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**

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.

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.

```
# extract features from each photo in the directory
def extract_features(directory):
	# load the model
	model = VGG16()
	# re-structure the model
	model.layers.pop()
	model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
	# summarize
	print(model.summary())
	# extract features from each photo
	features = dict()
	for name in listdir(directory):
		# load an image from file
		filename = directory + '/' + name
		image = load_img(filename, target_size=(224, 224))
		# convert the image pixels to a numpy array
		image = img_to_array(image)
		# reshape data for the model
		image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
		# prepare the image for the VGG model
		image = preprocess_input(image)
		# get features
		feature = model.predict(image, verbose=0)
		# get image id
		image_id = name.split('.')[0]
		# store feature
		features[image_id] = feature
		print('>%s' % name)
	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.

```
from os import listdir
from pickle import dump
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.models import Model

```

```
# extract features from each photo in the directory
def extract_features(directory):
	# load the model
	model = VGG16()
	# re-structure the model
	model.layers.pop()
	model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
	# summarize
	print(model.summary())
	# extract features from each photo
	features = dict()
	for name in listdir(directory):
		# load an image from file
		filename = directory + '/' + name
		image = load_img(filename, target_size=(224, 224))
		# convert the image pixels to a numpy array
		image = img_to_array(image)
		# reshape data for the model
		image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
		# prepare the image for the VGG model
		image = preprocess_input(image)
		# get features
		feature = model.predict(image, verbose=0)
		# get image id
		image_id = name.split('.')[0]
		# store feature
		features[image_id] = feature
		print('>%s' % name)
	return features

# extract features from all images
directory = 'Flickr8k_Dataset'
features = extract_features(directory)
print('Extracted Features: %d' % len(features))
# save to file
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.

```python
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

filename = 'Flickr8k_text/Flickr8k.token.txt'
# load descriptions
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.

```python
# extract descriptions for images
def load_descriptions(doc):
    mapping = dict()
    # process lines
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        if len(line) < 2:
            continue
        # take the first token as the image id, the rest as the description
        image_id, image_desc = tokens[0], tokens[1:]
        # remove filename from image id
        image_id = image_id.split('.')[0]
        # convert description tokens back to string
        image_desc = ' '.join(image_desc)
        # create the list if needed
        if image_id not in mapping:
            mapping[image_id] = list()
        # store description
        mapping[image_id].append(image_desc)
    return mapping

```

```
# parse descriptions
descriptions = load_descriptions(doc)
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.

```
import string

def clean_descriptions(descriptions):
	# prepare translation table for removing punctuation
	table = str.maketrans('', '', string.punctuation)
	for key, desc_list in descriptions.items():
		for i in range(len(desc_list)):
			desc = desc_list[i]
			# tokenize
			desc = desc.split()
			# convert to lower case
			desc = [word.lower() for word in desc]
			# remove punctuation from each token
			desc = [w.translate(table) for w in desc]
			# remove hanging 's' and 'a'
			desc = [word for word in desc if len(word)>1]
			# remove tokens with numbers in them
			desc = [word for word in desc if word.isalpha()]
			# store as string
			desc_list[i] =  ' '.join(desc)

# clean descriptions
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.

```
# convert the loaded descriptions into a vocabulary of words
def to_vocabulary(descriptions):
	# build a list of all description strings
	all_desc = set()
	for key in descriptions.keys():
		[all_desc.update(d.split()) for d in descriptions[key]]
	return all_desc

# summarize vocabulary
vocabulary = to_vocabulary(descriptions)
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.

```
# save descriptions to file, one per line
def save_descriptions(descriptions, filename):
	lines = list()
	for key, desc_list in descriptions.items():
		for desc in desc_list:
			lines.append(key + ' ' + desc)
	data = '\n'.join(lines)
	file = open(filename, 'w')
	file.write(data)
	file.close()

# save descriptions
save_descriptions(descriptions, 'descriptions.txt')
```

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

```
import string
```

```python

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# extract descriptions for images
def load_descriptions(doc):
    mapping = dict()
    # process lines
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        if len(line) < 2:
            continue
        # take the first token as the image id, the rest as the description
        image_id, image_desc = tokens[0], tokens[1:]
        # remove filename from image id
        image_id = image_id.split('.')[0]
        # convert description tokens back to string
        image_desc = ' '.join(image_desc)
        # create the list if needed
        if image_id not in mapping:
            mapping[image_id] = list()
        # store description
        mapping[image_id].append(image_desc)
    return mapping

def clean_descriptions(descriptions):
    # prepare translation table for removing punctuation
    table = str.maketrans('', '', string.punctuation)
    for key, desc_list in descriptions.items():
        for i in range(len(desc_list)):
            desc = desc_list[i]
            # tokenize
            desc = desc.split()
            # convert to lower case
            desc = [word.lower() for word in desc]
            # remove punctuation from each token
            desc = [w.translate(table) for w in desc]
            # remove hanging 's' and 'a'
            desc = [word for word in desc if len(word)>1]
            # remove tokens with numbers in them
            desc = [word for word in desc if word.isalpha()]
            # store as string
            desc_list[i] =  ' '.join(desc)

# convert the loaded descriptions into a vocabulary of words
def to_vocabulary(descriptions):
    # build a list of all description strings
    all_desc = set()
    for key in descriptions.keys():
        [all_desc.update(d.split()) for d in descriptions[key]]
    return all_desc

```

```
# save descriptions to file, one per line
def save_descriptions(descriptions, filename):
    lines = list()
    for key, desc_list in descriptions.items():
        for desc in desc_list:
            lines.append(key + ' ' + desc)
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()

filename = 'Flickr8k_text/Flickr8k.token.txt'
# load descriptions
doc = load_doc(filename)
# parse descriptions
descriptions = load_descriptions(doc)
print('Loaded: %d ' % len(descriptions))
# clean descriptions
clean_descriptions(descriptions)
# summarize vocabulary
vocabulary = to_vocabulary(descriptions)
print('Vocabulary Size: %d' % len(vocabulary))
# save to file
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).

```
Loaded: 8,092
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.

```
2252123185_487f21e336 bunch on people are seated in stadium
2252123185_487f21e336 crowded stadium is full of people watching an event
2252123185_487f21e336 crowd of people fill up packed stadium
2252123185_487f21e336 crowd sitting in an indoor stadium
2252123185_487f21e336 stadium full of people watch game
...
```

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.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load a pre-defined list of photo identifiers
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    # process line by line
    for line in doc.split('\n'):
        # skip empty lines
        if len(line) < 1:
            continue
        # get the image identifier
        identifier = line.split('.')[0]
        dataset.append(identifier)
```

```
	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.

```
# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
    # load document
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
        image_id, image_desc = tokens[0], tokens[1:]
        # skip images not in the set
        if image_id in dataset:
            # create list
            if image_id not in descriptions:
                descriptions[image_id] = list()
            # wrap description in tokens
            desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
            # store
            descriptions[image_id].append(desc)
    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.

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

We can pause here and test everything developed so far.

The complete code example is listed below.

```
from pickle import load

# load doc into memory
def load_doc(filename):
	# open the file as read only
	file = open(filename, 'r')
	# read all text
	text = file.read()
	# close the file
	file.close()
	return text

# load a pre-defined list of photo identifiers
def load_set(filename):
	doc = load_doc(filename)
	dataset = list()
	# process line by line
	for line in doc.split('\n'):
		# skip empty lines
		if len(line) < 1:
			continue
		# get the image identifier
		identifier = line.split('.')[0]
		dataset.append(identifier)
	return set(dataset)

# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
	# load document
	doc = load_doc(filename)
	descriptions = dict()
	for line in doc.split('\n'):
		# split line by white space
```

```
			tokens = line.split()
			# split id from description
			image_id, image_desc = tokens[0], tokens[1:]
			# skip images not in the set
			if image_id in dataset:
				# create list
				if image_id not in descriptions:
					descriptions[image_id] = list()
				# wrap description in tokens
				desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
				# store
				descriptions[image_id].append(desc)
	return descriptions

# load photo features
def load_photo_features(filename, dataset):
	# load all features
	all_features = load(open(filename, 'rb'))
	# filter features
	features = {k: all_features[k] for k in dataset}
	return features

# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# photo features
train_features = load_photo_features('features.pkl', train)
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.

```
Dataset: 6,000
Descriptions: train=6,000
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.

```
# convert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
	all_desc = list()
	for key in descriptions.keys():
		[all_desc.append(d) for d in descriptions[key]]
	return all_desc

# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
	lines = to_lines(descriptions)
	tokenizer = Tokenizer()
	tokenizer.fit_on_texts(lines)
	return tokenizer

# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
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:

```
X1,		X2 (text sequence), 						y (word)
photo	startseq, 									little
photo	startseq, little,							girl
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.

```
# create sequences of images, input sequences and output words for an image
def create_sequences(tokenizer, max_length, descriptions, photos, vocab_size):
	X1, X2, y = list(), list(), list()
	# walk through each image identifier
	for key, desc_list in descriptions.items():
		# walk through each description for the image
		for desc in desc_list:
			# encode the sequence
			seq = tokenizer.texts_to_sequences([desc])[0]
			# split one sequence into multiple X,y pairs
			for i in range(1, len(seq)):
				# split into input and output pair
				in_seq, out_seq = seq[:i], seq[i]
				# pad input sequence
				in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
				# encode output sequence
				out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
				# store
				X1.append(photos[key][0])
				X2.append(in_seq)
				y.append(out_seq)
	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.

```
# calculate the length of the description with the most words
def max_length(descriptions):
	lines = to_lines(descriptions)
	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

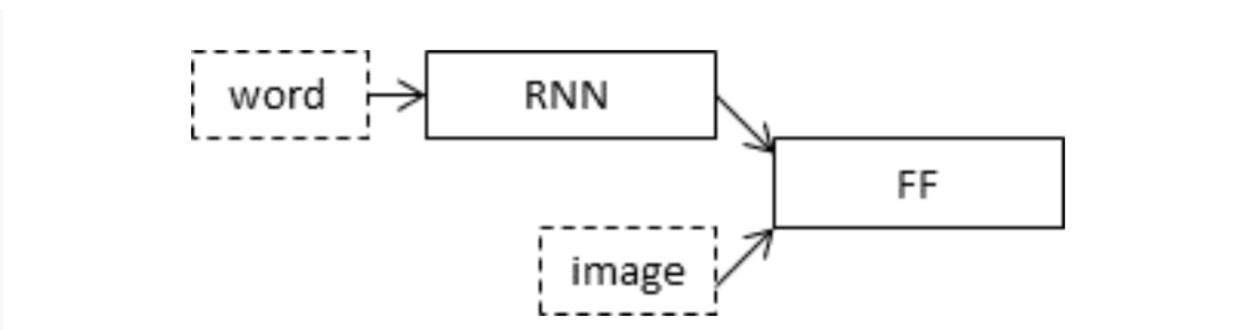
We will define a deep learning based on the "*merge-model*" described by Marc Tanti, et al. in their 2017 papers:

- 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.

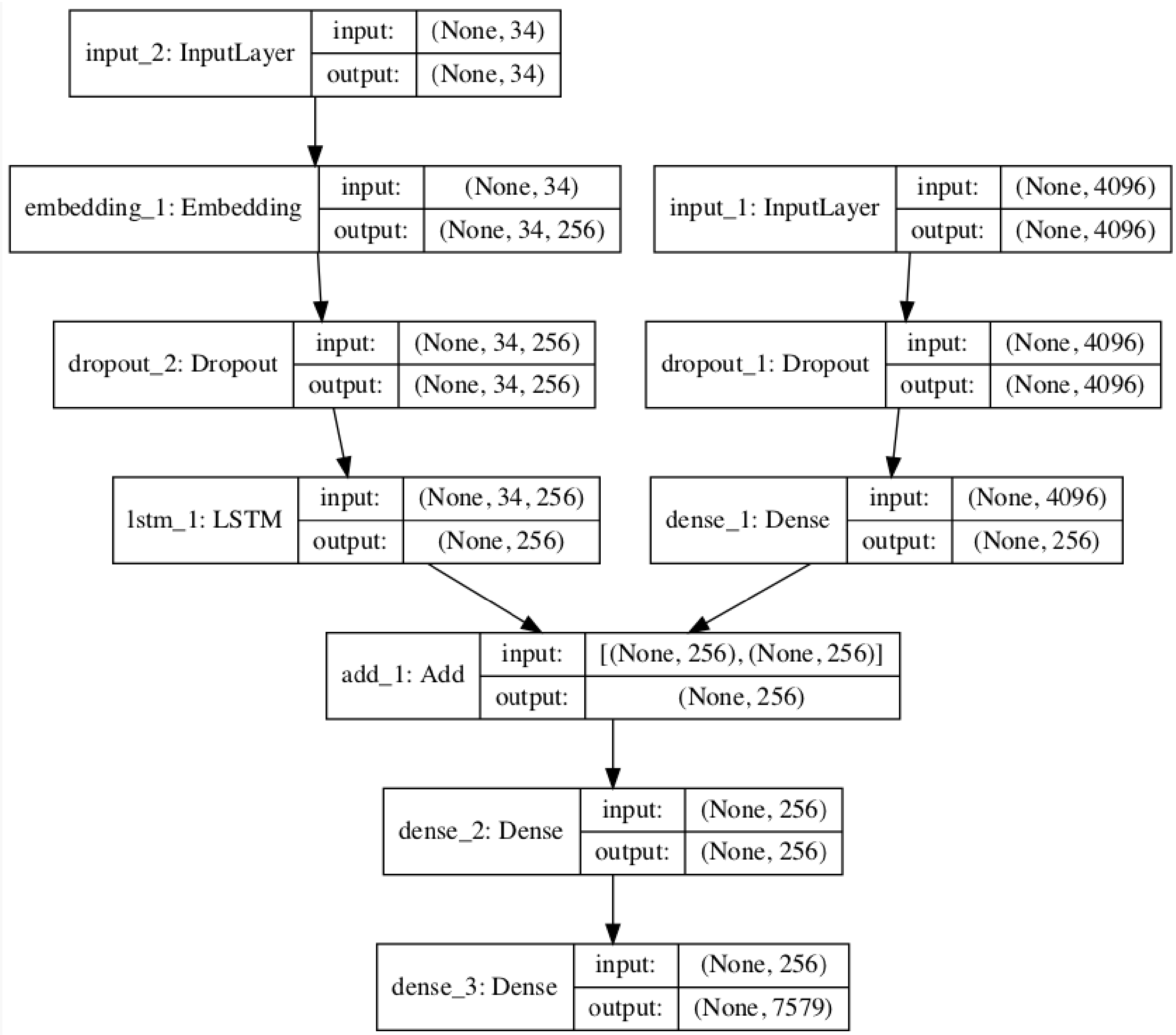
```
# define the captioning model
def define_model(vocab_size, max_length):
	# feature extractor model
	inputs1 = Input(shape=(4096,))
	fe1 = Dropout(0.5)(inputs1)
	fe2 = Dense(256, activation='relu')(fe1)
	# sequence model
	inputs2 = Input(shape=(max_length,))
	se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
	se2 = Dropout(0.5)(se1)
	se3 = LSTM(256)(se2)
	# decoder model
	decoder1 = add([fe2, se3])
	decoder2 = Dense(256, activation='relu')(decoder1)
	outputs = Dense(vocab_size, activation='softmax')(decoder2)
	# tie it together [image, seq] [word]
	model = Model(inputs=[inputs1, inputs2], outputs=outputs)
	model.compile(loss='categorical_crossentropy', optimizer='adam')
	# summarize model
	print(model.summary())
	plot_model(model, to_file='model.png', show_shapes=True)
```

```
return model
```

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

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

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.

```
# define checkpoint callback
filepath = 'model-ep{epoch:03d}-loss{loss:.3f}-val_loss{val_loss:.3f}.h5'
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.

```
# fit model
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.

```
from numpy import array
from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import LSTM
```

```
from keras.layers import Embedding
from keras.layers import Dropout
from keras.layers.merge import add
from keras.callbacks import ModelCheckpoint

# load doc into memory
def load_doc(filename):
	# open the file as read only
	file = open(filename, 'r')
	# read all text
	text = file.read()
	# close the file
	file.close()
	return text

# load a pre-defined list of photo identifiers
def load_set(filename):
	doc = load_doc(filename)
	dataset = list()
	# process line by line
	for line in doc.split('\n'):
		# skip empty lines
		if len(line) < 1:
			continue
		# get the image identifier
		identifier = line.split('.')[0]
		dataset.append(identifier)
	return set(dataset)

# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
	# load document
	doc = load_doc(filename)
	descriptions = dict()
	for line in doc.split('\n'):
		# split line by white space
		tokens = line.split()
		# split id from description
		image_id, image_desc = tokens[0], tokens[1:]
		# skip images not in the set
		if image_id in dataset:
			# create list
			if image_id not in descriptions:
				descriptions[image_id] = list()
			# wrap description in tokens
			desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
			# store
			descriptions[image_id].append(desc)
	return descriptions

# load photo features
def load_photo_features(filename, dataset):
	# load all features
	all_features = load(open(filename, 'rb'))
	# filter features
	features = {k: all_features[k] for k in dataset}
	return features

# covert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
```

```
    all_desc = list()
    for key in descriptions.keys():
        [all_desc.append(d) for d in descriptions[key]]
    return all_desc

# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
    lines = to_lines(descriptions)
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

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

# create sequences of images, input sequences and output words for an image
def create_sequences(tokenizer, max_length, descriptions, photos, vocab_size):
    X1, X2, y = list(), list(), list()
    # walk through each image identifier
    for key, desc_list in descriptions.items():
        # walk through each description for the image
        for desc in desc_list:
            # encode the sequence
            seq = tokenizer.texts_to_sequences([desc])[0]
            # split one sequence into multiple X,y pairs
            for i in range(1, len(seq)):
                # split into input and output pair
                in_seq, out_seq = seq[:i], seq[i]
                # pad input sequence
                in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
                # encode output sequence
                out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
                # store
                X1.append(photos[key][0])
                X2.append(in_seq)
                y.append(out_seq)
    return array(X1), array(X2), array(y)

# define the captioning model
def define_model(vocab_size, max_length):
    # feature extractor model
    inputs1 = Input(shape=(4096,))
    fe1 = Dropout(0.5)(inputs1)
    fe2 = Dense(256, activation='relu')(fe1)
    # sequence model
    inputs2 = Input(shape=(max_length,))
    se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
    se2 = Dropout(0.5)(se1)
    se3 = LSTM(256)(se2)
    # decoder model
    decoder1 = add([fe2, se3])
    decoder2 = Dense(256, activation='relu')(decoder1)
    outputs = Dense(vocab_size, activation='softmax')(decoder2)
    # tie it together [image, seq] [word]
    model = Model(inputs=[inputs1, inputs2], outputs=outputs)
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    # summarize model
    print(model.summary())
```

```
    plot_model(model, to_file='model.png', show_shapes=True)
    return model

# train dataset

# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# photo features
train_features = load_photo_features('features.pkl', train)
print('Photos: train=%d' % len(train_features))
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# determine the maximum sequence length
max_length = max_length(train_descriptions)
print('Description Length: %d' % max_length)
# prepare sequences
X1train, X2train, ytrain = create_sequences(tokenizer, max_length, train_descriptions, train_features, vocab_size)

# dev dataset

# load test set
filename = 'Flickr8k_text/Flickr_8k.devImages.txt'
test = load_set(filename)
print('Dataset: %d' % len(test))
# descriptions
test_descriptions = load_clean_descriptions('descriptions.txt', test)
print('Descriptions: test=%d' % len(test_descriptions))
# photo features
test_features = load_photo_features('features.pkl', test)
print('Photos: test=%d' % len(test_features))
# prepare sequences
X1test, X2test, ytest = create_sequences(tokenizer, max_length, test_descriptions, test_features, vocab_size)

# fit model

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

```
Dataset: 6,000
Descriptions: train=6,000
Photos: train=6,000
Vocabulary Size: 7,579
Description Length: 34
Dataset: 1,000
Descriptions: test=1,000
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.

```
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:

```
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.

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

```
# create sequences of images, input sequences and output words for an image
def create_sequences(tokenizer, max_length, desc_list, photo, vocab_size):
	X1, X2, y = list(), list(), list()
	# walk through each description for the image
	for desc in desc_list:
		# encode the sequence
		seq = tokenizer.texts_to_sequences([desc])[0]
		# split one sequence into multiple X,y pairs
		for i in range(1, len(seq)):
			# split into input and output pair
			in_seq, out_seq = seq[:i], seq[i]
			# pad input sequence
			in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
			# encode output sequence
			out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
			# store
			X1.append(photo)
			X2.append(in_seq)
			y.append(out_seq)
	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:

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

```
(47, 4096)
(47, 34)
(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:

```
# train the model, run epochs manually and save after each epoch
epochs = 20
steps = len(train_descriptions)
for i in range(epochs):
	# create the data generator
	generator = data_generator(train_descriptions, train_features, tokenizer, max_length, vocab_size)
	# fit for one epoch
	model.fit_generator(generator, epochs=1, steps_per_epoch=steps, verbose=1)
	# save model
	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.

```
from numpy import array
from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding
from keras.layers import Dropout
from keras.layers.merge import add
from keras.callbacks import ModelCheckpoint

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load a pre-defined list of photo identifiers
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    # process line by line
    for line in doc.split('\n'):
        # skip empty lines
        if len(line) < 1:
            continue
        # get the image identifier
        identifier = line.split('.')[0]
        dataset.append(identifier)
    return set(dataset)

# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
    # load document
    doc = load_doc(filename)
    descriptions = dict()
```

```
	for line in doc.split('\n'):
		# split line by white space
		tokens = line.split()
		# split id from description
		image_id, image_desc = tokens[0], tokens[1:]
		# skip images not in the set
		if image_id in dataset:
			# create list
			if image_id not in descriptions:
				descriptions[image_id] = list()
			# wrap description in tokens
			desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
			# store
			descriptions[image_id].append(desc)
	return descriptions

# load photo features
def load_photo_features(filename, dataset):
	# load all features
	all_features = load(open(filename, 'rb'))
	# filter features
	features = {k: all_features[k] for k in dataset}
	return features

# covert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
	all_desc = list()
	for key in descriptions.keys():
		[all_desc.append(d) for d in descriptions[key]]
	return all_desc

# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
	lines = to_lines(descriptions)
	tokenizer = Tokenizer()
	tokenizer.fit_on_texts(lines)
	return tokenizer

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

# create sequences of images, input sequences and output words for an image
def create_sequences(tokenizer, max_length, desc_list, photo, vocab_size):
	X1, X2, y = list(), list(), list()
	# walk through each description for the image
	for desc in desc_list:
		# encode the sequence
		seq = tokenizer.texts_to_sequences([desc])[0]
		# split one sequence into multiple X,y pairs
		for i in range(1, len(seq)):
			# split into input and output pair
			in_seq, out_seq = seq[:i], seq[i]
			# pad input sequence
			in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
			# encode output sequence
			out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
			# store
			X1.append(photo)
```

```
				X2.append(in_seq)
				y.append(out_seq)
	return array(X1), array(X2), array(y)

# define the captioning model
def define_model(vocab_size, max_length):
	# feature extractor model
	inputs1 = Input(shape=(4096,))
	fe1 = Dropout(0.5)(inputs1)
	fe2 = Dense(256, activation='relu')(fe1)
	# sequence model
	inputs2 = Input(shape=(max_length,))
	se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
	se2 = Dropout(0.5)(se1)
	se3 = LSTM(256)(se2)
	# decoder model
	decoder1 = add([fe2, se3])
	decoder2 = Dense(256, activation='relu')(decoder1)
	outputs = Dense(vocab_size, activation='softmax')(decoder2)
	# tie it together [image, seq] [word]
	model = Model(inputs=[inputs1, inputs2], outputs=outputs)
	# compile model
	model.compile(loss='categorical_crossentropy', optimizer='adam')
	# summarize model
	model.summary()
	plot_model(model, to_file='model.png', show_shapes=True)
	return model

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

# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# photo features
train_features = load_photo_features('features.pkl', train)
print('Photos: train=%d' % len(train_features))
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# determine the maximum sequence length
max_length = max_length(train_descriptions)
print('Description Length: %d' % max_length)

# define the model
model = define_model(vocab_size, max_length)
# train the model, run epochs manually and save after each epoch
```

```
epochs = 20
steps = len(train_descriptions)
for i in range(epochs):
	# create the data generator
	generator = data_generator(train_descriptions, train_features, tokenizer, max_length, vocab_size)
	# fit for one epoch
	model.fit_generator(generator, epochs=1, steps_per_epoch=steps, verbose=1)
	# 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.

```
# map an integer to a word
def word_for_id(integer, tokenizer):
	for word, index in tokenizer.word_index.items():
		if index == integer:
			return word
	return None

# generate a description for an image
def generate_desc(model, tokenizer, photo, max_length):
	# seed the generation process
	in_text = 'startseq'
```

```
    # iterate over the whole length of the sequence
    for i in range(max_length):
        # integer encode input sequence
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        # pad input
        sequence = pad_sequences([sequence], maxlen=max_length)
        # predict next word
        yhat = model.predict([photo,sequence], verbose=0)
        # convert probability to integer
        yhat = argmax(yhat)
        # map integer to word
        word = word_for_id(yhat, tokenizer)
        # stop if we cannot map the word
        if word is None:
            break
        # append as input for generating the next word
        in_text += ' ' + word
        # stop if we predict the end of the sequence
        if word == 'endseq':
            break
    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.

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

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.

```
from numpy import argmax
from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model
from nltk.translate.bleu_score import corpus_bleu

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load a pre-defined list of photo identifiers
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    # process line by line
    for line in doc.split('\n'):
```

```
		# skip empty lines
		if len(line) < 1:
			continue
		# get the image identifier
		identifier = line.split('.')[0]
		dataset.append(identifier)
	return set(dataset)

# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
	# load document
	doc = load_doc(filename)
	descriptions = dict()
	for line in doc.split('\n'):
		# split line by white space
		tokens = line.split()
		# split id from description
		image_id, image_desc = tokens[0], tokens[1:]
		# skip images not in the set
		if image_id in dataset:
			# create list
			if image_id not in descriptions:
				descriptions[image_id] = list()
			# wrap description in tokens
			desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
			# store
			descriptions[image_id].append(desc)
	return descriptions

# load photo features
def load_photo_features(filename, dataset):
	# load all features
	all_features = load(open(filename, 'rb'))
	# filter features
	features = {k: all_features[k] for k in dataset}
	return features

# covert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
	all_desc = list()
	for key in descriptions.keys():
		[all_desc.append(d) for d in descriptions[key]]
	return all_desc

# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
	lines = to_lines(descriptions)
	tokenizer = Tokenizer()
	tokenizer.fit_on_texts(lines)
	return tokenizer

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

# map an integer to a word
def word_for_id(integer, tokenizer):
	for word, index in tokenizer.word_index.items():
		if index == integer:
```

```
			return word
	return None

# generate a description for an image
def generate_desc(model, tokenizer, photo, max_length):
	# seed the generation process
	in_text = 'startseq'
	# iterate over the whole length of the sequence
	for i in range(max_length):
		# integer encode input sequence
		sequence = tokenizer.texts_to_sequences([in_text])[0]
		# pad input
		sequence = pad_sequences([sequence], maxlen=max_length)
		# predict next word
		yhat = model.predict([photo,sequence], verbose=0)
		# convert probability to integer
		yhat = argmax(yhat)
		# map integer to word
		word = word_for_id(yhat, tokenizer)
		# stop if we cannot map the word
		if word is None:
			break
		# append as input for generating the next word
		in_text += ' ' + word
		# stop if we predict the end of the sequence
		if word == 'endseq':
			break
	return in_text

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

# prepare tokenizer on train set

# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# determine the maximum sequence length
```

```
max_length = max_length(train_descriptions)
print('Description Length: %d' % max_length)

# prepare test set

# load test set
filename = 'Flickr8k_text/Flickr_8k.testImages.txt'
test = load_set(filename)
print('Dataset: %d' % len(test))
# descriptions
test_descriptions = load_clean_descriptions('descriptions.txt', test)
print('Descriptions: test=%d' % len(test_descriptions))
# photo features
test_features = load_photo_features('features.pkl', test)
print('Photos: test=%d' % len(test_features))

# load the model
filename = 'model-ep002-loss3.245-val_loss3.612.h5'
model = load_model(filename)
# evaluate model
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.

```
BLEU-1: 0.579114
BLEU-2: 0.344856
BLEU-3: 0.252154
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.

```
from keras.preprocessing.text import Tokenizer
from pickle import dump

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load a pre-defined list of photo identifiers
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    # process line by line
    for line in doc.split('\n'):
        # skip empty lines
        if len(line) < 1:
            continue
        # get the image identifier
        identifier = line.split('.')[0]
        dataset.append(identifier)
    return set(dataset)

# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
    # load document
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
        image_id, image_desc = tokens[0], tokens[1:]
        # skip images not in the set
        if image_id in dataset:
            # create list
            if image_id not in descriptions:
                descriptions[image_id] = list()
```

```
			# wrap description in tokens
			desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
			# store
			descriptions[image_id].append(desc)
	return descriptions

# covert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
	all_desc = list()
	for key in descriptions.keys():
		[all_desc.append(d) for d in descriptions[key]]
	return all_desc

# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
	lines = to_lines(descriptions)
	tokenizer = Tokenizer()
	tokenizer.fit_on_texts(lines)
	return tokenizer

# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
# save the tokenizer
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.

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

Then we must load the model, as before.

```
# load the model
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.

```
# extract features from each photo in the directory
def extract_features(filename):
    # load the model
    model = VGG16()
    # re-structure the model
    model.layers.pop()
    model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
    # load the photo
    image = load_img(filename, target_size=(224, 224))
    # convert the image pixels to a numpy array
    image = img_to_array(image)
    # reshape data for the model
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
    # prepare the image for the VGG model
    image = preprocess_input(image)
    # get features
    feature = model.predict(image, verbose=0)
    return feature

# load and prepare the photograph
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.

```
from pickle import load
from numpy import argmax
from keras.preprocessing.sequence import pad_sequences
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.models import Model
from keras.models import load_model

# extract features from each photo in the directory
def extract_features(filename):
    # load the model
    model = VGG16()
    # re-structure the model
    model.layers.pop()
    model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
    # load the photo
    image = load_img(filename, target_size=(224, 224))
    # convert the image pixels to a numpy array
    image = img_to_array(image)
    # reshape data for the model
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
    # prepare the image for the VGG model
    image = preprocess_input(image)
    # get features
    feature = model.predict(image, verbose=0)
    return feature

# map an integer to a word
def word_for_id(integer, tokenizer):
    for word, index in tokenizer.word_index.items():
        if index == integer:
            return word
    return None

# generate a description for an image
def generate_desc(model, tokenizer, photo, max_length):
    # seed the generation process
    in_text = 'startseq'
    # iterate over the whole length of the sequence
    for i in range(max_length):
        # integer encode input sequence
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        # pad input
        sequence = pad_sequences([sequence], maxlen=max_length)
        # predict next word
        yhat = model.predict([photo,sequence], verbose=0)
        # convert probability to integer
        yhat = argmax(yhat)
        # map integer to word
        word = word_for_id(yhat, tokenizer)
        # stop if we cannot map the word
        if word is None:
```

```
                break
        # append as input for generating the next word
        in_text += ' ' + word
        # stop if we predict the end of the sequence
        if word == 'endseq':
            break
    return in_text

# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
# pre-define the max sequence length (from training)
max_length = 34
# load the model
model = load_model('model-ep002-loss3.245-val_loss3.612.h5')
# load and prepare the photograph
photo = extract_features('example.jpg')
# generate description
description = generate_desc(model, tokenizer, photo, max_length)
print(description)
```

In this case, the description generated was as follows:

```
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 Flicrk8K 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.