embedding = nn.Embedding.from\_pretrained(pre\_trained\_embeddings\_tensor, freeze=not fine\_tune\_embeddings, padding\_idx=wordMap.get('<pad>'))

print(f"Loaded and aligned embeddings from '{pretrained\_embeddings\_path}'")

else:

print("Initializing embeddings randomly.")

self.embedding = nn.Embedding(vocab\_size, embed\_dim)

self.pos\_encoding = PositionalEncoding(embed\_dim, maxLen)

self.dropout = nn.Dropout(p=self.dropout)

decoder\_layer = nn.TransformerDecoderLayer(d\_model=embed\_dim, nhead=num\_heads, dim\_feedforward=decoder\_dim, dropout=dropout)

self.transformer\_decoder = nn.TransformerDecoder(decoder\_layer, num\_layers=num\_layers)

self.fc\_out = nn.Linear(embed\_dim, vocab\_size)

self.encoder\_proj = nn.Linear(encoder\_dim, embed\_dim) if encoder\_dim != embed\_dim else nn.Identity()

self.device = device

def forwardWithTeacherForcing(self, encoder\_out, encoded\_captions, caption\_lengths, tgt\_key\_padding\_mask):

batch\_size = encoder\_out.size(0)

encoder\_dim = encoder\_out.size(-1)

caption\_lengths = caption\_lengths.squeeze(1)

decode\_lengths = (caption\_lengths - 1).tolist()

encoder\_out = encoder\_out.view(batch\_size, -1, encoder\_dim) # (batch\_size, num\_pixels, encoder\_dim)

encoder\_out = self.encoder\_proj(encoder\_out).permute(1, 0, 2) # [num\_pixels, batch\_size, embed\_dim]

embeddings = self.embedding(encoded\_captions) # [batch\_size, max\_caption\_length, embed\_dim]

embeddings = self.pos\_encoding(self.dropout(embeddings))

tgt = embeddings.permute(1, 0, 2) # [max\_len, batch\_size, embed\_dim]

tgt\_seq\_len = tgt.size(0)

tgt\_mask = nn.Transformer.generate\_square\_subsequent\_mask(tgt\_seq\_len).to(self.device).bool() # [max\_caption\_length, max\_caption\_length]

decoder\_out = self.transformer\_decoder(tgt, encoder\_out, tgt\_mask=tgt\_mask, tgt\_key\_padding\_mask=tgt\_key\_padding\_mask) # [max\_len, batch\_size, embed\_dim]

decoder\_out = decoder\_out.permute(1, 0, 2) # [batch\_size, max\_caption\_length, embed\_dim]

predictions = self.fc\_out(decoder\_out) # [batch\_size, max\_caption\_length, vocab\_size]

return predictions, encoded\_captions, decode\_lengths

def forwardWithoutTeacherForcing(self, encoder\_out, wordMap, maxDecodeLen):

batch\_size = encoder\_out.size(0)

encoder\_dim = encoder\_out.size(-1)

encoder\_out = encoder\_out.view(batch\_size, -1, encoder\_dim) # (batch\_size, num\_pixels, encoder\_dim)

encoder\_out = self.encoder\_proj(encoder\_out).permute(1, 0, 2) # [num\_pixels, batch\_size, embed\_dim]

start\_token\_idx = wordMap['<start>']

end\_token\_idx = wordMap['<end>']

inputs = torch.full((batch\_size, 1), start\_token\_idx, dtype=torch.long, device=self.device)

predictions = torch.zeros(batch\_size, maxDecodeLen, self.vocab\_size, device=self.device)

sequences = torch.zeros(batch\_size, maxDecodeLen, dtype=torch.long, device=self.device)

finished = torch.zeros(batch\_size, dtype=torch.bool, device=self.device)

for t in range(maxDecodeLen):

active\_indices = (~finished).nonzero(as\_tuple=False).squeeze(1)

if len(active\_indices) == 0:

break

embeddings = self.embedding(inputs[active\_indices])

embeddings = self.pos\_encoding(self.dropout(embeddings))

tgt = embeddings.permute(1, 0, 2)

tgt\_seq\_len = tgt.size(0)

tgt\_mask = nn.Transformer.generate\_square\_subsequent\_mask(tgt\_seq\_len).to(self.device).bool()

decoder\_output\_sliced = self.transformer\_decoder(

tgt, # [current\_seq\_len, active\_batch\_size, embed\_dim]

encoder\_out[:, active\_indices, :], # [num\_pixels, active\_batch\_size, embed\_dim]

tgt\_mask=tgt\_mask) # Output shape: [current\_seq\_len, active\_batch\_size, embed\_dim]

last\_token\_output\_sliced = decoder\_output\_sliced[-1, :, :] # [active\_batch\_size, embed\_dim]

preds = self.fc\_out(last\_token\_output\_sliced)

predictions[active\_indices, t, :] = preds

pred\_ids = preds.argmax(dim=-1)

sequences[active\_indices, t] = pred\_ids

finished[active\_indices] |= (pred\_ids == end\_token\_idx)

new\_full\_inputs = torch.full(

(batch\_size, t + 2),

wordMap['<pad>'],

dtype=torch.long,

device=self.device)

new\_full\_inputs[:, :t+1] = inputs

new\_full\_inputs[active\_indices, t+1] = pred\_ids

inputs = new\_full\_inputs

return predictions, sequences

def forward(self, teacherForcing, encoder\_out, encoded\_captions=None, caption\_lengths=None, tgt\_key\_padding\_mask=None, wordMap=None, maxDecodeLen=None):

if teacherForcing is True:

predictions, encoded\_captions, decode\_lengths = self.forwardWithTeacherForcing(encoder\_out, encoded\_captions, caption\_lengths, tgt\_key\_padding\_mask)

return predictions, encoded\_captions, decode\_lengths

elif teacherForcing is not True:

predictions, sequences = self.forwardWithoutTeacherForcing(encoder\_out, wordMap, maxDecodeLen)

return predictions, sequences

## 4.5. transformerDecoderAttVis.py

import torch.nn as nn

import torch

import math

from typing import Optional, Tuple

import torch.nn.functional as F

# The PositionalEncoding class is adapted from a Datacamp tutorial on how to build a Transformer

# using PyTorch (Sarkar, 2025).

# Link to tutorial: https://www.datacamp.com/tutorial/building-a-transformer-with-py-torch

class PositionalEncoding(nn.Module):

def \_\_init\_\_(self, embed\_dim, maxLen):

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

pe = torch.zeros(maxLen, embed\_dim)

position = torch.arange(0, maxLen, dtype=torch.float).unsqueeze(1)

div\_term = torch.exp(torch.arange(0, embed\_dim, 2).float() \* (-math.log(10000.0) / embed\_dim))

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

pe = pe.unsqueeze(0) # shape (1, max\_len, embed\_dim)

self.register\_buffer('pe', pe)

def forward(self, x):

x = x + self.pe[:, :x.size(1)]

return x

# Helper functon for CustomTransformerDecoderLayer taken PyTorch's Transformer's official GitHub repository.

def \_get\_activation\_fn(activation):

if activation == "relu":

return F.relu

elif activation == "gelu":

return F.gelu

# The CustomTransformerDecoderLayer class is adapted from PyTorch's Transformer's official GitHub repository

# linked to its TransformerDecoderLayer documentation section.

# Link to the GitHub repository: https://github.com/pytorch/pytorch/blob/v2.8.0/torch/nn/modules/transformer.py#L966

# The forward function is modified to support capturing self-attention and cross-attention weights which are returned

# for each layer.

class CustomTransformerDecoderLayer(nn.Module):

\_\_constants\_\_ = ['batch\_first', 'norm\_first']

def \_\_init\_\_(self, d\_model, nhead, dim\_feedforward=2048, dropout= 0.1, activation="relu",

layer\_norm\_eps=1e-5, batch\_first=False, norm\_first=False,

device=None, dtype=None):

factory\_kwargs = {'device': device, 'dtype': dtype}

super().\_\_init\_\_()

self.self\_attn = nn.MultiheadAttention(d\_model, nhead, dropout=dropout, batch\_first=batch\_first, \*\*factory\_kwargs)

self.multihead\_attn = nn.MultiheadAttention(d\_model, nhead, dropout=dropout, batch\_first=batch\_first, \*\*factory\_kwargs)

self.linear1 = nn.Linear(d\_model, dim\_feedforward, \*\*factory\_kwargs)

self.dropout\_ffn = nn.Dropout(dropout)

self.linear2 = nn.Linear(dim\_feedforward, d\_model, \*\*factory\_kwargs)

self.norm1 = nn.LayerNorm(d\_model, eps=layer\_norm\_eps, \*\*factory\_kwargs)

self.norm2 = nn.LayerNorm(d\_model, eps=layer\_norm\_eps, \*\*factory\_kwargs)

self.norm3 = nn.LayerNorm(d\_model, eps=layer\_norm\_eps, \*\*factory\_kwargs)

self.dropout1 = nn.Dropout(dropout)

self.dropout2 = nn.Dropout(dropout)

self.dropout3 = nn.Dropout(dropout)

self.activation = \_get\_activation\_fn(activation)

self.norm\_first = norm\_first

self.batch\_first = batch\_first

def forward(self, tgt, memory= None, tgt\_mask= None, memory\_mask = None, tgt\_key\_padding\_mask= None, memory\_key\_padding\_mask= None, is\_causal= False, output\_attentions = False):

x = tgt;

attn\_weights\_sa = None

if self.norm\_first:

\_self\_attn\_input = self.norm1(x)

else:

\_self\_attn\_input = x

\_self\_attn\_output, attn\_weights\_sa = self.self\_attn(\_self\_attn\_input, \_self\_attn\_input, \_self\_attn\_input, attn\_mask=tgt\_mask, key\_padding\_mask=tgt\_key\_padding\_mask, is\_causal=is\_causal, need\_weights=output\_attentions, average\_attn\_weights=False)

x = x + self.dropout1(\_self\_attn\_output)

if not self.norm\_first:

x = self.norm1(x)

attn\_weights\_ca = None

if memory is not None:

if self.norm\_first:

\_cross\_attn\_input = self.norm2(x)

else:

\_cross\_attn\_input = x

\_cross\_attn\_output, attn\_weights\_ca = self.multihead\_attn(\_cross\_attn\_input, memory, memory, attn\_mask=memory\_mask, key\_padding\_mask=memory\_key\_padding\_mask, need\_weights=output\_attentions, average\_attn\_weights=False)

x = x + self.dropout2(\_cross\_attn\_output)

if not self.norm\_first:

x = self.norm2(x)

if self.norm\_first:

\_ffn\_input = self.norm3(x)

else:

\_ffn\_input = x

\_ffn\_output = self.linear2(self.dropout\_ffn(self.activation(self.linear1(\_ffn\_input))))

x = x + self.dropout3(\_ffn\_output)

if not self.norm\_first:

x = self.norm3(x)

return x, attn\_weights\_sa, attn\_weights\_ca

# The TransformerDecoderForAttentionViz class is a contribution of this study. It is adapted from the

# TransformerDecoder class defined in transformerDecoder.py however, PyTorch's default TransformerDecoderLayer

# is replaced by the CustomerTransformerDecoderLayer defined above to incorporate getting the self-attention and

# cross-attention weights from each decoder layer. The general structure is understood from the Datacamp tutorial

# (Sarkar, 2025) whereas PyTorch's Transformer's official GitHub repository linked to its TransformerDecoderLayer

# documentation section is used for implementing the CustomerTransformerDecoderLayer.

# Link to the GitHub repository: https://github.com/pytorch/pytorch/blob/v2.8.0/torch/nn/modules/transformer.py#L966

class TransformerDecoderForAttentionViz(nn.Module):

def \_\_init\_\_(self, embed\_dim, decoder\_dim, vocab\_size, maxLen, device, dropout=0.5, encoder\_dim=1024, num\_heads=8, num\_layers=6):

super().\_\_init\_\_()

self.encoder\_dim = encoder\_dim

self.decoder\_dim = decoder\_dim

self.embed\_dim = embed\_dim

self.vocab\_size = vocab\_size

self.num\_heads = num\_heads

self.num\_layers = num\_layers

self.dropout = dropout

self.embedding = nn.Embedding(vocab\_size, embed\_dim)

self.pos\_encoding = PositionalEncoding(embed\_dim, maxLen)

self.dropout = nn.Dropout(p=self.dropout)

self.decoder\_layers = nn.ModuleList([

CustomTransformerDecoderLayer(d\_model=embed\_dim, nhead=num\_heads, dim\_feedforward=decoder\_dim, dropout=dropout,

batch\_first=False)

for \_ in range(num\_layers)

])

self.fc\_out = nn.Linear(embed\_dim, vocab\_size)

self.encoder\_proj = nn.Linear(encoder\_dim, embed\_dim) if encoder\_dim != embed\_dim else nn.Identity()

self.device = device

def forwardWithTeacherForcing(self, encoder\_out, encoded\_captions, caption\_lengths, tgt\_key\_padding\_mask):

batch\_size = encoder\_out.size(0)

encoder\_dim = encoder\_out.size(-1)

caption\_lengths\_squeezed = caption\_lengths.squeeze(1)

decode\_lengths = (caption\_lengths\_squeezed - 1).tolist()

encoder\_out = encoder\_out.view(batch\_size, -1, encoder\_dim) # (batch\_size, num\_pixels, encoder\_dim)

encoder\_out = self.encoder\_proj(encoder\_out).permute(1, 0, 2) # [num\_pixels, batch\_size, embed\_dim]

embeddings = self.embedding(encoded\_captions)

embeddings = self.pos\_encoding(self.dropout(embeddings))

tgt = embeddings.permute(1, 0, 2)

tgt\_seq\_len = tgt.size(0)

tgt\_mask = nn.Transformer.generate\_square\_subsequent\_mask(tgt\_seq\_len).to(self.device).bool()

output = tgt

all\_cross\_attentions\_for\_all\_steps = []

for layer\_idx, layer in enumerate(self.decoder\_layers):

output, self\_attn\_weights, cross\_attn\_weights = layer(

output,

encoder\_out,

tgt\_mask=tgt\_mask,

tgt\_key\_padding\_mask=tgt\_key\_padding\_mask,

output\_attentions=True

)

all\_cross\_attentions\_for\_all\_steps.append(cross\_attn\_weights)

decoder\_out = output.permute(1, 0, 2) # [batch\_size, max\_caption\_length, embed\_dim]

predictions = self.fc\_out(decoder\_out)

stacked\_cross\_attentions = torch.stack(all\_cross\_attentions\_for\_all\_steps, dim=0)

alphas = stacked\_cross\_attentions.mean(dim=(0, 3))

alphas = alphas.permute(1, 0, 2)

return predictions, encoded\_captions, decode\_lengths, alphas

def forwardWithoutTeacherForcing(self, encoder\_out, wordMap, maxDecodeLen):

batch\_size = encoder\_out.size(0)

encoder\_dim = encoder\_out.size(-1)

encoder\_out = encoder\_out.view(batch\_size, -1, encoder\_dim) # (batch\_size, num\_pixels, encoder\_dim)

encoder\_out = self.encoder\_proj(encoder\_out).permute(1, 0, 2) # [num\_pixels, batch\_size, embed\_dim]

start\_token\_idx = wordMap['<start>']

end\_token\_idx = wordMap['<end>']

inputs = torch.full((batch\_size, 1), start\_token\_idx, dtype=torch.long, device=self.device)

predictions = torch.zeros(batch\_size, maxDecodeLen, self.vocab\_size, device=self.device)

sequences = torch.zeros(batch\_size, maxDecodeLen, dtype=torch.long, device=self.device)

alphas = torch.zeros(batch\_size, maxDecodeLen, encoder\_out.size(0), device=self.device)

finished = torch.zeros(batch\_size, dtype=torch.bool, device=self.device)

for t in range(maxDecodeLen):

active\_indices = (~finished).nonzero(as\_tuple=False).squeeze(1)

if len(active\_indices) == 0: break

embeddings = self.embedding(inputs[active\_indices])

embeddings = self.pos\_encoding(self.dropout(embeddings))

tgt = embeddings.permute(1, 0, 2)

tgt\_mask = nn.Transformer.generate\_square\_subsequent\_mask(tgt.size(0)).to(self.device).bool()

current\_layer\_output = tgt

all\_layer\_cross\_attentions\_for\_step = []

for layer\_idx, layer in enumerate(self.decoder\_layers):

layer\_output, self\_attn\_weights, cross\_attn\_weights = layer(

current\_layer\_output,

encoder\_out[:, active\_indices, :],

tgt\_mask=tgt\_mask,

output\_attentions=True

)

current\_layer\_output = layer\_output

all\_layer\_cross\_attentions\_for\_step.append(cross\_attn\_weights)

last\_token\_output\_sliced = current\_layer\_output[-1, :, :]

preds = self.fc\_out(last\_token\_output\_sliced)

predictions[active\_indices, t, :] = preds

pred\_ids = preds.argmax(dim=-1)

sequences[active\_indices, t] = pred\_ids

finished[active\_indices] |= (pred\_ids == end\_token\_idx)

new\_full\_inputs = torch.full( (batch\_size, t + 2), wordMap['<pad>'], dtype=torch.long, device=self.device)

new\_full\_inputs[:, :t+1] = inputs

new\_full\_inputs[active\_indices, t+1] = pred\_ids

inputs = new\_full\_inputs

stacked\_cross\_attentions = torch.stack(all\_layer\_cross\_attentions\_for\_step, dim=0)

cross\_attn\_for\_current\_token = stacked\_cross\_attentions[:, :, :, -1, :]

avg\_cross\_attention\_per\_token = cross\_attn\_for\_current\_token.mean(dim=(0, 2))

alphas[active\_indices, t, :] = avg\_cross\_attention\_per\_token

return predictions, sequences, alphas

def forward(self, teacherForcing, encoder\_out, encoded\_captions=None, caption\_lengths=None, tgt\_key\_padding\_mask=None, wordMap=None, maxDecodeLen=None):

if teacherForcing is True:

predictions, encoded\_captions, decode\_lengths, alphas = self.forwardWithTeacherForcing(encoder\_out, encoded\_captions, caption\_lengths, tgt\_key\_padding\_mask)

return predictions, encoded\_captions, decode\_lengths, alphas

elif teacherForcing is not True:

predictions, sequences, alphas = self.forwardWithoutTeacherForcing(encoder\_out, wordMap, maxDecodeLen)

return predictions, sequences, alphas

# 5. Training and Testing Scripts

## 5.1. train.py

import os

import torch

import random

import numpy as np

def set\_seed(seed=42):

random.seed(seed)

np.random.seed(seed)

torch.manual\_seed(seed)

set\_seed(42)

from torch.utils.data import DataLoader

import torch.backends.cudnn as cudnn

import torchvision.transforms as transforms

import json

import time

from torch import nn

import torch.optim as optim

from torch.nn.utils.rnn import pack\_padded\_sequence

from nltk.translate.bleu\_score import corpus\_bleu

import pandas as pd

from models.encoder import Encoder

from models.decoder import DecoderWithAttention

from models.lstmNoAttention import DecoderWithoutAttention

from models.transformerDecoder import TransformerDecoder

from dataLoader import CaptionDataset

from utils.utils import \*

import argparse

# Set device to GPU (if available) or CPU

device = torch.device("cuda")

# Data parameters

dataFolder = 'cocoDataset/inputFiles'

dataName = 'coco\_5\_cap\_per\_img\_5\_min\_word\_freq'

# Model parameters

embDim = 512 # dimension of word embeddings

attentionDim = 512 # dimension of attention linear layers

decoderDim = 512 # dimension of decoder RNN

dropout = 0.5

cudnn.benchmark = True # set to true only if inputs to model are fixed size; otherwise lot of computational overhead

maxLen = 52 # maximum length of captions (in words), used for padding

# Training parameters

startEpoch = 0

epochs = 120 # number of epochs to train for (if early stopping is not triggered)

epochsSinceImprovement = 0 # keeps track of number of epochs since there's been an improvement in validation BLEU

batchSize = 32 #32

workers = 6

# encoderLr = 1e-4 # learning rate for encoder if fine-tuning

decoderLr = 1e-4 # learning rate for decoder

gradClip = 5. # clip gradients at an absolute value of

alphaC = 1. # regularization parameter for 'doubly stochastic attention', as in the paper

bestBleu4 = 0. # BLEU-4 score right now

printFreq = 100 # print training/validation stats every \_\_ batches

fineTuneEncoder = False # fine-tune encoder

parser = argparse.ArgumentParser()

parser.add\_argument('--checkpoint', type=str, default=None, help='Path to checkpoint file')

parser.add\_argument('--lstmDecoder', action='store\_true', help='Use LSTM decoder instead of Transformer')

parser.add\_argument('--teacherForcing', action='store\_true', help='Use teacher forcing training strategy')

parser.add\_argument('--startingLayer', type=int, default=5, help='Starting layer index for encoder fine-tuning encoder')

parser.add\_argument('--encoderLr', type=float, default=1e-4, help='Learning rate for encoder if fine-tuning')

parser.add\_argument('--embeddingName', type=str, default=None, help='Pretrained embedding name from gensim')

args = parser.parse\_args()

checkpoint = args.checkpoint

lstmDecoder = args.lstmDecoder

teacherForcing = args.teacherForcing

startingLayer = args.startingLayer

encoderLr = args.encoderLr

pretrainedEmbeddingsName = args.embeddingName # word2vec-google-news-300, glove-wiki-gigaword-200

if pretrainedEmbeddingsName == 'word2vec-google-news-300':

embDim = 300

pretrainedEmbeddingsPath = 'wordEmbeddings/word2vec-google-news-300.gz'

elif pretrainedEmbeddingsName == 'glove-wiki-gigaword-200':

embDim = 200

pretrainedEmbeddingsPath = 'wordEmbeddings/glove-wiki-gigaword-200.gz'

def optimizer\_to\_device(optimizer, device):

for state in optimizer.state.values():

for k, v in state.items():

if isinstance(v, torch.Tensor):

state[k] = v.to(device)

# This main function, training with teacher forcing and validate functions have been adapted from the codebase of the original

# study (Ramos et al., 2024). Link to their GitHub repository: https://github.com/Leo-Thomas/ConvNeXt-for-Image-Captioning/tree/main

# The original study (Ramos et al., 2024) seem to have adapted their code from another repository (Vinodababu, 2019)

# which is a popular open source implementation of the 'Show, Attend and Tell' paper (Xu et al., 2015).

# Link to the (Vinodababu, 2019) repository: https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning

# Significant sections have been modified/added to these functions to handle training the Transformer decoder, fine-tuning the encoder

# and using pretrained word embeddings which are contributions of this study.

def main():

global bestBleu4, epochsSinceImprovement, checkpoint, startEpoch, fineTuneEncoder, dataName, wordMap

# Load word map

wordMapFile = os.path.join(dataFolder, 'WORDMAP\_' + dataName + '.json')

with open(wordMapFile, 'r') as j:

wordMap = json.load(j)

if checkpoint is None:

if lstmDecoder is True:

decoder = DecoderWithAttention(attention\_dim=attentionDim, embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), dropout=dropout, device=device)

# decoder = DecoderWithoutAttention(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), dropout=dropout, device=device)

else:

decoder = TransformerDecoder(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), maxLen=maxLen, dropout=dropout, device=device,

wordMap=wordMap, pretrained\_embeddings\_path=pretrainedEmbeddingsPath, fine\_tune\_embeddings=True)

decoderOptimizer = torch.optim.Adam(params=filter(lambda p: p.requires\_grad, decoder.parameters()), lr=decoderLr)

encoder = Encoder()

encoder.fine\_tune(fine\_tune=False)

if fineTuneEncoder is True:

encoderOptimizer = torch.optim.Adam(params=filter(lambda p: p.requires\_grad, encoder.parameters()), lr=encoderLr)

else:

encoderOptimizer = None

results = []

else:

if lstmDecoder is True:

decoder = DecoderWithAttention(attention\_dim=attentionDim, embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), dropout=dropout, device=device)

# decoder = DecoderWithAttention(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), dropout=dropout, device=device)

else:

decoder = TransformerDecoder(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), maxLen=maxLen, dropout=dropout, device=device,

wordMap=wordMap, pretrained\_embeddings\_path=pretrainedEmbeddingsPath, fine\_tune\_embeddings=True)

decoderOptimizer = torch.optim.Adam(params=filter(lambda p: p.requires\_grad, decoder.parameters()), lr=decoderLr)

encoder = Encoder()

checkpoint = torch.load(checkpoint, map\_location=device, weights\_only=False)

encoder.load\_state\_dict(checkpoint['encoder'])

startEpoch = checkpoint['epoch'] + 1

if startEpoch > 20:

fineTuneEncoder = True

encoder.fine\_tune(fine\_tune=fineTuneEncoder, startingLayer=startingLayer)

else:

fineTuneEncoder = False

encoder.fine\_tune(fine\_tune=fineTuneEncoder)

decoder.load\_state\_dict(checkpoint['decoder'])

decoderOptimizer.load\_state\_dict(checkpoint['decoderOptimizer'])

optimizer\_to\_device(decoderOptimizer, device)

if fineTuneEncoder is True:

encoderOptimizer = torch.optim.Adam(params=filter(lambda p: p.requires\_grad, encoder.parameters()), lr=encoderLr)

if checkpoint['encoderOptimizer'] is not None:

encoderOptimizer.load\_state\_dict(checkpoint['encoderOptimizer'])

optimizer\_to\_device(encoderOptimizer, device)

else:

encoderOptimizer = None

epochsSinceImprovement = checkpoint['epochsSinceImprovement']

bestBleu4 = checkpoint['bleu-4']

results = checkpoint['results']

decoder = decoder.to(device)

encoder = encoder.to(device)

criterion = nn.CrossEntropyLoss().to(device)

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

trainDataset = CaptionDataset(dataFolder, dataName, 'TRAIN', transform=transforms.Compose([normalize]))

trainDataLoader = DataLoader(trainDataset, batch\_size=batchSize, shuffle=True, num\_workers=workers, persistent\_workers=True, pin\_memory=True)

valDataset = CaptionDataset(dataFolder, dataName, 'VAL', transform=transforms.Compose([normalize]))

valDataLoader = DataLoader(valDataset, batch\_size=batchSize, shuffle=True, num\_workers=workers, persistent\_workers=True, pin\_memory=True)

for epoch in range(startEpoch, epochs):

if epoch == 20:

fineTuneEncoder = True

encoder.fine\_tune(fine\_tune=fineTuneEncoder, startingLayer=startingLayer)

encoderOptimizer = torch.optim.Adam(params=filter(lambda p: p.requires\_grad, encoder.parameters()), lr=encoderLr)

optimizer\_to\_device(encoderOptimizer, device)

print(f"Fine-tuning encoder from epoch 20 onwards (starting from layer {startingLayer})", flush=True)

# Decay learning rate if there is no improvement for 8 consecutive epochs, and terminate training after 20

if epochsSinceImprovement == 20:

break

if epochsSinceImprovement > 0 and epochsSinceImprovement % 8 == 0:

adjust\_learning\_rate(decoderOptimizer, 0.8)

if fineTuneEncoder:

adjust\_learning\_rate(encoderOptimizer, 0.8)

if teacherForcing is True:

trainLoss, trainTop5Acc, trainBatchTime, trainDataTime = trainWithTeacherForcing(trainDataLoader=trainDataLoader,

encoder=encoder,

decoder=decoder,

criterion=criterion,

encoderOptimizer=encoderOptimizer,

decoderOptimizer=decoderOptimizer,

epoch=epoch,

device=device)

else:

trainLoss, trainTop5Acc, trainBatchTime, trainDataTime = trainWithoutTeacherForcing(trainDataLoader=trainDataLoader,

encoder=encoder,

decoder=decoder,

criterion=criterion,

encoderOptimizer=encoderOptimizer,

decoderOptimizer=decoderOptimizer,

epoch=epoch,

device=device)

valLoss, valTop5Acc, bleu1, bleu2, bleu3, recentBleu4 = validate(valDataLoader=valDataLoader,

encoder=encoder,

decoder=decoder,

criterion=criterion,

device=device)

results.append({

'epoch': epoch,

'trainLoss': trainLoss,

'trainTop5Acc': trainTop5Acc,

'trainBatchTime': trainBatchTime,

'trainDataTime': trainDataTime,

'valLoss': valLoss,

'valTop5Acc': valTop5Acc,

'bleu1': bleu1,

'bleu2': bleu2,

'bleu3': bleu3,

'bleu4': recentBleu4

})

# Check if there was an improvement

isBest = recentBleu4 > bestBleu4

bestBleu4 = max(recentBleu4, bestBleu4)

if not isBest:

epochsSinceImprovement += 1

print("\nEpochs since last improvement: %d\n" % (epochsSinceImprovement,))

else:

epochsSinceImprovement = 0

# Save checkpoint

encoderSaved = encoder.state\_dict()

decoderSaved = decoder.state\_dict()

save\_checkpoint(dataName, epoch, epochsSinceImprovement, encoderSaved, decoderSaved, encoderOptimizer,

decoderOptimizer, recentBleu4, isBest, results, lstmDecoder, startingLayer, encoderLr,

pretrainedEmbeddingsName)

resultsDF = pd.DataFrame(results)

os.makedirs('results', exist\_ok=True)

if lstmDecoder is True:

resultsDF.to\_csv(f'results/metrics-LSTMdecoderNoAtt(trainingTF-inferenceNoTF-Finetuning{startingLayer}).csv', index=False)

else:

resultsDF.to\_csv(f'results/metrics-TransformerDecoder(trainingTF-inferenceNoTF-Finetuning{startingLayer}-{pretrainedEmbeddingsName}).csv', index=False)

def trainWithTeacherForcing(trainDataLoader, encoder, decoder, criterion, encoderOptimizer, decoderOptimizer, epoch, device):

encoder.train()

decoder.train()

batchTime = AverageMeter() # forward prop. + back prop. time

dataTime = AverageMeter() # data loading time

losses = AverageMeter() # loss (per word decoded)

top5accs = AverageMeter() # top5 accuracy

start = time.time()

for i, (imgs, caps, caplens) in enumerate(trainDataLoader):

dataTime.update(time.time() - start)

if (i % 100 == 0):

print(f"TF, Epoch {epoch}, Batch {i + 1}/{len(trainDataLoader)}", flush=True)

imgs = imgs.to(device)

caps = caps.to(device)

caplens = caplens.to(device)

imgs = encoder(imgs)

if lstmDecoder is True:

# scores, capsSorted, decodeLengths, alphas, sortInd = decoder(teacherForcing=True, encoder\_out=imgs, encoded\_captions=caps, caption\_lengths=caplens)

scores, capsSorted, decodeLengths, sortInd = decoder(teacherForcing=True, encoder\_out=imgs, encoded\_captions=caps, caption\_lengths=caplens)

targets = capsSorted[:, 1:] # still in the form of indices

scores = pack\_padded\_sequence(scores, decodeLengths, batch\_first=True).data # scores are logits

targets = pack\_padded\_sequence(targets, decodeLengths, batch\_first=True).data

loss = criterion(scores, targets)

# loss += alphaC \* ((1. - alphas.sum(dim=1)) \*\* 2).mean()

else:

tgt\_key\_padding\_mask = (caps == wordMap['<pad>'])

scores, capsSorted, decodeLengths = decoder(teacherForcing=True, encoder\_out=imgs, encoded\_captions=caps, caption\_lengths=caplens, tgt\_key\_padding\_mask=tgt\_key\_padding\_mask)

targets = capsSorted[:, 1:]

scores = pack\_padded\_sequence(scores, decodeLengths, batch\_first=True, enforce\_sorted=False).data # scores are logits

targets = pack\_padded\_sequence(targets, decodeLengths, batch\_first=True, enforce\_sorted=False).data

loss = criterion(scores, targets)

if encoderOptimizer is not None:

encoderOptimizer.zero\_grad()

decoderOptimizer.zero\_grad()

loss.backward()

# Clip gradients

if gradClip is not None:

clip\_gradient(decoderOptimizer, gradClip)

if encoderOptimizer is not None:

clip\_gradient(encoderOptimizer, gradClip)

if encoderOptimizer is not None:

encoderOptimizer.step()

decoderOptimizer.step()

top5 = accuracy(scores, targets, 5, 'single')

# Keep track of metrics

losses.update(loss.item(), sum(decodeLengths))

top5accs.update(top5, sum(decodeLengths))

batchTime.update(time.time() - start)

start = time.time()

print(f"TF, Epoch {epoch}: Training Loss = {losses.avg:.4f}, Top-5 Accuracy = {top5accs.avg:.4f}", flush=True)

return losses.avg, top5accs.avg, batchTime.avg, dataTime.avg

# The trainWithoutTeacherForcing method calls the corresponding non-teacher forcing forward method of each decoder and

# aligns their outputs in the preprocessDecoderOutputForMetrics function for the evaluation metrics. This is a contribution of this study.

def trainWithoutTeacherForcing(trainDataLoader, encoder, decoder, criterion, encoderOptimizer, decoderOptimizer, epoch, device):

encoder.train()

decoder.train()

batchTime = AverageMeter()

dataTime = AverageMeter()

losses = AverageMeter()

top5accs = AverageMeter()

start = time.time()

for i, (imgs, caps, caplens) in enumerate(trainDataLoader):

dataTime.update(time.time() - start)

if (i % 100 == 0):

print(f"No TF, Epoch {epoch}, Batch {i + 1}/{len(trainDataLoader)}", flush=True)

imgs = imgs.to(device)

caps = caps.to(device)

caplens = caplens.to(device)

imgs = encoder(imgs)

if lstmDecoder is True:

scores, alphas, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

# scores, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

scoresUpdated, targetsUpdated, totalTokensEvaluated, actualDecodeLengths = preprocessDecoderOutputForMetrics(scores, sequences, caps, wordMap['<end>'], wordMap['<pad>'], 51)

loss = criterion(scoresUpdated, targetsUpdated)

loss += alphaC \* ((1. - alphas.sum(dim=1)) \*\* 2).mean()

else:

scores, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

scoresUpdated, targetsUpdated, totalTokensEvaluated, actualDecodeLengths = preprocessDecoderOutputForMetrics(scores, sequences, caps, wordMap['<end>'], wordMap['<pad>'], 51)

loss = criterion(scoresUpdated, targetsUpdated)

if encoderOptimizer is not None:

encoderOptimizer.zero\_grad()

decoderOptimizer.zero\_grad()

loss.backward()

if gradClip is not None:

clip\_gradient(decoderOptimizer, gradClip)

if encoderOptimizer is not None:

clip\_gradient(encoderOptimizer, gradClip)

if encoderOptimizer is not None:

encoderOptimizer.step()

decoderOptimizer.step()

top5 = accuracy(scoresUpdated, targetsUpdated, 5, 'single')

losses.update(loss.item(), totalTokensEvaluated)

top5accs.update(top5, totalTokensEvaluated)

batchTime.update(time.time() - start)

start = time.time()

print(f"No TF, Epoch {epoch}: Training Loss = {losses.avg:.4f}, Top-5 Accuracy = {top5accs.avg:.4f}", flush=True)

return losses.avg, top5accs.avg, batchTime.avg, dataTime.avg

# The validate method calls the corresponding non-teacher forcing forward method of each decoder and aligns their outputs in the

# preprocessDecoderOutputForMetrics function for the evaluation metrics. It also calculates all four BLEU scores.

# These are contributions of this study.

def validate(valDataLoader, encoder, decoder, criterion, device):

decoder.eval()

if encoder is not None:

encoder.eval()

batchTime = AverageMeter()

losses = AverageMeter()

top5accs = AverageMeter()

start = time.time()

references = list() # references (true captions) for calculating BLEU-4 score

hypotheses = list() # hypotheses (predictions)

with torch.no\_grad():

for i, (imgs, caps, caplens, allcaps) in enumerate(valDataLoader):

if (i % 100 == 0):

print(f"No TF, Validation Batch {i + 1}/{len(valDataLoader)}", flush=True)

imgs = imgs.to(device)

caps = caps.to(device)

caplens = caplens.to(device)

if encoder is not None:

imgs = encoder(imgs)

if lstmDecoder is True:

scores, alphas, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

# scores, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

scoresUpdated, targetsUpdated, totalTokensEvaluated, actualDecodeLengths = preprocessDecoderOutputForMetrics(scores, sequences, caps, wordMap['<end>'], wordMap['<pad>'], 51)

loss = criterion(scoresUpdated, targetsUpdated)

# Add doubly stochastic attention regularization

loss += alphaC \* ((1. - alphas.sum(dim=1)) \*\* 2).mean()

else:

scores, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

scoresUpdated, targetsUpdated, totalTokensEvaluated, actualDecodeLengths = preprocessDecoderOutputForMetrics(scores, sequences, caps, wordMap['<end>'], wordMap['<pad>'], 51)

loss = criterion(scoresUpdated, targetsUpdated)

top5 = accuracy(scoresUpdated, targetsUpdated, 5, 'single')

losses.update(loss.item(), totalTokensEvaluated)

top5accs.update(top5, totalTokensEvaluated)

batchTime.update(time.time() - start)

start = time.time()

# References

allcaps = allcaps.to(device)

for j in range(allcaps.shape[0]): # Iterate through each image in the batch

imgCaps = allcaps[j].tolist() # This would be a list of lists, where each inner list is a reference

imgCaptions = []

for c\_list in imgCaps: # Iterate through each reference caption for the current image

filtered\_caption = [w for w in c\_list if w not in {wordMap['<start>'], wordMap['<pad>']}]

imgCaptions.append(filtered\_caption)

references.append(imgCaptions)

# Hypotheses

batchHypotheses = [] # Create a temporary list to hold all captions for this batch

for j, p\_seq\_tensor in enumerate(sequences):

truncated\_predicted\_list = p\_seq\_tensor[:actualDecodeLengths[j]].tolist()

batchHypotheses.append(truncated\_predicted\_list)

hypotheses.extend(batchHypotheses)

assert len(references) == len(hypotheses)

# bleu4 = corpus\_bleu(references, hypotheses)

bleu1 = corpus\_bleu(references, hypotheses, weights=(1.0, 0.0, 0.0, 0.0))

bleu2 = corpus\_bleu(references, hypotheses, weights=(0.5, 0.5, 0.0, 0.0))

bleu3 = corpus\_bleu(references, hypotheses, weights=(0.33, 0.33, 0.33, 0.0))

bleu4 = corpus\_bleu(references, hypotheses, weights=(0.25, 0.25, 0.25, 0.25))

print(f"No TF, Validation Loss = {losses.avg:.4f}, Top-5 Accuracy = {top5accs.avg:.4f}, Bleu-1 = {bleu1:.4f}, Bleu-2 = {bleu2:.4f}, Bleu-3 = {bleu3:.4f}, Bleu-4 = {bleu4:.4f}", flush=True)

return losses.avg, top5accs.avg, bleu1, bleu2, bleu3, bleu4

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

main()

## 5.2. trainMultiGPU.py

import os

import torch

import random

import numpy as np

def set\_seed(seed):

rank = dist.get\_rank() if dist.is\_initialized() else 0

seed = seed + rank

random.seed(seed)

np.random.seed(seed)

torch.manual\_seed(seed)

import torch.distributed as dist

from torch.nn.parallel import DistributedDataParallel as DDP

import torch.multiprocessing as mp

from torch.utils.data import DataLoader

import torch.backends.cudnn as cudnn

import torchvision.transforms as transforms

import json

import time

from torch import nn

import torch.optim as optim

from torch.nn.utils.rnn import pack\_padded\_sequence

from nltk.translate.bleu\_score import corpus\_bleu

import pandas as pd

from models.encoder import Encoder

from models.decoder import DecoderWithAttention

from models.lstmNoAttention import DecoderWithoutAttention

from models.transformerDecoder import TransformerDecoder

from models.transformerDecoderAttVis import TransformerDecoderForAttentionViz

from dataLoader import CaptionDataset

from utils.utils import \*

import pickle

import argparse

# Data parameters

dataFolder = 'cocoDataset/inputFiles'

dataName = 'coco\_5\_cap\_per\_img\_5\_min\_word\_freq'

# Model parameters

embDim = 512 # dimension of word embeddings

attentionDim = 512 # dimension of attention linear layers

decoderDim = 512 # dimension of decoder RNN

dropout = 0.5

cudnn.benchmark = True # set to true only if inputs to model are fixed size; otherwise lot of computational overhead

# cudnn.deterministic = True # for reproducibility

maxLen = 52 # maximum length of captions (in words), used for padding

# Training parameters

startEpoch = 0

epochs = 120 # number of epochs to train for (if early stopping is not triggered)

epochsSinceImprovement = 0 # keeps track of number of epochs since there's been an improvement in validation BLEU

batchSize = 32

workers = 6

# encoderLr = 1e-4 # learning rate for encoder if fine-tuning

decoderLr = 1e-4 # learning rate for decoder

gradClip = 5. # clip gradients at an absolute value of

alphaC = 1. # regularization parameter for 'doubly stochastic attention', as in the paper

bestBleu4 = 0. # BLEU-4 score right now

printFreq = 100 # print training/validation stats every \_\_ batches

fineTuneEncoder = False # fine-tune encoder

parser = argparse.ArgumentParser()

parser.add\_argument('--checkpoint', type=str, default=None, help='Path to checkpoint file')

parser.add\_argument('--lstmDecoder', action='store\_true', help='Use LSTM decoder instead of Transformer')

parser.add\_argument('--port', type=str, default='29500', help='Master port for distributed training')

parser.add\_argument('--teacherForcing', action='store\_true', help='Use teacher forcing training strategy')

parser.add\_argument('--startingLayer', type=int, default=7, help='Starting layer index for encoder fine-tuning encoder')

parser.add\_argument('--encoderLr', type=float, default=1e-4, help='Learning rate for encoder if fine-tuning')

parser.add\_argument('--embeddingName', type=str, default=None, help='Pretrained embedding name from gensim')

args = parser.parse\_args()

checkpoint = args.checkpoint

lstmDecoder = args.lstmDecoder

port = args.port

teacherForcing = args.teacherForcing

startingLayer = args.startingLayer

encoderLr = args.encoderLr

pretrainedEmbeddingsName = args.embeddingName

if pretrainedEmbeddingsName == 'word2vec-google-news-300':

embDim = 300

pretrainedEmbeddingsPath = 'wordEmbeddings/word2vec-google-news-300.gz'

elif pretrainedEmbeddingsName == 'glove-wiki-gigaword-200':

embDim = 200

pretrainedEmbeddingsPath = 'wordEmbeddings/glove-wiki-gigaword-200.gz'

else:

pretrainedEmbeddingsPath = None

def optimizer\_to\_device(optimizer, device):

for state in optimizer.state.values():

for k, v in state.items():

if isinstance(v, torch.Tensor):

state[k] = v.to(device)

def reduceLossAndTokens(loss, batchTokenCount, device):

localTokenCount = batchTokenCount

localTokenLossSum = loss.item() \* localTokenCount

totalTokenLossSum = torch.tensor(localTokenLossSum, device=device)

totalTokenCount = torch.tensor(localTokenCount, device=device)

dist.all\_reduce(totalTokenLossSum, op=dist.ReduceOp.SUM)

dist.all\_reduce(totalTokenCount, op=dist.ReduceOp.SUM)

globalLoss = (totalTokenLossSum / totalTokenCount).item()

totalTokens = totalTokenCount.item()

return globalLoss, totalTokens

def gather\_all\_data(data, world\_size, device):

data\_bytes = pickle.dumps(data)

data\_tensor = torch.ByteTensor(list(data\_bytes)).to(device)

local\_size = torch.tensor([data\_tensor.numel()], device=device)

sizes = [torch.tensor([0], device=device) for \_ in range(world\_size)]

dist.all\_gather(sizes, local\_size)

max\_size = max([s.item() for s in sizes])

if local\_size.item() < max\_size:

padding = torch.zeros(max\_size - local\_size.item(), dtype=torch.uint8, device=device)

data\_tensor = torch.cat([data\_tensor, padding], dim=0)

gathered = [torch.zeros(max\_size, dtype=torch.uint8, device=device) for \_ in range(world\_size)]

dist.all\_gather(gathered, data\_tensor)

all\_data = []

if dist.get\_rank() == 0:

for i, tensor in enumerate(gathered):

size = sizes[i].item()

bytes\_i = tensor[:size].cpu().numpy().tobytes()

data\_i = pickle.loads(bytes\_i)

all\_data.extend(data\_i)

return all\_data

# The setup\_distributed functon is used to setup the environment for multi-gpu training using PyTorch's

# DistributedDataParallel (DDP) package in a SLURM cluster. The information required to setup this function

# along with sample code is referenced from the following sources:

# 1. Manna, S. (2025) The Practical Guide to distributed training using PYTORCH - part 4: On multiple nodes using Slurm, Medium.

# Available at: https://medium.com/the-owl/the-practical-guide-to-distributed-training-using-pytorch-part-4-on-multiple-nodes-using-slurm-83cf306a3373

# 2. PyTorch. Multi-node training using slurm, Multi-Node Training using SLURM.

# Available at: https://pytorch-geometric.readthedocs.io/en/2.6.0/tutorial/multi\_node\_multi\_gpu\_vanilla.html

# 3. Diakogiannis, F. (2024) Distributed training on Slurm Cluster, PyTorch Forums.

# Available at: https://discuss.pytorch.org/t/distributed-training-on-slurm-cluster/150417/13

def setup\_distributed():

rank = int(os.environ['SLURM\_PROCID'])

world\_size = int(os.environ['SLURM\_NTASKS'])

local\_rank = int(os.environ['SLURM\_LOCALID'])

os.environ['MASTER\_ADDR'] = os.environ.get('MASTER\_ADDR', '127.0.0.1')

os.environ['MASTER\_PORT'] = port

dist.init\_process\_group(

backend='nccl',

init\_method='env://',

world\_size=world\_size,

rank=rank)

set\_seed(42)

torch.cuda.set\_device(local\_rank)

device = torch.device(f"cuda:{local\_rank}")

print(f"[Rank {rank}] is using GPU {local\_rank}", flush=True)

return rank, local\_rank, world\_size, device

# The main function, training with and without teacher forciing and validation functions are adapted from the

# ones in train.py hence the same citations apply. Some additions have been made to support multi-GPU training

# using PyTorch's DistributedDataParallel package. These additions include wrapping the models in the DPP package,

# splitting the data across multiple GPUs and syncing the losses and outputs from multiple GPUs. The information

# required to setup multi-GPU using DPP along with sample code is referenced from the following sources:

# 1. PyTorch. DistributedDataParallel - PyTorch 2.8 documentation.

# Available at: https://docs.pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html

# 2. namespace-Pt (2021) A Comprehensive Tutorial to Pytorch DistributedDataParallel, Medium.

# Available at: https://medium.com/codex/a-comprehensive-tutorial-to-pytorch-distributeddataparallel-1f4b42bb1b51

# 3. PyTorch (2017) Distributed communication package - torch.distributed - PyTorch 2.8 documentation.

# Available at: https://docs.pytorch.org/docs/2.8/distributed.html

def main():

rank, local\_rank, world\_size, device = setup\_distributed()

global bestBleu4, epochsSinceImprovement, checkpoint, startEpoch, fineTuneEncoder, dataName, wordMap

# Load word map

wordMapFile = os.path.join(dataFolder, 'WORDMAP\_' + dataName + '.json')

with open(wordMapFile, 'r') as j:

wordMap = json.load(j)

if checkpoint is None:

if lstmDecoder is True:

decoder = DecoderWithAttention(attention\_dim=attentionDim, embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), dropout=dropout, device=device)

else:

decoder = TransformerDecoder(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), maxLen=maxLen, dropout=dropout, device=device,

wordMap=wordMap, pretrained\_embeddings\_path=pretrainedEmbeddingsPath, fine\_tune\_embeddings=True)

# decoder = TransformerDecoderForAttentionViz(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), maxLen=maxLen, dropout=dropout, device=device)

decoderOptimizer = torch.optim.Adam(params=filter(lambda p: p.requires\_grad, decoder.parameters()), lr=decoderLr)

encoder = Encoder()

encoder.fine\_tune(fine\_tune=False)

if fineTuneEncoder is True:

encoderOptimizer = torch.optim.Adam(params=filter(lambda p: p.requires\_grad, encoder.parameters()), lr=encoderLr)

else:

encoderOptimizer = None

results = []

else:

if lstmDecoder is True:

decoder = DecoderWithAttention(attention\_dim=attentionDim, embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), dropout=dropout, device=device)

else:

decoder = TransformerDecoder(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), maxLen=maxLen, dropout=dropout, device=device,

wordMap=wordMap, pretrained\_embeddings\_path=pretrainedEmbeddingsPath, fine\_tune\_embeddings=True)

# decoder = TransformerDecoderForAttentionViz(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), maxLen=maxLen, dropout=dropout, device=device)

decoderOptimizer = torch.optim.Adam(params=filter(lambda p: p.requires\_grad, decoder.parameters()), lr=decoderLr)

encoder = Encoder()

checkpoint = torch.load(checkpoint, map\_location=device, weights\_only=False)

encoder.load\_state\_dict(checkpoint['encoder'])

startEpoch = checkpoint['epoch'] + 1

if startEpoch > 20:

fineTuneEncoder = True

encoder.fine\_tune(fine\_tune=fineTuneEncoder, startingLayer=startingLayer)

else:

fineTuneEncoder = False

encoder.fine\_tune(fine\_tune=fineTuneEncoder)

decoder.load\_state\_dict(checkpoint['decoder'])

decoderOptimizer.load\_state\_dict(checkpoint['decoderOptimizer'])

optimizer\_to\_device(decoderOptimizer, device)

if fineTuneEncoder is True:

encoderOptimizer = torch.optim.Adam(params=filter(lambda p: p.requires\_grad, encoder.parameters()), lr=encoderLr)

if checkpoint['encoderOptimizer'] is not None:

encoderOptimizer.load\_state\_dict(checkpoint['encoderOptimizer'])

optimizer\_to\_device(encoderOptimizer, device)

else:

encoderOptimizer = None

epochsSinceImprovement = checkpoint['epochsSinceImprovement']

bestBleu4 = checkpoint['bleu-4']

results = checkpoint['results']

decoder = decoder.to(device)

encoder = encoder.to(device)

decoder = DDP(decoder, device\_ids=[local\_rank], output\_device=local\_rank)

if fineTuneEncoder is True:

encoder = DDP(encoder, device\_ids=[local\_rank], output\_device=local\_rank)

criterion = nn.CrossEntropyLoss().to(device)

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

trainDataset = CaptionDataset(dataFolder, dataName, 'TRAIN', transform=transforms.Compose([normalize]))

trainSampler = torch.utils.data.distributed.DistributedSampler(trainDataset, num\_replicas=world\_size, rank=rank, shuffle=True, seed=42)

trainDataLoader = DataLoader(trainDataset, batch\_size=batchSize, shuffle=False, num\_workers=workers, persistent\_workers=True, pin\_memory=True, sampler=trainSampler)

valDataset = CaptionDataset(dataFolder, dataName, 'VAL', transform=transforms.Compose([normalize]))

valSampler = torch.utils.data.distributed.DistributedSampler(valDataset, num\_replicas=world\_size, rank=rank, shuffle=True, seed=42)

valDataLoader = DataLoader(valDataset, batch\_size=batchSize, shuffle=False, num\_workers=workers, persistent\_workers=True, pin\_memory=True, sampler=valSampler)

for epoch in range(startEpoch, epochs):

trainSampler.set\_epoch(epoch)

valSampler.set\_epoch(epoch)

if epoch == 20:

fineTuneEncoder = True

encoder.fine\_tune(fine\_tune=fineTuneEncoder, startingLayer=startingLayer)

encoderOptimizer = torch.optim.Adam(params=filter(lambda p: p.requires\_grad, encoder.parameters()), lr=encoderLr)

optimizer\_to\_device(encoderOptimizer, device)

encoder = DDP(encoder, device\_ids=[local\_rank], output\_device=local\_rank)

print(f"Fine-tuning encoder from epoch 20 onwards (starting from layer {startingLayer})", flush=True)

# Decay learning rate if there is no improvement for 8 consecutive epochs, and terminate training after 20

if epochsSinceImprovement == 40: # 20

break

if epochsSinceImprovement > 0 and epochsSinceImprovement % 8 == 0:

adjust\_learning\_rate(decoderOptimizer, 0.8)

if fineTuneEncoder:

adjust\_learning\_rate(encoderOptimizer, 0.8)

if teacherForcing is True:

trainLoss, trainTop5Acc, trainBatchTime, trainDataTime = trainWithTeacherForcing(trainDataLoader=trainDataLoader,

encoder=encoder,

decoder=decoder,

criterion=criterion,

encoderOptimizer=encoderOptimizer,

decoderOptimizer=decoderOptimizer,

epoch=epoch,

device=device,

world\_size=world\_size)

else:

trainLoss, trainTop5Acc, trainBatchTime, trainDataTime = trainWithoutTeacherForcing(trainDataLoader=trainDataLoader,

encoder=encoder,

decoder=decoder,

criterion=criterion,

encoderOptimizer=encoderOptimizer,

decoderOptimizer=decoderOptimizer,

epoch=epoch,

device=device,

world\_size=world\_size)

valLoss, valTop5Acc, bleu1, bleu2, bleu3, recentBleu4 = validate(valDataLoader=valDataLoader,

encoder=encoder,

decoder=decoder,

criterion=criterion,

device=device,

world\_size=world\_size)

if dist.get\_rank() == 0:

results.append({

'epoch': epoch,

'trainLoss': trainLoss,

'trainTop5Acc': trainTop5Acc,

'trainBatchTime': trainBatchTime,

'trainDataTime': trainDataTime,

'valLoss': valLoss,

'valTop5Acc': valTop5Acc,

'bleu1': bleu1,

'bleu2': bleu2,

'bleu3': bleu3,

'bleu4': recentBleu4

})

isBest = recentBleu4 > bestBleu4

bestBleu4 = max(recentBleu4, bestBleu4)

if not isBest:

epochsSinceImprovement += 1

print("\nEpochs since last improvement: %d\n" % (epochsSinceImprovement,))

else:

epochsSinceImprovement = 0

# Save checkpoint

encoderSaved = encoder.module.state\_dict() if hasattr(encoder, 'module') else encoder.state\_dict()

decoderSaved = decoder.module.state\_dict() if hasattr(decoder, 'module') else decoder.state\_dict()

save\_checkpoint(dataName, epoch, epochsSinceImprovement, encoderSaved, decoderSaved, encoderOptimizer,

decoderOptimizer, recentBleu4, isBest, results, lstmDecoder, startingLayer, encoderLr,

pretrainedEmbeddingsName)

epochsSinceImprovementTensor = torch.tensor(epochsSinceImprovement, device=device)

dist.broadcast(epochsSinceImprovementTensor, src=0)

epochsSinceImprovement = epochsSinceImprovementTensor.item()

if dist.get\_rank() == 0:

resultsDF = pd.DataFrame(results)

os.makedirs('results', exist\_ok=True)

if lstmDecoder is True:

resultsDF.to\_csv(f'results/metrics-lstmDecoder(trainingTF-inferenceNoTF-Finetuning{startingLayer}-{encoderLr}-{pretrainedEmbeddingsName}).csv', index=False)

else:

resultsDF.to\_csv(f'results/metrics-transformerDecoder(trainingTF-inferenceNoTF-Finetuning{startingLayer}-{encoderLr}-{pretrainedEmbeddingsName}).csv', index=False)

def trainWithTeacherForcing(trainDataLoader, encoder, decoder, criterion, encoderOptimizer, decoderOptimizer, epoch, device, world\_size):

encoder.train()

decoder.train()

batchTime = AverageMeter()

dataTime = AverageMeter()

losses = AverageMeter()

top5accs = AverageMeter()

start = time.time()

for i, (imgs, caps, caplens) in enumerate(trainDataLoader):

dataTime.update(time.time() - start)

rank = dist.get\_rank()

if (i % 1000 == 0):

print(f"TF, Rank: {rank}, Epoch {epoch}, Batch {i + 1}/{len(trainDataLoader)}", flush=True)

imgs = imgs.to(device)

caps = caps.to(device)

caplens = caplens.to(device)

imgs = encoder(imgs)

if lstmDecoder is True:

scores, capsSorted, decodeLengths, alphas, sortInd = decoder(teacherForcing=True, encoder\_out=imgs, encoded\_captions=caps, caption\_lengths=caplens)

# Since we decoded starting with <start>, the targets are all words after <start>, up to <end>

targets = capsSorted[:, 1:] # still in the form of indices

# Remove timesteps that we didn't decode at, or are pads

# pack\_padded\_sequence is an easy trick to do this

scores = pack\_padded\_sequence(scores, decodeLengths, batch\_first=True).data # scores are logits

targets = pack\_padded\_sequence(targets, decodeLengths, batch\_first=True).data

loss = criterion(scores, targets)

# Add doubly stochastic attention regularization

loss += alphaC \* ((1. - alphas.sum(dim=1)) \*\* 2).mean()

else:

tgt\_key\_padding\_mask = (caps == wordMap['<pad>'])

scores, capsSorted, decodeLengths = decoder(teacherForcing=True, encoder\_out=imgs, encoded\_captions=caps, caption\_lengths=caplens, tgt\_key\_padding\_mask=tgt\_key\_padding\_mask)

# scores, capsSorted, decodeLengths, alphas = decoder(teacherForcing=True, encoder\_out=imgs, encoded\_captions=caps, caption\_lengths=caplens, tgt\_key\_padding\_mask=tgt\_key\_padding\_mask)

targets = capsSorted[:, 1:] # still in the form of indices

scores = pack\_padded\_sequence(scores, decodeLengths, batch\_first=True, enforce\_sorted=False).data # scores are logits

targets = pack\_padded\_sequence(targets, decodeLengths, batch\_first=True, enforce\_sorted=False).data

loss = criterion(scores, targets)

# Add doubly stochastic attention regularization

# loss += alphaC \* ((1. - alphas.sum(dim=1)) \*\* 2).mean()

if encoderOptimizer is not None:

encoderOptimizer.zero\_grad()

decoderOptimizer.zero\_grad()

loss.backward()

# Clip gradients

if gradClip is not None:

clip\_gradient(decoderOptimizer, gradClip)

if encoderOptimizer is not None:

clip\_gradient(encoderOptimizer, gradClip)

if encoderOptimizer is not None:

encoderOptimizer.step()

decoderOptimizer.step()

globalLoss, totalTokens = reduceLossAndTokens(loss, sum(decodeLengths), device)

correct5, total = accuracy(scores, targets, 5, 'multi')

correct5 = torch.tensor(correct5, dtype=torch.float32, device=device)

total = torch.tensor(total, dtype=torch.float32, device=device)

dist.all\_reduce(correct5, op=dist.ReduceOp.SUM)

dist.all\_reduce(total, op=dist.ReduceOp.SUM)

top5 = (correct5 / total).item() \* 100

# Keep track of metrics

losses.update(globalLoss, totalTokens)

top5accs.update(top5, total.item())

batchTime.update(time.time() - start)

start = time.time()

batchTimeTensor = torch.tensor(batchTime.avg).to(device)

dataTimeTensor = torch.tensor(dataTime.avg).to(device)

dist.all\_reduce(batchTimeTensor, op=dist.ReduceOp.SUM)

dist.all\_reduce(dataTimeTensor, op=dist.ReduceOp.SUM)

batchTimeAvg = batchTimeTensor.item() / world\_size

dataTimeAvg = dataTimeTensor.item() / world\_size

print(f"TF, Rank: {rank}, Epoch {epoch}: Training Loss = {losses.avg:.4f}, Top-5 Accuracy = {top5accs.avg:.4f}", flush=True)

return losses.avg, top5accs.avg, batchTimeAvg, dataTimeAvg

def trainWithoutTeacherForcing(trainDataLoader, encoder, decoder, criterion, encoderOptimizer, decoderOptimizer, epoch, device, world\_size):

encoder.train()

decoder.train()

batchTime = AverageMeter() # forward prop. + back prop. time

dataTime = AverageMeter() # data loading time

losses = AverageMeter() # loss (per word decoded)

top5accs = AverageMeter() # top5 accuracy

start = time.time()

for i, (imgs, caps, caplens) in enumerate(trainDataLoader):

dataTime.update(time.time() - start)

rank = dist.get\_rank()

if (i % 1000 == 0):

print(f"No TF, Rank: {rank}, Epoch {epoch}, Batch {i + 1}/{len(trainDataLoader)}", flush=True)

imgs = imgs.to(device)

caps = caps.to(device)

caplens = caplens.to(device)

imgs = encoder(imgs)

if lstmDecoder is True:

scores, alphas, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

scoresUpdated, targetsUpdated, totalTokensEvaluated, actualDecodeLengths = preprocessDecoderOutputForMetrics(scores, sequences, caps, wordMap['<end>'], wordMap['<pad>'], 51)

loss = criterion(scoresUpdated, targetsUpdated)

loss += alphaC \* ((1. - alphas.sum(dim=1)) \*\* 2).mean()

else:

scores, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

# scores, sequences, alphas = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

scoresUpdated, targetsUpdated, totalTokensEvaluated, actualDecodeLengths = preprocessDecoderOutputForMetrics(scores, sequences, caps, wordMap['<end>'], wordMap['<pad>'], 51)

loss = criterion(scoresUpdated, targetsUpdated)

# loss += alphaC \* ((1. - alphas.sum(dim=1)) \*\* 2).mean()

if encoderOptimizer is not None:

encoderOptimizer.zero\_grad()

decoderOptimizer.zero\_grad()

loss.backward()

# Clip gradients

if gradClip is not None:

clip\_gradient(decoderOptimizer, gradClip)

if encoderOptimizer is not None:

clip\_gradient(encoderOptimizer, gradClip)

if encoderOptimizer is not None:

encoderOptimizer.step()

decoderOptimizer.step()

globalLoss, totalTokens = reduceLossAndTokens(loss, totalTokensEvaluated, device)

correct5, total = accuracy(scoresUpdated, targetsUpdated, 5, 'multi')

correct5 = torch.tensor(correct5, dtype=torch.float32, device=device)

total = torch.tensor(total, dtype=torch.float32, device=device)

dist.all\_reduce(correct5, op=dist.ReduceOp.SUM)

dist.all\_reduce(total, op=dist.ReduceOp.SUM)

top5 = (correct5 / total).item() \* 100

# Keep track of metrics

losses.update(globalLoss, totalTokens)

top5accs.update(top5, total.item())

batchTime.update(time.time() - start)

start = time.time()

batchTimeTensor = torch.tensor(batchTime.avg).to(device)

dataTimeTensor = torch.tensor(dataTime.avg).to(device)

dist.all\_reduce(batchTimeTensor, op=dist.ReduceOp.SUM)

dist.all\_reduce(dataTimeTensor, op=dist.ReduceOp.SUM)

batchTimeAvg = batchTimeTensor.item() / world\_size

dataTimeAvg = dataTimeTensor.item() / world\_size

print(f"No TF, Rank: {rank}, Epoch {epoch}: Training Loss = {losses.avg:.4f}, Top-5 Accuracy = {top5accs.avg:.4f}", flush=True)

return losses.avg, top5accs.avg, batchTimeAvg, dataTimeAvg

def validate(valDataLoader, encoder, decoder, criterion, device, world\_size):

decoder.eval()

if encoder is not None:

encoder.eval()

batchTime = AverageMeter()

losses = AverageMeter()

top5accs = AverageMeter()

start = time.time()

references = list() # references (true captions) for calculating BLEU-4 score

hypotheses = list() # hypotheses (predictions)

with torch.no\_grad():

for i, (imgs, caps, caplens, allcaps) in enumerate(valDataLoader):

rank = dist.get\_rank()

if (i % 100 == 0):

print(f"No TF, Rank: {rank}, Validation Batch {i + 1}/{len(valDataLoader)}", flush=True)

imgs = imgs.to(device)

caps = caps.to(device)

caplens = caplens.to(device)

if encoder is not None:

imgs = encoder(imgs)

if lstmDecoder is True:

scores, alphas, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

scoresUpdated, targetsUpdated, totalTokensEvaluated, actualDecodeLengths = preprocessDecoderOutputForMetrics(scores, sequences, caps, wordMap['<end>'], wordMap['<pad>'], 51)

loss = criterion(scoresUpdated, targetsUpdated)

# Add doubly stochastic attention regularization

loss += alphaC \* ((1. - alphas.sum(dim=1)) \*\* 2).mean()

else:

scores, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

# scores, sequences, alphas = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

scoresUpdated, targetsUpdated, totalTokensEvaluated, actualDecodeLengths = preprocessDecoderOutputForMetrics(scores, sequences, caps, wordMap['<end>'], wordMap['<pad>'], 51)

loss = criterion(scoresUpdated, targetsUpdated)

# loss += alphaC \* ((1. - alphas.sum(dim=1)) \*\* 2).mean()

globalLoss, totalTokens = reduceLossAndTokens(loss, totalTokensEvaluated, device)

correct5, total = accuracy(scoresUpdated, targetsUpdated, 5, 'multi')

correct5 = torch.tensor(correct5, dtype=torch.float32, device=device)

total = torch.tensor(total, dtype=torch.float32, device=device)

dist.all\_reduce(correct5, op=dist.ReduceOp.SUM)

dist.all\_reduce(total, op=dist.ReduceOp.SUM)

top5 = (correct5 / total).item() \* 100

losses.update(globalLoss, totalTokens)

top5accs.update(top5, total.item())

batchTime.update(time.time() - start)

start = time.time()

# References

allcaps = allcaps.to(device)

for j in range(allcaps.shape[0]): # Iterate through each image in the batch

imgCaps = allcaps[j].tolist() # This would be a list of lists, where each inner list is a reference

imgCaptions = []

for c\_list in imgCaps: # Iterate through each reference caption for the current image

filtered\_caption = [w for w in c\_list if w not in {wordMap['<start>'], wordMap['<pad>']}]

imgCaptions.append(filtered\_caption)

references.append(imgCaptions)

# Hypotheses

batchHypotheses = [] # Create a temporary list to hold all captions for this batch

for j, p\_seq\_tensor in enumerate(sequences):

truncated\_predicted\_list = p\_seq\_tensor[:actualDecodeLengths[j]].tolist()

batchHypotheses.append(truncated\_predicted\_list)

hypotheses.extend(batchHypotheses)

assert len(references) == len(hypotheses)

batchTimeTensor = torch.tensor(batchTime.avg).to(device)

dist.all\_reduce(batchTimeTensor, op=dist.ReduceOp.SUM)

batchTimeAvg = batchTimeTensor.item() / world\_size

all\_references = gather\_all\_data(references, world\_size, device)

all\_hypotheses = gather\_all\_data(hypotheses, world\_size, device)

if dist.get\_rank() == 0:

bleu1 = corpus\_bleu(all\_references, all\_hypotheses, weights=(1.0, 0.0, 0.0, 0.0))

bleu2 = corpus\_bleu(all\_references, all\_hypotheses, weights=(0.5, 0.5, 0.0, 0.0))

bleu3 = corpus\_bleu(all\_references, all\_hypotheses, weights=(0.33, 0.33, 0.33, 0.0))

bleu4 = corpus\_bleu(all\_references, all\_hypotheses, weights=(0.25, 0.25, 0.25, 0.25))

print(f"No TF, Rank = {rank}, Validation Loss = {losses.avg:.4f}, Top-5 Accuracy = {top5accs.avg:.4f}, Bleu-1 = {bleu1:.4f}, Bleu-2 = {bleu2:.4f}, Bleu-3 = {bleu3:.4f}, Bleu-4 = {bleu4:.4f}", flush=True)

else:

bleu1 = bleu2 = bleu3 = bleu4 = None

dist.barrier()

return losses.avg, top5accs.avg, bleu1, bleu2, bleu3, bleu4

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

main()

## 5.3. test.py

import torch

import os

os.environ["CUBLAS\_WORKSPACE\_CONFIG"] = ":4096:8"

import random

import numpy as np

from models.encoder import Encoder

from models.decoder import DecoderWithAttention

from models.lstmNoAttention import DecoderWithoutAttention

from models.transformerDecoder import TransformerDecoder

from models.transformerDecoderAttVis import TransformerDecoderForAttentionViz

def set\_seed(seed):

random.seed(seed)

np.random.seed(seed)

torch.manual\_seed(seed)

torch.cuda.manual\_seed(seed)

torch.cuda.manual\_seed\_all(seed)

os.environ["PYTHONHASHSEED"] = str(seed)

torch.use\_deterministic\_algorithms(True)

def seed\_worker(worker\_id):

worker\_seed = torch.initial\_seed() % 2\*\*32

np.random.seed(worker\_seed)

random.seed(worker\_seed)

from torch.utils.data import DataLoader

import torch.backends.cudnn as cudnn

from dataLoader import CaptionDataset

import torchvision.transforms as transforms

import json

import time

import os

from torch import nn

from torch.nn.utils.rnn import pack\_padded\_sequence

from nltk.translate.bleu\_score import corpus\_bleu

import pandas as pd

from utils.utils import \*

import torch.distributed as dist

from torch.nn.parallel import DistributedDataParallel as DDP

from torch.serialization import add\_safe\_globals

import argparse

device = torch.device("cuda")

# Model parameters

embDim = 512 # dimension of word embeddings

attentionDim = 512 # dimension of attention linear layers

decoderDim = 512 # dimension of decoder RNN

dropout = 0.5

maxLen = 52 # maximum length of captions (in words), used for padding

# Data parameters

dataFolder = 'cocoDataset/inputFiles'

dataName = 'coco\_5\_cap\_per\_img\_5\_min\_word\_freq'

batchSize = 32

workers = 6

alphaC = 1 # regularization parameter for 'doubly stochastic attention', as in the paper

cudnn.benchmark = False # set to true only if inputs to model are fixed size; otherwise lot of computational overhead

cudnn.deterministic = True # for reproducibility

parser = argparse.ArgumentParser()

parser.add\_argument('--checkpoint', type=str, default=None, help='Path to checkpoint file')

parser.add\_argument('--lstmDecoder', action='store\_true', help='Use LSTM decoder instead of Transformer')

parser.add\_argument('--startingLayer', type=int, default=None, help='Starting layer index for encoder fine-tuning encoder')

parser.add\_argument('--embeddingName', type=str, default=None, help='Pretrained embedding name from gensim')

args = parser.parse\_args()

modelPath = args.checkpoint

lstmDecoder = args.lstmDecoder

startingLayer = args.startingLayer

pretrainedEmbeddingsName = args.embeddingName # word2vec-google-news-300

if pretrainedEmbeddingsName == 'word2vec-google-news-300':

embDim = 300

pretrainedEmbeddingsPath = 'wordEmbeddings/word2vec-google-news-300.gz'

elif pretrainedEmbeddingsName == 'glove-wiki-gigaword-200':

embDim = 200

pretrainedEmbeddingsPath = 'wordEmbeddings/glove-wiki-gigaword-200.gz'

else:

pretrainedEmbeddingsPath = None

# The main function has been adapted from the main function in train.py hence the same citations apply.

# It has been modified to handle testing the Transformer decoder as well which is a contribution of this study.

def main():

g = torch.Generator()

g.manual\_seed(42)

global wordMap

wordMapFile = os.path.join(dataFolder, 'WORDMAP\_' + dataName + '.json')

with open(wordMapFile, 'r') as j:

wordMap = json.load(j)

checkpoint = torch.load(modelPath, map\_location=device, weights\_only=False)

if lstmDecoder is True:

decoder = DecoderWithAttention(attention\_dim=attentionDim, embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), dropout=dropout, device=device)

else:

decoder = TransformerDecoder(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), maxLen=maxLen, dropout=dropout, device=device,

wordMap=wordMap, pretrained\_embeddings\_path=pretrainedEmbeddingsPath, fine\_tune\_embeddings=True)

encoder = Encoder()

encoder.load\_state\_dict(checkpoint['encoder'])

decoder.load\_state\_dict(checkpoint['decoder'])

decoder = decoder.to(device)

encoder = encoder.to(device)

criterion = nn.CrossEntropyLoss().to(device)

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

testDataset = CaptionDataset(dataFolder, dataName, 'TEST', transform=transforms.Compose([normalize]))

testDataLoader = DataLoader(testDataset, batch\_size=batchSize, shuffle=False, num\_workers=workers, persistent\_workers=True, pin\_memory=True, worker\_init\_fn=seed\_worker, generator=g)

results = []

testLoss, testTop5Acc, bleu1, bleu2, bleu3, bleu4 = test(testDataLoader=testDataLoader,

encoder=encoder,

decoder=decoder,

criterion=criterion)

results.append({

'testLoss': testLoss,

'testTop5Acc': testTop5Acc,

'bleu1': bleu1,

'bleu2': bleu2,

'bleu3': bleu3,

'bleu4': bleu4

})

resultsDF = pd.DataFrame(results)

os.makedirs('results', exist\_ok=True)

if lstmDecoder is True:

resultsDF.to\_csv(f'results/test-lstmDecoder-TeacherForcing-Finetuning{startingLayer}.csv', index=False)

else:

resultsDF.to\_csv(f'results/test-TransformerDecoder-TeacherForcing-Finetuning{startingLayer}-{pretrainedEmbeddingsName}.csv', index=False)

# The original study (Ramos et al., 2024) did not have a test function hence this test function has been adapted from

# the validation function in train.py thus the same citations apply. The test method calls the corresponding non-teacher

# forcing forward method of each decoder and aligns their outputs in the preprocessDecoderOutputForMetrics function for

# the evaluation metrics. It also calculates all four BLEU scores. These are contribution of this study.

def test(testDataLoader, encoder, decoder, criterion):

decoder.eval()

if encoder is not None:

encoder.eval()

batchTime = AverageMeter()

losses = AverageMeter()

top5accs = AverageMeter()

start = time.time()

references = list() # references (true captions) for calculating BLEU-4 score

hypotheses = list() # hypotheses (predictions)

with torch.no\_grad():

for i, (imgs, caps, caplens, allcaps) in enumerate(testDataLoader):

print(f"Test Batch {i + 1}/{len(testDataLoader)}")

imgs = imgs.to(device)

caps = caps.to(device)

caplens = caplens.to(device)

if encoder is not None:

imgs = encoder(imgs)

if lstmDecoder is True:

scores, alphas, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

scoresUpdated, targetsUpdated, totalTokensEvaluated, actualDecodeLengths = preprocessDecoderOutputForMetrics(scores, sequences, caps, wordMap['<end>'], wordMap['<pad>'], 51)

loss = criterion(scoresUpdated, targetsUpdated)

# Add doubly stochastic attention regularization

loss += alphaC \* ((1. - alphas.sum(dim=1)) \*\* 2).mean()

else:

scores, sequences = decoder(teacherForcing=False, encoder\_out=imgs, wordMap=wordMap, maxDecodeLen=51)

scoresUpdated, targetsUpdated, totalTokensEvaluated, actualDecodeLengths = preprocessDecoderOutputForMetrics(scores, sequences, caps, wordMap['<end>'], wordMap['<pad>'], 51)

loss = criterion(scoresUpdated, targetsUpdated)

top5 = accuracy(scoresUpdated, targetsUpdated, 5, 'single')

losses.update(loss.item(), totalTokensEvaluated)

top5accs.update(top5, totalTokensEvaluated)

batchTime.update(time.time() - start)

start = time.time()

# References

allcaps = allcaps.to(device)

for j in range(allcaps.shape[0]): # Iterate through each image in the batch

imgCaps = allcaps[j].tolist() # This would be a list of lists, where each inner list is a reference

imgCaptions = []

for c\_list in imgCaps: # Iterate through each reference caption for the current image

filtered\_caption = [w for w in c\_list if w not in {wordMap['<start>'], wordMap['<pad>']}]

imgCaptions.append(filtered\_caption)

references.append(imgCaptions)

# Hypotheses

batchHypotheses = [] # Create a temporary list to hold all captions for this batch

for j, p\_seq\_tensor in enumerate(sequences):

truncated\_predicted\_list = p\_seq\_tensor[:actualDecodeLengths[j]].tolist()

batchHypotheses.append(truncated\_predicted\_list)

hypotheses.extend(batchHypotheses)

assert len(references) == len(hypotheses)

# bleu4 = corpus\_bleu(references, hypotheses)

bleu1 = corpus\_bleu(references, hypotheses, weights=(1.0, 0.0, 0.0, 0.0))

bleu2 = corpus\_bleu(references, hypotheses, weights=(0.5, 0.5, 0.0, 0.0))

bleu3 = corpus\_bleu(references, hypotheses, weights=(0.33, 0.33, 0.33, 0.0))

bleu4 = corpus\_bleu(references, hypotheses, weights=(0.25, 0.25, 0.25, 0.25))

print(f"Test Loss = {losses.avg:.4f}, Top-5 Accuracy = {top5accs.avg:.4f}, Bleu-1 = {bleu1:.4f}, Bleu-2 = {bleu2:.4f}, Bleu-3 = {bleu3:.4f}, Bleu-4 = {bleu4:.4f}")

return losses.avg, top5accs.avg, bleu1, bleu2, bleu3, bleu4

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

main()

# 6. caption.py

import torch

import torch.nn.functional as F

import torch.nn as nn

import numpy as np

import os

import json

import torchvision.transforms as transforms

import matplotlib.pyplot as plt

import matplotlib.cm as cm

import skimage.transform

import argparse

from PIL import Image

from models.encoder import Encoder

from models.decoder import DecoderWithAttention

from models.lstmNoAttention import DecoderWithoutAttention

from models.transformerDecoder import TransformerDecoder

from models.transformerDecoderAttVis import TransformerDecoderForAttentionViz

import csv

import pandas as pd

device = torch.device("cpu")

embDim = 512

attentionDim = 512

decoderDim = 512

dropout = 0.5

maxLen = 52

lstmDecoder = False

dataFolder = 'cocoDataset/inputFiles'

dataName = 'coco\_5\_cap\_per\_img\_5\_min\_word\_freq'

# The caption\_image\_beam\_search and visualize\_att functions are adapted from the codebase of the original study (Ramos et al., 2024).

# Link to their GitHub repository: https://github.com/Leo-Thomas/ConvNeXt-for-Image-Captioning/tree/main

# The original study (Ramos et al., 2024) seem to have adapted their code from another repository (Vinodababu, 2019)

# which is a popular open source implementation of the 'Show, Attend and Tell' paper (Xu et al., 2015).

# Link to the (Vinodababu, 2019) repository: https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning

# Some modifications were made to the visualize\_att function to overcome errors with displaying the attention weights

def caption\_image\_beam\_search(encoder, decoder, imagePath, wordMap, beamSize=3):

"""

Reads an image and captions it with beam search.

:param encoder: encoder model

:param decoder: decoder model

:param image\_path: path to image

:param word\_map: word map

:param beam\_size: number of sequences to consider at each decode-step

:return: caption, weights for visualization

"""

k = beamSize

vocabSize = len(wordMap)

# Read image and process

# img = imread(imagePath)

img = Image.open(imagePath).convert('RGB')

img = img.resize((256, 256), Image.Resampling.BICUBIC)

img = np.array(img)

if len(img.shape) == 2:

img = img[:, :, np.newaxis]

img = np.concatenate([img, img, img], axis=2)

img = img.transpose(2, 0, 1)

img = img / 255.

img = torch.FloatTensor(img).to(device)

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

transform = transforms.Compose([normalize])

image = transform(img) # (3, 256, 256)

# Encode

image = image.unsqueeze(0) # (1, 3, 256, 256)

encoderOut = encoder(image) # (1, enc\_image\_size, enc\_image\_size, encoder\_dim)

encImageSize = encoderOut.size(1)

encoderDim = encoderOut.size(3)

# Flatten encoding

encoderOut = encoderOut.view(1, -1, encoderDim) # (1, num\_pixels, encoder\_dim)

numPixels = encoderOut.size(1)

# We'll treat the problem as having a batch size of k

encoderOut = encoderOut.expand(k, numPixels, encoderDim) # (k, num\_pixels, encoder\_dim)

# Tensor to store top k previous words at each step; now they're just <start>

kPrevWords = torch.LongTensor([[wordMap['<start>']]] \* k).to(device) # (k, 1)

# Tensor to store top k sequences; now they're just <start>

seqs = kPrevWords # (k, 1)

# Tensor to store top k sequences' scores; now they're just 0

topKScores = torch.zeros(k, 1).to(device) # (k, 1)

# Tensor to store top k sequences' alphas; now they're just 1s

seqsAlpha = torch.ones(k, 1, encImageSize, encImageSize).to(device) # (k, 1, enc\_image\_size, enc\_image\_size)

# Lists to store completed sequences, their alphas and scores

completeSeqs = list()

completeSeqsAlpha = list()

completeSeqsScores = list()

# Start decoding

step = 1

h, c = decoder.init\_hidden\_state(encoderOut)

while True:

embeddings = decoder.embedding(kPrevWords).squeeze(1) # (k, embed\_dim)

awe, alpha = decoder.attention(encoderOut, h) # (k, encoder\_dim), (k, num\_pixels)

alpha = alpha.view(-1, encImageSize, encImageSize) # (k, enc\_image\_size, enc\_image\_size)

gate = decoder.sigmoid(decoder.f\_beta(h)) # gating scalar, (k, encoder\_dim)

awe = gate \* awe

h, c = decoder.decode\_step(torch.cat([embeddings, awe], dim=1), (h, c)) # (k, decoder\_dim)

scores = decoder.fc(h) # (k, vocab\_size)

scores = F.log\_softmax(scores, dim=1)

# Add

scores = topKScores.expand\_as(scores) + scores # (k, vocab\_size)

# For the first step, all k points will have the same scores (since same k previous words, h, c)

if step == 1:

topKScores, topKWords = scores[0].topk(k, 0, True, True) # (k)

else:

# Unroll and find top scores, and their unrolled indices

topKScores, topKWords = scores.view(-1).topk(k, 0, True, True) # (k)

# Convert unrolled indices to actual indices of scores

prevWordInds = topKWords / vocabSize # (k)

nextWordInds = topKWords % vocabSize # (k)

prevWordInds = prevWordInds.long() # my addition

# Add new words to sequences, alphas

seqs = torch.cat([seqs[prevWordInds], nextWordInds.unsqueeze(1)], dim=1) # (k, step+1)

seqsAlpha = torch.cat([seqsAlpha[prevWordInds], alpha[prevWordInds].unsqueeze(1)], dim=1) # (k, step+1, enc\_image\_size, enc\_image\_size)

# Which sequences are incomplete (didn't reach <end>)?

incompleteInds = [ind for ind, nextWord in enumerate(nextWordInds) if nextWord != wordMap['<end>']]

completeInds = list(set(range(len(nextWordInds))) - set(incompleteInds))

# Set aside complete sequences

if len(completeInds) > 0:

completeSeqs.extend(seqs[completeInds].tolist())

completeSeqsAlpha.extend(seqsAlpha[completeInds].tolist())

completeSeqsScores.extend(topKScores[completeInds])

k -= len(completeInds) # reduce beam length accordingly

# Proceed with incomplete sequences

if k == 0:

break

seqs = seqs[incompleteInds]

seqsAlpha = seqsAlpha[incompleteInds]

h = h[prevWordInds[incompleteInds]]

c = c[prevWordInds[incompleteInds]]

encoderOut = encoderOut[prevWordInds[incompleteInds]]

topKScores = topKScores[incompleteInds].unsqueeze(1)

kPrevWords = nextWordInds[incompleteInds].unsqueeze(1)

# Break if things have been going on too long

if step > 50:

break

step += 1

i = completeSeqsScores.index(max(completeSeqsScores))

seq = completeSeqs[i]

alphas = completeSeqsAlpha[i]

return seq, alphas

# This function generates a caption using the transformer decoder but does not return the attention weights since it uses the TransformerDecoder

# class in transformerDecoder.py

def caption\_image\_beam\_search\_transformer(encoder, decoder, imagePath, wordMap, beamSize=3, max\_decode\_len= 51):

# The initial section of this function is adapted from the caption\_image\_beam\_search function hence the same citations apply.

k = beamSize

vocab\_size = len(wordMap)

end\_token\_idx = wordMap['<end>']

img = Image.open(imagePath).convert('RGB')

img = img.resize((256, 256), Image.Resampling.BICUBIC)

img = np.array(img)

if len(img.shape) == 2:

img = np.stack([img, img, img], axis=2)

img = img.transpose(2, 0, 1)

img = img / 255.

img = torch.FloatTensor(img).to(device)

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

transform = transforms.Compose([normalize])

image = transform(img)

image = image.unsqueeze(0)

encoderOut = encoder(image)

encoderDim = encoderOut.size(3)

encoderOutProj = decoder.encoder\_proj(encoderOut.view(1, -1, encoderDim)).permute(1, 0, 2)

encoderOutExpanded = encoderOutProj.expand(-1, k, -1)

kPrevWords = torch.full((k, 1), wordMap['<start>'], dtype=torch.long, device=device)

topKScores = torch.zeros(k, 1, device=device) # (k, 1)

completeSeqs = list()

completeSeqsScores = list()

step = 0

finishedSequences = torch.zeros(k, dtype=torch.bool, device=device)

# This section of the function is also adapted from the caption\_image\_beam\_search function however, modifications have been

# made for the transformer decoder. These modifications are taken from the forwardWithoutTeacherForcing method in

# transformerDecoder.py for which the Datacamp tutorial (Sarkar, 2025) was used to understand the general structure of the

# transformer decoder whereas the TransformerDecoderLayer and TransformerDecoder classes from the PyTorch documentation were

# used to implement it. The same citations as TransformerDecoder in transformerDecoder.py apply to this.

while True:

active = (~finishedSequences).nonzero(as\_tuple=False).squeeze(1)

if len(active) == 0:

break

kPrevWordsActive = kPrevWords[active]

encoderOutActive = encoderOutExpanded[:, active, :]

embeddingsActive = decoder.embedding(kPrevWordsActive)

embeddingsActive = decoder.pos\_encoding(decoder.dropout(embeddingsActive))

tgtActive = embeddingsActive.permute(1, 0, 2)

tgtMask = nn.Transformer.generate\_square\_subsequent\_mask(tgtActive.size(0)).to(device).bool()

decoderOutput = decoder.transformer\_decoder(

tgtActive,

encoderOutActive,

tgt\_mask=tgtMask)

lastTokenOutputActive = decoderOutput[-1, :, :]

scoresActive = decoder.fc\_out(lastTokenOutputActive)

scoresActive = F.log\_softmax(scoresActive, dim=1)

topKScoresActive = topKScores[active]

scoresActive = topKScoresActive.expand\_as(scoresActive) + scoresActive

if step == 0:

topKScoresNew, topKUnrolledIndices = scoresActive[0].topk(k, 0, True, True)

else:

topKScoresNew, topKUnrolledIndices = scoresActive.view(-1).topk(k, 0, True, True)

prevWordActiveIndices = topKUnrolledIndices / vocab\_size

nextWordsIds = topKUnrolledIndices % vocab\_size

prevWordActiveIndices = prevWordActiveIndices.long()

kIndicesForNextStep = active[prevWordActiveIndices]

newKPrevWordsIds = torch.cat([kPrevWords[kIndicesForNextStep], nextWordsIds.unsqueeze(1)], dim=1)

newTopKScores = topKScoresNew.unsqueeze(1)

justCompletedMask = (nextWordsIds == end\_token\_idx)

justCompletedIndices = torch.nonzero(justCompletedMask, as\_tuple=False).squeeze(1)

if len(justCompletedIndices) > 0:

completeSeqs.extend(newKPrevWordsIds[justCompletedIndices].tolist())

completeSeqsScores.extend(newTopKScores[justCompletedIndices].squeeze(1).tolist())

incompleteMask = ~justCompletedMask

incompleteIndices = torch.nonzero(incompleteMask, as\_tuple=False).squeeze(1)

k -= len(justCompletedIndices)

if k == 0:

break

kPrevWords = newKPrevWordsIds[incompleteIndices]

topKScores = newTopKScores[incompleteIndices]

finishedSequences = finishedSequences[kIndicesForNextStep[incompleteIndices]]

if step + 1 >= max\_decode\_len:

break

step += 1

i = completeSeqsScores.index(max(completeSeqsScores))

seq = completeSeqs[i]

return seq, None

# This function generates a caption using the transformer decoder and it also returns the attention weights since it uses the

# TransformerDecoderForAttentionViz class in transformerDecoderAttVis.py

def caption\_image\_beam\_search\_transformer\_attention(encoder, decoder, imagePath, wordMap, filename, beamSize=3, max\_decode\_len=51):

# The initial section of this function is adapted from the caption\_image\_beam\_search function hence the same citations apply.

k = beamSize

vocab\_size = len(wordMap)

end\_token\_idx = wordMap['<end>']

img = Image.open(imagePath).convert('RGB')

img = img.resize((256, 256), Image.Resampling.BICUBIC)

img = np.array(img)

if len(img.shape) == 2:

img = np.stack([img, img, img], axis=2)

img = img.transpose(2, 0, 1)

img = img / 255.

img = torch.FloatTensor(img).to(device)

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

transform = transforms.Compose([normalize])

image = transform(img)

image = image.unsqueeze(0)

encoderOut = encoder(image)

encoderDim = encoderOut.size(3)

encoderOutProj = decoder.encoder\_proj(encoderOut.view(1, -1, encoderDim)).permute(1, 0, 2)

num\_pixels = encoderOutProj.size(0)

encoderOutExpanded = encoderOutProj.expand(-1, k, -1) # [num\_pixels, k, embed\_dim]

kPrevWordsIds = torch.full((k, 1), wordMap['<start>'], dtype=torch.long, device=device)

topKScores = torch.zeros(k, 1, device=device) # (k, 1)

seqsAlphas = torch.zeros(k, max\_decode\_len, num\_pixels, device=device)

completeSeqs = list()

completeSeqsAlphas = list()

completeSeqsScores = list()

step = 0

finishedSequences = torch.zeros(k, dtype=torch.bool, device=device)

# This section of the function is also adapted from the caption\_image\_beam\_search function however, modifications have been

# made for the transformer decoder. These modifications are taken from the forwardWithoutTeacherForcing method in

# transformerDecoderAttVis.py for which the Datacamp tutorial (Sarkar, 2025) was used to understand the general structure of the

# transformer decoder whereas PyTorch's Transformer's official GitHub repository linked to its TransformerDecoderLayer

# documentation section was used to implement the CustomerTransformerDecoderLayer. The same citations as TransformerDecoderForAttentionViz

# in transformerDecoderAttVis.py apply to this.

while True:

active = (~finishedSequences).nonzero(as\_tuple=False).squeeze(1)

if len(active) == 0:

break

kPrevWordsIdsActive = kPrevWordsIds[active] # (active\_k, current\_seq\_len)

encoderOutActive = encoderOutExpanded[:, active, :] # (num\_pixels, active\_k, embed\_dim)

embeddingsActive = decoder.embedding(kPrevWordsIdsActive)

embeddingsActive = decoder.pos\_encoding(decoder.dropout(embeddingsActive)) # Use dropout\_layer

tgtActive = embeddingsActive.permute(1, 0, 2)

tgtMask = nn.Transformer.generate\_square\_subsequent\_mask(tgtActive.size(0)).to(device).bool()

currentLayerOutput = tgtActive

allLayerCrossAttentionsForStep = []

for layer\_idx, layer in enumerate(decoder.decoder\_layers):

layer\_output, self\_attn\_weights, cross\_attn\_weights\_current\_layer = layer(

currentLayerOutput,

encoderOutActive,

tgt\_mask=tgtMask,

output\_attentions=True)

currentLayerOutput = layer\_output

allLayerCrossAttentionsForStep.append(cross\_attn\_weights\_current\_layer)

lastTokenOutputActive = currentLayerOutput[-1, :, :] # Shape: [active\_k, embed\_dim]

# Project to vocabulary size to get logits

scoresActive = decoder.fc\_out(lastTokenOutputActive) # [active\_k, vocab\_size]

scoresActive = F.log\_softmax(scoresActive, dim=1)

topKScoresActive = topKScores[active]

scoresActive = topKScoresActive.expand\_as(scoresActive) + scoresActive # (active\_k, vocab\_size)

stackedCrossAttentions = torch.stack(allLayerCrossAttentionsForStep, dim=0)

crossAttnForCurrentToken = stackedCrossAttentions[:, :, :, -1, :]

avgCrossAttentionPerToken = crossAttnForCurrentToken.mean(dim=(0, 2))

if step == 0:

topKScoresNew, topKUnrolledIndices = scoresActive[0].topk(k, 0, True, True)

else:

topKScoresNew, topKUnrolledIndices = scoresActive.view(-1).topk(k, 0, True, True)

prevWordActiveIndices = topKUnrolledIndices / vocab\_size

nextWordIds = topKUnrolledIndices % vocab\_size

prevWordActiveIndices = prevWordActiveIndices.long()

originalKIndicesForNextStep = active[prevWordActiveIndices]

newKPrevWordsIds = torch.cat([kPrevWordsIds[originalKIndicesForNextStep], nextWordIds.unsqueeze(1)], dim=1)

newSeqsALphas = torch.zeros(k, max\_decode\_len, num\_pixels, device=device)

if step > 0:

newSeqsALphas[:, :step, :] = seqsAlphas[originalKIndicesForNextStep, :step, :]

newSeqsALphas[:, step, :] = avgCrossAttentionPerToken[prevWordActiveIndices] # Add current step's alpha

newTopKScores = topKScoresNew.unsqueeze(1)

justCompletedMask = (nextWordIds == end\_token\_idx)

justCompletedIndices = torch.nonzero(justCompletedMask, as\_tuple=False).squeeze(1)

if len(justCompletedIndices) > 0:

completeSeqs.extend(newKPrevWordsIds[justCompletedIndices].tolist())

completeSeqsAlphas.extend(newSeqsALphas[justCompletedIndices].tolist())

completeSeqsScores.extend(newTopKScores[justCompletedIndices].squeeze(1).tolist())

incompleteMask = ~justCompletedMask

incompleteIndices = torch.nonzero(incompleteMask, as\_tuple=False).squeeze(1)

k -= len(justCompletedIndices)

if k == 0:

break

kPrevWordsIds = newKPrevWordsIds[incompleteIndices]

topKScores = newTopKScores[incompleteIndices]

seqsAlphas = newSeqsALphas[incompleteIndices]

finishedSequences = finishedSequences[originalKIndicesForNextStep[incompleteIndices]]

if step + 1 >= max\_decode\_len:

break

step += 1

i = completeSeqsScores.index(max(completeSeqsScores))

seq = completeSeqs[i]

alphas = completeSeqsAlphas[i]

return seq, alphas

def visualize\_att(imagePath, seq, alphas, revWordMap, smooth=True, enc\_image\_size=7):

image = Image.open(imagePath)

image = image.resize([enc\_image\_size \* 24, enc\_image\_size \* 24], Image.Resampling.LANCZOS)

words = [revWordMap[ind] for ind in seq]

num\_cols = 5

num\_rows = int(np.ceil(len(words) / num\_cols))

caption = ' '.join(words)

print(f"Caption: {caption}")

for t in range(len(words)):

if t > 50:

break

plt.subplot(num\_rows, num\_cols, t + 1)

plt.text(0, 1.09, '%s' % (words[t]), color='black', backgroundcolor='white', fontsize=12, va='bottom', transform=plt.gca().transAxes)

plt.imshow(image)

currentAlpha = alphas[t, :]

currentAlpha\_2d = currentAlpha.reshape(enc\_image\_size, enc\_image\_size)

if smooth:

alpha = skimage.transform.pyramid\_expand(currentAlpha\_2d.numpy(), upscale=24, sigma=8)

else:

alpha = skimage.transform.resize(currentAlpha\_2d.numpy(), [enc\_image\_size \* 24, enc\_image\_size \* 24])

if t == 0:

plt.imshow(alpha, alpha=0)

else:

plt.imshow(alpha, alpha=0.8)

plt.set\_cmap(cm.Greys\_r)

plt.axis('off')

plt.subplots\_adjust(hspace=0.05)

plt.show()

def remap\_transformer\_decoder\_keys(old\_state\_dict):

new\_state\_dict = {}

for key, value in old\_state\_dict.items():

if key.startswith('transformer\_decoder.layers.'):

new\_key = key.replace('transformer\_decoder.layers.', 'decoder\_layers.')

elif key.startswith('transformer\_decoder.encoder\_proj.'):

new\_key = key.replace('transformer\_decoder.encoder\_proj.', 'encoder\_proj.')

elif key.startswith('dropout.'):

new\_key = key.replace('dropout.', 'dropout\_layer.')

else:

new\_key = key

new\_state\_dict[new\_key] = value

return new\_state\_dict

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

parser = argparse.ArgumentParser(description='Show, Attend, and Tell - Tutorial - Generate Caption')

parser.add\_argument('--img', '-i', help='path to image')

parser.add\_argument('--model', '-m', help='path to model')

parser.add\_argument('--word\_map', '-wm', help='path to word map JSON')

parser.add\_argument('--beam\_size', '-b', default=5, type=int, help='beam size for beam search')

parser.add\_argument('--dont\_smooth', dest='smooth', action='store\_false', help='do not smooth alpha overlay')

args = parser.parse\_args()

# img = 'cocoDataset/trainval2014/val2014/COCO\_val2014\_000000394240.jpg'

# img = 'cocoDataset/trainval2014/val2014/COCO\_val2014\_000000184791.jpg'

img = 'cocoDataset/trainval2014/val2014/COCO\_val2014\_000000334321.jpg'

# img = 'cocoDataset/trainval2014/val2014/COCO\_val2014\_000000292301.jpg'

# img = 'cocoDataset/trainval2014/val2014/COCO\_val2014\_000000154971.jpg'

# image\_list = ['COCO\_val2014\_000000561100.jpg', 'COCO\_val2014\_000000354533.jpg', 'COCO\_val2014\_000000334321.jpg',

# 'COCO\_val2014\_000000368117.jpg', 'COCO\_val2014\_000000165547.jpg', 'COCO\_val2014\_000000455859.jpg',

# 'COCO\_val2014\_000000290570.jpg', 'COCO\_val2014\_000000017756.jpg', 'COCO\_val2014\_000000305821.jpg',

# 'COCO\_val2014\_000000459374.jpg']

# LSTM

# model = 'bestCheckpoints/mscoco/17-07-2025(lstmDecoder-trainingTF-inferenceNoTF-noFinetuning)/BEST\_checkpoint\_LSTM\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/01-09-2025(lstmNoAttDecoder-trainingTF-inferenceNoTF-noFinetuning)/BEST\_checkpoint\_LSTM\_FinetuningNone\_None\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# training strategies

# model = 'bestCheckpoints/mscoco/06\_20-07-2025(lstmDecoder-trainingNoTF-inferenceNoTF-noFinetuning)/BEST\_checkpoint\_LSTM\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/07\_20-07-2025(transformerDecoder-trainingNoTF-inferenceNoTF-noFinetuning)/BEST\_checkpoint\_Transformer\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/04\_17-07-2025(lstmDecoder-trainingTF-inferenceNoTF-noFinetuning)/BEST\_checkpoint\_LSTM\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/05\_17-07-2025(transformerDecoder-trainingTF-inferenceNoTF-noFinetuning)/BEST\_checkpoint\_Transformer\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# Transformer

# model = 'bestCheckpoints/mscoco/05\_17-07-2025(transformerDecoder-trainingTF-inferenceNoTF-noFinetuning)/BEST\_checkpoint\_Transformer\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/08\_24-07-2025(transformerDecoder-trainingTF-inferenceNoTF-Finetuning5-lr1e4)/BEST\_checkpoint\_Transformer\_Finetuning5\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/10\_28-07-2025(transformerDecoder-trainingTF-inferenceNoTF-Finetuning5-lr1e5-40epochs)/BEST\_checkpoint\_Transformer\_Finetuning5\_1e-05\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/11\_01-08-2025(transformerDecoder-trainingTF-inferenceNoTF-Finetuning5-lr1e6-40epochs)/BEST\_checkpoint\_Transformer\_Finetuning5\_1e-06\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/09\_24-07-2025(transformerDecoder-trainingTF-inferenceNoTF-Finetuning3-lr1e4)/BEST\_checkpoint\_Transformer\_Finetuning3\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/12\_12-08-2025(transformerDecoder-trainingTF-inferenceNoTF-Finetuning1-lr1e6-40epochs)/BEST\_checkpoint\_Transformer\_Finetuning1\_1e-06\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/04-09-2025(transformerAttDecoder-trainingTF-inferenceNoTF-noFinetuning)/BEST\_checkpoint\_TransformerAtt\_FinetuningNone\_None\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/03-09-2025(transformerAttDecoder-trainingTF-inferenceNoTF-Finetuning5-lr1e4)/BEST\_checkpoint\_TransformerAtt\_Finetuning5\_0.0001\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/03-09-2025(transformerAttDecoder-trainingTF-inferenceNoTF-Finetuning3-lr1e4)/BEST\_checkpoint\_TransformerAtt\_Finetuning3\_0.0001\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/10-09-2025(transformerAttDecoder-trainingTF-inferenceNoTF-Finetuning5-lr1e6)/BEST\_checkpoint\_TransformerAtt\_Finetuning5\_1e-06\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# word embeddings

# model = 'bestCheckpoints/mscoco/11\_01-08-2025(transformerDecoder-trainingTF-inferenceNoTF-Finetuning5-lr1e6-40epochs)/BEST\_checkpoint\_Transformer\_Finetuning5\_1e-06\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/14\_31-08-2025(transformerDecoder-trainingTF-Finetuning5-lr1e6-40epochs-wordEmbeddings)/BEST\_checkpoint\_Transformer\_Finetuning5\_1e-06\_word2vec-google-news-300\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/14\_31-08-2025(transformerDecoder-trainingTF-Finetuning5-lr1e6-40epochs-wordEmbeddings)/BEST\_checkpoint\_Transformer\_Finetuning5\_1e-06\_glove-wiki-gigaword-200\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

# model = 'bestCheckpoints/mscoco/20\_26-09-2025(transformerDecoder-trainingTF-Finetuning5-lr1e6-40epochs-wordEmbeddings)/BEST\_checkpoint\_Transformer\_Finetuning5\_1e-06\_word2vec-google-news-300\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

model = 'bestCheckpoints/mscoco/20\_26-09-2025(transformerDecoder-trainingTF-Finetuning5-lr1e6-40epochs-wordEmbeddings)/BEST\_checkpoint\_Transformer\_Finetuning5\_1e-06\_glove-wiki-gigaword-200\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.pth.tar'

word\_map = 'cocoDataset/inputFiles/WORDMAP\_coco\_5\_cap\_per\_img\_5\_min\_word\_freq.json'

beamSize = 1

smooth = False

wordMapFile = os.path.join(dataFolder, 'WORDMAP\_' + dataName + '.json')

with open(wordMapFile, 'r') as j:

wordMap = json.load(j)

checkpoint = torch.load(model, map\_location=device, weights\_only=False)

encoder = Encoder()

encoder.load\_state\_dict(checkpoint['encoder'])

if lstmDecoder is True:

decoder = DecoderWithAttention(attention\_dim=attentionDim, embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), dropout=dropout, device=device)

# decoder = DecoderWithoutAttention(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), dropout=dropout, device=device)

decoder.load\_state\_dict(checkpoint['decoder'])

else:

# decoder = TransformerDecoderForAttentionViz(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), maxLen=maxLen, dropout=dropout, device=device)

# remapped\_decoder\_state\_dict = remap\_transformer\_decoder\_keys(checkpoint['decoder'])

# decoder.load\_state\_dict(remapped\_decoder\_state\_dict)

# decoder = TransformerDecoder(embed\_dim=embDim, decoder\_dim=decoderDim, vocab\_size=len(wordMap), maxLen=maxLen, dropout=dropout, device=device,

# wordMap=None, pretrained\_embeddings\_path=None, fine\_tune\_embeddings=None)

decoder = TransformerDecoder(embed\_dim=200, decoder\_dim=decoderDim, vocab\_size=len(wordMap), maxLen=maxLen, dropout=dropout, device=device,

wordMap=None, pretrained\_embeddings\_path='wordEmbeddings/glove-wiki-gigaword-200.gz', fine\_tune\_embeddings=None)

decoder.load\_state\_dict(checkpoint['decoder'])

decoder = decoder.to(device)

encoder = encoder.to(device)

decoder.eval()

encoder.eval()

revWordMap = {v: k for k, v in wordMap.items()}

if lstmDecoder is True:

seq, alphas = caption\_image\_beam\_search(encoder, decoder, img, wordMap, beamSize)

# seq, alphas = caption\_image\_beam\_search\_noAtt(encoder, decoder, img, wordMap, beamSize)

else:

seq, alphas = caption\_image\_beam\_search\_transformer(encoder, decoder, img, wordMap, beamSize, max\_decode\_len=51)

# seq, alphas = caption\_image\_beam\_search\_transformer\_attention(encoder, decoder, img, wordMap, beamSize, max\_decode\_len=51)

alphas = torch.FloatTensor(alphas)

visualize\_att(img, seq, alphas, revWordMap, smooth)

# 7. makingGraphs.py

import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

import os

import json

import operator

# EDA

def visualizeWordFrequencies(baseDataPath, baseFilename, topN):

wordFreqDict = {}

wordMapPath = os.path.join(baseDataPath, 'WORDMAP\_' + baseFilename + '.json')

with open(wordMapPath, 'r') as j:

wordMap = json.load(j)

specialTokens = {wordMap['<start>'], wordMap['<end>'], wordMap['<pad>'], wordMap['<unk>']}

stopWords = {'a', 'an', 'the', 'and', 'but', 'or', 'on', 'in', 'at', 'with', 'by', 'of', 'for', 'is', 'it', 'its', 'to',

'from', 'as', 'that', 'this', 'he', 'she', 'his', 'her', 'we', 'our', 'they', 'their', 'be', 'are', 'was', 'were'}

revWordMap = {v: k for k, v in wordMap.items()}

for split in ['TRAIN', 'VAL', 'TEST']:

captionsFilePath = os.path.join(baseDataPath, split + '\_CAPTIONS\_' + baseFilename + '.json')

with open(captionsFilePath, 'r') as j:

allCaptionsList = json.load(j)

for captionIds in allCaptionsList:

for wordId in captionIds:

wordString = revWordMap.get(wordId)

if wordId not in specialTokens and wordString and wordString not in stopWords:

wordFreqDict[wordId] = wordFreqDict.get(wordId, 0) + 1

sortedWordFreq = sorted(wordFreqDict.items(), key=lambda item: item[1], reverse=True)

topWordsIdsWithFreqs = sortedWordFreq[:topN]

topWordsIds = [item[0] for item in topWordsIdsWithFreqs]

topWordsFreqs = [item[1] for item in topWordsIdsWithFreqs]

topWordsStrings = []

for wordId in topWordsIds:

wordString = revWordMap.get(wordId)

topWordsStrings.append(wordString)

plt.figure(figsize=(20, 10))

bars = plt.barh(topWordsStrings[::-1], topWordsFreqs[::-1], color='steelblue', alpha=0.9)

for bar in bars:

width = bar.get\_width()

plt.text(width + 50, bar.get\_y() + bar.get\_height()/2, f'{width}', va='center', fontsize=12)

plt.title(f'Top {topN} Most Frequent Words in the Dataset (Excluding Stop Words)', fontsize=18, fontweight='bold', pad=20)

plt.xlabel('Frequency', fontsize=16, labelpad=15)

plt.ylabel('Words', fontsize=16, labelpad=15)

plt.xticks(fontsize=14, rotation=0)

plt.yticks(fontsize=14)

plt.tight\_layout()

plt.grid(axis='x', linestyle='--', alpha=0.6)

outputPath = 'graphs/EDA/wordFrequencies.png'

plt.savefig(outputPath, dpi=300)

plt.show()

def visualizeCaptionLengths(baseDataPath, baseFilename, numBins):

allCaptionLengths = []

for split in ['TRAIN', 'VAL', 'TEST']:

caplensFilePath = os.path.join(baseDataPath, split + '\_CAPLENS\_' + baseFilename + '.json')

with open(caplensFilePath, 'r') as j:

captionLengthsList = json.load(j)

allCaptionLengths.extend(captionLengthsList)

lengthsArray = np.array(allCaptionLengths)

plt.figure(figsize=(12, 7))

plt.hist(lengthsArray, bins=numBins, color='steelblue', edgecolor='black', alpha=0.9)

plt.title('Distribution of Caption Lengths in the Dataset', fontsize=16, fontweight='bold', pad=20)

plt.xlabel('Caption Length (including special tokens)', fontsize=14, labelpad=15)

plt.ylabel('Frequency', fontsize=14, labelpad=15)

meanLength = lengthsArray.mean()

plt.axvline(meanLength, color='red', linestyle='--', linewidth=2, label=f'Mean Length: {meanLength:.2f}')

plt.legend(fontsize=12)

plt.tight\_layout()

plt.grid(axis='y', linestyle='--', alpha=0.6)

outputPath = 'graphs/EDA/captionLengths.png'

plt.savefig(outputPath, dpi=300)

plt.show()

# Results

def plotDecoderLosses(transformerCsvPath, lstmCsvPath):

transformerDf = pd.read\_csv(transformerCsvPath)

lstmDf = pd.read\_csv(lstmCsvPath)

plt.figure(figsize=(12, 7))

plt.plot(transformerDf['epoch'], transformerDf['trainLoss'], label='Transformer Train Loss', color='blue', linestyle='-')

plt.plot(transformerDf['epoch'], transformerDf['valLoss'], label='Transformer Val Loss', color='blue', linestyle='--')

plt.plot(lstmDf['epoch'], lstmDf['trainLoss'], label='LSTM Train Loss', color='red', linestyle='-')

plt.plot(lstmDf['epoch'], lstmDf['valLoss'], label='LSTM Val Loss', color='red', linestyle='--')

plt.title('Training and Validation Loss Comparison: Transformer vs. LSTM Decoder (Flickr8k Dataset)')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.grid(True, linestyle='--', alpha=0.6)

plt.legend()

plt.tight\_layout()

plt.show()

plt.savefig('graphs/lossComparisonTransformerVsLstm.png')

def plotBleu4Scores(lstm\_tf\_csv\_path, transformer\_tf\_csv\_path, lstm\_notf\_csv\_path, transformer\_notf\_csv\_path):

lstm\_tf\_df = pd.read\_csv(lstm\_tf\_csv\_path)

transformer\_tf\_df = pd.read\_csv(transformer\_tf\_csv\_path)

lstm\_notf\_df = pd.read\_csv(lstm\_notf\_csv\_path)

transformer\_notf\_df = pd.read\_csv(transformer\_notf\_csv\_path)

lstm\_tf\_df['epoch'] += 1

transformer\_tf\_df['epoch'] += 1

lstm\_notf\_df['epoch'] += 1

transformer\_notf\_df['epoch'] += 1

lstm\_tf\_df['bleu4'] \*= 100

transformer\_tf\_df['bleu4'] \*= 100

lstm\_notf\_df['bleu4'] \*= 100

transformer\_notf\_df['bleu4'] \*= 100

df\_epoch\_0 = pd.DataFrame([{'epoch': 0, 'bleu4': 0.0}])

lstm\_tf\_df = pd.concat([df\_epoch\_0, lstm\_tf\_df], ignore\_index=True)

transformer\_tf\_df = pd.concat([df\_epoch\_0, transformer\_tf\_df], ignore\_index=True)

lstm\_notf\_df = pd.concat([df\_epoch\_0, lstm\_notf\_df], ignore\_index=True)

transformer\_notf\_df = pd.concat([df\_epoch\_0, transformer\_notf\_df], ignore\_index=True)

lstm\_notf\_df = lstm\_notf\_df[lstm\_notf\_df['epoch'] <= 90]

transformer\_notf\_df = transformer\_notf\_df[transformer\_notf\_df['epoch'] <= 90]

plt.figure(figsize=(10, 6))

plt.plot(lstm\_tf\_df['epoch'], lstm\_tf\_df['bleu4'], label='LSTM + Att (TF)', color='blue', linestyle='-')

plt.plot(transformer\_tf\_df['epoch'], transformer\_tf\_df['bleu4'], label='Transformer (TF)', color='green', linestyle='-')

plt.plot(lstm\_notf\_df['epoch'], lstm\_notf\_df['bleu4'], label='LSTM + Att (No TF)', color='red', linestyle='--')

plt.plot(transformer\_notf\_df['epoch'], transformer\_notf\_df['bleu4'], label='Transformer (No TF)', color='orange', linestyle='--')

plt.title('BLEU-4 Score Comparison Across Decoder Architectures and Training Strategies', fontdict={'fontsize': 14, 'fontweight': 'bold'}, pad=20)

plt.xlabel('Epoch', fontdict={'fontsize': 14}, labelpad=10)

plt.ylabel('BLEU-4 Score', fontdict={'fontsize': 14}, labelpad=10)

plt.grid(True, linestyle='--', alpha=0.6)

plt.legend(fontsize=14)

plt.tight\_layout()

max\_epoch = max(lstm\_tf\_df['epoch'].max(), transformer\_tf\_df['epoch'].max())

plt.xticks(np.arange(0, max\_epoch + 1, 10), fontsize=12)

plt.yticks(np.arange(0, 40, 5), fontsize=12)

plt.savefig('graphs/bleuScoreComparisonTrainingStrategies.png', dpi=300)

plt.show()

def plotFinetunedBleu4Scores(no\_finetune\_csv, ft\_5\_7\_1e4\_20\_csv, ft\_5\_7\_1e5\_40\_csv, ft\_5\_7\_1e6\_40\_csv, ft\_3\_7\_1e4\_20\_csv, ft\_1\_7\_1e6\_40\_csv,

title, output\_filename):

df1 = pd.read\_csv(no\_finetune\_csv)

df2 = pd.read\_csv(ft\_5\_7\_1e4\_20\_csv)

df3 = pd.read\_csv(ft\_5\_7\_1e5\_40\_csv)

df4 = pd.read\_csv(ft\_5\_7\_1e6\_40\_csv)

df5 = pd.read\_csv(ft\_3\_7\_1e4\_20\_csv)

df6 = pd.read\_csv(ft\_1\_7\_1e6\_40\_csv)

df\_list = [df1, df2, df3, df4, df5, df6]

for df in df\_list:

df['epoch'] = df['epoch'] + 1

df['bleu4'] \*= 100

df\_epoch\_0 = pd.DataFrame([{'epoch': 0, 'bleu4': 0.0}])

df = pd.concat([df\_epoch\_0, df], ignore\_index=True)

plt.figure(figsize=(14, 8))

labels = [

'No Fine-tuning',

'Layers 5-7, LR=1×10-4, Patience=20',

'Layers 5-7, LR=1×10-5, Patience=40',

'Layers 5-7, LR=1×10-6, Patience=40',

'Layers 3-7, LR=1×10-4, Patience=20',

'Layers 1-7, LR=1×10-6, Patience=40'

]

colors = ['black', 'blue', 'green', 'orange', 'purple', 'red']

linestyles = ['-', '-', '-', '--', '-', '--']

for df, label, color, linestyle in zip(df\_list, labels, colors, linestyles):

plt.plot(df['epoch'], df['bleu4'], label=label, color=color, linestyle=linestyle, linewidth=2)

plt.title(title, fontsize=16, fontweight='bold', pad=20)

plt.xlabel('Epoch', fontsize=14, labelpad=15)

plt.ylabel('BLEU-4 Score', fontsize=14, labelpad=15)

plt.grid(True, linestyle='--', alpha=0.6)

plt.legend(fontsize=12, loc='upper left')

plt.tight\_layout()

all\_max\_epochs = []

for df in df\_list:

current\_max\_epoch = df['epoch'].max()

all\_max\_epochs.append(current\_max\_epoch)

max\_epoch = max(all\_max\_epochs)

plt.xticks(np.arange(0, max\_epoch + 1, 10), fontsize=12)

plt.yticks(np.arange(25, 40, 1), fontsize=12)

plt.savefig(output\_filename, dpi=300)

plt.show()

visualizeWordFrequencies('cocoDataset/inputFiles', 'coco\_5\_cap\_per\_img\_5\_min\_word\_freq', 20)

visualizeCaptionLengths(baseDataPath='cocoDataset/inputFiles', baseFilename='coco\_5\_cap\_per\_img\_5\_min\_word\_freq', numBins=40)

lstmMetricsTF = 'results/mscoco/17-07-2025(trainingTF-inferenceNoTF-noFinetuning)/metrics-lstmDecoder(trainingTF-inferenceNoTF-noFinetuning).csv'

transformerMetricsTF = 'results/mscoco/17-07-2025(trainingTF-inferenceNoTF-noFinetuning)/metrics-transformerDecoder(trainingTF-inferenceNoTF-noFinetuning).csv'

lstmMetricsNoTF = 'results/mscoco/20-07-2025(trainingNoTF-inferenceNoTF-noFinetuning)/metrics-lstmDecoder(trainingNoTF-inferenceNoTF-noFinetuning).csv'

transformerMetricsNoTF = 'results/mscoco/20-07-2025(trainingNoTF-inferenceNoTF-noFinetuning)/metrics-transformerDecoder(trainingNoTF-inferenceNoTF-noFinetuning).csv'

plotDecoderLosses(transformerMetricsTF, lstmMetricsTF, lstmMetricsNoTF, transformerMetricsNoTF)

plotBleu4Scores(lstmMetricsTF, transformerMetricsTF, lstmMetricsNoTF, transformerMetricsNoTF)

noFinetuned = 'results/mscoco/03\_17-07-2025(trainingTF-inferenceNoTF-noFinetuning)/metrics-transformerDecoder(trainingTF-inferenceNoTF-noFinetuning).csv'

fineTuned1 = 'results/mscoco/05\_24-07-2025(trainingTF-inferenceNoTF-Finetuning5-lr1e4)/metrics-transformerDecoder(trainingTF-inferenceNoTF-Finetuning5).csv'

fineTuned2 = 'results/mscoco/07\_28-07-2025(trainingTF-inferenceNoTF-Finetuning5-lr1e5-40epochs)/metrics-transformerDecoder(trainingTF-inferenceNoTF-Finetuning5-1e-05).csv'

fineTuned3 = 'results/mscoco/08\_01-08-2025(trainingTF-inferenceNoTF-Finetuning5-lr1e6-40epochs)/metrics-transformerDecoder(trainingTF-inferenceNoTF-Finetuning5-1e-06).csv'

fineTuned4 = 'results/mscoco/06\_24-07-2025(trainingTF-inferenceNoTF-Finetuning3-lr1e4)/metrics-transformerDecoder(trainingTF-inferenceNoTF-Finetuning3).csv'

fineTuned5 = 'results/mscoco/09\_12-08-2025(trainingTF-inferenceNoTF-Finetuning1-lr1e6-40epochs)/metrics-transformerDecoder(trainingTF-inferenceNoTF-Finetuning1-1e6).csv'

plotFinetunedBleu4Scores(

noFinetuned,

fineTuned1,

fineTuned2,

fineTuned3,

fineTuned4,

fineTuned5,

title='BLEU-4 Score Comparison for Transformer Decoder with Finetuning ConvNeXt',

output\_filename='graphs/bleuScoreComparisonFinetuning.png'

)