Generation of Captions from Images – Report

Surya Menon – S-89A Deep Learning for Natural Language Processing, Summer 2019

Abstract –

Problem Statement:
Use deep learning and natural language processing techniques to generate relevant captions from input images.

Approach:
Train a neural network with the Flickr 8k dataset, that takes an image and text input and produces an appropriate word sequence to describe the image.

Use / Benefits:
This process could be used to extract insights/summaries from images, provide image information to those with visual impairments, and to classify/identify patterns in social media photo albums (i.e. facial recognition), or perhaps even in medical imaging assessments.

Drawbacks / Challenges:
- There was necessary text and image preparation and heavy reliance on pretrained networks for extracting image features and representing words as vectors.
- Encountered overfitting (perhaps due to needing more data for training) despite applying regularization techniques and simplifying the network.
- The model's effectiveness is limited to the variety of images it is trained on.

Results:
We were able to get the validation loss of generating sequences similar to actual captions to around 3.2, but experienced overfitting. The corpus BLEU scores used to evaluate the model were not high, but conducting predictions on test images (using both greedy and beam searches) resulted in captions that generally effectively capture the context of the images. Training with larger samples or employing data augmentation, as well as including an attention mechanism, cross-validation, and further hyperparameter tuning may improve results.

Working Example Description:
The following notebooks were used to create the demonstration:
1. `project_img-features.ipynb`: Processes the dataset images in a pretrained convolutional neural network (CNN) and extracts/saves image features
2. `model_create.ipynb`: Processes and visualizes dataset captions; creates training, validation, test data; tokenizes words and establishes word embedding; trains a "merge" model with a data generator; visualizes losses; evaluates model with BLEU scores; demonstrates a few examples of captions using greedy and beam searches.
3. `model_demo.ipynb`: Loads final model and generates captions for a few random test images with image plotting.

Data Source:
The Flickr 8k dataset consists of around 8000 images (of everyday people and situations from Flickr) with five associated captions per image, with predefined training, validation, and test sets.
- Officially request data here: https://forms.illinois.edu/sec/1713398
- Alternative to get immediately: https://github.com/jbrownlee/Datasets/releases

Video: https://youtu.be/zPawquEoQFs

Introduction and Dataset –

To generate captions from images, we need to develop a model that includes image processing and sequence generation, employing both convolutional and recurrent neural networks[1]. We follow the logic employed by several resources for text and image preparation and techniques for training to create an effective model[2].

The dataset used is called Flickr 8k, which consists of 6000 training images (JPEG), 1000 validation images, and 1000 test images, with each image having five associated captions; these images are predefined and indexed into these categories. The dataset comes in multiple zip files, which you should unzip and put in a file directory (i.e. `project_data`). We must process both the image and text data before we use them in our model, which we see next.

NOTE: The data submitted in `CaptionGeneration_MenonSurya_Final.zip` (in the `project_data` folder) for this project only contains a **sample** of the dataset used for training and evaluating the model created. If you want to reproduce the results seen in the code below (including full training, BLEU evaluations, and specific test predictions with image plotting), you need to download the full dataset, which can be accessed here, and place the downloaded folders into a `project_data` directory. When using Google Colab, you can upload the needed files (i.e. model, captions text, indexing files, image features dictionary, specific JPEG images) via the Files tab; the paths defined in the code in `model_create.ipynb` follow this process. Alternatively, we provide the paths to use with Jupyter Notebook (or configuring with Google Drive) in the comments and other code files. The artifacts produced in the following code (except the intermediate model files) are in the `artifacts` folder.

Image Preparation –

Associated file: `project_img-features.ipynb`

```
# alternatively install keras separately
import tensorflow
from tensorflow import keras
from tensorflow.keras.applications.inception_v3 import InceptionV3
from tensorflow.keras.applications.inception_v3 import preprocess_input
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing import image
import numpy as np
import os
import pickle
```

We want to process our input images when training the model so they contain the most relevant features (i.e. capturing the most important context, content, patterns) but are in a compressed representation. For efficiency, we capture and save these image feature representations first, which we later pass into the model as inputs, rather than training the entire image with the text sequences all at once[3].

We use the `InceptionV3` model which is a convolutional neural network (CNN) that conducts image classification using pre-trained weights[4]. Specifically, this model uses `ImageNet` weights, which have been trained on millions of images and over thousands of categories[5]. With `Keras` we can load this model, remove the classification layer at the top of the network, and obtain outputs from this model that are compressed representations of the input images:

```
# process input for inception  model
```

```
def img_size(img):
    # turn image into array
    img = image.img_to_array(img)
    # add dimension
    img = np.expand_dims(img, axis=0)
    # use InceptionV3 preprocess
    img = preprocess_input(img)
    return img

# go through images in folder and load model
# using Jupyter Notebook here
img_folder = os.path.join(os.getcwd(), 'project_data\Flickr8k_Dataset\Flicker8k_Dataset\\')
model = InceptionV3(weights='imagenet')

# not use to classify but get fixed length vector for images (compression)
# want output to be 2nd to last layer
feature_mod = Model(model.input, model.layers[-2].output)
```

After loading and initializing the model, we go through the dataset images folder and pass each image through the model; each image is first processed so it can be passed into the network (including resizing and formatting), and the resulting 2048-dimensional vector output is stored in a dictionary, which is indexed by the image identifier (i.e. image file name). We save this dictionary using the `pickle` package (under name `img_features.pkl`), and we later load these image representations as inputs for the model created in `model_create.ipynb`.

```
# feature dictionary (index by image name)
imgs = dict()

# go through images in folder
for i in os.listdir(img_folder):
    # print image name - just for tracking
    print(i)
    # create path
    path = img_folder + i
    # load image with InceptionV3 default size
    img = image.load_img(path, target_size=(299,299))
    # prepare image for CNN model
    img = img_size(img)
    # get features
    feat = feature_mod.predict(img)
    feat = np.reshape(feat, feat.shape[1])
    # pass to dictionary
    imgs[i.split('.')[0]] = feat

# save for later
pickle.dump(imgs, open('img_features.pkl', 'wb'))
```

Text Preparation – Associated file: `model_create.ipynb`

```
import os
import string
import pickle
import numpy as np
```

Each input image in the dataset has five associated captions, so we also create a dictionary out of these captions, where each image identifier key results in a list of associated captions. First we open the file `Flickr8k.token.txt`, which contains all of the captions data:

Note: When reading these lines in Jupyter Notebook we had to increase our data rate limit, but on Google Colab using GPU we did not encounter this issue[6].

```
# load dataset with captions:
def get_captions(filename):
    # open captions file and return lines
    file = open(filename, 'r')
    text = file.read()
    file.close()
    return text

# path to captions data
# Google Colab version (loaded into Files section)
filename = 'Flickr8k.token.txt'

# Jupyter file path - put project_data folder in same directory as this file
# filename = 'project_data\Flickr8k_text\Flickr8k.token.txt'

path = os.path.join(os.getcwd(), filename)
# get all captions
doc = get_captions(path)

# preview original captions text
doc.split('\n')[:5]
```

```
Out[1]: ['1000268201_693b08cb0e.jpg#0\tA child in a pink dress is climbing up a set of stairs in an e
        ntry way .',
         '1000268201_693b08cb0e.jpg#1\tA girl going into a wooden building .',
         '1000268201_693b08cb0e.jpg#2\tA little girl climbing into a wooden playhouse .',
         '1000268201_693b08cb0e.jpg#3\tA little girl climbing the stairs to her playhouse .',
         '1000268201_693b08cb0e.jpg#4\tA little girl in a pink dress going into a wooden cabin .']
```

For each line of text, we remove punctuation, casing, and words that have numbers or 1 letter; we do this to avoid duplicates and to keep the vocabulary to more common words. Additionally we add start and end tokens (< and >), which the model will need when a generating sequences. We pass each cleaned line to the list of the corresponding dictionary entry. We also provide the option to save this dictionary as `captions.pkl`, which can be reloaded later:

```
# make captions clean
def new_caps(s):
    # take out punctuation
    s = s.translate(str.maketrans('', '', string.punctuation))
    # make lower case (prevent duplicates)
    s = s.lower()
    # no words with numbers
    s = ' '.join(w for w in s.split() if not any(d.isdigit() for d in w))
    # no 1 letter words (i.e. s that follows removed apostrophe)
    s = ' '.join(w for w in s.split() if len(w)>1)
    return s

# create dictionary of photo_id and captions
captions = dict()
# each line
```

```
for line in doc.split('\n'):
    # skip last empty line
    if len(line) < 1:
        continue
    # split on .jpg to get image name, 1st token is image id
    tokens = line.split('.')
    img_id = tokens[0]
    # check pics with no captions
    if '\t' not in tokens[1]:
        cap = tokens[1]
    # split 2nd token into descriptions, after \t, put in list
    else:
        cap = tokens[1].split('\t')[1]
        # make lower case, remove punctuation
        cap = new_caps(cap)
        # put start and end tokens
        cap = '< ' + cap + ' >'
    # append caption to list of associated image entry
    if img_id not in captions:
        captions[img_id] = list()
    captions[img_id].append(cap)

# test captions dictionary
print(captions['997722733_0cb5439472'])

# save captions dictionary in file
pickle.dump(captions, open('captions.pkl', 'wb'))
```

```
Out[2]: ['< man in pink shirt climbs rock face >',
         '< man is rock climbing high in the air >',
         '< person in red shirt climbing up rock face covered in assist handles >',
         '< rock climber in red shirt >',
         '< rock climber practices on rock climbing wall >']
```

Visualizing Vocabulary – Associated file: `model_create.ipynb`

```
import nltk
import pandas as pd
import matplotlib.pyplot as plt
```

We next visualize the vocabulary in our captions input text to assess whether it is worthwhile to limit the vocabulary size in our model, since noisier vocabulary with lots of uncommon words may result in longer model training[3]. First we go through all the text in the captions dictionary created above, and get a set of all of the words; we found that there were 8761 unique words in the captions "corpus", and 413286 total words:

```
# create vocabulary out of all captions (8000 x # of captions per each)
vocab = set()
# go through each image caption list
for index in captions.keys():
    # add words to set from all captions
    [vocab.update(c.split()) for c in captions[index]]

# remove start and end tokens for visualize
vocab.remove('<')
vocab.remove('>')

```

```
# get vocab size
print('Vocab size:', len(vocab))

# put all text in string - to get frequencies
txt = ''
# go through each image caption list
for index in captions.keys():
    # add words to set from all captions
    t = ''.join(captions[index])
    # remove tokens from count
    t = t.translate(str.maketrans('', '', '<>'))
    txt += t

txt = txt.split(' ')
print('Total words:', len(txt))
```

```
Vocab size: 8761
Total words: 413286
```

Using the `nltk` library, we create frequency distributions of the top 100 words in the captions text data, and plot these results:

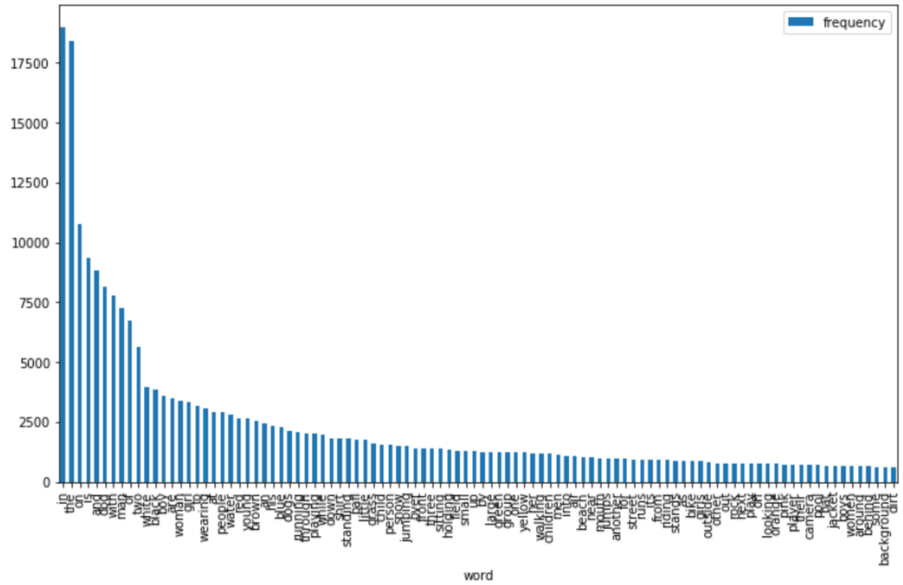
```
# get word frequencies
word_ct = nltk.FreqDist(txt)

# look at top 5 words
print(word_ct.most_common(5))

# select top 100 words (index 1 hold space so exclude)
wc = word_ct.most_common(101)[1:]

wc = pd.DataFrame(wc, columns=['word', 'frequency'])
# plot frequencies
wc.plot.bar(x='word', y='frequency', figsize=(12,7))
```

```
[('', 40458), ('in', 18972), ('the', 18418), ('on', 10741), ('is', 9344)]
Out[171]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd412dc6198>
```



We see that a large bulk of the words are taken from the top 100 words, so we determine that we can safely reduce the vocabulary size, which can help speed up model training without detrimentally impacting the model quality.

Loading Training Data and Tokenizing – Associated file: `model_create.ipynb`

```
import keras
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
```

Now we create functions to load the training (and later validation and test) data for our model. The file `Flickr_8k.trainImages.txt` in the text data folder indicates which 6000 images can be used for training the network; we create a function to get a list of these image identifiers. We then retrieve the associated training images and captions by going through the captions and image features dictionaries (created in the Image and Text Preparation sections above), and including entries that have keys with training image identifiers. The result is training image and caption dictionaries:

Note: Place `img_features.pkl` in the same directory as `model_create.ipynb` to run the code below:

```
# Google Colab version (loaded into Files section)
filename = 'Flickr_8k.trainImages.txt'

# Jupyter file path - put project_data folder in same directory as this file
# filename = 'project_data\Flickr8k_text\Flickr_8k.trainImages.txt'

f = os.path.join(os.getcwd(), filename)

# get (train) photo ids - put in list
def get_ids(f):
    # open file - list of training image ids
    doc = get_captions(f)
    train_ids = list()
    # by line
    for line in doc.split('\n'):
        # skip empty lines
        if len(line) < 1:
            continue
        # take out jpg ending
        train_ids.append(line.split('.')[0])
    return train_ids

# get captions and images with associated ids
def get_data(f, captions, img_file):
    idens = get_ids(f)
    # for each id, get associated captions
    caps = {i: list(captions[i]) for i in idens}
    # load image features file from pretrained CNN
    all_imgs = pickle.load(open(img_file, 'rb'))
    # associated id, get image feature
    imgs = {i: all_imgs[i] for i in idens}
    return idens, caps, imgs

```

```
# get training data
train_ids, train_caps, train_imgs = get_data(f, captions, 'img_features.pkl')
```

We then have to encode the words (creating an index that maps words to integers) by utilizing the `Tokenizer` class in `Keras`. We first convert the training captions dictionary into a list (so it can be passed to the tokenizer as a "corpus" of sorts). Based on the results in the Visualizing Vocabulary section, we decide to reduce our vocabulary size to the top 3500 words. We create an index (which includes our start and end tokens) and can save this tokenizer as `tokenizer.pkl`, which can be loaded later (such as in the file, `model_demo.ipynb`):

```
# turn caption dictionary to caption list to encode
def dict_list(caps):
    cap_list = list()
    for index in caps.keys():
        [cap_list.append(c) for c in caps[index]]
    return cap_list

# turn training captions into list
cap_list = dict_list(train_caps)

# use reduced vocabulary size
n_words = 3500
# include <, > as words in tokenizer, create index
tokens = Tokenizer(num_words = n_words, filters='!"#$%&()*+,-./:;=?@[\\]^_`{|}~\t\n')

# vectorize words
tokens.fit_on_texts(cap_list)

# add 1 for 0 padding
vocab_size = n_words + 1

# can save tokenizer for later
pickle.dump(tokens, open('tokenizer.pkl', 'wb')
```

Word Embeddings – Associated file: `model_create.ipynb`

```
import io
```

In our model, the text input will be passed through an `Embedding` layer, which provides a dense, low-dimensional vector representation of the words based on semantics[7]. We decide to use pretrained word embeddings, since our model will generally be looking at generic everyday images (and thus common word/semantic structures), based on the dataset content[8].

For our pre-trained word embedding space, we use the 300-dimensional GloVe word vector representations from the file `glove.6B.300d.txt`, which we can download here and put in the same directory as `model_create.ipynb`[9]. We create an embedding dimension of 300, go through our tokenizer word index, and add the vector representation of each word in our 3500-word vocabulary into an embedding matrix; this matrix will be passed into the `Embedding` layer of our model and frozen, so these pretrained representations are unchanged during training. We can save our matrix as `em3500_300d.npy`, so we can reload rather than recreate if needed.

```
# use pretrained word embeddings - follow lecture examples
embeddings_index = {}
# Google Colab file
```

```
with io.open('/content/drive/My Drive/nlp_project_data/glove.6B.300d.txt', encoding='utf8') as f:
# Jupyter file path
# with io.open('glove.6B.300d.txt', encoding='utf8') as f:
  for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:],dtype='float32')
    embeddings_index[word] = coefs

# embedding dimension layer
embedding_dim = 300

# create embedding matrix with weights, of appropriate size
embedding_matrix = np.zeros((vocab_size, embedding_dim))
# go through tokenizer
for w, i in tokens.word_index.items():
    embedding_vector = embeddings_index.get(w)
    # if within vocab size
    if i < vocab_size:
        if embedding_vector is not None:
        # add vector representation
        embedding_matrix[i] = embedding_vector

#  save embedding matrix
np.save('em3500_300d.npy', embedding_matrix)
```

Model Creation and Training – Associated file: `model_create.ipynb`

```
from keras import optimizers
from keras.models import Model
from keras import backend as K
from keras.layers.merge import add
from keras.models import load_model
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras.layers import Input, Dense, Embedding, Dropout, RepeatVector, Bidirectional, GRU
```

Now that we have loaded our training data (both images and text), we begin building our caption generator model. To manage the amount of memory used during training, due to the size of data being processed, we employ a data generator function and use `fit_generator()` for a model in `Keras`; only the current batch of data being processed is stored in memory, rather the all of the data[10].

We create a generator function which prepares all input (text and images) and output combinations for a specific batch size. During training, we go through each index and encode the current training captions text set (including padding for a fixed sized input), prepare the image features input, create classification labels for the actual word outputs, and generate all input-output pairs; when we reach the desired batch size, we yield this prepared data for training. We will create training and validation generators. We adapt a common generator structure design[2]:

```
# generator to run training - can alter batch size
def generator(captions, imgs, tokens, max_len, vocab_size, b_size=1):
    # store 2 inputs and 1 output
    inp_txt, inp_img, out = list(), list(), list()
    n = 0
    while 1:
```

```
        # go through captions and ids
        for index, caps in captions.items():
            n +=1
            # prep photo input
            img = np.array(imgs[index])
            # for each caption
            for cap in caps:
              # encode captions
              encode = tokens.texts_to_sequences([cap])[0]
              # create all possible pairs in a caption
              for i in range(1, len(encode)):
                # get all possible input/output sequences
                in_seq, out_seq = encode[:i], encode[i]
                # pad so input is fixed length
                in_seq = pad_sequences([in_seq], maxlen=max_len)[0]
                # create classification labels for output
                out_seq = to_categorical([out_seq], num_classes = vocab_size)[0]
                inp_img.append(img)
                inp_txt.append(in_seq)
                out.append(out_seq)
             # return if reach batch size
            if n == b_size:
                yield[[np.array(inp_img), np.array(inp_txt)], np.array(out)]
                # reset
                inp_txt, inp_img, out = list(), list(), list()
                n=0
```

We also define a function to calculate the maximum length, based on the largest caption in the training dataset; when the model is outputting a caption sequence, it is either looking for the end token or reaching a word limit.

```
# look at longest caption - use as max length in model
def max_length(caps):
    return max(len(c.split()) for c in dict_list(caps))

# get max_len of sequence for training
max_len = max_length(train_caps)
```

Next we design our actual model structure, which takes an initial encoded word input (i.e. start token) and an image features input vector, and generates the next word in a sequence; the model will continue to pass the currently constructed sequence and image as inputs until a full caption is generated[11].

We experimented with different model structures, complexities, and hyperparameters, but we only present the final implemented model here. For this model structure we chose to merge the text and image inputs first, before passing to a recurrent neural network (RNN) cell; however we also saw examples of the text input processed in the RNN first before being merged with the image features[11].

We previously extracted image features in the file `project_img-features.ipynb` using a pretrained CNN, so we use training data in `img_features.pkl` as the image input into our model. Using functional API, we further process this input by reducing from a 2048 to 300 dimension representation with a `Dense` layer; we also use a `Repeat Vector` to ensure that we can merge this vector with the text vector later in the model.

```
# take photo features of (2048, ) input
```

```
img_inputs = Input(shape=(2048,))
# handle overfitting
feats1 = Dropout(0.5)(img_inputs)
# reduce size
feats2 = Dense(300, activation='relu')(feats1)
# so same size as text input when merge
feats3 = RepeatVector(max_len)(feats2)
```

The encoded text input is passed into an `Embedding` layer, where we freeze the pretrained GloVe weights during training. The output of this process is also represented in 300 dimensions:

```
# pass text sequences
# expect max_len inputs since padded
txt_inputs = Input(shape=(max_len,))
# put in embedding - size of vocab, 0 is padding
# add embedding matrix, freeze (don't update weights)
seq1 = Embedding(vocab_size, 300, weights=[embedding_matrix], trainable=False,
                 mask_zero=True)(txt_inputs)
```

Finally, we concatenate the image and text input vectors together, and pass them into a RNN layer, specifically a bidirectional `GRU` layer with 500 memory units (actually 1000 since we are creating hidden layers in opposite directions). We then feed this output into a final output `Dense` classification layer the size of the vocabulary, that predicts the next encoded word in the sequence, using `softmax` activation. Furthermore, we insert `Dropout` layers before the image processing and RNN layers, in attempts to alleviate some overfitting[3]. From different training runs, we also discovered that starting with an initial learning rate of 0.0005 when using an `Adam` optimizer resulted in improved performance. We initialize and compile this model and preview our structure:

```
# merge inputs
merge1 = add([feats3, seq1])
# handle overfitting
merge2 = Dropout(0.5)(merge1)
# gru model to read sequences, make bidirectional
merge3 = Bidirectional(GRU(500, return_sequences=False))(merge2)
# generate classification/output layer
outputs = Dense(vocab_size, activation='softmax')(merge3)

# pass inputs and outputs into model
model = Model(inputs=[img_inputs, txt_inputs], outputs=outputs)

# look at layers
print(model.summary())

# compile model - set loss, optimizer, learning rate
model.compile(loss='categorical_crossentropy', optimizer=optimizers.Adam(lr=0.0005))
```

```
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
input_1 (InputLayer)             (None, 2048)          0
____________________________________________________________________________________________________
dropout_1 (Dropout)              (None, 2048)          0           input_1[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 300)           614700      dropout_1[0][0]
____________________________________________________________________________________________________
input_2 (InputLayer)             (None, 34)            0
____________________________________________________________________________________________________
repeat_vector_1 (RepeatVector)   (None, 34, 300)       0           dense_1[0][0]
____________________________________________________________________________________________________
embedding_1 (Embedding)          (None, 34, 300)       1050300     input_2[0][0]
____________________________________________________________________________________________________
add_1 (Add)                      (None, 34, 300)       0           repeat_vector_1[0][0]
                                                                   embedding_1[0][0]
____________________________________________________________________________________________________
dropout_2 (Dropout)              (None, 34, 300)       0           add_1[0][0]
____________________________________________________________________________________________________
bidirectional_1 (Bidirectional)  (None, 1000)          2403000     dropout_2[0][0]
____________________________________________________________________________________________________
dense_2 (Dense)                  (None, 3501)          3504501     bidirectional_1[0][0]
====================================================================================================
Total params: 7,572,501
Trainable params: 6,522,201
Non-trainable params: 1,050,300
```

As mentioned above, we conducted training using different model variations and found this model structure produced the best results. Prior models included the following variations: `Dropout` layers in different locations, passing just the text input into a RNN structure and merging these results with the image features input, fewer and more units in the `Dense` and RNN layers, multiple RNN layers, lack of bidirectionality, `LSTM` versus `GRU` layer, use of default learning rate, etc. Additionally, we adjusted different parameters in training as well, such as epochs and batch size. Again, the procedure performed in this report presents the training conducted to create our final model.

First we load our validation dataset images and captions, following the same procedure used to set up our training data in the Loading Training Data and Tokenizing section above:

```
# Google Colab version (loaded into Files section)
val_file = 'Flickr_8k.devImages.txt'

# Jupyter file path - put project_data folder in same directory as this file
# val_file = os.path.join(os.getcwd(),'project_data\Flickr8k_text\Flickr_8k.devImages)

# get validation data
val_ids, val_caps, val_imgs = get_data(val_file, captions, 'img_features.pkl')
```

We set up our training and validation generators and first train our model on 5 epochs, using a batch size of 5. Experimenting with batch size in particular was informative in improving initial training (i.e. larger batch sizes trained faster but often led to faster overfitting). We saved the current model state, along with the training and validation losses:

```
# define epochs and batches
epochs = 5
pics_batch = 5
t_steps = len(train_caps)//pics_batch
v_steps = len(val_caps)//pics_batch

# create generators
train_g = generator(train_caps, train_imgs, tokens, max_len, vocab_size, pics_batch)
```

```
val_g = generator(val_caps, val_imgs, tokens, max_len, vocab_size, pics_batch)

# run training
history = model.fit_generator(train_g, steps_per_epoch=t_steps, epochs=epochs,
                              validation_data=val_g, validation_steps=v_steps)

# save first part of model, and history
model.save('mod-fin.h5')
with open('mod-fin_history', 'wb') as handle:
    pickle.dump(history.history, handle)
```

```
Epoch 1/5
1200/1200 [==============================] - 183s 152ms/step - loss: 4.4144 - val_loss: 3.7254
Epoch 2/5
1200/1200 [==============================] - 179s 149ms/step - loss: 3.6171 - val_loss: 3.4637
Epoch 3/5
1200/1200 [==============================] - 179s 149ms/step - loss: 3.3415 - val_loss: 3.3691
Epoch 4/5
1200/1200 [==============================] - 180s 150ms/step - loss: 3.1585 - val_loss: 3.3310
Epoch 5/5
1200/1200 [==============================] - 179s 149ms/step - loss: 3.0128 - val_loss: 3.3170
```

Next we train on 5 more epochs (with the same batch size) with a further reduced learning rate, to allow for more fine-tuned weight adjustments[12]. Again, we save model weights and history:

```
# reduce learning rate
K.set_value(model.optimizer.lr, 0.0001)

history = model.fit_generator(train_g, steps_per_epoch=t_steps, epochs=epochs,
                              validation_data=val_g, validation_steps=v_steps)

model.save('mod-fin-1.h5')
with open('mod-fin-1_history', 'wb') as handle:
    pickle.dump(history.history, handle)
```

```
Epoch 1/5
1200/1200 [==============================] - 182s 151ms/step - loss: 2.7795 - val_loss: 3.2341
Epoch 2/5
1200/1200 [==============================] - 182s 152ms/step - loss: 2.7232 - val_loss: 3.2269
Epoch 3/5
1200/1200 [==============================] - 180s 150ms/step - loss: 2.6858 - val_loss: 3.2221
Epoch 4/5
1200/1200 [==============================] - 180s 150ms/step - loss: 2.6524 - val_loss: 3.2208
Epoch 5/5
1200/1200 [==============================] - 178s 149ms/step - loss: 2.6249 - val_loss: 3.2189
```

We run our training on 5 more epochs, increasing batch size to 10, which has been discussed to sometimes improve performance in later training[10]. We employ model checkpoints and early stopping, only saving models that continued to reduce validation loss, and terminating training when improvements were not made after 2 epochs. We save the final loss history:

```
epochs = 5
pics_batch = 10
t_steps = len(train_caps)//pics_batch
v_steps = len(val_caps)//pics_batch

train_g = generator(train_caps, train_imgs, tokens, max_len, vocab_size, pics_batch)
val_g = generator(val_caps, val_imgs, tokens, max_len, vocab_size, pics_batch)

```

```
# save models with improved performance
filepath = 'mod-fin-ep{epoch:03d}-loss{loss:.3f}-val_loss{val_loss:.3f}.h5'

# define callbacks for tracking loss
checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose=1, save_best_only=True, mode='min')
early = EarlyStopping(monitor='val_loss', patience=2, verbose=1)

history = model.fit_generator(train_g, steps_per_epoch=t_steps, epochs=epochs,
                          validation_data=val_g, validation_steps=v_steps,
                          callbacks=[early, checkpoint])

with open('mod-fin-1-1_history', 'wb') as handle:
    pickle.dump(history.history, handle)
```

```
Epoch 1/5
600/600 [==============================] - 123s 205ms/step - loss: 2.5863 - val_loss: 3.2138

Epoch 00001: val_loss improved from inf to 3.21377, saving model to mod-fin-ep001-loss2.588-val_loss3.214.h5
Epoch 2/5
600/600 [==============================] - 123s 205ms/step - loss: 2.5686 - val_loss: 3.2147

Epoch 00002: val_loss did not improve from 3.21377
Epoch 3/5
600/600 [==============================] - 122s 203ms/step - loss: 2.5537 - val_loss: 3.2158

Epoch 00003: val_loss did not improve from 3.21377
Epoch 00003: early stopping
```

We see that our final model is saved as `mod-fin-ep001-loss2.588-val_loss3.214.h5`, and we stopped training after 3 more epochs; our full training was 13 epochs, with a final validation loss at around 3.21. Through our various training runs and using different structures, complexities, and hyperparameters, we could not really reach losses below this threshold.

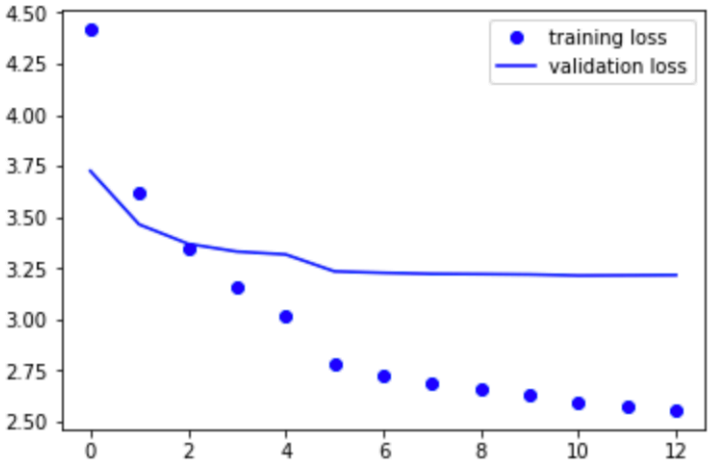
To plot training and validations losses, we reload the saved loss histories:

```
# includes losses from all epochs
losses1 = pickle.load(open('mod-fin_history', 'rb'))
losses2 = pickle.load(open('mod-fin-1_history', 'rb'))
# alt - losses3 = pickle.load(open('mod-fin-1-1_history', 'rb'))
losses3 = history.history

# combine losses
loss = losses1['loss'] + losses2['loss'] + losses3['loss']
val_loss = losses1['val_loss'] + losses2['val_loss'] + losses3['val_loss']

def plot_loss(loss, val):
    epochs=range(len(loss))
    plt.figure()
    plt.plot(epochs, loss, 'bo', label='training loss')
    plt.plot(epochs, val, 'b', label='validation loss')
    plt.legend()
    plt.show()

# plot training and validation losses
plot_loss(loss, val_loss)
```



We observe that the training loss declines as expected, and the validation loss also declines before starting to level off at around 3.2; after the second epoch, the validation loss remains larger than the training loss, which indicates overfitting in the model. Despite adjusting model complexity and parameters in previous training runs (such as reducing neurons, adjusting model structure, learning rates, batching, etc.) and playing around with regularization techniques (in particular `Dropout` layers), we ultimately found that our networks kept overfitting. This indicates that we may need to use a larger dataset to improve our training.

Model Evaluation – Associated file: `model_create.ipynb`

```
1. import matplotlib.pyplot as plt
2. from nltk.translate.bleu_score import corpus_bleu
```

We now evaluate our model on the test dataset, utilizing corpus BLEU scores, which compare a predicted sequence against an actual reference sentence; perfect scores are 1.0, and complete mismatches are 0.0[13]. It has been shown that caption generator models can be evaluated by looking at BLEU-(1,2,3,4) cumulative scores, by counting all matching n-grams in a sequence to determine overlap between generated and actual captions[11].

First we load the test dataset captions and image features:

```
1. # Google Colab version (loaded into Files section)
2. test_file = 'Flickr_8k.testImages.txt'
3.
4. # Jupyter file path - put project_data folder in same directory as this file
5. # test_file = os.path.join(os.getcwd(), 'project_data\Flickr8k_text\Flickr_8k.testImage
   s')
6.
7. # get test data
8. test_ids, test_caps, test_imgs = get_data(test_file , captions, 'img_features.pkl')
```

We also define a function to decode output text inputs (from integers back to words), using the word index from our tokenizer. We reload our saved final model (placed in the same directory as `model_create.ipynb`), so we can run predictions:

```
1. # covert index to word
```

```
def idtw(num, tokens):
    for w, i in tokens.word_index.items():
        if i == num:
            return w
    return None

# load best model
# Google Colab version (loaded into Files section)
model_name = 'mod-fin-ep001-loss2.588-val_loss3.214.h5'
mod = load_model(model_name)
```

To generate caption predictions, we create a function to which we can pass an image feature input and start token, and receive an output sequence. Specifically, this method performs a greedy search to generate a sequence by selecting the word with the highest probability from the `softmax` activation when creating a caption[14]:

```
# predict captions - greedy version
def make_cap(model, tokens, img, max_len):
    # start token
    in_txt='<'
    # go over entire possible sequence (of max length)
    for i in range(max_len):
        # encode input - similar set up to generator
        seq = tokens.texts_to_sequences([in_txt])[0]
        # pad
        seq = pad_sequences([seq], maxlen=max_len)
        # get next word in sequence
        pred = model.predict([np.array(img), seq])
        # get encoded of highest probability
        pred = np.argmax(pred)
        # decode output
        w = idtw(pred, tokens)
        # break if cannot map
        if w is None:
            break
        # append to sequence
        in_txt += ' ' + w
        # stop if reach end token
        if w == '>':
            break
    return in_txt
```

We can test this method on an image in the test dataset, by printing the caption and plotting the actual original JPEG image:

```
# Google Colab version (loaded into Files section)
i_path='2654514044_a70a6e2c21.jpg'

# Jupyter file path - put project_data folder in same directory as this file
# i_path = os.path.join(os.getcwd(), 'project_data\Flickr8k_Dataset\Flicker8k_Dataset\\
')
# i_path = os.path.join(img_folder, '2654514044_a70a6e2c21.jpg')

# in case just loading model without running training
max_len = max_length(train_caps)

# plot image
x=plt.imread(i_path)
```

```
plt.imshow(x)

# get caption - take out start/end tokens
print(make_cap(mod, tokens, test_imgs['2654514044_a70a6e2c21'].reshape(1,2048), max_len
)[1:-1])
```

```
brown dog is running through the grass
```

We follow a procedure to generate corpus BLEU scores similar to what was discussed in lecture, employing greedy search predictions with the test data:

```
# evaluate model on test images using greedy search
def bleu(mod, test_caps, test_imgs, tokens, max_len):
    act, pred = list(), list()
    for i, caps in test_caps.items():
        img = test_imgs[i].reshape(1,2048)
        p = make_cap(mod, tokens, img, max_len)
        act.append([c.split() for c in caps])
        pred.append(p.split())
    print('BLEU-1: %f' % corpus_bleu(act, pred, weights=(1.0, 0, 0, 0)))
    print('BLEU-2: %f' % corpus_bleu(act, pred, weights=(0.5, 0.5, 0, 0)))
    print('BLEU-3: %f' % corpus_bleu(act, pred, weights=(0.3, 0.3, 0.3, 0)))
    print('BLEU-4: %f' % corpus_bleu(act, pred, weights=(0.25, 0.25, 0.25, 0.25)))

bleu(mod, test_caps, test_imgs, tokens, max_len)
```

```
BLEU-1: 0.583389
BLEU-2: 0.348187
BLEU-3: 0.250425
BLEU-4: 0.128517
```

The BLEU-1 score is the best (close to .6) with subsequent cumulative n-gram scores declining; the overall scores are slightly lower but generally comparative to results seen in previous models[11]. Perhaps a larger training set could result in improved performance. Additionally, since we are using a reduced vocabulary based only on training captions, it is unlikely we will be able to generate exact test caption matches.

An alternative approach, beam search, has been proposed to generate caption sequences[3]. Instead of picking the most probable next word, we instead consider and store multiple alternatives and track possible sequences generated at each time step[15]. Some research has demonstrated that beam search may sometimes generate better captions, so we incorporate a certain implementation of this process to see if there are improvements[16].

We adapt a function to generate beam search captions below, which runs the model on start token and image features inputs, and goes through and stores the top possibilities (`beam_index`) at each time step until we finally select the highest probable sequence at the end:

```
# create beam search
def beam_caps(model, tokens, img, max_len, beam_index=3):
    # start sequence
    in_txt = [[tokens.texts_to_sequences(['<'])[0], 0.0]]

    # while less than max length
    while len(in_txt[0][0]) < max_len:
        temp = []
        # go through sequence
        for i in in_txt:
            # pad input sequences
            seq = pad_sequences([i[0]], maxlen=max_len, padding='post')
            # get predictions for next word
            pred = model.predict([np.array(img), seq])
            # sort and pick top beam_index possibilities
            beam_preds = np.argsort(pred[0])[-beam_index:]

            # create new lists from each top pick - words and probabilities
            # build each sequence option
            for w in beam_preds:
                # get current seq, prob
                new_cap, prob = i[0][:], i[1]
                # add word and associated probability
                new_cap.append(w)
                prob += pred[0][w]
                temp.append([new_cap, prob])

        # hold possibilities
        in_txt = temp
        # sort probabilities - lambda identifies value
        in_txt = sorted(in_txt, key=lambda p: p[1])
        # pick top in full list of options
        in_txt = in_txt[-beam_index:]

    # pick top sequence in all
    in_txt = in_txt[-1][0]
    # decode
    inter_cap = [idtw(i, tokens) for i in in_txt]

    # construct caption
    final_cap = []
    for i in inter_cap:
        # if not end token
        if i != '>':
            final_cap.append(i)
        else:
            break
    # take out start token, make string
    final_cap = ' '.join(final_cap[1:])
    return final_cap
```

We use another test image to run a function that generates caption predictions with greedy search, along with beam search results using indices of 3, 5, and 7:

```
# Google Colab version (loaded into Files section)
i_path='1322323208_c7ecb742c6.jpg'

# Jupyter file path - put project_data folder in same directory as this file
# i_path = os.path.join(os.getcwd(), 'project_data\Flickr8k_Dataset\Flicker8k_Dataset\\')
# i_path = os.path.join(img_folder, '1322323208_c7ecb742c6.jpg')

# acquire greedy and beam searches and display an image
def print_img_caps(path, mod, tokens, img, max_len):
    # plot image
    x=plt.imread(path)
    plt.imshow(x)
    # reshape image features
    img = img.reshape(1,2048)
    # get greedy caption
    txt = make_cap(mod, tokens, img, max_len)[1:-1]
    # get beam searches
    print('Greedy:', txt)
    print('Beam, k=3:', beam_caps(mod, tokens, img, max_len))
    print('Beam, k=5:', beam_caps(mod, tokens, img, max_len, beam_index=5))
    print('Beam, k=7:', beam_caps(mod, tokens, img, max_len, beam_index=7))

print_img_caps(i_path, mod, tokens, test_imgs['1322323208_c7ecb742c6'], max_len)
```

```
Greedy:  man in black shirt is standing on the beach with the sun behind him
Beam, k=3: two people are standing on the beach at dusk
Beam, k=5: two people are sitting on the beach next to the ocean
Beam, k=7: young boy running on the beach at sunset
```

In the above example, we see that beam search (in particular with index 3) results in a preferable caption based on the image, compared to the greedy search output. We decide to re-run our BLEU scores with beam search index 3 predictions to see if there is an improvement:

Note: The code below takes a while to run.

```
# evaluate model on test images using beam search, beam_index=3
def bleu_2(mod, test_caps, test_imgs, tokens, max_len):
    act, pred = list(), list()
    for i, caps in test_caps.items():
        img = test_imgs[i].reshape(1,2048)
        p = beam_caps(mod, tokens, img, max_len)
        # take out start/end tokens to compare
        act.append([c[1:-1].split() for c in caps])
        pred.append(p.split())
    print('BLEU-1: %f' % corpus_bleu(act, pred, weights=(1.0, 0, 0, 0)))
    print('BLEU-2: %f' % corpus_bleu(act, pred, weights=(0.5, 0.5, 0, 0)))
    print('BLEU-3: %f' % corpus_bleu(act, pred, weights=(0.3, 0.3, 0.3, 0)))
```

```
    print('BLEU-4: %f' % corpus_bleu(act, pred, weights=(0.25, 0.25, 0.25, 0.25)))

bleu_2(mod, test_caps, test_imgs, tokens, max_len)
```

```
BLEU-1: 0.488283
BLEU-2: 0.307064
BLEU-3: 0.227320
BLEU-4: 0.116195
```

We see that these BLEU scores are lower than the ones we calculated above. Although we saw that beam search may sometimes result in more accurate captions (when looking at specific examples), they may not exactly match actual captions in the dataset, which could explain the lower scores.

Next we create a demo file, `model_demo.ipynb`, which will plot images and generate greedy and beam search captions for a random set of test images in the dataset. We save the image features dictionary for the test data so we can reuse them in the demo:

```
# save test imgs dictionary in file
pickle.dump(test_imgs, open('test_imgs.pkl', 'wb'))
```

Demonstration – Associated file: `model_demo.ipynb`

```
import os
import keras
import random
import string
import pickle
import numpy as np
import matplotlib.pyplot as plt
from keras.models import load_model
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
```

We want to see actual generated greedy and beam search captions for some images, so in the code below, we select 5 random test images to plot and produce captions. We load some of the saved dictionaries and files generated in the file `model_create.ipynb` (and placed in the same directory): `tokenizer.pkl`, `test_imgs.pkl`, and `mod-fin-ep001-loss2.588-val_loss3.214.h5`. We also set the maximum caption length based on training captions:

Note: Place these files in the same directory as `model_demo.ipynb` to run the code below:

```
# based on training of model – determined in model_create.ipynb
max_len = 34

# load previous saved test image features
test_imgs = pickle.load(open('test_imgs.pkl', 'rb'))

# load saved tokenizer
tokens = pickle.load(open('tokenizer.pkl', 'rb'))

# load final model
model_name = 'mod-fin-ep001-loss2.588-val_loss3.214.h5'
mod = load_model(model_name)
```

Next we include functions created in `model_create.ipynb` to decode integer indices back to words and generate and print greedy and search captions: `idtw()`, `make_cap()`, `beam_caps()`, and `print_img_caps()`. See the Model Evaluation section for the code for these functions.

Note: We insert the following code in the `print_img_caps()` method before generating captions, to print the image identifier.

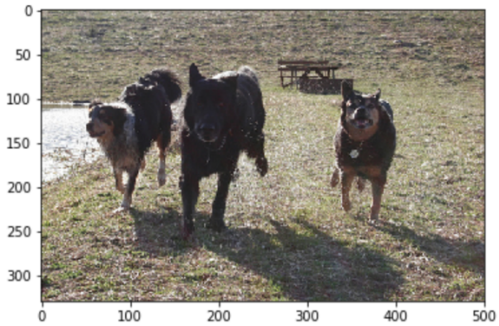
```
print('\033[1m', 'Image ID:','\033[0m', path.split('\\')[-1])
```

We test the methods on an image, and then we select 5 random images from the `test_imgs` dictionary. We create a loop that goes through each image, and plots the image and its generated greedy and beam search captions:

```
# set up full images folder
img_folder = os.path.join(os.getcwd(), 'project_data\Flickr8k_Dataset\Flicker8k_Dataset\\')

# test image
i ='3462454965_a481809cea.jpg'
path = os.path.join(img_folder, i)
print_img_caps(path, mod, tokens, test_imgs[i.split('.')[0]], max_len)
```

```
 Image ID:  3462454965_a481809cea.jpg
Greedy:  black dog is running through the grass
Beam, k=3: black dog and brown dog are playing in the grass
Beam, k=5: the black dog is running through the grass
Beam, k=7: the black dog is running through the green grass
```




```
# to reproduce - can take out if want different images
random.seed(8)

# pick 5 random images
keys = random.sample(list(test_imgs), 5)

for k in keys:
    img = k + '.jpg'
    path = os.path.join(img_folder, img)
    print_img_caps(path, mod, tokens, test_imgs[img.split('.')[0]], max_len)
    plt.show()
    # add space between images
    print()
```

Image ID: 3708177171_529bb4ff1d.jpg
Greedy: skateboarder in the air
Beam, k=3: skateboarder doing trick on ramp
Beam, k=5: the skateboarder does trick on ramp
Beam, k=7: the skateboarder does trick on ramp

Image ID: 2718024196_3ff660416a.jpg
Greedy: young boy in swim trunks is jumping into pool
Beam, k=3: young boy wearing blue swim trunks is jumping into swimming pool
Beam, k=5: young boy in swim trunks dives into pool
Beam, k=7: young boy in swim trunks jumping into swimming pool

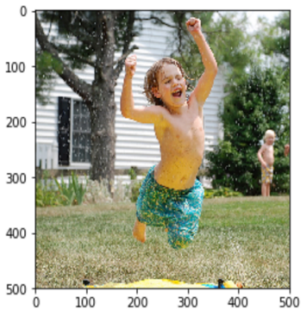


Image ID: 2167644298_100ca79f54.jpg
Greedy: man in black shirt and tie is smiling
Beam, k=3: man and woman wearing black are standing in front of store
Beam, k=5: the man in the black jacket is looking at the camera
Beam, k=7: man and woman pose for picture in front of brick wall

```
Image ID:  3502993968_4ee36afb0e.jpg
Greedy:  man in blue shirt and helmet is riding bike on the beach
Beam, k=3: man on bike is jumping over hill
Beam, k=5: the man is performing trick on bmx bike
Beam, k=7: the man is performing trick on bmx bike
```


```
Image ID:  2453971388_76616b6a82.jpg
Greedy:  boy in swimming trunks is jumping into pool
Beam, k=3: young boy jumping into swimming pool
Beam, k=5: young boy in swim trunks plays in the water
Beam, k=7: young boy jumping into swimming pool
```

To generate captions on our own images, we would first have to generate image features from these images as inputs to our model. We would have to pass the images into the `InceptionV3` model, running code similar to `project_img-features.ipynb`, to create an image features dictionary, and then we can pass these extracted features into our model to produce caption sequences, as seen above.

Takeaways and Future Steps –

We were able to create a caption generator model that provides fairly decent captions for the provided input images. Although our final model did not have particularly high corpus BLEU scores, our evaluation metric, when looking at actual predicted caption results, our network generated sequences that generally captured the context of the original images.

Through the model creation and training process, we ran different iterations of model structures and complexities, as well as hyperparameters. We found that starting with lower learning rates, utilizing `Dropout` layers, and running the RNN on a bidirectional `GRU` layer (rather than multiple layers, or an `LSTM` layer) resulted in preferable results. However, more

experimentation with a different structure (i.e. passing only the text input into the RNN) and further hyperparameter tuning could be fruitful, especially with cross-validation.

We also discovered that our final model, as well as our previous training models, ended up overfitting, despite regularization techniques[3]. It appears that we need to conduct training on larger amounts of data, such as the COCO dataset, or perhaps implement data augmentation to improve results[17]. Additional model improvements could include: training on a larger variety of images (versus everyday images, to allow for more generalization, which could improve testing on our own images), further reducing vocabulary size to speed up training, learning word embeddings rather than using pretrained vectors, and implementing an attention mechanism.

YouTube video link –
https://youtu.be/zPawquEoQFs

References –

1. Karpathy, A., & Fei-Fei, L. (2015). Deep Visual-Semantic Alignments for Generating Image Descriptions. Retrieved 27 July 2019, from https://cs.stanford.edu/people/karpathy/cvpr2015.pdf.
2. Brownlee, J. (2017). How to Develop a Deep Learning Photo Caption Generator from Scratch. Retrieved 27 July 2019, from https://machinelearningmastery.com/develop-a-deep-learning-caption-generation-model-in-python/.
3. Vinyals, O., Toshev, A., Benigo, S., & Erhan, D. (2015). Show and Tell: A Neural Image Caption Generator. Retrieved 27 July 2019, from https://arxiv.org/pdf/1411.4555.pdf.
4. Applications - Keras Documentation. Retrieved 27 July 2019, from https://keras.io/applications/#inceptionv3.
5. ImageNet. Retrieved 27 July 2019, from https://en.wikipedia.org/wiki/ImageNet.
6. IOPub data rate exceeded. The notebook server will temporarily stop sending output to the client in order to avoid crashing it. (2019). Retrieved 27 July 2019, from https://www.drjamesfroggatt.com/python-and-neural-networks/iopub-data-rate-exceeded-the-notebook-server-will-temporarily-stop-sending-output-to-the-client-in-order-to-avoid-crashing-it/.
7. Koehrsen, W. (2018). Neural Network Embeddings Explained. Retrieved 27 July 2019, from https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526.
8. Using word embeddings. Retrieved 27 July 2019, from https://jjallaire.github.io/deep-learning-with-r-notebooks/notebooks/6.1-using-word-embeddings.nb.html.
9. Pennington, J., Socher, R., & Manning, C. (2014). GloVe: Global Vectors for Word Representation. Retrieved 27 July 2019, from https://nlp.stanford.edu/projects/glove/.
10. Lamba, H. (2018). Image Captioning with Keras. Retrieved 27 July 2019, from https://towardsdatascience.com/image-captioning-with-keras-teaching-computers-to-describe-pictures-c88a46a311b8.
11. Tanti, M., Gatt, A., & Camilleri, K. (2018). Where to put the Image in an Image Caption Generator. Retrieved 27 July 2019, from https://arxiv.org/pdf/1703.09137.pdf.
12. Surmenok, P. (2017). Estimating an Optimal Learning Rate For a Deep Neural Network. Retrieved 27 July 2019, from https://towardsdatascience.com/estimating-optimal-learning-rate-for-a-deep-neural-network-ce32f2556ce0.

13. Brownlee, J. (2017). A Gentle Introduction to Calculating the BLEU Score for Text in Python. Retrieved 27 July 2019, from https://machinelearningmastery.com/calculate-bleu-score-for-text-python/.
14. Why is beam search required in sequence to sequence transduction using recurrent neural networks? - Quora. (2017). Retrieved 27 July 2019, from https://www.quora.com/Why-is-beam-search-required-in-sequence-to-sequence-transduction-using-recurrent-neural-networks.
15. Coursera. *Beam Search* [Video]. Retrieved from https://www.coursera.org/lecture/nlp-sequence-models/beam-search-4EtHZ.
16. Mustafa, F. (2018). Keras implementation of Image Captioning Model. Retrieved 27 July 2019, from https://medium.com/@faizanmustafa75/keras-implementation-of-image-captioning-model-3a7ab68e67d4.
17. COCO - Common Objects in Context. Retrieved 27 July 2019, from http://cocodataset.org/#overview.