CEC21: Project Report

Image Caption Generator

—

Ishank Jain, 2017UCO1546
Kartikeya Khullar, 2017UCO1568

Index

OBJECTIVE

Image Caption Generator is basically an application that will take an image as input and will provide the most suitable caption for that image as output. Such an application will be very beneficial in various real-life applications like:

- Self-driving cars — Automatic driving is one of the biggest challenges and if we can properly caption the scene around the car, it can give a boost to the self-driving system.
- Aid to the blind — We can create a product for the blind which will guide them traveling on the roads without the support of anyone else. We can do this by first converting the scene into text and then the text to voice. Both are now famous applications of Deep Learning.
- CCTV cameras are everywhere today, but along with viewing the world, if we can also generate relevant captions, then we can raise alarms as soon as there is some malicious activity going on somewhere. This could probably help reduce some crime and/or accidents.
- Automatic Captioning can help, make Google Image Search as good as Google Search, as then every image could be first converted into a caption, and then the search can be performed based on the caption.

We aim to use Deep learning concepts like Convolutional Neural Networks, Recurrent Neural Networks, Text Processing, etc. to implement this task.

Open Source Technologies Used

1) **Keras:** **Keras** is an open-source neural network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. Keras contains numerous implementations of commonly used neural-network building blocks such as layers, objectives, activation functions, optimizers, and a host of tools to make working with image and text data easier to simplify the coding necessary for writing deep neural network code. In addition to standard neural networks, Keras has support for convolutional and recurrent neural networks. It supports other common utility layers like dropout, batch normalization, and pooling.

2) **TensorFlow**: **TensorFlow** is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library and is also used for machine learning applications such as neural networks. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML, and developers easily build and deploy ML-powered applications.

3) **NumPy**: **NumPy** is an open-source library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

4) **NLTK: The Natural Language Toolkit,** or more commonly NLTK, is a suite of libraries and programs for symbolic and statistical natural language processing (NLP) for English written in the Python programming language. NLTK includes graphical demonstrations and sample data. NLTK supports classification, tokenization, stemming, tagging, parsing, and semantic reasoning functionalities.

Dataset Used

There are many open-source datasets available, like Flickr 8k (containing8k images), Flickr 30k (containing 30k images), MS COCO (containing 180k images), etc.

Training a model with a large number of images was not feasible for us due to the unavailability of high-end GPU on our laptops. Therefore, we used Flickr 8k dataset provided by the University of Illinois at Urbana-Champaign. This dataset contains 8000 images each with 5 captions (image can have multiple captions, all being relevant simultaneously).

These images are bifurcated as follows:

- Training Set — 6000 images
- Dev Set — 1000 images
- Test Set — 1000 images

Methodology

1) Understanding the data

The dataset contains:
-> 1 folder containing 8000 images, and
-> a text file containing 5 captions for every image.

Every line in the text file contains the <image name>#i <caption>, where $0 \leq i \leq 4$ i.e. the name of the image, caption number (0 to 4), and the actual caption.
Example:
101654506_8eb26cfb60.jpg#0 A brown and white dog is running through the Snow.

Now, we create a dictionary named "captions" which contains the name of the image (without the .jpg extension) as keys and a list of the 5 captions for the corresponding image as values.

Code:

```
def load_captions(filename):
   file = open(filename, 'r')
   doc = file.read()
   file.close()
   """
   Captions dict is of form:
   {
       image_id1 : [caption1, caption2, etc],
       image_id2 : [caption1, caption2, etc],
       ...
   }
   """
   captions = dict()
   # Process lines by line
   _count = 0
   for line in doc.split('\n'):
```

```
        # Split line on white space
        tokens = line.split()
        if len(line) < 2:
            continue
        # Take the first token as the image id, the rest as the
caption
        image_id, image_caption = tokens[0], tokens[1:]
        # Extract filename from image id
        image_id = image_id.split('.')[0]
        # Convert caption tokens back to caption string
        image_caption = ' '.join(image_caption)
        # Create the list if needed
        if image_id not in captions:
            captions[image_id] = list()
        # Store caption
        captions[image_id].append(image_caption)
        _count = _count+1
    print('{}: Parsed captions: {}'.format(mytime(),_count))
    return captions
```

2) Data Cleaning

->lower-casing all the words (otherwise“hello” and “Hello” will be regarded as two separate words)
-> removing special tokens (like ‘%’, ‘$’, ‘#’, etc.)
->eliminating words which contain numbers (like ‘hey199’, etc.)

Code:

```
def clean_captions(captions):
    # Prepare translation table for removing punctuation
    table = str.maketrans('', '', string.punctuation)
    for _, caption_list in captions.items():
        for i in range(len(caption_list)):
            caption = caption_list[i]
```

```
            # Tokenize i.e. split on white spaces
            caption = caption.split()
            # Convert to lowercase
            caption = [word.lower() for word in caption]
            # Remove punctuation from each token
            caption = [w.translate(table) for w in caption]
            # Remove hanging 's' and 'a'
            caption = [word for word in caption if len(word)>1]
            # Remove tokens with numbers in them
            caption = [word for word in caption if word.isalpha()]
            # Store as string
            caption_list[i] =  ' '.join(caption)
```

3) Data Preprocessing(Captions)

a) The model we'll develop will generate a caption for a given image and the caption will be generated one word at a time. The sequence of previously generated words will be provided as input. Therefore, we will need a 'first word' to kick-off the generation process and a 'last word' to signal the end of the caption. We'll use the strings 'startseq' and 'endseq' for this purpose. These tokens are added to the captions as they are loaded. It is important to do this now before we encode the text so that the tokens are also encoded correctly.

Code:

```
def load_cleaned_captions(filename, ids):
   file = open(filename, 'r')
   doc = file.read()
   file.close()
   captions = dict()
   _count = 0
   # Process line by line
   for line in doc.split('\n'):
```

```
        # Split line on white space
        tokens = line.split()
        # Split id from caption
        image_id, image_caption = tokens[0], tokens[1:]
        # Skip images not in the ids set
        if image_id in ids:
            # Create list
            if image_id not in captions:
                captions[image_id] = list()
            # Wrap caption in start & end tokens
            caption = 'startseq ' + ' '.join(image_caption) + '
endseq'
            # Store
            captions[image_id].append(caption)
            _count = _count+1
    return captions, _count
```

b) The captions will need to be encoded to numbers before it can be presented to the model. The first step in encoding the captions is to create a consistent mapping from words to unique integer values. Keras provides the Tokenizer class that can learn this mapping from the loaded captions.

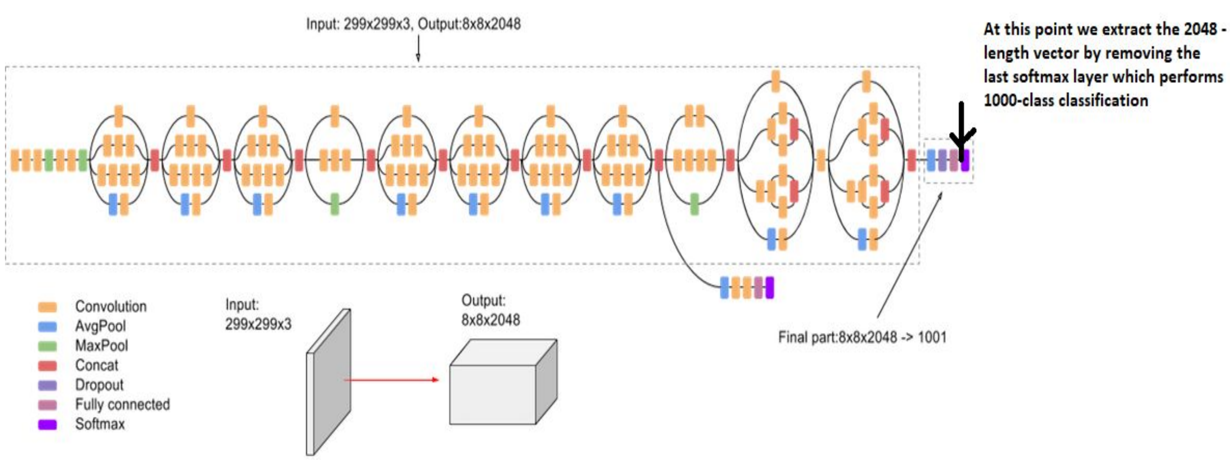
Fit a tokenizer on given captions:

```
def create_tokenizer(captions):
    lines = to_lines(captions)
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# Calculate the length of the captions with the most words
def calc_max_length(captions):
    lines = to_lines(captions)
    return max(len(line.split()) for line in lines)
```

4) Data Preprocessing(Images)

a) **Extract Features:** First, we need to convert every image into a fixed-sized vector which can then be fed as input to the neural network. For this purpose, we used for transfer learning by using the InceptionV3 model (Convolutional Neural Network) created by Google Research. This model was trained on the Imagenet dataset to perform image classification in 1000 different classes of images. However, our purpose here is not to classify the image but just get a fixed-length informative vector for each image. This process is called automatic feature engineering. Hence, we just remove the last softmax layer from the model and extract a 2048 length vector (bottleneck features) for every image as follows:



Code

```
def CNNModel(model_type):
   if model_type == 'inceptionv3':
       model = InceptionV3()
   elif model_type == 'vgg16':
       model = VGG16()
   model.layers.pop()
   model = Model(inputs=model.inputs,
outputs=model.layers[-1].output)
```

```
    return model

"""
    *This function returns a dictionary of the form:
    {
        image_id1 : image_features1,
        image_id2 : image_features2,
        ...
    }
"""
def extract_features(path, model_type):
    if model_type == 'inceptionv3':
        from keras.applications.inception_v3 import preprocess_input
        target_size = (299, 299)
    elif model_type == 'vgg16':
        from keras.applications.vgg16 import preprocess_input
        target_size = (224, 224)
    # Get CNN Model from model.py
    model = CNNModel(model_type)
    features = dict()
    # Extract features from each photo
    for name in tqdm(os.listdir(path)):
        # Loading and resizing image
        filename = path + name
        image = load_img(filename, target_size=target_size)
        # Convert the image pixels to a numpy array
        image = img_to_array(image)
        # Reshape data for the model
        image = image.reshape((1, image.shape[0], image.shape[1],
image.shape[2]))
        # Prepare the image for the CNN Model model
        image = preprocess_input(image)
        # Pass image into model to get encoded features
        feature = model.predict(image, verbose=0)
        # Store encoded features for the image
```

```
    image_id = name.split('.')[0]
    features[image_id] = feature
return features
```

5) Data Preparation using Generator Function

Each caption will be split into words. The model will be provided one word & the image and it generates the next word. Then the first two words of the caption will be provided to the model as input with the image to generate the next word. This is how the model will be trained.

For example, the input sequence “little girl running in field” would be split into 6 input-output pairs to train the model:

X1	X2(text sequence)	y(word)
image	startseq,	little
image	startseq, little,	girl
image	startseq, little, girl,	running
image	startseq, little, girl, running,	in
image	startseq, little, girl, running, in,	field
image	startseq, little, girl, running, in, field,	endseq

Code:

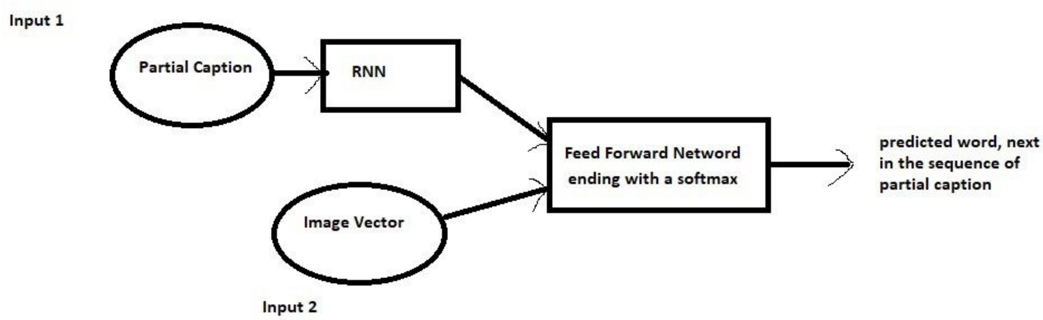
```
# Create sequences of images, input sequences and output words for an
image
def create_sequences(tokenizer, max_length, captions_list, image):
    # X1 : input for image features
    # X2 : input for text features
    # y  : output word
    X1, X2, y = list(), list(), list()
    vocab_size = len(tokenizer.word_index) + 1
    # Walk through each caption for the image
    for caption in captions_list:
        # Encode the sequence
        seq = tokenizer.texts_to_sequences([caption])[0]
        # Split one sequence into multiple X,y pairs
        for i in range(1, len(seq)):
            # Split into input and output pair
            in_seq, out_seq = seq[:i], seq[i]
            # Pad input sequence
            in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
            # Encode output sequence
            out_seq = to_categorical([out_seq],
num_classes=vocab_size)[0]
            # Store
            X1.append(image)
            X2.append(in_seq)
            y.append(out_seq)
    return X1, X2, y

# Data generator, intended to be used in a call to
model.fit_generator()
def data_generator(images, captions, tokenizer, max_length,
batch_size, random_seed):
    # Setting random seed for reproducibility of results
    random.seed(random_seed)
    # Image ids
    image_ids = list(captions.keys())
```

```
    _count=0
    assert batch_size<= len(image_ids), 'Batch size must be less than 
or equal to {}'.format(len(image_ids))
    while True:
        if _count >= len(image_ids):
            # Generator exceeded or reached the end so restart it
            _count = 0
        # Batch list to store data
        input_img_batch, input_sequence_batch, output_word_batch = 
list(), list(), list()
        for i in range(_count, min(len(image_ids), 
_count+batch_size)):
            # Retrieve the image id
            image_id = image_ids[i]
            # Retrieve the image features
            image = images[image_id][0]
            # Retrieve the captions list
            captions_list = captions[image_id]
            # Shuffle captions list
            random.shuffle(captions_list)
            input_img, input_sequence, output_word = 
create_sequences(tokenizer, max_length, captions_list, image)
            # Add to batch
            for j in range(len(input_img)):
                input_img_batch.append(input_img[j])
                input_sequence_batch.append(input_sequence[j])
                output_word_batch.append(output_word[j])
        _count = _count + batch_size
        yield [[np.array(input_img_batch), 
np.array(input_sequence_batch)], np.array(output_word_batch)]
```

6) Model Architecture

Since the input consists of two parts, an image vector, and a partial caption, we cannot use the Sequential API provided by the Keras library. For this reason, we use the Functional API which allows us to create Merge Models.



Code:

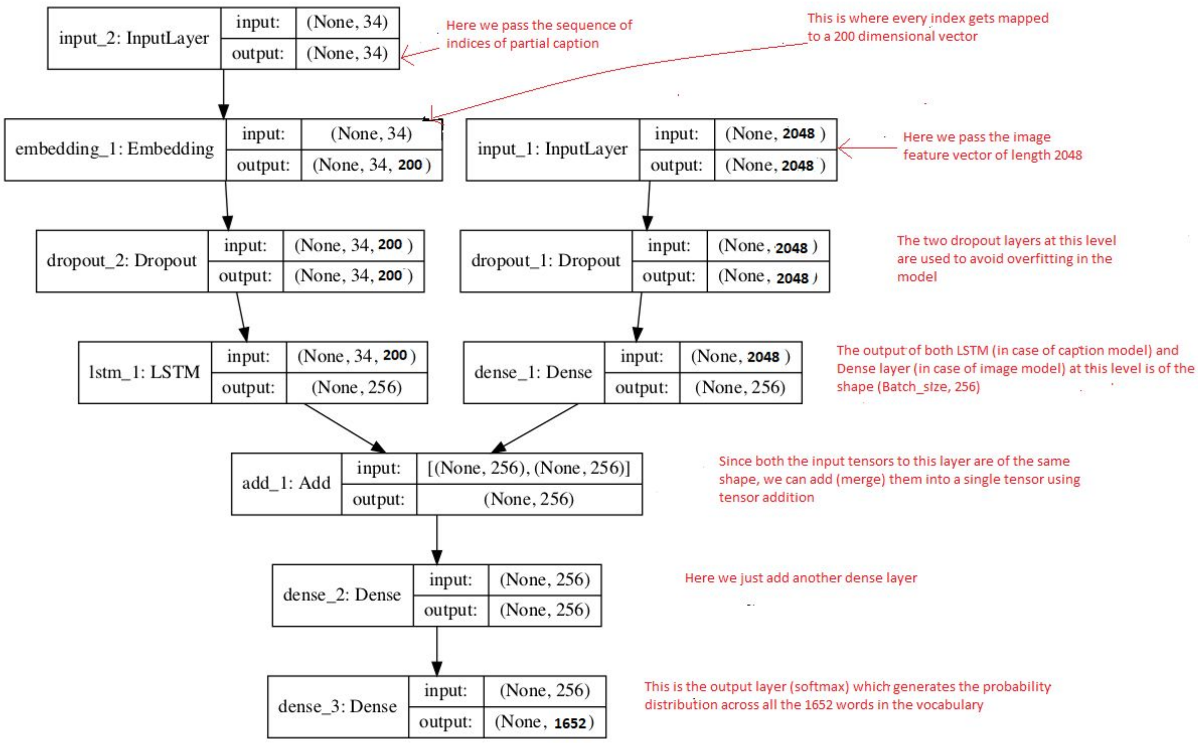
```
def RNNModel(vocab_size, max_len, rnnConfig, model_type):
   embedding_size = rnnConfig['embedding_size']
   if model_type == 'inceptionv3':
       # InceptionV3 outputs a 2048 dimensional vector for each
image, which we'll feed to RNN Model
       image_input = Input(shape=(2048,))
   elif model_type == 'vgg16':
       # VGG16 outputs a 4096 dimensional vector for each image,
which we'll feed to RNN Model
       image_input = Input(shape=(4096,))
   image_model_1 = Dropout(rnnConfig['dropout'])(image_input)
   image_model = Dense(embedding_size,
activation='relu')(image_model_1)

   caption_input = Input(shape=(max_len,))
```

```
    # mask_zero: We zero pad inputs to the same length, the zero mask
ignores those inputs. E.g. it is an efficiency.
    caption_model_1 = Embedding(vocab_size, embedding_size,
mask_zero=True)(caption_input)
    caption_model_2 = Dropout(rnnConfig['dropout'])(caption_model_1)
    caption_model = LSTM(rnnConfig['LSTM_units'])(caption_model_2)

    # Merging the models and creating a softmax classifier
    final_model_1 = concatenate([image_model, caption_model])
    final_model_2 = Dense(rnnConfig['dense_units'],
activation='relu')(final_model_1)
    final_model = Dense(vocab_size,
activation='softmax')(final_model_2)

    model = Model(inputs=[image_input, caption_input],
outputs=final_model)
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    return model
```



7) Training the Model

-> Now we have the input data and the model architecture.

-> The data is fed into the model, hyperparameters are set, and the model is trained.

```
config = {
   'images_path': 'train_val_data/Flicker8k_Dataset/',
'train_data_path': 'train_val_data/Flickr_8k.trainImages.txt',
   'val_data_path': 'train_val_data/Flickr_8k.devImages.txt',
   'captions_path': 'train_val_data/Flickr8k.token.txt',
   'tokenizer_path': 'model_data/tokenizer.pkl',
   'model_data_path': 'model_data/',
   'model_load_path':
'model_data/model_inceptionv3_epoch-20_train_loss-2.4050_val_loss-3.0
527.hdf5',
   'num_of_epochs': 20,
   'max_length': 40,
```

```
   'batch_size': 64,
   'beam_search_k':3,
   'test_data_path': 'test_data/',
   'model_type': 'inceptionv3',
   'random_seed': 1035
}

rnnConfig = {
   'embedding_size': 300,
   'LSTM_units': 256,
   'dense_units': 256,
   'dropout': 0.3
}

X1train, X2train, max_length = loadTrainData(config)
X1val, X2val = loadValData(config)
"""
   *Load the tokenizer
"""
tokenizer = load(open(config['tokenizer_path'], 'rb'))
vocab_size = len(tokenizer.word_index) + 1
"""
   *Now that we have the image features from CNN model, we need to
feed them to a RNN Model.
   *Define the RNN model
"""
model = RNNModel(vocab_size, max_length, rnnConfig,
config['model_type'])
print('RNN Model (Decoder) Summary : ')
print(model.summary())
"""
   *Train the model save after each epoch
"""
num_of_epochs = config['num_of_epochs']
batch_size = config['batch_size']
```

```
steps_train = len(X2train)//batch_size
if len(X2train)%batch_size!=0:
   steps_train = steps_train+1
steps_val = len(X2val)//batch_size
if len(X2val)%batch_size!=0:
   steps_val = steps_val+1
model_save_path =
config['model_data_path']+"model_"+str(config['model_type'])+"_epoch-
{epoch:02d}_train_loss-{loss:.4f}_val_loss-{val_loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(model_save_path, monitor='val_loss',
verbose=1, save_best_only=True, mode='min')
callbacks = [checkpoint]
# Shuffle train data
ids_train = list(X2train.keys())
random.shuffle(ids_train)
X2train_shuffled = {_id: X2train[_id] for _id in ids_train}
X2train = X2train_shuffled

# Create the train data generator
generator_train = data_generator(X1train, X2train, tokenizer,
max_length, batch_size, config['random_seed'])
# Create the validation data generator
generator_val = data_generator(X1val, X2val, tokenizer, max_length,
batch_size, config['random_seed'])
model.fit_generator(generator_train,
           epochs=num_of_epochs,
           steps_per_epoch=steps_train,
           validation_data=generator_val,
           validation_steps=steps_val,
           callbacks=callbacks,
           verbose=1)
```

8) Generating Caption using a trained model

We use beam search algorithm to generate a most accurate caption for an input image using the trained model.

The beam search algorithm selects multiple alternatives for an input sequence at each timestep based on conditional probability. The number of multiple alternatives depends on a parameter called **Beam Width B.** At each time step, the beam search selects B number of best alternatives with the highest probability as the most likely possible choices for the time step.

Code:

```
"""
   *Generate a caption for an image, given a pre-trained model
and a tokenizer to map integer back to word
   *Uses BEAM Search algorithm
"""
def generate_caption_beam_search(model, tokenizer, image,
max_length, beam_index=3):
   # in_text --> [[idx,prob]] ;prob=0 initially
   in_text = [[tokenizer.texts_to_sequences(['startseq'])[0],
0.0]]
   while len(in_text[0][0]) < max_length:
       tempList = []
       for seq in in_text:
           padded_seq = pad_sequences([seq[0]],
maxlen=max_length)
           preds = model.predict([image,padded_seq], verbose=0)
           # Take top (i.e. which have highest probailities)
`beam_index` predictions
           top_preds = np.argsort(preds[0])[-beam_index:]
           # Getting the top `beam_index` predictions and
           for word in top_preds:
               next_seq, prob = seq[0][:], seq[1]
               next_seq.append(word)
               # Update probability
```

```
                prob += preds[0][word]
                # Append as input for generating the next word
                tempList.append([next_seq, prob])
        in_text = tempList
        # Sorting according to the probabilities
        in_text = sorted(in_text, reverse=False, key=lambda l:
l[1])
        # Take the top words
        in_text = in_text[-beam_index:]
    in_text = in_text[-1][0]
    final_caption_raw = [int_to_word(i,tokenizer) for i in
in_text]
    final_caption = []
    for word in final_caption_raw:
        if word=='endseq':
            break
        else:
            final_caption.append(word)
    final_caption.append('endseq')
    return ' '.join(final_caption)
```

9) Performance evaluation of the model

- **Parameters on which the model is evaluated:**
    - **Cross entropy loss*(Lower the better)***

Categorical cross-entropy will compare the distribution of the predictions (the activations in the output layer, one for each class) with the true distribution, where the probability of the true class is set to 1 and 0 for the other classes. To put it in a different way, the true class is represented as a one-hot encoded vector, and the closer the model's outputs are to that vector, the lower the loss.

$$L(y, \hat{y}) = -\sum_{j=0}^{M} \sum_{i=0}^{N} (y_{ij} * log(\hat{y}_{ij}))$$

Our Model Results:

- **loss(train_loss): 2.4050**
- **val_loss: 3.0527**

  - ***BLEU Score on Validation data(Higher the better)***

BLEU (bilingual evaluation understudy) is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another. Quality is considered to be the correspondence between a machine's output and that of a human: "the closer a machine translation is to a professional human translation, the better it is" – this is the central idea behind BLEU. BLEU was one of the first metrics to claim a high correlation with human judgments of quality and remains one of the most popular automated and inexpensive metrics.

Scores are calculated for individual translated segments—generally sentences—by comparing them with a set of good quality reference translations. Those scores are then averaged over the whole corpus to reach an estimate of the translation's overall quality. Intelligibility or grammatical correctness are not taken into account.

BLEU's output is always a number between 0 and 1. This value indicates how similar the candidate text is to the reference texts, with values closer to 1 representing more similar texts. Few human translations will attain a score of 1 since this would indicate that the candidate is identical to one of the reference translations. For this reason, it is not necessary to attain a score of 1. Because there are more opportunities to match, adding additional reference translations will increase the BLEU score.

Our Model Results:

- **BLEU: 0.606086**

Code:

```
def evaluate_model_beam_search(model, images, captions,
tokenizer, max_length, beam_index=3):
   actual, predicted = list(), list()
   for image_id, caption_list in tqdm(captions.items()):
```

```
        yhat = generate_caption_beam_search(model, tokenizer,
images[image_id], max_length, beam_index=beam_index)
        ground_truth = [caption.split() for caption in
caption_list]
        actual.append(ground_truth)
        predicted.append(yhat.split())
    print('BLEU Scores :')
    print('A perfect match results in a score of 1.0, whereas a
perfect mismatch results in a score of 0.0.')
    print('BLEU-1: %f' % corpus_bleu(actual, predicted,
weights=(1.0, 0, 0, 0)))
    print('BLEU-2: %f' % corpus_bleu(actual, predicted,
weights=(0.5, 0.5, 0, 0)))
    print('BLEU-3: %f' % corpus_bleu(actual, predicted,
weights=(0.3, 0.3, 0.3, 0)))
    print('BLEU-4: %f' % corpus_bleu(actual, predicted,
weights=(0.25, 0.25, 0.25, 0.25)))
```

10) Generating Caption for a test image

```
# Load the tokenizer
tokenizer_path = config['tokenizer_path']
tokenizer = load(open(tokenizer_path, 'rb'))

# Max sequence length (from training)
max_length = config['max_length']

# Load the model
caption_model = load_model(config['model_load_path'])

image_model = CNNModel(config['model_type'])

# Load and prepare the image
```

```
for image_file in os.listdir(config['test_data_path']):
   if(image_file.split('--')[0]=='output'):
       continue
   if(image_file.split('.')[1]=='jpg' or
image_file.split('.')[1]=='jpeg'):
       print('Generating caption for {}'.format(image_file))
       # Encode image using CNN Model
       image = extract_features(config['test_data_path']+image_file,
image_model, config['model_type'])
       # Generate caption using Decoder RNN Model + BEAM search
       generated_caption =
generate_caption_beam_search(caption_model, tokenizer, image,
max_length, beam_index=config['beam_search_k'])
       # Remove startseq and endseq
       caption = 'Caption: ' +
generated_caption.split()[1].capitalize()
       for x in
generated_caption.split()[2:len(generated_caption.split())-1]:
           caption = caption + ' ' + x
       caption += '.'
       # Show image and its caption
       pil_im = Image.open(config['test_data_path']+image_file, 'r')
       fig, ax = plt.subplots(figsize=(8, 8))
       ax.get_xaxis().set_visible(False)
       ax.get_yaxis().set_visible(False)
       _ = ax.imshow(np.asarray(pil_im), interpolation='nearest')
       _ = ax.set_title("BEAM Search with
k={}\n{}".format(config['beam_search_k'],caption),fontdict={'fontsize
': '20','fontweight' : '40'})
       plt.savefig(config['test_data_path']+'output--'+image_file)
```

Examples:

Image	Generated Caption
	A man is riding a bicycle on a dirt path.
	Man in red jacket snowboarding
	A woman in a tennis racket on the court. (here a man is mispredicted as a woman)

	Race car spins down the road as spectators watch
	A little boy is on the water on the floor with a over the floor (the caption generated for this image is irrelevant)
	Girl in pink shirt is smiling whilst standing in front of tree

Conclusion

We have successfully implemented our image caption generator using Convolutional Neural Networks, Recurrent Neural Networks, and Natural Language Processing.
Our model is able to generate accurate captions for most of the images.