Task 2 -Code Implementation

Ankit Varma, Irfan, Pratik and Vimal

Documentation of the code:

Importing necessary libraries:

```python
import os
import pickle
import numpy as np
from tqdm.notebook import tqdm


from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical, plot_model
from tensorflow.keras.layers import Input, Dense, LSTM, Embedding, Dropout, add
```

- The code starts by importing necessary Python libraries, including os for operating system functions, pickle for serialization, NumPy for numerical operations, and tqdm for progress bars.
- Importing various modules from the TensorFlow Keras library, including the VGG16 model, image preprocessing functions, Tokenizer for text processing, pad_sequences for sequence padding, Model for creating neural network models, and various layers for constructing the architecture.

Loading vgg-16 model

```python
# load vgg16 model
model = VGG16()
```

```
# restructure the model
model = Model(inputs=model.inputs, outputs=model.layers[-2].output)
# summarize
print(model.summary())
```

- The VGG16 model typically consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers.
- This line creates a new model (model) by specifying its inputs and outputs.
- The inputs are set to be the same as the original VGG16 model,
- and the outputs are set to be the output of the second-to-last layer (model. layers[-2].output). This is a common practice when using pre-trained models for feature extraction.
- The last layer (output layer for classification) is excluded, and the new model focuses on extracting features.

Extraction of features from the images:

Python

```
# extract features from image
features = {}
directory = os.path.join(BASE_DIR, 'Images')


for img_name in tqdm(os.listdir(directory)):
  # load the image from file
  img_path = directory + '/' + img_name
  image = load_img(img_path, target_size=(224, 224))
  # convert image pixels to numpy array
  image = img_to_array(image)
  # reshape data for model
  image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
  # preprocess image for vgg
  image = preprocess_input(image)
  # extract features
  feature = model.predict(image, verbose=0)
  # get image ID
  image_id = img_name.split('.')[0]
  # store feature
  features[image_id] = feature
```

- Initializes an empty dictionary (features) to store image features.
- Specifies the directory where the images are stored (directory).
- Iterates through each image file in the specified directory.
- Loads, preprocesses, and reshapes each image for compatibility with the VGG16 model.
- Uses the VGG16 model to predict features for each image.
- Extracts the image ID from the filename and stores the corresponding feature vector in the features dictionary.
- This process results in a dictionary (features) where each image ID is associated with its corresponding feature vector obtained from the VGG16 model.

Storing the extracted features of the image in a pickle file in the form of a binary

```python
# store features in pickle
pickle.dump(features, open(os.path.join(WORKING_DIR, 'features.pkl'), 'wb'))
```

Using the pickle module, the code stores the extracted image features (from the VGG16 model) in a binary file named 'features.pkl'. The serialized features are written to the file in the working directory.

Loading the caption data:

```python
# create mapping of image to captions
mapping = {}
# process lines
for line in tqdm(captions_doc.split('\n')):
  # split the line by comma(,)
  tokens = line.split(',')
  if len(line) < 2:
    continue
```

```
    image_id, caption = tokens[0], tokens[1:]
    # remove extension from image ID
    image_id = image_id.split('.')[0]
    # convert caption list to string
    caption = " ".join(caption)
    # create list if needed
    if image_id not in mapping:
      mapping[image_id] = []
    # store the caption
    mapping[image_id].append(caption)
```

In summary, this code processes a document containing image captions, extracts relevant information (image ID and captions), and creates a mapping of image IDs to lists of captions. The resulting mapping dictionary can be used for tasks like training image captioning models.

Preprocessing the text data:

The preprocessing steps include converting captions to lowercase, removing digits and special characters, eliminating additional spaces, and adding start and end tags to each caption. The processed captions are then updated in the original mapping.

Tokenizing the text:

Python

```
# tokenize the text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(all_captions)
vocab_size = len(tokenizer.word_index) + 1
```

This code snippet involves tokenizing text using the Tokenizer class. It fits the tokenizer on a list of all captions (all_captions), which essentially builds a vocabulary and assigns a unique index to each word. The vocab_size is then calculated as the total number of unique words in the vocabulary, plus one.

Training and splitting the data:

```python
image_ids = list(mapping.keys())
split = int(len(image_ids) * 0.90)
train = image_ids[:split]
test = image_ids[split:]
```

In summary, this code organizes a list of image IDs into training and testing sets, with 90% of the data used for training (train) and the remaining 10% for testing (test).

Creating our model:

```python
# encoder model


# image feature layers
inputs1 = Input(shape=(4096,))
fe1 = Dropout(0.4)(inputs1)
fe2 = Dense(256, activation='relu')(fe1)
# sequence feature layers
inputs2 = Input(shape=(max_length,))
se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
se2 = Dropout(0.4)(se1)
se3 = LSTM(256)(se2)


# decoder model
decoder1 = add([fe2, se3])
decoder2 = Dense(256, activation='relu')(decoder1)
outputs = Dense(vocab_size, activation='softmax')(decoder2)


model = Model(inputs=[inputs1, inputs2], outputs=outputs)
model.compile(loss='categorical_crossentropy', optimizer='adam')


# plot the model
plot_model(model, show_shapes=True)

# train the model
epochs = 20
```

```
batch_size = 32
steps = len(train) // batch_size


for i in range(epochs):
  # create data generator
  generator = data_generator(train, mapping, features, tokenizer, max_length,
vocab_size, batch_size)
  # fit for one epoch
  model.fit(generator, epochs=1, steps_per_epoch=steps, verbose=1)


# save the model
model.save(WORKING_DIR+'/best_model.h5')
```

Encoder Model:

Image Feature Layers (inputs1, fe1, fe2):

- inputs1: Defines an input layer for image features with a shape of (4096,), representing the size of the VGG16 feature vector.
- fe1: Applies dropout with a rate of 0.4 to the image input.
- fe2: Adds a dense layer with 256 units and ReLU activation to process the image features.

Sequence Feature Layers (inputs2, se1, se2, se3):

- inputs2: Defines an input layer for the textual sequences with a shape of (max_length,), where max_length is the maximum length of the captions.
- se1: Embedding layer with a vocabulary size of vocab_size (number of unique words) and an embedding dimension of 256. It also masks zero values in the sequence.
- se2: Applies dropout with a rate of 0.4 to the embedded sequence.
- se3: LSTM layer with 256 units to process the embedded sequence.

Decoder Model:

- Combines the features from the image and sequence using the add layer.
- Applies a dense layer with 256 units and ReLU activation (decoder2) to further process the combined features.
- The final output layer (outputs) is a dense layer with vocab_size units and softmax activation to predict the probability distribution over the vocabulary.\

Model Compilation:

- The model is compiled using categorical cross-entropy loss and the Adam optimizer.
- Model Plotting:
- The architecture of the model is visualized using the plot_model function.

Training:

- The model is trained for 20 epochs with a batch size of 32.
- A data generator (data_generator) is used to generate training data on the fly.
- The fit function is called to train the model for one epoch, iterating over the specified number of steps per epoch.

Generating Captions for the images:

Python

```python
# generate caption for an image
def predict_caption(model, image, tokenizer, max_length):
  # add start tag for generation process
  in_text = 'startseq'
  # iterate over the max length of sequence
  for i in range(max_length):
    # encode input sequence
    sequence = tokenizer.texts_to_sequences([in_text])[0]
    # pad the sequence
    sequence = pad_sequences([sequence], max_length)
    # predict next word
    yhat = model.predict([image, sequence], verbose=0)
    # get index with high probability
    yhat = np.argmax(yhat)
    # convert index to word
    word = idx_to_word(yhat, tokenizer)
    # stop if word not found
    if word is None:
      break
    # append word as input for generating next word
    in_text += " " + word
    # stop if we reach end tag
    if word == 'endseq':
      break

  return in_text
```

Input Parameters:

model: The pre-trained image captioning model.
image: The feature vector of the input image.
tokenizer: The tokenizer is used to convert between words and indices.
max_length: The maximum length of the generated caption.

Initialisation:

The initial input text (in_text) is set to 'startseq' to initiate the caption generation process.

Caption Generation Loop:

- The function iterates over a maximum sequence length (max_length) to predict each word in the caption.
- Inside the loop:
- The current input sequence (in_text) is encoded using the tokenizer.
- The sequence is padded to match the expected input length.
- The model predicts the next word using the provided image features and the current sequence.
- The word with the highest predicted probability is selected.
- The selected word is appended to the input text for generating the next word.
- If the end tag ('endseq') is predicted, the loop is terminated.

Giving inputs to the model and visualizing the output:

```python
from gtts import gTTS
from IPython.display import Audio, display, Image


# Your existing code for image loading, VGG feature extraction, and caption prediction
image_path = 'umbrella.jpg'
image = load_img(image_path, target_size=(224, 224))
image = img_to_array(image)
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
image = preprocess_input(image)
feature = vgg_model.predict(image, verbose=0)
```

```
predicted_caption = predict_caption(model, feature, tokenizer, max_length)
predicted_caption = predicted_caption.replace('startseq', '').replace('endseq',
'').strip()
print(predicted_caption)


# Display the image in the Jupyter Notebook
display(Image(filename=image_path))


# Convert the predicted caption to audio and save it to a file
tts = gTTS(text=predicted_caption, lang='en', slow=False)
audio_path = 'predicted_caption.mp3'
tts.save(audio_path)


# Play the saved audio file in the Jupyter Notebook
display(Audio(audio_path, autoplay=True))
```

Image Processing and Caption Prediction:

- An image is loaded (image_path = 'umbrella.jpg') and preprocessed for compatibility with the VGG16 model.
- The VGG16 model (vgg_model) is used to extract features from the image.
- The image features are passed to the predict_caption function to generate a caption (predicted_caption).

Display the Image:

- The original image is displayed in the Jupyter Notebook using the display function.

Convert Predicted Caption to Audio:

- The predicted caption is converted to audio using the gTTS (Google Text-to-Speech) library. The language is set to English (lang='en'), and the audio is generated without slowing down (slow=False).
- The generated audio is saved to an MP3 file (audio_path = 'predicted_caption.mp3').