**Practical Assignment**

**Objective: - Image caption generator**

An Image caption generator combines both computer vision and natural language processing techniques to analyze and identify the context of an image and describe them accordingly in natural human languages

**Dataset Link: -**

Link :-<https://www.kaggle.com/adityajn105/flickr8k>

**Task: -** Create a Web Application using FASTAPI. Use the end user should be able to upload an image and get results with the captions.

**Assignment Submission: -** Only submit the GitHub Link. Create a proper Readme documentation.

**Imports Libraries**

**import** os

**import** json

**from** collections **import** Counter

**import** numpy **as** np

**import** pandas **as** pd

**import** spacy

**import** matplotlib.pyplot **as** plt

**from** torch.nn.utils.rnn **import** pad\_sequence

**from** torch.utils.data **import** DataLoader, Dataset

**import** torchvision.transforms **as** T

**from** PIL **import** Image

**import** torch

**import** torch.nn **as** nn

**import** torch.nn.functional **as** F

**import** torch.optim **as** optim

**import** torchvision.models **as** models

**from** tqdm **import** tqdm

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

**Building the Vocabulary**

* we use spacy as a nlp library for efficient tokenization
* word freqencies is taken into account because words that appear very infrequently in the corpus are often typos, rare terms, or irrelevant details. Including these words in the vocabulary can introduce noise and make the model unnecessarily complex.

**class** Vocabulary:

*# tokenizer*

spacy\_eng **=** spacy**.**load("en\_core\_web\_sm")

**def** \_\_init\_\_(self, freq\_threshold):

*# setting the pre-reserved tokens int to string tokens*

self**.**itos **=** {0: "<PAD>", 1: "<SOS>", 2: "<EOS>", 3: "<UNK>"}

*# string to int tokens*

*# its reverse dict self.itos*

self**.**stoi **=** {v: k **for** k, v **in** self**.**itos**.**items()}

self**.**freq\_threshold **=** freq\_threshold

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

**return** len(self**.**itos)

@staticmethod

**def** tokenize(text):

**return** [token**.**text**.**lower() **for** token **in** Vocabulary**.**spacy\_eng**.**tokenizer(text)]

**def** build\_vocab(self, sentence\_list):

frequencies **=** Counter()

*# staring index 4*

idx **=** 4

**for** sentence **in** sentence\_list:

**for** word **in** self**.**tokenize(sentence):

frequencies[word] **+=** 1

*# add the word to the vocab if it reaches minum frequecy threshold*

**if** frequencies[word] **==** self**.**freq\_threshold:

self**.**stoi[word] **=** idx

self**.**itos[idx] **=** word

idx **+=** 1

**def** numericalize(self, text):

"""For each word in the text corresponding index token for that word form the vocab built as list"""

tokenized\_text **=** self**.**tokenize(text)

**return** [

self**.**stoi[token] **if** token **in** self**.**stoi **else** self**.**stoi["<UNK>"]

**for** token **in** tokenized\_text

]

## Dataset and Dataloader

**class** FlickrDataset(Dataset):

**def** \_\_init\_\_(self, root\_dir, caption\_file, transform**=None**, freq\_threshold**=**5):

self**.**root\_dir **=** root\_dir

self**.**df **=** pd**.**read\_csv(caption\_file)

self**.**transform **=** transform

*# Get image and caption colum from the dataframe*

self**.**imgs **=** self**.**df["image"]

self**.**captions **=** self**.**df["caption"]

*# Initialize vocabulary and build vocab*

self**.**vocab **=** Vocabulary(freq\_threshold)

self**.**vocab**.**build\_vocab(self**.**captions**.**tolist())

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

**return** len(self**.**df)

**def** \_\_getitem\_\_(self, idx):

caption **=** self**.**captions[idx]

img\_name **=** self**.**imgs[idx]

img\_location **=** os**.**path**.**join(self**.**root\_dir, img\_name)

img **=** Image**.**open(img\_location)**.**convert("RGB")

*# apply the transfromation to the image*

**if** self**.**transform **is** **not** **None**:

img **=** self**.**transform(img)

*# numericalize the caption text*

caption\_vec **=** []

caption\_vec **+=** [self**.**vocab**.**stoi["<SOS>"]]

caption\_vec **+=** self**.**vocab**.**numericalize(caption)

caption\_vec **+=** [self**.**vocab**.**stoi["<EOS>"]]

**return** img, torch**.**tensor(caption\_vec)

**def** get\_data\_loader(dataset, vocab, batch\_size, shuffle**=False**, num\_workers**=**1):

pad\_idx **=** vocab**.**stoi["<PAD>"]

collate\_fn **=** CapsCollate(pad\_idx**=**pad\_idx, batch\_first**=True**)

data\_loader **=** DataLoader(

dataset**=**dataset,

batch\_size**=**batch\_size,

shuffle**=**shuffle,

num\_workers**=**num\_workers,

collate\_fn**=**collate\_fn,

)

**return** data\_loader

**class** CapsCollate:

**def** \_\_init\_\_(self, pad\_idx, batch\_first**=False**):

self**.**pad\_idx **=** pad\_idx

self**.**batch\_first **=** batch\_first

**def** \_\_call\_\_(self, batch):

imgs **=** [item[0]**.**unsqueeze(0) **for** item **in** batch]

imgs **=** torch**.**cat(imgs, dim**=**0)

targets **=** [item[1] **for** item **in** batch]

targets **=** pad\_sequence(

targets, batch\_first**=**self**.**batch\_first, padding\_value**=**self**.**pad\_idx

)

**return** imgs, targets

BASE\_DIRECTORY **=** "../input/flickr8k"

### Helper functions to plot the Tensor image

**def** read\_image(path):

**return** Image**.**open(path)

**def** show\_image(img, title**=None**):

"""Imshow for Tensor."""

*# unnormalize*

img[0] **=** img[0] **\*** 0.229

img[1] **=** img[1] **\*** 0.224

img[2] **=** img[2] **\*** 0.225

img[0] **+=** 0.485

img[1] **+=** 0.456

img[2] **+=** 0.406

img **=** img**.**numpy()**.**transpose((1, 2, 0))

plt**.**imshow(img)

**if** title **is** **not** **None**:

plt**.**title(title)

plt**.**pause(0.001) *# pause a bit so that plots are updated*

**Split Dataset to Train, Val and Test**

BATCH\_SIZE **=** 32

NUM\_WORKER **=** 4

train\_transforms **=** T**.**Compose(

[

T**.**Resize((256, 256)),

T**.**RandomCrop((224, 224)),

T**.**RandomHorizontalFlip(p**=**0.5),

T**.**RandomRotation(degrees**=**15),

T**.**ColorJitter(brightness**=**0.2, contrast**=**0.2, saturation**=**0.2, hue**=**0.2),

T**.**ToTensor(),

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

]

)

transforms **=** T**.**Compose(

[

T**.**Resize((224, 224)),

T**.**ToTensor(),

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

]

)

dataset **=** FlickrDataset(

root\_dir**=**BASE\_DIRECTORY **+** "/Images",

caption\_file**=**BASE\_DIRECTORY **+** "/captions.txt",

transform**=**transforms,

)

train\_dataset, val\_dataset, test\_dataset **=** torch**.**utils**.**data**.**random\_split(

dataset, [30000, 5000, 5455]

)

*# writing the dataloader*

train\_dataloader **=** get\_data\_loader(

dataset**=**train\_dataset,

vocab**=**dataset**.**vocab,

batch\_size**=**BATCH\_SIZE,

num\_workers**=**NUM\_WORKER,

shuffle**=True**,

)

val\_dataloader **=** get\_data\_loader(

dataset**=**val\_dataset,

vocab**=**dataset**.**vocab,

batch\_size**=**BATCH\_SIZE,

num\_workers**=**NUM\_WORKER,

)

test\_dataloader **=** get\_data\_loader(

dataset**=**test\_dataset,

vocab**=**dataset**.**vocab,

batch\_size**=**BATCH\_SIZE,

num\_workers**=**NUM\_WORKER,

)

vocab\_size **=** len(dataset**.**vocab)

print(vocab\_size)

device **=** torch**.**device("cuda:0" **if** torch**.**cuda**.**is\_available() **else** "cpu")

device

## Defining the Model Architecture

### CNN Encoder

Resnet is used for **Feature Extraction** with freezing weights (Transfer Learning)

**class** EncoderCNN(nn**.**Module):

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

super(EncoderCNN, self)**.**\_\_init\_\_()

resnet **=** models**.**resnet50(pretrained**=True**)

**for** param **in** resnet**.**parameters():

param**.**requires\_grad\_(**False**)

modules **=** list(resnet**.**children())[:**-**2]

self**.**resnet **=** nn**.**Sequential(**\***modules)

**def** forward(self, images):

features **=** self**.**resnet(images)

features **=** features**.**permute(0, 2, 3, 1)

features **=** features**.**view(features**.**size(0), **-**1, features**.**size(**-**1))

**return** features

### Attention Mechanism

**class** Attention(nn**.**Module):

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

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

self**.**attention\_dim **=** attention\_dim

self**.**W **=** nn**.**Linear(decoder\_dim, attention\_dim)

self**.**U **=** nn**.**Linear(encoder\_dim, attention\_dim)

self**.**A **=** nn**.**Linear(attention\_dim, 1)

**def** forward(self, features, hidden\_state):

u\_hs **=** self**.**U(features)

w\_ah **=** self**.**W(hidden\_state)

combined\_states **=** torch**.**tanh(

u\_hs **+** w\_ah**.**unsqueeze(1)

) *# (batch\_size,num\_layers,attemtion\_dim)*

attention\_scores **=** self**.**A(combined\_states) *# (batch\_size,num\_layers,1)*

attention\_scores **=** attention\_scores**.**squeeze(2) *# (batch\_size,num\_layers)*

alpha **=** F**.**softmax(attention\_scores, dim**=**1) *# (batch\_size,num\_layers)*

attention\_weights **=** features **\*** alpha**.**unsqueeze(

2

) *# (batch\_size,num\_layers,features\_dim)*

attention\_weights **=** attention\_weights**.**sum(dim**=**1) *# (batch\_size,num\_layers)*

**return** alpha, attention\_weights

### RNN Decoder

*# Attention Decoder*

**class** DecoderRNN(nn**.**Module):

**def** \_\_init\_\_(

self,

embed\_size,

vocab\_size,

attention\_dim,

encoder\_dim,

decoder\_dim,

drop\_prob**=**0.3,

):

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

*# save the model param*

self**.**vocab\_size **=** vocab\_size

self**.**attention\_dim **=** attention\_dim

self**.**decoder\_dim **=** decoder\_dim

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

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

self**.**init\_h **=** nn**.**Linear(encoder\_dim, decoder\_dim)

self**.**init\_c **=** nn**.**Linear(encoder\_dim, decoder\_dim)

self**.**lstm\_cell **=** nn**.**LSTMCell(embed\_size **+** encoder\_dim, decoder\_dim, bias**=True**)

self**.**f\_beta **=** nn**.**Linear(decoder\_dim, encoder\_dim)

self**.**fcn **=** nn**.**Linear(decoder\_dim, vocab\_size)

self**.**drop **=** nn**.**Dropout(drop\_prob)

**def** forward(self, features, captions):

*# vectorize the caption*

embeds **=** self**.**embedding(captions)

*# Initialize LSTM state*

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

*# get the seq length to iterate*

seq\_length **=** len(captions[0]) **-** 1 *# Exclude the last one*

batch\_size **=** captions**.**size(0)

num\_features **=** features**.**size(1)

preds **=** torch**.**zeros(batch\_size, seq\_length, self**.**vocab\_size)**.**to(device)

alphas **=** torch**.**zeros(batch\_size, seq\_length, num\_features)**.**to(device)

**for** s **in** range(seq\_length):

alpha, context **=** self**.**attention(features, h)

lstm\_input **=** torch**.**cat((embeds[:, s], context), dim**=**1)

h, c **=** self**.**lstm\_cell(lstm\_input, (h, c))

output **=** self**.**fcn(self**.**drop(h))

preds[:, s] **=** output

alphas[:, s] **=** alpha

**return** preds, alphas

**def** generate\_caption(self, features, max\_len**=**20, vocab**=None**):

*# Inference part*

*# Given the image features generate the captions*

batch\_size **=** features**.**size(0)

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

alphas **=** []

*# starting input*

word **=** torch**.**tensor(vocab**.**stoi["<SOS>"])**.**view(1, **-**1)**.**to(device)

embeds **=** self**.**embedding(word)

captions **=** []

**for** i **in** range(max\_len):

alpha, context **=** self**.**attention(features, h)

*# store the apla score*

alphas**.**append(alpha**.**cpu()**.**detach()**.**numpy())

lstm\_input **=** torch**.**cat((embeds[:, 0], context), dim**=**1)

h, c **=** self**.**lstm\_cell(lstm\_input, (h, c))

output **=** self**.**fcn(self**.**drop(h))

output **=** output**.**view(batch\_size, **-**1)

*# select the word with most val*

predicted\_word\_idx **=** output**.**argmax(dim**=**1)

*# save the generated word*

captions**.**append(predicted\_word\_idx**.**item())

*# end if <EOS detected>*

**if** vocab**.**itos[predicted\_word\_idx**.**item()] **==** "<EOS>":

**break**

*# send generated word as the next caption*

embeds **=** self**.**embedding(predicted\_word\_idx**.**unsqueeze(0))

*# covert the vocab idx to words and return sentence*

**return** [vocab**.**itos[idx] **for** idx **in** captions], alphas

**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

### The Model (Encoder Decoder)

**class** EncoderDecoder(nn**.**Module):

**def** \_\_init\_\_(

self,

embed\_size,

vocab\_size,

attention\_dim,

encoder\_dim,

decoder\_dim,

drop\_prob**=**0.3,

):

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

self**.**encoder **=** EncoderCNN()

self**.**decoder **=** DecoderRNN(

embed\_size**=**embed\_size,

vocab\_size**=**len(dataset**.**vocab),

attention\_dim**=**attention\_dim,

encoder\_dim**=**encoder\_dim,

decoder\_dim**=**decoder\_dim,

)

**def** forward(self, images, captions):

features **=** self**.**encoder(images)

outputs **=** self**.**decoder(features, captions)

**return** outputs

embed\_size **=** 300

vocab\_size **=** len(dataset**.**vocab)

attention\_dim **=** 256

encoder\_dim **=** 2048

decoder\_dim **=** 512

learning\_rate **=** 3e-4

**Initializing The Model**

* Criterion: **Cross Entropy Loss**
* Optimizer: **Adam**

model **=** EncoderDecoder(

embed\_size**=**300,

vocab\_size**=**len(dataset**.**vocab),

attention\_dim**=**256,

encoder\_dim**=**2048,

decoder\_dim**=**512,

)**.**to(device)

criterion **=** nn**.**CrossEntropyLoss(ignore\_index**=**dataset**.**vocab**.**stoi["<PAD>"])

optimizer **=** optim**.**Adam(model**.**parameters(), lr**=**learning\_rate)

**Saving Model Checkpoints**

**def** save\_checkpoint(state, filename**=**"/kaggle/working/checkpoint.pth.tar"):

torch**.**save(state, filename)

**def** load\_checkpoint(model, optimizer, filename**=**"/kaggle/working/checkpoint.pth.tar"):

checkpoint **=** torch**.**load(filename)

model**.**load\_state\_dict(checkpoint["model\_state\_dict"])

optimizer**.**load\_state\_dict(checkpoint["optimizer\_state\_dict"])

epoch **=** checkpoint["epoch"]

loss **=** checkpoint["loss"]

**return** model, optimizer, epoch, loss

**def** save\_epoch\_metrics(epoch\_metrics\_dict, filename):

**with** open(filename, "w") **as** f:

json**.**dump(epoch\_metrics\_dict, f)

**def** load\_epoch\_metrics(filename):

**with** open(filename, "r") **as** f:

epoch\_metrics\_dict **=** json**.**load(f)

**return** epoch\_metrics\_dict

epoch\_metrics\_dict **=** {}

**def** reverse\_process\_caption(caption):

ls **=** [dataset**.**vocab**.**itos[idx**.**item()] **for** idx **in** caption]

ls **=** [word **for** word **in** ls **if** word **not** **in** ["<PAD>", "<SOS>"]]

**return** " "**.**join(ls)

**Training and Evaluating Model on validation set with Hyperparameter Tuning**

resume\_training **=** **False**

start\_epoch **=** 1

**if** resume\_training:

start\_epoch, best\_loss **=** load\_checkpoint("/kaggle/working/checkpoint.pth.tar")

loaded\_dict **=** load\_epoch\_metrics("/kaggle/working/epoch\_metrics.json")

num\_epochs **=** 25

print\_every **=** 938

**for** epoch **in** range(start\_epoch, num\_epochs **+** 1):

progress\_bar **=** tqdm(

enumerate(iter(train\_dataloader)), total**=**len(train\_dataloader), leave**=False**

)

**for** idx, (image, captions) **in** progress\_bar:

image, captions **=** image**.**to(device), captions**.**to(device)

*# Zero the gradients.*

optimizer**.**zero\_grad()

*# Feed forward*

outputs, attentions **=** model(image, captions)

*# Calculate the batch loss.*

targets **=** captions[:, 1:]

loss **=** criterion(outputs**.**view(**-**1, vocab\_size), targets**.**reshape(**-**1))

*# Backward pass.*

loss**.**backward()

**if** epoch **not** **in** epoch\_metrics\_dict:

epoch\_metrics\_dict[epoch] **=** {}

**if** (idx **+** 1) **%** print\_every **==** 0:

epoch\_metrics\_dict[epoch]["train\_loss"] **=** loss**.**item()

*# Update the parameters in the optimizer.*

optimizer**.**step()

progress\_bar**.**set\_description(f"Epoch: {epoch} Loss: {loss**.**item():.5f}")

progress\_bar**.**close()

*# Validation loop*

model**.**eval() *# Set the model to evaluation mode*

val\_loss **=** 0

all\_references **=** []

all\_hypotheses **=** []

**with** torch**.**no\_grad(): *# Disable gradient computation*

**for** idxx, (image, captions) **in** enumerate(val\_dataloader):

image, captions **=** image**.**to(device), captions**.**to(device)

**for** caption **in** captions:

all\_references**.**append(reverse\_process\_caption(caption)**.**split())

outputs, attentions **=** model(image, captions)

targets **=** captions[:, 1:]

loss **=** criterion(outputs**.**view(**-**1, vocab\_size), targets**.**reshape(**-**1))

val\_loss **+=** loss**.**item()

**for** i **in** range(image**.**shape[0]):

features **=** model**.**encoder(image[i : i **+** 1]**.**to(device))

caps, alphas **=** model**.**decoder**.**generate\_caption(

features, vocab**=**dataset**.**vocab

)

caption **=** " "**.**join(caps)

all\_hypotheses**.**append(caption**.**split())

val\_loss **/=** len(val\_dataloader)

epoch\_metrics\_dict[epoch]["val\_loss"] **=** val\_loss

print(f"Validation Loss: {val\_loss}")

*# Compute BLEU score*

bleu\_score **=** corpus\_bleu(all\_references, all\_hypotheses)

print(f"Validation BLEU score: {bleu\_score}")

epoch\_metrics\_dict[epoch]["bleu\_score"] **=** bleu\_score

model**.**train() *# Set the model back to training mode*

*# save the latest model*

save\_checkpoint(

{

"epoch": epoch **+** 1,

"model\_state\_dict": model**.**state\_dict(),

"optimizer\_state\_dict": optimizer**.**state\_dict(),

"loss": loss**.**item(),

},

filename**=**"/kaggle/working/checkpoint.pth.tar",

)

save\_epoch\_metrics(epoch\_metrics\_dict, "/kaggle/working/epoch\_metrics.json")

**Visualizing Train Loss, Validation Loss and Bleu Score**

*# Extract the metrics*

epochs **=** list(epoch\_metrics\_dict**.**keys())

train\_losses **=** [epoch\_metrics\_dict[epoch]["train\_loss"] **for** epoch **in** epochs]

val\_losses **=** [epoch\_metrics\_dict[epoch]["val\_loss"] **for** epoch **in** epochs]

bleu\_scores **=** [epoch\_metrics\_dict[epoch]["bleu\_score"] **for** epoch **in** epochs]

*# Plot the losses*

plt**.**figure(figsize**=**(10, 4))

plt**.**subplot(1, 2, 1)

plt**.**plot(epochs, train\_losses, label**=**"Train Loss")

plt**.**plot(epochs, val\_losses, label**=**"Val Loss")

plt**.**xlabel("Epoch")

plt**.**ylabel("Loss")

plt**.**legend()

*# Plot the BLEU scores*

plt**.**subplot(1, 2, 2)

plt**.**plot(epochs, bleu\_scores, label**=**"BLEU Score")

plt**.**xlabel("Epoch")

plt**.**ylabel("BLEU Score")

plt**.**legend()

plt**.**tight\_layout()

plt**.**show()

*# generate caption*

**def** get\_caps\_from(features\_tensors):

*# generate the caption*

model**.**eval()

**with** torch**.**no\_grad():

features **=** model**.**encoder(features\_tensors**.**to(device))

caps, alphas **=** model**.**decoder**.**generate\_caption(features, vocab**=**dataset**.**vocab)

caption **=** " "**.**join(caps)

*# show\_image(features\_tensors[0], title=caption)*

**return** caps, alphas

**def** get\_caps\_from\_show(features\_tensors):

*# generate the caption*

model**.**eval()

**with** torch**.**no\_grad():

features **=** model**.**encoder(features\_tensors**.**to(device))

caps, alphas **=** model**.**decoder**.**generate\_caption(features, vocab**=**dataset**.**vocab)

caption **=** " "**.**join(caps)

show\_image(features\_tensors[0], title**=**caption)

**return** caps, alphas

**Maximum Test BLEU Score**

max\_bleu\_score **=** 0

bleu\_scores **=** []

**for** idx, (images, captions) **in** enumerate(iter(test\_dataloader)):

all\_references **=** []

all\_hypotheses **=** []

**for** caption **in** captions:

all\_references**.**append(reverse\_process\_caption(caption)**.**split())

**for** i **in** range(images**.**shape[0]):

img **=** images[0]**.**detach()**.**clone()

img1 **=** images[0]**.**detach()**.**clone()

caps, alphas **=** get\_caps\_from(img**.**unsqueeze(0))

all\_hypotheses**.**append(caps)

bleu\_score **=** corpus\_bleu(all\_references, all\_hypotheses)

bleu\_scores**.**append(bleu\_score)

b **=** np**.**array(bleu\_scores)

print(np**.**mean(bleu\_scores))

### Test Images

**for** i **in** range(20, 30):

data\_iter **=** iter(test\_dataloader)

images, caption **=** next(data\_iter)

img **=** images[i]**.**detach()**.**clone()

caps, alphas **=** get\_caps\_from\_show(img**.**unsqueeze(0))