

An Implementation of “Show and Tell: A neural Image Caption Generator”

-- Shizhao Wang, Chao Yu, Xinyuan Chen, Chutian Tai, Mingyu Zha

1. Introduction

The purpose of the project is to implement the network structure described in the Paper “Show and Tell: A neural Image Caption Generator” where the author proposed to use the state-of-the-art technology in computer vision and natural language processing field related to deep learning to solve the problem of automatically describing the content in the image.

1.1 Original Paper Description

The paper “Show and Tell” proposes using the encoder-decoder structures like the typical machine translation or seq2seq problem. The encoder here is a deep convolutional neural network(CNN) and the decoder here is a Recurrent Neural Network(RNN) which uses LSTM more specifically. Similar to machine translation, the encoder CNN is responsible for extracting the hidden information from the images and the hidden information is encoded as a vector as the input to the decoder LSTM. The LSTM is responsible to generate the sentences given the input vector. The structure are shown in Figure 1.

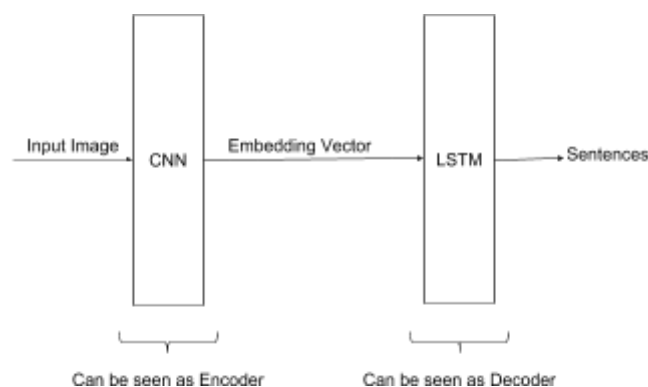


Figure 1. The basic structure of the model

It is very common in machine translation field to use another LSTM to perform as a encoder to get a embedding vector which represents the hidden information but here the paper chooses to use the CNN to extract hidden information from images to get a embedding vector. Then the paper suggests training the model and testing it on Flickr8K, Flickr30K and MSCOCO dataset and got very good results.

2. Details of the Implementation

2.1 Basic Information

Our group use PyTorch as the deep learning framework to implement the image caption generator. We train our network on BlueWaters and Google Cloud Platform.

2.2 Data preprocessing

In the training set, we need to convert the sentences tokens to a sequence of numbers where each number represents a word or a token. So as the code provided in HW7, we first use the NLTK toolkit

to tokenize all the sentences in MSCOCO captions, then we use all the tokens to build the vocabulary. Similarly, we sort the tokens by the frequency of the tokens. Then we set the vocabulary size as the numbers of tokens which appear more than 5 times. And then we build a map from the tokens to numbers. The number which is used to represent the token is actually the index of the token when all of the tokens are sorted by their frequency. After determining the vocabulary size, we store map for future use. And similar to machine translation, we have to introduce the <start> and <end> symbol at the beginning and the end of each sentence. We also introduce the <unknown> symbol for the words that is out of the vocabulary, we also assign relative numbers to these three symbols. Suppose we have a sequence of numbers of [4,5,10,12,23] and we use 1 and 2 to represent the start and end symbol, we will have a sequence of numbers of [1,4,5,10,12,23,2].

2.3 Padding

To make it possible for PyTorch to handle variable-length mini batches sequence, we have to pad some of the sequences to make sure that the sequences in one batch have the same length. Besides the start, end and unknown label, we will also introduce the padding symbol to pad the sequence to make them at the same length. First of all, we will select the longest sequence. Suppose the length of the longest sequence in this batch is N . And for each of the other sequences S_1, S_2, S_3, S_K with the length N_1, N_2, N_3, N_K . We will add pad symbols at the end of the each sequence. For sequence S_K , we will add $(N - N_K)$ padding symbol so that it has the same length with the longest sequence.

[1,4,10,11,21,23,42,20,2]	[1,4,10,11,21,23,42,20,2]
[1,4,10,12,20,2]	[1,4,10,12,20,2, 3, 3, 3]
[1,4,,42,20,2]	[1,4,42,20,2, 3, 3, 3, 3]
[1,42,20,2]	[1,42,20,2,3, 3, 3, 3, 3]
Before Padding	After Padding(3 as the pad symbol)

Figure 2. Padding Effect

To be more intuitive, this is the effect after we introduce the start, end and pad symbol:

I, love, Deep, Learning	<start>, I, love, Deep, Learning, so, much, <end>
I, love, Deep, so, much	<start>, I, love, Deep, Learning, <end>, <pad>, <pad>
I, love, study	<start>, I, love, study, <end>, <pad>, <pad>, <pad>
Before introducing symbols	After introducing symbol

Figure 3. effect of adding start, end and pad symbol

2.4 Training Phase

The forward pass structure for the CNN part is relatively simple, we use the network structure that is widely used for imageNet contest -- resnet50, and we change the last layer of the resnet50 so it could output the vector we wanted. The structure of the LSTM in forward pass is shown in figure 4.

After the image is fed into the CNN, we will get a Image embedding vector with size (1, embedding_length), then we will put the image vector into the LSTM and get the hidden state of the LSTM and predicted output at timestamp 0, and then as the Figure 3. shows, we will feed the caption sequence of this exact image to the same LSTM together with the hidden state we got after image input, then we will get the predicted output at timestamp 1 to timestamp N . Then we could use the output from timestamp 1 to timestamp N to calculate the loss.

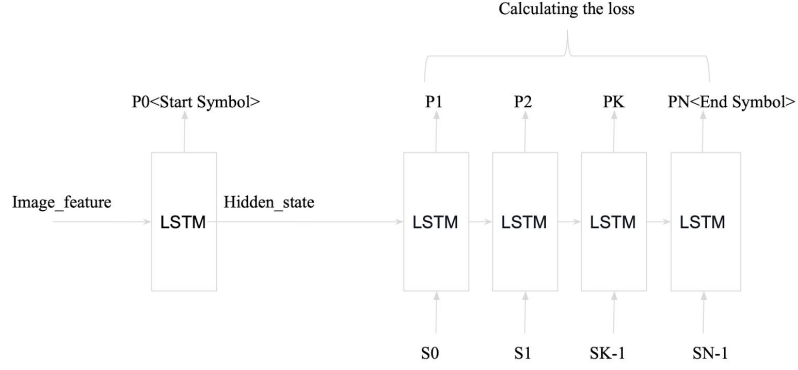


Figure 4. Training phase(RNN)

2.5 Loss Function

The loss function we use is to simply calculate the sum of the negative log probability of the correct word. We use p_t to denote the word probability distribution at timestamp t , so the probability of the correct word in time t is $p_t(S_t)$, suppose the S_t is the index of the correct word. Then we will get the loss sum which is $\sum_{t=1}^N \log p_t(S_t)$. One thing to mention is that since some of the sequences will be padded to be able to perform the batch processing, the loss of the padding token will not be calculated.

2.6 Inference phase

Figure 5. shows the process of how we use trained models to predict the caption. Similar to the training phase, we first feed the images to the CNN and get the image embedding vectors. Then we do the same thing like the training phase except that for the input of the LSTM in time t , we don't have the correct input token at time t since we will not have the correct caption in the inference phase. So we will use the predicted output at time $t-1$ as the input of the time t . Then we will have the predicted output from timestamp 1 to N which is the predicted sentence with a end symbol. We could use this to compute the score.

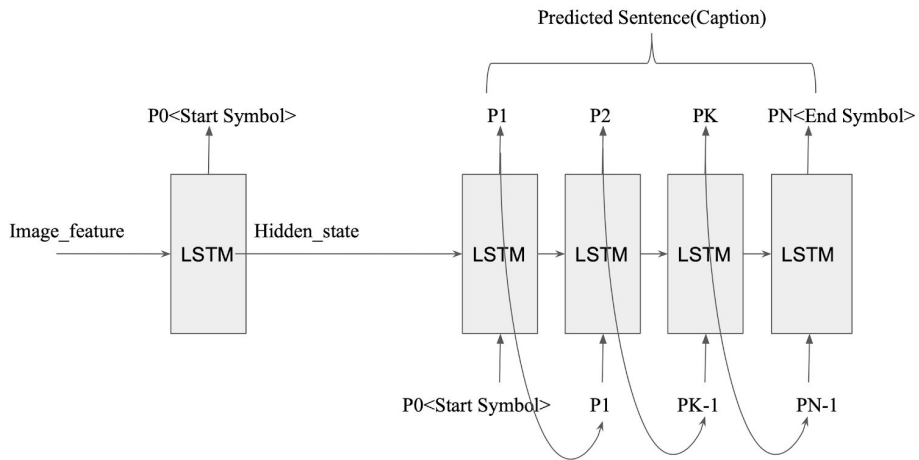


Figure 5. Inference phase

3. Results of our implementation

3.1. The choice of training the whole CNN or the last layer

In original paper, the author mentioned that only training the last layer of CNN and whole LSTM will make model have better generalization ability. So we perform the experiment of training the whole CNN or just the last layer of the CNN. The loss plot on training set is shown in Figure 6, the batch size is 128 and we record the loss every 500 batches(which is nearly 6 times per epoch).

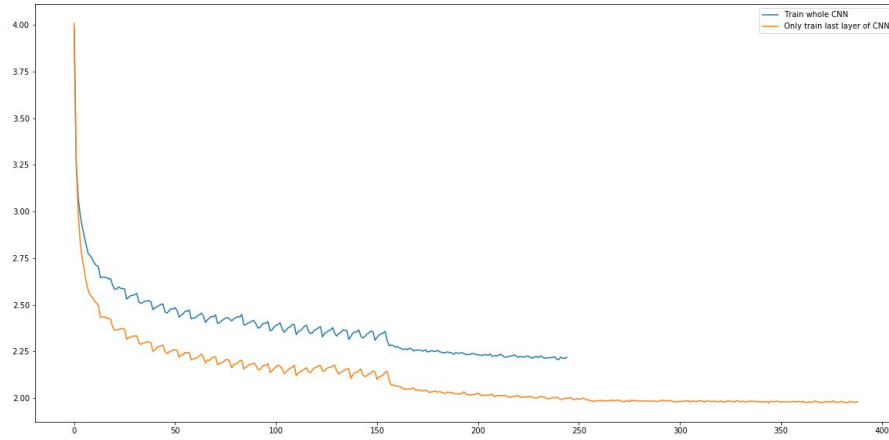


Figure 6. The loss plot of whether train the whole CNN

As the Figure 6. shows, only training the last layer of the CNN could help the model converge faster and have smaller loss on the training set. And given that training the whole network for one epoch would cost more than six times longer than only training the last layer of the CNN. So in the later experiments, we all choose to train the last layer of the CNN.

3.2. The choice of the CNN network

Figure 7. is the loss plot we got from different network, the batch size is 128 and we record the running loss every 500 batches(which is nearly 6 times per epoch), and we performing test on validation set every 5 epochs.

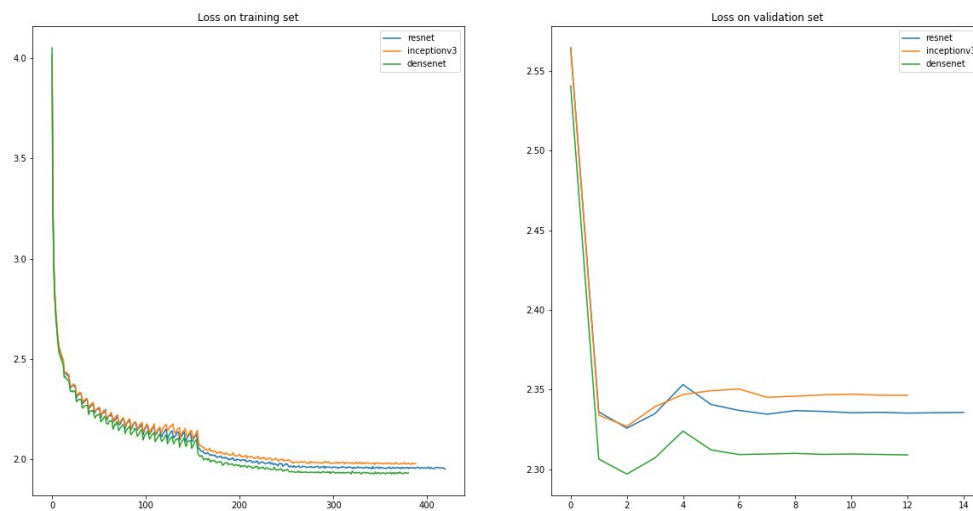


Figure 7. The loss with different network

It seems that we got the best result on densenet which is also similar to the results on latest image net score. And we also compute the score for three network structures shown in Table 1. All of these

three network are trained with vocabulary size of 8000 and embedding feature size of 1000. All the score in this report are computed exactly the same way with the original paper, we select 4000 images in validation set to run the score on.

Table 1, BLUE, Cider score with different networks.

	BLUE-1	BLUE-2	BLUE-3	BLUE-4	METEOR	CIDEr
Resnet	54.7	34.8	22.7	14.0	20.1	32.1
Inception_v3	58.6	39.8	26.2	17.5	19.4	42.3
Densenet	60.5	44.1	30.9	21.7	20.9	52.7

From the table we found that actually the inception v3 which has the largest loss during the training turns out to have the better score performance than resnet and Densenet still outperforms the other two like it actually did in testing and training loss.

3.3. The choice of hyper parameter

3.3.1 The choice of the vocabulary size.

We run two models on different vocabulary size which is 8000 and 12000, where 8000 is the number of tokens whose frequencies are greater than 5 and 12000 is nearly the half of the number of all tokens. We want to see whether the size of the vocabulary would affect the performance of the generator. It seems that these two different vocabulary size achieves nearly the same performance on testing and training set. It seems that the vocabulary size does not have much influence on the performance. But the scores we got from the model trained on vocabulary size = 12000 is higher than those of vocabulary size = 8000, this might because the larger the vocabulary size, the higher possibility that the predicted word is matched to the exact word since some of the rare words can not be generated when model is trained on a smaller vocabulary.

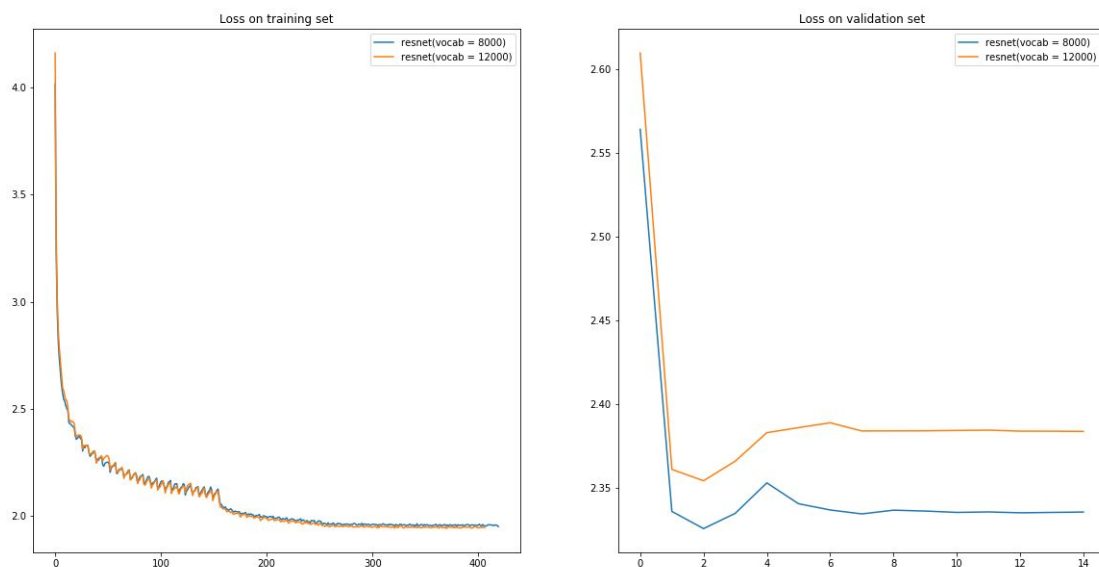


Figure 8. The loss with different vocabulary size

Table 2, BLUE, CIDEr score with different vocabulary size.

	BLUE-1	BLUE-2	BLUE-3	BLUE-4	METEOR	CIDEr
Resnet(vocab = 8000)	54.7	34.8	22.7	14.0	20.1	32.1
Resnet(vocab = 12000)	57.9	38.9	25.8	17.4	19.4	35.9

3.3.2 The choice of the embedding vector size

What is shown below is the experiments we try on different embedding vector sizes. Figure 8. shows the training and validation loss with embedding size of 512, 1000 and 2000. Table 3. shows the ranking scores of corresponding embedding sizes.

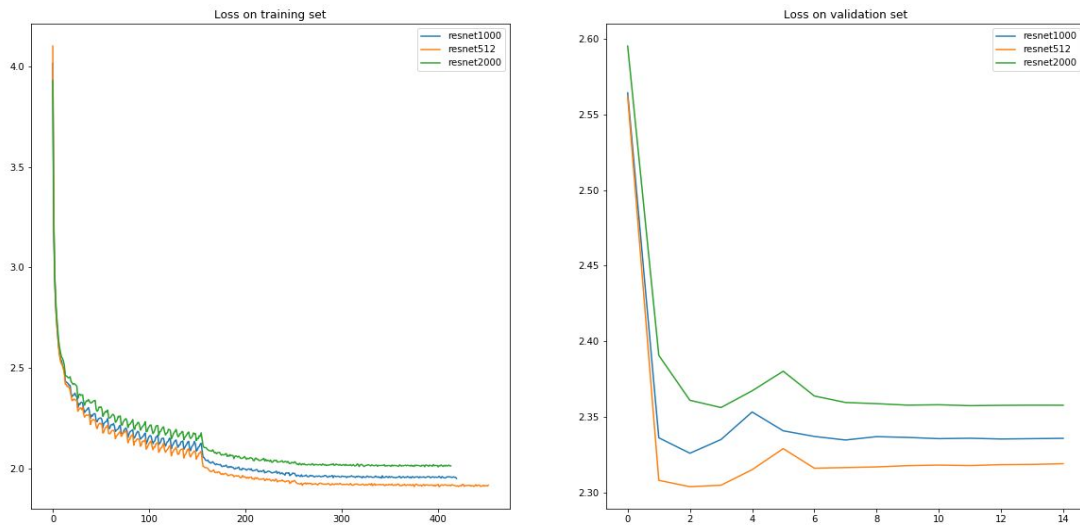


Figure 8. The loss of different embedding vector length

Table 3, BLUE, Cider score with different embedding network length.

	BLUE-1	BLUE-2	BLUE-3	BLUE-4	METEOR	CIDEr
Resnet(embed_size=1000)	54.7	34.8	22.7	14.0	20.1	32.1
Resnet(embed_size=512)	57.2	39.0	26.0	17.3	20.2	38.5
Resnet(embed_size=2000)	53.9	34.2	21.2	13.2	19.2	30.0

It seems that with more embedding features to describe the hidden information of the image will make the model more overfitting. The feature length with 512 achieve the best performance.

3.3.3 The choice of the number of hidden unit

We also perform experiment on whether the number of hidden units would affect the performance of the model. So we choose the network structure as resnet and 500 and 1000 as the number of hidden units in LSTM respectively to perform the training and testing. Figure 9. and Table 4 shows the result. From the figure we can tell that more hidden unit could help to better learn the training set which can be reflected from the smaller loss on training set, but the score and loss on validation set didn't improve too much. It shows that more hidden units could also cause overfitting. As the Figure 9.

shows, the loss on validation set of the model with 1000 hidden units go up after hitting on a local minimum which is an apparent sign of overfitting.

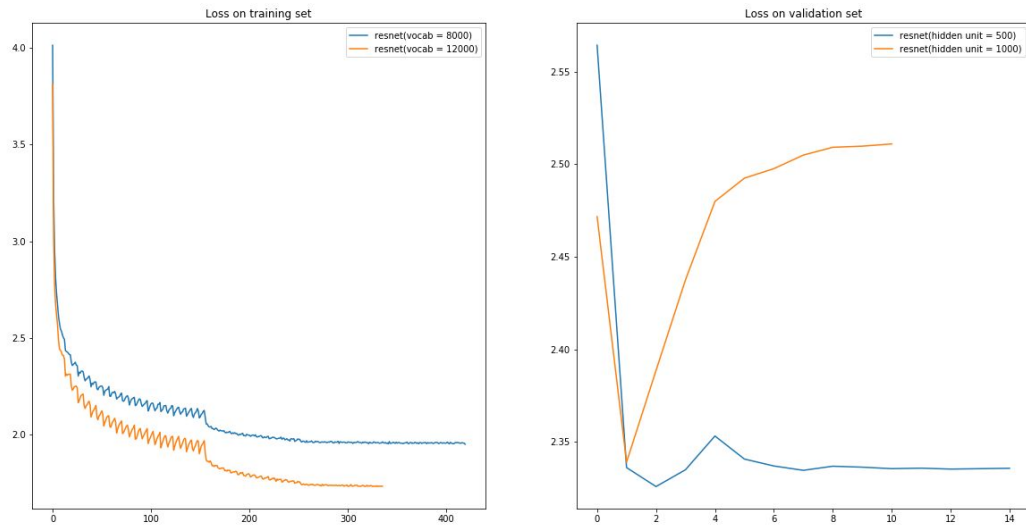


Figure 9. The loss of different number of hidden units in LSTM

Table 4, BLUE, Cider score with different number of hidden units in LSTM.

	BLUE-1	BLUE-2	BLUE-3	BLUE-4	METEOR	CIDEr
Resnet(500 hidden units)	54.7	34.8	22.7	14.0	20.1	32.1
Resnet(1000 hidden units)	55.1	35.8	22.9	14.7	19.0	31.3

3.4. Example Captions generated by our model

Similar to the original paper, we also selected several example captions generated by our model from model which has good examples to very ridiculous examples. Table 5 shows the results generated by sampling method with different quality. The good captions can accurately describe the main objects and the surrounding environment to some degree while the 'somewhere in between' captions can only identify the main objects but fail to describe the surroundings. In addition, the bad captions fail to give a reasonable overall description for the given images. Table 6 shows the results generated via beam search method for a single images with five best captions listed. The captions are very similar to each other and all have the similar meaning but different grammars. We can conclude from the images that if there is a dominant object in the image, the model could well recognized the image and then try to make a sentence based on that object, and when there several objects in the images, the model tend to perform badly.

Table 5. Captions generated by sampling method with different quality.








Good	Bad	Somewhere in between
 <p>a giraffe standing in a grassy field</p>	 <p>black and white photo of a man riding a bike</p>	 <p>a clock on the side of a wall</p>
 <p>an airplane is flying through the air</p>	 <p>flowers sitting in a glass vase</p>	 <p>a cat sitting on a toilet seat</p>

Table 6. Captions generated via beam search method for 1 picture

Image	Captions
	motorcycles are parked in a row
	rows of motorcycles parked in a lot
	motorcycles are parked along the street
	rows of motorcycles parked in a parking lot
	group of motorcycles parked in a parking lot

3.5 Ranking scores compared with original paper

In our project, we experimented three different approaches to evaluate the performance of our trained model (trained for 350 steps, around 20 computational hours). Since the test set of the MSCOCO is not provided with the annotations and online submission have maximum submission limit. For simplicity, we generate our captions on validation set and evaluate the score also on the evaluation set since it is well annotated. Here (Table 7.) are the best results we got of all our trained models with different hyperparameters and different network structures compared to the results from the paper. And we found that the results we extracted from the paper are also from the development set performed by authors, so it is still worthwhile to compare these two results.

Table 7, Ranking scores compared with original paper

Approach	Our Model	Result From Paper
BLEU_4	21.7%	27.7%
METEOR	20.9%	23.7%
CIDER	52.7%	85.5%

3.6 Results on transfer learning

To check whether the model has the ability to generalize, we also test the model trained on MSCOCO dataset on Flickr30K dataset. Here (Table 8) are the results:

Table 8. Score on different dataset.

	BLUE-1	BLUE-2	BLUE-3	BLUE-4	METEOR	CIDEr
Flickr30K	50.9	30.6	17.2	9.9	13.9	12.7
Flickr8K	46.7	27.3	14.0	6.0	15.8	12.4

As you can find, the results are much worse on different dataset which is true since the Flickr image has different patterns and the collections are done differently according to the original paper.

4. Total Computational Hours Consumed

We trained and tested our models mainly on Google Cloud Platform and BlueWaters. We mainly trained about 7-8 models for comparison parts on MSCOCO training set. It cost about 15-20 hours in average per model. So it cost about 160 hours training time on NVIDIA V100(whose speed is about 1.5-2 times faster than NVIDIA 1080Ti, and around 10 times faster than NVIDIA K20 on BlueWaters.) on Google Cloud Platform and about 60 hours on Bluewaters(before we switched it to Google Cloud Platform). We used up nearly three free accounts from Google Cloud Platform(300 dollars free credit on each account).

Reference:

- [1] Vinyals O , Toshev A , Bengio S , et al. Show and Tell: A Neural Image Caption Generator[J]. 2014.
- [2] Papineni, Kishore, et al. "Bleu: a Method for Automatic Evaluation of Machine Translation. " *Proc Meeting of the Association for Computational Linguistics* 2002.
- [3] William, F. (June 4) *Taming LSTMs: Variable-sized mini-batches and why PyTorch is good for your health*. Retrieved from: <https://towardsdatascience.com>
- [4] Sherlock Pytorch Data read. <https://zhuanlan.zhihu.com/p/30385675>
- [5] Pytorch: How to modify pre-trained models. https://blog.csdn.net/whut_ldz/article/details/78845947
- [6] How to deal with variable-length mini batches in RNN with pytorch. <https://zhuanlan.zhihu.com/p/34418001>