Implementation of Various Neural Network Models for Image Captioning

Berfin Kavşut, Beste Aydemir, Ege Ozan Özyedek
Electrical and Electronics Engineering Department, Bilkent University, Turkey
EEE 443 Neural Networks Fall 2020-2021 Final Project

Email: berfin.kavsut@ug.bilkent.edu.tr, beste.aydemir@ug.bilkent.edu.tr, ozan.ozyedek@ug.bilkent.edu.tr
Kaggle Notebook: https://www.kaggle.com/ohnochateau/eee443-project
OneDrive: https://1drv.ms/u/s!AuqIruPkk3lWg8tdXfre5tbS7xx0YQ?e=ssPzsE

***Abstract*—Image captioning task is generating a sequence of appropriate words for a given image. With recent advancements in neural networks, there has been a progress in implementing image captioning. Neural network architectures containing Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) are utilized to create captions that capture information about the contents of an image. In this project, various architectures were applied using TensorFlow Keras in order to compare their image captioning performances. Instances of Merge, Par-Inject, Pre-Inject and Init-Inject models and their modified versions along with one Bidirectional Multimodal Model were implemented. Model losses and training times were used to compare learning and computational performances. Also, evaluation metrics were calculated to measure aptness of generated captions to reference captions in the dataset. Moreover, sample images were examined to understand the performances further. Analyses of the results provide a general comparison of various models and suggest that implemented models can provide acceptable captions to images.**

I. Introduction

The goal of this project is to generate appropriate captions for a given image. The captions will be chosen in order to capture the contextual information on the images. Current methods utilize convolutional neural networks (CNNs) and recurrent neural network (RNNs) or their variants to generate appropriate captions. These networks provide an encoder-decoder method to achieve this task, where CNNs encode the image into feature vectors and RNNs are used as decoders to generate language descriptions [1].

The architecture of models can vary significantly. For the encoder part, a pre-trained CNN is used to extract features from an image. These features are taken from specific layers after the image is inputted to the CNN. Then, the feature vector is directed to image captioning architecture. Most commonly used CNNs for decoding are AlexNet, VGGNet, ResNet and GoogleNet (Inception models) [2].

For the decoder part, different RNNs are used. LSTM (Long Short Term Memory) are particularly useful for dealing with long term dependencies in sequences. Words from defined dictionaries are inputted to LSTMs by embedding. Words are converted to vectors by using statistical methods and algorithms such as Word2Vec or GloVe. These pre-trained methods employ techniques such as "Continuous Bag-of-Words", "Continuous Skip-Gram Model" and "Latent Semantic Analysis" [3]. First two (from Word2Vec) methods give embeddings by predicting a word from a window of words or finding the context from a word. Similarly, GloVe aims to capture co-occurrences of the words with "Latent Semantic Analysis" method [4]. The words in a certain dictionary are selected from these embedding matrices by multiplying with one hot representations. Moreover, this embedding matrix becomes trainable and open to fine tuning if the task requires. After matching the words in a certain dictionary to embedded vectors with these methods, the LSTM can use word-context vectors as inputs. Also, it is important to note that the least frequently used words are dropped from the dictionary and replaced with an umbrella unknown keyword.

Image features and word embeddings can be combined in various ways. Architectures can be divided to four general categories according to where the image and captions are located in the architecture: Init-Inject (image information or its variant is given to initial states of LSTM), Pre-Inject (image is inputted as the first time step), Par-Inject (image information and sequence information is somehow combined at each time step), Merge (image and sequence are combined after LSTM) [5]. Also, an architecture can implement a combination of these categories (e.g. Par-Inject and Init-Inject). These models can be implemented with varying RNNs, such as GRU (Gated Recurrent Unit) and Bidirectional LSTMs. In addition to these, more complex architectures can be developed. Attention mechanism focuses on certain part of the image and includes that information in LSTM time steps. Semantic concept based models extract semantic context information from CNNs and use that in sequence generation [6].

Large labeled data sets such as Microsoft COCO and Flickr image data sets contain image and caption pairs that capture information about the events and objects. These captions can be used to create appropriate descriptions to new images. In order to measure the aptness and relevance of the generated captions

to the data set captions, metrics such as BLUE [7], CIDER [8], METEOR [9] and ROGUE [10] are used. Roughly, BLEU score depends on the caption length and general similarity, ROUGE focuses on the text summary and METEOR uses a generalized unigram (one word) match between reference and generated functions [1]. CIDER score is based on human consensus and designed to be similar to how humans evaluate similarities in captions [8].

II. Methods

The methodology for the problem of image captioning can be divided into five main parts. The process of captioning an image starts by obtaining, cleaning and processing data, and then extracting features from the pre-processed images. Then, the network models are prepared for training. The pre-processed data is used for training on the prepared network models. Finally, the evaluation of the model is done on separate test data. For the entirety of the project, Google's TensorFlow library (and also its deep learning counterpart Keras) was used. We have utilized multiple development platforms, but Kaggle was the main development platform. We chose Kaggle since it provided us with an easy way of storing data and also because of its GPU support which decreased training times of the network models. Below, each section will undermine the followed methodology of each stage of the project.

A. Data Processing

The training and test data provided for the project consists of different data required for image captioning. The train data and its description can be viewed below, with its test counterpart given in brackets. It should be noted that for this project, the provided feature datasets "train_ims" and "test_ims" were **not used**. Data processing and feature extraction is a part of the submitted project.

- train_url [test_url]: A numpy array containing 82783 [40504] Flickr image URLs. The URLs include a range of different types of pictures, both indoors and outdoors. Animals, food, sports, indoor locations such as kitchens and bathrooms, outdoor locations such as train stations and many different scenarios are contained in the given dataset. In Figure 1, four sample images are shown.

- train_cap [test_caps]: The array which contain captions for the images contained in the "url" arrays. The captions are not stored as string literals but rather stored as integers. These integers can then be used in coalition with the word_count array to get a string representation of the descriptions. The caption amount for each picture varies, which is an important quality to keep in mind while writing the code.

- train_imid [test_imid]: An array containing indexes of the URL images. For example, the first value (at index zero) in this array is 53315. This represents the picture URL in the index (53315 - 1) of the train_url array [the minimum index of test_imid is zero hence we would do (53315 - 0) for test captions]. The values in the "imid" arrays can then be used to find corresponding captions in the "cap" arrays by using a numpy method called "where". Hence, the "imid" arrays are used to find the connection between the captions and its corresponding URL.

- word_count: A dictionary like array which contains words and their corresponding indices. This array has 1004 words, meaning our vocabulary size for the project is 1004. There are 4 notable special words, these are "x_START_" (used for start of captions), "x_NULL_" (used for zero-padding of captions), "x_END_" (used for end of captions) and "x_UNK_" (used for unknown words). Also, there were several words in the dictionary, e.g. "xWhile" and "xFor", which were just the word with an "x" added in front. These were corrected in the embedding matrix.

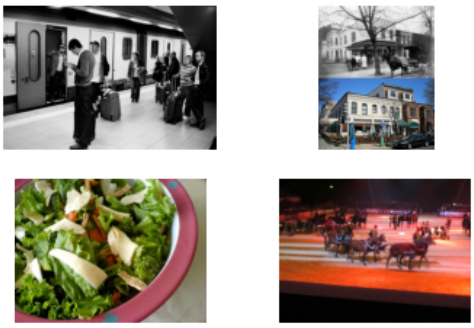

Figure 1. Sample images from train data.

The data collection and processing is very important since it creates the base of the project. Keeping this in mind, we have determined a road map to create a clean, easily understandable and efficient dataset for the prepared network models to use. Below, some important data processing steps are explained.

The first step was to acquire the JPEG images from their respective URLs. We have discussed different ways of using these URLs to extract the images. In one instance, we have discussed extracting features instantaneously from the URLs themselves. This idea proved to be highly inefficient and slow however, and in the end we decided it was best to download all pictures and use the ".jpg" files. The URLs provided were not "clean", some URL links were broken and the corresponding image was unreachable. To understand which URLs were broken and which would provide us with images, we checked the response status code, which would be 200 if the connection was successful [11]. This way, we were able to download both train and test images. This step was not enough to clean the data, some other problems occurred after obtaining the images. Some image names had white space after their type name

(.jpg) which would prevent it from being used on Kaggle. This was solved by traversing through all file names in the directory and stripping them from whitespace. There was also another notable problem with the test data, two images were inaccessible. These images were deprecated by decoding the images and checking whether they were actually JPEG images [12], [13]. For the train dataset, we recovered 87.7% of the provided images and for test, this percentage was 87.5%.

After the cleaning step was done, it was time to make an understandable and efficient dataset for the network models to use. Understandable in this context means that the code accesses the image features and its corresponding captions with the least resistance, and we should also be able to understand the data. The train and test datasets provided are not organized, and hence several different arrays were required to simply extract the captions (a simplified explanation of this is given above in the discussion about the data). The indexing system of the URL, caption and image id (named "imid") datasets required too many operations and was confusing. Following these observations on the existing data, we decided to keep the important information only. These were the image features, the image's corresponding captions and the name of the image. The name of the image file is used to link the captions to the images, similar to the way train_imid is used.

To reduce memory usage, we have used Tensorflow's Dataset library [14]. The Dataset library does not load all information into the memory, but rather loads only the information to be used at the moment its called. This means we do not have to load all ≈70,000 train image features and their corresponding captions (which would be inefficient) but only the ones that we will use in a time frame (only a batch of 64 image features at a training step for example). In the construction of this dataset, all image files are pre-processed for feature extraction and their names and caption arrays are obtained. Then the features, captions and name are put into a Python tuple, which is then stored in a file using Python's pickle library. The stored file is then linked to a Python generator function, and this function is used to create the final rendition of the dataset to be used. As the Dataset object is called, it obtains data from the "pickle" file sequentially, and returns a tuple of tensors that contain the features, captions and name of a single image.

B. Feature Extraction

The first half of an image captioning problem can be defined as the feature extraction part. Feature extraction summarizes the image information to a single vector. For the purposes of this project, we have decided to use Google's Inception v3. Inception v3 is a state of the art image recognition model which has its final output as the softmax function [15]. By deleting this layer we can obtain and use the features the network outputs.

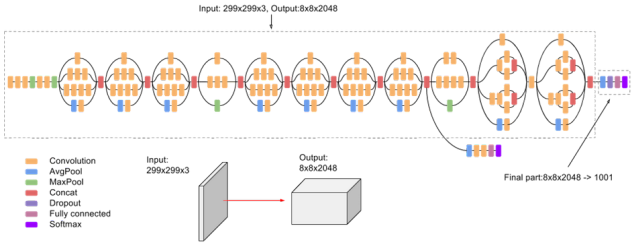

Figure 2. Representative image of the feature extraction network, Inception v3 [15].

To use the Inception model, some preprocessing is required. Inception takes 3-channel (RGB) (299, 299) images as inputs. The images provided to us varies in dimension, hence, we resize all images to the size mentioned using Tensorflow's image library. Then, the resized image is normalized to the interval [-2, 0] before entering the Inception network [16]. After the features are extracted, they are dumped into a file and then recovered as a Tensorflow Dataset to be used in model training/testing.

C. Model Architectures

In our models, we used many-to-one RNNs. Captions are divided into partial captions as prefixes and target word is the next word after the prefix. Let us say our caption is "<start> an empty kitchen with white and black appliances <end>". First inputs will be "<start>" and image vector and target will be "an". Second inputs will be "<start> an" for the same image, and target word will be "empty". Similarly, third input will be "<start> an kitchen" with image vector and output will be "with". In Figure 3, RNN structure with its inputs and output is shown.

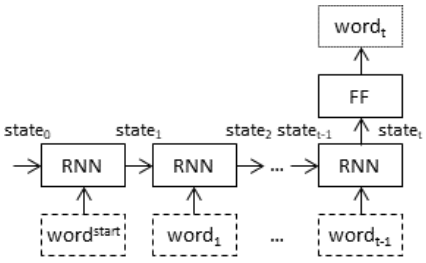

Figure 3. Many-to-One RNN Structure [5].

Partial captions are created until the maximum length of captions is reached. When partial captions are given as inputs, zero padding is applied to fill them so that their sizes will be identical. This procedure is applied for each image, which means images are replicated until the maximum length of captions is reached. This idea is taken from similar projects and our data generators are written with this principle of creating partial captions and their targets for each image [17].

There are two types of usage of RNNs for image captioning. The first one is encoding partial captions, and the second is using them as generators. Their purpose is dependent on where the image vector is included in our models. If image vector is used at the stage of RNNs, then these architectures are called as Inject Architectures. Image vectors are injected to our model and RNNs are generating the target word. If partial captions are fed to RNN and image vector is combined with output of RNN, i.e. RNN is used as encoder, then these architectures are called as Merge Model because image vector and output of RNN are merged. [18].

It is important to decide where to put the image in an image caption generator. As explained before, they can be put before or after RNNs. Their location of integration to architectures makes them differ from each other [5]. There are four models implemented in this project and their performances will compared to show the model with best performance. They are named with the type of architecture they are utilizing. Models are Init-Inject Model, Pre-Inject Model, Par-Inject Model, and Merge Model.

1) Init-Inject: In Init-Inject architectures, image vector is set as initial hidden state vector. Size of image vector is the same with the size of hidden state vector of RNN [5]. Since the images are given as initial hidden states, RNN has the information of image vector from the beginning and hidden state updates are related to initial hidden state. We are using the information of images from the beginning and it is expected to have high performance of target word prediction.

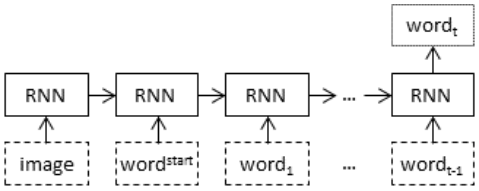

Figure 4. Init-Inject Architecture [5].

2) Pre-Inject: In Pre-Inject architectures, image vector is fed to RNN as first input, which means image vector is treated as the first word vector, and words of partial caption are fed as following inputs to RNN. It is similar to the idea of Init-Inject. In contrast, image vector is not given as initial hidden state but initial hidden state is updated by giving image vector as first input. We are giving the information of image from the beginning so that the model will learn how to guess next words of prefixes, i.e. partial captions, with image and image vector is not merged to the model after RNN encodes partial caption sequence [5].

3) Par-Inject: For these architectures, image information and sequence information are combined as inputs at each time step of LSTM. They can be two separate inputs or they can be combined by concatenation or other methods [5]. The image vector can change at every step, this is the case for attention models. For our implementation, similar to [19], we implemented element-wise multiplication of the image vector and the embedded word vector coming from LSTM. Then, this is directed to two dense layers ending with softmax. Also, dropout layers are used after image vector input and embedding layer for generalization purposes. This model could be improved by removing teacher enforcing at every time step. In [19] the image feature vector is multiplied with the previously predicted word, in contrast to our model. Also, LSTM cells can be modified (with time dependent guided LSTM called td-gLSTM [19]) in order to provide better guidance at each time step.

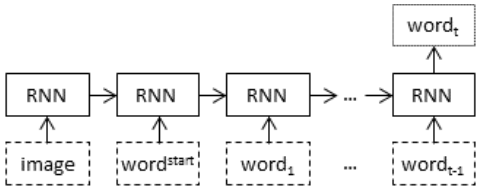

Figure 5. Pre-Inject Architecture [5].

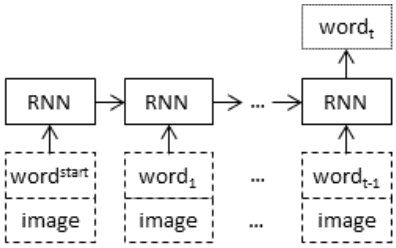

Figure 6. Par-Inject Architecture [5].

4) Merge: In Merge architecture, RNN is used as encoder for linguistic features. Output of RNN is later merged with image vector. We merged image vector and output of RNN via addition, therefore word vectors and image vector should have the same size [5]. Different approaches for merging could be applied. For instance, they could be also concatenated.

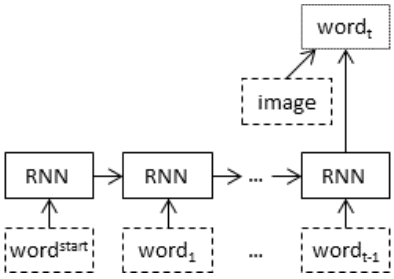

Figure 7. Merge Architecture [5].

5) Bidirectional LSTM Architecture: It is also possible to incorporate Bidirectional LSTMs in order to capture future values in a sequence. By reading the corpus backwards, it is possible to also understand upcoming word embeddings

in a sequence. In Bidirectional LSTMs, two hidden layers propagate in opposite directions, resulting in twice the memory space for weights and biases [20]. In [21], three parts are used. CNN for feature extraction, one LSTM for encoding texts and another LSTM called Multimodal LSTM. Both LSTMs are bidirectional.

After image features I are extracted and text LSTM is used for sentence inputs in both directions $\overrightarrow{S}$, $\overleftarrow{S}$, they are directed to Bidirectional Multimodal LSTM. This multimodal LSTM aims to "capture correlation of visual context and words at time steps" [21]. In our implementation, the input to Multimodal LSTM are obtained by concatenating I and text LSTM output. In contrast to implementation in [22], we used Layer Normalization instead to Batch Normalization. Also, dropout layers are used in all steps to obtain better generalized results and avoid overfitting. The architecture diagram can be found in Appendix.

D. Model Preparation

Implemented models are in Appendix part. All of the models have embedding layer for word representations. glove.6B.zip file is used for creating pre-trained embedding matrix. The dataset has 6B tokens, and 400K words. Vectors with dimension of 50, 100, 200 and 300 are available in the official website of GloVe [23]. We used 200d word vectors. Word vectors are taken according to our dictionary and embedding matrix is created by placing vectors with respect to the index of words. Weight matrix of embedding layers are set from this embedding matrix.In embedding layer, word vectors are taken from the embedding matrix with respect to word indices.

Some words in our dictionary are not available in GloVe dataset: x_NULL_, x_START_, x_END_, x_UNK_, xWhile, xFor, xCatch, xCase, xEnd. Word vector for x_UNK_ is taken from the word vector of unk in GloVe, which is representative of unknown keyword. After initializing weight matrix of embedding layers with the found embedding matrix, we set the layer to be trainable so that the model could learn how to represent the words which are unavailable in GloVe dataset.

Firstly, we started with Core Models of Init-Inject, Pre-Inject, Par-Inject and Merge architectures. As an addition to Core Models, we created Modified Models for better training time and model performance. Layer normalization is similar to batch normalization. Batch normalization is a common application especially in CNNs. Even though input images are preprocessed, parameters do change during training, and hence layer inputs need to be normalized again. With batch normalization, models become more robust to high learning rate and bad initialization of parameters. Also, saturating nonlinearities becomes harder [24].

It is shown that using layer normalization instead of batch normalization is easier to implement in RNNs. Moreover, it has better performance compared to batch normalization in RNNs. It was shown that layer normalization is good at stabilizing hidden states in RNNs and reduces training time [25]. Layer normalization is applied at each time step inside RNN in our models. We have seen the benefit of layer normalization in loss functions as it can be observed in Figure 10 and 11. Training loss and validation loss decreased faster, which means training time decreases with layer normalization as expected. Model structures are shown in Appendix. First, each model structure will be explained and then modifications will be discussed. Modifications are taken from similar applications in [5]. In Core Models, feature extracted image vectors are passed through dropout layer and then fully connected layer. Partial caption sequences are passed through embedding layer for word representation and then dropout layer. Dropout layers are used as regularizers for avoiding overfitting, and dropout value is 0.5 for all dropout layers. Intermediate layers were explained Model Architecture part. Size of hidden layers are 256. ReLU is used for activation function of all layers except output layer. Location of RNN layers and where the image vectors are fed into the models differ accordingly. After generating target word, outputs are passed to one more fully connected layer. In output layer, there is a fully connected layer with softmax activation function. Output layer size is 1004, which is number of words. Softmax activation gives the probabilities for target word. By taking the argmax argument of output, classification task is finalized.

As an addition to modification with respect to similar applications, layer normalization is implemented in our Modified Models. Modifications will be explained model by model. In Merge Model, embedding dropout is removed, image dropout is removed and RNN dropout is added. Layer size is changed from 256 to 128. In Init-Inject Model, image dropout is removed and RNN dropout is added. Layer size is changed from 256 to 512. In Pre-Inject Model, image dropout is removed, and RNN dropout is added. Layer size is changed from 256 to 512. In Par-Inject Model, only change is that RNN dropout is added. Modifications are taken from [5].

E. Training

Using the explained models, the training stage takes part. First the training and validation data is separated by 85% and 15%. For all methods, the epoch number is chosen to be 12. For Core Models, batch size is chosen to be equal and it is 64. In Modified Models, batch sizes are 128 for Init-Inject model, 32 for Pre-Inject model, 64 for Par-Inject model and 128 for Merge Model [5]. Loss is found by comparing the generated labels with the argmax argument of the softmax output, which is of size (1, 1004), representing the vocabulary size. Loss function is cross entropy function, which is used for back-propagation. As optimizer, Adam is used with its default parameter selections. The learning rate is 0.001. The

exponential decay rate for the 1st moment estimates (beta_1) is 0.9. The exponential decay rate for the 2nd moment estimates (beta_2) is 0.999. The small constant for numerical stability (epsilon) is 1e-7. "fit" function of Keras is used for training.

For both training and validation, a data generator is used and the model is trained by using these output of these generators. The data generator function creates a batch of training/validation data. The reason why we are using a generator function is because we want to customize the data we are inputting into the models. Using our own data shape was not practical and creating a custom output sequence was needed [26]. This is because of the way predictions occur. The data generator first iterates through the data, this is done batch_size times. The captions of the data are obtained at each iteration, and another iteration window occurs, this time for the amount of captions. Then for each caption, a sequence variable is defined. A final third iteration window is opened, and this time it traverses the 17 length caption elements, which is the maximum length of captions. This iteration window represents the 17 time steps required for the LSTM, and will sequentially fill the train input and the label. At each iteration in this window, an input sequence (in_seq, which represents the last k words of the caption pre-padded with zeros) and an output sequence (out_seq, which is a one-hot-encoded vector that represents the maximum argument that should be the output of the last layer softmax, this is the label) is defined and appended to a list. After each batch_size, the generator returns the features, 17 time step captions and its corresponding labels. This is done for both train and validation datasets.

F. Evaluation Methods

For the implementation of the evaluation, the built in "predict" method of Keras is used. Since this model only predicts the next word for a given sequence (many-to-one, as discussed above), the predict function is called 16 times for each image. At each call, the predicted next word index is appended to the prediction sequence, which then becomes the input of the next prediction. Using this, we were able to predict captions for all test images and then these were evaluated using common evaluation techniques in natural language processing.

In order to compare generated captions to reference captions in the data set BLEU, METEOR, ROGUE and CIDER scores are used from the repository from [27]. For BLEU-n metrics, n-grams are compared regardless of word orders [28]. The score is between 0-1 (or percentage), highest score indicating that the generated caption matches one of the reference captions exactly. During its calculation, too short captions are penalized and overlaps (of length n = 1,2,3,4) are rewarded. However, this score is not successful in capturing long term dependencies and overall meaning [29]. Since a higher n value means longer sequences (n-grams) are matched, it is reasonable to expect BLEU-n scores to decrease as n increases.

ROGUE score is used for summarizing purposes and looks at the general meaning. It checks recall for reference statements, largest common sequences and co-occurrence statistics [30].

CIDER is particularly for image captioning task. It checks frequency for each n-gram and is closer to human consensus. Usually, there are in average five reference captions in training sets. However, this metric uses up to 50 reference captions and returns the metric [8].

METEOR calculates the harmonic mean of unigram precision and recall, and returns similar responses to human consensus. It can show similarities in sentences and segmentation levels for the entire set [30].

III. Results

This section will display the results of the designed networks on test data. It is divided into three sections; training and loss, evaluation metrics and image captioning performance.

A. Training and Loss

Parameters numbers, total training time and average training time for one epoch in minutes are given for all models in Table I. All training is done on the Kaggle platform using GPU acceleration. The training time is in the expected range, considering the fact that models use RNN structures.

Table I
PARAMETERS AND TRAINING MINUTES PER EPOCH FOR DIFFERENT MODELS.

Model	Parameters(Mil)	Minutes	Minutes/Epoch
Core Merge	1.52	102.66	8.56
Core Par-Inject	1.40	99.99	8.33
Core Pre-Inject	1.64	107.98	9.00
Core Init-Inject	1.40	105.20	8.75
Modified Merge	0.78	101.55	8.46
Modified Par-Inject	1.41	115.14	9.59
Modified Pre-Inject	2.12	107.98	8.99
Modified Init-Inject	2.39	108.22	9.01
Bidirectional Multimodal	14.91	316.00	39.50

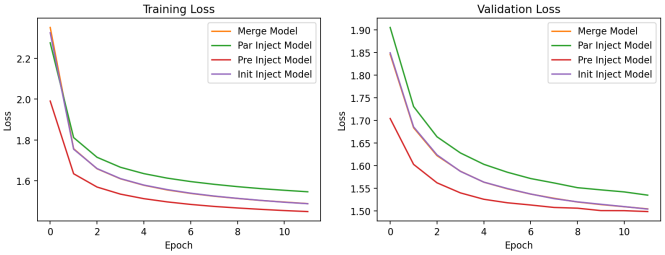

Figure 8. Train and validation loss for Core Models. Epoch number is 12.

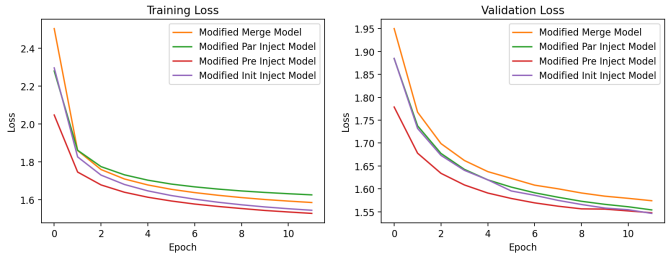

Figure 9. Train and validation loss for Modified Models. Epoch number is 12 for all models.

Training and validation loss of Core Models are shown in Figure 8 and training of validation loss of Modified Models are shown in Figure 9. Validation loss decreases with training loss, which means are models are continuing learning and can generalize to validation set as well. As can be seen in Figure 8 and 9, the minimum loss value reached at the end is from Pre-Inject Model compared to other model types. Pre-Inject Models also has higher evaluation scores compared to other model types.

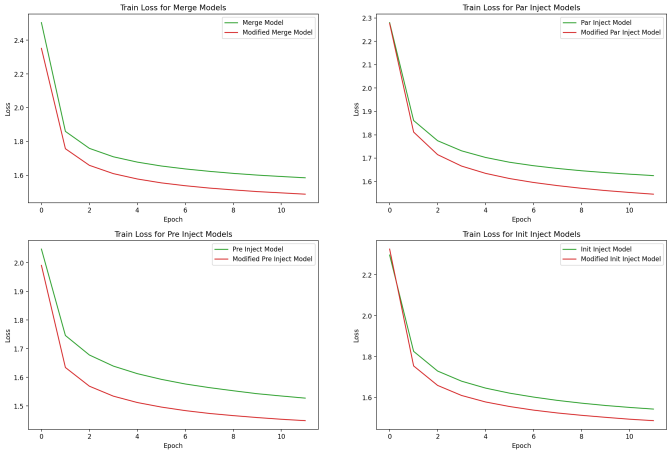

Figure 10. Comparison of train loss for Core Models and Modified Models. Epoch number is 12 for all models.

As can be seen in Figure 10 and Figure 11, both of training loss and validation loss decrease faster for Modified Models. In Table I, it can be observed that we did not gain from training time, which was expected because we had additional operation of layer normalization besides difference in model structure. To sum up, advantage of Modified Models are their rapid decrease in loss functions compared to Core Models.

For the Bidirectional Model, there are over 14 million parameters coming from two bidirectional LSTMs (Table 1). For this reason, training minutes per epoch are higher than other models. Because of increased complexity and larger parameter number of the model, it is prone to overfitting. Modified Merge Model is compared with Bidirectional Multimodal model on Figure 12 since the network architectures are similar and they both utilize layer normalization. The loss decrease is similar

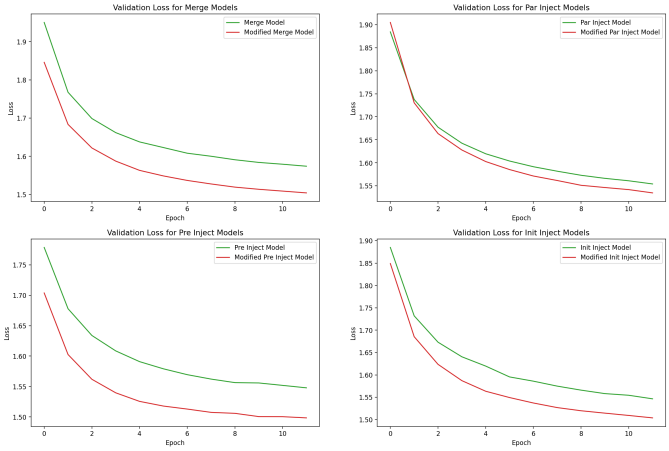

Figure 11. Comparison of validation loss for Core Models and Modified Models. Epoch number is 12 for all models.

in both for the first 6 epochs and because they use layer normalization, the loss decreases rapidly in both. However, in Bidirectional Model it does not converge unlike other models. At the 6th epoch, validation loss starts to increase. This means the model starts to overfit to the set. Learning is stopped early at 8th epoch.

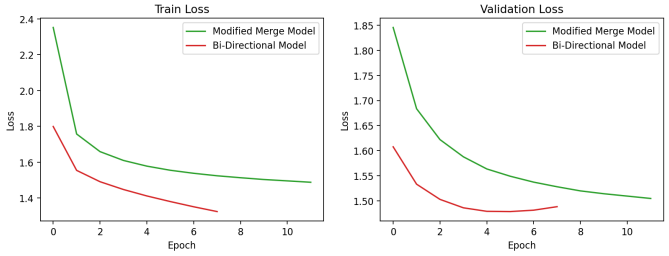

Figure 12. Train and validation loss for Bidirectional Multimodal and Modified Merge Models. Former is trained for 8 epochs and the latter for 12 epochs.

B. *Evaluation Metrics*

The models are evaluated using metrics that measure the semantic similarities between the generated captions from models and reference captions. All metrics for all 9 models can be seen on Table II. The best model in all scores (expect for BLEU-1) is Core Pre-Inject Model. Modified Merge Model has close scores. The worst model (with lowest score in 5 metrics) is Modified Par-Inject Model. In general, all models are similar to each other with close score ranges. But Modified Models resulted in lower evaluation scores, although they had faster convergence.

In order to understand current best values for the evaluation metrics, we researched current state-of-art models and their scores. Microsoft Oscar [31] has the highest BLEU-4 score of 41.7 for image captioning on COCO Captions. Other metrics are CIDER score of 140 and METEOR score of 30.6 [32].

Table II
EVALUATION SCORES FOR 9 DIFFERENT IMPLEMENTED MODELS WITH BEST SCORES (BLUE), WORST SCORES (RED).

Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	METEOR	ROGUE_L	CIDEr
Core Merge	68.6	49.9	34.5	23.8	27.5	49.4	87.4
Core Par-Inject	66.9	48.0	32.6	22.4	27.4	48.0	80.8
Core Pre-Inject	68.7	50.5	35.1	24.4	28.3	49.7	89.3
Core Init-Inject	68.8	50.3	34.8	24.1	27.4	49.6	88.6
Modified Merge	69.0	50.5	34.7	23.6	28.2	49.3	84.1
Modified Par-Inject	66.8	47.9	32.7	22.5	25.6	48.2	80.2
Modified Pre-Inject	68.0	49.8	34.7	24.2	27.1	49.2	85.5
Modified Init-Inject	68.1	49.7	34.5	24.0	25.6	49.4	86.6
Bidirectional Multimodal	68.1	49.3	34.1	23.6	27.6	49.5	85.6

Our best METEOR score of 28.3 is close to their results, but there are clear differences in BLEU-4 and CIDER scores, due to our computational and time constraints and working with simpler models.

C. Image Captioning Performance

Figure 14 and Figure 15 display both successful and unsuccessful predictions of the Pre-Inject Model. Additional caption predictions can also be found in the relevant section of the Appendix. There are several strong points of all designed networks and also some disadvantages. The discussion will be given with the examples from Pre-Inject; however, all discussed points are relevant for other tested models as well.

Starting off with the advantages, by inspecting Figure 14 it can be understood that the network can successfully detect the act, actor and environment (e.g. holding, a woman, tennis court). It should be noted that all three images given here are in well-lit environments where objects can be easily captured by the extracted features. This is also true for other successful captions, as well. The predicted captions also are close to the actual captions, and self-evidently better for the first picture. Another thing that should be noted is the linkage between environment and act. For images 1 and 3, there exists a link between the environment and the act that is ongoing on that environment. In many snowy pictures, the networks understand that people are skiing or snowboarding, and similarly for the tennis court. This is an advantage as the network can match a location with an act; however, it could also prove to be a disadvantage if this linkage is overly used and, for example, every picture containing snow is captioned as snowboarding. In our models, this leans more on the advantageous side and we can reach the conclusion that, the pictures contained in Figure 14 show that the network is able to predict test images sensibly when given a clear and understandable feature.

Although the models are successful, they do have some disadvantages and limitations as well. The first picture on Figure 15 can be defined as a picture with a lot of "noise", its an art piece and has a lot of actors and environment pieces in it. Although the network is able to predict that a group of people exist, it cannot predict the environment that the people are in. This is the first negative element of the model, which is that it cannot predict correctly when faced with a noisy image. The second image in Figure 15 displays a cow that is drinking water from a log, and the network guesses that it is a giraffe. This perhaps can be interpreted as a limitation of the model, there might not be enough cow pictures in the training set for the network to identify the animal as the cow correctly. Another limitation that arises very frequently over predictions can be observed in image 3 of Figure 15. Since the vocabulary provided to us is very small (1000 words + 4 special tokens) the network most of the time chooses the token "x_UNK_" (for unknown words). This perhaps isn't a disadvantage of the model itself though, since many training image caption contain this token, which leads the network to learn this word as a common occurrence.

Figure 13. The image which is predicted. The results of the prediction as well as the actual caption of the image can be found on Table III.

Table III
MODELS, THEIR PREDICTIONS AND ACTUAL CAPTIONS FOR THE IMAGE DISPLAYED IN FIGURE 13.

Model	Prediction
Actual Captions	a person cutting a slice of pizza with a fork and knife a person x_UNK_ a piece of cheese pizza with a fork eating pizza on a paper plate with knife and fork a slice of pizza cut by a knife and fork a person cutting up a slice of pizza with a knife and fork
Core Merge Model	a man is eating a slice of pizza
Core Par-Inject Model	a man is eating a slice of pizza
Core Pre-Inject Model	a person holding a slice of pizza on a table
Core Init-Inject Model	a person is eating a piece of pizza
Modified Merge Model	a woman is eating a slice of pizza
Modified Par-Inject Model	a woman is eating a slice of pizza
Modified Pre-Inject Model	a person holding a slice of pizza on a plate
Modified Init-Inject Model	a person is eating a slice of pizza
Bidirectional Multimodal	a man sitting at a table with a pizza

Table III shows predictions from all 9 designed models for

[!h]

Figure 14. Examples of successful predictions from the network. An actual caption is also displayed.

Figure 15. Examples of unsuccessful predictions from the network. An actual caption is also displayed.

the image displayed in Figure 13. As it can be observed, all models are able to predict that a pizza is displayed and a person (be it predicted as a woman or a man) is eating it. Some models, such as the Pre-Inject Model, also predict additional information, outputting the location of the pizza. This is perhaps the reason why Pre-Inject comes first in the evaluation metrics, it gives additional information of the image. Since the captions are mostly similar, our models are not significantly different from each other in terms of captioning. This is similar to the findings of [5], where there is no clear advantage or disadvantage to any particular model.

This ends the discussion on the predicted captions, it can be concluded that for most images the network is able to predict at least one of the three components of an image (the act, actor, environment) and successfully predicts all three in a notable amount of cases. However, some limitations existing in the vocabulary size and training data, and the fact that crowded and noisy pictures cannot be successfully detected should be noted.

IV. Discussion

When we compare model performances based on training times, training and validation loss plots, and evaluation metrics, it is hard to define one model as the best model. Based on loss function plots and evaluation metrics, Pre-Inject Model is better than the other models. Based on training time, Merge Model is observed to be better than the other models. Modifications resulted in more training time in Par-Inject Model and Init-Inject Model. Their advantage was that they learnt faster than Core Models, therefore it can be stated that Modified Models learnt faster than Core Models. However, they did not improve evaluation metric scores and 6 out of the 8 worst metric scores among different models are coming from Modified Models, which are shown with red coloring in Table II. In addition, Bidirectional Models showed slower learning performance since they have more than 5 times more parameters than other models. But that model did not provide better results and showed same performance can be achieved with similar networks. However, Bidirectional Models can be developed in other ways to benefit image captioning task.

We directly took image vectors as coming from Inception v3. We could have used fine tuning for feature extraction part with a lower learning rate so that we could customize Inception v3 for our task by using transfer learning. However, we already had fine tuning for Embedding layer since Embedding layer is trainable in our all models after initialization with embedding weight matrix taken from GloVe dataset. Hyperparameter selection process could be done via grid search method. Grid search method would be very time consuming when it is thought that each model had training time around 100 minutes. Instead of focusing on one model and optimizing it with grid search hyperparameter selection, instead we preferred working on different inject and merge model architectures and

compare their performances. For hyperparameter selection, parameters for modified models were selected according to similar applications [5]. As an another improvement method, we could use attention mechanism [33].

We ran the following analyses on existing data: loss and training performances, evaluation metrics on caption similarities and qualitative analyses on sample image captions. We also tried examples from outside training and test set to observe how the network behaves in a totally unfamiliar case. These analyses showed us how this task can be performed with different architectures using various models on TensorFlow Keras. In general, we reached our goal of trying different models and understanding how different models and modifications affect the network. We did not expect evaluation scores to match state-of-art sophisticated models with more computational power and complex architectures. Therefore, we expected and obtained fairly reasonable results. Our models surprised us with how well they generate captions for given images, even if the networks were relatively simple and implementable with our means.

REFERENCES

[1] H. Sharma, M. Agrahari, S. K. Singh, M. Firoj, and R. K. Mishra, "Image captioning: A comprehensive survey," in *2020 International Conference on Power Electronics IoT Applications in Renewable Energy and its Control (PARC)*, 2020, pp. 325–328.

[2] R. Staniūtė and D. Sesok, "A systematic literature review on image captioning," *Applied Sciences*, vol. 9, p. 2024, 2019.

[3] "What are word embeddings for text?" https://machinelearningmastery.com/what-are-word-embeddings/, (Accessed on 01/10/2021).

[4] "Short technical information about word2vec, glove and fasttext | by côme cothenet | towards data science," https://towardsdatascience.com/short-technical-information-about-word2vec-glove-and-fasttext-d38e4f529ca8, (Accessed on 01/10/2021).

[5] M. Tanti, A. Gatt, and K. P. Camilleri, "Where to put the image in an image caption generator," *Natural Language Engineering*, vol. 24, no. 3, p. 467–489, Apr 2018. [Online]. Available: http://dx.doi.org/10.1017/S1351324918000098

[6] M. Z. Hossain, F. Sohel, M. F. Shiratuddin, and H. Laga, "A comprehensive survey of deep learning for image captioning," 2018.

[7] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://www.aclweb.org/anthology/P02-1040

[8] R. Vedantam, C. L. Zitnick, and D. Parikh, "Cider: Consensus-based image description evaluation," *CoRR*, vol. abs/1411.5726, 2014. [Online]. Available: http://arxiv.org/abs/1411.5726

[9] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Ann Arbor, Michigan: Association for Computational Linguistics, Jun. 2005, pp. 65–72. [Online]. Available: https://www.aclweb.org/anthology/W05-0909

[10] C.-Y. Lin and F. J. Och, "Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics," in *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, Barcelona, Spain, Jul. 2004, pp. 605–612. [Online]. Available: https://www.aclweb.org/anthology/P04-1077

[11] "Http response status codes - http | mdn," https://developer.mozilla.org/en-US/docs/Web/HTTP/Status, (Accessed on 01/11/2021).

[12] "Understanding and decoding a jpeg image using python - yasoob khalid," https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python/, (Accessed on 01/11/2021).

[13] "Error when train on customized dataset: Invalid jpeg data or crop window, data size 36864 · issue #455 · tensorflow/tpu · github," https://github.com/tensorflow/tpu/issues/455, (Accessed on 01/11/2021).

[14] "tf.data.dataset | tensorflow core v2.4.0," https://www.tensorflow.org/api_docs/python/tf/data/Dataset, (Accessed on 01/11/2021).

[15] "Advanced guide to inception v3 on cloud tpu | google cloud," https://cloud.google.com/tpu/docs/inception-v3-advanced, (Accessed on 01/11/2021).

[16] "keras-inception-resnetv2/evaluate_image.py at master · kentsommer/keras-inception-resnetv2 · github," https://github.com/kentsommer/keras-inception-resnetV2/blob/master/evaluate_image.py#L9, (Accessed on 01/11/2021).

[17] H. Lamba, "Automatic image captioning," https://github.com/hlamba28/Automatic-Image-Captioning, 2018.

[18] M. Tanti, A. Gatt, and K. Camilleri, "What is the role of recurrent neural networks (rnns) in an image caption generator?" 08 2017.

[19] L. Zhou, C. Xu, P. Koch, and J. J. Corso, "Image caption generation with text-conditional semantic attention," *arXiv preprint arXiv:1606.04621*, 2016.

[20] "Cs 224d: Deep learning for nlp," https://cs224d.stanford.edu/lecture_notes/LectureNotes4.pdf, (Accessed on 01/10/2021).

[21] C. Wang, H. Yang, C. Bartz, and C. Meinel, "Image captioning with deep bidirectional lstms," *CoRR*, vol. abs/1604.00790, 2016. [Online]. Available: http://arxiv.org/abs/1604.00790

[22] Faizan-E-Mustafa, "Image-captioning," https://github.com/Faizan-E-Mustafa/Image-Captioning/blob/master/Notebook.ipynb,, 2018.

[23] R. S. J. Pennington and C. D. Mannin, "Glove: Global vectors for word representation," https://nlp.stanford.edu/projects/glove/, (Accessed on 01/11/2021).

[24] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[25] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016.

[26] "Image captioning with keras. table of contents: | by harshall lamba | towards data science," https://towardsdatascience.com/image-captioning-with-keras-teaching-computers-to-describe-pictures-c88a46a311b8, (Accessed on 01/11/2021).

[27] salaniz, "Microsoft coco caption evaluation," https://https://github.com/salaniz/pycocoevalcap, 2017.

[28] "A gentle introduction to calculating the bleu score for text in python," https://machinelearningmastery.com/calculate-bleu-score-for-text-python/#:~:text=Cumulative%20scores%20refer%20to%20the,%2C%20also%20called%20BLEU%2D4, (Accessed on 01/10/2021).

[29] "Evaluating models | automl translation documentation | google cloud," https://cloud.google.com/translate/automl/docs/evaluate, (Accessed on 01/10/2021).

[30] Y. Wang, J. Xu, Y. Sun, and B. He, "Image captioning based on deep learning methods: A survey," *CoRR*, vol. abs/1905.08110, 2019. [Online]. Available: http://arxiv.org/abs/1905.08110

[31] X. Li, X. Yin, C. Li, P. Zhang, X. Hu, L. Zhang, L. Wang, H. Hu, L. Dong, F. Wei, Y. Choi, and J. Gao, "Oscar: Object-semantics aligned pre-training for vision-language tasks," 2020.

[32] "Coco captions benchmark (image captioning) | papers with code," https://paperswithcode.com/sota/image-captioning-on-coco-captions, (Accessed on 01/09/2021).

[33] K. Xu, J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," *CoRR*, vol. abs/1502.03044, 2015. [Online]. Available: http://arxiv.org/abs/1502.03044

APPENDIX A
PLOTS OF NETWORK MODELS

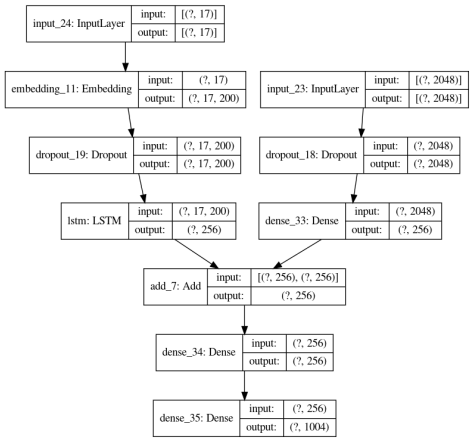

Figure A.1. Diagram of Merge Model.

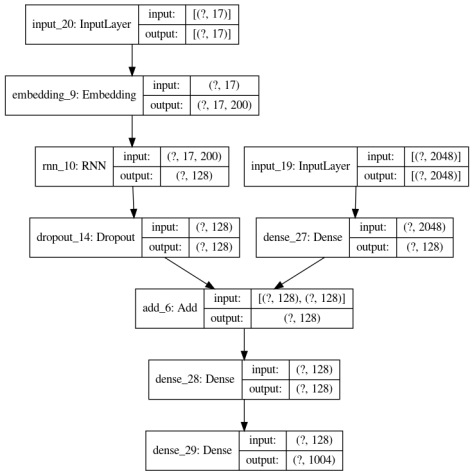

Figure A.2. Diagram of Modified Merge Model.

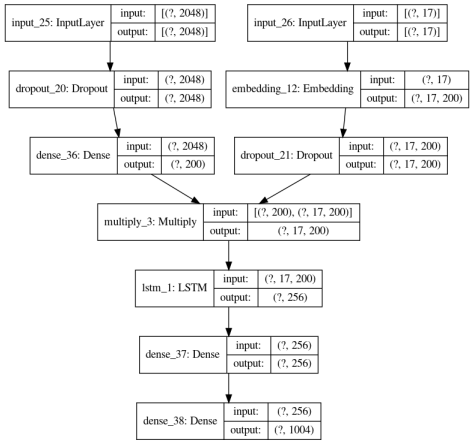

Figure A.3. Diagram of Par Inject Model.

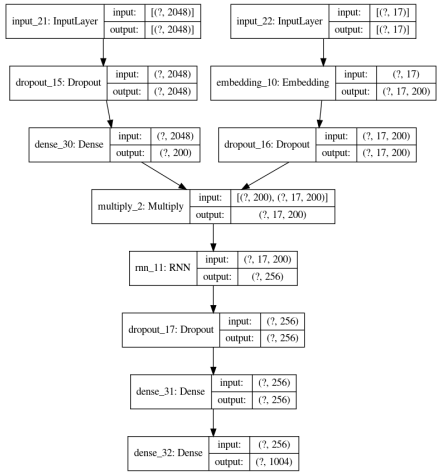

Figure A.4. Diagram of Modified Par Inject Model.

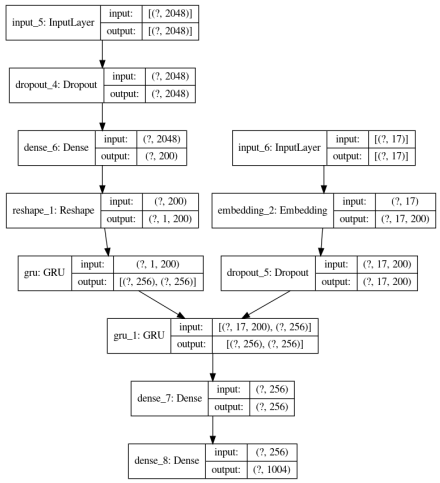

Figure A.5. Diagram of Pre Inject Model.

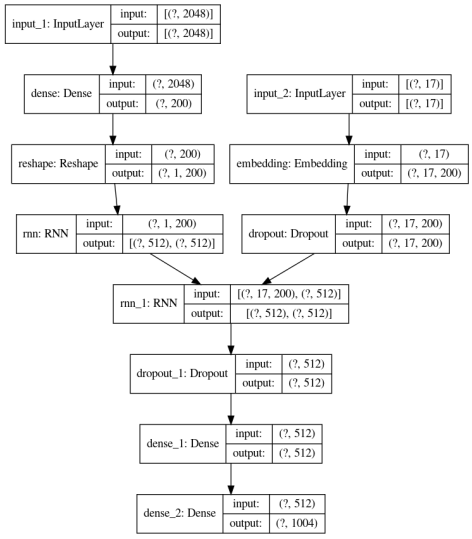

Figure A.6. Diagram of Modified Pre Inject Model

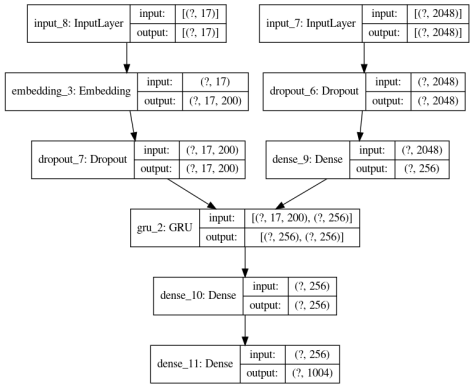

Figure A.7. Diagram of Init Inject Model

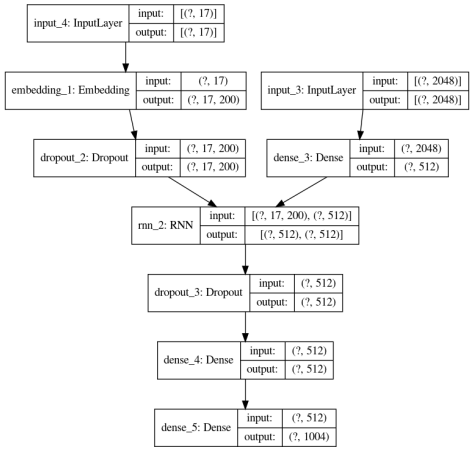

Figure A.8. Diagram of Modified Init Inject Model

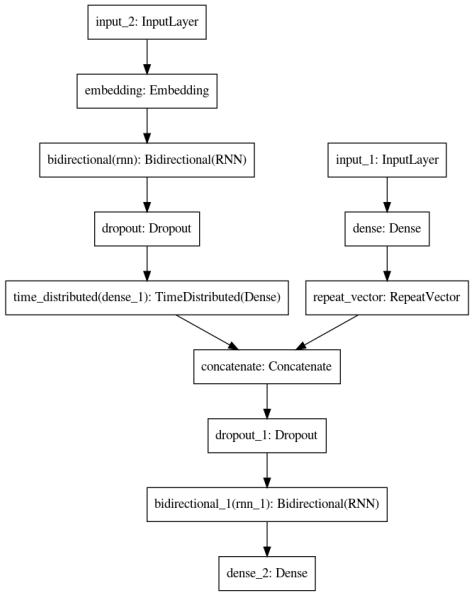

Figure A.9. Diagram of Bi-Directional Multi-Model

APPENDIX B
EXAMPLE CAPTION OUTPUTS FOR TEST IMAGES

Prediction: a baseball player holding a bat on a field
Actual Caption: a little boy that is standing at home plate

Prediction: a fire hydrant on a sidewalk near a street
Actual Caption: a fire hydrant is painted red white and green

Figure B.1. Two examples of caption predictions from the network. The network successfully predicts the images.

Prediction: a plate of food that x_UNK_ meat and vegetables
Actual Caption: a blue and white plate filled with meat and vegetables

Prediction: a group of people riding motorcycles down a street
Actual Caption: three motorcycles parked near a x_UNK_ vehicle

Prediction: a bathroom with a sink and a mirror
Actual Caption: a x_UNK_ bathroom with a sink and mirror

Figure B.2. Three examples of caption predictions from the network. The network successfully predicts the images.

Prediction: a clock on a wall with a x_UNK_ x_UNK_ on it
Actual Caption: a clock in front of three big windows

Prediction: a person on skis on a snowy mountain
Actual Caption: a group of people in the snow with skis

Prediction: a man riding a skateboard down a street
Actual Caption: a man riding a skateboard in a skate park

Figure B.3. Three examples of caption predictions from the network. The network successfully predicts the images.

APPENDIX C
EXAMPLE CAPTION OUTPUTS FOR IMAGES OUTSIDE OUR DATASETS

Prediction: a group of people riding horses on a snow covered slope

Prediction: a man and a dog are standing in a field

Prediction: a man and woman standing next to a dog

Figure C.1. Examples of caption predictions from the network. The network successfully predicts the images.

APPENDIX D
WRITTEN CODE FOR THE PROJECT

```
#!/usr/bin/env python
# coding: utf-8

# ### Pre-Import

# In[ ]:


get_ipython().system('pip install "git+https://github.com/salaniz/pycocoevalcap.git"')


# ### Imports

# In[ ]:


from tqdm.notebook import tqdm
import time
import os
import json
import requests
from struct import unpack

import h5py
import pickle
import pandas as pd
import numpy as np
import cv2
import matplotlib.pyplot as plt
import tensorflow as tf

from tensorflow_addons.rnn import LayerNormLSTMCell
from tensorflow_addons.rnn import LayerNormSimpleRNNCell

import keras
from keras.layers.merge import add
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.models import Model, Sequential
from keras.layers import Lambda, Input, LSTM, GRU, RNN, Embedding, Multiply,  Concatenate, TimeDistributed,
     Dense, Bidirectional, RepeatVector, Activation, Flatten, Reshape, concatenate, Dropout,
    BatchNormalization

TRAIN_DATA_FILENAME = "../input/project-data/eee443_project_dataset_train.h5"
TRAIN_IMAGES_DIRECTORY = "../input/images-for-train/train_images/"
TRAIN_FEATURES_DIRECTORY = "../input/tupled-data/train_tupled_data"

TEST_DATA_FILENAME = "../input/project-data/eee443_project_dataset_test.h5"
TEST_IMAGES_DIRECTORY = "../input/images-for-test/test_images/"
TEST_FEATURES_DIRECTORY = "../input/tupled-data/test_tupled_data"


# ### Downloading and Cleaning Data

# In[ ]:


def save_ims(data, pathname):
    """
    A function which saves images from given urls
    :param data: the array which holds url values
    :param pathname: the save path
    """
    s = time.time()
    i = 0

    if not os.path.exists(pathname):
        os.makedirs(pathname)

    for url in data:

```

```
        url = url.decode()
        name = url.split("/")[-1].strip()
        path = os.path.join(pathname, name)

        if not os.path.exists(path):

            headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.169 Safari/537.36'}
            response = requests.get(url, stream=True, headers=headers)

            # check if data is obttained successfully
            if response.status_code == 200:
                    with open(path, 'wb') as outfile:
                        outfile.write(response.content)

        # prints affirmation at each 1000 iterations
        if i % 1000 == 1:

            p = time.time() - s
            it = p/i
            print("{:.2f} mins passed. {:.2f} seconds per iter. Iteration {}".format(p/60, it, i))

        i += 1


class JPEG:
    """
    The JPEG class which helps in cleaning the data, more info can be found on the report.
    """
    def __init__(self, image_file):
        with open(image_file, 'rb') as f:
            self.img_data = f.read()

    def decode(self):
        data = self.img_data
        while (True):
            marker, = unpack(">H", data[0:2])
            # print(marker_mapping.get(marker))
            if marker == 0xffd8:
                data = data[2:]
            elif marker == 0xffd9:
                return
            elif marker == 0xffda:
                data = data[-2:]
            else:
                lenchunk, = unpack(">H", data[2:4])
                data = data[2 + lenchunk:]
            if len(data) == 0:
                break



def clear_bad_JPEG(root_img):
    """
    A funciton which clears a directory from bad JPEG's (.jpg files that are not actually images)
    :param root_img: tthe directory of jpg images
    """

    images = os.listdir(root_img)

    bads = []

    for img in tqdm(images):
        image = root_img + img
        image = JPEG(image)
        try:
            image.decode()
        except:
            bads.append(img)
    print(bads)
    for name in bads:
        os.remove(root_img + name)


```

```
# # Data Processing

# In[ ]:


def caption_array_to_str(caption_array):
    """
    A function which formats a caption numpy array to a list of string(s). Used for evaluation and
    prediction.
    :param caption_array: The numpy array which stores captions/predicted captions
    :return: a list of strings that contain the caption
    """

    list_of_captions = []

    caption = ""

    if(caption_array.ndim == 1):
        caption_array = np.expand_dims(caption_array, axis=0)

    for caps in caption_array:

        for word in caps:

            if (word == 'x_NULL_') or (word == 'x_START_') or (word == 'x_END_'):
                continue

            caption += word + " "


        list_of_captions.append(caption.strip())
        caption = ""


    return list_of_captions


def get_caption(name_list, imid, cap, name):
    """
    A function which returns the caption of an image given its name
    :param name_list:
    :param imid: image id vector
    :param cap: cap array (holds all captions)
    :param name: the name of the image
    :return: the caption numpy array
    """

    ind = name_list.index(name) + imid.min()

    return cap[np.where(imid == ind)]


def create_pre_processed_set(image_directory, shuffle=False):
    """
    A function which creates a set of preprocessed images ready for feature extraction.
    :param image_directory: The directory containing all images
    :param shuffle: whetther we want to shuffle data or not
    :return: a tf.data.Dataset that contains all preprocessed images (contains meaning it contains the
    formula to create them,
    however the data is not actual loaded into memory until called)
    """

    def process_files(path):
        img = tf.io.read_file(path)
        img = tf.image.decode_jpeg(img, channels=3)
        img = tf.image.resize(img, (299, 299))
        img = tf.keras.applications.inception_v3.preprocess_input(img)
        return img

    def process_name(path):
        name = path.numpy().decode().split("/")[-1]
        return name

    def process(path):
```

```
        name = tf.py_function(process_name, [path], tf.string)
        img = tf.py_function(process_files, [path], tf.float32)
        return (img, name)

    file_data = tf.data.Dataset.list_files(str(image_directory) + "*.jpg", shuffle=shuffle)

    return file_data.map(lambda x: process(x))


def create_features(filename, images, name_list, imid, cap, process_size = 250):
    """

    :param filename: the filename for the data to be dumped in
    :param images: the image tf.data.Dataset (created in the previous function)
    :param name_list: a list of image url names
    :param imid: image id array
    :param cap: captions array
    :param process_size: the batch size with which the feattures are extracted
    :return: the length of the data
    """

    inception = tf.keras.applications.InceptionV3(weights='imagenet')
    inception = tf.keras.Model(inception.input, inception.layers[-2].output)


    length = 0

    with open(filename, "wb") as outfile:

        for data in tqdm(images.batch(process_size)):

            image = data[0]
            name = data[1].numpy()
            feature = inception(image).numpy()

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

                f = feature[i].squeeze()
                n = name[i].decode()
                c = get_caption(name_list, imid, cap, n)

                tp = (f, c, n)
                pickle.dump(tp, outfile)

                length += 1

    outfile.close()
    return length


def loadpickle(filename):
    """
    A generator function which yields data in a file until there is none
    :param filename:
    :return: loaded tuple (or any other data that is stored)
    """

    with open(filename, "rb") as f:

        while True:

            try:
                yield pickle.load(f)

            except EOFError:
                break



def create_data(feature_directory, url=None, imid=None, cap=None, image_directory=None):
    """

    :param feature_directory: directory of the pickled data
    :param url: url array (from the given dataset)
```

```
    :param imid: image id vector (from the given dataset)
    :param cap: cap array (from the given dataset)
    :param image_directory: the directory images are stored in
    :return: the dataset and the length of said dataset
    """

    length = -1

    if not os.path.isfile(feature_directory):

        if not image_directory:
            raise Exception("No image directory given. Enter image directory for feature extraction.")

        name_list = [u.split("/")[-1].strip() for u in np.char.decode(url).tolist()]
        images = create_pre_processed_set(image_directory)
        length = create_features(feature_directory, images, name_list, imid, cap)

    dataset = tf.data.Dataset.from_generator(loadpickle, args=[feature_directory], output_types=(np.float32
    ,np.int32, tf.string))

    if length == -1:
        length = dataset.reduce(0, lambda x, _: x + 1).numpy()

    return dataset, length


#  ### Get Train Image Dataset

# In[ ]:


f = h5py.File(TRAIN_DATA_FILENAME, "r")

for key in list(f.keys()):
    print(key, ":", f[key][()].shape)

train_cap = f["train_cap"][()]
train_imid = f["train_imid"][()]
train_url = f["train_url"][()]
word_code = f["word_code"][()]


df = pd.DataFrame(word_code)
df = df.sort_values(0, axis=1)
words = np.asarray(df.columns)

wordtoix = {}
for i in range(len(words)):
  word = words[i]
  wordtoix[word] = i


train_data, train_data_length = create_data( TRAIN_FEATURES_DIRECTORY, train_url, train_imid, train_cap,
    TRAIN_IMAGES_DIRECTORY)

print("Vocab Size =", len(words))
print( "{} of {} retrieved. {:.1f}% of data is clean.".format(train_data_length, len(train_url), 100 *
    train_data_length/len(train_url) ) )

# delete after use so that memory is not loaded
del train_cap
del train_imid
del train_url
del word_code


# In[ ]:


for d in train_data.shuffle(1000).take(1):
    features = d[0]
    image_name= d[2].numpy().decode()
    captions = d[1].numpy()

```

```
    im = cv2.imread(TRAIN_IMAGES_DIRECTORY + image_name)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
    plt.imshow(im)
    plt.show()
    cap = caption_array_to_str(words[captions])

    for c in cap:
        print(c)

    print(features.shape, captions.shape)


#  ### Get Test Image Dataset

# In[ ]:


f = h5py.File(TEST_DATA_FILENAME, "r")

for key in list(f.keys()):
    print(key, ":", f[key][()].shape)

test_cap = f["test_caps"][()]
test_imid = f["test_imid"][()]
test_url = f["test_url"][()]


test_data, test_data_length = create_data(TEST_FEATURES_DIRECTORY, test_url, test_imid, test_cap,
    TEST_IMAGES_DIRECTORY)

print( "{} of {} retrieved. {:.1f}% of data is clean.".format(test_data_length, len(test_url), 100 *
    test_data_length/len(test_url) ))

# delete after use so that memory is not loaded
del test_cap
del test_imid
del test_url


# In[ ]:


for d in test_data.shuffle(1000).take(1):
    features = d[0]
    captions = d[1].numpy()
    image_name= d[2].numpy().decode()

    im = cv2.imread(TEST_IMAGES_DIRECTORY + image_name)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
    plt.imshow(im)
    plt.show()
    cap = caption_array_to_str(words[captions])

    for c in cap:
        print(c)


# # Model Preparation

# In[ ]:


def data_generator(dataset, max_length, num_photos_per_batch, vocab_size):
    """
    The data generatoor function which generates training data
    :param dataset: the tf.data.Dataset object containing tthe tuple (feature, captions, name)
    :param max_length: maximum sentence/caption length in terms of words
    :param num_photos_per_batch: batch size
    :param vocab_size: vocabulary size, word count
    :return: the training data to be used at every iteration
    """

    X1, X2, y = [], [], []
```

```
    i = 0

    while True:

        for data in dataset:

            i += 1
            feature = data[0].numpy()
            caps = data[1].numpy()

            for j in range(caps.shape[0]):

                seq = caps[j]

                for k in range(1, seq.shape[0]):

                    in_seq = pad_sequences([seq[:k]], maxlen=max_length)[0]
                    out_seq = to_categorical([seq[k]], num_classes=vocab_size)[0]

                    X1.append(feature)
                    X2.append(in_seq)
                    y.append(out_seq)


            if i == num_photos_per_batch:

                yield [np.array(X1), np.array(X2)], np.array(y)
                X1, X2, y = [], [], []
                i = 0

    yield [np.array(X1), np.array(X2)], np.array(y)


def create_embedding(wordtoix):
    """
    Mehod which creates the embedding matrix wih a given word index dictionary
    :param wordtoix: word intex
    :return: the embedding matrtix
    """
    # Load Glove vectors
    glove_dir = '../input/glove6b200d/glove.6B.200d.txt'
    embeddings_index = {} # empty dictionary
    f = open(glove_dir, encoding="utf-8")

    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        # if (word == 'startseq' or word == 'unk' ):
        #   print(word)

        embeddings_index[word] = coefs
    f.close()
    print('Found %s word vectors.' % len(embeddings_index))


    embedding_dim = 200
    vocab_size = 1004

    # Get 200-dim dense vector for each of the 10000 words in out vocabulary
    embedding_matrix = np.zeros((vocab_size, embedding_dim))

    for word, i in wordtoix.items():

        if (word == 'x_UNK_'):
          word = 'unk'

        embedding_vector = embeddings_index.get(word)
        if  embedding_vector is None:
          print(word)

        if embedding_vector is not None:
            # Words not found in the embedding index will be all zeros
            embedding_matrix[i] = embedding_vector
```

```
    return embedding_matrix


# ### 1st Iteration Models

# In[ ]:


def create_merge_model(embedding_matrix):

    inputs1 = Input(shape=(2048,))
    img1 = Dropout(0.5)(inputs1)
    img2 = Dense(256, activation='relu')(img1)

    inputs2 = Input(shape=(max_length,))
    seq1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
    seq2 = Dropout(0.5)(seq1)
    seq3 = LSTM(256)(seq2)

    #add, not concatenate! wrong
    dec1 = add([img1, seq3])
    dec2 = Dense(256, activation='relu')(dec1)
    outputs = Dense(vocab_size, activation='softmax')(dec2)

    model = Model(inputs=[inputs1, inputs2], outputs=outputs)

    #set embedding layer's weight matrix
    model.layers[2].set_weights([embedding_matrix])
    model.layers[2].trainable = True

    return model


def create_init_inject_model(embedding_matrix):

    inputs1 = Input(shape=(2048,))
    img1 = Dropout(0.5)(inputs1)
    img2 = Dense(256, activation='relu')(img1)

    inputs2 = Input(shape=(max_length,))
    seq1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
    seq2 = Dropout(0.5)(seq1)

    #image is set as state
    seq3,state = GRU(256,return_state = True)(seq2,initial_state = img1)

    dec2 = Dense(256, activation='relu')(seq3)
    outputs = Dense(vocab_size, activation='softmax')(dec2)

    model = Model(inputs=[inputs1, inputs2], outputs=outputs)

    #set embedding layer's weight matrix
    model.layers[2].set_weights([embedding_matrix])
    model.layers[2].trainable = True

    return model



def create_pre_inject_model(embedding_matrix):

    inputs1 = Input(shape=(2048,))
    img1 = Dropout(0.5)(inputs1)
    img2 = Dense(embedding_dim, activation='relu')(img1)
    img2_reshaped = Reshape((1, embedding_dim), input_shape=(embedding_dim,))(img2)

    inputs2 = Input(shape=(max_length,))
    seq1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
    seq2 = Dropout(0.5)(seq1)
    seq3,state3 = GRU(256,return_state = True)(img2_reshaped)
    seq4,state4 = GRU(256,return_state = True)(seq2, initial_state = state3)
    dec = Dense(256, activation='relu')(seq4)
    outputs = Dense(vocab_size, activation='softmax')(dec)
```

```
    model = Model(inputs=[inputs1, inputs2], outputs=outputs)

    model.layers[4].set_weights([embedding_matrix])
    model.layers[4].trainable = True


    return model



def create_par_inject_model(embedding_matrix):

    max_length = 17
    vocab_size = 1004
    embedding_dim = 200

    inputs1 = Input(shape=(2048,))
    img1 = Dropout(0.5)(inputs1)
    img2 = Dense(200, activation='relu')(img1)

    inputs2 = Input(shape=(max_length,))
    seq1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
    seq2 = Dropout(0.5)(seq1)

    mul = Multiply()([img2,seq2])

    seq3 = LSTM(256)(mul)
    dec = Dense(256, activation='relu')(seq3)
    outputs = Dense(vocab_size, activation='softmax')(dec)

    model = Model(inputs=[inputs1, inputs2], outputs=outputs)

    model.layers[3].set_weights([embedding_matrix])
    model.layers[3].trainable = True

    return model


# ### Models with Layer Normalization

# In[ ]:


def create_merge_model_best(embedding_matrix):

    inputs1 = Input(shape=(2048,))
    img = Dense(128, activation='relu', kernel_initializer='random_normal')(inputs1)

    inputs2 = Input(shape=(max_length,))
    seq1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)

    seq2 = RNN(LayerNormLSTMCell(128))(seq1)
    seq3 = Dropout(0.5)(seq2)

    dec1 = add([img, seq3])
    dec2 = Dense(128, activation='relu',kernel_initializer='random_normal')(dec1)
    outputs = Dense(vocab_size, activation='softmax',kernel_initializer='random_normal')(dec2)

    model = Model(inputs=[inputs1, inputs2], outputs=outputs)

    #set embedding layer's weight matrix
    model.layers[1].set_weights([embedding_matrix])
    model.layers[1].trainable = True

    return model



def create_par_inject_model_best(embedding_matrix):

    inputs1 = Input(shape=(2048,))
    img1 = Dropout(0.5)(inputs1)
```

```
    img2 = Dense(200, activation='relu',kernel_initializer='random_normal',kernel_regularizer=tf.keras.
    regularizers.l2(1e-8))(img1)

    inputs2 = Input(shape=(max_length,))
    seq1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
    seq2 = Dropout(0.5)(seq1)

    mul = Multiply()([img1,seq2])

    seq3 = RNN(LayerNormLSTMCell(256))(mul)
    seq4 = Dropout(0.5)(seq3)
    dec = Dense(256, activation='relu',kernel_initializer='random_normal', kernel_regularizer=tf.keras.
    regularizers.l2(1e-8))(seq4)
    outputs = Dense(vocab_size, activation='softmax',kernel_initializer='random_normal', kernel_regularizer
    =tf.keras.regularizers.l2(1e-8))(dec)

    model = Model(inputs=[inputs1, inputs2], outputs=outputs)

    model.layers[3].set_weights([embedding_matrix])
    model.layers[3].trainable = True

    return model




def create_pre_inject_model_best(embedding_matrix):

    inputs1 = Input(shape=(2048,))
    img = Dense(embedding_dim, activation='relu',kernel_initializer='random_normal')(inputs1)
    img_reshaped = Reshape((1, embedding_dim), input_shape=(embedding_dim,))(img)

    inputs2 = Input(shape=(max_length,))
    seq1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
    seq2 = Dropout(0.5)(seq1)


    seq3,state3 = RNN(LayerNormSimpleRNNCell(512),return_state = True)(img_reshaped)
    seq4,state4 = RNN(LayerNormSimpleRNNCell(512),return_state = True)(seq2, initial_state = state3)

    seq5 = Dropout(0.5)(seq4)
    dec = Dense(512, activation='relu',kernel_initializer='random_normal')(seq5)
    outputs = Dense(vocab_size, activation='softmax',kernel_initializer='random_normal')(dec)

    model = Model(inputs=[inputs1, inputs2], outputs=outputs)

    model.layers[3].set_weights([embedding_matrix])
    model.layers[3].trainable = True


    return model



def create_init_inject_model_best(embedding_matrix):

    inputs1 = Input(shape=(2048,))

    img = Dense(512, activation='relu')(inputs1)

    inputs2 = Input(shape=(max_length,))
    seq1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
    seq2 = Dropout(0.5)(seq1)

    #image is set as state
    seq3,state = RNN(LayerNormSimpleRNNCell(512),return_state = True)(seq2,initial_state = img)
    seq4 = Dropout(0.5)(seq3)

    dec = Dense(512, activation='relu')(seq4)
    outputs = Dense(vocab_size, activation='softmax')(dec)

    model = Model(inputs=[inputs1, inputs2], outputs=outputs)


```

```
    #set embedding layer's weight matrix
    model.layers[1].set_weights([embedding_matrix])
    model.layers[1].trainable = True

    return model


# ### Bi-Directional Model

# In[ ]:


def create_bilstm_model_layernorm(embedding_matrix):

    inputs1 = Input(shape=(2048,))
    img1 = Dense(embedding_dim, input_shape=(2048,), activation='relu')(inputs1)
    img2 = RepeatVector(max_length)(img1)

    inputs2 = Input(shape=(max_length,))
    seq1 = Embedding(vocab_size,embedding_dim, input_length=max_length, mask_zero=True)(inputs2)
    seq2 = Bidirectional(RNN(LayerNormLSTMCell(256), return_sequences = True))(seq1)
    seq3 = Dropout(0.5)(seq2)
    seq4 = TimeDistributed(Dense(embedding_dim))(seq3)

    comb1 = Concatenate(axis=2)([img2, seq4])
    comb2 = Dropout(0.5)(comb1)
    comb3 = Bidirectional(RNN(LayerNormLSTMCell(1000), return_sequences = False))(comb2)
    comb4 = Dense(vocab_size, activation = 'softmax')(comb3)

    model = Model(inputs=[inputs1, inputs2], outputs=comb4)
    model.summary()

    model.layers[1].set_weights([embedding_matrix])
    model.layers[1].trainable = True

    return model


# # Train (skip to load pre-existing models)

# ### Creating the Embedding Matrix

# In[ ]:


print("Creating embedding matrix...")
embedding_matrix = create_embedding(wordtoix)
print('Embedding matrix is ready!')


# ### Start Training

# In[ ]:


data_length = 10000 #train_datta_length
train_data = train_data.take(data_length)

val_length = round(data_length * 0.15)
train_length = data_length - val_length

val_dataset = train_data.take(val_length)
train_dataset = train_data.skip(val_length)


max_length = 17
vocab_size = 1004
embedding_dim = 200

epochs = 12


# In[ ]:

```

```
799
800 train_ = False
801
802 if train_ == True:
803     model_names = ["create_merge_model_best",
804                    "create_par_inject_model_best",
805                    "create_pre_inject_model_best",
806                    "create_init_inject_model_best"]
807
808     history_list = {}
809
810     for i in range(2, 3):
811
812         if i == 0:
813             model = create_merge_model_best(embedding_matrix)
814             batch_size = 128
815         if i == 1:
816             model = create_par_inject_model_best(embedding_matrix)
817             batch_size = 64
818         if i == 2:
819             model = create_pre_inject_model_best(embedding_matrix)
820             batch_size = 32
821         if i == 3:
822             model = create_init_inject_model_best(embedding_matrix)
823             batch_size = 128
824
825
826         model.compile(loss='categorical_crossentropy', optimizer='adam')
827         print("Model compiled...")
828
829         checkpoint = tf.keras.callbacks.ModelCheckpoint(model_names[i], monitor='val_loss', verbose=1,
    save_best_only=True, mode='min')
830         callbacks_list = [checkpoint]
831
832         print("Creating Generators")
833         train_generator = data_generator(train_dataset, max_length, batch_size, vocab_size)
834         val_generator = data_generator(val_dataset, max_length, batch_size, vocab_size)
835
836         train_step = train_length // batch_size
837         val_step = val_length // batch_size
838
839
840         print("Starting training...")
841         start = time.time()
842
843         history = model.fit(train_generator,
844                             steps_per_epoch = train_step,
845                             validation_data = val_generator,
846                             validation_steps = val_step,
847                             epochs = epochs,
848                             callbacks = callbacks_list,
849                                 workers = 0)
850
851         history_list[model_names[i]] = history.history
852
853
854         model.save(model_names[i] + "_save")
855
856
857         print("Time spent {:.2f} mins.".format( (time.time()-start)/60 ))
858
859
860     with open('loss_history.json', 'w') as fp:
861         json.dump(history_list, fp)
862
863
864 # ### Load Pre-Saved Models and History Dictionaries
865
866 # In[ ]:
867
868
869 models = {}
870
871 path = "../input/models/old_models/"
```

```
872
873 with open(path + 'old_loss_history.json', 'r') as fp:
874     history_dict = json.load(fp)
875
876 for key in history_dict:
877     try:
878         models[key] = tf.keras.models.load_model(path + key, custom_objects={'LayerNormLSTMCell':
    LayerNormLSTMCell})
879     except:
880         try:
881             models[key] = tf.keras.models.load_model(path + key, custom_objects={'LayerNormSimpleRNNCell':
    LayerNormSimpleRNNCell})
882         except:
883             print("Could not load model.")
884
885
886 # ### Plot History (Different Blocks for the Report)
887
888 # In[ ]:
889
890
891 def plot_loss(model_name, hist):
892
893     fig = plt.figure(figsize=(12, 8), dpi=160, facecolor='w', edgecolor='k')
894     fig.suptitle(model_name, fontsize=13)
895     plt.plot(hist["loss"], "C2", label="Train Sequential Cross Entropy Loss")
896     plt.plot(hist["val_loss"], "C3", label="Validation Sequential Cross Entropy Loss")
897     plt.title('model loss')
898     plt.ylabel('Loss')
899     plt.xlabel('Epoch')
900     plt.savefig("pre_inject.png", bbox_inches='tight')
901
902
903 # In[ ]:
904
905
906 fig = plt.figure(figsize=(18, 12), dpi=160, facecolor='w', edgecolor='k')
907 c = ["C2", "C3"]
908 names = ["Merge Model", "Par Inject Model", "Pre Inject Model", "Init Inject Model",
909         "Merge Model with Layer Normalization", "Par Inject Model with Layer Normalization",
910          "Pre Inject Model with Layer Normalization", "Init Inject Model with Layer Normalization"]
911
912
913 for i in range(4):
914     plt.subplot(2,2,i + 1)
915     h1 = list(history_dict.values())[i]
916     h2 = list(history_dict.values())[i + 4]
917
918     plt.plot(h1["val_loss"], c[0], label=names[i])
919     plt.plot(h2["val_loss"], c[1], label=names[i + 4])
920
921
922     plt.title('Validation Loss with Layer Normalization for ' + names[i] + "s")
923     plt.ylabel('Loss')
924     plt.xlabel('Epoch')
925     plt.legend()
926 plt.savefig("Loss_Plots_Comparison_Layer_Norm.png", bbox_inches='tight')
927
928
929 # In[ ]:
930
931
932 fig = plt.figure(figsize=(6, 4), dpi=160, facecolor='w', edgecolor='k')
933
934 c = ["C1", "C2", "C3", "C4"]
935 names = ["Merge Model", "Par Inject Model", "Pre Inject Model", "Init Inject Model"]
936 h = list(history_dict.values())[4:]
937 fig = plt.figure(figsize=(12, 8), dpi=160, facecolor='w', edgecolor='k')
938 plt.subplot(1,2,1)
939 for i in range(4):
940     plt.plot(h[i]["loss"], c[i], label=names[i])
941 plt.title('Training Loss with Layer Normalization')
942 plt.ylabel('Loss')
943 plt.xlabel('Epoch')
```

```
plt.legend()


plt.subplot(1,2,2)
for i in range(4):
    plt.plot(h[i]["val_loss"], c[i], label=names[i])
plt.title('Validation Loss with Layer Normalization')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()

plt.savefig("Loss_Plots_Layer_Norm.png", bbox_inches='tight')


# # Evaluation

# In[ ]:


def create_evaluation_dictionary(model, dataset, data_length, batch_size, words):
    """
    Creates a dictionary for the score listing function
    :param model: the model the predictions will be based on
    :param dataset: the tf.data.Dataset object, in our case this is the test dataset
    :param data_length: the data length of the dataset
    :param batch_size: this is the process size, how many predictions are to be done at once
    :param words: the words dictionary
    :return:
    """

    pred_container = {}
    actual_container = {}

    start = time.time()

    referenced = dataset.map(lambda x, y, z: (x, z))

    for batch in referenced.batch(batch_size):

        feats = batch[0].numpy()
        iteration = feats.shape[0]
        seq = np.tile(np.array([1] + [0]*16), (iteration, 1))

        for i in range(16):

            pred = model.predict([feats, seq])
            seq[:, i+1] = np.argmax(pred, axis=1)

        for i in range(iteration):

            name = batch[1].numpy()[i].decode()
            pred_container[name] = caption_array_to_str(words[seq[i]])

    for d in dataset.take(data_length):
        name = d[2].numpy().decode()
        caption = d[1].numpy()
        actual_container[name] = caption_array_to_str(words[caption])


    return actual_container, pred_container



from pycocoevalcap.bleu.bleu import Bleu
from pycocoevalcap.rouge.rouge import Rouge
from pycocoevalcap.cider.cider import Cider
from pycocoevalcap.meteor.meteor import Meteor

def score(ref, hypo):
    """
    ref, dictionary of reference sentences (id, sentence)
    hypo, dictionary of hypothesis sentences (id, sentence)
    score, dictionary of scores
    """
```

```
    scorers = [(Bleu(4), ["Bleu_1", "Bleu_2", "Bleu_3", "Bleu_4"]),
               (Meteor(),"METEOR"),
               (Rouge(), "ROUGE_L"),
               (Cider(), "CIDEr")]

    final_scores = {}

    for scorer, method in scorers:

        score, scores = scorer.compute_score(ref, hypo)

        if type(score) == list:

            for m, s in zip(method, score):

                final_scores[m] = s

        else:

            final_scores[method] = score


    return final_scores


def print_metrics(actual, preds, model_name):

    metric_dict = score(actual, preds)

    b1 = metric_dict["Bleu_1"] * 100
    b2 = metric_dict["Bleu_2"] * 100
    b3 = metric_dict["Bleu_3"] * 100
    b4 = metric_dict["Bleu_4"] * 100
    m = metric_dict["METEOR"] * 100
    r = metric_dict["ROUGE_L"] * 100
    c = metric_dict["CIDEr"] * 100
    string = "\n--------------------\nModel: {}\n--------------------\nBLEU-1: {:.1f}\nBLEU-2: {:.1f}\nBLEU-3: {:.1f}\nBLEU-4: {:.1f}\nMETEOR:  {:.1f}\nROUGE_L: {:.1f}\nCIDEr: {:.1f}\n".format(model_name, b1, b2, b3, b4, m, r, c)
    print(string)


# ### Evaluate Bleu, Meteor, CIDEr and Rouge_L Scores

# In[ ]:


print("Example of a score output (this might take a short while)...\n")
data_size = test_data_length
process_size = 250
prediction_data = test_data.take(data_size)

actual, preds = create_evaluation_dictionary(models["create_merge_model_best"], prediction_data, data_size, process_size, words)
print_metrics(actual, preds, key)


# ### Display Predicted Image with Caption (for the report)

# In[ ]:


def predict_for_extra(model, feature_model, img_path):
    """
    prediction function for extra images, not provided to us via test dataset
    this is for fun but also to see if the model can actually correctly guess any other image (spoiler: it can)
    :param model: the prediction model
    :param feature_model: feature extraction model
    :param img_path: path of image to be predicted
    """
    img = tf.io.read_file(img_path)
    img = tf.image.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, (299, 299))
```

```
    img = tf.keras.applications.inception_v3.preprocess_input(img)
    img = np.expand_dims(img, axis=0)
    img = feature_model(img)
    img = img.numpy().reshape(1, -1)

    seq = np.array([0]*17).reshape(1, -1)

    seq[:, 0] = 1

    for i in range(16):
        pred = model.predict([img,seq])
        seq[:, i+1] = np.argmax(pred)

    seq = seq.reshape(-1)

    im = cv2.imread(img_path)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)

    fig = plt.figure(dpi=160, facecolor='w', edgecolor='k')
    plt.imshow(im)
    plt.title("Prediction: " + caption_array_to_str(words[seq])[0], fontsize=9)
    plt.axis('off')

    plt.show()



def predict_caption(model, image_directory, feature_model, amount, dr):
    """
    A function which predicts and saves many caption/prediction duos plotted with their respective images
    :param model: prediction model
    :param image_directory: the image directory, test images directory
    :param feature_model: feature extraction model
    :param amount: how many pictures the function will be predicting
    :param dr: the save directory
    """

    for d in test_data.shuffle(test_data_length).take(amount):

        img_path = str(image_directory + d[2].numpy().decode())
        c = d[1].numpy()
        k = np.random.randint(c.shape[0])

        img = tf.io.read_file(img_path)
        img = tf.image.decode_jpeg(img, channels=3)
        img = tf.image.resize(img, (299, 299))
        img = tf.keras.applications.inception_v3.preprocess_input(img)
        img = np.expand_dims(img, axis=0)
        img = feature_model(img)
        img = img.numpy().reshape(1, -1)

        seq = np.array([0]*17).reshape(1, -1)

        seq[:, 0] = 1

        for i in range(16):
            pred = model.predict([img,seq])
            seq[:, i+1] = np.argmax(pred)

        seq = seq.reshape(-1)

        im = cv2.imread(img_path)
        im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)

        fig = plt.figure(dpi=160, facecolor='w', edgecolor='k')
        plt.imshow(im)
        plt.title("Prediction: " + caption_array_to_str(words[seq])[0] + "\nActual Caption: " +
    caption_array_to_str(words[c[k]])[0], fontsize=9)
        plt.axis('off')
        plt.show()
        # plt.savefig(dr + img_path.split("/")[-1], bbox_inches='tight')
        plt.close("all")
```

```
# In[ ]:


inception = tf.keras.applications.InceptionV3(weights='imagenet')
inception = tf.keras.Model(inception.input, inception.layers[-2].output)


# In[ ]:


best_model = models["create_pre_inject_model"] #change this to whatever model you want
predict_caption(best_model, "../input/images-for-test/test_images/", inception, 3, "")
```