Image Caption with Attention

0. Introduction

다층의 퍼셉트론을 통해 Deep Learning 으로 새로운 도약을 시작한 이후, 이미지 및 자연어 분야는 개별적이나 상호적인 보완을 통해 끊임없이 발전 중이다. 기존의 방법에서 새로운 방법을 시도하기도 하고, 실험을 통해 보다 효율적인 방법을 찾아내면서 각 분야는 높은 성능을 나타내기 시작했다. 그 과정에서 자연스럽게 다양한 분야의 데이터들을 융합해서 결과를 나타낼 수 있는 Task들이 부상하기 시작하였다.

Multi-Modal task 중 Cross Modal에 해당되는 Image caption은 자연어와 이미지를 동시에 활용하게 되는 task이다. 이미지와 자연어 Task에서 높은 성능을 나타냈음에도 불구하고, 이미지 caption 분야가 태동한 이후, 성능은 10% 남짓한 성능을 나타냈다. 이는 각각의 데이터가 충분한 정보를 담고 있음에도 불구하고 두 가지 이상의 modal 정보를 포함했을 때 담고 있는 정보들을 충분히 활용하지 못한다는 의미를 담고 있다.

SHOW AND TELL을 기반으로 Encoder에서 Image를 처리하는 CNN, Decoder에서 자연어를 처리하는 RNN 기반으로 (2015) 이미지 캡션 분야가 본격적으로 시작되었고, 이후 Show, Attend and Tell에서 Show and Tell을 기반으로 Decoder 부분에 Attention을 도입하여 설명문 생성 시 이미지의 어느 부분을 볼지 학습할 수 있도록 발전되었다. 동일하게 Attention을 활용하나 조금을 다른 방식으로 Image Caption에 Detection을 결합한 DenseCap 역시 나타났다. 현재는 CLIP(V1, V2), BLIP(V1, V2) 등이 나타나 효율적인 성능 개선으로 SOTA를 기록하고 있다.

1. 활용 라이브러리

[후첨1] 참조

참고: 기본적으로 colab 및 kaggle notebook을 활용하여 진행하였다.
학습 과정에서의 gpu 사용을 위해 kaggle notebook으로 전환해 진행하였다.

2. EDA 및 전처리

- 전체 EDA 및 전처리를 작성하는 것이 아닌 주요 처리 내용에 대해 포함하려한다.

1) 이미지 read & import

```
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

해당 파일의 경로 및 파일명을 출력하게 된다.
추가적으로 dataset에 존재하는 이미지 파일 개수를 확인하였다.

```
#images = '/content/drive/MyDrive/images' # colab
images='/kaggle/input/flickr8k/Images' # kaggle note

all_imgs = glob.glob(images + '/*.jpg',recursive=True)
print("The total images present in the dataset: {}".format(len(all_imgs)))
```

총 데이터의 수는 8091임을 확인하였다.

2) txt 데이터 데이터프레임 전환

기존 dataset은 txt 파일로 구분자 ‘,’를 통해 image와 해당 image에 대한 caption을 포함한다. pd.read_csv()를 통해 기존 dataset을 df로 저장해주었고, value_counts()를 통해 확인해본 결과 각 image에 대한 caption의 개수는 총 5개인 것으로 확인하였다.

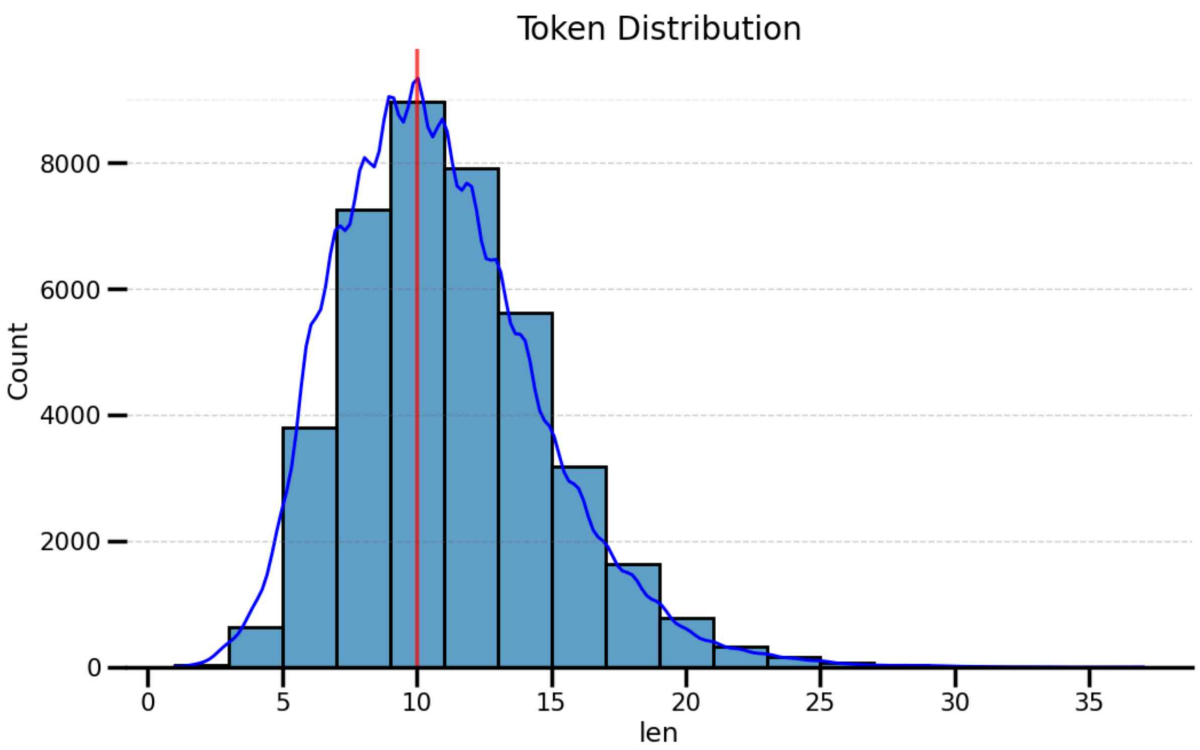
3) Text 전처리

[0. Tokenizing]

Text 전처리를 위해서는 해당 Text 데이터에 대한 문법적인 변환과 token이 선행화되어야 한다. 해당 데이터가 이미지와 영어 text를 기반으로 하기 때문에 영어 tokenizing 방법 중 효율적이라고 판단되는 방식을 선택하였다. 대표적으로 NLTK, Spacy를 활용한 방법과 Keras를 활용한 방식이 있다. NLTK는 정규화 기반으로 속도가 빠르며, Spacy는 NLTK 및 keras에 비해 처리 속도는 낮으나 다른 성능 측면에서 우수한 성능을 나타내고 있다. 전처리 방식으로 Keras 방식을 선택하여 활용하였는데, 정규화 기반으로 속도가 빠르다는 이점과, 모든 토큰을 소문자로 변경해주고, token화 과정에서도 “ ‘ “를 고려해 token화를 수행하기 때문에 해당 방식이 이미지 캡션이라는 task에 더 적합하다 판단되었다.

```
from tensorflow.keras.preprocessing.text import text_to_word_sequence

def token_word(string):
  '''
  token화 후 토큰들을 list로 return하는 함수 정의
  '''
  str_list = text_to_word_sequence(string)
  return str_list
```

전체 caption dataset에 적용시키기 위해 token_word()라는 함수를 정의해주었고, apply()를 통해 기존의 caption feature에 적용해 token_list라는 새로운 feature를 형성해주었다. 각각의 토큰이 어떠한 분포를 갖고 있는지 시각화를 위해 각 caption을 기반으로 len()을 적용해 distribution을 확인해보았다.



빨간색으로 나타난 직선은 중위수를 표시한 부분이다.

len()을 통해 확인한 부분은 각 caption에 대한 token 개수에 대한 부분이다.
추가적으로 Counter()를 통해 저장된 배열에서 각 원소(token)들이 몇 번씩 저장되었는지 확인함을 통해 각 token 별 사용 빈도수를 파악할 수 있다.

```
# Counter를 통해 중복된 데이터가 저장된 배열에서 각 원소가 몇 번씩 저장되었는지 확인
from collections import Counter

combined_token_list = [i for tok_list in df['token_list'] for i in tok_list] # 하나의 리스트에 담음

print(len(combined_token_list))
# output : 437665

frequent = Counter(combined_token_list).most_common()
#print(f'unique한 token의 개수 : {len(words)}')

print(len(frequent))
# output : 8478

frequent_df = pd.DataFrame(frequent)
frequent_df.T
```

collection의 Counter를 불러왔고, 이전에 token화한 feature인 token_list를 불러와 하나의 리스트에 담아 combined_token_list에 저장하였다. 총 길이는 437665였고, Counter 함수의 most_common()을 통해 fequent라는 변수에 담아주었다. 개별적인 token의 개수를 len(frequent)를 통해 확인한 결과 총 8478개의 token이 있는 것으로 확인되었다.
(해당 내용은 단어의 전처리인 replace()처리 된 이후의 결과를 반영하고 있음)

기본적인 결과는 다음과 같이 나타난다.

	0	1	2	3	4	5	6	7	8	9	...	8468	8469	8470	8471
0	a	in	the	on	is	and	dog	with	man	of	...	swatting	containig	rainstorm	breezeway
1	62992	18986	18419	10745	9345	8862	8138	7765	7274	6723	...	1	1	1	1

2 rows × 8478 columns

해당 내용에 추후 import 하는 NLTK의 stopword를 반영해 removed된 frequent2를 형성해주었고, 개별 token 개수는 8390으로 나타났다.

```
remove_stopwords = [x for x in combined_token_list if x not in stop_words]
frequent2 = Counter(remove_stopwords).most_common()
print(len(frequent2))
# output : 8390

frequent_df2 = pd.DataFrame(frequent2)
# 불용어 파악 위해 리스트 csv 저장
frequent_df2.to_csv('stopword_prepare.csv')
```

Counter()를 사용하여 token count한 내용을 기반으로 데이터프레임을 형성해 stopword_prepare.csv에 저장해주었고, 이를 기반으로 전처리가 필요한 token들을 파악하였다. 이를 기반으로 전처리의 앞단에 caption에 선적용시켜주었다.

```
def word_prep(str):
  str = str.replace('coca','')
  str = str.replace('cola','coca-cola')
  str = str.replace('hapily','happily')
  str = str.replace('playy','play')
  str = str.replace('fronmt','front')
  str = str.replace('litle','little')
  str = str.replace('offf','off')
  str = str.replace('giong','going')
  str = str.replace('rakes','lakes')
  str = str.replace('sidwalk','sidewalk')
  str = str.replace('mommy','mom')
  str = str.replace('woamn','woman')
  str = str.replace('feamle','female')
  str = str.replace('matchin','matching')
  str = str.replace('jello','hello')
  str = str.replace('silohuetted','silhouetted')
  str = str.replace('redhead','red head')
  str = str.replace('ypoung','young')
  str = str.replace('unner','under')
  return str

df['caption'] = df['caption'].apply(word_prep)
```

[1. Dictionary 생성 및 Stopwords 처리]

단어 사전을 정의하기 위해 우선 Tokenizer에 전처리 filter와 소문자 반환, oov words를 대체하는 'UNK'를 정의, dict의 크기를 제한하기 위해 word_dict_limit이라는 변수를 통해 초기화시켜 주었다

1) 단어 사전 부분 위한 준비

```
word_dict_limit = 5000  # dict 크기 제한시 사용 # None 가능

tokenizer = Tokenizer(num_words = word_dict_limit+1,
                      filters = '!"#$%^&*()_+.,:;-?/~`{}[]|\=@ ',
                      lower = True, # 소문자 반환
                      char_level = False, # True일 경우 모든 문자 토큰으로 처리
                      oov_token = 'UNK')
```

2) stopword 처리 위한 준비

```
import nltk
from nltk.corpus import stopwords

# 불용어 다운
nltk.download('stopwords')

# 불용어 집합에 담음
stop_words = set(stopwords.words('english'))
```

stopword를 처리하기 위해 기본적으로 stopword를 포함하고 있는 nltk에서의 stopword를 불러왔다. 추후 token화된 내용을 확인해 stopword에 추가해주고 처리하기 위함이다.

앞서 파악한 Counter()를 통한 token count 내용을 기반으로 전처리될 token 대상과 stopword 대상을 정의하였다. 기존의 nltk 토큰 이외에 추려낸 stopword는 다음과 같다. 숫자 정보에 대해서는 이미지 캡션 task에 대해서 중중요한 정보일 수 있으므로 stopword에 포함시키지 않았다.

```
stopword2 = ['s', "''", 'st', 'c', 'u', 'p', 'v', 'n', 'ac', 'ou', "n'", 'gin', 'od', 'lav', 'kelp', 'nd',
'gren', 'tbe', 'gith', 'dhe', 'h', 'la', 'osme', 'ot', 'djs', 'hte', 'fronr', 'ox', 'shire',
'hdr', 'rung', 'boe', 'twp', 'vfw', 'ti', 'chi', 'tho', 'ont', 'mma', 'iove', 'ump',
'outise', 'lei', 'pf', 'ee', 'thong', 'tge', 'ilks', 'r', 'floatlys', 'stucco', 'arc',
'brwon']
```

stopword 처리를 위해 기존의 stopwords를 처리하고, 새로 정의한 stopwords를 반영하는 filter_tokens()를 정의하여 token화한 token_list feature에 적용시켜주었다. 추가적으로 token 길이도 초기화시켜주었다.

```
def filter_tokens(string):
  string = [s for s in string if s not in stop_words]
  string = [s for s in string if s not in stopword2]
  return string

df['filtered_token']=df['token_list'].apply(filter_tokens)
df['filtered_len']=df['filtered_token'].apply(len) # token len에 새롭게 반영
```

[2. 단어 PAD 처리]

문장의 시작과 끝을 나타내는 padding 처리를 진행해줬다.

시작은 <start> 토큰으로 문장의 끝은 <end>라는 토큰을 반영해줬다.

```
def add_pad(tokens):
  padded_tokens = ['<start>'] + tokens + ['<end>']
  return padded_tokens

df['filtered_token'] = df['filtered_token'].apply(add_pad)
```

tokenizer.fit_on_texts()를 통해 문자 데이터를 입력 받아 리스트 형태로 변환해주면,

tokenizer의 word_index 속성은 단어와 숫자의 key-value 쌍을 포함하는 딕셔너리로 반환해준다.

PAD를 진행했기 때문에 PAD에 0를 더해주었다.

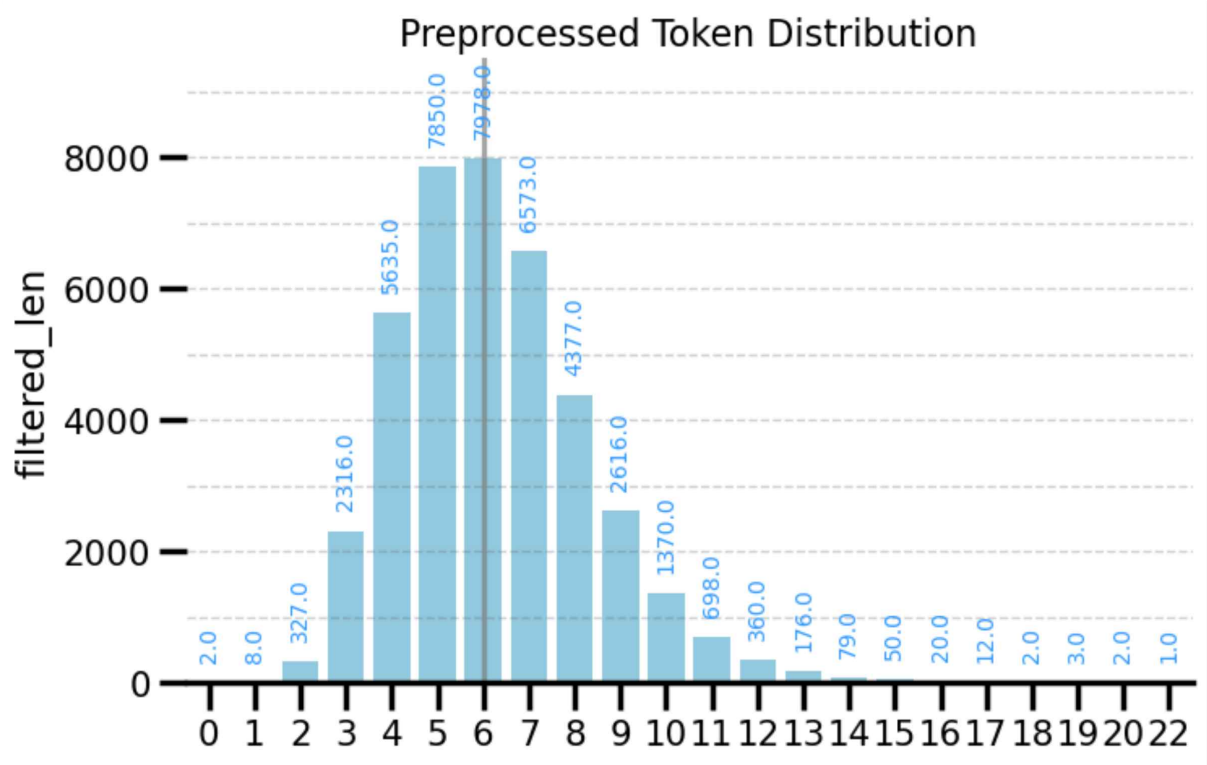
```
# word-to-index mapping
tokenizer.fit_on_texts(df['filtered_token'])

# embedding (integer)
train_seqs = tokenizer.texts_to_sequences(df['filtered_token'])

# PAD에 zero 더함
tokenizer.word_index['PAD'] = 0
tokenizer.index_word[0] = 'PAD'

print(tokenizer.oov_token)
# output : UNK
print(tokenizer.index_word[0])
# output : PAD
```

tokenizer.index_word를 통해 각 token에 대한 인덱스, 사전을 호출해 확인할 수 있는데 그 예시는

다음과 같다.

{1: 'UNK',
 2: '<start>',
 3: '<end>',
 4: 'dog',
 5: 'man',
 6: 'two',
 7: 'white',
 8: 'black',
 9: 'boy',
 10: 'woman',
 11: 'girl',
 12: 'wearing',
 13: 'people',
 14: 'water',
 15: 'red',
 16: 'young',
 17: 'brown',
 18: 'blue',
 19: 'dogs',
 20: 'running',
 …

Stopword들이 제거된 Token들의 Distribution은 다음과 같았다.



전처리 하기 이전의 Token DIstribution과 전처리 이후의 Preprocessed Token Distribution을 비교하면, 전처리 이전의 token lendms 0~35 사이의 분포를 찾고 있으며 중위수 10을 보였다. 전처리 이후의 token distribution은 0~22 사이의 분포를 갖고 있으며 왼쪽으로 skewed된 형태이다. 전처리 이전엔 보통 0~25 사이의 분포를, 전처리 이후에는 0~15정도의 분포로 token의 len이 줄어드는 것을 확인 할 수 있다. 해당 프로젝트의 목적은 attention을 활용한 image captioning으로 skewed data에 대한 추가적인 조치나 실험을 수행하지 않았으나 추후 확인해볼 수 있을 것이다.

[3. PAD vector 처리]

추후 model 학습을 위해서는 padding 과정이 필요하다. 해당 과정을 위해 먼저 이를 위해 texts_to_sequences()를 통해 integer 형태로 embedding하여 저장한 train_seqs에 대해 각각의 길이를 반환하는 train_seqs_len list를 정의해주었고, 이를 기반으로 max_word_length를 파악하였다.padding 처리를 위해 tensorflow의 keras processing sequence pad_sequence (tf.keras.preprocessing.sequence.pad_sequences())를 통해 pad의 길이를 지정해 padding 처리를 진행해주었다. padding은 뒷 부분에 처리되게 하였다.

```
train_seqs_len = [len(seq) for seq in train_seqs]
max_word_length = max(train_seqs_len)

caption_vec = tf.keras.preprocessing.sequence.pad_sequences(train_seqs,
                                padding='post',
                                maxlen=max_word_length,
                                dtype='int32', value=0)
print(f'Caption vector shape : {caption_vec.shape}')
# output : Caption vector shape : (40455, 24)
```

4) Image 전처리

[0. Data 형식 및 규칙 확인]

먼저 Image preprocessing을 수행하기 위해 image data들의 형식과 규칙의 값을 확인하였다.

```
# check image format (이미지 데이터 형식 규칙의 값을 설정)
tf.keras.backend.image_data_format()
# output : 'channels_last'
```

2차원 데이터와 3차원 데이터에 따라 해당 결과가 다르게 나오는데, 다음과 같이 나타나게 된다. 해당 데이터의 경우 이미지 데이터로 2차원의 데이터이고, channels_last 형태로 (rows, cols, channels) 로 나타나게 됨을 확인할 수 있다.

[1. Image Preprocessing]

1) image들이 잘 처리가 되는지 사전 확인 작업을 수행했다.

```
prep_img = []
IMAGE_SHAPE = (299, 299)

for img in all_imgs[0:5] :
    img = tf.io.read_file(img, name=None)
    img = tf.image.decode_jpeg(img, channels=0)
    img = tf.image.resize(img, IMAGE_SHAPE)
    img = tf.keras.applications.inception_v3.preprocess_input(img)
    prep_img.append(img)

# image들이 제대로 전처리 되었는지 5개만 추출해 확인
Display_Images = prep_img[0:5]
fig, axes = plt.subplots(1,5)
fig.set_figwidth(25)

for ax, image in zip(axes, Display_Images):
  print(f'resized shape : {image.shape}')
  ax.imshow(image)
  ax.grid('off')
```

이미지들이 잘 나오는 것을 확인해 본격적인 image preprocessing을 진행하였다. 앞서 본 방식과 조금 다른 방식으로 전처리를 수행했고 내용은 다음과 같다.

```
def load_images(image_path) :
  img = tf.io.read_file(image_path, name = None)
  img = tf.image.decode_jpeg(img, channels=3)
  img = tf.image.resize(img, IMAGE_SHAPE)
  img = tf.keras.applications.inception_v3.preprocess_input(img)
  return img, image_path

training_list = sorted(set(df['path']))

# 로드된 데이터에 대한 데이터 생성
New_Img = tf.data.Dataset.from_tensor_slices(training_list)

# 데이터 변환 병렬화
New_Img = New_Img.map(load_images,
                num_parallel_calls = tf.data.experimental.AUTOTUNE)

New_Img = New_Img.batch(64, drop_remainder=False)
```

image들을 불러 읽어와 jpeg로 decoding을 수행해줬고, IMAGE_SHAPE에 맞게 resizing 처리를 했다. 기존에 예시로 확인해본 것과 다르게 inceptionV3를 통해 이미지에 대한 feature

extracting을 수행하여 반환하였다. tensorflow Dataset의 from_tensor_slices()를 통해 메모리에 로드된 데이터들에 대한 Dataset을 생성하게 된다.

tf.data.Dataset.map은 사용자 정의 함수를 데이터셋의 각 요소에 정의할 수 있게 한다. 위에서 정의한 image 전처리 함수에 대해서 수행하고, num_parallel_calls는 런타임에 가용되는 병렬화 수준에서 가능하도록 tf.data.experimental.AUTOTUNE을 통해 autotuned 되도록 설정하였다.

다음으로 batch()를 통해 모델 학습시 batch_size를 지정해 수행되게 하였다. 마지막 남은 데이터를 drop하지 않기 때문에 drop_remainder는 False로 지정해주었다.

5) Split Dataset for Training

모델 학습을 위해 앞서 전처리한 데이터들을 분할해주었다. 비율은 8:2로 test_size=0.2로 설정해 진행하였다. 추후 같은 결과가 나오게 하기 위해 random_state = 24로 지정해줬다. 앞서 경로에 대한 전처리를 통해 path를 지정해 feature를 생성해줬던 것을 활용한다.

```
#img_path = '/content/drive/MyDrive/images/' # colab
img_path = '/kaggle/input/flickr8k/Images/'  # kaggle notebook
df['path'] = img_path + df['image']
```

```
path_train, path_test, caption_train, caption_test = train_test_split(df['path'], caption_vec,
test_size=0.2,
                 random_state = 24)

print(f'Train data (images)  : {str(len(path_train))}')
print(f'Test  data (images)  : {str(len(path_test))}')
print(f'Train data (caption) : {str(len(caption_train))}')
print(f'Test  data (caption) : {str(len(caption_test))}')
```

3. Feature Extracting & Prepare Dataset

1) Feature Extracting

Encoder 부분에 CNN 계열 중 , 이처럼 이미지, 자연어, Image Caption 등의 분야에서 활발하게 사용이 되고 있는 Attention을 기반으로 기초적인 Image Caption project를 수행하고자 하였다. CNN 계열에서는 InceptionV3를 활용하였는데, ResNet, EfficientNet, VIT

등의 아키텍처가 좋은 성능을 나타내고 있으나 데이터셋의 크기가 상대적으로 작고, 계산의 리소스 측면에서 효율적인 효과를 나타낼 수 있을 것이라 생각하여 InceptionV3를 활용하여 Feature Extracting을 진행하였다.

[0. Import Pretained InceptionNetV3]

```
image_model = tf.keras.applications.InceptionV3(include_top = False, weights='imagenet')
new_input = image_model.input
hidden_layer = image_model.layers[-1].output
inceptionv3_model = tf.compat.v1.keras.Model(new_input, hidden_layer)
```

사전학습된 InceptionV3 모델을 불러왔고, new_input과 은닉층을 정의해 새로 정의한 inceptionv3_model 이라는 변수명으로 저장해주었다.

[1. Feature Extracting]

앞에서 정의한 inceptionv3_model을 통해 feature extracting 작업을 진행하고 해당 내용을 flatten 해주었다. img_feature의 utf-8로 decoding 한 path에 numpy() 형태로 저장된다.

```
img_features = {}
for img, image_path in tqdm(New_Img):
  batch_features = inceptionv3_model(img)
  # squeeze features in batch
  flattened_feature = tf.reshape(batch_features, (batch_features.shape[0],-1,
batch_features.shape[3]))

  for batch_feat, path in zip(flattened_feature, image_path):
    feature_path = path.numpy().decode('utf-8')
    img_features[feature_path] = batch_feat.numpy()
```

batch_feat의 TensorShape은 shape 함수를 통해 살펴본 결과 TensorShape([64, 2048])로 나타났다.

2) Prepare Dataset for training

[0. Generate Data]

전처리한 이미지를 통해 데이터들을 이미지와 캡션을 묶어 데이터를 생성할 수 있도록 정의하는 부분이다. 위에서도 Feautre extracting을 위해 유사한 과정을 거쳤는데, 해당 부분은 img와 caption을 묶어 하나의 dataset으로 형성하기 위한 과정이다.

먼저 map 함수를 정의하였다. 이미지와 이름과 caption을 묶는 함수 부분이다.

원래는 image, caption을 동시에 시각화하기 위한 함수였는데, img와 caption을 통해 새로운 dataset을 정의하기 위한 과정에서도 활용하였다.

```
def map(image_name, caption):
  '''
  # image, caption을 동시에 시각화하기 위한 함수 정의
  1. feature extracting 단계에서 flattened한 feature에서 설정한
     img_features[feature_path] utf-8로 decoding
  '''
  img_tensor = img_features[image_name.decode('utf-8')]
  return img_tensor, caption
```

먼저 tf.data.Dataset.from_tensor_slices()에 (img,capt)을 묶어 집어넣어 data라는 변수로 초기화시켰고, data.map()를 통해 지정한 함수를 병렬화 처리할 수 있게 하였다. 함수 정의시 tf.numpy_function()을 사용하였는데 함수를 매핑해 tensorflow operation에 활용할 수 있다.

다음으로 미리 정의한 BUFFER_SIZE에 따라 data들을 shuffle하고, 미리 정의한 BATCH_SIZE에 따라 batch를 수행하도록 한다. 이는 앞서 수행한 내용과 같고, 결과를 반환하도록 설정하였다.

```
BUFFER_SIZE = 1000
BATCH_SIZE = 64


def gen_dataset(img, capt):
  # 데이터셋 생성
  data = tf.data.Dataset.from_tensor_slices((img, capt))

  # tf.numpy_function() : 함수를 매핑하여 tensorflow operation에 활용
  data = data.map(lambda ele1, ele2 :
        tf.numpy_function(map, [ele1, ele2], [tf.float32, tf.int32]),
        num_parallel_calls = tf.data.experimental.AUTOTUNE)

  # 데이터 셔플 및 가져오기
  data = (data.shuffle(BUFFER_SIZE,
         reshuffle_each_iteration= True).batch(batch_size =
         BATCH_SIZE, drop_remainder = False)
         .prefetch(tf.data.experimental.AUTOTUNE))
  return data
```

최종 데이터셋은 다음과 같이 train_dataset과 test_dataset으로 저장해줬다.

```
train_dataset = gen_dataset(path_train,caption_train)
test_dataset = gen_dataset(path_test,caption_test)
```

[1.데이터의 구조 파악]

next() 함수를 통해 batch size, feature map 크기, caption의 길이와 같은 데이터의 구조를 파악할 수 있다. next(iter(train_dataset))을 통해 첫 번째 배치를 가져올 수 있어 작업이 올바르게 수행되는지 확인할 수 있다.

```
sample_img_batch, sample_cap_batch = next(iter(train_dataset))

print(f'현재 배치의 이미지 데이터 모양 : {sample_img_batch.shape}')
# output : 현재 배치의 이미지 데이터 모양 : (64, 64, 2048)
print(f'현재 배치의 caption 데이터의 모양 : {sample_cap_batch.shape}')
# output : 현재 배치의 caption 데이터의 모양 : (64, 24)
```

[2. 파라미터 설정]

```
# Parameter 설정
embedding_dim = 256 # 결과 embedding 크기
units = 512

# vocab size 및 step 조정
vocab_size = 5001   # top 5,000 words +1
train_num_steps = len(path_train) // BATCH_SIZE
test_num_steps = len(path_test) // BATCH_SIZE

max_length = 31
feature_shape = batch_feat.shape[1]
attention_feature_shape = batch_feat.shape[0]
```

[3. 모델 학습을 위한 그래프 초기화]

이전에 정의된 텍서 및 연산을 제거하고, 모델 간의 충돌을 방지하기 위해 다음 과정을 수행했다. tensorflow v2 부터는 그래프 초기화 문제는 사라졌다고한다. 그러나 tensorflow v1에서는 tf.function 데코레이터를 통해 그래프를 정적으로 컴파일 및 최적화가 가능하다는 이점이 있다고 한다.

```
tf.compat.v1.reset_default_graph()
print(tf.compat.v1.get_default_graph())
```

4. Model Architecture

1) Encoder

우선 Tensorflow.keras의 Model을 상속받게 하여 Encoder와 Decoder를 생성했다. 모델의 실제 동작은 각 class 모듈의 call 함수를 통해 정의했다. Encoder 에서는 Attention을 활용하지 않고 embedding의 크기인 embed_dim을 받으면 Feature extracting을 수행하도록 하였다. 활성화함수는 ReLU를 활용하였으나, 음수 부분의 기울기를 0으로 완전 제한하는 것을 방지하기 위해 alpha값을 0.01로 설정하여 Leaky ReLU와 같은 결과를 나타내게 하였다. 출력값은 제한하지 않기 때문에 max_value는 따로 설정하지 않고 None으로 지정하였으며, 임계값 활성화를 위한 threshold는 기본적인 0으로 설정하였다.

```
class Encoder(Model):

  def __init__(self,embed_dim):
    super(Encoder, self).__init__() # 부모 클래스의 생성자를 불러와 상속받음
    self.dense = tf.keras.layers.Dense(embed_dim)


  def call(self, features):
    '''
    모델의 동작을 정의
    1) self.dense(features) : Feature extract (FC layer)
    -- 선형 변환 수행 : 입력 데이터 * 가중치 행렬 + bias
    -- 데이터 형상 변환, 데이터 포함된 주요 특성 추출에 도움

    2) relu 활성화함수 사용
    -- # ReLU(x)= max(0,x)
    -- alpha : 음수 부분의 기울기 (default=0)
       -- alpha=0.01 : Leaky ReLU (음수값이 모두 0이 되는 것 방지 위함)
    -- max_value : ReLU의 출력값 제한
       -- max_value =6 : 출력값이 6보다 큰 경우 6 출력
    -- threshold : 임계값 활성화를 위한 값
       -- 입력값이 0보다 작을 때 0 출력
       -- threshold 설정시 다른 값 출력되도록 조절
       -- ex) threshold=0.2 -> 0.2보다 작을 때 0 대신 음수 출력
    '''
    features =  self.dense(features) # image shape: (batch, 8*8, embed_dim)
    features =  tf.keras.activations.relu(features, alpha=0.01,
                                max_value=None, threshold=0)
    return features
```

다음으로 Attention을 활용하기 위해 Attention 방식을 정의했다. Attention의 방식은 유사하나 중간 Attention score에 따라 Attention의 종류가 결정되게 되는데, 어텐션의 개념을

제공한 연구였던 Bahdanau Attention을 활용하는 것이 의미가 있을 것으로 생각되어 활용하였다.

	수식
Luong Attention (dot score)	$score(s_t, h_i) = s_t^T * h_i$
Bahdanau Attention (concat score)	$score(s_t, h_i) = W_a^T * tanh(W_b * s_t + W_c * h_i$

[표1] Attention 정의 수식 (Luong과 Bahdanau 중 Bahanau 사용)

Bahdanau Attention은 Key와 Value가 동일하며, query가 decoder cell에서의 t-1 시점의 은닉 상태로 정의된다. 먼저 score를 계산하기 위해 차원을 맞춰주는 작업을 수행했다. Attention Mechanism이 각 time step에 가중치를 할당해 중요 정보에 집중하기 위한 계산을 위해서는 time step과 hidden state 차원이 일치해야 한다.

두 번째로 score를 계산하게 되는데 위에 정의한 Attention의 종류를 결정하는 부분이다. concat된 score를 활용하게 되고, 가중치 행렬과 곱해진 후 tanh()를 거치게 되는데 -1~1 사이의 범위를 갖고 있고, 입력에 대한 양수, 음수 구분이 가능해 다양한 유형의 정보를 처리할 수 있게 함으로서 새로운 표현을 생성하고 모델의 표현 능력 향상에 도움이 된다고 한다.

계산한 score값을 통해 attention distribution과 그 각 값인 attention weight를 구하게 되는데 softmax()를 통과하는 이유는 attention distribution을 구하기 위함이다. 이는 0~1 사이의 확률값으로 변환시켜준다. 마지막으로 attention value를 구해주는 부분이다. 각 encoder의 attention weight과 encoder의 은닉 상태를 가중합하게 된다. encoder의 문맥을 포함하기 때문에 context vector로도 표현된다.

```
class Attention_model(Model):
    def __init__(self, units):
        super(Attention_model, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)
        self.units=units

    def call(self, features, hidden):
        '''
        1. bahdanau attention은 Key, Value 동일
        -- query가 decoder cell의 t-1 시점의 은닉 상태
        2. hidden_with_time_axis
        -- score 계산을 위해 뒤에서 할 덧셈을 위해 차원을 변경해줌
        -- 3가지 방안
            1) expand_dims : 늘리고 싶은 axis 지정하면 해당 axis에 차원 추가됨
            -- -1 이면 가장 뒷부분 차원 추가
            2) tf.newaxis : 기존 배열을 적고 추가하고 싶은 위치에 적으면 간단히
              size 변경 가능
            -- 차원이 정해져 있지 않음, 간단히 변경
            3)reshape() : 차원이 정해져 있음
        3. score 계산
        -- score(s_(t), h_{i}) = W_{a}^T * tanh(W_{b}*s_{t} + W_{c}*H_{i}})
        4. tf.reduce_sum() : # 텐서의 차원을 탐색해 개체의 총합을 계산
        -- tf.reduce_sum(x,0) : 열 단위로 sum
        -- tf.reduce_sum(x,1) : 행 단위로 sum
        -- tf.reduces_sum(x,1, keep_dims=True) : keep_dims= 원래 상태 유지
        '''
        # 1. score 계산 위한 차원 변경
        hidden_with_time_axis = hidden[:, tf.newaxis]

        # 2. score 계산
        score = tf.keras.activations.tanh(self.W1(features) +
                                          self.W2(hidden_with_time_axis))

        # 3. attention weight 구함
        # attention distribution의 각 값인 attention_weight를 구함
        # -- softmax()를 활용해 attention distribution을 구함
        attention_weights = tf.keras.activations.softmax(self.V(score), axis=1)

        # 4. attention value (context vector) 구함
        # -- 각 encoder의 attention weight과 encoder의 은닉 상태를 가중합
        # -- encoder의 문맥을 포함하는 차원에서 context vector
        context_vector = attention_weights * features
        context_vector = tf.reduce_sum(context_vector, axis=1)  # 행 단위 sum
        return context_vector, attention_weights
```

3) Decoder

다음으로 Decoder 모듈을 형성했다. 위에서 정의한 Attention을 활용하게된다. 역시 모듈 안의 call 함수로 동작을 정의하게 된다.

attention 모델로부터 context vector와 attention weight을 받아오고, embedding 층을 거치게 된다. '분포 가설'에 기반하면 모든 단어를 고정 차원의 벡터로 표현할 때 유사한 맥락에서 나타나는 단어는 그 의미도 비슷하다는 것인데, 이를 기반으로 얻어지는 단어 벡터를 분산 표현이라 한다. embedding 층은 단어를 고정된 길이의 밀집 벡터로 변환하는 작업을 수행하며 고차원 벡터 공간으로 투영시켜 단어를 의미론적으로 풍부한 표현으로 변환해 모델이 문장을 더 잘 이해할 수 있게 한다. embedding 한 input과 attention으로부터 얻은 context vector를 concat해주게 되고 이를 이용해 연산을 진행한다. LSTM을 기반으로 간단한 아키텍처를 갖고 있는 GRU를 활용하였다. LSTM과 구조상에 큰 차이가 없고 결과 역시 큰 차이가 없으나 상대적으로 적은 dataset을 갖고 있기 때문에 GRU가 효율적이라 생각되어 활용하였다. GRU를 통해 output과 hidden state를 받아왔고, Dense 층을 거치고 shape을 맞춰준 후 vocab_size에 맞게 출력해주었다.

```
class Decoder(Model):
    def __init__(self, embed_dim, units, vocab_size):
        super(Decoder, self).__init__()
        self.units=units
        self.attention = Attention_model(self.units) # Attention 모델 초기화
        self.embed = tf.keras.layers.Embedding(vocab_size, embed_dim)
        self.gru =
tf.keras.layers.GRU(self.units,return_sequences=True,return_state=True,recurrent_initializer='glorot
_uniform')
        self.d1 = tf.keras.layers.Dense(self.units)
        self.d2 = tf.keras.layers.Dense(vocab_size)


    def call(self,x,features, hidden):
        # attention 모델로부터 context vector & attention weights
        context_vector, attention_weights = self.attention(features, hidden)

        # input을 shape (batch_size, 1, embedding_dim)로 embed함
        # 입력 : 2D 정수 텐서 (number of samples, input_length)
        # 출력 : 3D 정수 텐서 (number of samples, input_length,
                        embedding word dimensionality)
        embed = self.embed(x)
        # concatenate : input + context vector from attention
        # shape: (batch_size, 1, embedding_dim + embedding_dim)
        embed = tf.concat([tf.expand_dims(context_vector, 1), embed], axis = -1)

        # GRU (context vector)
        ## 1. 상호작용 : 입력 시퀀스의 각 위치에서 중요 정보 추출, 예측
        ## 2. 모델이 이전 위치의 상태를 잘 유지하고 활용 가능.
          #  Attention 통해 장기 의존성을 더 잘 모델링 가능
        ## 3. LSTM 대비 간단한 구조 -> 추론, 학습에 더 적은 비용
        output,state = self.gru(embed)
        # Output shape : (batch_size, max_length, hidden_size)
        output = self.d1(output)
        output = tf.reshape(output, (-1, output.shape[2]))
        # shape : (batch_size * max_length, hidden_size)
        output = self.d2(output)
        # shape : (batch_size * max_length, vocab_size)

        return output, state, attention_weights

    def init_state(self, batch_size):
        '''
        모든 요소가 0인 tensor를 생성
        '''
        return tf.zeros((batch_size, self.units))
```

```
encoder=Encoder(embedding_dim)
decoder=Decoder(embedding_dim, units, vocab_size)
```

encoder와 decoder라는 변수에 해당 모듈들을 저장하였다.
해당 모델이 잘 작동하는지 확인하기 위해 기존에 next()함수를 통해 추출한 데이터를 이용해 사전 확인을 진행하였다.

```
# 사전 확인하기 (기존 next() 통해 추출한 데이터)
features=encoder(sample_img_batch)
hidden = decoder.init_state(batch_size=sample_cap_batch.shape[0])

dec_input = tf.expand_dims([tokenizer.word_index['<start>']] *
sample_cap_batch.shape[0], 1)

predictions, hidden_out, attention_weights = decoder(dec_input, features, hidden)

print('Feature shape from Encoder: {}'.format(features.shape))
#(batch, 8*8, embed_dim)

print('Predcitions shape from Decoder: {}'.format(predictions.shape)) #(batch,vocab_size)

print('Attention weights shape from Decoder: {}'.format(attention_weights.shape))
#(batch, 8*8, embed_dim)

output :
# Feature shape from Encoder: (64, 64, 256)
# Predcitions shape from Decoder: (64, 5001)
# Attention weights shape from Decoder: (64, 64, 1)
```

[0. Optimizer / Loss Function 정의 및 Masking처리리]
해당 내용이 잘 작동하기 때문에 추후 train_step() 함수를 정의해 학습에 활용하였다.
해당 학습을 위한 함수를 정의하기 앞서 optimizer와 loss function을 정의하였다.

```
optimizer = tf.keras.optimizers.Adam(learning_rate = 0.001)
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
                    from_logits = True, # 값을 그대로 loss 입력으로
                    reduction = tf.keras.losses.Reduction.NONE)
```

optimizer는 adam을 사용했으며 learning rate는 0.001로 지정해 학습시켰다.
loss function의 경우 sparse categorical crossentropy를 활용하였는데, model의 출력값이 normalized 되지 않고 값을 그대로 loss 입력으로 사용하기 위해 from_logits=True로 설정했다.
reduction의 경우 값들을 각자에 집어넣은 값을 반환하는지 합쳐 반환하는지 결정하는

부분으로 NONE을 지정해 각자 나올 수 있도록 지정하였다. SUM을 사용할 경우 default값으로 값들이 합쳐져 나오게 된다.

sequnce한 모델에 대해 동일한 것에 대해 과적합이 발생할 수 있기 때문에 masking 방식을 사용할 수 있다. 아래는 masking을 처리하는 함수이다.

```
def loss_function(real, pred):
    '''
    1. padding이 모델에 penalty를 부과
    2. padding 후 masking을 진행해야
    -> timestep마다 pad 무시하라 말해줘야 하는데 loss를 증가시킴
    3. 모든 caption과 경로를 담은 list를 생성해야 함
    '''
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)

    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

    return tf.reduce_mean(loss_)
```

[1. Checkpoint 저장]

훈련 과정에 checkpoint를 통해 저장할 수 있고, 모델을 재사용할 경우 활용할 수 있다.

```
checkpoint_path = "Flickr8K/checkpoint1"
ckpt = tf.train.Checkpoint(encoder=encoder,
                           decoder=decoder,
                           optimizer = optimizer)

ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path,
                                          max_to_keep=5)

start_epoch = 0
if ckpt_manager.latest_checkpoint:
    start_epoch = int(ckpt_manager.latest_checkpoint.split('-')[-1])
```

5. Model Training / Result

위에서 model architecture가 원활히 수행되는 것을 확인하였기 때문에 train_step()이라는 모델을 학습시키는 함수를 정의하였다.
추가적으로 tf.GradientTape()을 통해 자동 미분한 내용을 tape에 기록할 수 있게 하였다.

```
@tf.function
def train_step(img_tensor, target):
    loss = 0
    hidden = decoder.init_state(batch_size=target.shape[0])
    dec_input = tf.expand_dims([tokenizer.word_index['<start>']] *
                               target.shape[0], 1)

    # 자동 미분 -> tape에 기록
    with tf.GradientTape() as tape:

        for r in range(1, target.shape[1]) :
            # Decoder를 호출해 예측값, 새로운 은닉 상태 반환
            predictions, hidden, _ = decoder(dec_input,
                                             encoder(img_tensor),
                                             hidden)
            # 현 위치에서의 예측값, 실제 단어 사이 손실 계산하고 누적
            loss = loss + loss_function(target[:, r], predictions)
            # 다음 위치 Decoder 입력으로 현재 위치 대상 단어 사용
            dec_input = tf.expand_dims(target[:, r], 1)

    # 배치 당 평균 손실
    avg_loss = (loss / int(target.shape[1]))

    # 업데이트 가능 변수들을 가져옴
    trainable_vars = encoder.trainable_variables +
                     decoder.trainable_variables

    # 손실에 대한 경사값
    grad = tape.gradient (loss, trainable_vars)

    # 경사값 이용해 모델 변수 update
    optimizer.apply_gradients(zip(grad, trainable_vars))

    return loss, avg_loss
```

test_step() 함수도 동일한 과정으로 생략하였다.

test_dataset을 위한 를 위한 함수는 다음과 같다.

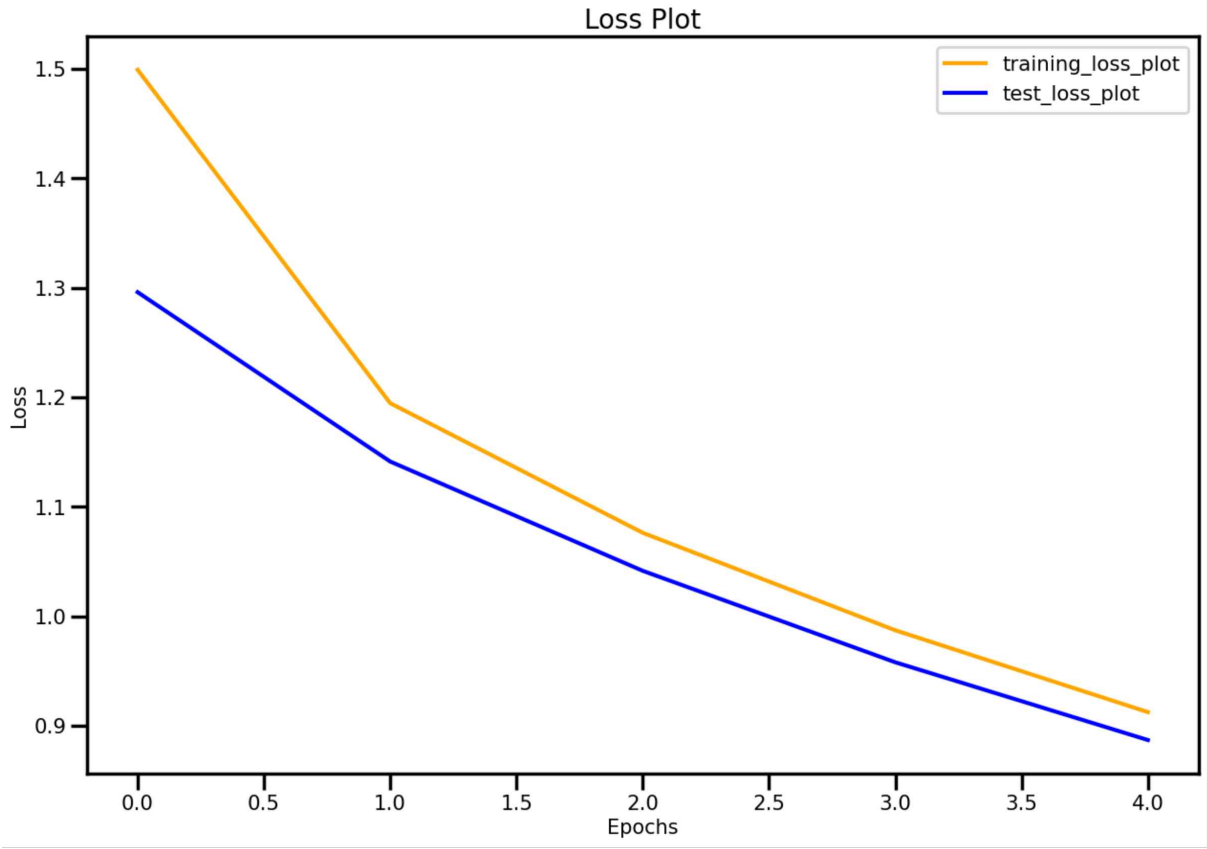
```
def test_loss_cal(test_dataset):
    total_loss = 0
    for (batch, (img_tensor, target)) in enumerate(test_dataset) :
      batch_loss, t_loss = test_step(img_tensor, target)
      total_loss = total_loss + t_loss
      avg_test_loss = total_loss/ test_num_steps

    return avg_test_loss
```

totoal_loss를 0으로 초기화하고, test_dataset에서 batch, image_tensor, target에 대해 loss를 반복적으로 계산해준다. 해당 값에 대한 avg_test_loss를 반환한다.

모델 학습은 우선 EPOCHS=5 로 설정하여 5회만 학습을 진행했다.

```
loss_plot = []
test_loss_plot = []
EPOCHS = 5
best_test_loss=100

for epoch in tqdm(range(0, EPOCHS)):
    start = time.time()
    total_loss = 0
    for (batch, (img_tensor, target)) in enumerate(train_dataset):
        batch_loss, t_loss = train_step(img_tensor, target)
        total_loss += t_loss
        avg_train_loss=total_loss / train_num_steps

    loss_plot.append(avg_train_loss)
    test_loss = test_loss_cal(test_dataset)
    test_loss_plot.append(test_loss)

    print ('For epoch: {}, the train loss is {:.3f}, & test loss is
{:.3f}'.format(epoch+1,avg_train_loss,test_loss))
    print ('Time taken for 1 epoch {} sec\n'.format(time.time() - start))

    if test_loss < best_test_loss:
        print('Test loss has been reduced from %.3f to %.3f' % (best_test_loss, test_loss))
        best_test_loss = test_loss
        ckpt_manager.save()
```

결과 loss에 대한 plot은 다음과 같이 나타났다.



6. Roles / Contribution / 느낀점

1) EDA

- 전처리를 수행하기 이전에 데이터셋에 대한 기본적 EDA를 수행했다.
- 각 이미지에 따른 caption의 개수 및 token의 개수를 파악하는 과정에서 데이터의 분포를 확인 할 수 있었다.
- 전처리 전과 후의 token 분포를 비교해보았다.
- token len 개수와 성능의 상관성에 대해 추후 실험해 볼 수 있을 것 같다.
- 더불어 skewed data에 대한 정규분포로의 변환을 통한 성능 비교로 수행해볼 수 있을 것 같다.

2) Text 전처리

- Text를 전처리함에 있어 해당 Task에 적합한 방식에 대해 고민하는 과정을 통해 keras tokenizer를 활용해 tokenize 하였다.
- token화 내용들을 list에 담아 collection의 Counter()를 통해 token별 사용 횟수를 파악하였고, 이를 csv 파일에 담아 전처리가 필요한 text를 파악, 불용어를 파악하여 replace 함수 및 apply()를 통해 기존 caption text에 적용시켜 전처리를 수행했다.
  같은 단어의 spelling이 다르게 작성이 되어있거나 동일한 단어를 다르게 표현할 수 있기

때문에 해당 내용을 고민하는 과정이 필요했다.

- 이를 통해 초기 데이터의 중요성에 대해서도 생각해 볼 수 있었는데, 데이터 구축시 해당 내용에 대한 고민을 고려해 데이터를 구축하는 것이 추후 분석 및 모델링시 효율적일 것이라 생각되었다. 현실적으로 고민해보면, 원시 데이터가 원하는대로 구축되어있을 확률은 극히 낮기 때문에, 이를 보완하기 위해 초기 데이터를 구축 후 대량의 데이터를 구축할 때 방향성을 세워 효율적으로 적용시킬 수 있을 것 같다.
- 관련한 내용으로 Text 처리에 LLM이 효율성을 증명하고 있음에 따라 Image에 대해서도 LLM을 적용하고자 하는 흐름이 생겨났다. 기존 지도 기반 학습에 대한 결과는 약 10%의 성능을 보였다. 이에 대한 대안으로 class label 학습에서 벗어나 raw text data를 지도학습으로 활용하고자 하는 방법과 image와 text pair를 contrasting 기법으로 처리해 학습하는 CLIP 모델이 2021년 등장해 SOTA를 보였다. Large web scale 데이터 상에 image 및 text 데이터에 noise가 많다는 한계가 존재해 이후 잘못된 캡션을 걸러내고 새로운 캡션을 사용해 한계를 극복한 BLIP이 2022년 등장해 SOTA를 기록했다. 이후 BLIPv2 역시 등장한 상태이다.

- 기존에 NLTK에 있는 stopword를 import해 적용시키고, 나머지 token 중 stopword로 적용 가능한 token들을 추려내면서 task에 따른 stopword 정의가 다를 수 있음을 상기시켰다. 보통 숫자를 stopword에 포함시키기도 하는데, 해당 프로젝트의 목적은 image caption으로 숫자 정보가 영향을 미칠 수 있을 것이라 판단되어 stopwords에서 제외하였다.

- 모델을 처리하기 위한 PAD 처리 및 단어 사전을 구축하고 padding을 통한 embedding을 수행하였는데, 각 함수를 적용시키면서 해당 프로젝트에 적합한 방향성에 대해 고민해 볼 수 있었다.

3) Image 전처리

- tensorflow의 backend의 image_data_format()을 활용해 데이터의 포맷 확인을 통해 2D와 3D 데이터에서의 처리 방향에 고려할 수 있음을 알게 되었다.
- 이미지 전처리 및 feature extracting 과정에서 VGGNet과 비슷한 연산량을 보이나 합성곱 분해 방식을 통해 연상량을 최소하며 모델의 크기를 키운 Inception을 활용하였다. InceptionV1, V2의 최적 기법을 모두 활용한 InceptionV3를 활용하였는데, image feature map size를 줄이면서도 표현력을 줄이지 않아 더욱 효과적인 방식이 될 것이라 생각되었다.
- 추출한 특징을 학습에 활용하기 위해 flatten 과정이 필요한데, reshape을 활용하였다. reshape(arr, newshape, order='C') 함수는 변경할 배열, 변경하고자 하는 새로운 형태인 튜플 또는 정수, 재배열 순서 ('C' / 'F')를 작성하게 된다. 해당 프로젝트에서는 튜플의 형태로 추출한 batch 크기 (batch_features.shape[0]) , -1을 통한 차원 크기 자동 조정, 차원 수 (batch_features.shape[3])의 형태를 갖게 조정하였다.

- reshape을 통한 크기 조정, shape에 대한 이해, 원하는 차원으로의 확장 방법에 대한 내용에 보다 심도있는 이해와 경험이 필요할 것이다.
- flatten feature에 대해 flatten feature와 image 경로를 묶어 path는 numpy utf-8 형태로 디코딩시켰고, flatten된 batch_feat을 numpy()로 변환하였다.

4) 데이터 생성

- 모델에 input으로 들어가는 데이터 처리 방식에 대해 고민했다. image와 자연어를 동시에 처리함에 따라 2개의 data 형태를 묶어 하나의 input으로 처리해줘야 하는지, 또 그에 대한 학습 방법은 어떻게 해야 할지에 대한 고민을 하였다. Encoder에서는 이미지에 대한 처리를 Decoder에서는 자연어에 대한 처리를 할 것인데, 학습시 두 가지 data 형태 input이 같이 들어가서 처리되어야 하는지, 아니면 따로 학습이 되어 연결될 수 있도록 해야 할지에 대한 고민이 쉽지만은 않았다. 그 과정에서 tf.data.Dataset.from_tensor_slices()라는 함수를 통해 데이터셋을 생성하고 data.map()을 통해 전처리 함수를 병렬적으로 활용할 수 있게 됨을 알게 되었다. 이에 해당 프로젝트에서는 해당 방식을 활용하여 처리해 보았다.
  그 과정에서 research를 통해 처리 방식에 대한 이해들을 다양하게 경험해 볼 수 있었다.

5) 모델 architecture 생성

- Attention을 활용한 architecture를 형성함에 따라 Attention에 대한 이해가 필요했다. 해당 과정에서 Attention에 대한 스터디를 진행하였는데 최근 Transformer가 주요하게 활용되고 있음에 따라 해당 방식을 이해하는 것이 유익했다. 더불어 지속적으로 좋은 성능을 보이는 새로운 모델들이 등장하고 있으나 해당 모델에 대한 이해를 기반으로 구현을 해보거나 이해하는 과정이 필요하다는 것을 느낄 수 있었다.
- 구현 뿐 아니라 다양한 연구들에 대한 논문 리딩에 대한 필요성도 느꼈는데, 다양한 선행 연구를 기반으로 해당 방법들을 활용해 나갈 수 있어야 할 것 같다.
- Encoder의 경우 Attention을 활용하지 않기 때문에 Encoder 모듈을 만들고 call함수를 정의해 dense layer와 relu 층을 지날 수 있도록 정의했다.
- Decoder에서 Attention을 활용하기 때문에 Attention model을 형성했는데, score 계산 위한 차원 변경, score function을 구성하고 가중치 행렬과 곱해 score를 계산하는 과정, attention distribution과 그 각 값인 attention weight을 구하는 과정, attention value를 구하는 과정을 수행해 볼 수 있었다. attention value는 특히 attention에서의 encoder에 대한 attention weight과 hidden state에 대한 가중합으로 문맥을 포함해 context vector로 불린다고 한다.
- Decoder에서 attention을 통해 context vector와 attention weight를 추출해 연산에 활용하였다. input에 대한 embedding을 수행하고 attention으로부터의 context vector를 concat하여 GRU, dense layer를 거치게 했다.

6) 모델 학습

- optimizer, loss function을 정의하고 학습을 수행했다. 학습 당시 colab 사용시 gpu 한계가 있어 kaggle notebook을 활용해 학습을 수행했다. epoch의 경우 학습 시간의 한계로 5로 지정하여 수행하였다.

7) 시행착오

- dim size가 맞지 않는 문제가 있었다. 해당 error message를 통해 trouble shooting 하는 과정이 있었다. shape을 맞추는 과정, tensor들을 적합한 형태로 조정하는 연습이 필요하다.
- 모델 학습시 loss가 제대로 나타나지 않아 nan이 나오는 문제가 있었다. loss nan이 발생되는 몇몇 요인들을 파악하고 해당 내용에 대한 처리를 해당 프로젝트 이외에도 시도해봐야겠다.

[후첨1] 사용 라이브러리

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import collections
from collections import Counter
import string
import time

# for deep learning
import tensorflow as tf
import keras

# 이미지 처리 및 자연어 처리
from tensorflow.keras.utils import load_img
from PIL import Image
from IPython import display
from sklearn.model_selection import train_test_split
from keras.preprocessing.text import Tokenizer
# progress bar
from tqdm import tqdm

# import models
from tensorflow.keras.applications.inception_v3 import InceptionV3
from tensorflow.keras.models import Model

# for modeling
from tensorflow.keras import layers
from tensorflow.keras import activations
from tensorflow.keras import Input

# 파일 경로명을 통해 파일 리스트를 뽑기 위한 라이브러리
import glob
import os
```