Deep Learning Term Project Report

Team Mind Mavericks (G-37)

In this report, we describe our methodology for creating the two models for image captioning. We also provide the various scores we obtained.

Part A: CNN-based encoder + RNN-based decoder

In this part, we had to create a CNN-RNN architecture for image captioning. The CNN is used to extract the features from the images and then that is passed as context to the RNN. The RNN then does continuous next word prediction based on the context as well as the last hidden state to generate captions one after the other.

The model is able to generalize well to test set images. This implies that our combined model has learned general features quite well without overfitting. Now, we provide the methodology used for this part.

Methodology:

Data Preprocessing

We first loaded the images (of the train set) and the captions from the csv files. The captions were then split into tokens using the **nltk.tokenize.word_tokenize()** function and converted to lowercase. Then the **start** and **padding** token (which serves as end token here) were added to this split. In order to get the vocabulary, we have put every token generated above into a set to get all the distinct tokens . Then every word in the token is assigned an integer value (ranging from 1 to size of vocabulary). This assignment of value to every token is stored in a **dictionary** (called **tokens** in the code) as key value pairs. The aim is, while passing a word to the model, the word is first converted to its key value (say i) which is then transformed to a **one hot encoding** in which the ith bit is set to 1 and rest 0s. This encoding is then passed to the embedding layer.

The images loaded are also transformed into the dimension required for the VGG16 encoder **(224x224x3)** and normalized.

Models

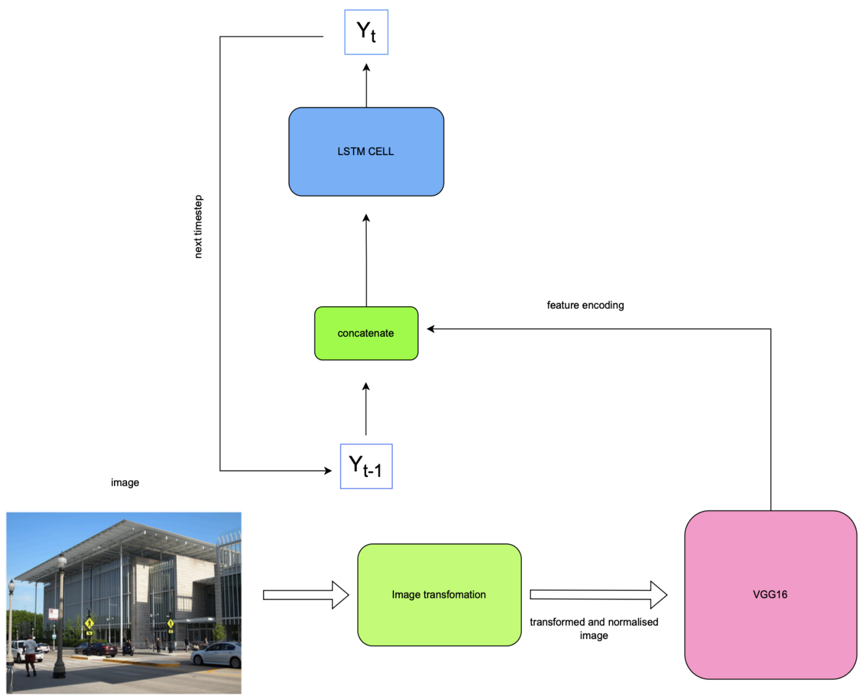
For the CNN Encoder model, we used the **VGG16** model with the pretrained weights from the ImageNet dataset. As the model had already learnt the basic features from a large dataset of images, we **froze** the layers of the CNN except the last linear layer. The last linear layer (trainable) was modified to output a vector of the length ‘embed_size’ instead of the usual 1000 length vector.

The feature vector from the CNN was then passed to the RNN. The RNN comprises a **single LSTM cell** which takes as input the **features** (from the CNN) and a **token** (initially this is the embedding for the start_token). The token creation is done using an embedding layer which takes the one hot encoding of a word and transforms it to a vector of fixed size. This embedding and the features are **concatenated** and supplied to the LSTM cell. This cell uses this input as well the older hidden state and cell state to do language modeling and predicts the embedding for the next word. This output will serve as the **token input** for the next word. This is done till the **padding_token** is predicted or a **max caption length** value is reached.

Training

For training the model, we first collected all the trainable params in a list and passed it to the **Adam Optimiser** with a **low learning rate of 1e-4**. We observed that higher rates like 1e-3 made the model stuck in similar losses after a few epochs while lower rates like 1e-5 made training too slow. We also tried using SGD Optimiser but Adam outperformed it due to its adaptive rate adjustment. The loss used was the **Cross Entropy** loss. We trained the model for **5 epochs** and saw a steady decrease in the loss throughout training.

One notable thing in our training is that we are using **batches of size 1**, i.e. we are passing 1 image at a time. The primary reason for this is that if we add padding tokens to the end (which is needed for batching), the model gives very high weight to the padding tokens and does not learn properly.



Our Basic Pipeline for Part A

Results

We have iterated through the test set and printed every 50th image along with its actual caption and predicted caption. It can be seen that it is quite accurate in more than **50%** of the images. Along with this, we calculate the **BLEU** and **Rouge-L** score for each image in the test set and print the average in a later cell. We have obtained a BLEU score of **0.08** (out of 1) and Rouge-L score of **0.15** (out of 1). The scores are somewhat low but still the caption generation is good in general. We have stored the captions generated of our Part A model in a **result.txt** file. This is so that we can later visit the captions and get the required inference. It also helped us to recalculate the **BLEU** and **Rouge-L** scores.

Part B: Visual Transformer as Image encoder and Text decoder

We are setting up a vision encoder-decoder model for text generation, specifically for text summarization. We define the encoder as a Vision Transformer (**ViT**) pre-trained on images, and the decoder as a **GPT-2** model, which is a transformer-based language model known for its text generation capabilities.
We initialize tokenizers for both the encoder and the decoder, using the **GPT2TokenizerFast** from the transformers library. The image processor is initialized using the **ViTImageProcessor** class from the transformers library, which is specifically designed for processing images with Vision Transformers.
The VisionEncoderDecoderModel is then instantiated with the pre-trained encoder and decoder models. We configure the model to use the appropriate start and end of sequence tokens for decoding. In this case, we set the decoder's start token as the beginning-of-sequence token (**bos_token**) and the pad token as the end-of-sequence token (**eos_token**).

Methodology:

Data Preprocessing

1. Initialization: Initially we set hyperparameters such as **max_length** for the maximum length of captions in tokens and **batch_size** for the batch size.
2. Folder and File Paths: We have defined paths to the training, testing, and validation folders containing image files, as well as paths to CSV files containing captions corresponding to those images for easier access.
3. Data Preprocessing: In the first step of data preprocessing, we examine each image in the dataset using the **Image.open()** function from the PIL library for its dimensions. If an image's dimensions don't match the expected 3 or 4, suggesting it's invalid, we note down its filename in a list called **invalid_images**. Then, we go through each dataset segment (train, test, validation) and filter out any rows linked to file names listed in the invalid_images list. This step ensures that only images with valid dimensions, along with their respective captions, remain in the DataFrames. Additionally, we clean up any rows containing missing or NaN values. Finally, we reset the invalid_images list to prepare for future preprocessing steps, maintaining a clean slate for subsequent operations.

Models

In this setup, we've selected two key components for our model architecture. Firstly, we've chosen the "**google/vit-base-patch16-224**" model as our encoder, representing the Vision Transformer (**ViT**) specifically trained on image data. Secondly, we've opted for the "**gpt2**" model as our decoder, which is renowned for its text generation capabilities and is based on transformer architecture.

To facilitate seamless integration of these components, we've initialized specific tools from the transformers library. The **GPT2TokenizerFast** is utilized as the tokenizer, tailored for efficient tokenization compatible with the GPT-2 model. Additionally, the **ViTImageProcessor** is employed for preprocessing images, ensuring they are properly formatted for input into the vision encoder.

For model instantiation, we utilize the VisionEncoderDecoderModel, instantiated through the **from_encoder_decoder_pretrained()** method. This initializes the model with pre-trained weights for both the encoder and decoder selected earlier. Subsequently, the model is transferred to the designated device for computation. Lastly, various configuration parameters such as **pad_token**, **eos_token_id**, **pad_token_id**, and **decoder_start_token_id** are configured to optimize model performance and behavior.

<u>Training</u>

The **Seq2SeqTrainer** from the transformers library is instantiated to handle model training. It requires the model, tokenizer, training arguments, compute metrics function, and datasets for training and evaluation.
The **optimizer (AdamW)** is defined and initialized with the model's parameters using a **learning rate of 1e-5.**
The provided code implements a training loop for a sequence-to-sequence model over **10 epochs**, with a **batch size of 16**.
During each epoch, the model is trained on the training dataset, and its performance is evaluated on the validation dataset. The training loop includes steps to update the model's weights based on training data, calculate both training and validation losses, and compute evaluation metrics such as **BLEU score and Rouge-L score**. It can be seen that both the scores steadily increase to **0.18** and **0.468** respectively. (out of 1)

<u>Results</u>

We have iterated through the test set and printed the first 10 images along with their actual caption and predicted caption. It can be seen that it is quite accurate in more than **60%** of the images. Along with this, we calculated the **BLEU** and **Rouge-L** score for each image in the test set and printed the average in another cell. We have obtained a BLEU score of **0.1750** (out of 1) and Rouge-L score of **0.463** (out of 1). The scores are much better than our previous part which is to be expected as transformers are state of the art models.