Project Code

Table of Contents

1. Utils

1.1. utils.py

```
import os
import numpy as np
import torch
import h5py
import json
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
    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, 1024, image_size/32, image_size/32)
        out = self.adaptive_pool(out)  # (batch_size, 1024, encoded_image_size,
encoded_image_size)
        out = out.permute(0, 2, 3, 1)  # (batch_size, encoded_image_size, encoded_image_size,
1024)
        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


    # 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)
```

```
        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)
        self.register_buffer('pe', pe)

    def forward(self, x):
        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)  # [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)
        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

    # This section of the function was generated using Gemini. It consolidates the logic of
_sa_block,
    # _mha_block, and _ff_block from PyTorch's Transformer's official GitHub repository into a
single forward method

    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

            # This section of the function was generated using Gemini. It computes the average
cross-attention weights
            # across all layers for the current word and updates the alphas tensor accordingly

            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
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)
        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)
        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]):
            imgCaps = allcaps[j].tolist()
            imgCaptions = []
            for c_list in imgCaps:
               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 = []
         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)


      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.ht
ml
# 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:
            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)
            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)
            # 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]):
                imgCaps = allcaps[j].tolist()
                imgCaptions = []
                for c_list in imgCaps:
                    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 = []
            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]):
                imgCaps = allcaps[j].tolist()
                imgCaptions = []
                for c_list in imgCaps:
                    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 = []
            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)


        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))

    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, :, :] # [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)
```

```
    # This section of the function was generated using Gemini. It computes the average
cross-attention weights
    # across all layers for the current word and updates the alphas tensor accordingly

    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]

    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_mi
n_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.t
ar'
    # 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.t
ar'

    # 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.t
ar'
    # 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_i
mg_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$\\times 10^{-4}$, Patience=20',
        'Layers 5-7, LR=1$\\times 10^{-5}$, Patience=40',
        'Layers 5-7, LR=1$\\times 10^{-6}$, Patience=40',
        'Layers 3-7, LR=1$\\times 10^{-4}$, Patience=20',
        'Layers 1-7, LR=1$\\times 10^{-6}$, 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=18, fontweight='bold', pad=20)
    plt.xlabel('Epoch', fontsize=16, labelpad=15)
    plt.ylabel('BLEU-4 Score', fontsize=16, 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=14)
    plt.yticks(np.arange(25, 40, 1), fontsize=14)
    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/resultsGraphs/bleuScoreComparisonFinetuning.png'
)
```