Image Captioning With Attention and Transfer Learning

Oğuzhan Karaahmetoğlu
o.karaahmetoglu@bilkent.edu.tr
ID: 21901271

Selim Furkan Tekin
furkan.tekin@bilkent.edu.tr
ID: 21501391

Furkan Şahinuç
furkan.sahinuc@bilkent.edu.tr
ID: 21400503

Mümtaz Torunoğlu
mumtaz.torunoglu@ug.bilkent.edu.tr
ID: 21101549

***Abstract*—This document is prepared for the final project of the course EEE 543 Neural Networks. Aim of the project is to generate meaningful sentences describing a given image which is referred as Image Captioning in the literature. Due to its real-life applications, various solutions and datasets related to the Image Captioning problem were published and are available online. In this report, we first describe the analyses performed on the dataset. Then, we describe our approach to the problem by describing our model architecture and data flow. At every step, we compare our approach with the current works in the literature and explain the reasoning behind our algorithmic decisions by stating the advantages and disadvantages for different possibilities. Lastly, we provide results that present the performance of our approach.**

I. Introduction

Image captioning is the problem of generating meaningful sentences describing the contents of an image. Due to its possible real-life applications, it has attracted the attention of the researchers. Therefore various solutions regarding this problem were proposed in the literature. Almost all of these solutions divide the problem into two main parts, visual feature extraction and word generation mechanism.

Convolutional Neural Networks (CNN) [1] are the most common models used in the visual feature extraction part. The main reason is because the CNNs have proved their ability to model complex patterns in the input images for many different tasks. Because of their superior abilities in visual tasks, they became the most frequently used approach in visual feature extraction module. Although a simple CNN could produce a feature vector that is able to describe the image alone, in some approaches, an attention mechanism [2] is added to the CNN model to increase the feature extraction performance. Another approach is to detect objects in an image and feed the segmented objects and their spatial locations to a CNN model to guide the model to infer the spatial relations of the objects in an image [3]. Both of these approaches have shown significant performance improvements compared to a simple CNN model. However, it would mean a longer training process and more computational and memory resources if one decides to extend a simple CNN architecture with such modifications. On the other hand, some famous CNN models such as Inception [4] and VGGNET [5] were previously trained famous image classification benchmarks such as ImageNet challenge[6]. Inherent structure of these models have residual connections and pooling/normalization layers which have added performance improvements to a simple CNN model. Moreover, pretrained weights of these models allow them to be used in different tasks without training from scratch which is known as transfer learning [7].

In the second main part, words are generated using the visual features generated by extractor module. The dominant approach in this section is to use a Recurrent Neural Network (RNN) [8] to generate words iteratively. Although, some approaches use a simple Multi Layer Perceptron (MLP) and padded inputs to generate a word by observing the previously generated words only. However, thanks to their inherent structure, the recurrent networks have shown superior performance compared to other models in time series problems such as time series classification, regression, translation and also the sentence generation tasks. A special version of the RNNs is a Long Short Term Memory (LSTM) model [9]. This model has gated structures which allows it to be trained for longer iterations without suffering too much from the exploding or vanishing gradients problems as standard RNNs do. In LSTM sentence generation approaches, a word is generated at every step using the propagated state vector which contains all the temporal information from start until the current time step. Hence, they can capture the sentence generation rules and some grammatical rules. A more advanced approach is to use a bidirectional LSTM which passes through a sentence not only in forward direction but also in the backward direction [10]. Although this approach has shown better performance, it would require a more complex data flow.

Although a CNN model and an LSTM model is sufficient to model the visual features and generate caption describing those features, such model lacks the information about the word relations. In sentence generation task, co-occurrences and positions of word pairs are important. Some words and phrases can occur in pairs which should be inferred by the model. To model these word interactions, word embedding layers are used. A word embedding model generates a fixed

length vector for each word in the available vocabulary, i.e. the set of possible words to be generated. These vectors represent the connections between any two words in high dimensional vectors. It is possible to use pretrained embedding vectors for many words. Some examples are fast text [11] and GLOVE [12]. These embeddings were trained on online sources such a Wikipedia. Hence, they can model the word interactions without requiring a training from scratch.

In our work, we aim to present an image captioning model that is able to extract useful visual features and use them to generate meaningful and grammatically correct sentences. However, we have computational resource and memory limitations. Hence, we aim to present a simple model that does not require too much computational resources while showing adequate performance. Enforcing a model to generate grammatically correct sentences is hard. Hence, we evaluate our model with both a commonly used bilingual evaluation understudy (BLEU) score [13] and by manual visualizations.

II. Methods

A. Problem Description

We observe a set of images $X = \{x_i\}_{i=1}^{N}$ which are 3-dimensional arrays with dimensions (c, h, w) where c is the color channels, h is the height and w is the width of the image. For every image, there is a ground-truth caption y_i. A caption that has length n is a sequence of n words, i.e. $y_i = [w_j]_{j=0}^{n-1}$ is a caption with words w_j. The aim is to generate a caption, $\hat{y}_i = [\hat{w}_j]_{j=0}^{n-1}$, for a given image x_i. Hence, a mathematical model F should be designed such that a caption $\hat{y}_i$ will be generated for an image x_i as in (1).

$$\hat{y}_i = F(x_i) \tag{1}$$

We suffer the loss $\lambda(\hat{w}_i, w_i)$ for each generated word with corresponding ground truth word w_i. For a sentence, losses of all the words are summed which is $l(\hat{y}_i, y_i) = \sum_{j=0}^{n-1} \lambda(\hat{w}_j, w_j)$. And for all the images in the dataset, we suffer the sum of all caption losses which is $L(\hat{Y}, Y) = \sum_{i=1}^{N} \lambda(\hat{y}_i, y_i)$ where $\hat{Y} = \{\hat{y}_i\}_{i=1}^{N}$. In this problem, $w_i \in \{0, 1, 2, ..., W-1\}$ where W is the size of vocabulary, i.e. the number of possible ground-truth words. Hence, generated words, $\hat{w}_i \in \{0, 1, 2, ..., W-1\}$ also. For that reason, the problem is a multistep and multilabel classification problem with W classes.

In the following subsection, we will present the analyses that we have performed on the provided dataset and on a real-life publicly available dataset for comparison.

B. Data Analysis

A dataset was provided to us for the project. We have performed our experiments and analysis on the provided dataset and, for comparison purposes only, on a public dataset, Flickr8k. The provided dataset contains 72k images whereas publicly available Flickr8k dataset contains 8k images. In order to understand the structure (i.e. image dimensions, number of channels) and content of images (i.e. the common objects, light conditions, background), we have randomly sampled some images from the provided dataset. We realized that both the provided dataset and the Flickr8k dataset contains images with different width and height values. Also, some of the images had single color channel whereas others have three color channels. To simplify the data flow in our model, we have decided to work with fixed width, height and channel dimensions. To this end, we have transformed all the images to a fixed width, height and channel size which are 224, 224 and 3.

(a) Sample image drawn from the provided dataset. Outdoor light conditions.

(b) Sample image drawn from the provided dataset. Indoor light conditions.

Fig. 1: Two samples drawn from the provided dataset.

To analyze the image contents, we have sampled some images and inspected them. As seen from the figure 5, the dataset contains images with different ambiences. In figure 5, we can observe an image of a forest view which has high brightness level. On the contrary, the figure 1b, is taken indoors and hence it has a lower brightness level. Moreover, the objects that are referred in the corresponding caption are not always at the same places in the images.

We have also inspected the label sentences. Label sentences contain some special words marking special points in sentences. For example, every sentence starts with a start token and ends with an end token. All words after the end token are null tokens. Moreover, the provided dataset has an unknown token. Each sentence has a fixed length of 17. We have computed the histogram of the location of the end token in all the sentences. Also, we have computed that the ratio of unknown tokens in the whole dataset is 7.29 %. Also, we have noted that the location of the unknown token is not a fixed location. Therefore, we have concluded that we can simply ignore the presence of the unknown token and regard it as a regular word. Since it is not at a fixed location and has no connection with the sentence, our model should be able to learn that it is an out-of-vocabulary word that should not be generated. There are 1004 words in the vocabulary of the provided dataset whereas there are 6690 words in the Flickr8k dataset including the special tokens.

We also computed the histogram of every word which gives us an insight about the distribution of the words in all sentences. We have seen that the words ”a” and ”the” were the most commonly used two words. However, such words do not give any information since they were expected to be present in most, if not all, of the sentences. We have focused on the nouns and objects since they give an accurate information about

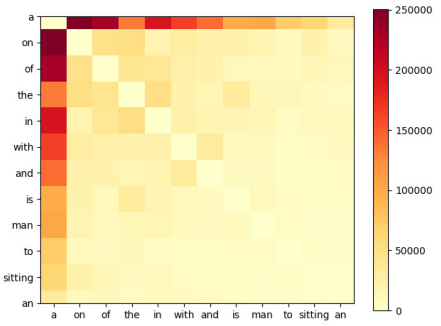


Fig. 2: Co-occurrence matrix for the top 10 most frequent words in the dataset. It can be seen that the word ”a” is used with all other words frequently which is an expected result.

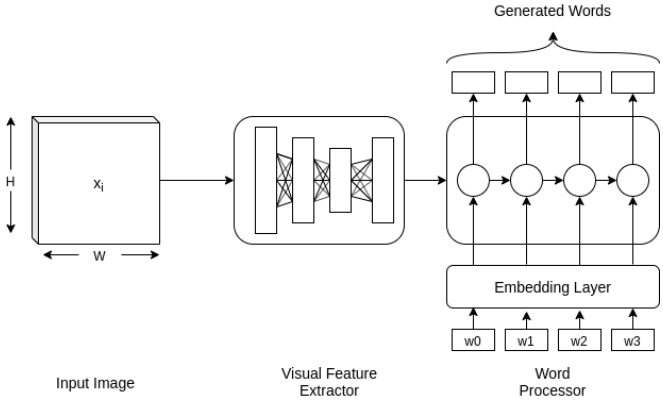


Fig. 3: Overall block diagram of our approach.

the content of the images. Most frequently used objects were common objects like ”bus”, ”tree”, ”person”. With this word analysis, we are able to get a crude estimation of the image contents. Since most of the objects appearing in the images are common objects, we can conclude that it is possible to apply previously proposed models that were applied to datasets with similar content.

Lastly, we have computed the co-occurence histogram for all the words in the vocabulary. This analysis gives insight about the correlation between word pairs. Co-occurence matrix is shown in figure 2. This knowledge could be used as a prior estimate for the word embedding matrix which relates the words in fixed length vectors. However, since the most frequent words are common objects, we have decided to use a pretrained embedding layer to generate word embeddings.

The main point we can conclude from our observations in data analysis is that the provided dataset has a content that is similar to the publicly available image and sentence datasets. Hence, we conclude that using pretrained model weights without training them would not limit the performance of our overall architecture.

C. Model Description

In this section we describe the mathematical structure of our proposed model. We also describe the design step by step and analyze each decision. We provide analysis of advantages and disadvantages of some possible design choices and explain the reason why we have chosen the current design. Our architecture consists of two main models, a visual feature extractor module and a word processing module. We can introduce the overall diagram with figure 3.

As seen from the figure 3, input images are first fed into a visual feature extractor model which extracts the visual features containing spatial information and connections about the image. These visual features are then fed to a word processor module which produces words that will form the caption for the corresponding input image. This word processor module has an inherent memory allowing it to carry past information while generating words iteratively. Hence, it can extract temporal features if necessary.

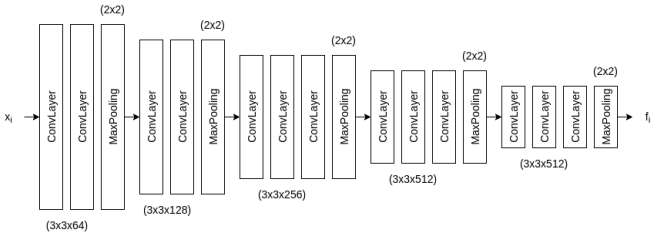


Fig. 4: VGGNet model architecture. Three-tuples represent the kernel size and number of filters of the convolutional layers whereas two-tuples represent the kernel size of the pooling layers.

In the following subsection, we describe the structure of the visual feature extractor and word processor separately.

1) Visual Feature Extractor: In the data analysis part, we have mentioned that the input images contain various different conditions. An input image may be a landscape which does not contain any dominant object or it can contain a group of many objects, e.g. 2 children. Hence, spatial connections and features in an image should be extracted from the images. For this reason, we have used a convolutional neural network (CNN).

Although they are powerful, training a CNN model requires too much time and memory resource which we lack. We could construct a minimal network with small number of parameters and train it on the given dataset but, that would limit our model performance by our memory and time constraints. For that reason, we have downloaded a pretrained model and disabled training to save memory and time. One could argue that using a pretrained model that was trained on a specific task would not achieve an adequate performance on a different problem. However, in the data analysis section we have concluded that the contents of the images mostly consist common objects. Hence, they are familiar to famous vision benchmarks, e.g. Imagenet. Therefore, we have decided to use a VGG16 model that was pretrained on the Imagenet dataset. We have discarded the fully connected layer attached to the output of the CNN part of the VGG16 and used the convolutional section as our visual feature extractor.

Structure of the VGG16 can be seen from the figure 4. Note that the pretrained VGG16 model was trained on images with size (224, 224). Hence, we are restricted to use these dimensions. So, we preprocess the input images, by random cropping and scaling, to have fixed sizes (224, 224). We have also tested our approach with the pretrained Inception model which did not bring any significant improvements. We conclude that this is because the VGG16 model was able to model visual features in the input images well enough.

There are also object detection approaches that aim to model spatial connections between objects better by focusing on the objects. We have abandoned this idea for two reasons, firstly because some of the images do not contain any dominant object, but a view. Secondly, it would require more time and computational resources to train such models since it is not easy to find pretrained models. Also, in some of the approaches, attention mechanism was added to the CNN model to increase performance.

At the output of the visual feature extractor, we obtain a tensor with size (7, 7, 512). This tensor contains all the description of the input image which will be used in word generation process. In the next section, we describe how we use this feature vector to obtain words of a caption one by one.

2) Word Processing Module: This module takes the visual features extracted by the CNN model and generates the words of the caption of the input image. We generate each word one by one. Each generated word needs information about the words generated before it. Hence an inherent memory is required in the word processing module. We solve this problem by using a Long Short Term Memory (LSTM) to carry state information about the caption sentence. There were other alternatives to LSTM that carry the state information such as RNN or GRU. In our experiments, we have also observed that LSTM has shown better performance.

We feed the visual feature vector as the state vector of the LSTM as shown in figure 5a. We represent the words as one-hot vectors with length W which is the size of the vocabulary. At first step, the start token is fed as input to the LSTM as one-hot vector and the new state vector is generated. After this step, the new state vector is passed to a Multi Layer Perceptron (MLP) to obtain a length W vector, one value for each character. This vector is obtained after a dense layer with dropout rate 0.3 followed by a dense layer followed by a softmax layer. Hence the elements of the vector represent the probability distributions of all the words that can be generated for the next step. The next state vector can be generated in two ways. We can either use the word that was generated by our model as the input of the next step and continue to generate in this manner. Or, we could generate a word and use the ground-truth word as input to the next step which is called teacher-forcing method [14]. We have experimented with both of these methods and observed that the latter approach converges to a score faster while the former approach yields captions with more word repetitions.

If the input word vectors are fed as one-hot vectors, we expect the model to infer the correlations and connections between two words. However, in order to provide this knowledge prior to training, we are using an embedding layer which converts the one-hot word vectors to fixed length embedding vectors. These embedding vectors are used to model the connection and distance between two words.

To overcome this problem, we use a pretrained embedding model Fasttext which covers all the vocabularies that are present in the provided dataset. There were also other pretrained embedding vectors such ass GLOVE. We could use most of the available embedding vectors since they are all trained in similar tasks. Since we are using pretrained vectors, we do not need to infer the word connections from scratch. Special tokens, start, end and unknown, have random embedding vectors since they have no meaning and connection to other words.

We compute the initial state of the LSTM layer from the visual feature vectors as shown in figure 5a. We have added an attention mechanism to our LSTM model which is shown in figure 5b. The same flow structure is implemented in a previous work [2]. Attention mechanism weights some parts of the visual feature vector. As in [15], our goal is to create a context vector and feed it to input gates of LSTM cells. Paper defines a mechanism ϕ, that computes the context vector, $\hat{z}_t$, from annotation vectors, n_i for $i = 1, \ldots, L$, corresponding to the features extracted at different image locations. This weighting is controlled by α_i vector which is calculated by the attention model, f_{att} showed in figure 5c. This can be interpreted as the relative importance of a location in an image. Here is the mathematical representation of attention mechanism,

$$
\begin{aligned}
e_j^{(k)} &= f_{\text{att}}\left(\mathbf{n}_i^{(k)}, \mathbf{h}_{j-1}\right) \\
\alpha_j^{(k)} &= \frac{\exp\left(e_j^{(k)}\right)}{\sum_{k=1}^{L} \exp\left(e_j^{(k)}\right)}
\end{aligned}
\tag{2}
$$

After the weights are calculated paper proposes two methods to implement ϕ. We selected the soft attention mechanism, which is the expectation of context vector. Then this context vector is concatenated with word embedding and given as input to the first layer of LSTM units.

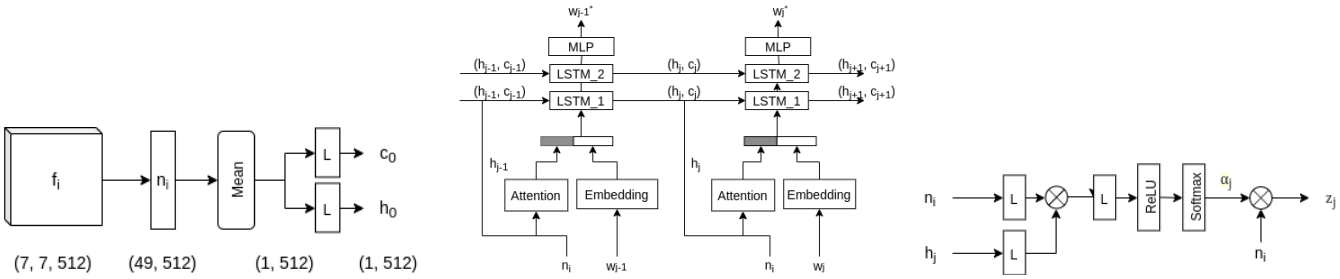
$$
\mathbb{E}_{(s_j|n_i)}\left[\hat{\mathbf{z}}_j\right] = \sum_{k=1}^{L} \alpha_j^{(k)} \mathbf{n}_i^{(k)} \tag{3}
$$

where s_j is the location to focus attention when generating the jth word.

We believe that by adding this mechanism, we both overcome any possible limitation introduced by the pretrained model weights and introduced a mechanism to capture complex connections in a sentence.

D. Optimization And Scoring

In this section, we describe the optimization technique and the score function that was used to evaluate the proposed solution.



(a) State generating functions. (b) Word processor flow. (c) Attention mechanism.

Fig. 5: Word processor submodule structures.

For a given image x_i, our model generates a caption $\hat{y}_i = [\hat{w}_j]_{j=0}^{n-1}$ which has the ground-truth caption $y_i = [w_j]_{j=0}^{n-1}$. After generating the whole caption for an image, we compute the sum of losses for all words and update the model parameters according to this loss function with the ADAM optimizer [16] with learning rate 0.003. Loss function that was used for calculating the word errors is categorical cross-entropy function which has the following form in (4).

$$\lambda(\hat{w}_j, w_j) = -\sum_{k=0}^{W-1} w_j^{(k)} \log \hat{w}_j^{(k)} \tag{4}$$

Where the vectors w_j are ground-truth one-hot word vectors and the vectors $\hat{w}_j$ are the generated word vectors that are obtained after a softmax layer. After calculating the loss for a single word, losses are summed for all the words in a caption and then the model parameters are updated. As mentioned in the data analysis section, we have noted that there is a high imbalance in the word occurrences in the dataset. Computing cross-entropy and gradients may result in a limited performance due to this imbalance. Because the model could simply ignore the visual features and generate the word "a" for all the images even if it is not needed in the caption. We solve this problem by applying a weighting mechanism for the losses of all classes. We assign a higher weight for the rare words and assign low weighting for the loss of the frequent words. In this way, we aim to solve the imbalance problem with the weighting mechanism so that the probability distribution of the words are not affected by the occurrences of the words. We specifically use the logarithm of the inverse frequency of the words as their weights. This weight for a single word, say the ith word, is computed as $a_i = \log \frac{m_i}{M}$ where m_i is the number of occurrences of the the ith word and M is the total number of words in the whole dataset. Hence, we modify the loss function with these class weights as shown in (5). Throughout our experiments, we have seen that using class weights improved our performance greatly by not generating the most common words over and over again.

$$\lambda_a(\hat{w}_j, w_j) = -\sum_{k=0}^{W-1} a_k w_j^{(k)} \log \hat{w}_j^{(k)} \tag{5}$$

We have divided the dataset into three groups, training, validation and test which have ratios 0.7, 0.2, 0.1. And instead of computing the derivative for a single sample and updating, we have used the batch learning system where we feed 200 images simultaneously and compute the gradients for all the images. With this setup we have trained our model for approximately 50 epochs which took around 7 hours on NVIDIA 1080 Ti GPU. This time consumption is relatively lower than the training times of many other models in the literature since some parts of our model consists of pretrained parts which reduce the training time greatly.

In order to decide when to finish the training session, we are using an evaluation score. We stop the training procedure after the training score is not increased for 3 iterations. The score we are using is the bilingual evaluation understudy (BLEU) score. BLEU score is basically the average of the n-gram matches. For unigram, bigram and up to n-gram, all possible n-grams in the generated sequence are checked if they exist in the reference sentences. Recall that in the provided dataset we were given multiple sentences for each image. Hence, we can easily compute the BLEU score. The reason we are using the BLEU score is because it is considered as the most successful evaluation metric in machine translation by many sources.

III. Results

As we have mentioned before, we have used cross-entropy loss to update our model. We also keep track of the cross-entropy loss during the training and the validation process. The loss curve for training and validation is shown in figure 6. We can see that loss converges to a certain score. To test the significance of the converged level, we have computed p-value for the loss. We found that the converged value is below 2.4 with probability 0.95. Therefore, we conclude that the performance of our approach is upper-bounded by a certain level.

In addition to the loss curve, we also have BLEU scores for validation step. We noted that BLEU score increases and converges to a certain level. Again, to test the significance of this level, we have computed p-value. We found that the probability of our algorithm reaching to a level higher than the 0.45 is 0.89 for unigram. Again, we conclude that the validation BLEU score of our algorithm is lower bounded by a certain level.

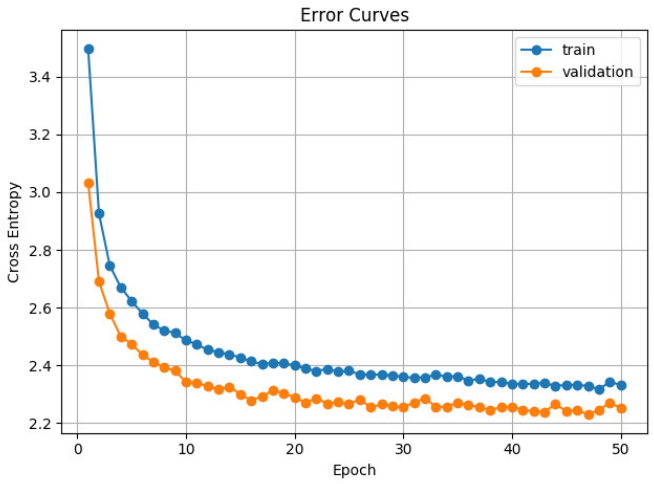


Fig. 6: Categorical cross entropy score of the proposed model for every epoch on training and validation sets. As it can be seen from this figure, cross entropy loss converges to a certain level. Since the validation loss also does not increase, model does not overfit the training data.

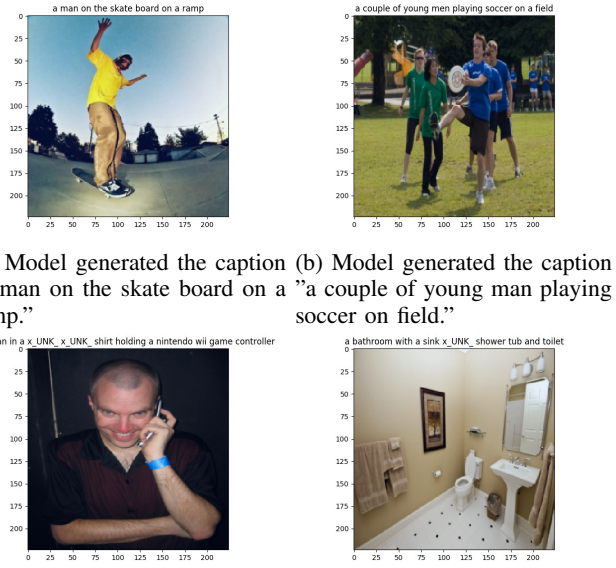
N-GRAM	BLEU Score
1	0.5044
2	0.2817
3	0.1490
4	0.0794

TABLE I: BLEU scores calculated for different lengths of ngrams.

We have also visualized the generated captions to see whether they are meaningful sentences, i.e. if they grammatically correct sentences. Some sample images can be seen in figure 7. As it can be seen from figure , generated captions have learned some grammatical structures and some special structures such as generating null token after the end token. We have checked this property in the captions generated by our model because we use the BLEU score to evaluate our model. Although it is considered as the best way to measure translation performance by many sources, it does not consider the grammatical structure of the generated captions which we evaluate manually by visualizing.

IV. Discussion

We have performed analyses on both the data before the training process and on the model during the training process. In data analysis part, we have checked the content and properties of the input images and also the content of the captions. These analyses have given us insight about the possible problems that could be encountered in the training. For example, we have computed histogram of word occurrences which had pushed us to the idea of weighting the losses for all classes. We have seen the benefit of this analysis by inspecting the generated captions.



(a) Model generated the caption "a man on the skate board on a ramp."

(b) Model generated the caption "a couple of young man playing soccer on field."

(c) Model generated the caption "a man in a x_UNK_ x_UNK_ shirt holding a wii controller."

(d) Model generated the caption "a bathroom with a sink x_UNK_ shower tub and toilet."

Fig. 7: Four example captions generated for the shown figures. In 7a, we can see that the model generated a meaningful sentence describing the figure completely. In figure 7b, generated sentence describes the image content although the model missed a woman in the picture. In figure 7c, caption describes the image however, the object that the man is holding is a phone. Also in figures 7c, 7d, model generated the unknown token since we do not restrict the model to do so. However, they do not ruin the generated caption.

Another analysis we have performed is the co-occurrences of words in the captions. We have performed this analysis on both the provided dataset and on the Flickr8k dataset. The outcomes have shown similar results which led us to conclude that the content of the captions are similar. Though, we failed to use this information directly like the word occurrences. We have assumed that the prior co-occurrences could aid the learning process of the model without requiring an additional mechanism that connects related words. We have assumed that the embedding vectors of the Fasttext contains this information already.

A similar reasoning was followed in the visual feature extraction part where we have analyzed the most frequent object occurrences in the captions to gain an insight about the contents of the images. Again, we have assumed that contents of the images are similar to the famous ImageNet dataset which had pretrained convolutional models available. We assumed that a pretrained model could identify the objects that are present in the images that are in the provided dataset.

As we have stated earlier, we were aiming to propose a captioning model that could generate captions related to the input images and captions that are grammatically correct. However, we constrained ourselves with memory and time limits. Hence,

we have preferred to use pretrained models. Although we have achieved our goals, we could further increase this performance by training our models from scratch. Also, the ability of our model to connect too distant words in a sentence is somewhat limited although the captions describe the images. To solve this problem, we could use multi-layer recurrent network to overcome this problem. But again, this would require extra computational resources and time.

To sum up, we have reached our goals that we have stated in the introduction part with small limitations. The proposed solution could be improved by adding additional features to the existing model such as visual attentions, object detection, training from scratch and more complex training flows. However, these additional features are out of the scope of this report which was aimed to propose a captioning model that could produce describe the contents of the images with minimal computational and memory resource usage.

REFERENCES

[1] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. IEEE, 2010, pp. 253–256.

[2] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *International conference on machine learning*, 2015, pp. 2048–2057.

[3] J. Johnson, A. Karpathy, and L. Fei-Fei, "Densecap: Fully convolutional localization networks for dense captioning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4565–4574.

[4] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[7] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[8] J. Koutnik, K. Greff, F. Gomez, and J. Schmidhuber, "A clockwork rnn," *arXiv preprint arXiv:1402.3511*, 2014.

[9] S. Hochreiter and J. Schmidhuber, "Lstm can solve hard long time lag problems," in *Advances in neural information processing systems*, 1997, pp. 473–479.

[10] C. Wang, H. Yang, C. Bartz, and C. Meinel, "Image captioning with deep bidirectional lstms," in *Proceedings of the 24th ACM international conference on Multimedia*. ACM, 2016, pp. 988–997.

[11] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[12] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[13] 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 on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[14] A. M. Lamb, A. G. A. P. Goyal, Y. Zhang, S. Zhang, A. C. Courville, and Y. Bengio, "Professor forcing: A new algorithm for training recurrent networks," in *Advances In Neural Information Processing Systems*, 2016, pp. 4601–4609.

[15] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," in *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2016, pp. 4945–4949.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[17] "Show Attend Tell Code Implementation," Feb. 2019. [Online]. Available: https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning

[18] "neural-dialogue-metrics/BLEU," Apr. 2019. [Online]. Available: https://github.com/neural-dialogue-metrics/BLEU

V. APPENDIX

Below you can find the implementation of the proposed algorithm that is written in Python 3.7 using PyTorch. We have implemented and tested the code on the NVIDIA 1080 Ti GPU. We have surveyed some of the blogs to gain insight about the approaches. Also, to extend our pytorch knowledge, we have checked some of the online sources and libraries such as[17] and [18].

```
model_params = {
        'drop_prob': 0.3,
        'n_layers': 2,
        'lstm_dim': 512,
        'conv_dim': 512,
        'att_dim': 512,
        'transfer_learn': True,
        'pre_train': True,
    }

train_params = {
    'n_epoch': 50,
    'clip': 5,
    'lr': 0.003,
    'seq_len': 17,
    'eval_every': 100,
    'show_image': False
}

batch_params = {
    'batch_size': 16,
    'num_works': 0,
    'shuffle': True,
    'use_transform': True,
    "input_size": (224, 224)
}
```

```
    1: import os
    2: import csv
    3: import random
    4: import pandas as pd
    5:
    6: from PIL import Image
    7:
    8:
    9: class LoadData:
   10:     def __init__(self, dataset_path, images_path, **data_params):
   11:         self.word2int, self.int2word = self.__create_dicts(dataset_path)
   12:         self.caption_words = self.__create_caption_words(dataset_path)
   13:         self.captions_int = self.__create_captions_int(dataset_path)
   14:         self.image_addr = self.__create_image_addr(dataset_path)
   15:         self.image_paths = self.__create_image_paths(images_path)
   16:
   17:         self.test_ratio = data_params.get('test_ratio', 0.1)
   18:         self.val_ratio = data_params.get('val_ratio', 0.1)
   19:         self.shuffle = data_params.get('shuffle', True)
   20:         self.data_dict = self.__split_data()
   21:
   22:     def __split_data(self):
   23:         dataset_length = len(self.image_paths)
   24:         if self.shuffle:
   25:             random.shuffle(self.image_paths)
   26:
   27:         test_count = int(dataset_length * self.test_ratio)
   28:         val_count = int(dataset_length * self.val_ratio)
   29:
   30:         data_dict = dict()
   31:         data_dict['test'] = self.image_paths[:test_count]
   32:         data_dict['validation'] = self.image_paths[test_count:test_count + val_count]
   33:         data_dict['train'] = self.image_paths[test_count + val_count:]
   34:
   35:         return data_dict
   36:
   37:     @staticmethod
   38:     def __create_caption_words(dataset_path):
   39:         """
   40:         :param dataset_path: string
   41:         :return: pd.DataFrame
   42:         """
   43:         caption_word_path = os.path.join(dataset_path, 'captions_words.csv')
   44:         caption_words = pd.read_csv(caption_word_path)
   45:         return caption_words
   46:
   47:     @staticmethod
   48:     def __create_captions_int(dataset_path):
   49:         """
   50:         :param dataset_path: string
   51:         :return: pd.DataFrame
   52:         """
   53:         caption_path = os.path.join(dataset_path, 'captions.csv')
   54:         captions = pd.read_csv(caption_path)
   55:         return captions
   56:
   57:     @staticmethod
   58:     def __create_image_addr(dataset_path):
   59:         """
   60:         :param dataset_path: str
   61:         :return: pd.DataFrame
   62:         """
   63:         im_addr_path = os.path.join(dataset_path, 'imid.csv')
```

```
   64:         im_addr = pd.read_csv(im_addr_path)
   65:         return im_addr
   66:
   67:     @staticmethod
   68:     def __create_image_paths(images_path):
   69:         """
   70:         :param images_path: string
   71:         :return: list of strings
   72:         """
   73:         image_paths = [os.path.join(images_path, f) for f in sorted(os.listdir(images_p
ath))]
   74:         image_paths.pop(0)  # remove .gitignore file from the list
   75:
   76:         # clean the dataset
   77:         for im_path in image_paths:
   78:             try:
   79:                 image = Image.open(im_path)
   80:                 image.verify()
   81:             except OSError:
   82:                 os.remove(im_path)
   83:                 image_paths.remove(im_path)
   84:
   85:         return image_paths
   86:
   87:     @staticmethod
   88:     def __create_dicts(dataset_path):
   89:
   90:         word2int_csv_path = os.path.join(dataset_path, 'word2int.csv')
   91:
   92:         data = []
   93:         with open(word2int_csv_path, mode='r') as infile:
   94:             reader = csv.reader(infile)
   95:             for row in reader:
   96:                 data.append(row)
   97:
   98:         word2int = {}
   99:         for word, value in zip(data[0], data[1]):
  100:             word2int[word] = int(float(value))
  101:
  102:         word2int = {k: v for k, v in sorted(word2int.items(), key=lambda item: item[1])
}
  103:         int2word = {v: k for k, v in word2int.items()}
  104:         return word2int, int2word
```

```
    1: import torch
    2: import collections
    3: import numpy as np
    4: import matplotlib.pyplot as plt
    5: import torch.nn.functional as F
    6:
    7: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    8:
    9:
   10: def predict(net, image, x_cap, h=None, top_k=None):
   11:     image = image.unsqueeze(dim=0)
   12:
   13:     x_cap = torch.tensor([[x_cap]]).to(device)
   14:
   15:     out, h = net(image, x_cap, sentence_len=1, hidden=h)
   16:     p = F.softmax(out, dim=1).data
   17:
   18:     if torch.cuda.is_available():
   19:         p = p.cpu()
   20:
   21:     p, top_ch = p.topk(top_k)
   22:     top_ch = top_ch.numpy().squeeze()
   23:
   24:     p = p.numpy().squeeze()
   25:     word_int = np.random.choice(top_ch, p=p / p.sum())
   26:
   27:     return word_int, h
   28:
   29:
   30: def sample(net, batch_gen, top_k=None, **kwargs):
   31:     net.to(device)
   32:     net.eval()
   33:
   34:     batch_size = batch_gen.batch_size
   35:     seq_length = kwargs['seq_len']
   36:
   37:     im, _, y_cap = next(batch_gen.generate('validation'))
   38:     im, y_cap = im.to(device), y_cap.to(device)
   39:
   40:     x_cap = 1  # x_START_
   41:     captions = []
   42:     for i in range(batch_size):
   43:         print('\rsample:{}'.format(i), flush=True, end='')
   44:         caption = []
   45:         h = None
   46:         for ii in range(seq_length):
   47:             x_cap, h = predict(net, im[i, :], x_cap, h, top_k=top_k)
   48:             caption.append(x_cap)
   49:         captions.append(caption)
   50:
   51:     print('\n')
   52:     for i in range(batch_size):
   53:         caption_str = translate(captions[i], net.embed_layer.int2word)
   54:         print('Caption {}: {}'.format(str(i), caption_str))
   55:         if kwargs['show_image']:
   56:             show_image(im[i], caption_str)
   57:     plt.show()
   58:
   59:     net.train()
   60:     return captions
   61:
   62:
   63: def show_image(img, caption_str):
```

```
64:     if torch.cuda.is_available():
65:         img = img.cpu()
66:
67:     plt.figure()
68:     img = (img.permute(1, 2, 0) - img.min()) / (img.max() - img.min())
69:     plt.imshow(img)
70:     plt.tight_layout()
71:     plt.title(caption_str)
72:
73:
74: def translate(captions, int2word, remove_unk=False):
75:     caption_str = ' '.join([int2word[cap] for cap in captions])
76:     caption_str = caption_str.replace('x_END_', '').replace('x_NULL_', '')
77:     caption_str = ' '.join(caption_str.split())
78:     if remove_unk:
79:         caption_str = caption_str.replace('x_UNK_', '')
80:         caption_str = ' '.join(caption_str.split())
81:     return caption_str
82:
83:
84: def calc_class_weights(captions_int):
85:     counts = collections.Counter(captions_int.flatten())
86:     counts_array = np.array(list(counts.values()))
87:
88:     # calculating idf
89:     word_count = len(captions_int.flatten())
90:     counts_array = np.log(word_count / counts_array)
91:     counts_tensor = torch.from_numpy(counts_array).float().to(device)
92:
93:     return counts_tensor
```

```
 1: import pickle
 2:
 3: from config import model_params, batch_params, train_params
 4: from load_data import LoadData
 5: from batch_generator import BatchGenerator
 6: from models.caption_model import CaptionLSTM
 7: from train_helper import calc_class_weights
 8: from train_helper import sample
 9: from train import train
10: from test import test
11:
12:
13: def main(mode):
14:     print('Loading data...')
15:     data = LoadData(dataset_path='dataset',
16:                     images_path='dataset/images/')
17:
18:     print('Creating Batch Generator...')
19:     batch_creator = BatchGenerator(data_dict=data.data_dict,
20:                                    captions_int=data.captions_int,
21:                                    image_addr=data.image_addr,
22:                                    **batch_params)
23:
24:     if mode == 'train':
25:         print('Creating Models...')
26:         caption_model = CaptionLSTM(model_params=model_params,
27:                                     int2word=data.int2word)
28:
29:         print('Starting training...')
30:         class_weights = calc_class_weights(data.captions_int.values)
31:         train(caption_model, batch_creator, class_weights, **train_params)
32:
33:     elif mode == 'sample':
34:         print('Loading model...')
35:         model_file = open('vgg_lstm.pkl', 'rb')
36:         model = pickle.load(model_file)
37:         print('Creating sample..')
38:         sample(model, batch_creator, top_k=10, seq_len=16, show_image=True)
39:
40:     elif mode == 'test':
41:         print('Loading model')
42:         model_file = open('vgg_lstm.pkl', 'rb')
43:         model = pickle.load(model_file)
44:         print('Testing model...')
45:         test(model, batch_creator, top_k=10, seq_len=16)
46:
47:
48: if __name__ == '__main__':
49:     run_mode = 'test'
50:     main(run_mode)
```

```
 1: import torch
 2: import pickle
 3: import numpy as np
 4: import torch.nn as nn
 5: import torch.optim as optim
 6:
 7:
 8: from train_helper import sample
 9:
10: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
11:
12:
13: def train(net, batch_gen, weights, **kwargs):
14:     net.train()
15:     net.to(device)
16:
17:     batch_size = batch_gen.batch_size
18:     seq_length = kwargs['seq_len']
19:
20:     opt = optim.Adam(net.parameters(), lr=kwargs['lr'])
21:     criterion = nn.CrossEntropyLoss(weight=weights)
22:
23:     train_loss_list = []
24:     val_loss_list = []
25:     for epoch in range(kwargs['n_epoch']):
26:         running_loss = 0
27:         for idx, (im, x_cap, y_cap) in enumerate(batch_gen.generate('train')):
28:
29:             print('\rtrain:{}'.format(idx), flush=True, end='')
30:
31:             im, x_cap, y_cap = im.to(device), x_cap.to(device), y_cap.to(device)
32:
33:             opt.zero_grad()
34:             output, _ = net(im, x_cap)
35:
36:             loss = criterion(output, y_cap.view(batch_size * seq_length).long())
37:             loss.backward()
38:
39:             nn.utils.clip_grad_norm_(net.parameters(), kwargs['clip'])
40:             opt.step()
41:
42:             running_loss += loss.item()
43:
44:             if (idx+1) % kwargs['eval_every'] == 0:
45:                 print('\n')
46:                 val_loss = evaluate(net, batch_gen, weights, **kwargs)
47:                 print("\nEpoch: {}/{}...".format(epoch + 1, kwargs['n_epoch']),
48:                       "Step: {}...".format(idx),
49:                       "Loss: {:.4f}...".format(running_loss / idx),
50:                       "Val Loss: {:.4f}".format(val_loss))
51:
52:         # After 15 epochs open last parts of conv model
53:         if epoch > 15:
54:             net.fine_tune()
55:
56:         print('Creating sample captions')
57:         sample(net, batch_gen, top_k=5, **kwargs)
58:         print('\n')
59:
60:         train_loss_list.append(running_loss / idx)
61:         val_loss_list.append(val_loss)
62:
63:         loss_file = open('losses.pkl', 'wb')
```

```
64:         model_file = open('vgg_lstm.pkl', 'wb')
65:         pickle.dump([train_loss_list, val_loss_list], loss_file)
66:         pickle.dump(net, model_file)
67:
68:     print('Training finished, saving the model')
69:     model_file = open('vgg_lstm.pkl', 'wb')
70:     pickle.dump(net, model_file)
71:
72:
73: def evaluate(net, batch_gen, weights, **kwargs):
74:     net.eval()
75:
76:     batch_size = batch_gen.batch_size
77:     seq_length = kwargs['seq_len']
78:
79:     criterion = nn.CrossEntropyLoss(weight=weights)
80:
81:     val_losses = []
82:     for idx, (im, x_cap, y_cap) in enumerate(batch_gen.generate('validation')):
83:
84:         print('\rval:{}'.format(idx), flush=True, end='')
85:
86:         im, x_cap, y_cap = im.to(device), x_cap.to(device), y_cap.to(device)
87:
88:         output, _ = net(im, x_cap)
89:         val_loss = criterion(output, y_cap.view(batch_size * seq_length))
90:
91:         val_losses.append(val_loss.item())
92:
93:     net.train()
94:     return np.mean(val_losses)
95:
```

```
from dataset import ImageDataset
from torchvision import transforms
from torch.utils.data import DataLoader


class BatchGenerator:
    def __init__(self, data_dict, captions_int, image_addr, **kwargs):
        self.data_dict = data_dict
        self.captions_int = captions_int
        self.image_addr = image_addr

        self.batch_size = kwargs.get('batch_size', 16)
        self.num_works = kwargs.get('num_works', 4)
        self.shuffle = kwargs.get('shuffle', True)
        self.use_transform = kwargs.get('use_transform', True)
        self.input_size = kwargs.get("input_size", (224, 224))

        self.dataset_dict, self.dataloader_dict = self.__create_data()

    def generate(self, data_type):
        """
        :param data_type: can be 'test', 'train' and 'validation'
        :return: img tensor, label numpy_array
        """
        selected_loader = self.dataloader_dict[data_type]
        yield from selected_loader

    def __create_data(self):
        if self.use_transform:
            im_transform = transforms.Compose([
                transforms.Resize(self.input_size),
                transforms.RandomHorizontalFlip(),
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])
            ])
        else:
            im_transform = None

        im_dataset = {}
        for i in ['test', 'train', 'validation']:
            return_all = True if i == 'test' else False
            im_dataset[i] = ImageDataset(image_path_names=self.data_dict[i],
                                         captions_int=self.captions_int,
                                         im_addr=self.image_addr,
                                         transformer=im_transform,
                                         return_all=return_all)

        im_loader = {}
        for i in ['test', 'train', 'validation']:
            im_loader[i] = DataLoader(im_dataset[i],
                                      batch_size=self.batch_size,
                                      shuffle=self.shuffle,
                                      num_workers=self.num_works,
                                      drop_last=True)
        return im_dataset, im_loader
```

```
 1: import torch
 2: import numpy as np
 3: import torch.nn as nn
 4:
 5: from transformers.word2vec import Word2VecTransformer
 6:
 7:
 8: class Embedding(nn.Module):
 9:     def __init__(self, int2word):
10:         nn.Module.__init__(self)
11:
12:         self.int2word = int2word
13:         self.vocab_size = len(int2word)
14:
15:         self.weights = self.create_embed_tensor()
16:         self.embed_dim = self.weights.shape[1]
17:         self.embedding = nn.Embedding.from_pretrained(self.weights)
18:
19:     def forward(self, captions):
20:         """
21:         :param captions: (B, S)
22:         :return: vectors: (B, S, Vector_dim)
23:         """
24:         vectors = self.embedding(captions)
25:         return vectors
26:
27:     def create_embed_tensor(self):
28:         transformer = Word2VecTransformer()
29:         vectors = []
30:         for i in range(self.vocab_size):
31:             vec = transformer.transform(self.int2word[i])
32:             vectors.append(vec)
33:         embed = np.stack(vectors, axis=0)
34:         return torch.from_numpy(embed)
```

```
    1: import torch.nn as nn
    2:
    3:
    4: class Attention(nn.Module):
    5:     def __init__(self, conv_dim, lstm_dim, att_dim):
    6:         super(Attention, self).__init__()
    7:         self.conv_att = nn.Linear(conv_dim, att_dim)
    8:         self.lstm_att = nn.Linear(lstm_dim, att_dim)
    9:         self.full_att = nn.Linear(att_dim, 1)
   10:         self.relu = nn.ReLU()
   11:         self.softmax = nn.Softmax(dim=1)
   12:
   13:     def forward(self, conv_out, lstm_hidden):
   14:         """
   15:         Forward propagation.
   16:         :param conv_out: encoded images, a tensor of dimension (batch_size, num_pixels,
 encoder_dim)
   17:         :param lstm_hidden: previous decoder output, a tensor of dimension (batch_size,
 decoder_dim)
   18:         :return: attention weighted encoding, weights
   19:         """
   20:         att1 = self.conv_att(conv_out)  # (batch_size, num_pixels, attention_dim)
   21:         att2 = self.lstm_att(lstm_hidden)  # (batch_size, attention_dim)
   22:         att = self.full_att(self.relu(att1 + att2.unsqueeze(1))).squeeze(2)  # (batch_s
ize, num_pixels)
   23:         alpha = self.softmax(att)  # (batch_size, num_pixels)
   24:         attention_weighted_encoding = (conv_out * alpha.unsqueeze(2)).sum(dim=1)  # (ba
tch_size, encoder_dim)
   25:
   26:         return attention_weighted_encoding
```

```
 1: import torch.nn as nn
 2: import torchvision.models as models
 3:
 4:
 5: class VGG16(nn.Module):
 6:     def __init__(self, model_params):
 7:         super(VGG16, self).__init__()
 8:         self.transfer_learn = model_params.get('transfer_learn', True)
 9:         self.pre_train = model_params.get('pre_train', True)
10:
11:         # init model
12:         self.vgg = models.vgg16(pretrained=self.pre_train)
13:
14:         modules = list(self.vgg.children())[:-2]
15:         self.vgg = nn.Sequential(*modules)
16:
17:         self.initialize_model()
18:
19:     def forward(self, image):
20:         return self.vgg(image)
21:
22:     def initialize_model(self):
23:         # close the parameters for training
24:         if self.transfer_learn:
25:             for param in self.vgg.parameters():
26:                 param.requires_grad = False
27:
28:     def fine_tune(self):
29:         for c in list(self.vgg.children())[5:]:
30:             for p in c.parameters():
31:                 p.requires_grad = True
```

```
 1: import torch
 2: import torch.nn as nn
 3:
 4: from models.embed import Embedding
 5: from models.attention import Attention
 6: from models.vgg import VGG16
 7:
 8: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
 9:
10:
11: class CaptionLSTM(nn.Module):
12:     def __init__(self, model_params, int2word):
13:         super(CaptionLSTM, self).__init__()
14:         self.drop_prob = model_params.get('drop_prob', 0.3)
15:         self.n_layers = model_params.get('n_layers', 2)
16:         self.lstm_dim = model_params.get('lstm_dim', 512)
17:         self.conv_dim = model_params.get('conv_dim', 512)
18:         self.att_dim = model_params.get('att_dim', 512)
19:
20:         self.embed_layer = Embedding(int2word)
21:         self.embed_dim = self.embed_layer.embed_dim
22:         self.vocab_dim = self.embed_layer.vocab_size
23:
24:         self.conv_model = VGG16(model_params)
25:         self.attention = Attention(self.conv_dim, self.lstm_dim, self.att_dim)
26:         self.lstm = nn.LSTM(input_size=self.embed_dim + self.lstm_dim,
27:                             hidden_size=self.lstm_dim,
28:                             num_layers=self.n_layers,
29:                             dropout=self.drop_prob,
30:                             batch_first=True)
31:         self.drop_out = nn.Dropout(self.drop_prob)
32:         self.fc = nn.Linear(self.lstm_dim, self.vocab_dim)
33:
34:         self.lin_h = nn.Linear(self.conv_dim, self.lstm_dim)
35:         self.lin_c = nn.Linear(self.conv_dim, self.lstm_dim)
36:
37:     def forward(self, image, x_cap, sentence_len=17, hidden=None):
38:         """
39:         :param image:
40:         :param x_cap: b, seq_len
41:         :param sentence_len: int
42:         :param hidden: tuple((num_layers, b, n_hidden), (num_layers, b, n_hidden))
43:         :return:
44:         """
45:         batch_size = x_cap.shape[0]
46:
47:         image_vec = self.conv_model(image)
48:         image_vec = image_vec.view(batch_size, -1, self.conv_dim)
49:
50:         if hidden is None:
51:             h, c = self.init_hidden(image_vec)
52:             # expand it for each layer of image
53:             h = h.expand((self.n_layers, batch_size, self.lstm_dim)).contiguous()
54:             c = c.expand((self.n_layers, batch_size, self.lstm_dim)).contiguous()
55:         else:
56:             h, c = hidden
57:
58:         sentence = []
59:         for word_idx in range(sentence_len):
60:             weigted_conv_output = self.attention(image_vec, h[0, :])
61:             embed = self.embed_layer(x_cap[:, word_idx]).float()
62:             lstm_in = torch.cat([embed, weigted_conv_output], dim=1).unsqueeze(1)
63:             r_out, (h, c) = self.lstm(lstm_in, (h, c))
```

```
64:
65:             out = self.fc(self.drop_out(r_out))
66:             sentence.append(out)
67:
68:         output = torch.cat(sentence, dim=1)
69:         output = output.contiguous().view(-1, self.vocab_dim)
70:         return output, (h, c)
71:
72:     def init_hidden(self, image_vec):
73:         image_vec = image_vec.mean(dim=1)
74:         h = self.lin_h(image_vec)
75:         c = self.lin_c(image_vec)
76:         return h, c
77:
78:     def fine_tune(self):
79:         self.conv_model.fine_tune()
80:
```

```
    1: import torch
    2: import numpy as np
    3:
    4: from transformers.bleu import compute_bleu
    5: from train_helper import predict, translate
    6:
    7: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    8:
    9:
   10: def test(net, batch_gen, top_k, **kwargs):
   11:     net.eval()
   12:     net.to(device)
   13:
   14:     batch_size = batch_gen.batch_size
   15:     seq_length = kwargs['seq_len']
   16:
   17:     translate_caps = []
   18:     referance_caps = []
   19:     for idx, (im, x_cap, y_cap) in enumerate(batch_gen.generate('test')):
   20:
   21:         print('\rtest:{}'.format(idx), flush=True, end='')
   22:
   23:         im, x_cap, y_cap = im.to(device), x_cap.to(device), y_cap.to(device)
   24:
   25:         word_int = 1  # x_START_
   26:         for i in range(batch_size):
   27:             caption = []
   28:             h = None
   29:             for ii in range(seq_length):
   30:                 word_int, h = predict(net, im[i, :], word_int, h, top_k=top_k)
   31:                 caption.append(word_int)
   32:             caption = translate(caption, net.embed_layer.int2word)
   33:             translate_caps.append(caption.split())
   34:
   35:         for i in range(batch_size):
   36:             y_trimed = trim_empty_rows(y_cap[i].cpu().numpy())
   37:             y_str = [translate(y_trimed[ii], net.embed_layer.int2word).split() for ii i
n range(y_trimed.shape[0])]
   38:             referance_caps.append(y_str)
   39:
   40:         print('\n')
   41:         for i in range(1, 5):
   42:             bleu, geo_mean, bp = compute_bleu(referance_caps, translate_caps, max_order
=i)
   43:             print('BLEU: {}, Geometric_mean: {}, BP:{}'.format(bleu, geo_mean, bp))
   44:
   45:     print('\n')
   46:     for i in range(1, 5):
   47:         bleu, geo_mean, bp = compute_bleu(referance_caps, translate_caps, max_order=i)
   48:         print('BLEU: {}, Geometric_mean: {}, BP:{}'.format(bleu, geo_mean, bp))
   49:
   50:
   51: def trim_empty_rows(y_cap):
   52:     return y_cap[~np.all(y_cap == 0, axis=1)]
   53:
```

```
 1: import numpy as np
 2:
 3: from torch.utils.data import Dataset
 4: from PIL import Image
 5: from PIL import ImageFile
 6:
 7: ImageFile.LOAD_TRUNCATED_IMAGES = True
 8:
 9:
10: class ImageDataset(Dataset):
11:     def __init__(self, image_path_names, captions_int,
12:                  im_addr, transformer, return_all=False):
13:         self.image_path_names = image_path_names
14:         self.captions_int = captions_int
15:         self.im_addr = im_addr
16:         self.transformer = transformer
17:         self.return_all = return_all
18:
19:     def __len__(self):
20:         return len(self.image_path_names)
21:
22:     def __getitem__(self, idx):
23:
24:         im_path = self.image_path_names[idx]
25:         image_name = im_path.split('/')[-1]
26:         image_id = int(image_name.split('.')[0]) + 1
27:
28:         caption_idx = self.im_addr[self.im_addr['im_addr'] == image_id].index
29:         if self.return_all:
30:             # not all images have 5 captions
31:             train_captions = np.zeros((6, 17))
32:             select_cap = self.captions_int.iloc[caption_idx].values
33:             train_captions[:select_cap.shape[0], :select_cap.shape[1]] = select_cap
34:         else:
35:             caption_idx = np.random.choice(caption_idx.values)
36:             train_captions = self.captions_int.iloc[caption_idx].values
37:
38:         # Target captions are one step forward of train captions
39:         target_captions = np.roll(train_captions, -1)
40:         if self.return_all:
41:             target_captions[:, -1] = 0
42:         else:
43:             target_captions[-1] = 0
44:
45:         image = Image.open(im_path)
46:         image = image.convert('RGB')
47:         if self.transformer is not None:
48:             image = self.transformer(image)
49:
50:         return image, train_captions, target_captions
```

```
# Copyright 2017 Google Inc. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================

"""Python implementation of BLEU and smooth-BLEU.
This module provides a Python implementation of BLEU and smooth-BLEU.
Smooth BLEU is computed following the method outlined in the paper:
Chin-Yew Lin, Franz Josef Och. ORANGE: a method for evaluating automatic
evaluation metrics for machine translation. COLING 2004.
"""

import collections
import math


def _get_ngrams(segment, max_order):
    """Extracts all n-grams upto a given maximum order from an input segment.
    Args:
      segment: text segment from which n-grams will be extracted.
      max_order: maximum length in tokens of the n-grams returned by this
          methods.
    Returns:
      The Counter containing all n-grams upto max_order in segment
      with a count of how many times each n-gram occurred.
    """
    ngram_counts = collections.Counter()
    for order in range(1, max_order + 1):
      for i in range(0, len(segment) - order + 1):
        ngram = tuple(segment[i:i+order])
        ngram_counts[ngram] += 1
    return ngram_counts


def compute_bleu(reference_corpus, translation_corpus, max_order=4,
                 smooth=False):
    """Computes BLEU score of translated segments against one or more references.
    Args:
      reference_corpus: list of lists of references for each translation. Each
          reference should be tokenized into a list of tokens.
      translation_corpus: list of translations to score. Each translation
          should be tokenized into a list of tokens.
      max_order: Maximum n-gram order to use when computing BLEU score.
      smooth: Whether or not to apply Lin et al. 2004 smoothing.
    Returns:
      3-Tuple with the BLEU score, n-gram precisions, geometric mean of n-gram
      precisions and brevity penalty.
    """
    matches_by_order = [0] * max_order
    possible_matches_by_order = [0] * max_order
    reference_length = 0
    translation_length = 0
    for (references, translation) in zip(reference_corpus,
```

```
                                        translation_corpus):
        reference_length += min(len(r) for r in references)
        translation_length += len(translation)

        merged_ref_ngram_counts = collections.Counter()
        for reference in references:
            merged_ref_ngram_counts |= _get_ngrams(reference, max_order)
        translation_ngram_counts = _get_ngrams(translation, max_order)
        overlap = translation_ngram_counts & merged_ref_ngram_counts
        for ngram in overlap:
            matches_by_order[len(ngram)-1] += overlap[ngram]
        for order in range(1, max_order+1):
            possible_matches = len(translation) - order + 1
            if possible_matches > 0:
                possible_matches_by_order[order-1] += possible_matches

    precisions = [0] * max_order
    for i in range(0, max_order):
        if smooth:
            precisions[i] = ((matches_by_order[i] + 1.) /
                             (possible_matches_by_order[i] + 1.))
        else:
            if possible_matches_by_order[i] > 0:
                precisions[i] = (float(matches_by_order[i]) /
                                 possible_matches_by_order[i])
            else:
                precisions[i] = 0.0

    if min(precisions) > 0:
        p_log_sum = sum((1. / max_order) * math.log(p) for p in precisions)
        geo_mean = math.exp(p_log_sum)
    else:
        geo_mean = 0

    ratio = float(translation_length) / reference_length

    if ratio > 1.0:
        bp = 1.
    else:
        bp = math.exp(1 - 1. / ratio)

    bleu = geo_mean * bp

    return bleu, geo_mean, bp
```

```
    1: import os
    2: import pickle
    3: import numpy as np
    4: from gensim.models import KeyedVectors
    5:
    6:
    7: class Word2VecTransformer:
    8:     def __init__(self):
    9:         self.model_path = 'embedding/word2vec.pkl'
   10:         self.vector_path = 'embedding/word2vec.vec'
   11:
   12:         if os.path.isfile(self.model_path):
   13:             print('loading pretrained embeddings from pickle')
   14:             model_file = open(self.model_path, 'rb')
   15:             self.model = pickle.load(model_file)
   16:         else:
   17:             self.model = KeyedVectors.load_word2vec_format(self.vector_path, binary=Fal
se, unicode_errors='replace')
   18:             model_file = open(self.model_path, 'wb')
   19:             pickle.dump(self.model, model_file)
   20:
   21:         self.vector_size = self.model.vector_size
   22:
   23:     def transform(self, x):
   24:         if x in self.model.wv:
   25:             r = self.model.wv[x]
   26:         elif x.lower() in self.model.wv:
   27:             r = self.model.wv[x.lower()]
   28:         else:
   29:             r = self.create_random_vec()
   30:         return r
   31:
   32:     def create_random_vec(self):
   33:         out_of_vocab_vector = np.random.rand(1, self.vector_size)[0]
   34:         out_of_vocab_vector = out_of_vocab_vector - np.linalg.norm(out_of_vocab_vector)
   35:         return out_of_vocab_vector
```