Final report

1. Introduction.

Overview of project:

The project topic we choose is Visual Question Answer. The idea comes from a paper of Virginia Tech named "VQA: Visual Question Answering" [1]. They posted their resource on www.visualqa.org, like dataset, demo. Our goal is to reproduce the LSTM+CNN Model in their paper by keras, and use our model to answer some open-ended questions.

In vqa dataset, it has three parts: images, answers, and questions. Images from MSCOCO, answers and questions created by human. Based on our syllabus, we are going to reach some approaches like CNN, LSTM and MLP. Our project uses top 1000 answers as the labels. CNN part deals with extracting image features and LSTM deal with extracting features from questions, then we feed above two parts of features into a fully connected network to predict the most answer (the most likely answer in top 1000 answers).

1 Antol, S., Agrawal, A., Lu, J., Mitchell, M., Batra, D., Lawrence Zitnick, C., & Parikh, D. (2015). Vqa: Visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2425-2433).

2. Data set

2.1. Image

We use the 204,721 training images from the Microsoft Common Objects in Context (MS COCO) datase[2]t. COCO is a large-scale object detection, segmentation, and captioning dataset. Each image in COCO contains a lot of informations. What we need is just the image file and image id.

Figure 1: COCO_val2017_000000324158.jpg

2.2. Text Question and Answers

There are 1,105, 904 questions and answers. The answers and questions are created by human. For each image, dataset collected three question and answered them. Some of questions only require low level computer vision techniques. Like ‘Is that a dog?’, ‘How many apples in the picture?’ Moreover, dataset also has some questions like ‘What sound does the pictured animal make? ’ which is not only based on recognizing objects from picture.

Q1: Where is he looking? A1:down

Q2: What are the people in the background doing? A2:watching

Figure 2: COCO_val2017_000000000338.jpg and related questions and answers

2 Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., ... & Zitnick, C. L. (2014, September). Microsoft coco: Common objects in context. In *European conference on computer vision* (pp. 740-755). Springer, Cham.

3. Description of the deep learning network and training algorithm

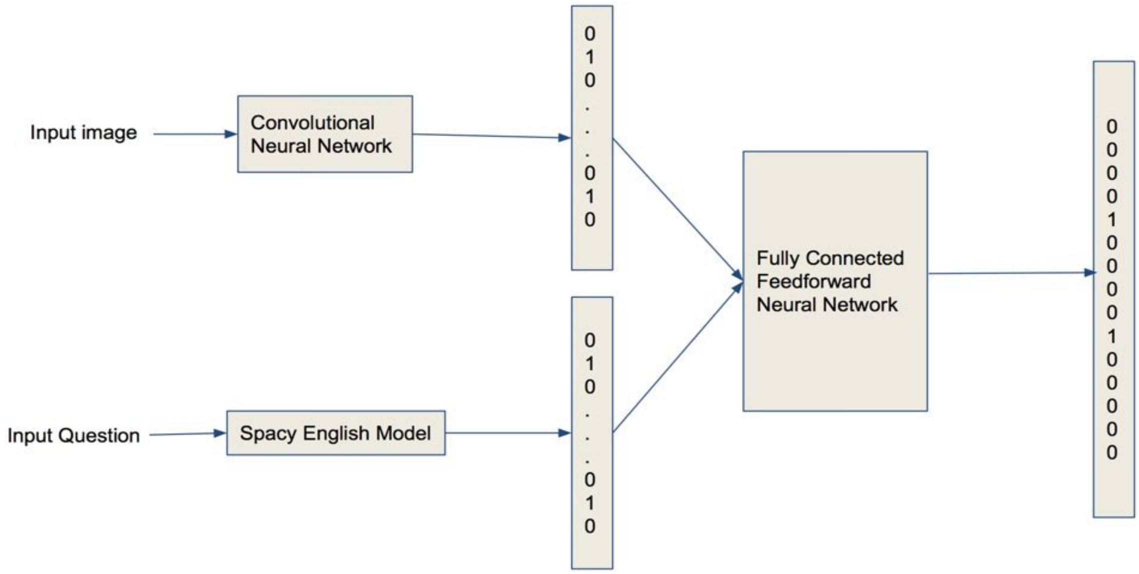


Figure 3: Whole model

For our algorithm, we combine two parts input images and input questions together to generate a large vectors as the data input. We choose top 1000 answers which can represent 82.67% of the whole data set. Following we will describe details of techniques and our three models:

(1) Implement VGG 16 in our model, use the LSTM to process questions.
(2) Directly get image's feature from other's well-trained CNN model,directly use question vector (question part don't use LSTM)
(3) Directly get image's feature from other's well-trained CNN model, use the LSTM to process questions.

3.1 Other's well-trained CNN model - NeuralTalk2

NeuralTalk2[3] is an open-sourced and artificially-intelligent image captioning program. NeuralTalk2 can generate a description of a specific image.It also posted his model of CNN part and weights files. In his git, he also provides the extract_features.py which can get 4096 features from images by using his CNN model.

---

[3] Karpathy, A., & Fei-Fei, L. (2015). Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 3128-3137).

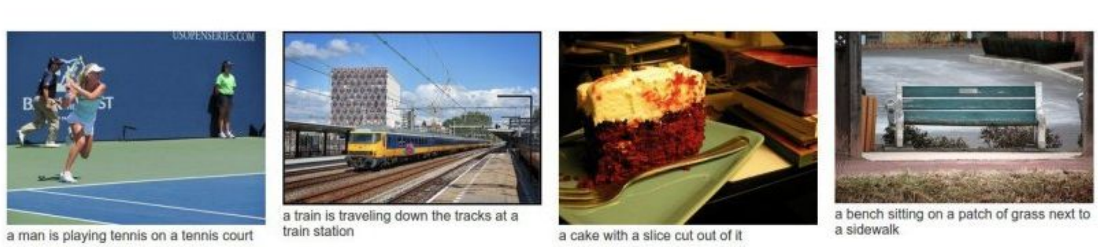
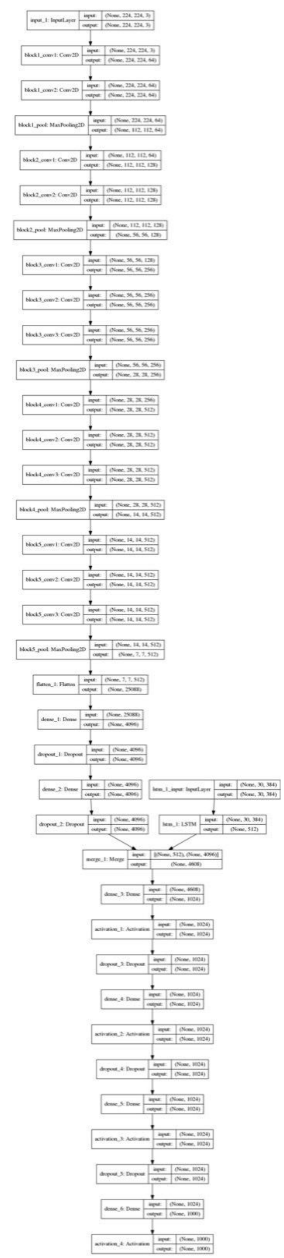


Figure 3: NeuralTalk2 generated a description of a image



3.2 VGG 16

We use the last hidden layer of the 16-layer Oxford VGG Convolutional network that was trained on the newly-released Microsoft Common Objects in Context(MS COCO) dataset as the image vector input. While VGG Net is not the best algorithm for extract features, Google Net and ResNet have a better classification results, but VGG Net is very versatile, simple, relatively small and more importantly portable to use

3.3 SpaCy

For the question, we transform each word to its word vector, and sum up all the vectors. The length of this feature vector will be same as the length of a single word vector, and the word vectors(also called embedding) that we use have a length of 384. There are a lot of different algorithms to transfer text into vector. The most popular is Word2Vec whereas these days state of the art uses skip-thought vectors or positional encodings. We will use Word2Vec from Stanford called Glove(Spacy English Model). Glove reduces a given token into a 384 dimensional representation.

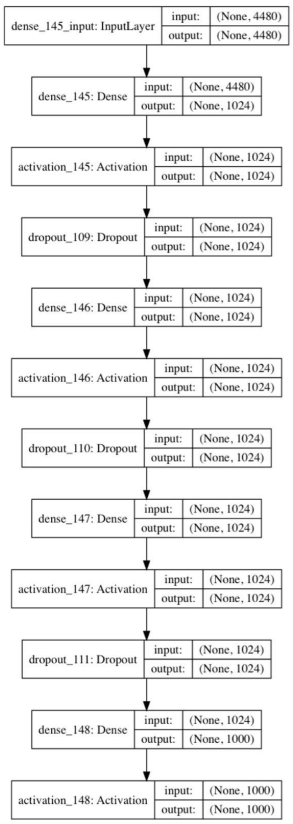
3.4 Fully implement model

In this model, we implement whole by ourselves. For each sample, the input are image and question, the labels are answers(top1000 frequent word).

We called VGG model from keras and drop the top layers like the last dropout layer and softmax layer.So the result is a 4096-dimension vector.

We convert question text to a word matrix which rows are index of word in sentence and the columns are the vectors representing each word. Then we feed the word matrix into LSTM to extract 384 features.

Finally we concat image features and question as the input of final fully connected network.And the most likely answer.



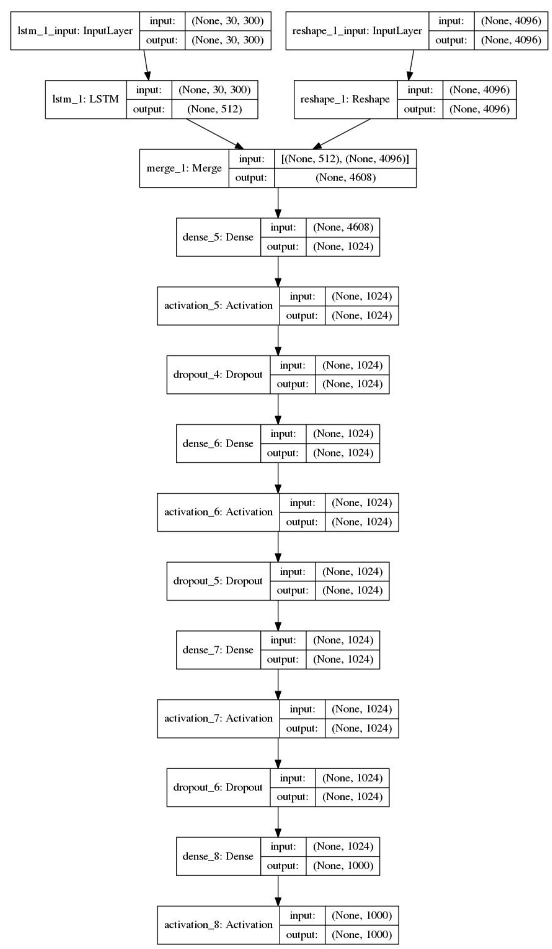
3.5 other's well-trained CNN + LSTM model

In this model, we implement whole by ourselves except CNN part. For each sample, the input are image and question, the labels are answers(top1000 frequent word).

We called NeuralTalk2 to get images' features.So the result is a 4096-dimension vector.

We convert question text to a word matrix which rows are index of word in sentence and the columns are the vectors representing each word. Then we feed the word matrix into LSTM to extract 384 features.

Finally we concat image features and question as the input of final fully connected network.And the most likely answer.



3.6 other's well-trained CNN + MLP model

In this model, we implement whole by ourselves except CNN part. For each sample, the input are image and question, the labels are answers(top1000 frequent word).

We called NeuralTalk2 to get images' features.So the result is a 4096-dimension vector.

For the question, we transform each word to its word vector, and sum up all the vector as the input of next network.

Finally we concat image features and question as the input of final fully connected network.And the most likely answer.

4. Experimental setup.

4.1. Describe how you are going to use the data to train and test the network.

From the MSCOCO dataset, we are able to download the all dataset of images/questions/answers which was already divided by train and test, yet as running the COCO API is time consuming, we retrieve the data by the release version of json dataset provided by Virginia Tech.

Index of /vqa/release_data/mscoco/vqa

Name	Last modified	Size	Descriptio
Parent Directory		-	
Annotations_Train_mscoco.zip	03-Oct-2015 18:16	12M	
Annotations_Val_mscoco.zip	03-Oct-2015 18:16	5.8M	
MultipleChoice_mscoco_test-dev2015_questions.json	03-Oct-2015 18:13	16M	
MultipleChoice_mscoco_test2015_questions.json	02-Oct-2015 15:47	64M	
MultipleChoice_mscoco_train2014_questions.json	02-Oct-2015 15:33	66M	
MultipleChoice_mscoco_train2014_questions_filtered.json	18-Aug-2016 11:43	62M	
MultipleChoice_mscoco_val2014_questions.json	02-Oct-2015 15:38	32M	
MultipleChoice_mscoco_val2014_questions_filtered.json	18-Aug-2016 11:43	30M	
OpenEnded_mscoco_test-dev2015_questions.json	03-Oct-2015 18:12	5.3M	
OpenEnded_mscoco_test2015_questions.json	02-Oct-2015 15:41	21M	
OpenEnded_mscoco_train2014_questions.json	02-Oct-2015 15:28	22M	
OpenEnded_mscoco_val2014_questions.json	02-Oct-2015 15:35	11M	
Questions_Test_mscoco.zip	03-Oct-2015 18:16	25M	
Questions_Train_mscoco.zip	03-Oct-2015 18:16	21M	
Questions_Val_mscoco.zip	03-Oct-2015 18:16	10M	
intermediate_formats/	28-Jan-2016 15:29	-	
mscoco_train2014_annotations.json	02-Oct-2015 15:29	189M	
mscoco_train2014_annotations_filtered.json	18-Aug-2016 11:43	158M	
mscoco_val2014_annotations.json	02-Oct-2015 15:35	93M	
mscoco_val2014_annotations_filtered.json	18-Aug-2016 11:43	77M	

https://vision.ece.vt.edu/vqa/release_data/mscoco/vqa/

4.2. Explain how you will implement the network in the chosen framework and how you will judge the performance.

We use the keras framework for the implement. By implement different learning rate, optimizer and batch size, we tune our model to the best performance

VGG model

We use VGG_16 model but remove the softmax layer for the image features extracted:

```
def VGG_16(weights_path=None):
    model = Sequential()
    model.add(ZeroPadding2D((1, 1), input_shape=(3, 224, 224)))
    model.add(Convolution2D(64, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(64, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))

    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(128, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(128, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))

    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))

    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))

    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1, 1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))

    model.add(Flatten())
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(4096, activation='relu'))
    #model.add(Dropout(0.5))
    #model.add(Dense(1000, activation='softmax'))
```

From the VGG model feature we retrieve:

```
vgg_model_path = '../Downloads/coco/vgg_feats.mat'

features_struct = scipy.io.loadmat(vgg_model_path)
VGGfeatures = features_struct['feats']
image_ids = open('../data/coco_vgg_IDMap.txt').read().splitlines()
id_map = {}
```

Getting the CNN features:

```
def get_images_matrix(img_coco_ids, img_map, VGGfeatures):
    nb_samples = len(img_coco_ids)
    nb_dimensions = VGGfeatures.shape[0]
    image_matrix = np.zeros((nb_samples, nb_dimensions))
    for j in range(len(img_coco_ids)):
        image_matrix[j,:] = VGGfeatures[:,img_map[img_coco_ids[j]]]
    return image_matrix
```

Vectorize the string by sum the word vector in one sentence, we can get the NLP features:

```
def get_questions_matrix_sum(questions, nlp):
    nb_samples = len(questions)
    word_vec_dim = nlp(questions[0].decode('utf-8'))[0].vector.shape[0]
    questions_matrix = np.zeros((nb_samples, word_vec_dim))
    for i in range(len(questions)):
        tokens = nlp(questions[i].decode('utf-8'))
        for j in range(len(tokens)):
            questions_matrix[i,:] += tokens[j].vector
    return questions_matrix
```

We stack stack the 4096 image features and 384 question features together as the final fully connected layer and feed it into next neural network, MLP and LSTM.

MLP Model

Building MLP:

```
# MLP
# Input
model = Sequential()
model.add(Dense(num_hidden_units, input_dim=img_dim+word_vec_dim, kernel_initializer='uniform'))
model.add(Activation(activation))
model.add(Dropout(dropout))

# Hidden
for i in range(num_hidden_layers-1):
    model.add(Dense(num_hidden_units, kernel_initializer='uniform'))
    model.add(Activation(activation))
    model.add(Dropout(dropout))

# Output
model.add(Dense(nb_classes, kernel_initializer='uniform'))
model.add(Activation('softmax'))
```

To train the MLP model, we first shuffle the data and read all question/answer/image data and feed it by mini batch:

```
from itertools import izip_longest
def batches(iterable, n, fillvalue=None):
    args = [iter(iterable)] * n
    return izip_longest(*args, fillvalue=fillvalue)

for k in range(num_epochs):
    index_shuf = [i for i in range(len(questions_train))]
    shuffle(index_shuf)

    questions_train = [questions_train[i] for i in index_shuf]
    answers_train = [answers_train[i] for i in index_shuf]
    images_train = [images_train[i] for i in index_shuf]
    progbar = generic_utils.Progbar(len(questions_train))

    for qu_batch,an_batch,im_batch in zip(batches(questions_train, batch_size, fillvalue=questions_train[-1]),
                                          batches(answers_train, batch_size, fillvalue=answers_train[-1]),
                                          batches(images_train, batch_size, fillvalue=images_train[-1])):
        X_q_batch = get_questions_matrix_sum(qu_batch, nlp)
        X_i_batch = get_images_matrix(im_batch, id_map, VGGfeatures)
        X_batch = np.hstack((X_q_batch, X_i_batch))
        Y_batch = get_answers_matrix(an_batch, labelencoder)
        loss = model.train_on_batch(X_batch, Y_batch)
        progbar.add(batch_size, values=[("train loss", loss)])
    if k%model_save_interval == 0:
        model.save_weights(model_file_name + '_epoch_{:02d}.hdf5'.format(k))

model.save_weights(model_file_name + '_epoch_{:02d}.hdf5'.format(k))
```

LSTM Model

Consider the time step, we define a new function to get the matrix:

```
def get_questions_tensor_timeseries(questions, nlp, timesteps):
    nb_samples = len(questions)
    word_vec_dim = nlp(questions[0].decode('utf-8'))[0].vector.shape[0]
    questions_tensor = np.zeros((nb_samples, timesteps, word_vec_dim))
    for i in xrange(len(questions)):
        tokens = nlp(questions[i].decode('utf-8'))
        for j in xrange(len(tokens)):
            if j< timesteps:
                questions_tensor[i, j, :] = tokens[j].vector

    return questions_tensor
```

Dealing with images:

```
image_model = Sequential()
image_model.add(Reshape((img_dim,), input_shape = (img_dim,)))
```

Dealing with Questions:

```
language_model = Sequential()
if num_hidden_layers_lstm == 1:
    language_model.add(LSTM(output_dim = num_hidden_units_lstm, return_sequences=False, input_shape=(max_len, word_vec_dim)))
else:
    language_model.add(LSTM(output_dim = num_hidden_units_lstm, return_sequences=True, input_shape=(max_len, word_vec_dim)))
    for i in xrange(num_hidden_layers_lstm-2):
        language_model.add(LSTM(output_dim = num_hidden_units_lstm, return_sequences=True))
    language_model.add(LSTM(output_dim = num_hidden_units_lstm, return_sequences=False))
```

LSTM model training:

Merge the image model with lstm model then do the classification.

```
model = Sequential()
model.add(Merge([language_model, image_model], mode='concat', concat_axis=1))
for i in xrange(num_hidlayer_mlp):
    model.add(Dense(num_hidunit_mlp, init='uniform'))
    model.add(Activation(activation_mlp))
    model.add(Dropout(dropout))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
```

4.3. Will you use minibatches? How will you determine the size of the minibatches?

Yes we use the minibatches by creating the batches function, we tried 3 batch size as 64, 128 and 256 and with below data we can tell that with a larger batch size, the run time will shrink yet the loss will descent slower. Meaning it will require more epochs' running if we want to reach the same loss, and the running time drop in a very slow rate after 256, we decide to use 128.

Batch Size	64	128	256
Descent in the 1st epoch	7.31 - 4.25	7.39 - 4.59	7.31 - 4.98
Run time(min)	40.13	36.5	34.32

MLP batches:

```
from itertools import izip_longest
def batches(iterable, n, fillvalue=None):
    args = [iter(iterable)] * n
    return izip_longest(*args, fillvalue=fillvalue)
```

LSTM batches:

```
for qu_batch,an_batch,im_batch in zip(batches(questions_train, batch_size, fillvalue=questions_train[-1]),
                                    batches(answers_train, batch_size, fillvalue=answers_train[-1]),
                                    batches(images_train, batch_size, fillvalue=images_train[-1])):
    X_q_batch = get_questions_matrix_sum(qu_batch, nlp)
    X_i_batch = get_images_matrix(im_batch, id_map, VGGfeatures)
    X_batch = np.hstack((X_q_batch, X_i_batch))
    Y_batch = get_answers_matrix(an_batch, labelencoder)
    loss = model.train_on_batch(X_batch, Y_batch)
    progbar.add(batch_size, values=[("train loss", loss)])
```

4.4. How will you determine training parameters (e.g., learning rate)?

We test three different learning rate (0.1, 0.01 and 0.001) for different training model.

```
json_string = model.to_json()
model_file_name = '/Users/jiafangliu/Documents/class/ML/Final/lr_001/mlp
open(model_file_name  + '.json', 'w').write(json_string)
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)
```

4.5. How will you detect/prevent overfitting and extrapolation?

(1) We can use validation dataset to evaluate our model or do cross validation if model works well on training dataset but not on validation dataset which means model is overfitting.
(2) We can use dropout to reduce overfitting.
(3) A larger dataset is the way to prevent overfitting.
(4) Add regularization

5. Results. Describe the results of your experiments, using figures and tables wherever possible. Include all results (including all figures and tables) in the main body of the report, not in appendices. Provide an explanation of each figure and table that you include. Your discussions in this section will be the most important part of the report. • Summary and conclusions. Summarize the results you obtained, explain what you have learned, and suggest improvements that could be made in the future.

MLP(SGD)

Learning Rate	0.001	0.01	0.1
Loss(Catogorical - cross entropy)	2.11	2.32	13.10

MLP(RMSprop)

Learning Rate	0.001	0.01	0.1
Loss(Catagorical - cross entropy)	3.07	15.36	11.84

MLP

Epochs number	1	5	10	...	20
Time(min)	36.5	182.5	420	...	1680
Loss(lr = 0.01)	4..25	3.89	2.98	...	2.11

LSTM

Epochs Number	1	5	10	...	20
Time(min)	63	315	630	...	2520
Loss(lr = 0.01)	3.52	3.01	2.90	...	3.02

References

Geman, D., Geman, S., Hallonquist, N. and Younes, L., 2015. Visual turing test for computer vision systems. Proceedings of the National Academy of Sciences, 112(12), pp.3618-3623.

Source：
https://zh.wikipedia.org/wiki/%E5%9B%BE%E7%81%B5%E6%B5%8B%E8%AF%95

http://www.bbc.co.uk/news/technology-27762088

http://www.robots.ox.ac.uk/~vgg/

https://github.com/VT-vision-lab

Appendix

Appendix 1: python files.

- extract_features.py
- fully_implement_vqa.py
- lstm_vqa_load_model_continue_train.py
- lstm_vqa.py
- mlp_vqa_load_model_continue_train.py
- mlp_vqa.py
- preprocess.py

Appendix 2:running screenshots

MLP

Learning rate = 0.01 epoch = 5 optimizer = sgd

```
214144/215519 [============================>.] - ETA: 13s - train loss: 2.9466
214272/215519 [============================>.] - ETA: 11s - train loss: 2.9466
214400/215519 [============================>.] - ETA: 10s - train loss: 2.9466
214528/215519 [============================>.] - ETA: 9s - train loss: 2.9463
214656/215519 [============================>.] - ETA: 8s - train loss: 2.9461
214784/215519 [============================>.] - ETA: 7s - train loss: 2.9462
214912/215519 [============================>.] - ETA: 5s - train loss: 2.9462
215040/215519 [============================>.] - ETA: 4s - train loss: 2.9459
215168/215519 [============================>.] - ETA: 3s - train loss: 2.9459
215296/215519 [============================>.] - ETA: 2s - train loss: 2.9459
215424/215519 [============================>.] - ETA: 0s - train loss: 2.9458
215552/215519 [==============================] - 2067s 10ms/step - train loss: 2.9456
```

Learning rate = 0.1 epoch = 5 optimizer = sgd

```
214528/215519 [============================>.] - ETA: 9s - train loss: 13.2021
214656/215519 [============================>.] - ETA: 8s - train loss: 13.2022
214784/215519 [============================>.] - ETA: 7s - train loss: 13.2024
214912/215519 [============================>.] - ETA: 6s - train loss: 13.2018
215040/215519 [============================>.] - ETA: 4s - train loss: 13.2007
215168/215519 [============================>.] - ETA: 3s - train loss: 13.2005
215296/215519 [============================>.] - ETA: 2s - train loss: 13.2011
215424/215519 [============================>.] - ETA: 0s - train loss: 13.2015
215552/215519 [==============================] - 2141s 10ms/step - train loss: 13.1996
```

LSTM

Learning rate = 0.001 epoch = 1 optimizer = rmsprop

```
213632/215375 [============================>.] - ETA: 32s - train loss: 3.5244
213760/215375 [============================>.] - ETA: 29s - train loss: 3.5241
213888/215375 [============================>.] - ETA: 27s - train loss: 3.5237
214016/215375 [============================>.] - ETA: 25s - train loss: 3.5236
214144/215375 [============================>.] - ETA: 22s - train loss: 3.5235
214272/215375 [============================>.] - ETA: 20s - train loss: 3.5234
214400/215375 [============================>.] - ETA: 17s - train loss: 3.5235
214528/215375 [============================>.] - ETA: 15s - train loss: 3.5233
214656/215375 [============================>.] - ETA: 13s - train loss: 3.5230
214784/215375 [============================>.] - ETA: 10s - train loss: 3.5229
214912/215375 [============================>.] - ETA: 8s - train loss: 3.5228
215040/215375 [============================>.] - ETA: 6s - train loss: 3.5227
215168/215375 [============================>.] - ETA: 3s - train loss: 3.5224
215296/215375 [============================>.] - ETA: 1s - train loss: 3.5220
215424/215375 [==============================] - 3973s - train loss: 3.5226
```

Learning rate = 0.001 epoch = 5 optimizer = rmsprop

```
202368/215375 [===========================>..] - ETA: 238s - train loss: 2.8975
203392/215375 [===========================>..] - ETA: 219s - train loss: 2.8976
204544/215375 [===========================>..] - ETA: 198s - train loss: 2.8974
205568/215375 [===========================>..] - ETA: 179s - train loss: 2.8974
206720/215375 [===========================>..] - ETA: 158s - train loss: 2.8973
207744/215375 [===========================>..] - ETA: 140s - train loss: 2.8983
208768/215375 [============================>.] - ETA: 121s - train loss: 2.8983
209920/215375 [============================>.] - ETA: 100s - train loss: 2.8976
210944/215375 [============================>.] - ETA: 81s - train loss: 2.8980
212096/215375 [============================>.] - ETA: 60s - train loss: 2.8974
213120/215375 [============================>.] - ETA: 41s - train loss: 2.8984
214272/215375 [============================>.] - ETA: 20s - train loss: 2.8988
215296/215375 [============================>.] - ETA: 1s - train loss: 2.8995
215424/215375 [==============================] - 3954s - train loss: 2.9000
```

Learning rate = 0.001 epoch = 10 optimizer = rmsprop

```
213888/215375 [============================>.] - ETA: 27s - train loss: 2.9026
214016/215375 [============================>.] - ETA: 24s - train loss: 2.9027
214144/215375 [============================>.] - ETA: 22s - train loss: 2.9029
214272/215375 [============================>.] - ETA: 20s - train loss: 2.9030
214400/215375 [============================>.] - ETA: 17s - train loss: 2.9034
214528/215375 [============================>.] - ETA: 15s - train loss: 2.9034
214656/215375 [============================>.] - ETA: 13s - train loss: 2.9035
214784/215375 [============================>.] - ETA: 10s - train loss: 2.9036
214912/215375 [============================>.] - ETA: 8s - train loss: 2.9037
215040/215375 [============================>.] - ETA: 6s - train loss: 2.9038
215168/215375 [============================>.] - ETA: 3s - train loss: 2.9038
215296/215375 [============================>.] - ETA: 1s - train loss: 2.9036
215424/215375 [==============================] - 3943s - train loss: 2.9042
```

Learning rate = 0.001 epoch = 20 optimizer = rmsprop

```
213504/215375 [============================>.] - ETA: 33s - train loss: 2.9552
213632/215375 [============================>.] - ETA: 31s - train loss: 2.9553
213760/215375 [============================>.] - ETA: 29s - train loss: 2.9553
213888/215375 [============================>.] - ETA: 26s - train loss: 2.9552
214016/215375 [============================>.] - ETA: 24s - train loss: 2.9553
214144/215375 [============================>.] - ETA: 22s - train loss: 2.9556
214272/215375 [============================>.] - ETA: 20s - train loss: 2.9557
214400/215375 [============================>.] - ETA: 17s - train loss: 2.9562
214528/215375 [============================>.] - ETA: 15s - train loss: 2.9562
214656/215375 [============================>.] - ETA: 13s - train loss: 2.9562
214784/215375 [============================>.] - ETA: 10s - train loss: 2.9564
214912/215375 [============================>.] - ETA: 8s - train loss: 2.9564
215040/215375 [============================>.] - ETA: 6s - train loss: 2.9567
215168/215375 [============================>.] - ETA: 3s - train loss: 2.9566
215296/215375 [============================>.] - ETA: 1s - train loss: 2.9564
215424/215375 [==============================] - 3908s - train loss: 2.9571
```

LSTM - Verify

Epoch = 1

```
Loaded
  32/1000 [..............................] - ETA: 3s
  64/1000 [>.............................] - ETA: 2s
  96/1000 [=>............................] - ETA: 2s
 128/1000 [==>...........................] - ETA: 2s
 160/1000 [===>..........................] - ETA: 2s
 192/1000 [====>.........................] - ETA: 1s
 224/1000 [=====>........................] - ETA: 1s
 256/1000 [======>.......................] - ETA: 1s
 320/1000 [========>.....................] - ETA: 1s
 352/1000 [=========>....................] - ETA: 1s
 384/1000 [==========>...................] - ETA: 1s
 416/1000 [===========>..................] - ETA: 1s
 448/1000 [============>.................] - ETA: 1s
 480/1000 [=============>................] - ETA: 1s
 512/1000 [==============>...............] - ETA: 1s
 544/1000 [===============>..............] - ETA: 1s
 576/1000 [================>.............] - ETA: 0s
 608/1000 [=================>............] - ETA: 0s
 640/1000 [==================>...........] - ETA: 0s
 672/1000 [===================>..........] - ETA: 0s
 704/1000 [====================>.........] - ETA: 0s
 736/1000 [=====================>........] - ETA: 0s
 800/1000 [=======================>......] - ETA: 0s
 832/1000 [=======================>......] - ETA: 0s
 864/1000 [========================>.....] - ETA: 0s
 896/1000 [=========================>....] - ETA: 0s
 928/1000 [==========================>...] - ETA: 0s
 960/1000 [===========================>..] - ETA: 0s
 992/1000 [============================>.] - ETA: 0sloss is 3.90891549873 accuracy is 0.282
```

Epoch = 20

```
Loaded
  32/1000 [..............................] - ETA: 3s
  64/1000 [>.............................] - ETA: 2s
  96/1000 [=>............................] - ETA: 2s
 128/1000 [==>...........................] - ETA: 1s
 224/1000 [=====>........................] - ETA: 1s
 256/1000 [======>.......................] - ETA: 1s
 288/1000 [=======>......................] - ETA: 1s
 320/1000 [========>.....................] - ETA: 1s
 352/1000 [=========>....................] - ETA: 1s
 384/1000 [==========>...................] - ETA: 1s
 416/1000 [===========>..................] - ETA: 1s
 448/1000 [============>.................] - ETA: 1s
 480/1000 [=============>................] - ETA: 1s
 512/1000 [==============>...............] - ETA: 1s
 544/1000 [===============>..............] - ETA: 0s
 576/1000 [================>.............] - ETA: 0s
 608/1000 [=================>............] - ETA: 0s
 640/1000 [==================>...........] - ETA: 0s
 704/1000 [====================>.........] - ETA: 0s
 736/1000 [=====================>........] - ETA: 0s
 768/1000 [======================>.......] - ETA: 0s
 800/1000 [=======================>......] - ETA: 0s
 832/1000 [=======================>......] - ETA: 0s
 864/1000 [========================>.....] - ETA: 0s
 896/1000 [=========================>....] - ETA: 0s
 928/1000 [==========================>...] - ETA: 0s
 960/1000 [===========================>..] - ETA: 0s
 992/1000 [============================>.] - ETA: 0sloss is 3.14102084351 accuracy is 0.299
```