Image Caption Generator Based On Deep Neural Networks

Jianhui Chen
CPSC 503
CS Department

Wenqiang Dong
CPSC 503
CS Department

Minchen Li
CPSC 540
CS Department

Abstract

In this project, we systematically analyze a deep neural networks based image caption generation method. With an image as the input, the method can output an English sentence describing the content in the image. We analyze three components of the method: convolutional neural network (CNN), recurrent neural network (RNN) and sentence generation. By replacing the CNN part with three state-of-the-art architectures, we find the VGGNet performs best according to the BLEU score. We also propose a simplified version the Gated Recurrent Units (GRU) as a new recurrent layer, implementing by both MATLAB and C++ in Caffe. The simplified GRU achieves comparable result when it is compared with the long short-term memory (LSTM) method. But it has few parameters which saves memory and is faster in training. Finally, we generate multiple sentences using Beam Search. The experiments show that the modified method can generate captions comparable to the-state-of-the-art methods with less training memory.

1 Introduction

Automatically describing the content of images using natural languages is a fundamental and challenging task. It has great potential impact. For example, it could help visually impaired people better understand the content of images on the web. Also, it could provide more accurate and compact information of images/videos in scenarios such as image sharing in social network or video surveillance systems. This project accomplishes this task using deep neural networks. By learning knowledge from image and caption pairs, the method can generate image captions that are usually semantically descriptive and grammatically correct.

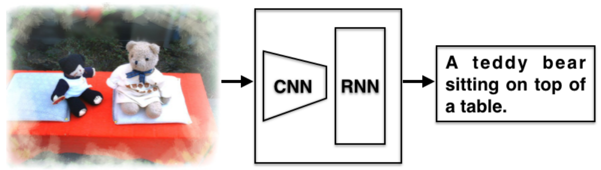


Figure 1: Image caption generation pipeline. The framework consists of a convulitional neural netwok (CNN) followed by a recurrent neural network (RNN). It generates an English sentence from an input image.

Human beings usually describe a scene using natural languages which are concise and compact. However, machine vision systems describes the scene by taking an image which is a two dimension arrays. From this perspective, Vinyal *et al.* (Vinyals et al.,) models the image captioning problem as a language translation problem in their Neural Image Caption (NIC) generator system. The idea is mapping the image and captions to the same space and learning a mapping from the image to the sentences. Donahue *et al.* (Donahue et al.,) proposed a more general Long-term Recurrent Convolutional Network (LRCN) method. The LRCN method not only models the one-to-many (words) image captioning, but also models many-to-one action generation and many-to-many video description. They also provides publicly available implementation based on Caffe framework (Jia et al., 2014), which further boosts the research on image captioning. This work is based on the LRCN method.

Although all the mappings are learned in an end-to-end framework, we believe the benefits of better understanding of the system by analyzing different components separately. Fig. 1 shows the pipeline. The model has three components. The first component is a CNN which is used to understand the content of the image. Image understanding answers the typical questions in computer vision such as "What are the objects?", "Where are the objects?" and "How are the objects interactive?". For example, the CNN has to recognize the "teddy bear", "table" and their relative locations in the image. The second component is a RNN which is used to generate a sentence given the visual feature. For example, the RNN has to generate a sequence of probabilities of words given two words "teddy bear, table". The third component is used to generate a sentence by exploring the combination of the probabilities. This component is less studied in the reference paper (Donahue et al.,).

This project aims at understanding the impact of different components of the LRCN method (Donahue et al.,).We have following contributions:

- understand the LRCN method at the implementation level.
- analyze the influence of the CNN component by replacing three CNN architectures (two from author's and one from our implementation).
- analyze the influence of the RNN component by replacing two RNN architectures. (one from author's and one from our implementation).
- analyze the influence of sentence generation method by comparing two methods (one from author's and one from our implementation).

2 Related work

Automatically describing the content of an image is a fundamental problem in artificial intelligence that connects computer vision and natural language processing. Earlier methods first generate annotations (i.e., nouns and adjectives) from images (Sermanet et al., 2013; Russakovsky et al., 2015), then generate a sentence from the annotations (Gupta and Mannem,). Donahue *et al.* (Donahue et al.,) developed a recurrent convolutional architecture suitable for large-scale visual learning, and demonstrated the value of the models on three different tasks: video recognition, image description and video description. In these models, long-term dependencies are incorporated into the network state updates and are end-to-end trainable. The limitation is the difficulty of understanding the intermediate result. The LRCN method is further developed to text generation from videos (Venugopalan et al.,).

Instead of one architecture for three tasks in LRCN, Vinyals *et al.* (Vinyals et al.,) proposed a neural image caption (NIC) model only for the image caption generation. Combining the GoogLeNet and single layer of LSTM, this model is trained to maximize the likelihood of the target description sentence given the training images. The performance of the model is evaluated qualitatively and quantitatively. This method was ranked first in the MS COCO Captioning Challenge (2015) in which the result was judged by humans. Comparing LRCN with NIC, we find three differences that may indicate the performance differences. First, NIC uses GoogLeNet while LRCN uses VGGNet. Second, NIC inputs visual feature only into the first unit of LSTM while LRCN inputs the visual feature into every LSTM unit. Third, NIC has simpler RNN architecture (single layer LSTM) than LRCN (two factored LSTM layers). We verified that the mathematical models of LRCN and NIC are exactly the same for image captioning. The performance difference lies in the implementation and LRCN has to trade off between simplicity and generality, as it is designed for three different tasks.

Instead of end-to-end learning, Fang *et al.* (Fang et al.,) presented a visual concepts based method. First, they used multiple instance learning to train visual detectors of words that commonly occur in captions such as nouns, verbs, and adjectives. Then, they trained a language model with a set of over 400,000 image descriptions to capture the statistics of word usage. Finally, they re-ranked caption candidates using sentence-level features and a deep multi-modal similarity model. Their captions have equal or better quality 34% of the time than those written by human beings. The limitation of the method is that it has more human controlled parameters which make the system less re-producible. We believe the web application **captionbot** (Microsoft,) is based on this method.

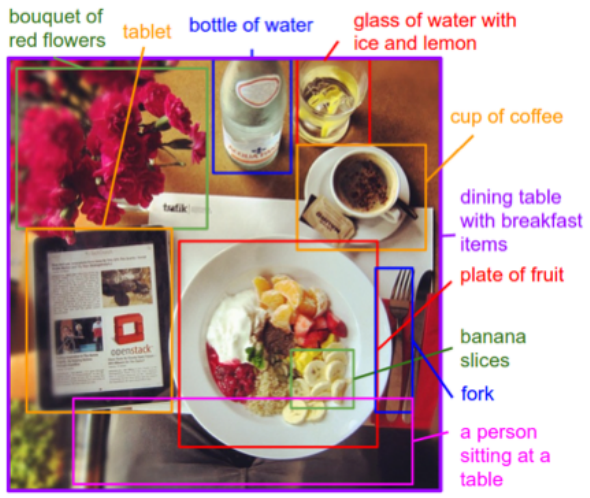

bouquet of red flowers
tablet
bottle of water
glass of water with ice and lemon
cup of coffee
dining table with breakfast items
plate of fruit
banana slices
fork
a person sitting at a table

Figure 2: The visual-semantic alignment method can generate descriptions of image regions. Figure from (Karpathy and Fei-Fei,).

Karpathy *et al.* (Karpathy and Fei-Fei,) proposed a visual-semantic alignment (VSA) method. The method generates descriptions of different regions of an image in the form of words or sentences (see Fig. 2). Technically, the method replaces the CNN with Region-based convolutional Networks (RCNN) so that the extracted visual features are aligned to particular regions of the image. The experiment shows that the generated descriptions significantly outperform retrieval baselines on both full images and on a new dataset of region-level annotations. This method generates more diverse and accurate descriptions than the whole image method such as LRCN and NIC. The limitation is that the method consists of two separate models. This method is further developed to dense captioning (Johnson et al., 2016) and image based question and answering system (Zhu et al., 2016).

3 Description of problem

Task In this project, we want to build a system that can generate an English sentence that describes objects, actions or events in an RGB image:

$$S = f(I) \tag{1}$$

where I is an RGB image and S is a sentence, f is the function that we want to learn.

Corpus We use the MS COCO Caption (Chen et al., 2015) as the corpus. The captions are gathered from human beings using Amazon's Mechanical Turk (AMT). We manually checked some examples by side-by-side comparing the image and corresponding sentences. We found the captions are very expressive and diverse. The COCO Caption is the largest image caption corpus at the time of writing. There are 413,915 captions for 82,783 images in training, 202,520 captions for 40,504 images in validation and 379,249 captions for 40,775 images in testing. Each image has at least 5 captions. The captions for training and validation are publicly available while the captions for testing is reserved by the authors. In the experiment, we use all the training data in the training process and 1,000 randomly selected validation data in the testing process.

4 Method

For image caption generation, LRCN maximizes the probability of the description giving the image:

$$\theta^* = \arg\max_{\theta} \sum_{(I,S)} log\ p(S|I;\theta) \tag{2}$$

where θ are the parameters of the model, I is an image, and S is a sample sentence. Let the length of the sentence be N, the method applies the chain rule to model the joint probability over $S_0, \cdots, S_N$:

$$log\ p(S|I) = \sum_{t=0}^{N} log\ p(S_t|I, S_0, \cdots, S_{t-1}) \tag{3}$$

where the θ is dropped for convenience, S_t is the word at step t.

The model has two parts. The first part is a CNN which maps the image to a fixed-length visual feature. The visual feature is embedded to as the input v to the RNN.

$$v = W_v(\mathbb{CNN}(I)) \tag{4}$$

where W_v is the visual feature embedding. The visual feature is fixed for each step of the RNN.

In the RNN, each word is represented a one-hot vector S_t of dimension equal to the size of the dictionary. S_0 and S_N are for special start and stop words. The word embedding parameter is W_s:

$$x_t = W_t S_t,\ t \in \{0 \cdots N-1\} \tag{5}$$

In this way, the image and words are mapped to the same space. After the internal processing of the RNN, the features v, x_t and internal hidden parameter h_t are decoded into a probability to predict the word at current time:

$$p_{t+1} = LSTM(v, x_t, h_t),\ t \in \{0 \cdots N-1\} \quad (6)$$

Because a sentence with higher probability does not necessary mean this sentence is more accurate than other candidate sentences, post-processing method such as **Beam Search** is used to generate more sentences and pick top-K sentences.

4.1 Convolutional neural network

In this project, a convolutional neural network (CNN) maps an RGB image to a visual feature vector. The CNN has three most-used layers: convolution, pooling and fully-connected layers. Also, Rectified Linear Units (ReLU) $f(x) = max(0, x)$ is used as the non-linear active function. The ReLU is faster than the traditional $f(x) = tanh(x)$ or $f(x) = (1 + e^{-x})^{-1}$. Dropout layer is used to prevent overfitting. The dropout sets the output of each hidden neuron to zero with a probability (i.e., 0.5). The "dropped out" neurons do not contribute to the forward pass and do not participate in back-propagation.

The AlexNet (Krizhevsky et al., 2012), VGGNet (Simonyan and Zisserman, 2014) and GoogLeNet (Szegedy et al., 2015) are three widely used deep convolutional neural network architecture. They share the convolution $\rightarrow$ pooling $\rightarrow$ fully-connection $\rightarrow$ loss function pipeline but with different shapes and connections of layers, especially the convolution layer. AlexNet is the first deep convolutional neural network used in large scale image classification. VGGNet and GoogLeNet achieves the-start-of-the-art performance in ImageNet recognition challenge 2014 and 2015.

When the CNN combines the RNN, there are specific considerations of convergence since both of them has millions parameters. For example, Vinyals *et al.* (Vinyals et al.,) found that it is better to fix the parameters of the convolutional layer as the parameters trained from the ImageNet. As a result, only the non-convolution layer parameters in CNN and the RNN parameters are actually learned from caption examples.

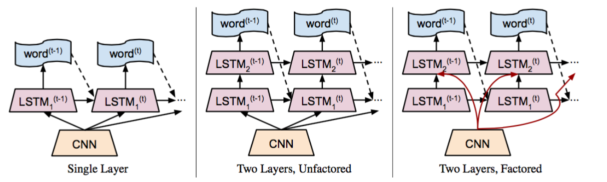


Figure 3: Three variations of the LRCN image captioning architecture. The most right two-layers factored LSTM is used in the method. Figure from (Donahue et al.,).

4.2 Recurrent neural network

To prevent the gradients vanishing problem, the long short-term memory (LSTM) method is used as the RNN component. A simplified LSTM updates for time step t given inputs x_t, h_{t-1}, and c_{t_1} are:

$$\begin{aligned}
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\
g_t &= \phi(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \phi(c_t)
\end{aligned} \quad (7)$$

where $\sigma(x) = (1+e^{-x})^{-1}$ and $\phi(x) = 2\sigma(2x) - 1$. In addition to a hidden unit $h_t \in R^N$, the LSTM includes an input gate $i_t \in R^N$, forget gate $f_t \in R^N$, output gate $o_t \in R^N$, input modulation gate $g_t \in R^N$, and memory cell $c_t \in R^N$. These additional cells enable the LSTM to learn extremely complex and long-term temporal dynamics. Additional depth can be added to LSTMs by stacking them on top of each other. Fig. 3 shows three version of LSTMs. The two-layers factored LSTM achieves the best performance and is used in the method.

In this project, we proposed a simplified version of GRU in section 5.1 which also avoids the vanishing gradient problem and can be easily implemented in Caffe based on the current Caffe LSTM framework. We also provide the MATLAB program in the Appendices verifying our derivation of BPTT on the original GRU model.

4.3 Sentence generation

The output of LSTM is the probability of each word in the vocabulary. **Beam search** is used to generate sentences. Beam search is a heuristic search algorithm that explores a graph by expanding the most promising node in a limited set. In addition to beam search, we also use k-best search to generate sentences. It is very similar to the time synchronous Viterbi search. The method iteratively selects the k best sentences from all the candidate sentences up to time t, and keeps only the resulting best k of them.

5 Implementation

Preprocessing Because we want to keep the architecture of the CNN, the input image are randomly cropped to the size of 224×224. As a result, only part of the images are used in training at particular iteration. Because one image will be cropped multiple times in the training, the CNN can probably see the whole image in the training (once for part of the image). However, the method only sees part of the image in the testing except the dense cropping is also used (our project does not use dense crop). For the sentences, the method first creates a vocabulary only from the training captions and removes lower frequency words (less than 5). Then, words are represented by one-hot vectors.

5.1 Caffe architecture

Caffe (Jia et al., 2014) provides a modifiable framework for the state-of-the-art deep learning algorithms. It is implemented using C++ and also provides Python and MATLAB interfaces. Caffe model (network) definitions are written as configuration files using the Protocol Buffer Language[1] so that the net representation and implementation are separated. The separation abstracts from memory underlying location in CPU or GPU so that switching between a CPU and GPU implementation is exactly by one function call. However, the separation makes the implementation less convenient as we will show in the next paragraph.

5.2 Simplify and implement GRU in Caffe

In Caffe, a **layer** is the fundamental unit of computation. A **blob** is a wrapper over the actual data providing synchronization capability between CPU and GPU. We tried to implement the Gated Recurrent Units (GRU) (Cho et al., 2014) in Caffe. The GRU updates for time step t given inputs x_t, s_{t-1} are:

$$\begin{aligned} z &= \sigma(U_z x_t + W_z s_{t-1} + b_z) \\ r &= \sigma(U_r x_t + W_r s_{t-1} + b_r) \\ h &= \tanh(U_h x_t + W_h(s_{t-1} \odot r) + b_h) \\ s_t &= (1 - z) \odot h + z \odot s_{t-1} \end{aligned} \tag{8}$$

where z is the update gate, r is the reset gate. s is used as both hidden states and cell states. With fewer parameters, GRU can reach a comparable performance to LSTM (Jozefowicz et al.,). To implement GRU, we first wrote a MATLAB program to check our BPTT[2] gradient derivation. This is due to the fact that automatic differentiation in Caffe is not supported at layer units level. Followed by our derivation, the calculated gradients only deviate from the numerical gradients by around 10^{-5} relatively. However, implementing GRU in Caffe is not straight forward since Caffe is based on a complicated software architecture trying to provide convenience for assembling, not further developing. This is the bottleneck of GRU implementation. We have tried a number of implementations based on the original GRU (Equ. 8), with no good results. Finally we simplified GRU model inspired by the simplified SLTM in (Donahue et al.,). We omit the reset gate and add a transfer gate to make it easily fit into the current Caffe LSTM framework as:

$$\begin{aligned} z &= \sigma(U_z x_t + W_z s_{t-1} + b_z) \\ h &= \tanh(U_h x_t + W_h s_{t-1} + b_h) \\ c_t &= (1 - z) \odot h + z \odot c_{t-1} \\ s_t &= c_t \end{aligned} \tag{9}$$

Note that the omitted reset gate won't bring back the vanishing gradient problem which we see in traditional RNN because we still have the update gate z acting as a weight between the previous state and the current processed input. The added transfer gate, c, seems to be less useful, but it is actually very important for calculating the gradient in the framework. The parameter gradients in an RNN within a single step, t, depends not only on $\partial L_t/\partial s_t$, but also $\partial L_t/\partial s_{t-i}$ where $i = 1, 2, ..., t$. In Caffe, $\partial L_t/\partial s_t$ is calculated by outer layers automatically, while $\partial L_t/\partial s_{t-i}$ need to be calculated by inside layer unit. To hold and transfer these two parts of gradients to

[1]`https://developers.google.com/protocol-buffers/docs/proto`

[2]Backward propagation through time

CNNs	layer	#param	memory	B-4
AlexNet	8	60	0.9	0.253
VGGNet	16	138	11.6	**0.294**
GoogLeNet	22	12	5.8	0.211

Table 1: Quantitative comparison of CNNs. The number of parameter (#param) is in the unit of million, and the training memory is in the unit of Gb. In experiment, we found that the BLEU 4 performance is positively related to the number of parameters.

Method	B-1	B-2	B-3	B-4
AlexNet + LSTM	0.650	0.467	0.324	0.221
AlexNet + GRU	0.623	0.433	0.292	0.194
VGGNet + LSTM	0.588	0.406	0.264	0.168
VGGNet + GRU	0.583	0.393	0.256	0.168

Table 2: AlexNet, VGGNet with different RNN models. Our GRU model achieves comparable result with the LSTM model, but with less parameter and training time. The beam size is 1.

the next time step, we use another intermediate variable, which is the added transfer gate c. This is just an engineering issue that might not be avoided while developing new models in Caffe. The theory is always clear and concise (see Appendices for the MATLAB program verifying our BPTT derivation to the original GRU).

5.3 Training method

The neural network is trained using the mini-patch stochastic gradient descent (SGD) method. The base learning rate is 0.01. The learning rate drops 50% in every 20,000 iterations. Because the number of training samples is much smaller than the number of parameters of the neural network, overfitting is our big concern. Besides the dropout layer, we fixed the parameters of the convolutional layers as suggested by (Vinyals et al.,). All the network are trained in a Linux machine with a Tesla K40c graphic card with 12Gb memory.

6 Results

6.1 Quantitative result

Evaluation metrics We use BLEU (Papineni et al., 2002) to measure the similarity of the captions generated by our method and human beings. BLEU is a popular machine translation metric that analyzes the co-occurrences of n-grams between the candidate and reference sentences. The unigram scores (B-1) account for the adequacy of the translation, while longer n-gram scores (B-2, B-3, B-4) account for the fluency.

Different CNNs Table 1 compares the performance of three CNN architectures (the RNN part use LSTM). The VGGNet achieves the best performance (BLEU 4) and GoogLeNet has the lowest score. It is out of our expectation at first because GoogLeNet achieves the best performance in the ImageNet classification task. We discussed this phenomenon with our fellows students. One of them pointed out that despite its slightly weaker classification performance, the VGGNet features outperform those of GoogLeNet in multiple transfer learning tasks (Karpathy, 2015). A downside of the VGGNet is that it is more expensive to evaluate and it uses a lot more memory (11.6 Gb) and parameters (138 million). It takes more time to train VGGNet and GoogleNet than AlexNet (about 8 hours vs 4 hours).

Different RNNs Table 2 compares the performance of LSTM and GRU. The GRU model achieves comparable results with less parameters and training time.

Different sentence generation methods Table 3 also analyze the impact of beam size in the **Beam Search** for different CNN architectures. In general, larger beam size achieves higher BLEU score. This phenomenon is much more obvious in the VGGNet than other two CNNs. When the beam size is 1, AlexNet outperforms VGGNet. When the beam size is 10, the VGGNet outperforms AlexNet. The most probable reason is that AlexNet is good at detecting a single or few objects in an image while VGGNet is good at detecting multiple objects in the same image. When the beam size becomes larger, the VGGNet based method can generate more accurate sentences.

# beam	B-1	B-2	B-3	B-4
	AlexNet			
1	**0.650**	0.467	0.324	0.221
5	**0.650**	0.467	0.343	0.247
10	0.644	**0.474**	**0.347**	**0.253**
	VGGNet			
1	0.588	0.406	0.264	0.168
5	0.632	0.450	0.310	0.212
10	<u>**0.681**</u>	<u>**0.513**</u>	<u>**0.390**</u>	<u>**0.294**</u>
	GoogLeNet			
1	0.533	0.353	0.222	0.139
5	0.568	0.385	0.262	0.180
10	**0.584**	**0.410**	**0.292**	**0.211**

Table 3: AlexNet, VGGNet and GoogleNet with different beam sizes. Using AlexNet, the impact of the number of beam size is not significant. Using the VGG net, the impact is significant. Using the GoogLeNet net, the impact is moderate. The best scores are highlighted.

Method	B-1	B-2	B-3	B-4
LRCN	0.669	0.489	0.349	0.249
NIC	N/A	N/A	N/A	0.277
VSA	0.584	0.410	0.292	0.211
This project	0.681	0.513	0.390	0.294

Table 4: Evaluation of image caption of different methods. LRCN is tested on the validation set (5,000 images). NIC is tested on the validation set (4,000 images). VSA is tested on the test set (40,775 images). This project is tested on the validation set (1,000 images for B-1, B-2, B-3, and 100 images for B-4).

Comparison with other systems Table 4 compares BLEU scores of the results from LRCN, NIC, VSA and this project. The BLEU score of the result of this project is comparable or better than those from other systems although our project is tested on less data set (1,000 images).

6.2 Qualitative result

Taking Fig. 4 as an example, we analyze the captions generated by AlexNet, VGGNet and GoogLeNet.

When beam size is 1, the captions are as follows,

- (AlexNet) A group of people sitting at a table with a pizza.
- (VGGNet) A man and woman sitting at a table with a pizza.
- (GoogLeNet) A group of people sitting at a dinner table.

When beam size is 5, the captions are as follows,

- (AlexNet) A group of people sitting at a table.
- (VGGNet) A man and woman sitting at a table with food.
- (GoogLeNet) A group of people sitting at a dinner table.

When beam size is 10, the captions are as follows,

- (AlexNet) A group of people sitting at a table.
- (VGGNet) A man and woman sitting at a table.
- (GoogLeNet) A group of people sitting at a dinner table.

From the result listed above, we can see that when the beam size is fixed, VGGNet can generate captions with more details. When the beam size increases, the captions become short and detailed information disappears.

Although the sentence generated by our method has the highest probability, we don't know if there are other sentences that can describe the image better. So we use 3-best search to explore the top 3 captions. For Fig. 4, the captions generated by GoogLeNet with beam size 5 using 3-best search are listed as follows,

- A group of people sitting at a dinner table.
- A group of people sitting around a dinner table.
- A group of people sitting at a dinner table with plates of food.

The above captions are listed in probability descending order. We can see that the third sentence is actually the best one, although it does not have the highest probability. This is because when the sentence is long, it is more probable to make mistakes. So, sentences with high probability sometimes tend to be short, which may miss some detailed information. However, it does not mean that the sentence with the highest probability is bad. In most cases we observed, sentences with the highest probability are good enough to describe an image while long sentences often include redundant information and often make grammatical mistakes.

Fig. 5 shows the good examples of the sentences generated by this project. Most of them successfully describe the main objects and events in images. Fig. 6 shows failed examples of the system. The errors

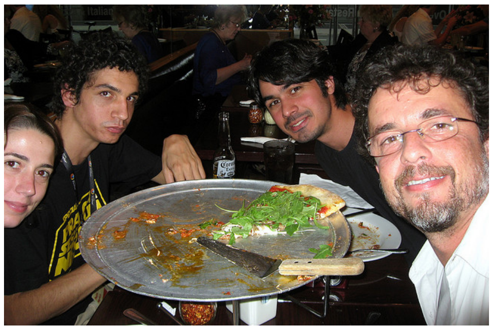


Figure 4: Sample image for qualitative analysis.

are mainly from object mis-detections such as an airplane is mis-detected as a kite (row 3 column 1), cellphones are detected as laptop (row 4 column 2). The generated sentences are also has minor grammar error. For example, "A motorcycle with a motorcycle" (row 4 column 3) is hard to understand.

7 Lessons learned and future work

This project provides a valuable learning experience. First, the LRCN method has a sophisticated pipeline so that modifying part of the pipeline is complicated than we expected. We learned how to use one of the most popular deep learning frameworks Caffe through the project.

Second, mathematics and the knowledge of particular software architecture are equally important for the success of the project. Although we implemented the MATLAB version of GRU very early before the deadline of the project, we spent a large amount of time on implementing the GRU layer in Caffe. The benefit is that we learned valuable first-hand experience on the developing level of Caffe instead of purely using existing layers in Caffe.

Third, working in a team, we could discuss and refine a lot of initial ideas. We could also anticipate problems that could become critical of the cases we were working alone. Table 5 roughly shows the work division among team members.

8 Evaluation

- The project is successful. We have finished all the goals before the deadline. The system can generate sentences that are semantically correct according to the image. We also proposed a simplified version of GRU that has less parameters and achieves comparable result with the LSTM method.
- The strength of the method is on its end-to-end learning framework. The weakness is that it requires large number of human labeled data which is very expensive in practice. Also, the current method still has considerable errors in both object detection and sentence generation.

Task	Wenqiang	Minchen	Jianhui
CNN			100%
GRU		70%	30%
Beam Search	100%		
Writing	40%	20%	40%

Table 5: Division of work. These only measure the implementation and experimenting workload. All the analyses and discussions are conducted by all of us.

9 Conclusion and future work

We analyzed and modified an image captioning method LRCN. To understand the method deeply, we decomposed the method to CNN, RNN, and sentence generation. For each part, we modified or replaced the component to see the influence on the final result. The modified method is evaluated on the COCO caption corpus. Experiment results show that: first the VGGNet outperforms the AlexNet and GoogLeNet in BLEU score measurement; second, the simplified GRU model achieves comparable results with more complicated LSTM model; third, increasing the beam size increase the BLEU score in general but does not necessarily increase the quality of the description which is judged by humans.

Future work In the future, we would like to explore methods to generate multiple sentences with different content. One possible way is to combine interesting region detection and image captioning. The VSA method (Karpathy and Fei-Fei,) gives a direction of our future work. Taking Fig. 2 as an example, we hope the output will be a short paragraph: "Jack has a wonderful breakfast in a Sunday morning. He is sitting at a table with a bouquet of red flowers. With his new iPad air on the left, he enjoys a plate of fruits and a cup of coffee." The short paragraph naturally describes the image content in a story-telling fashion which is more attractive to the human beings.

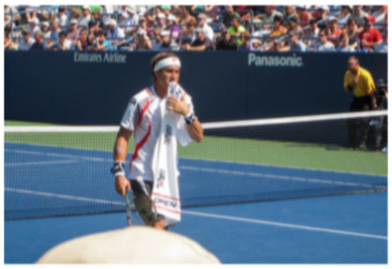
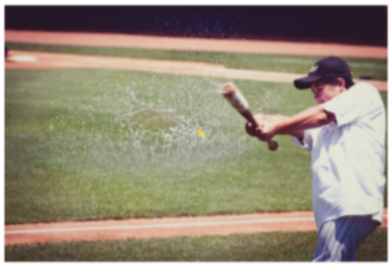
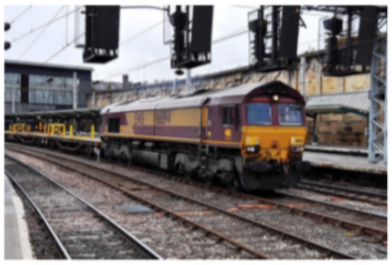
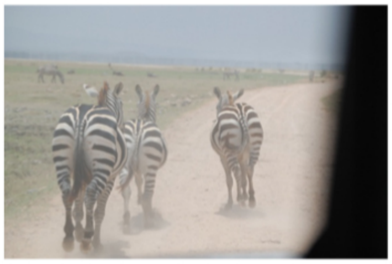
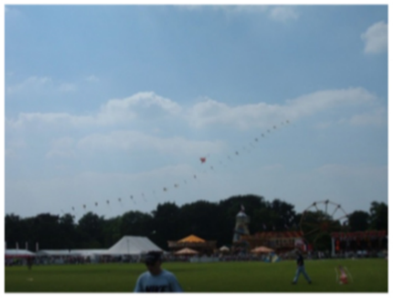
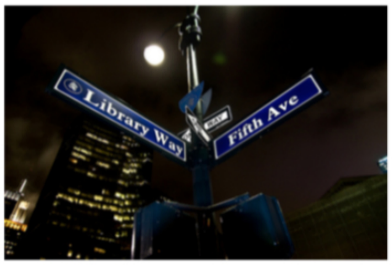
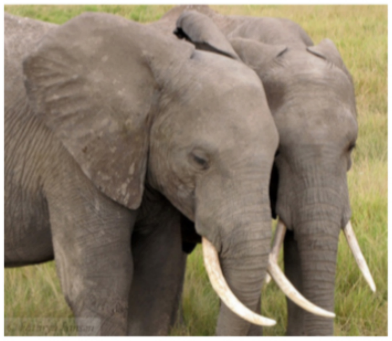
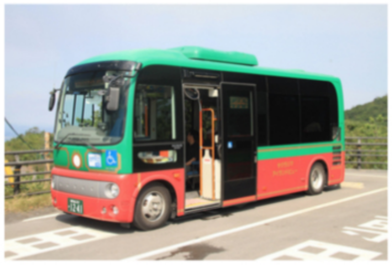
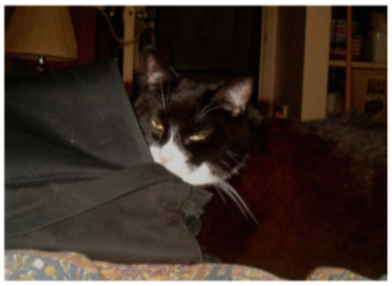
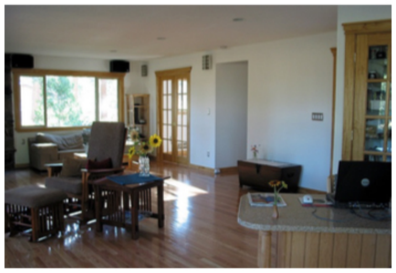
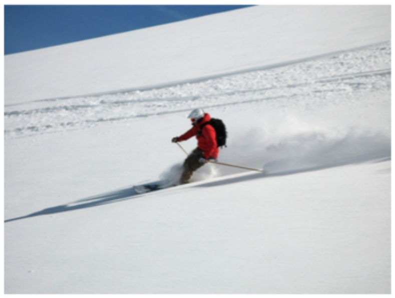
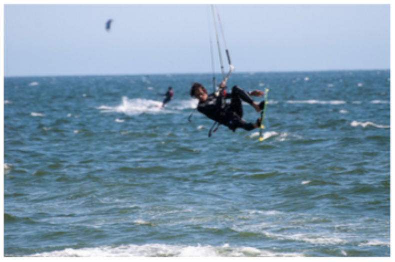
		
A tennis player in a tennis court with a racket.	A man in a baseball shirt is playing a game of baseball.	A train is traveling down a train track.
		
A group of zebras standing next to each other.	A large kite flying over a cloudy day.	A street sign on a street next to a building.
		
A large elephant with a large baby elephant in a grassy field.	A red and green bus parked on a street.	A cat is sitting on a couch.
		
A living room with a large window.	A man riding a snowboard on a snow covered slope.	A man riding a surfboard in the ocean.

Figure 5: A selection of evaluation results, when the method can generate accurate captions.

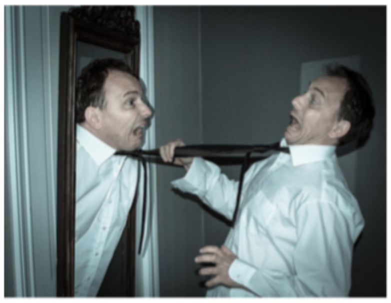
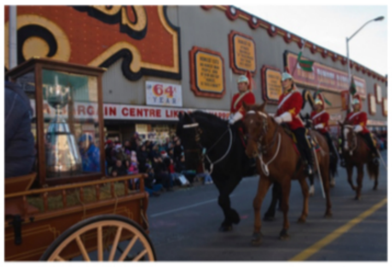
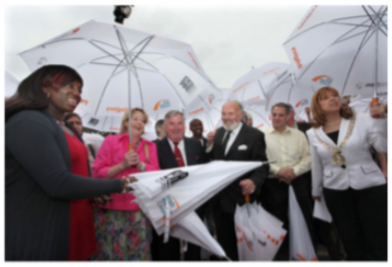
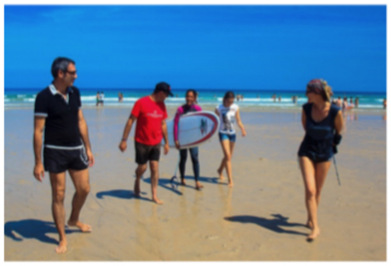
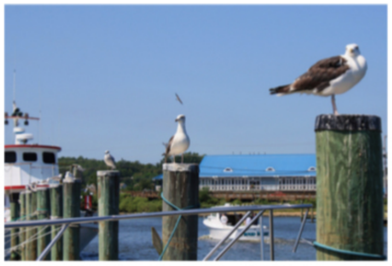
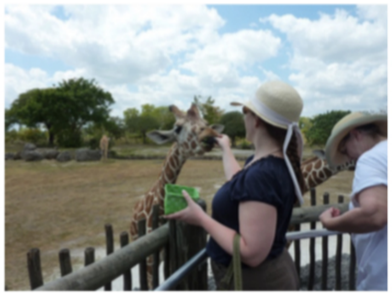
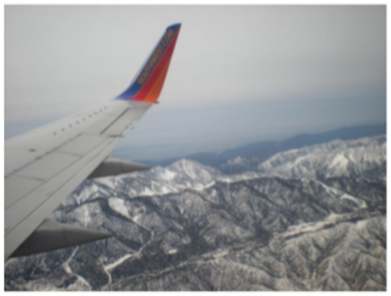
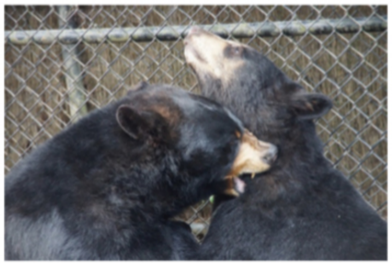
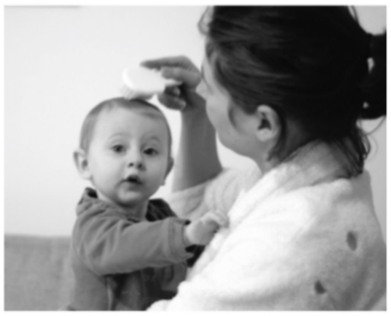
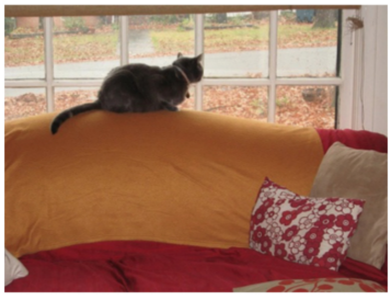
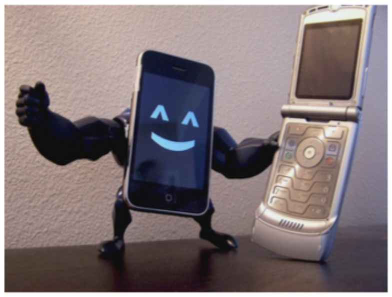
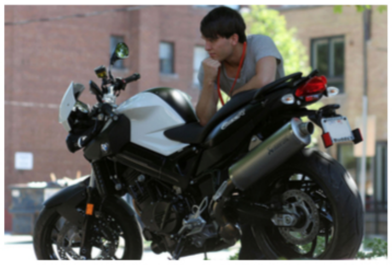
 A man in a suit and tie is holding a tie.	 A horse with a horse on top of a building.	 A group of people standing in a room with a picture of people.
 A man in a blue shirt is holding a frisbee.	 A large bird flying over a large building.	 A group of people standing next to a tree.
 A person flying a kite in the air.	 A black bear laying on top of a dirt field.	 A man with a hat and a tie.
 A black and white cat laying on a bed.	 A toilet sitting on a desk next to a laptop.	 A motorcycle with a motorcycle on the back of a motorcycle.

Figure 6: A selection of evaluation results, when the method cannot generate accurate captions.

References

[Chen et al.2015] Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollar, and C Lawrence Zitnick. 2015. Microsoft coco captions: Data collection and evaluation server. *arXiv preprint arXiv:1504.00325*.

[Cho et al.2014] Kyunghyun Cho, Bart Van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

[Donahue et al.] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *IEEE CVPR*.

[Fang et al.] Hao Fang, Saurabh Gupta, Forrest Iandola, Rupesh K Srivastava, Li Deng, Piotr Dollar, Jianfeng Gao, Xiaodong He, Margaret Mitchell, John C Platt, et al. From captions to visual concepts and back. In *IEEE CVPR*.

[Gupta and Mannem] Ankush Gupta and Prashanth Mannem. From image annotation to image description. In *Neural information processing*.

[Jia et al.2014] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678.

[Johnson et al.2016] Justin Johnson, Andrej Karpathy, and Li Fei-Fei. 2016. Densecap: Fully convolutional localization networks for dense captioning. In *IEEE CVPR*.

[Jozefowicz et al.] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *ICML*.

[Karpathy and Fei-Fei] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *IEEE CVPR*.

[Karpathy2015] Andrej Karpathy. 2015. Cs231n: Convolutional neural networks for visual recognition. `http://cs231n.github.io/`. [Online; accessed 11-April-2015].

[Krizhevsky et al.2012] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

[Microsoft] Microsoft. captionbot. `https://www.captionbot.ai/`. [Online; accessed 22-April-2015].

[Papineni et al.2002] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318.

[Russakovsky et al.2015] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *IJCV*, 115(3):211–252.

[Sermanet et al.2013] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. 2013. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*.

[Simonyan and Zisserman2014] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[Szegedy et al.2015] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *IEEE CVPR*.

[Venugopalan et al.] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence-video to text. In *IEEE ICCV*.

[Vinyals et al.] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *IEEE CVPR*.

[Zhu et al.2016] Yuke Zhu, Oliver Groth, Michael Bernstein, and Li Fei-Fei. 2016. Visual7W: Grounded Question Answering in Images. In *IEEE CVPR*.

Appendices

The MS COCO Caption corpus is in http://mscoco.org/.

Here is a list of code:

- GoogleNet with LSTM script.
- GRU MATLAB version.
- GRU layer Caffe implemenation.
- GRU unit layer Caffe implementation in C++
- GRU unit layer Caffe implementation in CUDA C (running on GPU)
- Sentence generation code.

```
  bottom: "inception_5b/pool_proj"
  top: "inception_5b/output"
}
layer {
  name: "pool5/7x7_s1"
  type: "Pooling"
  bottom: "inception_5b/output"
  top: "pool5/7x7_s1"
  pooling_param {
    pool: AVE
    kernel_size: 7
    stride: 1
  }
}
layer {
  name: "pool5/drop_7x7_s1"
  type: "Dropout"
  bottom: "pool5/7x7_s1"
  top: "pool5/7x7_s1"
  dropout_param {
    dropout_ratio: 0.4
  }
}
layer {
  name: "loss3/classifier"
  type: "InnerProduct"
  bottom: "pool5/7x7_s1"
  top: "loss3/classifier"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 1000
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}

# start LSTM
layer {
```

Figure 7: Page 51 in 55: connect the GoogLeNet to LSTM

```
    num_output: 1000
    weight_filler {
      type: "uniform"
      min: -0.08
      max: 0.08
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "lstm1"
  type: "LSTM"
  bottom: "embedded_input_sentence"
  bottom: "cont_sentence"
  top: "lstm1"
  include { stage: "factored" }
  recurrent_param {
    num_output: 1000
    weight_filler {
      type: "uniform"
      min: -0.08
      max: 0.08
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "lstm2"
  type: "LSTM"
  bottom: "lstm1"
  bottom: "cont_sentence"
  bottom: "loss3/classifier"
  top: "lstm2"
  include { stage: "factored" }
  recurrent_param {
    num_output: 1000
    weight_filler {
      type: "uniform"
      min: -0.08
      max: 0.08
    }
    bias_filler {
      type: "constant"
      value: 0
```

Figure 8: Page 53 in 55: connect two factored LSTM.

```
% This program tests the BPTT process we manually developed for GRU.
% We calculate the gradients of GRU parameters with chain rule, and then
% compare them to the numerical gradients to check whether our chain rule
% derivation is correct.

% Here, we provided 2 versions of BPTT, backward_direct() and backward().
% The former one is the direct idea to calculate gradient within each step
% and add them up (O(sentence_size^2) time). The latter one is optimized to
% calculate the contribution of each step to the overall gradient, which is
% only O(sentence_size) time.

% This is very helpful for people who wants to implement GRU in Caffe since
% Caffe didn't support auto-differentiation. This is also very helpful for
% the people who wants to know the details about Backpropagation Through
% Time algorithm in the Reccurent Neural Networks (such as GRU and LSTM)
% and also get a sense on how auto-differentiation is possible.

% NOTE: We didn't involve SGD training here. With SGD training, this
% program would become a complete implementation of GRU which can be
% trained with sequence data. However, since this is only a CPU serial
% Matlab version of GRU, applying it on large datasets will be dramatically
% slow.

% by Minchen Li, at The University of British Columbia. 2016-04-21

...

% Forward propagate calculate s, y_hat, loss and intermediate variables for each step
function [s, y_hat, L, z, r, c] = forward(x, y, ...
    U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0)
    % count sizes
    [vocabulary_size, sentence_size] = size(x);
    iMem_size = size(V, 2);

    % initialize results
    s = zeros(iMem_size, sentence_size);
    y_hat = zeros(vocabulary_size, sentence_size);
    L = zeros(sentence_size, 1);
    z = zeros(iMem_size, sentence_size);
    r = zeros(iMem_size, sentence_size);
    c = zeros(iMem_size, sentence_size);

    % calculate result for step 1 since s_0 is not in s
    z(:,1) = sigmoid(U_z*x(:,1) + W_z*s_0 + b_z);
    r(:,1) = sigmoid(U_r*x(:,1) + W_r*s_0 + b_r);
    c(:,1) = tanh(U_c*x(:,1) + W_c*(s_0.*r(:,1)) + b_c);
    s(:,1) = (1-z(:,1)).*c(:,1) + z(:,1).*s_0;
    y_hat(:,1) = softmax(V*s(:,1) + b_V);
    L(1) = sum(-y(:,1).*log(y_hat(:,1)));
    % calculate results for step 2 - sentence_size similarly
    for wordI = 2:sentence_size
        z(:,wordI) = sigmoid(U_z*x(:,wordI) + W_z*s(:,wordI-1) + b_z);
        r(:,wordI) = sigmoid(U_r*x(:,wordI) + W_r*s(:,wordI-1) + b_r);
        c(:,wordI) = tanh(U_c*x(:,wordI) + W_c*(s(:,wordI-1).*r(:,wordI)) + b_c);
        s(:,wordI) = (1-z(:,wordI)).*c(:,wordI) + z(:,wordI).*s(:,wordI-1);
        y_hat(:,wordI) = softmax(V*s(:,wordI) + b_V);
        L(wordI) = sum(-y(:,wordI).*log(y_hat(:,wordI)));
    end
end

% Backward propagate to calculate gradient using chain rule
% (O(sentence_size) time)
function [dV, db_V, dU_z, dU_r, dU_c, dW_z, dW_r, dW_c, db_z, db_r, db_c, ds_0] = ...
    backward(x, y, U_z, U_r, U_c, W_z, W_r, W_c, b_z, b_r, b_c, V, b_V, s_0)
    % forward propagate to get the intermediate and output results
    [s, y_hat, L, z, r, c] = forward(x, y, U_z, U_r, U_c, W_z, W_r, W_c, ...
        b_z, b_r, b_c, V, b_V, s_0);
    % count sentence size
    [~, sentence_size] = size(x);

    % calculate gradient using chain rule
    delta_y = y_hat - y;
    db_V = sum(delta_y, 2);

    dV = zeros(size(V));
    for wordI = 1:sentence_size
        dV = dV + delta_y(:,wordI)*s(:,wordI)';
    end

    ds_0 = zeros(size(s_0));
    dU_c = zeros(size(U_c));
    dU_r = zeros(size(U_r));
    dU_z = zeros(size(U_z));
    dW_c = zeros(size(W_c));
    dW_r = zeros(size(W_r));
    dW_z = zeros(size(W_z));
    db_z = zeros(size(b_z));
    db_r = zeros(size(b_r));
    db_c = zeros(size(b_c));
    ds_single = V'*delta_y;
    % calculate the derivative contribution of each step and add them up
    ds_cur = zeros(size(ds_single,1), 1);
    for wordJ = sentence_size:-1:2
        ds_cur = ds_cur + ds_single(:,wordJ);
        ds_cur_bk = ds_cur;

        dtanhInput = (ds_cur.*(1-z(:,wordJ)).*(1-c(:,wordJ).*c(:,wordJ)));
        db_c = db_c + dtanhInput;
        dU_c = dU_c + dtanhInput*x(:,wordJ)'; %could be accelerated by avoiding add 0
        dW_c = dW_c + dtanhInput*(s(:,wordJ-1).*r(:,wordJ))';
        dsr = W_c'*dtanhInput;
        ds_cur = dsr.*r(:,wordJ);
        dsigInput_r = dsr.*s(:,wordJ-1).*r(:,wordJ).*(1-r(:,wordJ));
        db_r = db_r + dsigInput_r;
        dU_r = dU_r + dsigInput_r*x(:,wordJ)'; %could be accelerated by avoiding add 0
        dW_r = dW_r + dsigInput_r*s(:,wordJ-1)';
        ds_cur = ds_cur + W_r'*dsigInput_r;

        ds_cur = ds_cur + ds_cur_bk.*z(:,wordJ);
        dz = ds_cur_bk.*(s(:,wordJ-1)-c(:,wordJ));
        dsigInput_z = dz.*z(:,wordJ).*(1-z(:,wordJ));
        db_z = db_z + dsigInput_z;
        dU_z = dU_z + dsigInput_z*x(:,wordJ)'; %could be accelerated by avoiding add 0
        dW_z = dW_z + dsigInput_z*s(:,wordJ-1)';
        ds_cur = ds_cur + W_z'*dsigInput_z;
    end

    % s_1
    ds_cur = ds_cur + ds_single(:,1);

    dtanhInput = (ds_cur.*(1-z(:,1)).*(1-c(:,1).*c(:,1)));
    db_c = db_c + dtanhInput;
    dU_c = dU_c + dtanhInput*x(:,1)'; %could be accelerated by avoiding add 0
    dW_c = dW_c + dtanhInput*(s_0.*r(:,1))';
    dsr = W_c'*dtanhInput;
    ds_0 = ds_0 + dsr.*r(:,1);
    dsigInput_r = dsr.*s_0.*r(:,1).*(1-r(:,1));
    db_r = db_r + dsigInput_r;
    dU_r = dU_r + dsigInput_r*x(:,1)'; %could be accelerated by avoiding add 0
    dW_r = dW_r + dsigInput_r*s_0';
    ds_0 = ds_0 + W_r'*dsigInput_r;

    ds_0 = ds_0 + ds_cur.*z(:,1);
    dz = ds_cur.*(s_0-c(:,1));
    dsigInput_z = dz.*z(:,1).*(1-z(:,1));
    db_z = db_z + dsigInput_z;
    dU_z = dU_z + dsigInput_z*x(:,1)'; %could be accelerated by avoiding add 0
    dW_z = dW_z + dsigInput_z*s_0';
    ds_0 = ds_0 + W_z'*dsigInput_z;
end
```

testBPTT_GRU.m

```
#include <string>
#include <vector>

#include "caffe/blob.hpp"
#include "caffe/common.hpp"
#include "caffe/filler.hpp"
#include "caffe/layer.hpp"
#include "caffe/sequence_layers.hpp"
#include "caffe/util/math_functions.hpp"

namespace caffe {

template <typename Dtype>
void GRULayer<Dtype>::RecurrentInputBlobNames(vector<string>* names) const {
  names->resize(2);
  (*names)[0] = "s_0";
  (*names)[1] = "c_0";
}

template <typename Dtype>
void GRULayer<Dtype>::RecurrentOutputBlobNames(vector<string>* names) const {
  names->resize(2);
  (*names)[0] = "s_" + this->int_to_str(this->T_);
  (*names)[1] = "c_T";
}

template <typename Dtype>
void GRULayer<Dtype>::RecurrentInputShapes(vector<BlobShape>* shapes) const {
  const int num_output = this->layer_param_.recurrent_param().num_output();
  const int num_blobs = 2;
  shapes->resize(num_blobs);
  for (int i = 0; i < num_blobs; ++i) {
    (*shapes)[i].Clear();
    (*shapes)[i].add_dim(1);  // a single timestep
    (*shapes)[i].add_dim(this->N_);
    (*shapes)[i].add_dim(num_output);
  }
}

template <typename Dtype>
void GRULayer<Dtype>::OutputBlobNames(vector<string>* names) const {
  names->resize(1);
  (*names)[0] = "s";
}

// modified from lstm_layer.cpp
/*  h -- > s
*   c -- > c, omit the reset gate
*/
```

Figure 9: Page 1 in 5 GRU implementation in Caffe.

```
template <typename Dtype>
void GRULayer<Dtype>::FillUnrolledNet(NetParameter* net_param) const {
  const int num_output = this->layer_param_.recurrent_param().num_output();
  CHECK_GT(num_output, 0) << "num_output must be positive";
  const FillerParameter& weight_filler =
      this->layer_param_.recurrent_param().weight_filler();
  const FillerParameter& bias_filler =
      this->layer_param_.recurrent_param().bias_filler();

  // Add generic LayerParameter's (without bottoms/tops) of layer types we'll
  // use to save redundant code.
  LayerParameter hidden_param;
  hidden_param.set_type("InnerProduct");
  hidden_param.mutable_inner_product_param()->set_num_output(num_output * 2);
  hidden_param.mutable_inner_product_param()->set_bias_term(false);
  hidden_param.mutable_inner_product_param()->set_axis(2);
  hidden_param.mutable_inner_product_param()->
      mutable_weight_filler()->CopyFrom(weight_filler);

  LayerParameter biased_hidden_param(hidden_param);
  biased_hidden_param.mutable_inner_product_param()->set_bias_term(true);
  biased_hidden_param.mutable_inner_product_param()->
      mutable_bias_filler()->CopyFrom(bias_filler);

  LayerParameter sum_param;
  sum_param.set_type("Eltwise");
  sum_param.mutable_eltwise_param()->set_operation(
      EltwiseParameter_EltwiseOp_SUM);

  LayerParameter scalar_param;
  scalar_param.set_type("Scalar");
  scalar_param.mutable_scalar_param()->set_axis(0);

  LayerParameter slice_param;
  slice_param.set_type("Slice");
  slice_param.mutable_slice_param()->set_axis(0);

  LayerParameter split_param;
  split_param.set_type("Split");

  vector<BlobShape> input_shapes;
  RecurrentInputShapes(&input_shapes);
  CHECK_EQ(2, input_shapes.size());

  net_param->add_input("c_0");
  net_param->add_input_shape()->CopyFrom(input_shapes[0]);

  net_param->add_input("s_0");
  net_param->add_input_shape()->CopyFrom(input_shapes[1]);
```

Figure 10: Page 2 in 5 GRU implementation in Caffe.

```
LayerParameter* cont_slice_param = net_param->add_layer();
cont_slice_param->CopyFrom(slice_param);
cont_slice_param->set_name("cont_slice");
cont_slice_param->add_bottom("cont");
cont_slice_param->mutable_slice_param()->set_axis(0);

// Add layer to transform all timesteps of x to the hidden state dimension.
//     W_xc_x = W_xc * x + b_c
{
  LayerParameter* x_transform_param = net_param->add_layer();
  x_transform_param->CopyFrom(biased_hidden_param);
  x_transform_param->set_name("x_transform");
  x_transform_param->add_param()->set_name("W_xc");
  x_transform_param->add_param()->set_name("b_c");
  x_transform_param->add_bottom("x");
  x_transform_param->add_top("W_xc_x");
}

if (this->static_input_) {
  // Add layer to transform x_static to the gate dimension.
  //     W_xc_x_static = W_xc_static * x_static
  LayerParameter* x_static_transform_param = net_param->add_layer();
  x_static_transform_param->CopyFrom(hidden_param);
  x_static_transform_param->mutable_inner_product_param()->set_axis(1);
  x_static_transform_param->set_name("W_xc_x_static");
  x_static_transform_param->add_param()->set_name("W_xc_static");
  x_static_transform_param->add_bottom("x_static");
  x_static_transform_param->add_top("W_xc_x_static_preshape");

  LayerParameter* reshape_param = net_param->add_layer();
  reshape_param->set_type("Reshape");
  BlobShape* new_shape =
       reshape_param->mutable_reshape_param()->mutable_shape();
  new_shape->add_dim(1);  // One timestep.
  // Should infer this->N as the dimension so we can reshape on batch size.
  new_shape->add_dim(-1);
  new_shape->add_dim(
      x_static_transform_param->inner_product_param().num_output());
  reshape_param->add_bottom("W_xc_x_static_preshape");
  reshape_param->add_top("W_xc_x_static");
}

LayerParameter* x_slice_param = net_param->add_layer();
x_slice_param->CopyFrom(slice_param);
x_slice_param->add_bottom("W_xc_x");
x_slice_param->set_name("W_xc_x_slice");

LayerParameter output_concat_layer;
output_concat_layer.set_name("h_concat");
```

Figure 11: Page 3 in 5 GRU implementation in Caffe.

```
output_concat_layer.set_type("Concat");
output_concat_layer.add_top("s");
output_concat_layer.mutable_concat_param()->set_axis(0);

for (int t = 1; t <= this->T_; ++t) {
  string tm1s = this->int_to_str(t - 1);
  string ts = this->int_to_str(t);

  cont_slice_param->add_top("cont_" + ts);
  x_slice_param->add_top("W_xc_x_" + ts);

  // Add layers to flush the hidden state when beginning a new
  // sequence, as indicated by cont_t.
  //     h_conted_{t-1} := cont_t * h_{t-1}
  //
  // Normally, cont_t is binary (i.e., 0 or 1), so:
  //     h_conted_{t-1} := h_{t-1} if cont_t == 1
  //                       0   otherwise
  {
    LayerParameter* cont_h_param = net_param->add_layer();
    cont_h_param->CopyFrom(scalar_param);
    cont_h_param->set_name("h_conted_" + tm1s);
    cont_h_param->add_bottom("s_" + tm1s);
    cont_h_param->add_bottom("cont_" + ts);
    cont_h_param->add_top("h_conted_" + tm1s);
  }

  // Add layer to compute
  //     W_hc_h_{t-1} := W_hc * h_conted_{t-1}
  {
    LayerParameter* w_param = net_param->add_layer();
    w_param->CopyFrom(hidden_param);
    w_param->set_name("transform_" + ts);
    w_param->add_param()->set_name("W_hc");
    w_param->add_bottom("h_conted_" + tm1s);
    w_param->add_top("W_hc_h_" + tm1s);
    w_param->mutable_inner_product_param()->set_axis(2);
  }

  // Add the outputs of the linear transformations to compute the gate input.
  //     gate_input_t := W_hc * h_conted_{t-1} + W_xc * x_t + b_c
  //                   = W_hc_h_{t-1} + W_xc_x_t + b_c
  {
    LayerParameter* input_sum_layer = net_param->add_layer();
    input_sum_layer->CopyFrom(sum_param);
    input_sum_layer->set_name("gate_input_" + ts);
    input_sum_layer->add_bottom("W_hc_h_" + tm1s);
    input_sum_layer->add_bottom("W_xc_x_" + ts);
    if (this->static_input_) {
      input_sum_layer->add_bottom("W_xc_x_static");
```

Figure 12: Page 4 in 5 GRU implementation in Caffe.

```
      }
      input_sum_layer->add_top("gate_input_" + ts);
    }

    // Add GRUUnit layer to compute the cell & hidden vectors c_t and s_t.
    {
      LayerParameter* gru_unit_param = net_param->add_layer();
      gru_unit_param->set_type("GRUUnit");
      gru_unit_param->add_bottom("c_" + tm1s);
      gru_unit_param->add_bottom("gate_input_" + ts);
      gru_unit_param->add_bottom("cont_" + ts);
      gru_unit_param->add_top("c_" + ts);
      gru_unit_param->add_top("s_" + ts);
      gru_unit_param->set_name("unit_" + ts);
    }
    output_concat_layer.add_bottom("s_" + ts);
  }  // for (int t = 1; t <= this->T_; ++t)

  {
    LayerParameter* c_T_copy_param = net_param->add_layer();
    c_T_copy_param->CopyFrom(split_param);
    c_T_copy_param->add_bottom("c_" + this->int_to_str(this->T_));
    c_T_copy_param->add_top("c_T");
  }
  net_param->add_layer()->CopyFrom(output_concat_layer);
}

INSTANTIATE_CLASS(GRULayer);
REGISTER_LAYER_CLASS(GRU);

}  // namespace caffe
```

Figure 13: Page 5 in 5 GRU implementation in Caffe. Note GRUUnit layer is used in this page.

```cpp
#include <algorithm>
#include <cmath>
#include <vector>

#include "caffe/layer.hpp"
#include "caffe/sequence_layers.hpp"

namespace caffe {

template <typename Dtype>
inline Dtype sigmoid(Dtype x) {
  return 1. / (1. + exp(-x));
}

template <typename Dtype>
inline Dtype tanh(Dtype x) {
  return 2. * sigmoid(2. * x) - 1.;
}

template <typename Dtype>
void GRUUnitLayer<Dtype>::Reshape(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const int num_instances = bottom[0]->shape(1);
  for (int i = 0; i < bottom.size(); ++i) {
    if (i == 2) {
      CHECK_EQ(2, bottom[i]->num_axes());
    } else {
      CHECK_EQ(3, bottom[i]->num_axes());
    }
    CHECK_EQ(1, bottom[i]->shape(0));
    CHECK_EQ(num_instances, bottom[i]->shape(1));
  }
  hidden_dim_ = bottom[0]->shape(2);
  CHECK_EQ(num_instances, bottom[1]->shape(1));
  CHECK_EQ(2 * hidden_dim_, bottom[1]->shape(2));
  top[0]->ReshapeLike(*bottom[0]);
  top[1]->ReshapeLike(*bottom[0]);
  X_acts_.ReshapeLike(*bottom[1]);
}

template <typename Dtype>
void GRUUnitLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const int num = bottom[0]->shape(1);
  const int x_dim = hidden_dim_ * 2;
  const Dtype* C_prev = bottom[0]->cpu_data();
  const Dtype* X = bottom[1]->cpu_data();
  const Dtype* flush = bottom[2]->cpu_data();
  Dtype* C = top[0]->mutable_cpu_data();
  Dtype* S = top[1]->mutable_cpu_data();
  for (int n = 0; n < num; ++n) {
    for (int d = 0; d < hidden_dim_; ++d)
    {
      const Dtype z = (*flush == 0) ? 0 : (*flush * sigmoid(X[d]));
      const Dtype h = tanh(X[hidden_dim_ + d]);
      S[d] = C[d] = (1-z)*h + z*C_prev[d];
    }
    C_prev += hidden_dim_;
    X += x_dim;
    C += hidden_dim_;
    S += hidden_dim_;
    ++flush;
  }
}
```

Figure 14: GRU unit layer implementation in C++ page 1/2

```
template <typename Dtype>
void GRUUnitLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom) {
  CHECK(!propagate_down[2]) << "Cannot backpropagate to sequence indicators.";
  if (!propagate_down[0] && !propagate_down[1]) { return; }

  const int num = bottom[0]->shape(1);
  const int x_dim = hidden_dim_ * 2;
  const Dtype* C_prev = bottom[0]->cpu_data();
  const Dtype* X = bottom[1]->cpu_data();
  const Dtype* flush = bottom[2]->cpu_data();
  const Dtype* C = top[0]->cpu_data();
  const Dtype* S = top[1]->cpu_data();
  const Dtype* C_diff = top[0]->cpu_diff();
  const Dtype* S_diff = top[1]->cpu_diff();
  Dtype* C_prev_diff = bottom[0]->mutable_cpu_diff();
  Dtype* X_diff = bottom[1]->mutable_cpu_diff();
  for (int n = 0; n < num; ++n) {
    for (int d = 0; d < hidden_dim_; ++d)
    {
      const Dtype z = (*flush == 0) ? 0 : (*flush * sigmoid(X[d]));
      const Dtype h = tanh(X[hidden_dim_ + d]);
      Dtype* c_prev_diff = C_prev_diff + d;
      Dtype* z_diff = X_diff + d;
      Dtype* h_diff = X_diff + hidden_dim_ + d;
      const Dtype c_term_diff = C_diff[d] + S_diff[d];
      *c_prev_diff = c_term_diff * z;
      *z_diff = c_term_diff * (C_prev[d] - h) * z * (1 - z);
      *h_diff = c_term_diff * (1 - z) * (1 - h*h);
    }
    C_prev += hidden_dim_;
    X += x_dim;
    C += hidden_dim_;
    S += hidden_dim_;
    C_diff += hidden_dim_;
    S_diff += hidden_dim_;
    X_diff += x_dim;
    C_prev_diff += hidden_dim_;
    ++flush;
  }
}

#ifdef CPU_ONLY
STUB_GPU(GRUUnitLayer);
#endif

INSTANTIATE_CLASS(GRUUnitLayer);
REGISTER_LAYER_CLASS(GRUUnit);

}  // namespace caffe
```

Figure 15: GRU unit layer implementation in C++ page 2/2

```
#include <algorithm>
#include <cmath>
#include <vector>

#include "caffe/layer.hpp"
#include "caffe/sequence_layers.hpp"

namespace caffe {

template <typename Dtype>
__device__ Dtype sigmoid(const Dtype x) {
  return Dtype(1) / (Dtype(1) + exp(-x));
}

template <typename Dtype>
__device__ Dtype tanh(const Dtype x) {
  return Dtype(2) * sigmoid(Dtype(2) * x) - Dtype(1);
}

template <typename Dtype>
__global__ void GRUActsForward(const int nthreads, const int dim,
                                const Dtype* X, Dtype* X_acts) {
  CUDA_KERNEL_LOOP(index, nthreads) {
    const int x_dim = 2 * dim;
    const int d = index % x_dim;
    if (d < dim) {
      X_acts[index] = sigmoid(X[index]);
    } else {
      X_acts[index] = tanh(X[index]);
    }
  }
}

template <typename Dtype>
__global__ void GRUUnitForward(const int nthreads, const int dim,
    const Dtype* C_prev, const Dtype* X, const Dtype* flush,
    Dtype* C, Dtype* H) {
  CUDA_KERNEL_LOOP(index, nthreads) {
    const int n = index / dim;
    const int d = index % dim;
    const Dtype* X_offset = X + 2 * dim * n;
    const Dtype z = (flush[n] == Dtype(0)) ? Dtype(0) : (flush[n] * X_offset[d]);
    const Dtype h = X_offset[dim + d];
    const Dtype c_prev = C_prev[index];
    H[index] = C[index] = z * c_prev + (Dtype(1)-z)*h;
  }
}

template <typename Dtype>
void GRUUnitLayer<Dtype>::Forward_gpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const int count = top[1]->count();
  const Dtype* C_prev = bottom[0]->gpu_data();
  const Dtype* X = bottom[1]->gpu_data();
  const Dtype* flush = bottom[2]->gpu_data();
  Dtype* X_acts = X_acts_.mutable_gpu_data();
  Dtype* C = top[0]->mutable_gpu_data();
  Dtype* H = top[1]->mutable_gpu_data();
  const int X_count = bottom[1]->count();
  // NOLINT_NEXT_LINE(whitespace/operators)
  GRUActsForward<Dtype><<<CAFFE_GET_BLOCKS(X_count), CAFFE_CUDA_NUM_THREADS>>>(
      X_count, hidden_dim_, X, X_acts);
  CUDA_POST_KERNEL_CHECK;
  // NOLINT_NEXT_LINE(whitespace/operators)
```

Figure 16: GRU unit layer implementation in CUDA C (running on GPU) page 1/3

```
  GRUUnitForward<Dtype><<<CAFFE_GET_BLOCKS(count), CAFFE_CUDA_NUM_THREADS>>>(
      count, hidden_dim_, C_prev, X_acts, flush, C, H);
  CUDA_POST_KERNEL_CHECK;
}

template <typename Dtype>
__global__ void GRUUnitBackward(const int nthreads, const int dim,
    const Dtype* C_prev, const Dtype* X, const Dtype* C, const Dtype* H,
    const Dtype* flush, const Dtype* C_diff, const Dtype* H_diff,
    Dtype* C_prev_diff, Dtype* X_diff) {
  CUDA_KERNEL_LOOP(index, nthreads) {
    const int n = index / dim;
    const int d = index % dim;
    const Dtype* X_offset = X + 2 * dim * n;
    const Dtype z = (flush[n] == Dtype(0)) ? Dtype(0) : (flush[n] * X_offset[d]);
    const Dtype h = X_offset[dim + d];
    const Dtype c_prev = C_prev[index];
    const Dtype c = C[index];
    Dtype* c_prev_diff = C_prev_diff + index;
    Dtype* X_diff_offset = X_diff + 2 * dim * n;
    Dtype* z_diff = X_diff_offset + d;
    Dtype* h_diff = X_diff_offset + dim + d;
    const Dtype c_term_diff = C_diff[index] + H_diff[index];

    *c_prev_diff = c_term_diff * z;
    *z_diff = c_term_diff * (c_prev - h);
    *h_diff = c_term_diff * (Dtype(1) - z);
  }
}

template <typename Dtype>
__global__ void GRUActsBackward(const int nthreads, const int dim,
    const Dtype* X_acts, const Dtype* X_acts_diff, Dtype* X_diff) {
  CUDA_KERNEL_LOOP(index, nthreads) {
    const int x_dim = 2 * dim;
    const int d = index % x_dim;
    const Dtype X_act = X_acts[index];
    if (d < dim) {
      X_diff[index] = X_acts_diff[index] * X_act * (Dtype(1) - X_act);
    } else {
      X_diff[index] = X_acts_diff[index] * (Dtype(1) - X_act * X_act);
    }
  }
}

template <typename Dtype>
void GRUUnitLayer<Dtype>::Backward_gpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
  CHECK(!propagate_down[2]) << "Cannot backpropagate to sequence indicators.";
  if (!propagate_down[0] && !propagate_down[1]) { return; }

  const int count = top[1]->count();
  const Dtype* C_prev = bottom[0]->gpu_data();
  const Dtype* X_acts = X_acts_.gpu_data();
  const Dtype* flush = bottom[2]->gpu_data();
  const Dtype* C = top[0]->gpu_data();
  const Dtype* H = top[1]->gpu_data();
  const Dtype* C_diff = top[0]->gpu_diff();
  const Dtype* H_diff = top[1]->gpu_diff();
  Dtype* C_prev_diff = bottom[0]->mutable_gpu_diff();
  Dtype* X_acts_diff = X_acts_.mutable_gpu_diff();
  GRUUnitBackward<Dtype>  // NOLINT_NEXT_LINE(whitespace/operators)
      <<<CAFFE_GET_BLOCKS(count), CAFFE_CUDA_NUM_THREADS>>>(count, hidden_dim_,
```

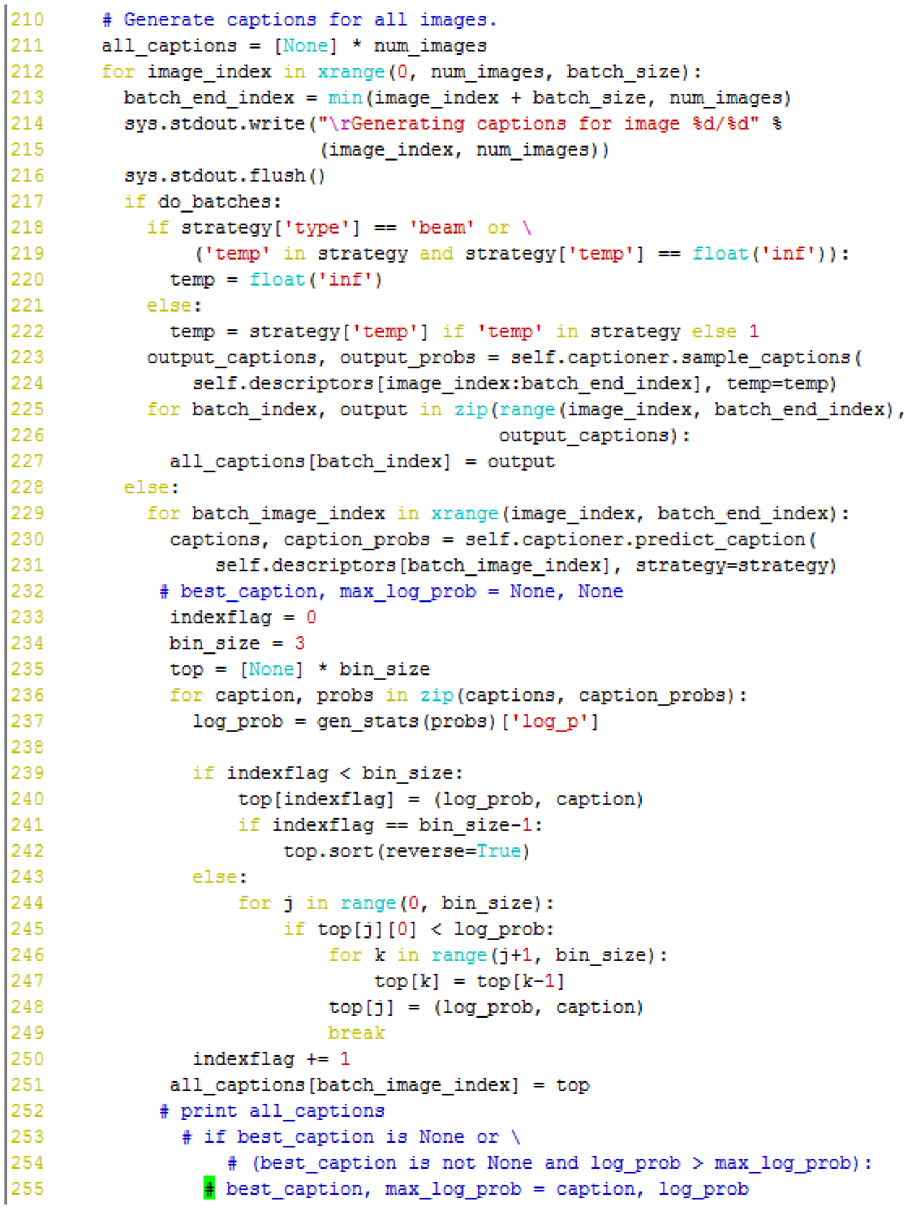
Figure 17: GRU unit layer implementation in CUDA C (running on GPU) page 2/3

```
      C_prev, X_acts, C, H, flush, C_diff, H_diff, C_prev_diff, X_acts_diff);
  CUDA_POST_KERNEL_CHECK;
  const int X_count = bottom[1]->count();
  Dtype* X_diff = bottom[1]->mutable_gpu_diff();
  GRUActsBackward<Dtype>  // NOLINT_NEXT_LINE(whitespace/operators)
      <<<CAFFE_GET_BLOCKS(X_count), CAFFE_CUDA_NUM_THREADS>>>(
      X_count, hidden_dim_, X_acts, X_acts_diff, X_diff);
  CUDA_POST_KERNEL_CHECK;
}

INSTANTIATE_LAYER_GPU_FUNCS(GRUUnitLayer);

}  // namespace caffe
```

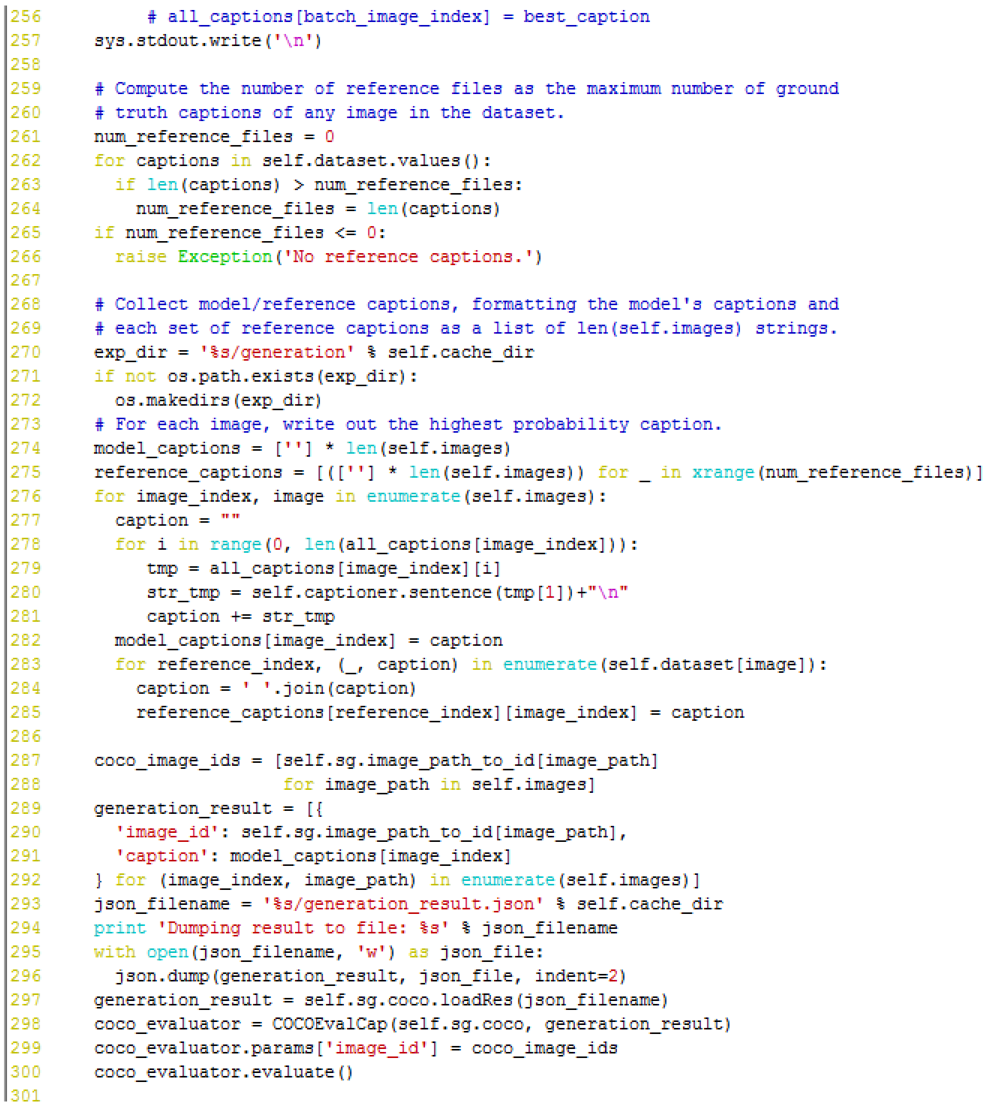
Figure 18: GRU unit layer implementation in CUDA C (running on GPU) page 3/3



```
    # Generate captions for all images.
    all_captions = [None] * num_images
    for image_index in xrange(0, num_images, batch_size):
      batch_end_index = min(image_index + batch_size, num_images)
      sys.stdout.write("\rGenerating captions for image %d/%d" %
                       (image_index, num_images))
      sys.stdout.flush()
      if do_batches:
        if strategy['type'] == 'beam' or \
            ('temp' in strategy and strategy['temp'] == float('inf')):
          temp = float('inf')
        else:
          temp = strategy['temp'] if 'temp' in strategy else 1
        output_captions, output_probs = self.captioner.sample_captions(
            self.descriptors[image_index:batch_end_index], temp=temp)
        for batch_index, output in zip(range(image_index, batch_end_index),
                                       output_captions):
          all_captions[batch_index] = output
      else:
        for batch_image_index in xrange(image_index, batch_end_index):
          captions, caption_probs = self.captioner.predict_caption(
              self.descriptors[batch_image_index], strategy=strategy)
         # best_caption, max_log_prob = None, None
          indexflag = 0
          bin_size = 3
          top = [None] * bin_size
          for caption, probs in zip(captions, caption_probs):
            log_prob = gen_stats(probs)['log_p']

            if indexflag < bin_size:
                top[indexflag] = (log_prob, caption)
                if indexflag == bin_size-1:
                    top.sort(reverse=True)
            else:
                for j in range(0, bin_size):
                    if top[j][0] < log_prob:
                        for k in range(j+1, bin_size):
                            top[k] = top[k-1]
                        top[j] = (log_prob, caption)
                        break
            indexflag += 1
          all_captions[batch_image_index] = top
         # print all_captions
           # if best_caption is None or \
               # (best_caption is not None and log_prob > max_log_prob):
             # best_caption, max_log_prob = caption, log_prob
```

Figure 19: Sentence generation part 1.



```
         # all_captions[batch_image_index] = best_caption
    sys.stdout.write('\n')

    # Compute the number of reference files as the maximum number of ground
    # truth captions of any image in the dataset.
    num_reference_files = 0
    for captions in self.dataset.values():
      if len(captions) > num_reference_files:
        num_reference_files = len(captions)
    if num_reference_files <= 0:
      raise Exception('No reference captions.')

    # Collect model/reference captions, formatting the model's captions and
    # each set of reference captions as a list of len(self.images) strings.
    exp_dir = '%s/generation' % self.cache_dir
    if not os.path.exists(exp_dir):
      os.makedirs(exp_dir)
    # For each image, write out the highest probability caption.
    model_captions = [''] * len(self.images)
    reference_captions = [([''] * len(self.images)) for _ in xrange(num_reference_files)]
    for image_index, image in enumerate(self.images):
      caption = ""
      for i in range(0, len(all_captions[image_index])):
         tmp = all_captions[image_index][i]
         str_tmp = self.captioner.sentence(tmp[1])+"\n"
         caption += str_tmp
      model_captions[image_index] = caption
      for reference_index, (_, caption) in enumerate(self.dataset[image]):
        caption = ' '.join(caption)
        reference_captions[reference_index][image_index] = caption

    coco_image_ids = [self.sg.image_path_to_id[image_path]
                      for image_path in self.images]
    generation_result = [{
      'image_id': self.sg.image_path_to_id[image_path],
      'caption': model_captions[image_index]
    } for (image_index, image_path) in enumerate(self.images)]
    json_filename = '%s/generation_result.json' % self.cache_dir
    print 'Dumping result to file: %s' % json_filename
    with open(json_filename, 'w') as json_file:
      json.dump(generation_result, json_file, indent=2)
    generation_result = self.sg.coco.loadRes(json_filename)
    coco_evaluator = COCOEvalCap(self.sg.coco, generation_result)
    coco_evaluator.params['image_id'] = coco_image_ids
    coco_evaluator.evaluate()
```

Figure 20: Sentence generation part 2.