

<https://machinelearningmastery.com/develop-a-deep-learning-caption-generation-model-in-python/>

Jason Brownlee

How to Develop a Deep Learning Photo Caption Generator from Scratch

by [Jason Brownlee](#) on November 27, 2017 in [Deep Learning for Natural Language Processing](#)

TweetShare

Develop a Deep Learning Model to Automatically Describe Photographs in Python with Keras, Step-by-Step.

Caption generation is a challenging artificial intelligence problem where a textual description must be generated for a given photograph.

It requires both methods from computer vision to understand the content of the image and a language model from the field of natural language processing to turn the understanding of the image into words in the right order. Recently, deep learning methods have achieved state-of-the-art results on examples of this problem.

Deep learning methods have demonstrated state-of-the-art results on caption generation problems. What is most impressive about these methods is a single end-to-end model can be defined to predict a caption, given a photo, instead of requiring sophisticated data preparation or a pipeline of specifically designed models.

In this tutorial, you will discover how to develop a photo captioning deep learning model from scratch.

After completing this tutorial, you will know:

- How to prepare photo and text data for training a deep learning model.
- How to design and train a deep learning caption generation model.
- How to evaluate a train caption generation model and use it to caption entirely new photographs.

Note: This is an excerpt from: “[Deep Learning for Natural Language Processing](#)”.

Take a look, if you want more step-by-step tutorials on getting the most out of deep learning methods when working with text data.

Let's get started.

- **Update Nov/2017:** Added note about a bug introduced in Keras 2.1.0 and 2.1.1 that impacts the code in this tutorial.

- **Update Dec/2017:** Updated a typo in the function name when explaining how to save descriptions to file, thanks Minel.
- **Update Apr/2018:** Added a new section that shows how to train the model using progressive loading for workstations with minimum RAM.
- **Update Feb/2019:** Provided direct links for the Flickr8k_Dataset dataset, as the official site was taken down.
- **Update Jun/2019:** Fixed typo in dataset name. Fixed minor bug in `create_sequences()`.



How to Develop a Deep Learning Caption Generation Model in Python from Scratch

Photo by [Living in Monrovia](#), some rights reserved.

Tutorial Overview

This tutorial is divided into 6 parts; they are:

1. Photo and Caption Dataset
2. Prepare Photo Data
3. Prepare Text Data
4. Develop Deep Learning Model
5. Train With Progressive Loading (**NEW**)
6. Evaluate Model
7. Generate New Captions

Python Environment

This tutorial assumes you have a Python SciPy environment installed, ideally with Python 3.

You must have Keras (2.2 or higher) installed with either the TensorFlow or Theano backend.

The tutorial also assumes you have scikit-learn, Pandas, NumPy, and Matplotlib installed.

If you need help with your environment, see this tutorial:

- [How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda](#)

I recommend running the code on a system with a GPU. You can access GPUs cheaply on Amazon Web Services. Learn how in this tutorial:

- [How to Setup Amazon AWS EC2 GPUs to Train Keras Deep Learning Models \(step-by-step\)](#)

Let's dive in.

Need help with Deep Learning for Text Data?

Take my free 7-day email crash course now (with code).

Click to sign-up and also get a free PDF Ebook version of the course.

Start Your FREE Crash-Course Now

Photo and Caption Dataset

A good dataset to use when getting started with image captioning is the Flickr8K dataset.

The reason is because it is realistic and relatively small so that you can download it and build models on your workstation using a CPU.

The definitive description of the dataset is in the paper "[Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics](#)" from 2013.

The authors describe the dataset as follows:

We introduce a new benchmark collection for sentence-based image description and search, consisting of 8,000 images that are each paired with five different captions which provide clear descriptions of the salient entities and events.

...

The images were chosen from six different Flickr groups, and tend not to contain any well-known people or locations, but were manually selected to depict a variety of scenes and situations.

— [Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics](#), 2013.

The dataset is available for free. You must complete a request form and the links to the dataset will be emailed to you. I would love to link to them for you, but the email address expressly requests: “*Please do not redistribute the dataset*”.

You can use the link below to request the dataset:

- [Dataset Request Form](#)

Within a short time, you will receive an email that contains links to two files:

- **Flickr8k_Dataset.zip** (1 Gigabyte) An archive of all photographs.
- **Flickr8k_text.zip** (2.2 Megabytes) An archive of all text descriptions for photographs.

UPDATE (Feb/2019): The official site seems to have been taken down (although the form still works). Here are some direct download links from my [datasets GitHub repository](#):

- [Flickr8k_Dataset.zip](#)
- [Flickr8k_text.zip](#)

Download the datasets and unzip them into your current working directory. You will have two directories:

- **Flickr8k_Dataset:** Contains 8092 photographs in JPEG format.
- **Flickr8k_text:** Contains a number of files containing different sources of descriptions for the photographs.

The dataset has a pre-defined training dataset (6,000 images), development dataset (1,000 images), and test dataset (1,000 images).

One measure that can be used to evaluate the skill of the model are BLEU scores. For reference, below are some ball-park BLEU scores for skillful models when evaluated on the test dataset (taken from the 2017 paper “[Where to put the Image in an Image Caption Generator](#)”):

- BLEU-1: 0.401 to 0.578.
- BLEU-2: 0.176 to 0.390.
- BLEU-3: 0.099 to 0.260.
- BLEU-4: 0.059 to 0.170.

We describe the BLEU metric more later when we work on evaluating our model.

Next, let's look at how to load the images.

Prepare Photo Data

We will use a pre-trained model to interpret the content of the photos.

There are many models to choose from. In this case, we will use the Oxford Visual Geometry Group, or VGG, model that won the ImageNet competition in 2014. Learn more about the model [here](#):

- [Very Deep Convolutional Networks for Large-Scale Visual Recognition](#)

Keras provides this pre-trained model directly. Note, the first time you use this model, Keras will download the model weights from the Internet, which are about 500 Megabytes. This may take a few minutes depending on your internet connection.

We could use this model as part of a broader image caption model. The problem is, it is a large model and running each photo through the network every time we want to test a new language model configuration (downstream) is redundant.

Instead, we can pre-compute the “photo features” using the pre-trained model and save them to file. We can then load these features later and feed them into our model as the interpretation of a given photo in the dataset. It is no different to running the photo through the full VGG model; it is just we will have done it once in advance.

This is an optimization that will make training our models faster and consume less memory.

We can load the VGG model in Keras using the VGG class. We will remove the last layer from the loaded model, as this is the model used to predict a classification for a photo. We are not interested in classifying images, but we are interested in the internal representation of the photo right before a classification is made. These are the “features” that the model has extracted from the photo.

Keras also provides tools for reshaping the loaded photo into the preferred size for the model (e.g. 3 channel 224 x 224 pixel image).

Below is a function named `extract_features()` that, given a directory name, will load each photo, prepare it for VGG, and collect the predicted features from the VGG model. The image features are a 1-dimensional 4,096 element vector.

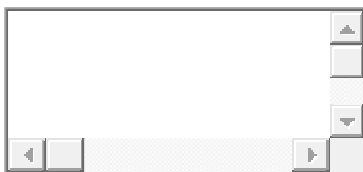
The function returns a dictionary of image identifier to image features.



```
1 # extract features from each photo in the directory
2 def extract_features(directory):
3     # load the model
4     model = VGG16()
5     # re-structure the model
6     model.layers.pop()
7     model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
8     # summarize
9     print(model.summary())
10    # extract features from each photo
11    features = dict()
12    for name in listdir(directory):
13        # load an image from file
14        filename = directory + '/' + name
15        image = load_img(filename, target_size=(224, 224))
16        # convert the image pixels to a numpy array
17        image = img_to_array(image)
18        # reshape data for the model
19        image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
20        # prepare the image for the VGG model
21        image = preprocess_input(image)
22        # get features
23        feature = model.predict(image, verbose=0)
24        # get image id
25        image_id = name.split('.')[0]
26        # store feature
27        features[image_id] = feature
28        print('>%s' % name)
29    return features
```

We can call this function to prepare the photo data for testing our models, then save the resulting dictionary to a file named '*features.pkl*'.

The complete example is listed below.



```
1 from os import listdir
2 from pickle import dump
3 from keras.applications.vgg16 import VGG16
4 from keras.preprocessing.image import load_img
5 from keras.preprocessing.image import img_to_array
6 from keras.applications.vgg16 import preprocess_input
7 from keras.models import Model
8
```

```

9 # extract features from each photo in the directory
10 def extract_features(directory):
11     # load the model
12     model = VGG16()
13     # re-structure the model
14     model.layers.pop()
15     model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
16     # summarize
17     print(model.summary())
18     # extract features from each photo
19     features = dict()
20     for name in listdir(directory):
21         # load an image from file
22         filename = directory + '/' + name
23         image = load_img(filename, target_size=(224, 224))
24         # convert the image pixels to a numpy array
25         image = img_to_array(image)
26         # reshape data for the model
27         image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
28         # prepare the image for the VGG model
29         image = preprocess_input(image)
30         # get features
31         feature = model.predict(image, verbose=0)
32         # get image id
33         image_id = name.split('.')[0]
34         # store feature
35         features[image_id] = feature
36         print('>%s' % name)
37     return features
38
39 # extract features from all images
40 directory = 'Flickr8k_Dataset'
41 features = extract_features(directory)
42 print('Extracted Features: %d' % len(features))
43 # save to file
44 dump(features, open('features.pkl', 'wb'))

```

Running this data preparation step may take a while depending on your hardware, perhaps one hour on the CPU with a modern workstation.

At the end of the run, you will have the extracted features stored in *'features.pkl'* for later use. This file will be about 127 Megabytes in size.

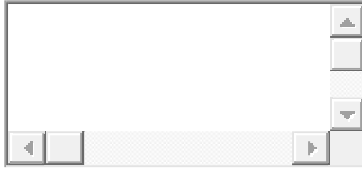
Prepare Text Data

The dataset contains multiple descriptions for each photograph and the text of the descriptions requires some minimal cleaning.

If you are new to cleaning text data, see this post:

- [How to Clean Text for Machine Learning with Python](#)

First, we will load the file containing all of the descriptions.



```
1 # load doc into memory
2 def load_doc(filename):
3     # open the file as read only
4     file = open(filename, 'r')
5     # read all text
6     text = file.read()
7     # close the file
8     file.close()
9     return text
10
11 filename = 'Flickr8k_text/Flickr8k.token.txt'
12 # load descriptions
13 doc = load_doc(filename)
```

Each photo has a unique identifier. This identifier is used on the photo filename and in the text file of descriptions.

Next, we will step through the list of photo descriptions. Below defines a function *load_descriptions()* that, given the loaded document text, will return a dictionary of photo identifiers to descriptions. Each photo identifier maps to a list of one or more textual descriptions.



```
1 # extract descriptions for images
2 def load_descriptions(doc):
3     mapping = dict()
4     # process lines
5     for line in doc.split('\n'):
6         # split line by white space
7         tokens = line.split()
8         if len(tokens) < 2:
9             continue
10        # take the first token as the image id, the rest as the description
11        image_id, image_desc = tokens[0], tokens[1:]
12        # remove filename from image id
13        image_id = image_id.split('.')[0]
14        # convert description tokens back to string
15        image_desc = ' '.join(image_desc)
16        # create the list if needed
17        if image_id not in mapping:
18            mapping[image_id] = list()
19        # store description
20        mapping[image_id].append(image_desc)
21    return mapping
22
```



```

23 # parse descriptions
24 descriptions = load_descriptions(doc)
25 print('Loaded: %d ' % len(descriptions))

```

Next, we need to clean the description text. The descriptions are already tokenized and easy to work with.

We will clean the text in the following ways in order to reduce the size of the vocabulary of words we will need to work with:

- Convert all words to lowercase.
- Remove all punctuation.
- Remove all words that are one character or less in length (e.g. 'a').
- Remove all words with numbers in them.

Below defines the *clean_descriptions()* function that, given the dictionary of image identifiers to descriptions, steps through each description and cleans the text.



```

1 import string
2
3 def clean_descriptions(descriptions):
4     # prepare translation table for removing punctuation
5     table = str.maketrans("", "", string.punctuation)
6     for key, desc_list in descriptions.items():
7         for i in range(len(desc_list)):
8             desc = desc_list[i]
9             # tokenize
10            desc = desc.split()
11            # convert to lower case
12            desc = [word.lower() for word in desc]
13            # remove punctuation from each token
14            desc = [w.translate(table) for w in desc]
15            # remove hanging 's' and 'a'
16            desc = [word for word in desc if len(word)>1]
17            # remove tokens with numbers in them
18            desc = [word for word in desc if word.isalpha()]
19            # store as string
20            desc_list[i] = ' '.join(desc)
21
22 # clean descriptions
23 clean_descriptions(descriptions)

```

Once cleaned, we can summarize the size of the vocabulary.

Ideally, we want a vocabulary that is both expressive and as small as possible. A smaller vocabulary will result in a smaller model that will train faster.

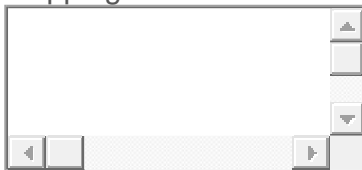
For reference, we can transform the clean descriptions into a set and print its size to get an idea of the size of our dataset vocabulary.



```
1 # convert the loaded descriptions into a vocabulary of words
2 def to_vocabulary(descriptions):
3     # build a list of all description strings
4     all_desc = set()
5     for key in descriptions.keys():
6         [all_desc.update(d.split()) for d in descriptions[key]]
7     return all_desc
8
9 # summarize vocabulary
10 vocabulary = to_vocabulary(descriptions)
11 print('Vocabulary Size: %d' % len(vocabulary))
```

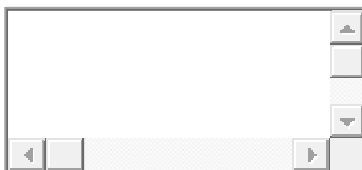
Finally, we can save the dictionary of image identifiers and descriptions to a new file named *descriptions.txt*, with one image identifier and description per line.

Below defines the *save_descriptions()* function that, given a dictionary containing the mapping of identifiers to descriptions and a filename, saves the mapping to file.



```
1 # save descriptions to file, one per line
2 def save_descriptions(descriptions, filename):
3     lines = list()
4     for key, desc_list in descriptions.items():
5         for desc in desc_list:
6             lines.append(key + ' ' + desc)
7     data = '\n'.join(lines)
8     file = open(filename, 'w')
9     file.write(data)
10    file.close()
11
12 # save descriptions
13 save_descriptions(descriptions, 'descriptions.txt')
```

Putting this all together, the complete listing is provided below.



```
1 import string
```

```

2
3 # load doc into memory
4 def load_doc(filename):
5     # open the file as read only
6     file = open(filename, 'r')
7     # read all text
8     text = file.read()
9     # close the file
10    file.close()
11    return text
12
13 # extract descriptions for images
14 def load_descriptions(doc):
15     mapping = dict()
16     # process lines
17     for line in doc.split('\n'):
18         # split line by white space
19         tokens = line.split()
20         if len(line) < 2:
21             continue
22         # take the first token as the image id, the rest as the description
23         image_id, image_desc = tokens[0], tokens[1:]
24         # remove filename from image id
25         image_id = image_id.split('.')[0]
26         # convert description tokens back to string
27         image_desc = ' '.join(image_desc)
28         # create the list if needed
29         if image_id not in mapping:
30             mapping[image_id] = list()
31         # store description
32         mapping[image_id].append(image_desc)
33     return mapping
34
35 def clean_descriptions(descriptions):
36     # prepare translation table for removing punctuation
37     table = str.maketrans("", "", string.punctuation)
38     for key, desc_list in descriptions.items():
39         for i in range(len(desc_list)):
40             desc = desc_list[i]
41             # tokenize
42             desc = desc.split()
43             # convert to lower case
44             desc = [word.lower() for word in desc]
45             # remove punctuation from each token
46             desc = [w.translate(table) for w in desc]
47             # remove hanging 's' and 'a'
48             desc = [word for word in desc if len(word)>1]
49             # remove tokens with numbers in them
50             desc = [word for word in desc if word.isalpha()]
51             # store as string
52             desc_list[i] = ' '.join(desc)
53
54 # convert the loaded descriptions into a vocabulary of words
55 def to_vocabulary(descriptions):
56     # build a list of all description strings
57     all_desc = set()
58     for key in descriptions.keys():
59         [all_desc.update(d.split()) for d in descriptions[key]]
60     return all_desc
61

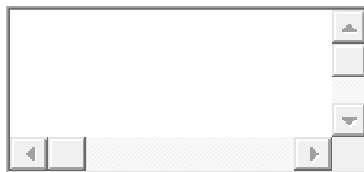
```

```

62 # save descriptions to file, one per line
63 def save_descriptions(descriptions, filename):
64     lines = list()
65     for key, desc_list in descriptions.items():
66         for desc in desc_list:
67             lines.append(key + ' ' + desc)
68     data = '\n'.join(lines)
69     file = open(filename, 'w')
70     file.write(data)
71     file.close()
72
73 filename = 'Flickr8k_text/Flickr8k.token.txt'
74 # load descriptions
75 doc = load_doc(filename)
76 # parse descriptions
77 descriptions = load_descriptions(doc)
78 print('Loaded: %d ' % len(descriptions))
79 # clean descriptions
80 clean_descriptions(descriptions)
81 # summarize vocabulary
82 vocabulary = to_vocabulary(descriptions)
83 print('Vocabulary Size: %d' % len(vocabulary))
84 # save to file
85 save_descriptions(descriptions, 'descriptions.txt')

```

Running the example first prints the number of loaded photo descriptions (8,092) and the size of the clean vocabulary (8,763 words).



```

1 Loaded: 8,092
2 Vocabulary Size: 8,763

```

Finally, the clean descriptions are written to *'descriptions.txt'*.

Taking a look at the file, we can see that the descriptions are ready for modeling. The order of descriptions in your file may vary.



```

1 2252123185_487f21e336 bunch on people are seated in stadium
2 2252123185_487f21e336 crowded stadium is full of people watching an event
3 2252123185_487f21e336 crowd of people fill up packed stadium
4 2252123185_487f21e336 crowd sitting in an indoor stadium
5 2252123185_487f21e336 stadium full of people watch game
6 ...

```

Develop Deep Learning Model

In this section, we will define the deep learning model and fit it on the training dataset.

This section is divided into the following parts:

1. Loading Data.
2. Defining the Model.
3. Fitting the Model.
4. Complete Example.

Loading Data

First, we must load the prepared photo and text data so that we can use it to fit the model.

We are going to train the data on all of the photos and captions in the training dataset.

While training, we are going to monitor the performance of the model on the development dataset and use that performance to decide when to [save models to file](#).

The train and development dataset have been predefined in the *Flickr_8k.trainImages.txt* and *Flickr_8k.devImages.txt* files respectively, that both contain lists of photo file names. From these file names, we can extract the photo identifiers and use these identifiers to filter photos and descriptions for each set.

The function *load_set()* below will load a pre-defined set of identifiers given the train or development sets filename.



```
1 # load doc into memory
2 def load_doc(filename):
3     # open the file as read only
4     file = open(filename, 'r')
5     # read all text
6     text = file.read()
7     # close the file
8     file.close()
9     return text
10
11 # load a pre-defined list of photo identifiers
12 def load_set(filename):
13     doc = load_doc(filename)
14     dataset = list()
15     # process line by line
16     for line in doc.split('\n'):
17         # skip empty lines
18         if len(line) < 1:
19             continue
20         # get the image identifier
21         identifier = line.split('.')[0]
22         dataset.append(identifier)
```

23 return set(dataset)

Now, we can load the photos and descriptions using the pre-defined set of train or development identifiers.

Below is the function *load_clean_descriptions()* that loads the cleaned text descriptions from '*descriptions.txt*' for a given set of identifiers and returns a dictionary of identifiers to lists of text descriptions.

The model we will develop will generate a caption given a photo, 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 will use the strings '*startseq*' and '*endseq*' for this purpose. These tokens are added to the loaded descriptions as they are loaded. It is important to do this now before we encode the text so that the tokens are also encoded correctly.

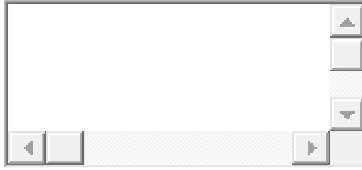


```
1 # load clean descriptions into memory
2 def load_clean_descriptions(filename, dataset):
3     # load document
4     doc = load_doc(filename)
5     descriptions = dict()
6     for line in doc.split('\n'):
7         # split line by white space
8         tokens = line.split()
9         # split id from description
10        image_id, image_desc = tokens[0], tokens[1:]
11        # skip images not in the set
12        if image_id in dataset:
13            # create list
14            if image_id not in descriptions:
15                descriptions[image_id] = list()
16            # wrap description in tokens
17            desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
18            # store
19            descriptions[image_id].append(desc)
20    return descriptions
```

Next, we can load the photo features for a given dataset.

Below defines a function named *load_photo_features()* that loads the entire set of photo descriptions, then returns the subset of interest for a given set of photo identifiers.

This is not very efficient; nevertheless, this will get us up and running quickly.



```
1 # load photo features
2 def load_photo_features(filename, dataset):
3     # load all features
4     all_features = load(open(filename, 'rb'))
5     # filter features
6     features = {k: all_features[k] for k in dataset}
7     return features
```

We can pause here and test everything developed so far.

The complete code example is listed below.



```
1 from pickle import load
2
3 # load doc into memory
4 def load_doc(filename):
5     # open the file as read only
6     file = open(filename, 'r')
7     # read all text
8     text = file.read()
9     # close the file
10    file.close()
11    return text
12
13 # load a pre-defined list of photo identifiers
14 def load_set(filename):
15     doc = load_doc(filename)
16     dataset = list()
17     # process line by line
18     for line in doc.split('\n'):
19         # skip empty lines
20         if len(line) < 1:
21             continue
22         # get the image identifier
23         identifier = line.split('.')[0]
24         dataset.append(identifier)
25     return set(dataset)
26
27 # load clean descriptions into memory
28 def load_clean_descriptions(filename, dataset):
29     # load document
30     doc = load_doc(filename)
31     descriptions = dict()
32     for line in doc.split('\n'):
33         # split line by white space
```

```

34         tokens = line.split()
35         # split id from description
36         image_id, image_desc = tokens[0], tokens[1:]
37         # skip images not in the set
38         if image_id in dataset:
39             # create list
40             if image_id not in descriptions:
41                 descriptions[image_id] = list()
42             # wrap description in tokens
43             desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
44             # store
45             descriptions[image_id].append(desc)
46     return descriptions
47
48 # load photo features
49 def load_photo_features(filename, dataset):
50     # load all features
51     all_features = load(open(filename, 'rb'))
52     # filter features
53     features = {k: all_features[k] for k in dataset}
54     return features
55
56 # load training dataset (6K)
57 filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
58 train = load_set(filename)
59 print('Dataset: %d' % len(train))
60 # descriptions
61 train_descriptions = load_clean_descriptions('descriptions.txt', train)
62 print('Descriptions: train=%d' % len(train_descriptions))
63 # photo features
64 train_features = load_photo_features('features.pkl', train)
65 print('Photos: train=%d' % len(train_features))

```

Running this example first loads the 6,000 photo identifiers in the test dataset. These features are then used to filter and load the cleaned description text and the pre-computed photo features.

We are nearly there.



```

1 Dataset: 6,000
2 Descriptions: train=6,000
3 Photos: train=6,000

```

The description text will need to be encoded to numbers before it can be presented to the model as in input or compared to the model's predictions.

The first step in encoding the data 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 description data.

Below defines the *to_lines()* to convert the dictionary of descriptions into a list of strings and the *create_tokenizer()* function that will fit a Tokenizer given the loaded photo description text.



```

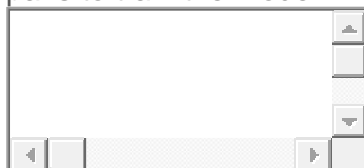
1 # convert a dictionary of clean descriptions to a list of descriptions
2 def to_lines(descriptions):
3     all_desc = list()
4     for key in descriptions.keys():
5         [all_desc.append(d) for d in descriptions[key]]
6     return all_desc
7
8 # fit a tokenizer given caption descriptions
9 def create_tokenizer(descriptions):
10     lines = to_lines(descriptions)
11     tokenizer = Tokenizer()
12     tokenizer.fit_on_texts(lines)
13     return tokenizer
14
15 # prepare tokenizer
16 tokenizer = create_tokenizer(train_descriptions)
17 vocab_size = len(tokenizer.word_index) + 1
18 print('Vocabulary Size: %d' % vocab_size)

```

We can now encode the text.

Each description will be split into words. The model will be provided one word and the photo and generate the next word. Then the first two words of the description 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:



1 X1,	X2 (text sequence),	y (word)
2 photo	startseq,	little
3 photo	startseq, little,	girl
4 photo	startseq, little, girl,	running

```

5 photo    startseq, little, girl, running,                in
6 photo    startseq, little, girl, running, in,            field
7 photo    startseq, little, girl, running, in, field, endseq

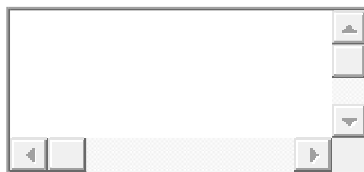
```

Later, when the model is used to generate descriptions, the generated words will be concatenated and recursively provided as input to generate a caption for an image.

The function below named *create_sequences()*, given the tokenizer, a maximum sequence length, and the dictionary of all descriptions and photos, will transform the data into input-output pairs of data for training the model. There are two input arrays to the model: one for photo features and one for the encoded text. There is one output for the model which is the encoded next word in the text sequence.

The input text is encoded as integers, which will be fed to a word embedding layer. The photo features will be fed directly to another part of the model. The model will output a prediction, which will be a probability distribution over all words in the vocabulary.

The output data will therefore be a one-hot encoded version of each word, representing an idealized probability distribution with 0 values at all word positions except the actual word position, which has a value of 1.

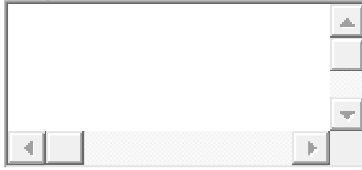


```

1 # create sequences of images, input sequences and output words for an image
2 def create_sequences(tokenizer, max_length, descriptions, photos, vocab_size):
3     X1, X2, y = list(), list(), list()
4     # walk through each image identifier
5     for key, desc_list in descriptions.items():
6         # walk through each description for the image
7         for desc in desc_list:
8             # encode the sequence
9             seq = tokenizer.texts_to_sequences([desc])[0]
10            # split one sequence into multiple X,y pairs
11            for i in range(1, len(seq)):
12                # split into input and output pair
13                in_seq, out_seq = seq[:i], seq[i]
14                # pad input sequence
15                in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
16                # encode output sequence
17                out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
18                # store
19                X1.append(photos[key][0])
20                X2.append(in_seq)
21                y.append(out_seq)
22    return array(X1), array(X2), array(y)

```

We will need to calculate the maximum number of words in the longest description. A short helper function named `max_length()` is defined below.



```
1 # calculate the length of the description with the most words
2 def max_length(descriptions):
3     lines = to_lines(descriptions)
4     return max(len(d.split()) for d in lines)
```

We now have enough to load the data for the training and development datasets and transform the loaded data into input-output pairs for fitting a deep learning model.

Defining the Model

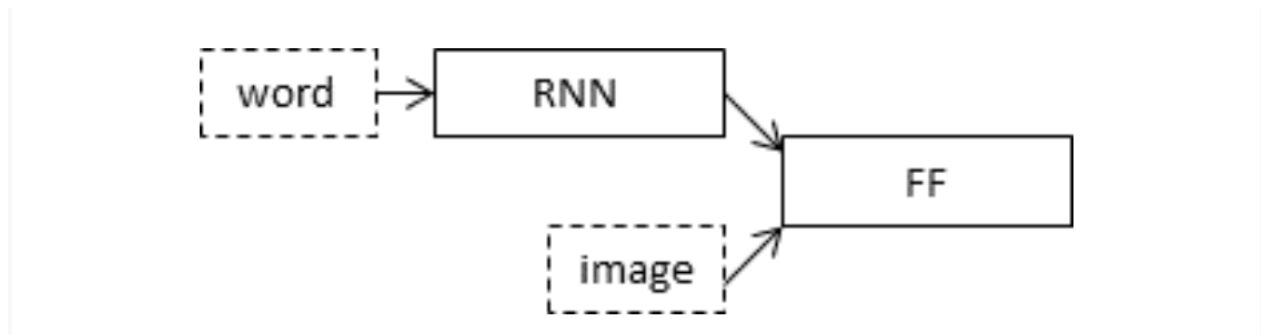
We will define a deep learning based on the “*merge-model*” described by Marc Tanti, et al. in their 2017 papers:

- [Where to put the Image in an Image Caption Generator](#), 2017.
- [What is the Role of Recurrent Neural Networks \(RNNs\) in an Image Caption Generator?](#), 2017.

For a gentle introduction to this architecture, see the post:

- [Caption Generation with the Inject and Merge Architectures for the Encoder-Decoder Model](#)

The authors provide a nice schematic of the model, reproduced below.



Schematic of the Merge Model For Image Captioning

We will describe the model in three parts:

- **Photo Feature Extractor.** This is a 16-layer VGG model pre-trained on the ImageNet dataset. We have pre-processed the photos with the VGG model (without the output layer) and will use the extracted features predicted by this model as input.
- **Sequence Processor.** This is a word embedding layer for handling the text input, followed by a Long Short-Term Memory (LSTM) recurrent neural network layer.

- **Decoder** (for lack of a better name). Both the feature extractor and sequence processor output a fixed-length vector. These are merged together and processed by a Dense layer to make a final prediction.

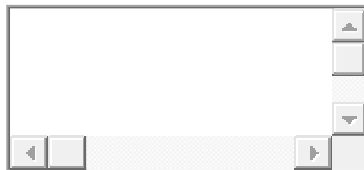
The Photo Feature Extractor model expects input photo features to be a vector of 4,096 elements. These are processed by a Dense layer to produce a 256 element representation of the photo.

The Sequence Processor model expects input sequences with a pre-defined length (34 words) which are fed into an Embedding layer that uses a mask to ignore padded values. This is followed by an LSTM layer with 256 memory units.

Both the input models produce a 256 element vector. Further, both input models use regularization in the form of 50% dropout. This is to reduce overfitting the training dataset, as this model configuration learns very fast.

The Decoder model merges the vectors from both input models using an addition operation. This is then fed to a Dense 256 neuron layer and then to a final output Dense layer that makes a softmax prediction over the entire output vocabulary for the next word in the sequence.

The function below named *define_model()* defines and returns the model ready to be fit.



```

1 # define the captioning model
2 def define_model(vocab_size, max_length):
3     # feature extractor model
4     inputs1 = Input(shape=(4096,))
5     fe1 = Dropout(0.5)(inputs1)
6     fe2 = Dense(256, activation='relu')(fe1)
7     # sequence model
8     inputs2 = Input(shape=(max_length,))
9     se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
10    se2 = Dropout(0.5)(se1)
11    se3 = LSTM(256)(se2)
12    # decoder model
13    decoder1 = add([fe2, se3])
14    decoder2 = Dense(256, activation='relu')(decoder1)
15    outputs = Dense(vocab_size, activation='softmax')(decoder2)
16    # tie it together [image, seq] [word]
17    model = Model(inputs=[inputs1, inputs2], outputs=outputs)
18    model.compile(loss='categorical_crossentropy', optimizer='adam')
19    # summarize model
20    print(model.summary())
21    plot_model(model, to_file='model.png', show_shapes=True)

```

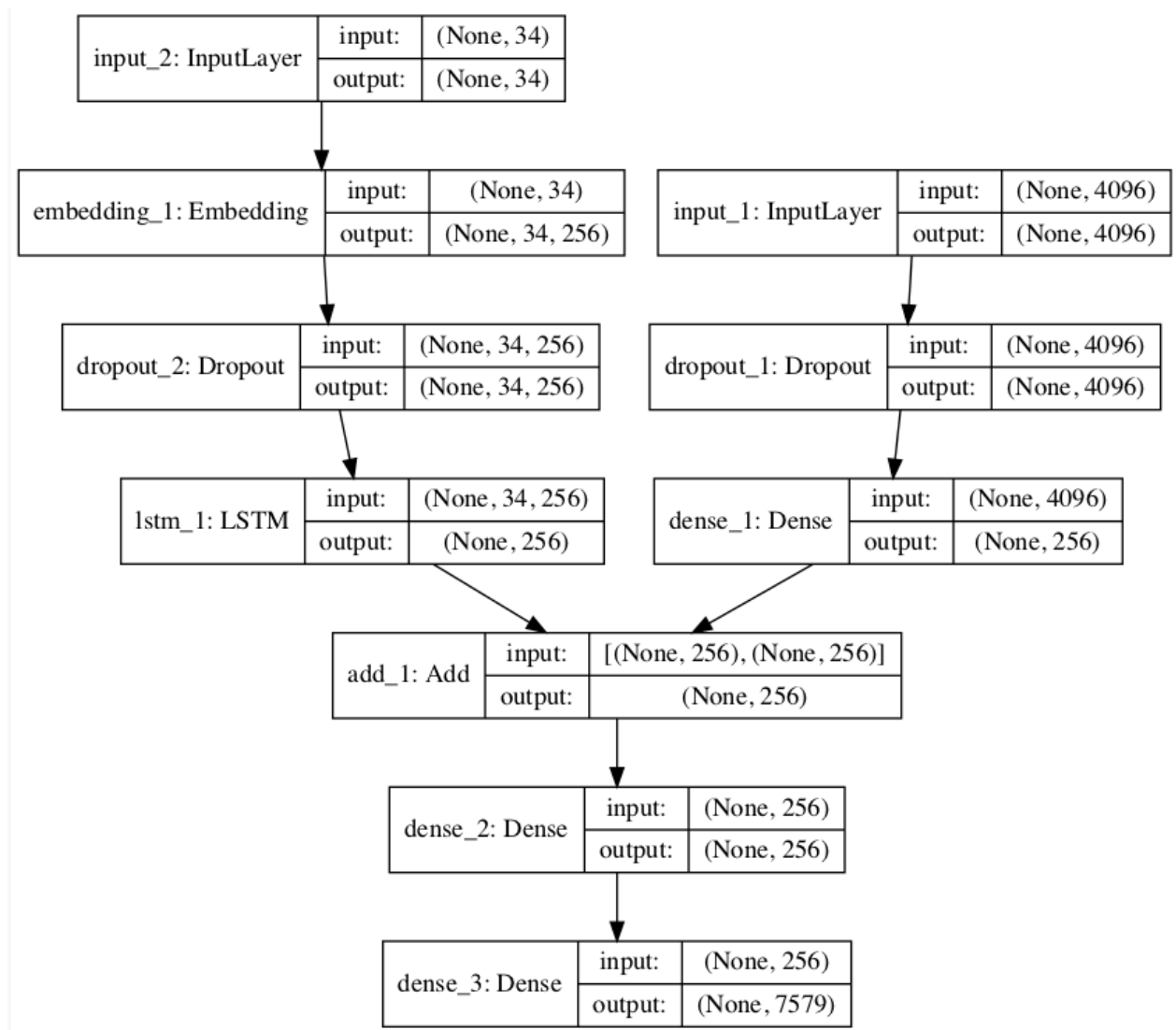
22 return model

To get a sense for the structure of the model, specifically the shapes of the layers, see the summary listed below.



```
1
2
3
4
5
6
7
8
9
1
1 Layer (type)          Output Shape      Param #   Connected to
0 =====
1 =====
1 input_2 (InputLayer)   (None, 34)        0
1
2 input_1 (InputLayer)   (None, 4096)      0
1
3 embedding_1 (Embedding) (None, 34, 256)   1940224   input_2[0][0]
1
4 dropout_1 (Dropout)    (None, 4096)      0         input_1[0][0]
1
5 dropout_2 (Dropout)    (None, 34, 256)   0         embedding_1[0][0]
1
6 dense_1 (Dense)        (None, 256)       1048832   dropout_1[0][0]
1
7 lstm_1 (LSTM)          (None, 256)       525312    dropout_2[0][0]
1
8 add_1 (Add)            (None, 256)       0         dense_1[0][0]
1                                lstm_1[0][0]
9
2 dense_2 (Dense)        (None, 256)       65792     add_1[0][0]
0
2 dense_3 (Dense)        (None, 7579)      1947803   dense_2[0][0]
1
2 =====
2 =====
2 Total params: 5,527,963
2 Trainable params: 5,527,963
3 Non-trainable params: 0
2
4
2
5
2
6
2
7
2
8
```

We also create a plot to visualize the structure of the network that better helps understand the two streams of input.



Plot of the Caption Generation Deep Learning Model

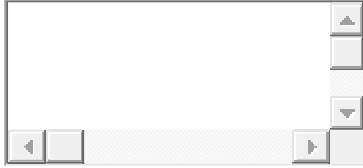
Fitting the Model

Now that we know how to define the model, we can fit it on the training dataset.

The model learns fast and quickly overfits the training dataset. For this reason, we will monitor the skill of the trained model on the holdout development dataset. When the skill of the model on the development dataset improves at the end of an epoch, we will save the whole model to file.

At the end of the run, we can then use the saved model with the best skill on the training dataset as our final model.

We can do this by defining a *ModelCheckpoint* in Keras and specifying it to monitor the minimum loss on the validation dataset and save the model to a file that has both the training and validation loss in the filename.



```
1 # define checkpoint callback
2 filepath = 'model-ep{epoch:03d}-loss{loss:.3f}-val_loss{val_loss:.3f}.h5'
3 checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose=1, save_best_only=True, mode='min')
```

We can then specify the checkpoint in the call to *fit()* via the *callbacks* argument. We must also specify the development dataset in *fit()* via the *validation_data* argument.

We will only fit the model for 20 epochs, but given the amount of training data, each epoch may take 30 minutes on modern hardware.



```
1 # fit model
2 model.fit([X1train, X2train], ytrain, epochs=20, verbose=2, callbacks=[checkpoint], validation_data=([X1test, X2test],
ytest))
```

Complete Example

The complete example for fitting the model on the training data is listed below.



```
1 from numpy import array
2 from pickle import load
3 from keras.preprocessing.text import Tokenizer
4 from keras.preprocessing.sequence import pad_sequences
5 from keras.utils import to_categorical
6 from keras.utils import plot_model
7 from keras.models import Model
8 from keras.layers import Input
9 from keras.layers import Dense
10 from keras.layers import LSTM
```

```

11 from keras.layers import Embedding
12 from keras.layers import Dropout
13 from keras.layers.merge import add
14 from keras.callbacks import ModelCheckpoint
15
16 # load doc into memory
17 def load_doc(filename):
18     # open the file as read only
19     file = open(filename, 'r')
20     # read all text
21     text = file.read()
22     # close the file
23     file.close()
24     return text
25
26 # load a pre-defined list of photo identifiers
27 def load_set(filename):
28     doc = load_doc(filename)
29     dataset = list()
30     # process line by line
31     for line in doc.split("\n"):
32         # skip empty lines
33         if len(line) < 1:
34             continue
35         # get the image identifier
36         identifier = line.split('.')[0]
37         dataset.append(identifier)
38     return set(dataset)
39
40 # load clean descriptions into memory
41 def load_clean_descriptions(filename, dataset):
42     # load document
43     doc = load_doc(filename)
44     descriptions = dict()
45     for line in doc.split("\n"):
46         # split line by white space
47         tokens = line.split()
48         # split id from description
49         image_id, image_desc = tokens[0], tokens[1:]
50         # skip images not in the set
51         if image_id in dataset:
52             # create list
53             if image_id not in descriptions:
54                 descriptions[image_id] = list()
55             # wrap description in tokens
56             desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
57             # store
58             descriptions[image_id].append(desc)
59     return descriptions
60
61 # load photo features
62 def load_photo_features(filename, dataset):
63     # load all features
64     all_features = load(open(filename, 'rb'))
65     # filter features
66     features = {k: all_features[k] for k in dataset}
67     return features
68
69 # covert a dictionary of clean descriptions to a list of descriptions
70 def to_lines(descriptions):

```



```

71     all_desc = list()
72     for key in descriptions.keys():
73         [all_desc.append(d) for d in descriptions[key]]
74     return all_desc
75
76 # fit a tokenizer given caption descriptions
77 def create_tokenizer(descriptions):
78     lines = to_lines(descriptions)
79     tokenizer = Tokenizer()
80     tokenizer.fit_on_texts(lines)
81     return tokenizer
82
83 # calculate the length of the description with the most words
84 def max_length(descriptions):
85     lines = to_lines(descriptions)
86     return max(len(d.split()) for d in lines)
87
88 # create sequences of images, input sequences and output words for an image
89 def create_sequences(tokenizer, max_length, descriptions, photos, vocab_size):
90     X1, X2, y = list(), list(), list()
91     # walk through each image identifier
92     for key, desc_list in descriptions.items():
93         # walk through each description for the image
94         for desc in desc_list:
95             # encode the sequence
96             seq = tokenizer.texts_to_sequences([desc])[0]
97             # split one sequence into multiple X,y pairs
98             for i in range(1, len(seq)):
99                 # split into input and output pair
100                 in_seq, out_seq = seq[:i], seq[i]
101                 # pad input sequence
102                 in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
103                 # encode output sequence
104                 out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
105                 # store
106                 X1.append(photos[key][0])
107                 X2.append(in_seq)
108                 y.append(out_seq)
109     return array(X1), array(X2), array(y)
110
111 # define the captioning model
112 def define_model(vocab_size, max_length):
113     # feature extractor model
114     inputs1 = Input(shape=(4096,))
115     fe1 = Dropout(0.5)(inputs1)
116     fe2 = Dense(256, activation='relu')(fe1)
117     # sequence model
118     inputs2 = Input(shape=(max_length,))
119     se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
120     se2 = Dropout(0.5)(se1)
121     se3 = LSTM(256)(se2)
122     # decoder model
123     decoder1 = add([fe2, se3])
124     decoder2 = Dense(256, activation='relu')(decoder1)
125     outputs = Dense(vocab_size, activation='softmax')(decoder2)
126     # tie it together [image, seq] [word]
127     model = Model(inputs=[inputs1, inputs2], outputs=outputs)
128     model.compile(loss='categorical_crossentropy', optimizer='adam')
129     # summarize model
130     print(model.summary())

```

```

131         plot_model(model, to_file='model.png', show_shapes=True)
132     return model
133
134 # train dataset
135
136 # load training dataset (6K)
137 filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
138 train = load_set(filename)
139 print('Dataset: %d' % len(train))
140 # descriptions
141 train_descriptions = load_clean_descriptions('descriptions.txt', train)
142 print('Descriptions: train=%d' % len(train_descriptions))
143 # photo features
144 train_features = load_photo_features('features.pkl', train)
145 print('Photos: train=%d' % len(train_features))
146 # prepare tokenizer
147 tokenizer = create_tokenizer(train_descriptions)
148 vocab_size = len(tokenizer.word_index) + 1
149 print('Vocabulary Size: %d' % vocab_size)
150 # determine the maximum sequence length
151 max_length = max_length(train_descriptions)
152 print('Description Length: %d' % max_length)
153 # prepare sequences
154 X1train, X2train, ytrain = create_sequences(tokenizer, max_length, train_descriptions, train_features, vocab_size)
155
156 # dev dataset
157
158 # load test set
159 filename = 'Flickr8k_text/Flickr_8k.devImages.txt'
160 test = load_set(filename)
161 print('Dataset: %d' % len(test))
162 # descriptions
163 test_descriptions = load_clean_descriptions('descriptions.txt', test)
164 print('Descriptions: test=%d' % len(test_descriptions))
165 # photo features
166 test_features = load_photo_features('features.pkl', test)
167 print('Photos: test=%d' % len(test_features))
168 # prepare sequences
169 X1test, X2test, ytest = create_sequences(tokenizer, max_length, test_descriptions, test_features, vocab_size)
170
171 # fit model
172
173 # define the model
174 model = define_model(vocab_size, max_length)
175 # define checkpoint callback
176 filepath = 'model-ep{epoch:03d}-loss{loss:.3f}-val_loss{val_loss:.3f}.h5'
177 checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose=1, save_best_only=True, mode='min')
178 # fit model
179 model.fit([X1train, X2train], ytrain, epochs=20, verbose=2, callbacks=[checkpoint], validation_data=([X1test,
    X2test], ytest))

```

Running the example first prints a summary of the loaded training and development datasets.



1 Dataset: 6,000
2 Descriptions: train=6,000
3 Photos: train=6,000
4 Vocabulary Size: 7,579
5 Description Length: 34
6 Dataset: 1,000
7 Descriptions: test=1,000
8 Photos: test=1,000

After the summary of the model, we can get an idea of the total number of training and validation (development) input-output pairs.



1 Train on 306,404 samples, validate on 50,903 samples

The model then runs, saving the best model to .h5 files along the way.

On my run, the best validation results were saved to the file:

- *model-ep002-loss3.245-val_loss3.612.h5*

This model was saved at the end of epoch 2 with a loss of 3.245 on the training dataset and a loss of 3.612 on the development dataset

Your specific results will vary.

Let me know what you get in the comments below.

If you ran the example on AWS, copy the model file back to your current working directory.

If you need help with commands on AWS, see the post:

- [10 Command Line Recipes for Deep Learning on Amazon Web Services](#)

Did you get an error like:



1 Memory Error

If so, see the next section.

Train With Progressive Loading

Note: If you had no problems in the previous section, [please skip this section](#). This section is for those who do not have enough memory to train the model as described in the previous section (e.g. cannot use AWS EC2 for whatever reason).

The training of the caption model does assume you have a lot of RAM.

The code in the previous section is not memory efficient and assumes you are running on a large EC2 instance with 32GB or 64GB of RAM. If you are running the code on a workstation of 8GB of RAM, you cannot train the model.

A workaround is to use progressive loading. This was discussed in detail in the second-last section titled “*Progressive Loading*” in the post:

- [How to Prepare a Photo Caption Dataset for Training a Deep Learning Model](#)

I recommend reading that section before continuing.

If you want to use progressive loading, to train this model, this section will show you how.

The first step is we must define a function that we can use as the data generator.

We will keep things very simple and have the data generator yield one photo’s worth of data per batch. This will be all of the sequences generated for a photo and its set of descriptions.

The function below `data_generator()` will be the data generator and will take the loaded textual descriptions, photo features, tokenizer and max length. Here, I assume that you can fit this training data in memory, which I believe 8GB of RAM should be more than capable. How does this work? Read the post I just mentioned above that introduces data generators.



```
1 # data generator, intended to be used in a call to model.fit_generator()
2 def data_generator(descriptions, photos, tokenizer, max_length, vocab_size):
3     # loop for ever over images
4     while 1:
5         for key, desc_list in descriptions.items():
6             # retrieve the photo feature
7             photo = photos[key][0]
8             in_img, in_seq, out_word = create_sequences(tokenizer, max_length, desc_list, photo,
9             vocab_size)
10            yield [[in_img, in_seq], out_word]
```

You can see that we are calling the `create_sequence()` function to create a batch worth of data for a single photo rather than an entire dataset. This means that we must update the `create_sequences()` function to delete the “iterate over all descriptions” for-loop. The updated function is as follows:



```
1 # create sequences of images, input sequences and output words for an image
2 def create_sequences(tokenizer, max_length, desc_list, photo, vocab_size):
3     X1, X2, y = list(), list(), list()
4     # walk through each description for the image
5     for desc in desc_list:
6         # encode the sequence
7         seq = tokenizer.texts_to_sequences([desc])[0]
8         # split one sequence into multiple X,y pairs
9         for i in range(1, len(seq)):
10            # split into input and output pair
11            in_seq, out_seq = seq[:i], seq[i]
12            # pad input sequence
13            in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
14            # encode output sequence
15            out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
16            # store
17            X1.append(photo)
18            X2.append(in_seq)
19            y.append(out_seq)
20    return array(X1), array(X2), array(y)
```

We now have pretty much everything we need.

Note, this is a very basic data generator. The big memory saving it offers is to not have the unrolled sequences of train and test data in memory prior to fitting the model, that these samples (e.g. results from `create_sequences()`) are created as needed per photo.

Some off-the-cuff ideas for further improving this data generator include:

- Randomize the order of photos each epoch.
- Work with a list of photo ids and load text and photo data as needed to cut even further back on memory.
- Yield more than one photo’s worth of samples per batch.

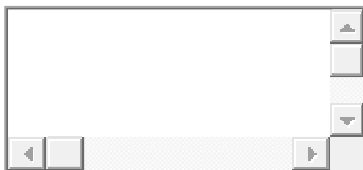
I have experienced with these variations myself in the past. Let me know if you do and how you go in the comments.

You can sanity check a data generator by calling it directly, as follows:



```
1 # test the data generator
2 generator = data_generator(train_descriptions, train_features, tokenizer, max_length, vocab_size)
3 inputs, outputs = next(generator)
4 print(inputs[0].shape)
5 print(inputs[1].shape)
6 print(outputs.shape)
```

Running this sanity check will show what one batch worth of sequences looks like, in this case 47 samples to train on for the first photo.



```
1 (47, 4096)
2 (47, 34)
3 (47, 7579)
```

Finally, we can use the *fit_generator()* function on the model to train the model with this data generator.

In this simple example we will discard the loading of the development dataset and model checkpointing and simply save the model after each training epoch. You can then go back and load/evaluate each saved model after training to find the one we the lowest loss that you can then use in the next section.

The code to train the model with the data generator is as follows:



```
1 # train the model, run epochs manually and save after each epoch
2 epochs = 20
3 steps = len(train_descriptions)
4 for i in range(epochs):
5     # create the data generator
6     generator = data_generator(train_descriptions, train_features, tokenizer, max_length, vocab_size)
7     # fit for one epoch
8     model.fit_generator(generator, epochs=1, steps_per_epoch=steps, verbose=1)
9     # save model
10    model.save('model_' + str(i) + '.h5')
```

That's it. You can now train the model using progressive loading and save a ton of RAM. This may also be a lot slower.

The complete updated example with progressive loading (use of the data generator) for training the caption generation model is listed below.



```
1  from numpy import array
2  from pickle import load
3  from keras.preprocessing.text import Tokenizer
4  from keras.preprocessing.sequence import pad_sequences
5  from keras.utils import to_categorical
6  from keras.utils import plot_model
7  from keras.models import Model
8  from keras.layers import Input
9  from keras.layers import Dense
10 from keras.layers import LSTM
11 from keras.layers import Embedding
12 from keras.layers import Dropout
13 from keras.layers.merge import add
14 from keras.callbacks import ModelCheckpoint
15
16 # load doc into memory
17 def load_doc(filename):
18     # open the file as read only
19     file = open(filename, 'r')
20     # read all text
21     text = file.read()
22     # close the file
23     file.close()
24     return text
25
26 # load a pre-defined list of photo identifiers
27 def load_set(filename):
28     doc = load_doc(filename)
29     dataset = list()
30     # process line by line
31     for line in doc.split('\n'):
32         # skip empty lines
33         if len(line) < 1:
34             continue
35         # get the image identifier
36         identifier = line.split('.')[0]
37         dataset.append(identifier)
38     return set(dataset)
39
40 # load clean descriptions into memory
41 def load_clean_descriptions(filename, dataset):
42     # load document
43     doc = load_doc(filename)
44     descriptions = dict()
```

```

45     for line in doc.split('\n'):
46         # split line by white space
47         tokens = line.split()
48         # split id from description
49         image_id, image_desc = tokens[0], tokens[1:]
50         # skip images not in the set
51         if image_id in dataset:
52             # create list
53             if image_id not in descriptions:
54                 descriptions[image_id] = list()
55             # wrap description in tokens
56             desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
57             # store
58             descriptions[image_id].append(desc)
59     return descriptions
60
61 # load photo features
62 def load_photo_features(filename, dataset):
63     # load all features
64     all_features = load(open(filename, 'rb'))
65     # filter features
66     features = {k: all_features[k] for k in dataset}
67     return features
68
69 # covert a dictionary of clean descriptions to a list of descriptions
70 def to_lines(descriptions):
71     all_desc = list()
72     for key in descriptions.keys():
73         [all_desc.append(d) for d in descriptions[key]]
74     return all_desc
75
76 # fit a tokenizer given caption descriptions
77 def create_tokenizer(descriptions):
78     lines = to_lines(descriptions)
79     tokenizer = Tokenizer()
80     tokenizer.fit_on_texts(lines)
81     return tokenizer
82
83 # calculate the length of the description with the most words
84 def max_length(descriptions):
85     lines = to_lines(descriptions)
86     return max(len(d.split()) for d in lines)
87
88 # create sequences of images, input sequences and output words for an image
89 def create_sequences(tokenizer, max_length, desc_list, photo, vocab_size):
90     X1, X2, y = list(), list(), list()
91     # walk through each description for the image
92     for desc in desc_list:
93         # encode the sequence
94         seq = tokenizer.texts_to_sequences([desc])[0]
95         # split one sequence into multiple X,y pairs
96         for i in range(1, len(seq)):
97             # split into input and output pair
98             in_seq, out_seq = seq[:i], seq[i]
99             # pad input sequence
100            in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
101            # encode output sequence
102            out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
103            # store
104            X1.append(photo)

```



```

105             X2.append(in_seq)
106             y.append(out_seq)
107     return array(X1), array(X2), array(y)
108
109 # define the captioning model
110 def define_model(vocab_size, max_length):
111     # feature extractor model
112     inputs1 = Input(shape=(4096,))
113     fe1 = Dropout(0.5)(inputs1)
114     fe2 = Dense(256, activation='relu')(fe1)
115     # sequence model
116     inputs2 = Input(shape=(max_length,))
117     se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
118     se2 = Dropout(0.5)(se1)
119     se3 = LSTM(256)(se2)
120     # decoder model
121     decoder1 = add([fe2, se3])
122     decoder2 = Dense(256, activation='relu')(decoder1)
123     outputs = Dense(vocab_size, activation='softmax')(decoder2)
124     # tie it together [image, seq] [word]
125     model = Model(inputs=[inputs1, inputs2], outputs=outputs)
126     # compile model
127     model.compile(loss='categorical_crossentropy', optimizer='adam')
128     # summarize model
129     model.summary()
130     plot_model(model, to_file='model.png', show_shapes=True)
131     return model
132
133 # data generator, intended to be used in a call to model.fit_generator()
134 def data_generator(descriptions, photos, tokenizer, max_length, vocab_size):
135     # loop for ever over images
136     while 1:
137         for key, desc_list in descriptions.items():
138             # retrieve the photo feature
139             photo = photos[key][0]
140             in_img, in_seq, out_word = create_sequences(tokenizer, max_length, desc_list, photo,
141 vocab_size)
142             yield [[in_img, in_seq], out_word]
143
144 # load training dataset (6K)
145 filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
146 train = load_set(filename)
147 print('Dataset: %d' % len(train))
148 # descriptions
149 train_descriptions = load_clean_descriptions('descriptions.txt', train)
150 print('Descriptions: train=%d' % len(train_descriptions))
151 # photo features
152 train_features = load_photo_features('features.pkl', train)
153 print('Photos: train=%d' % len(train_features))
154 # prepare tokenizer
155 tokenizer = create_tokenizer(train_descriptions)
156 vocab_size = len(tokenizer.word_index) + 1
157 print('Vocabulary Size: %d' % vocab_size)
158 # determine the maximum sequence length
159 max_length = max_length(train_descriptions)
160 print('Description Length: %d' % max_length)
161
162 # define the model
163 model = define_model(vocab_size, max_length)
164 # train the model, run epochs manually and save after each epoch

```

```

165 epochs = 20
166 steps = len(train_descriptions)
167 for i in range(epochs):
168     # create the data generator
169     generator = data_generator(train_descriptions, train_features, tokenizer, max_length, vocab_size)
170     # fit for one epoch
171     model.fit_generator(generator, epochs=1, steps_per_epoch=steps, verbose=1)
172     # save model
    model.save('model_' + str(i) + '.h5')

```

Perhaps evaluate each saved model and choose the one final model with the lowest loss on a holdout dataset. The next section may help with this.

Did you use this new addition to the tutorial?

How did you go?

Evaluate Model

Once the model is fit, we can evaluate the skill of its predictions on the holdout test dataset.

We will evaluate a model by generating descriptions for all photos in the test dataset and evaluating those predictions with a standard cost function.

First, we need to be able to generate a description for a photo using a trained model.

This involves passing in the start description token '*startseq*', generating one word, then calling the model recursively with generated words as input until the end of sequence token is reached '*endseq*' or the maximum description length is reached.

The function below named *generate_desc()* implements this behavior and generates a textual description given a trained model, and a given prepared photo as input. It calls the function *word_for_id()* in order to map an integer prediction back to a word.



```

1 # map an integer to a word
2 def word_for_id(integer, tokenizer):
3     for word, index in tokenizer.word_index.items():
4         if index == integer:
5             return word
6     return None
7
8 # generate a description for an image
9 def generate_desc(model, tokenizer, photo, max_length):
10     # seed the generation process
11     in_text = 'startseq'

```

```

12     # iterate over the whole length of the sequence
13     for i in range(max_length):
14         # integer encode input sequence
15         sequence = tokenizer.texts_to_sequences([in_text])[0]
16         # pad input
17         sequence = pad_sequences([sequence], maxlen=max_length)
18         # predict next word
19         yhat = model.predict([photo, sequence], verbose=0)
20         # convert probability to integer
21         yhat = argmax(yhat)
22         # map integer to word
23         word = word_for_id(yhat, tokenizer)
24         # stop if we cannot map the word
25         if word is None:
26             break
27         # append as input for generating the next word
28         in_text += ' ' + word
29         # stop if we predict the end of the sequence
30         if word == 'endseq':
31             break
32     return in_text

```

We will generate predictions for all photos in the test dataset and in the train dataset.

The function below named *evaluate_model()* will evaluate a trained model against a given dataset of photo descriptions and photo features. The actual and predicted descriptions are collected and evaluated collectively using the corpus BLEU score that summarizes how close the generated text is to the expected text.



```

1  # evaluate the skill of the model
2  def evaluate_model(model, descriptions, photos, tokenizer, max_length):
3      actual, predicted = list(), list()
4      # step over the whole set
5      for key, desc_list in descriptions.items():
6          # generate description
7          yhat = generate_desc(model, tokenizer, photos[key], max_length)
8          # store actual and predicted
9          references = [d.split() for d in desc_list]
10         actual.append(references)
11         predicted.append(yhat.split())
12     # calculate BLEU score
13     print('BLEU-1: %f' % corpus_bleu(actual, predicted, weights=(1.0, 0, 0, 0)))
14     print('BLEU-2: %f' % corpus_bleu(actual, predicted, weights=(0.5, 0.5, 0, 0)))
15     print('BLEU-3: %f' % corpus_bleu(actual, predicted, weights=(0.3, 0.3, 0.3, 0)))
16     print('BLEU-4: %f' % corpus_bleu(actual, predicted, weights=(0.25, 0.25, 0.25, 0.25)))

```

BLEU scores are used in text translation for evaluating translated text against one or more reference translations.

Here, we compare each generated description against all of the reference descriptions for the photograph. We then calculate BLEU scores for 1, 2, 3 and 4 cumulative n-grams.

You can learn more about the BLEU score here:

- [A Gentle Introduction to Calculating the BLEU Score for Text in Python](#)

The [NLTK Python library implements the BLEU score](#) calculation in the `corpus_bleu()` function. A higher score close to 1.0 is better, a score closer to zero is worse.

We can put all of this together with the functions from the previous section for loading the data. We first need to load the training dataset in order to prepare a Tokenizer so that we can encode generated words as input sequences for the model. It is critical that we encode the generated words using exactly the same encoding scheme as was used when training the model.

We then use these functions for loading the test dataset.

The complete example is listed below.



```
1 from numpy import argmax
2 from pickle import load
3 from keras.preprocessing.text import Tokenizer
4 from keras.preprocessing.sequence import pad_sequences
5 from keras.models import load_model
6 from nltk.translate.bleu_score import corpus_bleu
7
8 # load doc into memory
9 def load_doc(filename):
10     # open the file as read only
11     file = open(filename, 'r')
12     # read all text
13     text = file.read()
14     # close the file
15     file.close()
16     return text
17
18 # load a pre-defined list of photo identifiers
19 def load_set(filename):
20     doc = load_doc(filename)
21     dataset = list()
22     # process line by line
23     for line in doc.split("\n"):
```

```

24         # skip empty lines
25         if len(line) < 1:
26             continue
27         # get the image identifier
28         identifier = line.split('.')[0]
29         dataset.append(identifier)
30     return set(dataset)
31
32 # load clean descriptions into memory
33 def load_clean_descriptions(filename, dataset):
34     # load document
35     doc = load_doc(filename)
36     descriptions = dict()
37     for line in doc.split("\n"):
38         # split line by white space
39         tokens = line.split()
40         # split id from description
41         image_id, image_desc = tokens[0], tokens[1:]
42         # skip images not in the set
43         if image_id in dataset:
44             # create list
45             if image_id not in descriptions:
46                 descriptions[image_id] = list()
47             # wrap description in tokens
48             desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
49             # store
50             descriptions[image_id].append(desc)
51     return descriptions
52
53 # load photo features
54 def load_photo_features(filename, dataset):
55     # load all features
56     all_features = load(open(filename, 'rb'))
57     # filter features
58     features = {k: all_features[k] for k in dataset}
59     return features
60
61 # covert a dictionary of clean descriptions to a list of descriptions
62 def to_lines(descriptions):
63     all_desc = list()
64     for key in descriptions.keys():
65         [all_desc.append(d) for d in descriptions[key]]
66     return all_desc
67
68 # fit a tokenizer given caption descriptions
69 def create_tokenizer(descriptions):
70     lines = to_lines(descriptions)
71     tokenizer = Tokenizer()
72     tokenizer.fit_on_texts(lines)
73     return tokenizer
74
75 # calculate the length of the description with the most words
76 def max_length(descriptions):
77     lines = to_lines(descriptions)
78     return max(len(d.split()) for d in lines)
79
80 # map an integer to a word
81 def word_for_id(integer, tokenizer):
82     for word, index in tokenizer.word_index.items():
83         if index == integer:

```

```

84             return word
85     return None
86
87 # generate a description for an image
88 def generate_desc(model, tokenizer, photo, max_length):
89     # seed the generation process
90     in_text = 'startseq'
91     # iterate over the whole length of the sequence
92     for i in range(max_length):
93         # integer encode input sequence
94         sequence = tokenizer.texts_to_sequences([in_text])[0]
95         # pad input
96         sequence = pad_sequences([sequence], maxlen=max_length)
97         # predict next word
98         yhat = model.predict([photo, sequence], verbose=0)
99         # convert probability to integer
100        yhat = argmax(yhat)
101        # map integer to word
102        word = word_for_id(yhat, tokenizer)
103        # stop if we cannot map the word
104        if word is None:
105            break
106        # append as input for generating the next word
107        in_text += ' ' + word
108        # stop if we predict the end of the sequence
109        if word == 'endseq':
110            break
111    return in_text
112
113 # evaluate the skill of the model
114 def evaluate_model(model, descriptions, photos, tokenizer, max_length):
115     actual, predicted = list(), list()
116     # step over the whole set
117     for key, desc_list in descriptions.items():
118         # generate description
119         yhat = generate_desc(model, tokenizer, photos[key], max_length)
120         # store actual and predicted
121         references = [d.split() for d in desc_list]
122         actual.append(references)
123         predicted.append(yhat.split())
124     # calculate BLEU score
125     print('BLEU-1: %f' % corpus_bleu(actual, predicted, weights=(1.0, 0, 0, 0)))
126     print('BLEU-2: %f' % corpus_bleu(actual, predicted, weights=(0.5, 0.5, 0, 0)))
127     print('BLEU-3: %f' % corpus_bleu(actual, predicted, weights=(0.3, 0.3, 0.3, 0)))
128     print('BLEU-4: %f' % corpus_bleu(actual, predicted, weights=(0.25, 0.25, 0.25, 0.25)))
129
130 # prepare tokenizer on train set
131
132 # load training dataset (6K)
133 filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
134 train = load_set(filename)
135 print('Dataset: %d' % len(train))
136 # descriptions
137 train_descriptions = load_clean_descriptions('descriptions.txt', train)
138 print('Descriptions: train=%d' % len(train_descriptions))
139 # prepare tokenizer
140 tokenizer = create_tokenizer(train_descriptions)
141 vocab_size = len(tokenizer.word_index) + 1
142 print('Vocabulary Size: %d' % vocab_size)
143 # determine the maximum sequence length

```

```

144 max_length = max_length(train_descriptions)
145 print('Description Length: %d' % max_length)
146
147 # prepare test set
148
149 # load test set
150 filename = 'Flickr8k_text/Flickr_8k.testImages.txt'
151 test = load_set(filename)
152 print('Dataset: %d' % len(test))
153 # descriptions
154 test_descriptions = load_clean_descriptions('descriptions.txt', test)
155 print('Descriptions: test=%d' % len(test_descriptions))
156 # photo features
157 test_features = load_photo_features('features.pkl', test)
158 print('Photos: test=%d' % len(test_features))
159
160 # load the model
161 filename = 'model-ep002-loss3.245-val_loss3.612.h5'
162 model = load_model(filename)
163 # evaluate model
164 evaluate_model(model, test_descriptions, test_features, tokenizer, max_length)

```

Running the example prints the BLEU scores.

We can see that the scores fit within and close to the top of the expected range of a skillful model on the problem. The chosen model configuration is by no means optimized.



```

1 BLEU-1: 0.579114
2 BLEU-2: 0.344856
3 BLEU-3: 0.252154
4 BLEU-4: 0.131446

```

Generate New Captions

Now that we know how to develop and evaluate a caption generation model, how can we use it?

Almost everything we need to generate captions for entirely new photographs is in the model file.

We also need the Tokenizer for encoding generated words for the model while generating a sequence, and the maximum length of input sequences, used when we defined the model (e.g. 34).

We can hard code the maximum sequence length. With the encoding of text, we can create the tokenizer and save it to a file so that we can load it quickly whenever we need it without needing the entire Flickr8K dataset. An alternative would be to use our own vocabulary file and mapping to integers function during training.

We can create the Tokenizer as before and save it as a pickle file *tokenizer.pkl*. The complete example is listed below.



```
1 from keras.preprocessing.text import Tokenizer
2 from pickle import dump
3
4 # load doc into memory
5 def load_doc(filename):
6     # open the file as read only
7     file = open(filename, 'r')
8     # read all text
9     text = file.read()
10    # close the file
11    file.close()
12    return text
13
14 # load a pre-defined list of photo identifiers
15 def load_set(filename):
16     doc = load_doc(filename)
17     dataset = list()
18     # process line by line
19     for line in doc.split('\n'):
20         # skip empty lines
21         if len(line) < 1:
22             continue
23         # get the image identifier
24         identifier = line.split('.')[0]
25         dataset.append(identifier)
26     return set(dataset)
27
28 # load clean descriptions into memory
29 def load_clean_descriptions(filename, dataset):
30     # load document
31     doc = load_doc(filename)
32     descriptions = dict()
33     for line in doc.split('\n'):
34         # split line by white space
35         tokens = line.split()
36         # split id from description
37         image_id, image_desc = tokens[0], tokens[1:]
38         # skip images not in the set
39         if image_id in dataset:
40             # create list
41             if image_id not in descriptions:
42                 descriptions[image_id] = list()
```



```

43             # wrap description in tokens
44             desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
45             # store
46             descriptions[image_id].append(desc)
47     return descriptions
48
49 # covert a dictionary of clean descriptions to a list of descriptions
50 def to_lines(descriptions):
51     all_desc = list()
52     for key in descriptions.keys():
53         [all_desc.append(d) for d in descriptions[key]]
54     return all_desc
55
56 # fit a tokenizer given caption descriptions
57 def create_tokenizer(descriptions):
58     lines = to_lines(descriptions)
59     tokenizer = Tokenizer()
60     tokenizer.fit_on_texts(lines)
61     return tokenizer
62
63 # load training dataset (6K)
64 filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
65 train = load_set(filename)
66 print('Dataset: %d' % len(train))
67 # descriptions
68 train_descriptions = load_clean_descriptions('descriptions.txt', train)
69 print('Descriptions: train=%d' % len(train_descriptions))
70 # prepare tokenizer
71 tokenizer = create_tokenizer(train_descriptions)
72 # save the tokenizer
73 dump(tokenizer, open('tokenizer.pkl', 'wb'))

```

We can now load the tokenizer whenever we need it without having to load the entire training dataset of annotations.

Now, let's generate a description for a new photograph.

Below is a new photograph that I chose randomly on Flickr (available under a permissive license).



Photo of a dog at the beach.

Photo by [bambe1964](#), some rights reserved.

We will generate a description for it using our model.

Download the photograph and save it to your local directory with the filename “*example.jpg*”.

First, we must load the Tokenizer from *tokenizer.pkl* and define the maximum length of the sequence to generate, needed for padding inputs.



```
1 # load the tokenizer
2 tokenizer = load(open('tokenizer.pkl', 'rb'))
3 # pre-define the max sequence length (from training)
4 max_length = 34
```

Then we must load the model, as before.



```
1 # load the model
2 model = load_model('model-ep002-loss3.245-val_loss3.612.h5')
```

Next, we must load the photo we which to describe and extract the features.

We could do this by re-defining the model and adding the VGG-16 model to it, or we can use the VGG model to predict the features and use them as inputs to our existing model. We will do the latter and use a modified version of the *extract_features()* function used during data preparation, but adapted to work on a single photo.



```
1 # extract features from each photo in the directory
2 def extract_features(filename):
3     # load the model
4     model = VGG16()
5     # re-structure the model
6     model.layers.pop()
7     model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
8     # load the photo
9     image = load_img(filename, target_size=(224, 224))
10    # convert the image pixels to a numpy array
11    image = img_to_array(image)
12    # reshape data for the model
13    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
14    # prepare the image for the VGG model
15    image = preprocess_input(image)
16    # get features
17    feature = model.predict(image, verbose=0)
18    return feature
19
20 # load and prepare the photograph
21 photo = extract_features('example.jpg')
```

We can then generate a description using the *generate_desc()* function defined when evaluating the model.

The complete example for generating a description for an entirely new standalone photograph is listed below.



```
1 from pickle import load
2 from numpy import argmax
3 from keras.preprocessing.sequence import pad_sequences
4 from keras.applications.vgg16 import VGG16
5 from keras.preprocessing.image import load_img
6 from keras.preprocessing.image import img_to_array
7 from keras.applications.vgg16 import preprocess_input
8 from keras.models import Model
9 from keras.models import load_model
10
11 # extract features from each photo in the directory
12 def extract_features(filename):
13     # load the model
14     model = VGG16()
15     # re-structure the model
16     model.layers.pop()
17     model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
18     # load the photo
19     image = load_img(filename, target_size=(224, 224))
20     # convert the image pixels to a numpy array
21     image = img_to_array(image)
22     # reshape data for the model
23     image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
24     # prepare the image for the VGG model
25     image = preprocess_input(image)
26     # get features
27     feature = model.predict(image, verbose=0)
28     return feature
29
30 # map an integer to a word
31 def word_for_id(integer, tokenizer):
32     for word, index in tokenizer.word_index.items():
33         if index == integer:
34             return word
35     return None
36
37 # generate a description for an image
38 def generate_desc(model, tokenizer, photo, max_length):
39     # seed the generation process
40     in_text = 'startseq'
41     # iterate over the whole length of the sequence
42     for i in range(max_length):
43         # integer encode input sequence
44         sequence = tokenizer.texts_to_sequences([in_text])[0]
45         # pad input
46         sequence = pad_sequences([sequence], maxlen=max_length)
47         # predict next word
48         yhat = model.predict([photo, sequence], verbose=0)
49         # convert probability to integer
50         yhat = argmax(yhat)
51         # map integer to word
52         word = word_for_id(yhat, tokenizer)
53         # stop if we cannot map the word
54         if word is None:
```

```

55             break
56         # append as input for generating the next word
57         in_text += ' ' + word
58         # stop if we predict the end of the sequence
59         if word == 'endseq':
60             break
61     return in_text
62
63 # load the tokenizer
64 tokenizer = load(open('tokenizer.pkl', 'rb'))
65 # pre-define the max sequence length (from training)
66 max_length = 34
67 # load the model
68 model = load_model('model-ep002-loss3.245-val_loss3.612.h5')
69 # load and prepare the photograph
70 photo = extract_features('example.jpg')
71 # generate description
72 description = generate_desc(model, tokenizer, photo, max_length)
73 print(description)

```

In this case, the description generated was as follows:



1 startseq dog is running across the beach endseq

You could remove the start and end tokens and you would have the basis for a nice automatic photo captioning model.

It's like living in the future guys!

It still completely blows my mind that we can do this. Wow.

Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Alternate Pre-Trained Photo Models.** A small 16-layer VGG model was used for feature extraction. Consider exploring larger models that offer better performance on the ImageNet dataset, such as Inception.
- **Smaller Vocabulary.** A larger vocabulary of nearly eight thousand words was used in the development of the model. Many of the words supported may be misspellings or only used once in the entire dataset. Refine the vocabulary and reduce the size, perhaps by half.
- **Pre-trained Word Vectors.** The model learned the word vectors as part of fitting the model. Better performance may be achieved by using word vectors either pre-trained on the training dataset or trained on a much larger corpus of text, such as news articles or Wikipedia.
- **Tune Model.** The configuration of the model was not tuned on the problem. Explore alternate configurations and see if you can achieve better performance.

Did you try any of these extensions? Share your results in the comments below.

Further Reading

This section provides more resources on the topic if you are looking go deeper.

Caption Generation Papers

- [Show and Tell: A Neural Image Caption Generator](#), 2015.
- [Show, Attend and Tell: Neural Image Caption Generation with Visual Attention](#), 2015.
- [Where to put the Image in an Image Caption Generator](#), 2017.
- [What is the Role of Recurrent Neural Networks \(RNNs\) in an Image Caption Generator?](#), 2017.
- [Automatic Description Generation from Images: A Survey of Models, Datasets, and Evaluation Measures](#), 2016.

Flickr8K Dataset

- [Framing image description as a ranking task: data, models and evaluation metrics](#)(Homepage)
- [Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics](#), (PDF) 2013.
- [Dataset Request Form](#)
- [Old Flickr8K Homepage](#)

API

- [Keras Model API](#)
- [Keras pad_sequences\(\) API](#)
- [Keras Tokenizer API](#)
- [Keras VGG16 API](#)
- [Gensim word2vec API](#)
- [nltk.translate package API Documentation](#)

Summary

In this tutorial, you discovered how to develop a photo captioning deep learning model from scratch.

Specifically, you learned:

- How to prepare photo and text data ready for training a deep learning model.
- How to design and train a deep learning caption generation model.
- How to evaluate a train caption generation model and use it to caption entirely new photographs.

Do you have any questions?

Ask your questions in the comments below and I will do my best to answer.

Note: This is an excerpt chapter from: “[Deep Learning for Natural Language Processing](#)”.

Take a look, if you want more step-by-step tutorials on getting the most out of deep learning methods when working with text data.