Gregory Brown https://github.com/toadSTL/CS5542_Lab
Class ID: 4
Big Data Analytics and Applications
CS 5542

Lab 4

1. VQA

Visual Question Answering models are similar to Image Captioning models in many ways: both use image features obtained using a CNN, both use an RNN for language embedding, and fundamentally both seek to provide a user with information about a scene. However where Image Captioning models use the trained RNN (together with the image features) in order to generate a natural language description of the image word-by-word, Visual Question Answering Models use the trained RNN (together with the image features) to first interpret the question and then generate an answer (in some sense the VQA RNN 'finishes' the provided questions with answers). Ultimately our team has successfully implemented a functioning VQA system based on , however our accuracy is low.

2. Method

2.1 Dataset and Data Pre-processing

We have changed our datasets; it is almost the same set of images but a totally different set of questions. Previously we were using the COCO QA questions together with the MSCOCO 2017 training images. Currently we are using the VQA dataset together with the MSCOCO 2014 training images, which is a subset of the 2017 training image. Where the COCO QA dataset had questions, answers, and image Ids in text, the VQA dataset stores this information in json format. Additionally, as before, we are filtering by the MSCOCO captions as discussed thoroughly in Lab 1, mainly in order to reduce the size of the dataset. Our filtering is updated to only output the image ids of images associated with relevant questions (i.e. questions relevant to specified, park-related keywords) which enables easy extraction of the relevant questions and answers as well as construction of a 'captions' file, which is merely a json format list of associated image ids, image filenames, image heights and widths (the original 'captions' file also contains license information associated with the MS COCO dataset on a per-image bases as well as flicker urls for each image, however that information is not readily accessible for the images we are using, and thus

is not included in our ‘captions’ json file). The following is the entirety of our question filtering script:

```
import json
input_file = open ('Data/v2_OpenEnded_mscoco_train2014_questions.json')
json_array = json.load(input_file)
store_list = []
ids = []
# 'Data/imgIds_train.txt'
# 'Data/imgIds_val.txt'
filepath = 'Data/imgIds_train.txt'
with open(filepath, "r") as ins:
   array = []
   for line in ins:
       ids.append(line.rstrip('\n'))
cnt = 0
for item in json_array['questions']:
   store_details = {"image_id":None, "question":None, "question_id":None}
   store_details['image_id'] = int(item['image_id'])
   store_details['question'] = item['question']
   store_details['question_id'] = item['question_id']
   if store_details['image_id'] in ids:
       store_list.append(store_details)
       cnt = cnt + 1
#print(store_list)
# 'Data/questions_train.json'
# 'Data/questions_val.json'
with open('Data/questions_train.json', 'w') as outfile:
   json.dump(store_list, outfile)
print(cnt)
```

This script loads the entire VQA Open ended question set from MS COCO as well as the filtered list of image ids and populates a list of associated image ids, questions, and question ids for the image ids in the filtered list. Finally this list of associated information is output. Currently, this script must be run separately for training and validation datasets and will output ‘questions_train.json’ and ‘questions_val.json’. These files will be used by our data_loader.py along with the answers (annotations json files) in order to prepare our vocabulary and perform word embedding on the question and answers. The script which filters the answers, reproduced below, is very similar to question pre-processing script:

```
import json
input_file = open ('Data/v2_mscoco_train2014_annotations.json')
json_array = json.load(input_file)
store_list = []
ids = []
# 'Data/imgIds_train.txt'
# 'Data/imgIds_val.txt'
filepath = 'Data/imgIds_train.txt'
```

```
with open(filepath, "r") as ins:
   array = []
   for line in ins:
       ids.append(line.rstrip('\n'))
cnt = 0
for item in json_array['annotations']:
   if str(item['image_id']) in ids:
       store_list.append(item)
       cnt = cnt + 1
#print(store_list)
# 'Data/annotations_train.json'
# 'Data/annotations_val.json'
with open('Data/annotations_train.json', 'w') as outfile:
   json.dump(store_list, outfile)
print(cnt)
```

The main difference is that in this script the entire json entry associated with each image id from our filtered lists of image ids, referred to within VQA dataset as annotations, and saves the resulting list of annotations. Each annotation includes a question id and a list of answer entries, each including an answer, answer confidence, and answer id.

In addition to filtering our questions and answers we must make the 'captions' json file. Below is the script I have used for this task:

```
import json
import imageio
ids = []
# 'Data/imgIds_train.txt'
# 'Data/imgIds_val.txt'
filepath = 'Data/imgIds_val.txt'
with open(filepath, "r") as ins:
   array = []
   for line in ins:
       ids.append(line.rstrip('\n'))
store_list = []
for id in ids:
   store_details = {"file_name": None, "width": None, "height": None, "id": None}
   store_details['id'] = int(id)
   idstr = "{0:0>12}".format(id)
   file_name =  "COCO_train2014_"+idstr+".jpg"
   store_details['file_name'] = file_name
   # "Data/val2014/
   img = imageio.imread("Data/train2014/"+file_name).shape
   # print(img)
   height = img[0]
   width = img[1]
   store_details['height'] = height
   store_details['width'] = width
   store_list.append(store_details)
```

```
print(len(ids))
#print(store_list)
# 'Data/captions_train.json'
# 'Data/captions_val.json'
with open('Data/captions_val.json', 'w') as outfile:
   json.dump(store_list, outfile)
```

After reverse-engineering the filename from the image ids in our filtered lists of image ids, this script loads each image in order to obtain the image dimensions and saves the associated filenames, widths, heights, and ids into captions_train.json and captions_val.json. Similar to the other preprocessing scripts we currently run this separately for the training and validation image sets.

Unlike with the previous 3 scripts which can be run in any order data_loader.py is our final preprocessing script which must be run afterward. I can be run using the following command:

```
python data_loader.py --version=1
```

Here version 1 refers to the VQA datasets open ended (rather than multiple choice) questions, as those the questions for which we have created these pre-processed datafiles. Within our code we provide the locations of these datafiles:

```
if version == 1:
   print("Version 1")
   t_q_json_file = join(data_dir, 'questions_train.json')
   t_a_json_file = join(data_dir, 'annotations_train.json')

   v_q_json_file = join(data_dir, 'questions_val.json')
   v_a_json_file = join(data_dir, 'annotations_val.json')
   qa_data_file = join(data_dir, 'qa_data_file1.json')
   vocab_file = join(data_dir, 'vocab_file1.json')
else:
...
```

As shown, we also specify at this point the output files for the for the embedded question answer data and the vocab, or word embedding. After loading these json files the training questions and training answers are combined by id and the same is done for validation question and validation answers. Next the answer and question vocabularies are each generated using `get_top_answers` and `make_question_vocab` functions on the training data. Defined within data_loader.py, each of these functions associates indices to answer words and question words respectively. Finally, using the `process_data` function, again defined within data_loader.py, these vocabs are used to

produce embeddings of questions and answers for both training and validation data. These embeddings are then saved in the specified output files. Further code snippets from this program have been omitted for the sake of brevity. The embeddings produced by this program will be used by the LSTM.

2.2 Pre-trained Models and Feature Extraction

Our VQA system can extract image features using one of two pre-trained CNN models: resnet and VGG16. We would ultimately like to test image feature extraction using additional pre-trained models, such as inception, but may not have time to complete with our final submission. Additionally, in theory we can specify what layers of the pre-trained CNN models we would like to get image features from, however thus far we have only successfully tested 'block4' of resnet and 'pool5' of VGG16. Here are links to download resnet and vgg16. Resnet, the more recent of these two CNN models, is organized in blocks, specifically 4 of them, and we pull features from the 4th block or 'block4'. VGG16 is has a more traditional CNN architecture, and we pull features from the 5th max pooling layer or 'pool5'. Our feature extraction is done using our extract_conv_features.py program which can be run using the following command lines:

```
python extract_conv_features.py --feature_layer="pool5" --model="vgg" --split="train"
--batch_size=16

python extract_conv_features.py --feature_layer="pool5" --model="vgg" --split="val"
--batch_size=16

python extract_conv_features.py --feature_layer="block4" --split="train"

python extract_conv_features.py --feature_layer="block4" --split="val"
```

This program, extrac_conv_features.py, loads the data about the images for which we want features from the captions_train.json and captions_val.json, the creation of which is discussed in section 2.1. Using the image information an image id list is constructed. Then a CNN model is created based on either vgg, or resnet (resnet is default). Next the shape of the features we wish to extract is established and a dataset is created to house the extracted features based on that feature shape. The below redacted snippet shows the code for these steps:

```
if args.split == "train":
   with open('Data/captions_train.json') as f:
       images = json.loads(f.read())['images']
else:
   with open('Data/captions_val.json') as f:
```

```
        images = json.loads(f.read())['images']

image_ids = {image['id'] : 1 for image in images}
image_id_list = [img_id for img_id in image_ids]

...

if args.model=="vgg":
    cnn_model = vgg16.create_vgg_model(448, only_conv)
else:
    cnn_model = resnet.create_resnet_model(448)

...

    if args.model=="vgg":
        conv_features = None
        feature_shape =  (len(image_id_list), 14, 14, 512)
        img_dim = 448
    else:
        conv_features = None
        feature_shape =  (len(image_id_list), 14*14*2048)
        img_dim = 448

hdf5_data = hdf5_conv_file.create_dataset('conv_features', feature_shape,
                                          dtype='f')
```

The next section of code performs the feature extraction, pulling the desired features from the CNN model. Each iteration of this while loop will pull features for a batch of images, until all images have been pulled. Feature extraction is done in batches to reduce system load, and the default batch size of 64 can be specified by the user as a command line argument (see commands to run feature extraction prior). First the image batch is collected, then each image in the batch is loaded, and ultimately winds up in the array image_batch. Then the image features for that batch of images loaded from the CNN model. Once all features for all images have been loaded the model is saved. This model will be used

```
sess = cnn_model['session']
images = cnn_model['images_placeholder']
image_feature_layer = cnn_model[args.feature_layer]

idx = 0
while idx < len(image_id_list):
    start = time.clock()
    image_batch = np.ndarray( (args.batch_size, img_dim, img_dim, 3 ) )
    count = 0
    for i in range(0, args.batch_size):
        if idx >= len(image_id_list):
            break
```

```
        image_file = join('Data',
'train2014/COCO_train2014_%.12d.jpg'%(image_id_list[idx]))#'%s2014/COCO_%s2014_%.12d.jpg'%(a
rgs.split, args.split, image_id_list[idx]) )

        if args.model == 'resnet':
            image_array = sess.run(cnn_model['processed_image'], feed_dict = {
                cnn_model['pre_image'] : utils.load_image_array(image_file, img_dim = None)
                })
        else:
            image_array = utils.load_image_array(image_file, img_dim = img_dim)

        image_batch[i,:,:,:] = image_array
        idx += 1
        count += 1

    feed_dict  = { images : image_batch[0:count,:,:,:] }
    conv_features_batch = sess.run(image_feature_layer, feed_dict = feed_dict)
    # Below line is required for 'block4'
    if(args.feature_layer == 'block4'):
        conv_features_batch = np.reshape(conv_features_batch, ( conv_features_batch.shape[0],
-1 ))
    hdf5_data[(idx - count):idx] = conv_features_batch[0:count]

    end = time.clock()
    print("Time for batch of photos", end - start)
    print("Hours Remaining" , ((len(image_id_list) - idx) * 1.0)*(end -
start)/60.0/60.0/args.batch_size)
    print("Images Processed", idx)

hdf5_conv_file.close()
```

Similar to the pre-processing programs, this program must be run separately for the training and test sets and also separately for each CNN model. At this point we have the extracted image features for the same set of images for which we have the embedded question and answers and the embedding vocabulary. Using these image features and question-answer embedding we will train our model.

2.3 Training and Evaluation

We use the following command to train VQA models:

```
python train_evaluate.py --version=1
python train_evaluate.py --version=1 --feature_layer="pool5" --
cnn_model="vgg"
```

Again, here version 1 refers to the open ended questions. Our program, train_evaluate.py by default uses block4 of resnet, but can be made to train using the image features from pool5 of vgg16 by command line arguments as shown. First train_evaluate.py loads question and answer embeddings. The metadata, or features of the loaded data are used as model options to define the initial VQA model. The creation of this initial VQA model is handled by VQA_model_attention.py. As this initial VQA model is constructed the embedded question and answer data are encoded using the ByteNet LSTM (though by command line argument, again, our program could handle the construction of a new LSTM). Additionally constructing the initial VQA model involves constructing initial image attention maps which are constructed probabilistically using encoded question and answers and the image features. After the model is initialized training is performed via the following loop:

```
for epoch in range(args.epochs):
   batch_no = 0
   while (batch_no*args.batch_size) < len(qa_data['training']):
       start = time.clock()
       question, answer, image_features, image_ids, _ = get_batch(
           batch_no, args.batch_size, qa_data['training'],
           conv_features, image_id_map, 'train', model_options
           )

       _, loss_value = sess.run([train_op, model.loss],
           feed_dict={
               model.question: question,
               model.image_features: image_features,
               model.answers: answer
           }
       )
       end = time.clock()

       ...

       if step % args.sample_every == 0:

           ...

       if step % args.evaluate_every == 0:

           ...

       if step >= args.max_steps:
           break
```

Notably this code brings up the hyperparameters which we are still tuning in hopes of improving the resulting question answering models. Specifically,

earlier in the code we specify the desired number of epochs (or times which the whole dataset will be used to update the model weights), the maximum number of steps (where at each step training is performed using one batch of images, or 64 images). Additionally there are two hyperparameters for output to the user while training progresses, one for sampling i.e. outputting sample image question inputs along with actual and predicted answers, and one for evaluation i.e. determining the accuracy of the model on the provided validation set (the model is only training using the training data). Notably, currently models are saved at evaluation time. As shown, one we have performed the desired number of epochs of training, or once we have reached the maximum number of steps, the training ceases.

We intend to perform further evaluation using the program generate.py, however this is still in development. The goal of this evaluation would be to output a manually verifiable json of image filenames, questions, ground-truth answers, and predicted answers along with the accuracy. Since we do get a validation accuracy from running train_evaluate.py, for this submission, we consider that accuracy result to be sufficient.

3. Results

After the processes initiated by each the command lines from 2.3 resolve, we are left with the final models produced using each CNN model, as well as training logs and sampled predictions. Below are the set of hyperparameters used to obtain the results which follow:

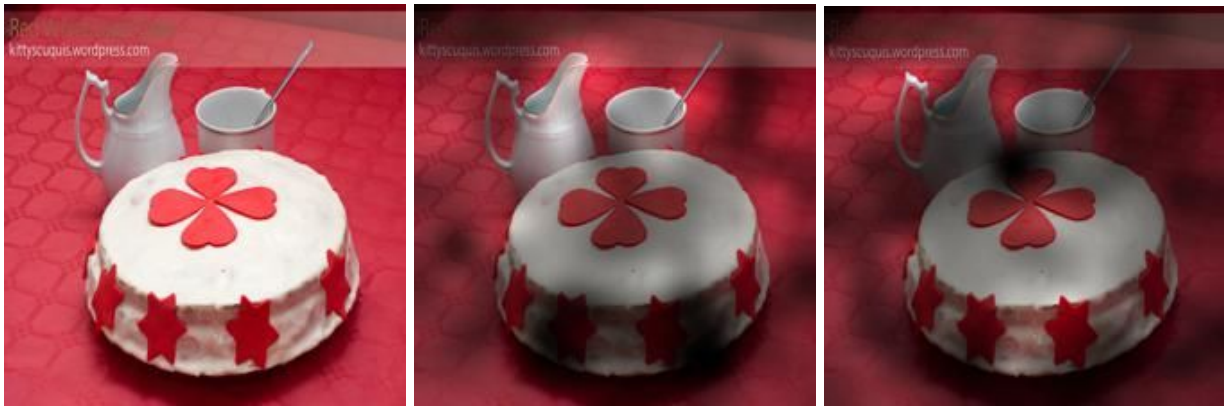
Hyperparameters					
Batch Size	Epochs	Sample Every (# steps)	Evaluate Every	Learning Rate	Max Steps
64	200	33	99	0.005	20000

While these hyperparameters are specified within the code I am also including them in this table for readability. Additionally cnn_model and feature_layer hyperparameters are by default set to ‘resnet’ and ‘block4’, and specifying them we have done produces an additional model. Hence we have produced two models, which we will refer to by the source of each model’s image features i.e. the resnet model and the vgg16 model. While they are separate models their accuracies are very similar:

Model	Validation Accuracy @ epoch 0	Validation Accuracy @ epoch 199
resnet	0.30817205613659276	0.3643010490171371
vgg16	0.29075270129788305	0.36688173355594755

Notably these accuracies are slightly improved over accurices of the same model trained for 100 epoch with learning rate 0.01 (accuracies for those models grew from ~0.295 to ~0.356). Also, I tested produced some additional samples using generate.py for this report, hoping to find differences between the answers produced by each model but they were identical:

Image_id: 193386
Question: What color is the cat?
Ground-truth-answer: ‘black and white’ or ‘black white’
Predicted-answer: ‘black’



Image_id: 387102
Question: "What is the round object?"
Ground-truth-answer: ‘cake’
Predicted-answer: ‘flower’

Image_id:
Question: "How many metal rods are holding this umbrella?
Ground-truth-answer: 8 or 12
Predicted Answer: 8

Each model makes the same partial-error of not returning both black and white for the color of the cat, and the same explicit-error of labeling the cake as a ‘flower’. This may be because of a skew in the dataset, I have attempted to correct this skew in later datasets, however, for the models produced for this report there are 773 images of flowers within the 1541 total images. Below is a table of statistics about the data used to produce these models:

	Images	Question-Answer
Training	1233	6287
Validation	308	1550

The embedding and encoding of these question-answers produced a question vocab size of 2354, and an answer vocab size of 1067.

In future work the dataset I will be using a less skewed dataset with the following statistics:

	Images	Question-Answers
Training	690	3448
Validation	181	898